UNIVERSITY OF CALIFORNIA
SANTA CRUZ

MISC:
A Massively Parallel Architecture for Associative-Based Artificial Intelligence

A dissertation submitted in partial satisfaction
of the requirements for the degree of
DOCTOR OF PHILOSOPHY
in
COMPUTER ENGINEERING
by
James D. Roberts
June 1995

The dissertation of James D. Roberts is approved:

Richard Hughey

Robert Levinson

Charles McDowell

Dean of Graduate Studies and Research

Contents

List of Figures

List of Tables

MISC:
A Massively Parallel Architecture
for Associative-Based Artificial Intelligence

James D. Roberts

ABSTRACT

Parallel architectures are required to provide the computational power for future artificial intelligence research and applications. Massively parallel processing is emerging as an important alternative to prior approaches to parallel AI. Unfortunately, contemporary general-purpose machines are optimized for numeric codes while application-specific machines do not fully address the wide variety of algorithms needed by large-scale knowledge processing. This dissertation investigates overcoming these limitations through associative content-based processing of complex dynamic data objects, focusing on retrieval from large knowledge bases.

Analysis of algorithm-architecture-technology interactions is the foundation of the MISC architecture development. To accomplish this, we develop an associative pseudocode, new parallel algorithms, and an enhanced design tool. The SEGUL pseudocode features nested multisets, allowing the expression of knowledge processing algorithms in a machine-independent manner for subsequent characterization. Three algorithms serve as the focus of the analysis: an associative memory neural network, a semantic network, and a graph knowledge-base. Suspense, the early analysis tool, projects speed and cost measures for architectural alternatives. Functional-level simulation combined with early analysis of a hypothetical implementation provide cost-performance projections for evaluating the resulting architecture.

The MISC system architecture is comprised of multiple small data-parallel arrays, as the the number of processors required by any one of the algorithms scales sublinearly with respect to the knowledge base size. The array processors feature a general-purpose core and additional functional units to improve content-based knowledge processing. Each processor also has high bandwidth access to a large off-chip memory, and inter-processor communication is by a wide-link extended mesh network. Analysis shows that advanced packaging is a cost-effective means of achieving the required high connectivity. Simulation of the MISC architecture for the suite of three algorithms indicates more than a factor of ten improvement in both speed and cost-performance over a massively parallel architecture tailored for numeric codes.

Acknowledgments

This dissertation, like most others, owes a great deal to many people. My greatest appreciation goes to my committee members for maintaining a friendly, supportive research atmosphere. I thank Richard Hughey for his enthusiasm for the MISC research, guidance on parallel architecture and algorithms, and valiant efforts to break me of old writing habits. I am also grateful for his efforts in obtaining research funding from the Natural Sciences Division of UCSC for the crucial initial phase of the research. I thank Robert Levinson, in many ways a co-supervisor, for his patient tutoring in AI algorithms, enthusiasm for my parallelizations, guidance in developing papers and the dissertation, and research assistantship funding. I thank Charlie McDowell for his course in parallel processing, his pragmatic approach to computer science and engineering, and for a challenging evaluation of the MISC architecture.

Appreciation also goes to the professors of my classes and teaching assistantships for providing me with an excellent education as well as for their enthusiasm and continued encouragement. I especially thank Dan Helman for his supervision of my Master's Thesis and encouragement to pursue both parallel VLSI systems and knowledge processing and Wayne Dai for his introduction to early analysis tools and packaging technology as well as for his financial support. I thank all of the Board staff members for their friendly assistance and for their efforts in mitigating the effects of the UC budget crisis.

I also thank Pentti Kanerva and Claude Nosphitz for stimulating discussions over the internet and while walking through the redwoods. Special recognition also goes to Amit Gulati for his assistance in implementing the PCAM chip.

1. Introduction

Artificial intelligence systems are already reducing errors and cost and providing faster and better solutions in areas as diverse as information retrieval, design and manufacturing, finance and management, entertainment, and medicine. Although many applications are running on single-processor workstations and even personal computers, the long-range goals for intelligent systems will require tremendous computational power. The capability of current computers has greatly limited research and applications in the fields of artificial intelligence (AI) and cognitive modeling. In addition to new algorithms, parallelism is needed to provide reasonable response times. The characteristics of AI codes, however, make parallelism difficult.

This dissertation extends highly parallel computing to more effectively meet the demands of AI. The proposed MISC architecture results from analysis of example algorithms and the general characteristics of AI programs, incorporating new architectural features to extend the applicability of massively parallel processing. The combined results have the potential to expand current uses and enable new research and applications in knowledge-based systems, possibly placing computers in new and broader roles.

The Problem

Why is highly parallel processing needed for AI and why hasn't it already been fully achieved? Although microprocessor speeds increase by some 40% each year and rival supercomputers' performance of not long ago, combining multiple processors can provide magnitudes-higher performance. AI is not well served by conventional uniprocessor and parallel processors due to its unique characteristics, yet prior efforts towards special-purpose AI architectures have met with limited success. Special-purpose uniprocessor systems such, as LISP workstations, were quickly outstripped by general-purpose machines. Special-purpose parallel Prolog engines provided less than the hoped for speedup. A key difficulty in obtaining sufficient parallel speedup for AI has been limited high-level (program control) parallelism and thus many researchers are turning to low-level (data structure) massive parallelism. Unlike special-purpose AI processors, massive parallelism can more than make up for the additional system development time and multiprocessing inefficiencies through the use of less costly technologies. Contemporary massively parallel machines are, however, optimized for numeric rather than symbolic operations, limiting performance. Other efforts include application-specific hardware. These projects address a specific methodology or task rather than the wide variety of techniques required by large-scale intelligent systems. A general-purpose cost-effective massively parallel machine tailored to knowledge processing is needed.

Nearly all parallel machines with a thousand or more processors use the Single-Instruction stream Multiple-Data stream (SIMD) model in which all processors receive the same instruction and operate on their own but identically structured data. This is in contrast to Multiple-Instruction stream Multiple-Data stream(MIMD) model used in many parallel machines, in which each processor is a full computer able to execute its own program independently of the others. SIMD architectures offer significant cost reduction and performance benefits, and may be the only approach scalable to tens of thousands or even millions of processors. SIMD processing does, however, enforce a great deal of regularity in data and control structures. Characteristics of typical AI codes, such as the complex, dynamic control and data structures, do not mesh well with the SIMD architecture, and incur significant efficiency penalties.

The MISC Solution

The proposed MISC architecture[1] extends SIMD processing to better support AI. In contrast to Prolog and LISP engines, MISC is based on massively parallel algorithms rather than programming language requirements and relies on the parallelization of *associative* operations in which data is treated as collections, accessed and manipulated directly by content rather than by address. MISC is unique in its emphasis on pruned search for complex objects in extremely large knowledge bases and the resulting architectural implications. Prior work on massively parallel AI processing has for the most part emphasized simple matching, exhaustive search for retrieval, and neural networks.

As shown in Figure 1.1, the full MISC architecture consists of multiple SIMD arrays (MSIMD), each with a scalar supervisor microprocessor, and a scalable communication network. The hierarchical architecture combines two levels of processing, the custom SIMD arrays and the standard microprocessors in a MIMD network, reaping the benefits of both powerful microprocessors and custom tailoring for AI. In large MISC implementations the communication network may itself be a multi-level hierarchy and incorporate additional levels of processing. This dissertation emphasizes the heart of the MISC architecture: the SIMD associative array. The array architecture returns several features of associative processing to a contemporary massively parallel "core" and makes significant additions to the processor microarchitecture, array memory subsystem, and array communication network to improve speed and cost-performance for knowledge retrieval codes while retaining general-purpose capabilities.

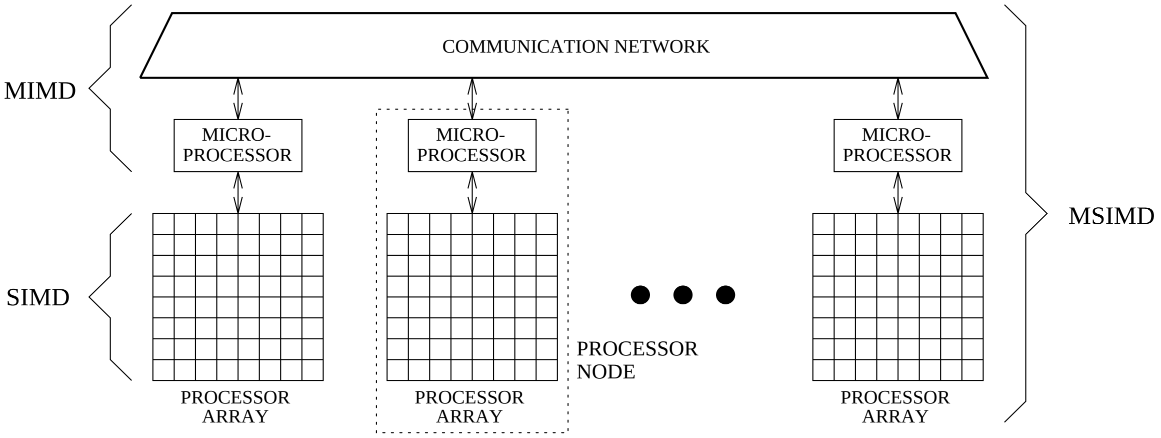


Figure 1.1: A High-Level View of the MISC Architecture

---

[1] Misc Is Symbolic Computing or Machine Intelligence Santa Cruz.

In addition to the MISC architecture itself, dissertation contributions include an associative processing model for pseudocoding AI algorithms, including multiple levels of parallelism and supporting more fundamental analysis of particular algorithms. An early analysis tool projects speed and cost-metrics given an architectural-level description for various technologies, supporting analysis of potential implementations.

1.1 Context

This section introduces the conceptual framework of the MISC research. We first discuss many of the general issues of knowledge processing and the importance of explicitly supporting complex structured data, especially their retrieval from large knowledge databases. We next discuss tailoring parallel processors for machine intelligence, including general requirements and the promise of extending SIMD architectures and programming models.

1.1.1 Knowledge Processing

Knowledge processing is perhaps the most appropriate term to refer to existing methods incorporated into knowledge-based systems which might lead to machine intelligence. The development of digital computers was initially driven by the demands of numerical computations, such as numeric solutions to differential equations. Computer pioneers, most notably Allan Turing, quickly realized the machines' ability to manipulate symbols as well. The potential for such manipulations to achieve results that would be said to require intelligence if performed by a human forms both a definition of AI and its most central tenet, the physical symbol hypothesis [Kur90]. There are numerous other definitions, and to most AI is a yet to be achieved *goal* so that other terms may be more appropriate for existing systems. Other authors often use the term *symbolic processing* synonymously with knowledge processing and AI; this dissertation generally uses knowledge processing in distinction from symbolic math and to emphasize semantic as opposed to syntactic manipulations (although the former are ultimately built upon syntactic manipulations of Boolean values in a digital computer).

Search

Retrieval and search are among the most crucial knowledge processing operations. Backtracking search is a central AI paradigm in which possible solutions are methodically generated and examined until an optimal or acceptable one is found; heuristics and algorithmic techniques to reduce the number of solutions examined give rise to a wide variety of search algorithms. The trade-off between stored knowledge and search is a central AI paradigm. Early AI researchers believed that intelligent behavior could be achieved by search and other formal methods with a minimum of domain knowledge; it is now generally accepted that search and inference must be accompanied by large amounts of stored knowledge. Stored information, including prior solutions and partial solutions, can be a powerful tool in reducing the amount of searching. There is also a trade-off between search and the complexity of knowledge representation [Lev94a]. Effective knowledge representation is thereby highly important in knowledge processing. In addition to including all non-redundant information, a knowledge representation must support fast access and updates and efficient processing as well as compact storage. Rapidly retrieving knowledge objects from extremely large knowledge bases is an increasingly crucial (and demanding) operation. Knowledge processing systems must support inference (eliciting implicit information) and pattern matching. Neural networks and other connectionist methodologies have recently challenged the physical symbol hypothesis and deserve processing support, particularly in areas such as pattern matching. This dissertation addresses both connectionist and symbolic approaches, and emphasizes *associative* retrieval of complex objects from large knowledge bases.

Association

It is important to distinguish between several closely related terms.

Associative processing refers to a computational and programming model in which data objects are accessed and manipulated by content rather than by coordinate addresses or pointers. Parallel operations can be implicitly applied to collections of data objects as though they were a single object.

Content-addressable memory refers to a specific type of hardware to support data-object selection based on matching.

Associative memory refers to various models of human memory emphasizing the recall of one memory triggered by another.

Here we introduce the basic features of associative memory as they relate to the goals of this dissertation. Chapter 2 describes each of the above terms in greater detail.

Aristotle first formulated the classical laws of association in *On Memory and Reminiscence*. Concepts, memories, and sensations are said to be connected if they occur simultaneously (spatial association), if they occur in close succession (temporal association), if they are similar, or if they are contrary [Koh80]. Such work continues into the present, but associative recall and mappings alone are not enough. Although they can be a critical component of a knowledge processing system, additional mechanisms are necessary. As Touretzky notes, associative memories require training data covering most of the input space and may not scale to efficiently construct rule-following behavior [Tou88]. In particular he comments that the compositional structure of language cannot be reduced to input-output mappings unless the training examples are nearly exhaustive. Associative memory models' ability to express relationships between objects and to efficiently support variables and binding between variables is an open question.

Relations

It is thereby crucial that any associative environment for knowledge processing be extended to include *relational data objects* containing both primitive objects and explicit relations between them. The shield in the medieval coat of arms shown in Figure 1.2 depicts a relational object. The independent symbols in the list (father son holy_ghost god is is is isn't isn't isn't) don't capture the meaning ("God is Father, Son and Holy ghost; but the Father, Son, and Holy Ghost aren't the same.") Enforcing a particular order (god is father isn't son ...) still does not completely capture the concept conveyed by the figure, and the meaning of the order must be denoted somewhere. The same issues are involved in describing visual scenes and physical objects. The list (large_flat_surface leg leg leg leg) is insufficient to describe the image of a table to a robot from Tau-Ceti; the relationships between the spatial primitives must be denoted as well. By explicitly representing relations, this information is directly available and generalization and other processing can take place at a higher level. Baclawski notes that relational structures provide semantically richer means of expressing information-retrieval queries, semantic proximity being very difficult even to approximate using attribute lists, but that they have been little used due to the high computational requirements [BS93]

Relational structures are readily visualized as graphs. Indeed, a graph is formally a set of binary relations, and a hypergraph is a set of n-ary relations. A tremendous range of problems can be expressed as relational structures and thereby solved using graph processing algorithms. Matching relational objects, particularly determining if one structure is a substructure of another (subgraph isomorphism) can be especially important in knowledge retrieval and reasoning by analogy. Subgraph isomorphism is particularly challenging, requiring variable binding, and in the worst case exponential processing time.

Figure 1.2: A Medieval Relational Knowledge Representation (from [Leh92])

Sets

Sets and hierarchical data objects are important as well. Figure 1.3 depicts an antiquated cognitive model which illustrates a relational object. Note that the figure also uses set diagrams to show hierarchies of containment and overlap denoting inheritance of commonalities, important in knowledge representation.

Figure 1.3: An Early Model of the Mind (from [Bow92])

Graph and Set Based Inference and Retrieval

In the late 19^{th} century, Charles Peirce developed a somewhat similar notation, existential graphs, for graphically notating symbolic logic with an expressiveness equivalent to first-order logic [EL92]. This work has recently gained renewed interest ([Sow84]) as an alternative to Russell and Whitehead's linear notation.

In a survey book, Kurzweil describes knowledge as facts and their relationships and discusses the importance of *inheritance*, the characteristics of a set applying to its subsets, in knowledge processing [Kur90]. Knowledge processing systems often feature an ontology (a set of all objects and relationships in the knowledge system), elucidating inheritance and other relationships between concepts and categories. The primary portion of the ontology, the schema, is itself a relational structure: a directed graph whose vertices are concepts and categories and whose edges are relationships such as is_a and part_of. These graphs often take the form of a tree; where a node can have more than one "parent" (for example multiple inheritance of a "a beagle is_a dog and is_a pet) the graph a Hasse diagram denoting a partial order. These hierarchical relational structures are important in dramatically improving the retrieval of complex objects, serving as a *multi-level index* on the objects.

1.1.2 Knowledge Processors

Although massively parallel architectures for numeric applications have been extensively researched, there has been much less work on massive parallelism for symbolic applications. Wah, Lowrie, and Li discuss the growing need for increased computational power and new symbolic architectures.

> Recent advances in applications of computers suggest that the processing of symbols rather than numbers will be the basis for the next generation of computers.... Symbolic processing has been applied in a wide spectrum of areas; among them are pattern recognition, natural language processing, speech understanding, theorem proving, robotics, computer vision, and expert systems. Researchers in artificial intelligence, databases, programming languages, cognitive science, psychology, and many other fields have addressed overlapping issues in symbolic processing. Conventional computers have been designed with numeric processing power as their focus. The disparity between symbolic and numeric operations, therefore, calls for innovative research in alternative architectures for symbolic processing. [WLL90]

Although early efforts, such as the Japanese 5^{th} Generation and related U.S. and European projects gave less than the expected results, perhaps due to emphasis on logic programming and language-oriented processors, the progress and potential indicate the merits of continued research. In addition to special-purpose machines for vision and image processing, contemporary work includes machines designed specifically for databases and knowledge base processing [KT88, S^+85, M^+92, H^+91]. Unfortunately these do not address the full needs of knowledge processing. Special-purpose machines lack the flexibility to perform well on differing algorithms and problem domains. Knowledge base machines, typically optimized for search, are largely limited to supporting logic programming and production systems [KT88]. Relational database processors store n-ary relations independently, requiring time consuming operations to access and manipulate particular sets of relations. Syntactic manipulation of regularly structured symbolic data, such as databases, has had a tremendous affect on business and research; the potential for knowledge processing in the sense of semantic manipulation of more complex information structures, could be even greater.

Knowledge Processing Characteristics

Some characteristics of knowledge processing algorithms place special demands on parallel processors [HGC87, WL89, WLL90] while others suggest promising avenues for research. Most knowledge processing programs have limited coarse-grain (control) parallelism; efforts to parallelize existing codes generally show poor processor utilization beyond 16 processors and negligible speedup beyond 32 [KT88, Kog91, WLL90]. On the other hand, there are numerous effective codes developed specifically for massively parallel machines including neural networks and natural language processing [AAA93]. Fine-grain (massive) parallelism is highly encouraging but there has been limited work to date. Knowledge processing has several characteristics that make efficient parallelization difficult on current general-purpose machines. Instead of the statically-sized arrays of numeric operations, knowledge processing programs emphasize dynamic, variable length, and linked data structures. Memory referencing is in less regular patterns than in numeric processing, control structures are more often complex and data driven, and programs are more subroutine and recursion intensive. Furthermore, parallel knowledge processing codes are often communication intensive with bursty, nondeterministic communication patterns. These characteristics can result in low performance on serial architectures and small speedups on parallel systems. On the other hand, extensive use of matching operations and knowledge base search can be used to advantage. Knowledge processing often emphasizes pattern matching and classification (associative retrieval) as the single mechanism for accessing components of compound data structures, selecting procedures, and passing parameters [BF82]. Moldovan considers matching to be one of the "inner loops" of reasoning, and comments that it is trivial for single attributes but difficult for structured relations [M^+92]. Similarly, Fahlman characterizes knowledge processing programs as matching operations on a large databases followed by minimal processing [Fah85], while Kogge emphasizes matching, substitution, sets and relations, and Levinson argues for inference and intelligent behavior based on pattern associativity [Kog91, Lev92b].

Associative Processing, SIMD Architecture, and AI

Researchers and designers have long sought to exploit extensive low-level matching for improved performance by developing associative processors based on content-addressable memory (CAM). Indeed, the beginnings of this effort were contemporaneous with the development of programmable digital computers. Perhaps the first is the relay-based CAM designed by Conrad Zuse in 1943, shown in Figure 1.4. These designs have been plagued by high cost and the difficult competition with the ubiquitous von Neumann architecture. Efforts to reduce cost resulted in the forerunners to today's massively parallel SIMD machines, such as in the STARAN (1972 [Fos76]). Unfortunately, CAM architectures require data to be in a fixed length, fixed location format. More importantly, as just discussed, associative memory and simple matching are not enough to provide thorough massively parallel support for knowledge processing.

One early associative processor proposal, the ASP (1962 [Koh80]) showed that SIMD processing can support operations on relational structures, and more recently Willett *et al.* [WWR91] have demonstrated subgraph isomorphism on a general-purpose SIMD architecture. In general, however, SIMD architectures are restricted to well-structured problems that feature regular patterns of control and uniformly structured data. Modern SIMD machines are optimized almost exclusively for numeric applications, so it is necessary to extend current SIMD designs to better tailor them to knowledge processing. This includes recovering some of the lost CAM features. Other extensions such as local data addressing generated at each SIMD processor and limited local control can greatly reduce the restriction to uniform data structures.

In light of these many issues, extending associative processing and the SIMD architecture to serve as massively parallel support for knowledge processing is highly promising. Emphasis on processing relational structures, particularly retrieving such complex objects from large knowledge bases offers new architectural insights and innovations. Although simple hardware speedup will reduce processing time for a given problem size, many knowledge processing tasks require time exponential in the problem size. It is necessary to employ a

Figure 1.4: The First Content-Addressable Memory? [Zus93]

combination of algorithmic improvements, architectural features, and hardware technology to provide the significant increase in problem size required by new applications and research projects, as is done in the MISC research. It is particularly important to exploit heuristics and algorithms for avoiding exhaustive search and object comparisons, both in terms of speed and reducing system cost through more effective use of hardware resources.

1.2 Statement of Thesis

This dissertation addresses associative processing as a framework for providing massively parallel retrieval from large knowledge bases. The thesis, briefly stated, is:

> **SIMD processing can be extended to efficiently support operations on dynamic and structured data objects characteristic of knowledge processing.**

The author originally thought it would be necessary to make fundamental changes to the SIMD architecture; this has not proven to be the case. The key new results are as follows.

- Multi-level indexed search methods can provide speedups similar to full data parallelism but with far fewer processors.
- Multi-level indexed search and other pruning effects introduce scaling effects in the number of processors required and the memory per processor.
- The scaling effects and AI and example algorithm characteristics have important architectural implications. These implications include multiple small SIMD arrays each with a high ratio of internal computation to external communication to other

arrays, high bandwidth access to a large off-chip memory for each processor, a wide-link enhanced mesh for communication within an array, and processor enhancements to speed content-based processing, flag manipulation, and distance-metric calculation, and improve local control and local addressing.

- For the mix of example algorithms it is better to emulate content-addressing operations on a standard arithmetic-logic unit than to include dedicated matching circuits in an "active memory" architecture.
- The resulting architecture provides an order of magnitude speedup for the three example algorithms given similar hardware resources, despite requiring high pin count die and advanced packaging.

Other contributions include the following.

- Parallelizations of several associative retrieval algorithms, ranging from an associative memory on simple attribute vectors to subgraph isomorphism and partial order maintenance.
- An extended associative processing model that succinctly expresses knowledge processing algorithms (including multiple grains of parallelism implicit in compound data objects) emphasizing the underlying mathematical structure and simplifying programming.
- Extension and improvement of a prior early analysis tool, allowing projection of speed, area, power, and cost metrics at die, packaged chip, and systems level from a high-level architectural description.

1.3 Dissertation Roadmap

Developing the MISC architecture through the analysis of fundamental operations of example algorithms (rather than of particular programming language requirements or execution profiles of existing codes) encourages innovation by minimizing the bias of a particular language or programming style. Examining a range of algorithms and incorporating general knowledge processing requirements and generic primitives makes results more widely applicable; MISC is intended to be a general-purpose symbolic processor design rather than an application-specific architecture. An early analysis tool is used to evaluate the implementation implications of architectural choices in advance of detailed design, establishing cost-effectiveness through an integrated approach to algorithm, architecture, and implementation issues.

Chapter 2 provides background and prior art. The first section describes associative processing and its basic hardware implementation, content-addressable memory. The next section discusses three example algorithms used to focus MISC's development and evaluation. The following section surveys numerous parallel processors and discusses several parallel machines. This is followed by a sampling of parallel programming languages. The final section discusses the general characteristics of AI programs and prior work on massively parallel AI.

Chapter 3 describes the research methodology in more detail, and presents the first contribution of the research: an early analysis tool, Suspense, that projects system metrics such as speed and silicon area given only a high-level description of the system. Chapter

3 also describes the architectural-level simulator, 2MISC, that "executes" MISC code to generate instruction profiles and cycle counts.

Chapter 4 presents the associative programming model used to express the example algorithms for subsequent analysis. After discussing the relationship between AI operations, associative primitives, and mathematical formalisms, the chapter annotates SEGUL, the specific pseudocode notation for MISC.

Chapter 5 describes massively parallel codes for the three example algorithms expressed in the SEGUL pseudocode and analyzes their characteristics. Much of this dissertation's results come from this analysis, particularly how the number of processors and the amount of memory per processor scales with the problem size, and the effect on system architecture.

Chapter 6 describes the development of the MISC architecture, beginning with a discussion of implementation technology implications and constraints. The next section describes architectural requirements and series of trade-off analyses.

Chapter 7 presents the resulting MISC architecture. The first section covers the benefits of the system architecture and describes the supervisors. Subsequent sections cover new features incorporated into the PE microarchitecture, the enhanced mesh communication network, and the array memory in detail. Sections also cover implementation details of critical elements and present early-analysis projections of the performance and other metrics for a proposed complete MISC implementation. The final section gives machine-level instruction formats and microprogramming examples.

Chapter 8 evaluates research results. This includes how well the architecture and programming model meet the challenge of knowledge processing as well as their generality. This chapter also presents simulation results of MISC's performance, and compares them to those of other machines. The simulations validate prior analysis and design decisions.

Finally, Chapter 9 summarizes the work, highlights the contributions of this dissertation, and suggests implications and avenues for further research.

2. Background and Supporting Research

This chapter provides background for the MISC research topics, summarizing influences and establishing prior art. The first section defines associative processing, providing a general framework for this dissertation's approach to parallelizing KP. The second section introduces the example algorithms used to evaluate MISC's architecture and guide its development. The third section describes prior parallel architectures, emphasizing their strengths and weaknesses for AI algorithms, followed by a section summarizing programming methodologies for these and other machines. The final section discusses issues in providing massively parallel support for AI.

2.1 Associative Processing

Associative processing is central to this dissertation's approach to retrieving and manipulating complex data objects as a highly parallel foundation for knowledge processing. It extends the capabilities and efficiency of the exhaustive content-matching of content-addressable memory hardware to better serve as the basis for recall and reasoning. Here we discuss high-level associative memory, low-level content-addressable memory, and associative processing as the means of bridging the two as background for the associative processing hardware and programming issues discussed throughout this dissertation.

Associative Memory

Associative memory encompasses a wide variety of phenomena related to human memory performance. Kohonen defines two types of associative memory [Koh80]. The first, direct association (the most common usage of the term), refers to the recall of one or more patterns by the input of another pattern. As basic recall, direct association provides a single input-to-output mapping based on similarity of content, physical, or logical relations and typically deals with ordered (positionally encoded) sets of inputs. Second, indirect association involves *inference* via multiple intermediate associative mappings. In indirect association, *structural relationships* are an important part of the patterns [Koh80]. By these general definitions, all three of the example algorithms (SDM, semantic nets, and graph database) described in Section 2.2 are different types of associative memory. In addition to mappings and inference, pattern completion from a partial or erroneous input is an important feature of associative memory. Most important, associative memory modules can include learning. In its most basic form, associative memory is simply table-lookup, hashing, or pointer following; at its most complex it encompasses aspects of human recognition, recollection, learning, and reasoning.

Content-Addressable Memory

Content-addressable memory (CAM) is a physical embodiment of a minimal associative memory, accessing data by content rather than by an address as in conventional computer memory. Matching operations are inherently parallel and all operations are performed within the memory itself rather than in a central processing unit. CAM hardware typically provides an $O(N)$ speedup for N data elements, at the expense of $O(N)$ hardware. Since data memory also requires $O(N)$ hardware, CAM provides a factor of N improvement in space-time for matching operations. Although $O(\lg N)$ and even $O(1)$ time software alternatives exist, such as hashing and indexing, these lose efficiency or break down altogether when faced with approximate matching and similar tasks. CAM advantages are largely restricted to matching operations.

Figure 2.1 shows a generic high-level view of CAM systems. The comparand register contains the bit-vector to be matched and the mask register contains a bit-vector indicating in which bit-positions mismatches are to be ignored. In *bit-cell CAM* the CAM array contains memory with a comparator at each bit. Mismatches within a given word are ORd together and the word processing element (WPE) is simply one or more flags for match and intermediate results. Although the added hardware is only a small constant-factor increase in area over the passive storage of conventional RAM, most non-matching uses of CAM are inherently bit-serial and the bulk of the added circuitry idle at any given time. This penalty is particularly severe when the relative production costs of low-volume (and usually fully custom) CAM chips are compared to high-volume commodity memory chips. *Bit-serial CAM* achieves a cost-performance improvement by providing only a single bit-serial comparator for each memory word, enabling economical extensions. In bit-serial CAM, the CAM array is regular memory (often a separate commodity RAM chip) and the WPE is a bit-serial matcher and flags. In both types the multiple match (*responder*) resolver selects one of potentially many matching words based on the arbitrary physical ordering in the CAM array, typically in $O(\lg N)$ time. Communication between the WPEs can be provided to further support algorithms with more than just independent matching operations.

CAM hardware primitives include masked compare, conditional write, set or clear flags, communicate, and read or mark first responder. Numerous operations can be build from the CAM hardware primitives, as shown in Table 2.1, where N is the number of data items and B is the number of bits in the data item. The asymptotic times in the table require *full data parallelism* (one processing element per data item). CAM operations are highly implementation independent; several proposed optical CAMs provide virtually the same matching operations [AB91, Lou91]. Both types of CAM are restricted to static, fixed-length, ordered lists of attributes.

Associative Processing

CAM hardware and programming techniques can be extended to support more sophisticated matching and faster manipulation of the selected data. With the addition of arithmetic-logic unit and communication hardware the CAM WPE generalizes to a SIMD processing element. The difference between bit-serial CAM, associative processors and SIMD machines is fuzzy, and occasionally more in the nature of the intended applications and programming style than architecture. Nevertheless, there are several distinctions. Associative and SIMD processors typically have much more memory per processing element (PE). SIMD processing elements have evolved to be more complex than the CAM WPEs, which have remained very simple in order to permit placing hundred of words (of typically

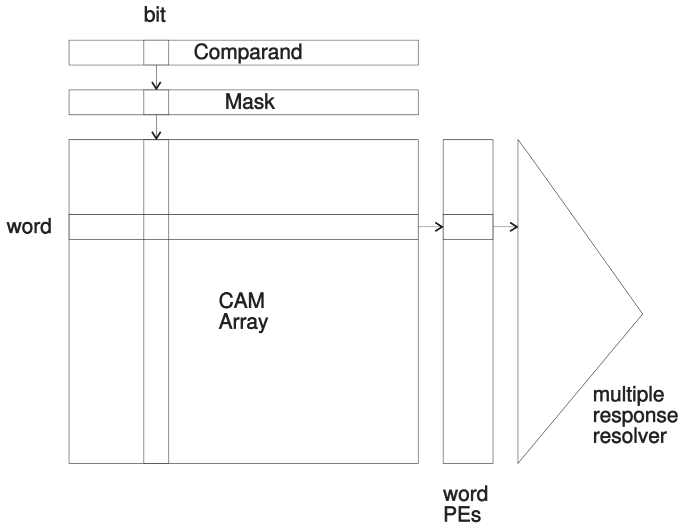


Figure 2.1: CAM Programming Model

Time	Operations
$O(1)$	exact match, partial match, and, or, not, set union, set intersection
$O(B)$	closest match, magnitude compare, range compare, maximum, minimum, mean, next above, increment, decrement, add, subtract
$O(B \lg B)$	Hamming distance
$O(B^2)$	multiply
$O(\lg N)$	set membership, is set empty
$O(B \lg N)$	permute, median, size of set, merge, log-sum
$O(N)$	minimum spanning tree
$O(N^2)$	all pairs shortest path
$O(N^2 B^2)$	matrix multiplication

Table 2.1: Bit-Cell CAM Operations (after [FKS85, Fos76])

of 32 to 64 bits each) on a single chip. In comparison to generic SIMD architectures, associative processors emphasize matching operations and broadcast comparands, and include a multiple-response resolver.

The key focus of this dissertation is generalizing and extending hardware for simple matching operations to efficiently support content-based selection and manipulation of complex data objects as the basis for massively parallel AI.

2.2 Example Algorithms

Although MISC is a general-purpose architecture, several knowledge processing methodologies were particularly influential in its development and evaluation. They were selected to cover a variety of important knowledge retrieval and inferencing approaches, spanning connectionist to symbolic AI paradigms, with differing computation and communication requirements. Other applications and the general characteristics of AI codes were also an important part of MISC's development.

2.2.1 Sparse Distributed Memory

Kanerva's sparse distributed memory (SDM) [Kan88] implements a neural network associative memory but avoids the vector-matrix multiplication normally used to determine neural activations. It is most similar to a two-layer perceptron neural network in which the weights between the input and hidden layer are fixed random Boolean values, the weights between the hidden layer and outputs are learned integer values, and inputs and outputs are Boolean. Kanerva's key contribution is the analysis of the associative memory's behavior in computational geometry terms, drawing directly from prior models of cerebellar function [Mar69, Alb91] to provide multi-layer functionality without backpropagation. It is both computationally efficient and biologically plausible.

In SDM an m-dimensional input vector induces a $\{0,1\}^m$ parameter space, which is sparsely populated by storage locations (weights). An output is not stored at a single location, but rather is distributed over many locations. Operation is summarized in Figure 2.2(a). To store associations, all locations within a specified Hamming distance of the input vector are incremented in positions corresponding to 1s in the associated output, and decremented in positions corresponding to 0s in the output. To retrieve associations, all locations within a given Hamming distance are activated. The corresponding bit positions in the active locations are summed and thresholded to produce the output vector. With input vectors typically 100 to 1000 bits long, the parameter space is enormous and with the fixed weights one would expect that only an infinitesimal portion of the possible Boolean functions could be learned. Kanerva's insight was that the distance distribution of points in the high-dimensional space gives rise to desirable associative properties for common functions and classifications even if extremely sparsely populated. The distance distribution of points in the Boolean hyperspace approximates a binomial distribution. At low dimensionality ($m \leq 10$) the distribution approximates that of Euclidean distance, but at higher dimensionality the distribution becomes more and more narrow. As a result, for any given point (input vector) there are very few close neighbors and very few 'opposites' (distance approaching m); the vast majority of points are 'indifferent' at a distance nearly equal to $m/2$. At high dimensionality a specific input is nearly orthogonal to any other point in the parameter space, and retrieval of (or storage at) irrelevant points is highly unlikely. As shown in Figure 2.2(b), for a given input all locations within a given radius are active. If a second input is sufficiently close many locations will be shared between the two. This gives rise to associative mappings, generalization, and other properties typical of low-level cognitive processes. These behaviors can be used to remove noise, complete partial inputs, and generalize from a training set. Feeding output back to the input can be used to store sequences and improve output convergence.

SDM continues to be a rich research area (in topics such as modeling human memory, algorithm alternatives, and emerging applications), requiring a general-purpose high-performance computer. Prager and Fallside have extended the model to include continuous

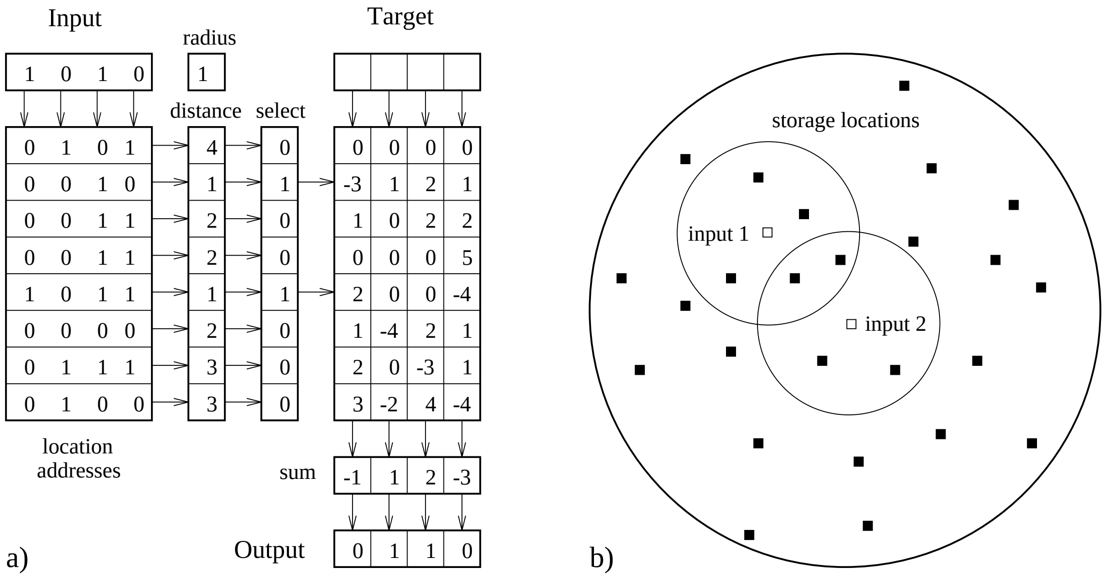


Figure 2.2: Sparse Distributed Memory

variable inputs, with significant improvement in accuracy over the Boolean version [PF89]. In Jaeckel's selected-coordinate design, only some of the input pattern bits are processed for storage location activation, and in his hyperplane design each storage location corresponds to a hyperplane in the parameter space [Jae89a, Jae89b]. Rogers applied genetic learning to modifying the first layer of weights, significantly improving accuracy and developed methods for interpreting the resulting weights [Rog90]. Kanerva proposed a k-fold network for distinguishing between temporal patterns such as 'ABDA' versus 'ACDG.' Research into potential end-user applications have included speech recognition [PF89], character recognition [HC91], and weather prediction [Rog90]. Investigating these variations requires the flexibility of a general-purpose processor, and exploration of modular nets composed of multiple SDMs requires extremely high processing speed.

SDM provides associative memory and direct inference behavior through activating near neighbors in the input space, offering several advantages. Based on a geometric formalism, SDM is amenable to analysis for uncorrelated inputs, but experimentation is necessary for correlated inputs and unusual input distributions. SDM does not use backpropagation and thereby learns more quickly. Only those weights relevant to the current input are processed and SDM does not require multiplication or matrix operations, greatly improving computational efficiency.

2.2.2 Semantic Networks

Semantic networks also attempt to capture human memory performance, and are considered a form of associative recall in which data is retrieved based on content and *structural relations* [Fah85]. Semantic networks represent symbolic knowledge in terms of concepts, properties, and their relations. As shown in Figure 2.3, a semantic network (SN) is a graph in which nodes correspond to objects, facts, concepts, or attributes and edges represent typed relationships, making structural knowledge explicit. Inheritance and various inference techniques are supported [DM93]. Extensive research over the last two decades has demonstrated the expressive power of semantic networks and their similarity to human associative recall and analogic reasoning [CM87]. Semantic networks are equivalent in representational power to first order predicate calculus, but their knowledge representation can greatly reduce both computation and storage requirements. Semantic networks have been particularly successful in research relating to end-user areas such as natural language understanding [DM93].

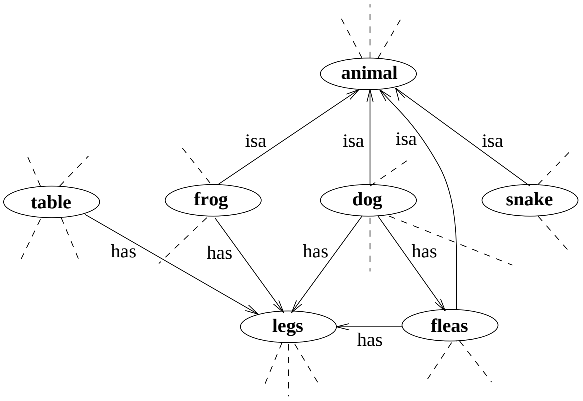


Figure 2.3: Portion of a Semantic Network

Semantic networks operate by *marker propagation.* Operation is typically in three phases. The first activates nodes via various matching operations. The second propagates flags (or values such as probabilities or confidence levels) along the edges of the graph. The third performs various operations when a marker is received by a node, based on marker type and node state. For example, determining if there are any animals with legs in the semantic network of Figure 2.3 would involve marking the animal node with marker 1 and the legs node with marker 2, propagating the markers backwards across the isa and has links, then identifying all nodes that have received both markers. High-level functions such as attribute inheritance and inference are achieved by iterating through multiple marker propagation cycles.

2.2.3 Graph Knowledge Bases

Matching complex, structured objects such as graphs requires processing considerably beyond the attribute matching of content-addressable memory. Indeed, an individual graph comparison requires exponential time in the worst case. Here we discuss a particular graph-based approach to knowledge processing, conceptual graphs. The algorithms and issues present are also important in other graph-based representations, with applications such as chemical databases, molecular biology, natural language processing, representing visual scenes, robot planning, and reasoning by analogy.

Conceptual Graphs

Conceptual graphs (CGs) are a specific instance of a graph-based knowledge representation in which concept-types are represented as nodes and relations as edges. Figure 2.4 gives a simple example. Unlike semantic networks which operate by marker or message passing, CGs manipulate entire graphs as data objects, performing operations such as joining graphs and testing for subgraph isomorphism. The CG formalism maintains the hierarchy of concepts (such as the isa links in semantic networks) separately, so that a CG knowledge base is a collection of separate graphs whereas a semantic network knowledge base is typically a single huge graph.

Sowa combined earlier work in existential graphs (Charles Peirce, 1896) with more recent work in semantic networks to develop CGs with the representational and inference power of predicate calculus [Sow84]. Many common representations such as type hierarchies, entity-relationship diagrams, parse trees, dataflow graphs, flow charts, state-transition diagrams, and petri nets can be considered special cases of CGs, and other knowledge such as chemical formulas and chess positions are readily expressible in CGs [ELR95]. CGs represent knowledge as a set of small graphs (i.e. semantic networks) with a standard mapping to both logic and natural language [Sow84] in a fully detailed formalism. This knowledge representation is particularly demanding in its need for match and search operations involving relational and recursively defined patterns. Key operations include subgraph isomorphism, subsumption, and canonical formation operations [Sow84]. A recent call-for-papers announced that CGs have been approved as a standard for information interchange and recently proposed as a normative language for conceptual schemas by the ANSI Committee on Information Resources Dictionary Systems [Tep93]. The same call-for-papers notes that

> Over the past 10 years, they [conceptual graphs] have been widely used as a semantic representation for natural language and as a graphic system of logic for expert systems, theorem provers, and database design. Researchers have developed a sizable software base and continue to build upon it. Successful implementations include: rule-based systems, database systems, knowledge-based systems, knowledge engineering tools, enterprise modeling, management information systems, conceptual information retrieval, medical informatics and natural language applications, among others [Tep93].

Subgraph Isomorphism

Subgraph isomorphism (SI) has been practically applied for years in chemical database retrieval, CAD, and more recently in CG implementations but is nevertheless NP-complete; in the worst case comparison time is exponential in the size of the graphs. There is a growing recognition of the importance of graphs as a representation of logical relations in general [Gen83, Win79] so that SI is an important operation despite its time complexity. Fortunately, the expected case time is empirically polynomial [Ull76, WWR91]. The worst-case time results from generally pathological cases that rarely occur in practice [Bar88, CK80, Sus65] so that parallelization can provide a difference in tractable graph size. Further, graph size is bounded for most applications. Ullmann's refinement SI algorithm uses Boolean matrix multiplication to prune the binding search, empirically giving expected

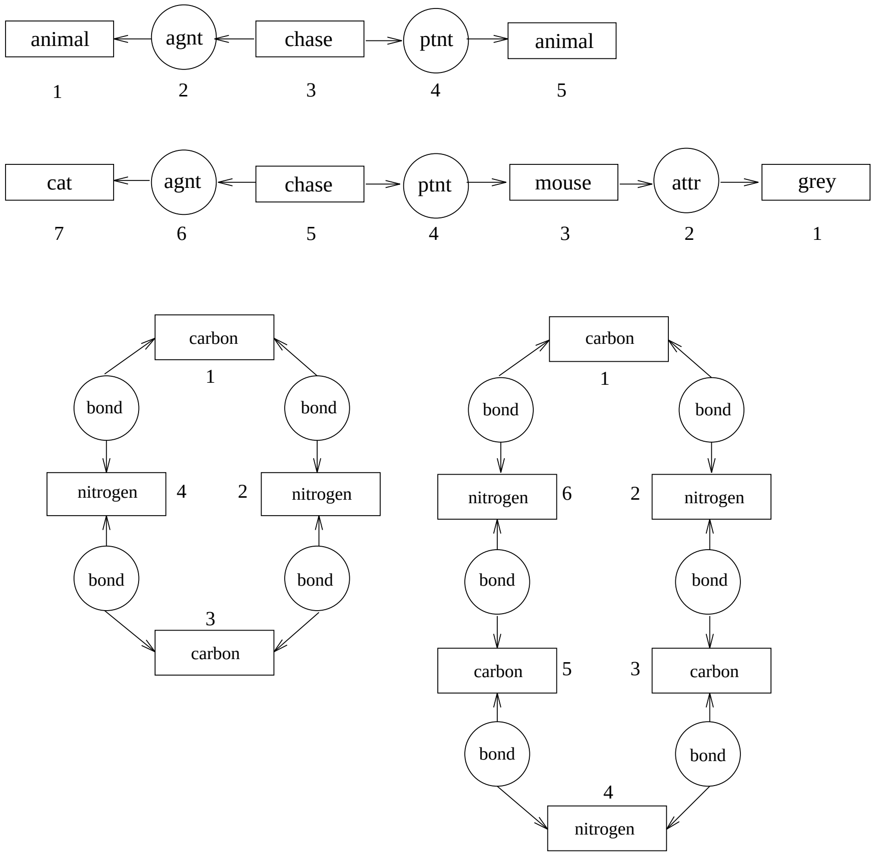


Figure 2.4: Simple Conceptual Graphs

time $O(n^4)$ where n is the number of nodes in the largest graph [Ull76]. Willett *et al.* have parallelized Ullmann's algorithm, using $O(n^2)$ 1-bit SIMD PEs to reduce time to typically $O(n^2)$ [WWR91].

Multi-Level Indexed Search

Multi-level indexed search (MIS) techniques can dramatically reduce search time when individual comparisons are time consuming (such as with subgraph isomorphism) by reducing the number of necessary comparisons [Lev92a]. Instead of providing only a single level of indexing as in conventional databases, Levinson's Method III creates a partial ordering (poset) over the knowledge base (KB), typically using the subsumption relation more-general-than. Objects are screened by predecessors in the poset and in turn screen successors. Smaller and simpler objects prune out comparisons on larger more complex objects. Various codings on the transitive closure of the poset can perform much of the pruning without traversing the partial order. Method III empirically prunes by a factor of $O(N/\lg^2 N)$, where N is the number of objects in the KB.

It has been proven that Method III is superior to conventional single-level indexing if there is structure to be taken advantage of in the KB [Lev92a]. Baader, Hollunder, and Nebel show Method III to be the fastest of all surveyed complex-object retrieval methods [BHN+92]. In addition to significantly increasing the speed and efficiency of retrieval, Method III supports conceptual clustering, generalization, planning, parsing, rule selection, and machine learning. The subsumption ordering relation allows quick access to generalizations, directly useful in case-based reasoning (reasoning by analogy), as well as to specializations, useful as a basis for machine learning [ELR95]. Sowa has noted that Method III, with its inherent retrieval of specializations and generalizations of a graph (rather than just a single best or exact match), has important implications for inferencing and may prove dramatically better than "any other knowledge-based system going" for large KBs [Sow], and the methodology is being incorporated into KL-ONE.

Hierarchy Coding

CG and other KR systems include a concept-type hierarchy to represent subsumption between the atomic concepts. Figure 2.5 shows a small example; the hierarchy is essentially a semantic network with only *isa* relations. Both the concept-type hierarchy and the MIS partial order can be encoded using *lattice codes* in order to speed certain operations [AK+89]. The simplest code is just the transitive closure of the graph edges, shown in Figure 2.5. These codes are N bits long, where N is the number of objects in the graph, for total storage $O(N^2)$. More efficient coding schemes use don't care states and reuse bit-patterns to reduce storage to $O(N \lg N)$ [Ell93]. The lattice codes can be used to test subsumption and find greatest-lower and least-upper bounds in constant time. Other operations, such as marking sub-hierarchies and determining the intersection of sub-hierarchies, can be done in constant time on a parallel processor.

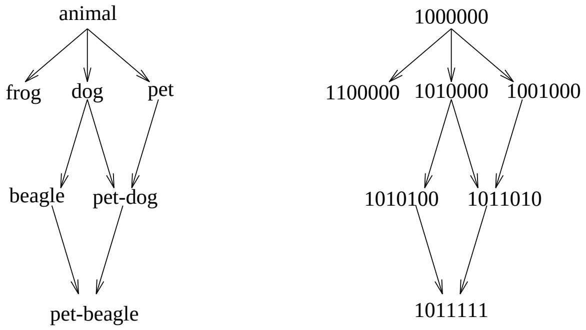


Figure 2.5: Lattice Encoding by Transitive Closure

A Graph-Based System

In summary, when knowledge is represented as independent graphs, complex graph algorithms such as subgraph isomorphism are required to access and manipulate the knowledge. As these algorithms are time-consuming, special techniques such as multi-level indexed search and hierarchy coding are needed to manage graphs and similar complex objects [ELR95].

The CG workbench *PEIRCE* is the result of a significant group effort to provide a common toolkit for research and applications in CGs and related graph-based retrieval and knowledge processing [EL92]. Organized in 1992, the PEIRCE group includes over 80 researchers with subgroups covering topics such as programming standards, storage and retrieval, parallel hardware, linear and graphic notation, conceptual catalogs, constraint programming, inference and theorem proving, learning, natural language, and vision. The resulting public domain software workbench incorporates multi-level indexed search (Method III and its successors) as well as lattice encoding, and is a superset of logic programming.

2.2.4 Related Algorithms

The example algorithms all feature a quick and simple exhaustive comparison or preprocessing to significantly reduce the number of data objects subsequently processed. The author believes that this approach is applicable to a number of other approaches to knowledge processing. Here we briefly discuss a few examples.

Combining Representations

Levinson has recently proposed a *universal data structure* (UDS) that combines semantic network, conceptual graph, and relational database representations [Lev94b]. It incorporates features of neural networks, case-based reasoning, and the Rete match-compilation algorithm. The foundation of UDS is the representation of all data as sets of objects and mathematical relations over the objects, organized as in the multi-level indexed search methods. A node hierarchy stores the primitive objects and their subsumption relationships over one another, a relation hierarchy stores templates and instances of relations on the objects, and a graph hierarchy stores relational structures (including conceptual graphs) as sets of relations and binding (join) information. Preliminary results indicate that UDS efficiently supports retrieval, inference, domain-independent state-space search, and pattern-based machine learning. The multiple hierarchies support multiple views of the data, offering the benefits of SN, relational database, and CG systems as appropriate for varying types of queries. CGs are stored in a compacted relation-based form speeding subgraph isomorphism testing, with most subgraph matchings pre-compiled [Ell92], further speeding matching and retrieval. The dependency on the extrinsic representations (*i.e.* which structures and sub-structures are stored) is a current difficulty. UDS is currently being tested on specific problem-search domains as well as in the PEIRCE workbench.

Nearest-Neighbor Classifiers

SDM is closely related to nearest-neighbors classifiers; classification is a function of some number of closest previously classified points. Cover and Hart show that nearest-neighbors classifiers make no more than twice the number of classification errors of a Bayes classifier (which uses the statistics of all observed points as well as *a priori* expectations of the occurrence of class members) for large samples of pre-classified points, irrespective of the underlying distributions [CH67]. For low-dimensional inputs, a k-d tree data structure permits locating the nearest neighbors in less time than exhaustive search. This data structure recursively partitions a k-dimensional space into subregions, responding to the data distribution along each dimension, building a binary tree whose interior nodes contain the partitioning information and whose leaves contain a small number of data points each. The subspace containing a particular point can be found in $O(\lg N)$ time by traversing the tree, where N is the total number of points. In this general sense, the algorithm is a type of multi-level indexed search. Unfortunately, for nearest neighbor search $O(2^k)$ adjacent bins must be searched. In addition to summarizing k-D tree algorithms, Omohundro shows how nearest-neighbor search can serve as the basis for neural-net like behavior, including generalization, clustering, dimension reduction, and nonlinear mapping [Omo87]. Cost and Salzberg recently developed a symbolic-distance measure for a nearest-neighbors classifier, with good results in predicting protein secondary structure and identifying DNA promoter sequences [CS93].

Miscellaneous

More traditional AI algorithms, such as IDA*, unstructured tree search, and general tree-recursive algorithms map well to SIMD and MSIMD architectures [Coo93, Kum93], as do branch and bound and other state-space search methods. Algorithmic and architectural similarities indicate additional applications should be found in areas such as sequence comparison, computational geometry, other graph algorithms, relational databases, general searching and sorting machine learning, matrix operations, and FFT with end-user applications such as CAD, computer graphics, image processing and signal processing, machine vision, robotics and autonomous vehicles, data compression, computational biology, planning and scheduling, management information systems.

2.3 Architectures

Space does not permit a survey of even the parallel processors most influential in architectural research and commercial production. Instead, this section summarizes and compares examples of different architectural classes to provide a context for MISC. The reader is referred to several surveys of specialized database, knowledge base, neural network, CAM, LISP, Prolog, and functional programming machines [KT88, Kog91, DFM89, CD89, WLL90].

2.3.1 Associative Processors

Associative processors offer fast highly-parallel matching with low silicon area and minimal communication. Among the numerous CAM integrated circuits developed in the last two decades, two are (unsuccessful) commercial offerings: the Am95C85, a CAM-based “articulated queue,” [Adv88b, Adv88a] and the MD1210, a bit-serial fuzzy matcher [Mic89]. Two 20 K bit CAMs, one by Motomura *et a.* and one by Ogura *et al.* are among the largest research research chips [$\mathrm{M}^{+}90$, $\mathrm{O}^{+}89$]. Content-addressable memories are, however, generally restricted to “flat” data such as attribute lists and tuples in a relational database, and are usually slow at operations other than matching. We note a few exceptions below. Several associative processors combine a bit-cell CAM array with simple bit-serial units for additional operations. The *Coherent Processor* features trinary logic (stored don’t-care states) for applications such as tree and substring search, graphics, pattern and symbol recognition, neural networks, and as a Prolog accelerator [Sto91]. Wade presents a system with a similar architecture [WS89].

PCAM

Proximity Content-Addressable Memory (PCAM) (1990) is a smart-memory architecture for close-match database search. Designed by the author, it is the kernel from which the MISC processor microarchitecture grew. PCAM exploits massively parallel hardware for flexible searches in large knowledge and databases [Rob89a, Rob89b, Rob90]. Several matching functions are supported: individual Boolean attributes, bit strings, integer magnitude, and within/outside of range. The ℓ_1 norm[1] is used as the distance metric, with the extension that distances in each dimension can be individually weighted. By selecting the maximum distance a record can be from the query and specifying the number of responses to be returned, PCAM can execute several types of search: exact match, partial match, fixed radius, best match, and k-nearest neighbors. Results can be read out in order of distance without sorting. PCAM quickly executes searches that would otherwise be difficult due to partial or noisy queries or high-dimensional data, with speedup linear in the number of PCAM words. PCAM has four subsystems: conventional memory, PE array, multiple response resolver network, and controller, as shown in Figure 2.6. The array PEs are bit-serial CAM augmented with magnitude comparators and an 8-bit distance adder, permitting simultaneous comparison and distance accumulation. This improves performance on Hamming distance calculation by a factor of nearly 40 over bit-serial alone. The resolver network handles multiple matches, providing identifiers of matching elements in an arbitrary order. Resolution time is $O(\lg P)$,where P is the number of PCAM PEs, but constant factors are such that a 1,024 PE system can identify the first of any number of matches in one or two PE cycle times. With full data parallelism, a 1,000 Boolean attribute k-nearest neighbors search can be performed in less than $100\,\mu$s. Each 0.2 cm CMOS (complementary metal oxide semiconductor) VLSI (very large scale integration) prototype chip has eight PEs, each with only 545 transistors at a minimum feature size of 2 μm. The chips worked on first fabrication and a small proof-of-concept system with 16 PEs was built as a memory-mapped IBM PC peripheral board. A simple threaded interpreter translates high-level operations into a stream of instruction codes written to the appropriate locations. Algorithms programmed and executed include the activation phase of SDM and several computational geometry algorithms. A single-board system with several hundred PEs in $1.2\,\mu$m CMOS could accelerate a workstation by a factor of nearly 100 for high-dimensional k-nearest neighbors search.

---

[1] The ℓ-norms are specific instances of the Minkowski distance metric, $\rho_M(x,y) = \left(\sum_{i=1}^{n} \mid x_i - y_i \mid^{\lambda}\right)^{1/\lambda}$ where x and y are two vectors of length (dimensionality) n and $\lambda \in \mathbf{R}$ [Koh80]. The ℓ_1 norm, also called Manhattan or city-block distance, is the Minkowski distance metric with λ of 1. When x and y are Boolean, the ℓ_1 norm is Hamming distance. The ℓ_2 norm is Euclidean distance. The ℓ_∞ norm (Minkowski metric in the limit as λ approaches infinity) is also commonly used in pattern matching and classification.

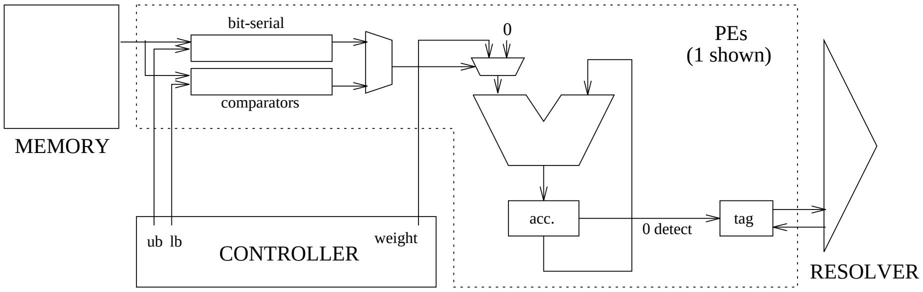


Figure 2.6: PCAM PE Datapath

LUCAS

LUCAS (1986) is a typical bit-serial associative array processor [FKS85]. Each PE consists of a 5-input 4-output ALU implemented as a 1 K by 4 bit PROM, and four 1-bit flags. A single-stage shuffle-exchange interconnect network connects the PEs to each other and to the off-chip memory, a mere 4 K bits per PE. LUCAS also includes a multiple-response resolver network. A full system includes 128 PEs, a control processor, and an I/O processor. Proposed PE enhancements include an up/down counter, local addressing, and a bit-serial multiplier [FKS85]. Although containing the necessary minimal support for associative processing, most of the example algorithms used to evaluate LUCAS are numerical. The algorithms cover areas such as matrix multiplication and FFTs, simple graph algorithms, relational database operations, and image processing.

ASP

The Association-Storing Processor (ASP, 1967) is of both historical and architectural interest [Koh80]. Although apparently never built, the architecture foreshadowed recent work in semantic network parallel processors, relation-based processing, and routing algorithms. The design is a 2-dimensional mesh of PEs, each of which contains one word of CAM and routing circuitry. Each PE stores either a source/destination node label, link label, or relation. Relations are represented as triplets, [source, relation, destination] with each element stored as the coordinates of the PE holding the label. Processing is performed by cycles of associative matching and marker routing. The proposed routing is very similar to dimension-order wormhole routing, with a simple pass-through to cope with contention. Programming is described in terms of control structures to be matched and replacement rules, as shown in Figure 2.7.

2.3.2 SIMD

Single-instruction multiple-data stream (SIMD) architectures have the advantage of avoiding the duplication of program memory, instruction decoding, and sequencing circuitry at each processor as required in multiple-instruction multiple-data stream (MIMD) systems. SIMD also avoids the synchronization overhead incurred by MIMD architectures. Unfortunately, speed and processor utilization are often severely compromised due to the single instruction stream.

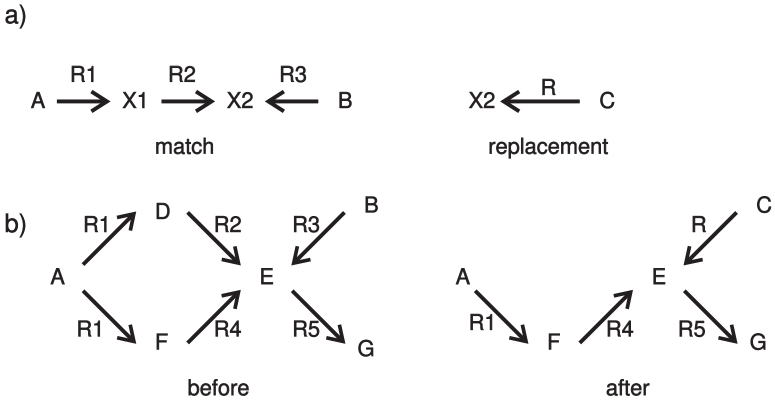


Figure 2.7: ASP Relational Structures (after [Koh80])

CM-1, CM-2

The Connection Machine CM-1 (1985) is an early massively parallel machine with up to 64 K PEs, and was among the first to use a hypercube interconnection network [Hil85]. Originally designed for semantic networks and AI, it turned out to be poor for such applications. The CM-1 PE features a simple bit-serial ALU that can compute arbitrary 3-input, 2-output Boolean functions. Each PE has eight 1-bit flags and only 4 K bits of off-chip memory. Performance per PE is low due to bit-serial operation and a slow 4 MHz clock (2 μm CMOS gate array). The large latency of the hypercube implementation, which is 30 times slower than computation, limits performance. Communication speed restricts effective use of the Connection Machine to algorithms with a high computation to communication ratio, the opposite of the original design philosophy.

The CM-2 (1987), produced by Thinking Machines, Inc., is commercial machine widely used in scientific, engineering, and database applications. The CM-2 has more memory than the CM-1 (up to 64 K bits per PE), adds one floating-point processor for every 32 bit-serial PEs, and doubles the clock speed to 8 MHz (2 μm CMOS) [TR88]. These improvements (in particular the floating point chips) greatly increase performance, primarily for numeric codes.

MPP

The Goodyear Massively Parallel Processor (MPP) (1983), developed for NASA / Goddard, is based on the STARAN and ASPRO associative processors [Bat82]. In addition to its original purpose of Landsat image processing, applications include weather forecasting, digital system simulation, and neural networks. The full system has 16 K bit-serial processors in a 2-d mesh network. The edges of the network can be connected to form a linear array, cylinder, torus, or skewed torus. In addition to the processor array, a full system includes a staging memory for transposing data to and from the bit-serial format, the array control unit, and the program and data management unit. Clock frequency is 10 MHz. Off-chip memory is 1 K bit per PE, expandable to 64 K bit per PE. As shown in Figure 2.8, each PE has a 1-bit full adder and general purpose logic unit, programmable shift register, and various flag, control, and routing registers (all 1 bit). A mask register controls conditional execution. Horizontal microcode permits several PE operations per cycle. In particular, input-output (I/O) data can be shifted into and out of the array simultaneously with arithmetic and logical operations, providing 160 M bytes per second I/O with only 1% processor overhead. This is particularly important for image processing. Despite having one quarter the number of processors as the CM-1, the MPP is 50% faster due to the increased clock rate, additional PE functional units, and communication network.

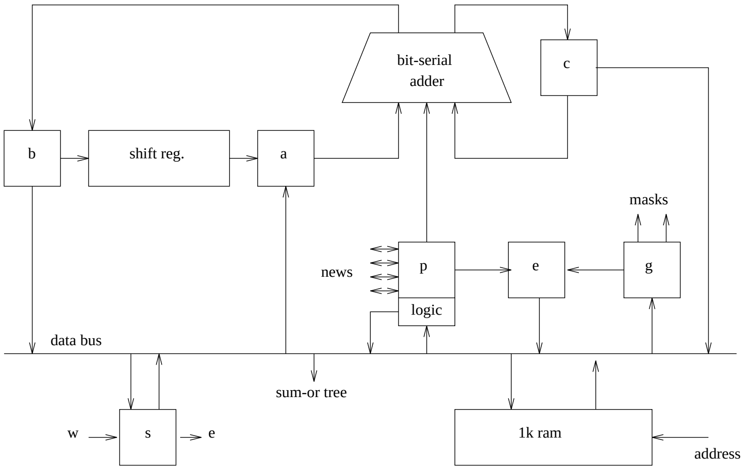


Figure 2.8: The MPP Processing Element

Blitzen

Blitzen is a VLSI successor to the MPP, with many of the improvements suggested by Batcher [Bat86, B+90a]. The ability to modify the SIMD address by or-ing 10 bits from the shift register is a significant enhancement. Due to pin count limitations, however, this can only be used to access a small on-chip memory of 1 K bit per PE. Pin count limitations also require that 4 PEs share each off-chip memory. Another enhancement allows each PE to use a state bit flag to select between the broadcast logical operation or its compliment. This, in conjunction with modifications to the shift register, speeds bit-serial multiplication. Clock frequency is 20 MHz (1.25μm CMOS). Unlike Batcher's proposal, the interconnection topology is not a multi-stage interconnect network or hypercube; the developers consider both of these impractical for a highly integrated VLSI implementation. The designers note that the 2-d mesh has a diameter of 128, offers quick and efficient communication, and that many applications map directly to the topology. The mesh's performance is improved for some communication patterns by allowing each PE to directly connect to all 8 neighbors while still requiring only 4 connections per processor ("X-mesh.") In addition to the reduced speed of bit-serial operation, Blitzen lacks many of the requirements of associative processing. Although a 1-bit wired-or line is provided for matching operations, far more support is needed; Blitzen is designed primarily for numeric computation.

MP-1, MP-2

The MasPar MP-1 (1989) [Nic90] and MP-2 (1992) [Mas92a] are contemporary commercial processors, and feature 32-bit PEs. The MP-2 increases on-chip datapath widths by a factor of eight and doubles bandwidth to off-chip memory, more than doubling speed over the MP-1. In addition to the ALU, a separate exponent unit speeds floating point operations. The MasPars do not include hardware multiply; B-bit multiplication and division require $O(B)$ cycles. The PEs support local addressing, and memory access, computation, and communication can be pipelined. On-chip memory consists of forty 32-bit registers, and up to 64 K bytes per PE of off-chip memory. Off-chip memory datapaths are only 4 bits wide, and several PEs share each memory. Although this is not a crucial limiting factor for scientific and engineering codes where most data resides in on-chip registers and many computation cycles are spent on each value, this can be a significant bottleneck for other types of applications. The MasPars include both a 2-d mesh and a multistage interconnection network (MIN). The MIN provides general global communication and is implemented as a 3-stage crossbar network. The mesh network can bypass PEs, permitting skipping over intermediate PEs in SIMD long-distance communications. System speed is 12.5 MHz with 32 PEs per 1.0 μm CMOS chip. Other than a global-or for status checking, the MasPars provide no special support for associative or symbolic processing.

BSYS

BSYS is a 1-d systolic processor [Hug91]. Although this dissertation research does not survey application-specific systolic architectures, BSYS is a programmable systolic array and suggests several algorithms that could be efficiently executed on SIMD arrays using a systolic programming style. It influenced the specific MISC implementation proposed and analyzed in Section 7.2.1. In addition to its programmability, BSYS features a shared-register architecture in which register banks are placed between the PEs, and each PE can access the banks on its right and left. All memory is on-chip, to take advantage of the systolic paradigm, 128 bits per PE. Each PE has an 8-bit wide ALU capable of performing addition/subtraction, shifts, and all logical operations of 2-inputs, plus 8 1-bit flags. A single 2 μm CMOS chip contains 47 PEs and register banks. System frequency is 12 MHz. A 32-chip single-board system should provide 5,000 MOPS performance. Designed primarily for DNA sequence comparison, BSYS should be efficient for numerous other algorithms.

Stanford SDM Prototype

Stanford developed an application-specific processor for SDM capable of iterating a 256-bit input 80 K storage location SDM 50 times per second (10^9 connections per second) [FKB89]. The system supports Kanerva's k-fold model, with 8192 storage locations per fold and up to 16 folds. Hamming distance calculation and radius thresholding is performed by special-purpose hardware. Calculation is location-serial, with a 256 bit wide computation circuit. Weight storage and summation is performed by standard 68030 microprocessor single board computers. There is one special-purpose card and one single-board computer per fold.

2.3.3 MIMD

MIMD architectures provide full programming- and control-parallelism flexibility, but at the expense of larger and more complex processors and communication synchronization and data coherency overheads.

CM-5

The CM-5 (1991) is a significant departure from the prior Connection Machines [Thi92]. The designers abandoned bit-serial PEs, SIMD, and the hypercube network of the CM-2. The CM-5 does retain a data-parallel SIMD-like programming model. Each processing node is a 40-MHz RISC microprocessor with four 20-MHz floating-point vector co-processors[2]. Systems with as many as 1,024 processing nodes have been shipped; a system of 16 K nodes with a peak performance of 2 TFLOPS is claimed technologically possible (though economically infeasible) for the CM-5 architecture. The CM-5 uses a 4-ary fat tree communication network, and increases link bandwidth by a factor of 4 at each level, maintaining a capacity of 5 M bytes/s per processor and avoiding congestion near the root. Links near the leaves have higher capacity (20 M bytes/s per processor at the leaves) to allow for local communication with 1-, 2-, and 3-d physical simulations. The architecture is clearly intended for scientific and engineering numeric operations, with no specific support for symbolic processing.

Paragon

The Intel Paragon (1991) is a parallel system intended for scientific and engineering applications. Unlike the prior hypercube Intel iPSC systems, Paragon features a 2-dimensional mesh network, with sections partitioned for computing and I/O. The I/O processors are based on the 80386 microprocessor, and provide disk and other input/output scalability and the ability to operate completely independently of any host. The computational processors are based on the i860 microprocessor and include a floating-point coprocessor, local memory, and interfacing on a single printed-circuit board (PCB). The system backplane includes a router for message-passing communication.

T3D

The Cray T3D (1993) is a 3-d mesh architecture designed for use in conjunction with the Cray Y-MP and C90 vector processors [Oed93]. Each processor is a DEC-Alpha RISC microprocessor with DRAM memory. Two processors share a routing interface to the communication network. Memory is distributed and globally addressed. The Alpha chips incorporate features for parallelism: a fetch-and-increment register, an atomic-swap register, and message processing independent of the ALU. The T3D emphasizes advanced technology high-speed communications. Each bidirectional link has 300 M Byte/s bandwidth. Coupled with the 3-d (as opposed to 2-d) mesh topology, this improves system scalability, although as systems become larger link congestion will eventually occur [B^+94]. An additional tree

---

[2] One might thereby argue that the CM-5 is a MSIMD architecture as well as a vector processor.

network provides global-**and** as well as global-**or** for barrier and status synchronization. The current system scales to 300 GFLOPs peak (at 150 MHz), and future systems target 1 TFLOPS sustained. Using standard RISC microprocessors, the PEs are optimized for numeric and workstation applications and no support is provided specifically for symbolic processing.

2.3.4 MSIMD

A hybrid of SIMD and MIMD, MSIMD architectures combine the advantages of each. This category includes systems composed of both SIMD and MIMD processing elements, as well as those with processors that operate in both MIMD and explicitly SIMD modes. Three such machines, DADO, NON-VON, and FAIM, are particularly relevant to MISC.

DADO

DADO (1987) was designed for production systems, sorting, and matching [Sto87]. Systems include 1,023 standard microprocessors each with a custom communication co-processor and 20 K bytes memory, and were intended to scale to 8,191 processors. The binary tree communication network allows dynamic partitioning of the machine into multiple SIMD-mode subtrees rooted at a processor operating in MIMD mode. Although elegant for production systems, tree networks have significant latency and congestion problems with general communication patterns. Further, a slight increase in the data-set size can double the size of a partition, roughly halving PE utilization. The prototype showed low speedups (2 to 31 times over a VAX 11/750) on production rule matching. The designers projected a second-generation DADO with ten times the performance, but it was never implemented.

NON-VON

NON-VON (1985) is a machine for AI that emphasizes basic database search for production systems and similar applications [Sha88]. SIMD PEs incorporate bit-serial and byte-wide functional units and registers but limits their simultaneous use. Multiple binary trees of SIMD PEs are rooted at a MIMD processor, and multiple trees are interconnected by a shuffle-based network. Memory is extremely small, at most 256 bytes per PE. Estimates showed a 16 K PE NON-VON to be only 10 times faster than earlier database machines [Sha88, AG89].

FAIM

FAIM (1987) supports object-oriented, logic, and procedural programming (LISP and Prolog) by attaching a small amount of CAM to RISC processors [A^+87]. Each CAM has only a small number of data objects per match circuit and provides only masked-comparison. The CAM is used to match the heads of logic clauses, providing single-level indexing for clause selection and subsequent unification. Each RISC processor is also accelerated by a separate instruction memory with sequencer and address calculator, a context-switching co-processor, and a communication co-processor. Communication is by a 2-d hexagonal toroidal mesh. Although the prototype has only 19 processors, systems with over 100 processors are discussed and a “hierarchical tessellation” of 52,281 processors with a communication diameter of 5 is presented. In the light of the link bandwidth required to avoid congestion, the latter is not realistic despite the hierarchical approach.

SNAP

SNAP combines CAM and additional processing to support the marker-passing SN paradigm [DeM92]. The initial design and long-range plans include fully custom processors combining CAM (for relation memory) with control, communication, marker, and ALU units [MLL90]. The current prototype implements these functions with several standard digital signal processing chips and multi-port memories. The SNAP prototype with 144 processing nodes is several thousand times faster than contemporary workstation SN implementations on applications involving inheritance, recognition, probabilistic reasoning, and learning, such as integrated speech and natural language understanding [M^+92].

IXM2

The IXM2 is a semantic-network machine directly inspired by the NETL model, its developers viewing “episodic memory as the foundation of intelligence” [H^+94]. The specific semantic network model incorporates “group identifiers,” in essence basic transitive-closure subsumption codes, to speed inheritance operations. The IMX2 features 64 processing nodes using Transputers, standard microprocessor chips designed specifically for MIMD parallel processing and featuring built-in serial communication links. Each processing node is augmented with 4 K 40-bit words of bit-cell CAM. The IXM2 has a hierarchical interconnection scheme in which clusters of 8 processors are fully interconnected, and the 8 resulting clusters are fully interconnected (using 9 additional Transputers for communication) [H^+91]. The machine is used primarily for research in natural language processing, including translation.

2.3.5 Other Processors

Table 2.2 presents additional systems surveyed in background research for the MISC architecture. The table excludes the numerous instances of application-specific architectures and CAM implementations, and only samples the wide range of research machines. The table includes the year of introduction (commercial availability or research publication), the architectural classification, type of interconnection network, number of processors, and system peak performance.

The H.A.P.S. is a hierarchical associative processor. The lower level performs high-speed matching via a 16-bit ALU, and includes an additional adder for local addressing. The second level performs slower disk data-stream processing. The AAP2, designed for circuit simulation, can modify bit-level instructions based on a local table [K^+86]. PASM is a research platform for AI applications, including vision, speech understanding, and expert systems [MSS85]. PASM has SIMD and MIMD operating modes in a partitionable topology scalable to a thousand processors. Paradigm and the much older cm* feature hierarchical communication networks and multi-level caching [CGB91, SFS77]. WASP, currently under development, was originally intended for image and signal processing.

Target applications now include supporting expert systems, LISP, and Prolog with low-level matching operations [LJ91]. Array modules can operate independently as SIMD or in groups as MIMD. Design principles emphasize waferscale integration (WSI) and fault tolerance. Each PE includes a 70 bit comparator, data and activity registers, a bit-serial adder and 1-bit flags. Memory is small, less than 16 K bytes per PE and communication is via a ring with bypass. Martinez proposed a "smart-memory" model architecture that combines CAM with additional processing [Mar90]. His model features "descriptors," a method for using CAM fields as indices into large, dynamic data structures. This work is not is not included in Table 2.2 as it did not progress past a general architectural concept.

Machine		Year	Description	Processors	Peak Performance	
ASP	[Koh80]	1967	SIMD, 2-d mesh, assoc.	proposed		
ILLIAC IV	[AG89]	1968	SIMD, 2-d mesh	64	500	MFLOPS
cm*	[SFS77]	1977	MIMD, multibus clusters	100	10	MIPS
UNC Cellular	[AG89]	1979	MSIMD, reduction, tree	1 M		
MPP	[Bat82]	1983	SIMD, bit-serial, 2-d mesh	16 K	3,000	MIPS
CM-1	[AG89]	1985	SIMD, bit serial, hypercube	64 K	2,000	MIPS
H.A.P.S.	[Stu85]	1985	SIMD, bus, assoc.	proposed		
nCUBE	[H^{+}86]	1985	MIMD, hypercube	1 K	1,945	MIPS
NON-VON	[AG89]	1985	MSIMD, bit-byte, tree, MIN	16 K+		
AAP2	[K^{+}86]	1986	SIMD, 2-d mesh, assoc.	64 K	10,000	MOPS
CLIP7A	[FMD88]	1986	SIMD, 2-d mesh	256	64	MIPS
LUCAS	[FKS85]	1986	SIMD, bit-serial, MIN, assoc.	128		
CM-2	[TR88]	1987	SIMD, bit-serial, hypercube	64 K	4,000	MIPS
AMT DAP	[TW91]	1987	SIMD, 2-d mesh, orth. bus	1 K	800	MFLOPS
GAPP	[NCR87]	1987	SIMD, 1,2-d mesh	2 K	900	MOPS
DADO	[AG89]	1987	MIMD, tree	8 K		
PASM	[MSS85]	1987	MSIMD, bus, xbar	1 K		
TRAC	[AG89]	1987	MSIMD, banyan	4		
FAIM	[A^{+}87]	1987	MSIMD, 2-d mesh	1 K+		
MasPar MP-1	[Nic90]	1989	SIMD, 2-d mesh, MIN	16 K	26,000	MIPS
nCUBE2	[H^{+}86]	1989	MIMD, hypercube	8 K	61,440	MIPS
April	[A^{+}90]	1990	MIMD, 2,3-d mesh, dataflow	1000+		
BLITZEN	[B^{+}90a]	1990	SIMD, bit serial, 2-d mesh	16 K	3,000	MIPS
iWarp	[B^{+}90b]	1990	MIMD, 2-d mesh	64	1,300	MFLOPS
BSYS	[Hug91]	1990	systolic, 1-d mesh	470	108	MOPS
NAP	[KF90]	1990	SIMD, various simulated			
SNAP	[M^{+}92]	1990	MSIMD, modified hypercube	144		
CM-5	[Thi92]	1991	MSIMD, fat-tree, vector	1 K	128,000	MFLOPS
Paragon	[Hwa93]	1991	MIMD, 2-d mesh		300,000	MFLOPS
IXM2	[H^{+}91]	1991	pseudo MIMD, hierarchical	64	278	MIPS
Paradigm	[CGB91]	1991	MIMD, hierarchical	100+		
Coherent	[Sto91]	1992	SIMD, CAM	4 K		
MasPar MP-2	[Mas92a]	1992	SIMD, 2-d mesh, MIN	16 K	68,000	MIPS
Cray T3D	[Oed93]	1993	MIMD, 3-d mesh	2 K	300,000	MFLOPS
WASP	[LJ91]	devel	MSIMD, bypass ring	64 K	40,000	MIPS

Table 2.2: Sample of Other Parallel Machines

2.3.6 Relationship to MISC

The MISC architecture, developed through analysis of the example algorithms and the general characteristics of AI codes, follows many of the general trends in parallel processing. Although early SIMD machines are bit-serial, MISC and recent machines have byte-wide and larger ALUs and some form of local addressing. MISC also uses a mesh topology for communication within the array, enhanced for long-distance communications so that supplemental networks such as the router in the MasPars is unnecessary. Like several of the AI-oriented systems, MISC also features multiple SIMD arrays, combining general-purpose "supervisor" processors with a more specialized SIMD arrays. MISC also combines a word-wide ALU with a single-bit ALU, as in PCAM and DADO.

Unlike contemporary parallel machines that have evolved for numerically-oriented applications, MISC returns many of the architectural features of associative processors and incorporates many new features to support complex matching and search operations. Rather than providing hardware for basic matching operations in support of higher-level AI instructions (primarily production systems), MISC is tailored to distance metrics, matching complex relational and dynamic data objects, and methodologies that minimize the number of required comparisons. This change in emphasis has important architectural implications in the array memory subsystem and the processor microarchitecture not present in prior architectures. Unlike the machines specifically for semantic networks or the Stanford SDM prototype, MISC is a general-purpose MSIMD architecture with features supporting many classes of knowledge processing methodologies.

2.4 Programming Languages

It is not sufficient to just have a faster machine; that machine must also be readily programmable or it will fall into disuse. This section describes some of the general requirements of parallel programming languages, then presents examples of collection-oriented languages. The MISC research includes joining collection-oriented and associative programming, described in Chapter 4, as a means of programming massively parallel knowledge processing systems with most of the parallelism hidden (implicit).

One reason for the relatively slow acceptance of parallel processing is the difficulty of programming multiple processors. Commercial parallel machines are programmed almost exclusively through extensions to existing serial languages, typically Fortran and C, as a means of leveraging programming skills and porting existing serial codes with some revision. These languages typically provide automatic parallelization of simple loops, inherently parallel array operations, or both. Thus programmers can ignore parallel processing issues if they are willing to pay a performance penalty. This approach appears less applicable to AI codes. Prior work on parallelizing LISP and Prolog shows limited high-level parallelism and disappointing speedups. It is important to recognize that using originally serial languages may overly bias algorithm development. Highly scalable, efficient parallel programs often involve specific parallel algorithms significantly different from their sequential counterparts. People often have difficulty porting a serial program to effective parallel code, and automatic translation has even more difficulty. On the other hand, recent results suggest that parallel programs can be efficiently executed on serial machines [BS90]. Although the dissertation emphasizes algorithm-architecture-technology interactions in the development of the MISC architecture, this section discusses parallel programming languages to provide a background for establishing the programmability of MSIMD architectures and a context for the associative programming model.

A parallel processing language must include mechanisms for controlling and communicating between multiple processors in a manner that does not unduly complicate the programmer's job. The three major models are message passing, shared memory, and data parallelism. In message passing, processors operate independently. Data exchange occurs by one processor sending another over the communication network; synchronization occurs by data dependencies and explicit messages. In the second model data in a single or distributed memory is shared between processors. Here the communication network serves to interconnect processors to memory rather than processor to processor. Mechanisms such as locks and semaphores are used to control access to data, maintaining consistent behavior and synchronization [Hwa93]. Most massively parallel machines use data parallelism in which a single operation is applied uniformly to a given collection of data. Data parallelism blurs the distinction between parallel extensions to existing languages and inherently parallel languages. Extensions can include functions that treat collections of data as a single object with implicit parallelism, as with matrix and linear algebra operations. Collection orientation is important to both ease of programming and efficiency in AI algorithms.

2.4.1 Example Collection-Oriented Languages

This section samples collection-oriented parallel languages, with an emphasis on those for SIMD architectures. The discussion is from the perspective of applications programming, excluding operating system and compiler issues. The reader is referred to Andrews and Schneider for a survey of general parallel languages and additional discussion of parallel programming requirements [AS83]. Sabot and Blelloch provide concise summaries of several collection-oriented programming languages [Sab88, Ble90]. Tables 2.3 and 2.4 combine these summaries and additional information. Prolog and LISP are excluded from the scope of this dissertation, due in part to limited speedups with control-level parallelism and their difficulty in programming applications. The sample also excludes systolic languages. The tables do include LISP characteristics from Sabot and Blelloch's study as a convenient comparison. The use of CAM to accelerate LISP and Prolog in a form of low-level parallelism is summarized in Section 4.1.

Several of the languages provide some support for matching and associative operations, but this support is generally restricted; only Gamma has matching as an integral part of its programming paradigm. Many of the languages have nested heterogeneous collection constructs, but only NIAL and Paralation provide fully parallel execution for such collections. Most of the languages support ordered collections (vectors or tuples) and many support ordering by a key value. Although many permit simple constructs for basis set data structures, only MultiLISP and SETL have this built in. pSather and dSather do not include collection operations as intrinsics, but provide support for many types of collections in its object libraries. Roughly half the languages allow explicit user control of communication. The remainder make communication implicit as part of the computation functions. Very

	Language		Year	Type	Base Language Parallel	Parallel Side Effects	Garbage Collection	Matching Emphasized
a	APL	[Ble90]	1962	serial	·	●	●	○
b	ASC	[P+94]	1992	'all'	●		●	●
c	C*	[Ble90]		SIMD	○	●	○	○
d	CM-LISP	[Ble90]	1986	SIMD	○	●	●	○
e	Cray Fortran	[Ble90]	1978	vector	○	●	○	○
f	Gamma	[BM93]	1986		●	○	○	●
g	Linda	[CG89]	1982	MIMD	○			·
h	*LISP	[Ble90]		SIMD	○	●	○	○
i	MPL	[Chr90]	1990	SIMD	○		○	○
j	MultiLISP	[Ble90]			○	●	●	○
k	NIAL	[Ble90]	1982					
l	Occam	[Bur88]		MIMD	●			○
m	Paralation	[Ble90]	1987	serial, vector, SIMD, MIMD	·	·	·	○
n	Parallel Pascal	[Ble90]			○	●	○	○
o	Pascal-L	[FKS85]	1983	SIMD	○		○	·
p	pSather, dSather	[FLR91, Sch92]	1991	MIMD, SIMD	○	○	●	·
q	SETL	[SDDS86]	1981	serial	·		·	·
r	SQL	[Ble90]		serial	·	(○)	·	·

○ no · somewhat ● yes

Table 2.3: Parallel Languages

Characteristic	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r
Collections																		
dynamic creation	●	●	○	●	○	●	●	●		○		○	●	●		·	●	
dynamic change	○		●	●	○	●	○	○		○		○	○	○	○	·	●	
atomic elements	●	●		●	●	●	●			○	●	●	●	●	●	·	●	●
structure elements	○	○		●	○	●	○			○	○	○	●		●	·	○	●
nested	●	·	●	·	○	●	○	○	○	·	●	○	●	○	·	·	●	○
homogeneous	●	●	○	●	●	●	●	●		○	●	○	●	●	●	·	●	●
heterogeneous	●	○	○	●	○	●	●	○		○	●	○	●	○	○	·	●	○
unordered	○	●		●	○	●	●			○		○	○		○	·	●	○
linearly ordered	●	●		●	●	●	●			○	●	○	●		●	·	●	○
grid ordered	●	●		○	●	○	○			○	●		○		○	·	○	○
key ordered	○	●		●	○	●	●			○	○	○	○		·	·	●	●
basis set	○	○	·	·	○	○	○	○		●		○	·	○	○	·	●	
Communication																		
explicit to user	○	○	●	●	○	○	(○)	●	●	○		●	●	●	●		○	○
integer mapping	●	○	○	○	○	○	○	●	●	○		○	○	●	●		○	○
pointer mapping	○	○	●	○	○	○	○	○		○		○	○	○	○		○	○
predicate mapping	○	○	○	●	○	○	○	○		○		○	●	○	○		○	○
scan collision	·	○	○	○	○	○	○	○		○		○	○	○	○		○	○
reduction collision	○	○	·	●	○	○	○	·		○		○	●	○	○		○	○
fixed shape	●	○	●	○	●	○	○	●		○		○	○	●	○		○	○
shape specification	○	○	○	●	○	○	○	○		○		○	●	○	○		○	○
sync at func call	○		●	●	○	○	○	●		○		○	○	●			○	
sync at intrinsics	○		○	○	●	○	○	○		○		○	○	○			○	
sync at elwise	○		○	○	○	○	○	○		○		○	●	○			○	
sync at data dep	○		○	○	○	○	●	○		●		●	○	○			○	

○ no · somewhat ● yes

Table 2.4: Parallel Language Characteristics

few of the languages provide combining operations on communication collisions, and only CM-LISP and Paralation allow the user to specify the operation for a given communication.

Inherently Parallel Languages

Of the languages in Table 2.3, only APL, Gamma, Occam, SETL, and SQL are "inherently" parallel; the others are parallel extensions to serial languages. APL is one of the earliest parallel languages in the sense that it treats arrays as single variables. Even so, it is implemented on serial machines and parallelization has proven difficult due to dynamic data objects. Similarly, SETL treats sets and tuples as single objects but to date is implemented only on serial machines. SQL (structured query language) for databases can be considered a parallel collection-oriented language in its treatment of sets of tuples in a non-procedural style although most implementations are on serial machines. Only Occam and Gamma were parallel from the beginning. Occam was developed specifically for the Occam follows the communicating sequential process model [Hoa78] rather than data-parallel processing. Each MIMD processor executes its own program and shares information by message passing. Computation continues after a send, but waits if a receive is executed and the information has not yet arrived. Occam also features multiple processes executing on a single microprocessor. Section 2.4.1 describes the Gamma programming model in detail.

Extensions to Serial Languages

The languages C*, CM-LISP, and *LISP are data-parallel extensions to their base languages, and were developed for the CM-2. They form the basis for the more recent Paralation work. Parallel Fortran extensions, including Cray Fortran, Fortran90, and HPF similarly parallize array operations and loops without critical data-dependencies. MPL, for the MasPar MP-1 and MP-2, provides more extensive extensions to C, fully supporting a data-parallel programming style with explicit inter-PE communication, reductions, and expansions. Pascal/L was developed for the LUCAS processor. Although LUCAS is an associative array processor, the extensions to Pascal are essentially the same as for more general language extensions, Pascal/L emphasizes parallel variables such as arrays, with expansions and elementwise operations and communication constructs. It includes instructions to find the first marked PE and sequence to the next marked PE using a multiple response resolver network. Sather is a serial inherently object-oriented language which compiles to efficient C code. It is public domain, and two groups are involved in developing parallel versions [Omo91]. pSather is a MIMD implementation using monitors. It allows multiple threads within a single object, and implementation includes versions for the CM-5, Sequent Symmetry, and Sparc Stations. dSather is a data-parallel SIMD version for the MasPar MP-2 and Fujitsu AP1000. Of particular interest, the Sather object libraries include numerous neural network models, computational geometry routines (including k-D trees), graph algorithms, string search, parsers, image processing and numerous additional data structures [OL92].

2.4.2 ASC

ASC is a recent associative programming model featuring dynamic memory allocation of structured data as well as MSIMD operations [$\mathrm{P}^{+}94$]. The underlying associative virtual machine can be emulated on a wide variety of platforms. Potter *et al.* have implemented ASC on the CM-2, Staran, Aspro, and other processors and propose associative processing as a programming paradigm for massively parallel computing. Algorithms coded under the model include minimum spanning tree, a Prolog interpreter, ray tracing, convolution, database management, and a compiler.

Pre-defined data types such as stacks, queues, trees, and graphs are stored as 2-dimensional tables. Each row consists of a primitive data element and a "structure code" which indicates the element's position in the data structure and supports fast associative processing of key operations. For example, an element of a matrix would be accompanied by its row and column coordinates. Tree connectivity is represented by structure codes similar to Stormon's tree coding; each node is given the code of its parent with an additional integer digit indicating which child it is. Another coding scheme related to quad trees represents binary graphs (graphs with maximum in and out arities of 2); general graphs are represented as adjacency matrices. ASC also includes single-level collections such as sequences, multisets, and sets but does not appear to have any set-based operations. Individual elements are dynamically allocated from a pool of free processors, with associative garbage collection.

The ASC model also incorporates MSIMD processing, presumed to be supported by multiple sub-array controllers in the virtual machine. Associative operations partition responders into mutually exclusive subsets, each of which gets a sub-controller.

The author encountered ASC after this dissertation was nearly complete. The reader will note features in the MISC model and coding examples in common with ASC and with associative programming techniques in general. The MISC model differs in its use of a uniform general representation for all compound data structures, high-level operations for matching and manipulating those structures, greater emphasis on sets, and incorporation of explicit communication. Further development may benefit from consideration of ASC's features. The following languages have already influenced the MISC research.

2.4.3 Gamma

The inherently parallel Gamma model is based on multiset transformation and emphasizes matching [BM93]. Its developers consider it a methodology for developing parallel programs, and not a practically implementable language [BM93]. All data is treated as a multiset of objects. Programs are specified as a matching condition (R) and action (A) to be performed on the match results. Wherever the R() predicate is true, its objects in the multiset are modified by the A() operator. The syntax includes binding between match and action variables, and the program terminates when there are no more objects in the multiset satisfying the match predicate. As the objects are contained in a multiset, the order of matching-action operations are not specified. Multiset operations such as union, difference, product, minimum and maximum are pre-defined. Multiset transformations are categorized as relaxation (start with an approximation and improve accuracy), expansion (decompose values into a collection of items), and reduction (multiset reduced to a singleton). These categories can be important in conceptualizing and developing inherently parallel algorithms. Since objects can be multisets or tuples, data structures representing objects such as graphs are readily definable.

Figure 2.9 gives an example of Gamma code for the Sieve of Eratosthenes method for finding prime numbers. The first line specifies that the result is a gamma-function of the matching condition `R` and the action `A` on the set of numbers 2 through n. A match occurs whenever any element `x1` of the set is a multiple of any other element `x2`, and the action is to replace `x1` and `x2` with `x1`. Banatre and Metayer give other examples, including generating the Fibonacci series and factorials, prime factorization, sorting, longest motonically increasing sequence, majority, graph connectivity, shortest path, minimum spanning tree, convex hull, and image edge-detection [BM93].

```
sieve(n) := Γ(R,A)(2...n)
      R(x1,x2) = multiple(x1,x2)
      A(x1,x2) = x2
```

Figure 2.9: Sieve of Eratosthenes in Gamma [BM93]

The Gamma model has two fundamental limitations. First, only local information is available. Problems requiring control decisions based on the overall state of computation cannot be described by the Gamma model. Unfortunately, this excludes backtracking and other recursive strategies such as divide-and-conquer; additionally, algorithms such as maximum flow, and productions are extremely difficult to describe in the Gamma formalism [BM93]. Second, each object is implicitly compared to all others. For binary comparisons, such as in the sieve example, time is $\Omega(N^2)$ but in the general case it is $\Omega(N^n/n)$ where N is the number of objects and n is the arity of the comparison, because all subsets of length n are match candidates. In its full generality a Gamma implementation is impractical; time is at least polynomial in the size of the multiset and exponential in the relation arity. This restricts any practical Gamma program to low-order matching relations.[3]

The Gamma model is important in its emphasis on inherently parallel algorithms and multiset transformation. In particular, the match-act structure confirmed the present author's initial work on the MISC associative processing model, demonstrating the power and expressiveness of a match-set-set operation cycle for a variety of locally-controlled algorithms.

2.4.4 Linda

Linda provides mechanisms for adding MIMD parallelism to existing serial languages through the addition of a few simple functions [CG89]. One if its strong features is its language and machine independence. Linda extensions have been included in C, C++, Fortran, Scheme, and Modula-2, and there are Linda implementations for a wide range of standard MIMD machines. Linda's key mechanism is a global "tuple bag," a multiset of tuples; the basic data object is a tuple rather than a hardware word, and tuples can represent data or processes. Data tuples are accessed by logical names rather than by address so that the tuple-bag is content-addressed. Communication is through persistent objects in the tuple bag rather than processor-to-processor communication.

---

[3] The present author notes that when restricted to binary relations, a simplified implementation is similar to systolic processing. Each object resides in a PE and a copy of the objects stream past for comparison. Both the originals and copies may be modified by the Gamma action.

Linda adds only four functions to the base language, out(), in(), read(), and eval(). The out() function places a tuple, whose elements are defined by the function's arguments, into the tuple bag and continues execution. The in() function attempts to retrieve a tuple that matches in the specified elements. If a matching tuple is found, it is removed from the tuple bag; if no matches exist, the calling process waits. For example, in('whats', ?x, 7) attempts to match a 3-tuple with whats as the first element and 7 as the last, returning a value for x if found (if more than one tuple matches, an arbitrary one is returned). The read() function operates the same as out(), but does not remove the tuple. The eval() function adds an "unevaluated" tuple to the global tuple bag. Its arguments include the name of a function to compute, along with its arguments, and typically a tuple identifier. Free processors can then retrieve unevaluated tuples to begin execution.

The logically shared tuple bag simplifies programming, avoids context switching (each processor executes a single process, then retrieves another), and provides scalability and dynamic load balancing [CG89]. On the negative side, an implementation must deal with either the intensive communication to a single tuple memory or the cost and time overhead of maintaining coherency in a distributed tuple memory. Additionally, the eval() mechanism typically creates one process for each element of the result, incurring significant overhead. Linda was influential in the MIMD-level programming model for MISC, suggesting matching tuple-messages as a means of process activation and inter-processor communication. Linda explicitly supports neither fine-grain parallelism nor nested collections.

2.4.5 Paralation

Paralation is a vector-based data-parallel processing model whose constructs can be applied to an existing base language. Although the existing compiler is a partial implementation of the model for the CM-2 (Paralation-LISP), the model is applicable to MIMD and vector-processor architectures as well as to SIMD and serial machines [BS90]. The abstract processor model is that of a scalar computer with an attached vector processor, which can be efficiently emulated on other machine architectures. Paralation adds *nested parallelism* to collection oriented languages; its key contribution is allowing operations on nested collections (collections whose members are themselves collections) to proceed in parallel, supporting multiple levels (grains) of parallelism. This, however, excludes iterative and other coarse-grain parallelism typical of MIMD models.

Blelloch extends [Ble90] Sabot's prior work [Sab88] to include compiler techniques for compiling operations on nested collections, including conditionals, by concatenating the nestings and including descriptor information to "flattened" vector operations. In a SIMD implementation, elements of a collection are uniformly and sequentially divided over the available PEs. Key operations can be categorized as *elementwise, scan (parallel prefix),* and "*permutations.*" Elementwise operations (such as plus, logical-or, maximum, and equality test) are applied between the corresponding elements of two collections. Scans apply similar operators over the elements of a single collection, providing partial results (*e.g.* max-scan[3 1 5 7 6] = [0 3 3 5 7], working from left to right and replacing each element with the maximum encountered thus far). The final category permutes elements of a collection in the general sense of a re-arrangement, not the formal mathematical definition of a permutation, and is the basis for expressing communication. In Paralation, one vector is the source and another is the destination indices for the corresponding source elements. A source element may be omitted from the destination and several source elements can map to the same destination element, in which case a reduction operator is applied. Reductions include plus, maximum, and concatenate. The programmer can also specify a default value for destination elements not receiving any source element.

In his dissertation, Blelloch demonstrates asymptotic performance similar to parallel random access machine (PRAM) computational models [Ble90]. He treats scans and permutations as constant-time operations; these in fact require computation and communication time, typically $O(\sqrt{N})$ or $O(\lg N)$ for 2-d mesh and hypercube networks respectively. With this adjustment, Blelloch's results show that time for nearly all algorithms analyzed is asymptotically as good as the exclusive-read exclusive-write (EREW) PRAM model, and typically a factor of communication time slower than the CRCW (concurrent-read concurrent-write) PRAM model. More particularly, he shows that the required segmented-scan operations (in which scans are applied independently over sub-collections) can be implemented in a small constant-factor number of calls to unsegmented scans and that slow-down for nested-collections is $O(n + a)$ where n is the maximum depth of nesting and a is the total number of different atomic element types (integer, Boolean, etc.) in the collection.

Algorithms coded and analyzed include sorting, computational geometry, simple graph algorithms, and matrix operations [Ble90]. The graph algorithms demonstrate efficient processing on graphs represented as nested collections. As such, Paralation provides a means of expressing multiple levels of parallelism on compound objects, but within the confines of a single instruction stream (serial control). In his dissertation Blelloch proposes exploring the limitations of serial control and considering parallel control via multiple vector-processors. This dissertation explores these issues and adds exploring nested collections for knowledge representation and nested parallelism for knowledge retrieval and processing.

2.4.6 SETL

Although intended for serial processors, the set-oriented language SETL has implicit parallelism in its syntax through its use of sets and tuples as data objects with compound operators [SDDS86]. These collections can be of arbitrary length, and their elements can themselves be sets or tuples. Operations on sets include the standard set-theoretic operations, and operations on tuples include concatenation, indexed retrieval, and sub-tuple operations. Set-former operations follow mathematical notation, such as {y in s | y < x}, and universal and existential quantifiers are part of the SETL syntax. Iteration over collections is included in addition to conventional program control constructs. Numerous string operations are provided and SETL supports recursion with specific program constructs. Collection operations include reductions and a `forall` construct which applies the body of the `forall` to all elements of the collection. Nested tuples form tables that can be sorted and searched. In particular, SETL includes map operations that operate on a set of 2-tuples. The first element of the tuple is a domain element and the second is the corresponding range element. Sample programs emphasize the set-oriented approach and include sorting, topological sort, binary search, Towers of Hanoi, Euler Path, text processing, and Huffman coding. SETL provides explicit programming constructs to support general backtracking.

Collections are represented internally using a variety of data structures including linked lists, bit strings, basis sets and hash tables; each offers different speed advantages for particular operations and varying space efficiencies. Selection is currently by programmer declaration, but subsequent work investigates heuristics for automating this, including a suggestion to use CAM for dynamic basis sets [SSS81]. A parallel SIMD implementation might offer significant speed advantages and simpler internal data structures and algorithms.

2.4.7 Programming MISC

The MISC associative model and and notation SEGUL, described in Chapter 4, draws freely from the various collection-oriented languages as an *ad hoc* pseudocode for parallelizing the example algorithms (SDM, semantic network, and graph knowledge base), avoiding the bias of existing serial languages. SEGUL may itself be the basis for an associative programming language. The model and notation is combination of nested heterogeneous collections for implicit low-level parallelism (drawing primarily from the Paralation model) with explicit high-level parallelism based primarily on LINDA. SEGUL also incorporates set operations and limited vector and matrix operations as well as an associative programming style similar to Gamma but with global information and control.

In addition to coherently combining the language features to support associative processing with multiple levels of parallelism, SEGUL emphasizes dynamic collections and collection matching.

2.5 Massively Parallel AI

This section summarizes AI research and applications using parallel and massively processors. Although there are many definitions, parallel processing is generally considered to effectively use at most a few hundred processors while massively parallel processing uses thousands of processors. The summary highlights general characteristics of massively parallel AI and their requirements of computer architectures. Although a fairly new topic, massively parallel AI is gaining widespread acceptance. In a recent two-volume book on parallel AI, over one fourth of the papers are on massively parallel AI [K^+94a, K^+94b]. Kitano and Hendler note that mainstream AI conferences now routinely publish work on massively parallel AI [KH94], and in a special issue of *IEEE Computer* on associative processing two of the six papers concern knowledge processing [IEE94]. Unfortunately, these publications arrived during the final stages of this dissertation and could not be integrated. In the remainder of this section, we summarize these recent publications, noting the difference in approaches between parallel and massively parallel knowledge processing and the difficulties SIMD architecture and AI processing pose for one another.

2.5.1 AI Research and Applications on Massively Parallel Processors

Table 2.5 lists papers from [K^+94a, K^+94b, KH94, IEE94, Gel93a] for parallel AI, defined here somewhat arbitrarily as using fewer than 512 processors. Table 2.6 does the same for massively parallel AI (work using 512 or more processors). Tabulating categories shows distinct differences between parallel AI approaches (often parallelizing methodologies developed for serial execution models) and massively parallel approaches (typically developed with massive parallelism in mind from the beginning). Massively parallel AI can be more than just faster parallel AI; perhaps its greatest importance is the promise of new heuristics and approximation algorithms exploiting highly parallel processing.

The parallel papers cover mostly logic programming and theorem proving, expert systems (rule-based systems), constraint satisfaction, and state-space search. As Kitano and Hendler note, this is for the most part serial reasoning [KH94]. Knowledge base machines, although their processor architectures are optimized for search, have almost exclusively limited their applications to supporting logic programming and production systems [KT88]. The work in Table 2.5 typically uses about 64 processors and achieves a factor of 16 to 32 speedup.

The papers on massively parallel AI cover mostly semantic network and related methodologies, neural networks, and search. The work typically uses 8 K to 16 K processors; parallel speedups range from an asymptote of 10 to 20 (game-tree search, DADO on databases), to linear at about 10% peak processor speed, to linear at nearly peak performance (randomized state-space search, structure matching in semantic networks). Many of these papers emphasize the difficulties of achieving good processor utilization and the importance of load balancing. Kitano and Hendler note the emphasis on associative processing, marker passing, nearest neighbor and subsumption operations, memory-based reasoning, neural networks, and genetic algorithms [KH94]. Semantic networks (particularly Fahlman [Fah85]) and neural networks (particularly Hopfield, Rumelhart, and Hinton [Hop82, RHW86]) were especially influential in reviving work on massively parallel AI.

Paper	Machine	PEs	Category
[SJ94b]			search
[ACP93]			search
[CR94]			vision
[Hen94]			search
[HJ93]			constraint satisfaction
[FFG94]			other
[SLH94]			other
[Pin94]			neural net, logic
[Ass94]			theorem proving, logic
[SS94a]			rule-based system
[Sha93]			neural net
[AG93]			search
[Kum93]			SN and related
[AB93]			neural net
[WG93]	CM-5		other
[LHB93]	Symmetry		constraint satisfaction
[Rob93]	custom		other
[HIM94]	custom		other
[LCV94]	transputer		neural net
[OMT94]	custom	10	other
[SJ94a]	transputer	16	theorem proving, logic
[Gel93b]	CM-5	32	SN and related
[CS94]	CM-5	32	other
[GW94]	WARP	32	vision
[LP94]	iPSC	32	constraint satisfaction
[ZM94]	transputer	48	constraint satisfaction
[MNT94]	AP1000	64	other
[CGA94]	Pixel	64	rule-based system, vision
[SS94b]	many	64	theorem proving
[NW93]	PIM	70	theorem proving, logic
[Ert93]	LAN	110	search
[Pea94]	Fujitsu	128	theorem proving, logic
[CMD94]	SNAP	144	SN, natural language
[Sat94]	nCUBE3	256	natural language

Table 2.5: Sample of Recent Parallel AI Publications

Paper	Machine	PEs	Category
[Kur93]			theorem proving, logic
[JS94]			vision
[Gel94b]			SN and related
[DB93]			other
[Kur94]			theorem proving, logic
[DN94]	CM-2		vision, search
[EAH93]	CM-2		SN and related
[Lan93]	CM-2		SN, natural language
[H^+94]	IXM2		SN, natural language
[O^+94]	IXM2		SN and related
[Fah93]	NETL		SN and related
[Twa94]	custom		genetic, classifier systems
[LH94]	custom		other
[YK94]	custom		neural net, learning
[CGK94]	nCUBE2	512	other
[HJ94]	transputer	512	theorem proving, logic
[BKS93]	CM-2	1024	other
[S^+93c]	DADO2	1024	rule-based system
[CGK93]	nCUBE	1024	search
[NM93]	CM-2	4096	genetic, classifier systems
[Gel94a]	CM-2	8192	SN and related
[CG94]	CM-2	8192	search
[KK93]	CM-2	8192	search
[JSS93]	MP-1	8192	theorem proving, logic
[PKF93]	CM-2	16384	search
[PW94]	CM-2	16384	natural language
[WD94]	CM-2	16384	neural net, learning
[ND94]	CM-2	16384	neural net, learning
[AHEK94]	CM-2	16384	SN and related
[EAH94]	CM-2	16384	SN and related
[JB94]	MP-1	16384	vision, neural net
[ZMHV93]	MP-1	16384	neural net, learning
[Coo93]	CM-2	32768	search

Table 2.6: Sample of Recent Massively Parallel AI Publications

Prior work emphasizes a relatively small number of algorithms, with generally disappointing results for logic programming and expert systems. Together, these results suggest that conventional SIMD programming restricts the types of algorithms which can be efficiently implemented.

2.5.2 Selected Examples of Massively Parallel AI

We have already reviewed three massively parallel AI algorithms in Section 2.2 and several architectures developed for AI applications in Section 2.3. From Table 2.6 we see that our example algorithms, semantic networks, sparse distributed memory, and knowledge base search are typical of current work in massively parallel knowledge processing. We now consider additional examples of massively parallel AI research and applications most relevant to this dissertation's research.

Inheritance Codes in Semantic Networks

Geller presents a coding for trees in which each node is represented by an integer-valued interval and proposes their use in semantic networks to avoid propagating across inheritance hierarchy links [Gel94a, Gel94b]. The code is formed by a right to left pre-order tree traversal; the resulting node label forms the lower boundary of the interval. The maximum of all descendants' labels form the upper boundary. A node subsumes its descendents by having an interval that covers those of all its descendents. These codes can be efficiently created, and Geller gives algorithms for incrementally updating the coded tree in time linear in the number of trees moved, logarithmic or constant in the size of the trees [Gel94a]. Operations on the codes appear to be limited to inheritance, however, and have not been extended beyond tree structures as would be required for typical hierarchies with multiple inheritance.

Semantic Networks and Structure Matching

PARKA is a knowledge representation and inferencing system combining the features of frames and semantic networks [EH90, EAH94]. Hendler *et al.* present an algorithm for finding subgraph isomorphisms in the large "semantic network" graph given a query graph in linear time [Kit94]. This is, however, at the expense of $O(Nn^2)$ processors (to use the graph knowledge base terminology and symbols) as exhaustive parallelism is needed to achieve linear time; each node and edge in the network must have its own processor. Hendler *et al.* report queries taking from 0.5 to 1.6 s on a 16 K processor CM-2. They also more restricted structure matching by DeMara on SNAP [EAH94] and by Higuchi on the IXM2 [AHEK94].

Memory-Based Reasoning

In essence, memory-based reasoning stores a large body of knowledge and experience and deals with new situations by retrieving and processing old situations, making the knowledge-rule tradeoff clearly on the side of knowledge. Retrieval techniques and measures of similarity are important. More formally, as developed by Stanfill and Waltz memory-based reasoning (MBR) combines retrieved results for a statistical interpretation [SW86]. As first proposed, MBR refers specifically to similarity-based induction directly from memory. MBR has been used in a wide variety of applications [YK94].

MISC and the Memory-Based Reasoning Philosophy

Most of the massively parallel implementations, including our example algorithms, can be loosely categorized as memory-based reasoning, the remainder as primarily state-space search. The underlying philosophy of MBR, as taken in the MISC research, is that of a large store of knowledge or experience, of which a small portion is accessed as relevant to a given problem and used as the basis for more traditional inference, reasoning, and planning algorithms. Retrieval algorithms to date, however, have concentrated primarily on matching attribute vectors rather than relational structures and exhaustive or single-level indexed search. Case-based reasoning [Kol93] also follows this generic memory-based reasoning model, and appears ripe for parallelization and advanced retrieval techniques. In Chapter 4 we briefly summarize additional projects incorporating content-addressable memory in support of associative processing for traditional AI methodologies such as logic programming and rule-based systems. These projects also follow the basic philosophy of extensive low-level matching in support of traditional AI procedures.

2.5.3 Knowledge Processing Characteristics and SIMD Parallelism

Excluding special-purpose hardware for specific paradigms, most massively parallel AI research and applications development is being conducted on SIMD machines more suited to numeric applications. The majority of work in Table 2.6 uses existing SIMD machines: over 50% use the CM-2, 10% use the MP-1, and 10% use the nCUBE (a MIMD hypercube machine). The remaining quarter use custom processors, including machines built from Transputer chips (4%).[4] These machines are tailored much more to numeric than symbolic applications.

The characteristics of knowledge processing introduce difficulties in running on SIMD architectures, and may be unduly influencing the development of massively parallel approaches. Developing SIMD architectures for AI codes is not an easy task. In addition to maximizing processor power, any parallel architecture faces three competing goals: minimize memory and communication latency, minimize communication time, and hide the remaining latencies. Parallel symbolic processing has several characteristics that further complicate system architecture and implementation, especially for SIMD architectures, making efficient system design even more difficult.

Most conventional AI programs have limited coarse-grain (control-level) parallelism; efforts to parallelize existing serial AI codes generally show poor processor utilization beyond 16 processors and negligible speedup beyond 32 [KT88, Kog91, WLL90, K^+94a, K^+94b, Gel93a, KH94]. This is true even for control-level parallel machines designed specifically for Prolog and LISP. Fine-grain (massive) and medium-grain parallelism must be used to achieve significant speedup, but AI and symbolic processing have several characteristics that make such parallelization particularly difficult [HGC87, WL89, WLL90]. The following characteristics of AI programs result in small speedups on conventional massively parallel systems.

- AI programs emphasize dynamic, variable-length, and linked data structures instead of the statically sized arrays of numeric applications.
- AI control structures are data dependent.
- Communication patterns are date dependent and often bursty.
- Referencing objects in memory exhibits less locality than in numeric processing.
- AI programs are recursion and subroutine intensive.

---

[4] The low-parallelism work uses a wide variety of machines, mostly MIMD, including the CM-5 and transputer networks.

Knowledge processing and machine perception do have two beneficial characteristics: progressive abstraction and reference locality. As an example of progressive abstraction, advanced vision systems progressively reduce massive quantities of input to more and more compact and abstract representation. As an extreme, data can begin as 200,000 pixels and end as a single yes/no indication of whether a particular object is present. As depicted in Figure 2.10, the lower levels of processing have high bandwidth communications in regular patterns whereas the upper levels exhibit nondeterministic patterns with much lower data rates. Although the extreme near-neighbor locality and highly iterative processing of finite-element numeric simulation models is not available, caching studies have found that AI programs show greater memory reference locality than general programs, albeit with different characteristics. In one study, AI codes required 20% smaller cache for an optimal hit rate and exhibited significantly greater spatial and structural locality [WCHC89].

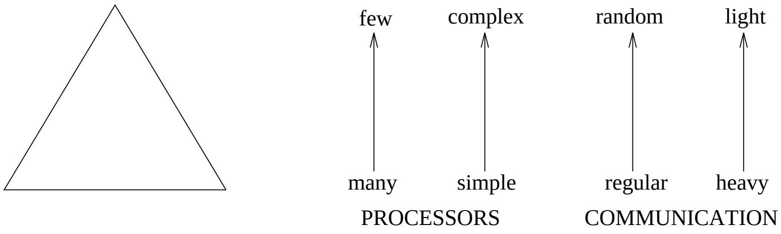


Figure 2.10: The Symbolic Abstraction Pyramid

Massively parallel AI, in distinction to coarse-grain AI, has generally operated within the limits of numerically-oriented SIMD machines. AI codes, however, pose special difficulties for SIMD architectures. This dissertation addresses both dealing with the difficulties and with exploiting the beneficial characteristics of AI codes.

2.6 Summary

In this chapter we have reviewed background material for the MISC research in terms of the context of prior work and the motivation for the current work, and in terms of introducing the example algorithms and key concepts of the MISC approach to massively parallel knowledge processing. MISC draws from a variety of areas: associative processing, AI algorithms, parallel architectures, programming languages, and massively parallel approaches to AI.

Most massively parallel AI takes an approach substantially different from coarse-grain parallel AI, emphasizing intensive retrieval from a large knowledge base on SIMD systems. The MISC example algorithms (sparse distributed memory, semantic networks, and graph knowledge bases) are typical of current research in the field and represent a range of computation and communication requirements. Parallel knowledge processing in general exhibits characteristics difficult to accommodate on SIMD architectures, particularly recent machines optimized for numeric codes, and may be unduly influencing the development of massively parallel AI techniques.

Early associative processors were largely inspired by issues in AI and featured bit-cell CAM with a comparator at each bit. As a cost-saving measure, these machines evolved to bit-serial CAM, and in turn to the early bit-serial SIMD machines such as the MPP and CM-1. Continued evolution of SIMD architectures has led to numerically-optimized machines such as the CM-2 and MP-2. Most of the MSIMD machines covered in this chapter are intended for AI applications, but except for the semantic network machine SNAP and IXM2, provide only simple low-level matching for production systems and logic programming, and similar approaches to AI.

Collection-oriented languages, treating compound data as a single object, form the basis of MISC's programming model. To this MISC adds an emphasis on associative processing, content-matching based access and manipulation as well as dynamic parallel data structures.

Most of the difficulties posed by AI code characteristics for parallel processing can be directly met by a MIMD architecture, but at the expense of the SIMD advantages discussed in Section 2.3. This dissertation investigates means of reducing the mismatch between parallel knowledge processing characteristics and *SIMD* machines, improving SIMD performance and hopefully encouraging new approaches to parallel AI. In particular, the research develops extensions to the generic SIMD architecture to better support knowledge processing, focusing on retrieval from extremely large knowledge bases. The key distinctions from prior work are emphasis on an associative collection-oriented processing model, relational structures, and retrieval pruning.

3. Research Methodology

This chapter describes the methodology used to develop and evaluate the MISC architecture. MISC's design results from analysis of architectural trade-offs regarding the fundamental operations of knowledge processing. Core algorithms from the target applications form the basis for architectural simulation and analysis, and early analysis tools anticipate implementation technology, providing feedback on the hardware implications of architectural alternatives. The following section summarizes the methodology used and the goals of the design development and analysis. This is followed by descriptions of the two analysis tools developed as part of this research: the *Suspense* implementation estimator, and the *2MISC* behavioral simulator. Appendices A and B detail the specific models used in Suspense and Appendix C briefly describes using 2MISC and gives several example simulations.

3.1 Design Philosophy and Methodology

Before describing the specific tools developed as part of this dissertation, we first look at the overall methodology into which they fit. The proposed MISC architecture is intended to be a cost-effective approach to general-purpose massively parallel knowledge processing, *not* the fastest application-specific hardware for the example algorithms. The design and evaluation methodology is based on generic application characteristics to encourage innovation by avoiding the local minima implied by a particular language or programming style. The analysis tools are flexible and quick in order to maximize the number of significant alternatives that can be investigated, and emphasize application-architecture-technology interactions to maximize cost-effectiveness. A range of algorithms as well as general knowledge-processing and generic machine primitives are examined to help assure that the result will be a widely applicable, rather than an application-specific, architecture.

3.1.1 Design History

A major design effort almost inevitably involves several iterations with considerable back-tracking and the simultaneous pursuit of several issues. The following describes a 'linearized' summary of MISC's development process, in three phases.

- Establishment of the characteristics of a range of key application algorithms.
- Evaluation of major alternatives and development of a baseline architecture.
- Evaluation of a proposed implementation and comparison to existing architectures.

Design development began with pseudocoding and manually profiling core algorithms from the example applications as described in Chapter 5. The algorithms were chosen not only as key examples of memory-based approaches to massively parallel AI but also to cover a variety of operations and communication patterns. The resulting instruction profiles and memory referencing characteristics, plus a qualitative understanding of processing, memory, and communication requirements for AI formed the basis of the initial architecture. The pseudocode notation SEGUL and programming model described in Chapter 4 was developed concurrently with the initial application parallelization and evaluation.

The research then evaluated the merits of the initial architecture's features and considered the trade-offs of numerous alternatives and additions as detailed in Chapter 6, again based on the manual profiles of instructions and memory referencing. The early analysis tool Suspense, described below in Section 3.2 provided feedback on the speed and cost implications of the various architectural choices.

Research then evaluated a proposed implementation of the resulting architecture. The implementation chosen was mid-range between that which could be completed as a university proof-of-concept prototype and that which would utilize advanced, unproven technologies. The behavioral simulator 2MISC generated profiles and cycle counts from a variety of algorithms expressed in SEGUL for the array architecture. Unlike the manual profiling, the simulation included specific algorithm input data, though for relatively small data sets. Future simulations of full applications with much larger data sets are appropriate to further evaluate and refine the MISC architecture. Suspense provided estimates of the time per cycle as well as other design metrics, permitting comparison to existing architectures.

3.1.2 Evaluation Metrics

In many mission-critical real-time applications elapsed time (T) is the only relevant criteria. MISC is intended to be a cost-effective architecture, so that a space-time (S-T) performance metric is necessary. The product of silicon area and execution time is appropriate for most parallel systems; the area per processor can be directly traded off against a larger number of processors of equal total area. Area and time are only two of many possible metrics for evaluating an architecture's implementations. Others include pin count, size, and power consumption; these often dominate a system's cost. The cost metric[1] calculated by Suspense is used as the "space" measure in the S-T product for evaluating MISC alternatives. Although this crudely approximates manufacturing cost in dollars, it is not a price estimate but rather a 'real-world' weighting of the different system components.

Of course, the evaluation of architecture-technology interactions requires some selection of a specific technology. The MISC analysis is based on CMOS VLSI, currently the leading technology for implementing high-performance cost-effective digital systems. More specifically, current leading-edge technology parameters are used, with the assumption that they represent mainstream commercial production four to six years from now, as has historically been the case. This avoids assuming any untested technologies. These parameters include $.35\,\mu$m minimum-feature size with 4 metal layers, 3 volt V_{dd}, and 500–1500 pin packages. Although $2\,\text{cm}^2$ die are feasible, smaller die are more cost-effective. DRAMs are assumed to be as large as 64 M bits[2].

---

[1] The cost measures may not be metrics in the formal sense, but it is standard practice to refer to such cost and performance measures as design metrics.

[2] Experimental 1 Gbit DRAMS have already been built. These cannot, however, be expected to be commodity-priced commercial products in the 5-year time-frame.

3.2 The Suspense Early Analysis Tool

3.2.1 Early Analysis Tools

Early analysis tools predict chip and system metrics, such as area and speed, while still in specification or other pre-netlist phases of design by employing a variety of first-order models and correlations. This allows evaluation of trade-offs at the earliest phases of design. Suspense analysis evaluated major alternatives such as bit-cell CAM versus SIMD, providing important input to the early MISC architectural specification. After introducing the prior tool, SUSPENS, and the key improvements incorporated into Suspense, this section describes each tool in greater detail.

Bakoglu's *SUSPENS* (Stanford University System Performance Simulator) is the basis for this dissertation's Suspense. SUSPENS can predict area, speed, power consumption, and other metrics at the die, module, circuit board, and system level with a minimum of user input. Input can be as little as circuit category (such as memory, microprocessor, and random logic), approximate number of gates, and minimum CMOS feature size. SUSPENS models interactions between key parameters such as device properties, on-chip interconnections, off-chip interconnect, and empirical correlations between chip characteristics and system architecture and organization (such as Rent's rule and typical logic depths and gate fan-outs) [Bak90]. The off-chip models include wafer-scale integration, multi-chip modules, and printed circuit boards.

Suspense (SUSPENS Extensions) adds models specific to RAM and CAM and allows 'tiles' of different circuit types to be combined on a single die. Extensions permit analysis of differently-sized dies and mixtures of package types. Finally, Suspense incorporates improvements in die timing and area estimators, and adds yield and cost-metric estimators. Suspense verification at the tile (register and RAM), chip (SRAM, DRAM, CAM, microprocessor, SIMD PE), and system (BLITZEN, MPP, MP-1) levels gave average errors of 17% for area, 13% for speed, and 34% for power. Cost metrics are based on regressions to published data, as described in Appendix B.

Early analysis permits quick exploration of architecture-technology interactions to improve system designs and to identify implementation pitfalls such as excessive power density or routing capacity requirements. Suspense is an important part of the MISC research effort, particularly in the following areas.

- Estimating speed and area for numerous architectural alternatives.
- Projecting the effects of VLSI scaling and other technology developments, and adjusting characteristics of other systems to permit comparison at equivalent technology.
- Establishing communication and multiple-response resolution times in terms of PE cycle times.
- Determining off-chip delay and module area effects for different system packaging and module technologies.
- Identifying the importance of pin count and packaging technology on die area.
- Projecting MISC performance and S-T comparisons with existing processors.

Early analysis tools are also important in projecting updates to new technologies, and Suspense would be useful in most computer architecture design studies.

3.2.2 Bakoglu's SUSPENS

This section summarizes SUSPENS, which forms the basis of Suspense. The SUSPENS equations are described in Appendix A. The chip and system models include numerous implicit and explicit assumptions in addition to those mentioned here, and the reader is referred to Bakoglu [Bak90] for a more complete description. Validation tests indicate that the assumptions are appropriate for the MISC domain. There are three basic models: die, module, and system. Bakoglu validated the die model against over 100 microprocessors (both research and commercial) and gate arrays, and the module and system models against 3 supercomputers. SUSPENS is divided into three models: die, module, and system.

Die Model

In the CMOS die model,[3] internal table-lookup based on user-input minimum feature size (λ) determines CMOS technology parameters such as transistor equivalent resistance and capacitance, interconnect resistance and capacitance, the number and pitch of metal wiring layers, and a default wiring utilization. A user-input circuit category (memory, microprocessor, gate array or random logic, supercomputer) selects effective logic-depth (including typical critical-path gate sizing, latch and setup delays, and on-chip memory access), gate fanout, and Rent's coefficients from empirical correlations, also by table look-up. Next, Rent's rule[Bak90] is applied to the user-input number of logic gates on the die to determine average on-chip interconnect length in terms of average gate size. The technology wiring parameters are then used to approximate the average gate size and convert interconnect length to centimeters (SUSPENS makes the assumption that gate density is limited by routing capacity). Die area is then simply the product of the number of gates and the gate area. Because pad area is ignored, SUSPENS is only valid for large chips with a relatively small number of pins. Effective gate output resistance and input capacitance are calculated from the technology and gate-fanout parameters. Average gate delay is then calculated from the gate capacitance and resistance, gate fanout, and interconnect capacitance and resistance at the average interconnect length. Die frequency is calculated as the gate delay times the effective logic depth plus the time to propagate a signal diagonally across die (to allow for clock and control-signal skew). Output driver delay is included at the module level. Total die capacitance is wiring (based on the assumption that all available wiring area is used up to the utilization factor) plus total transistor capacitance derived from the number of gates. Die power consumption is calculated as the product of V_{dd}^2, die capacitance, die frequency, and a default ratio of total capacitance to switched capacitance. Die pin count is calculated by Rent's rule.

Module Model

In the module (interconnected die) model, parameters are based on table look-up from user selection of the module technology: wafer-scale integration (WSI), multi-chip module (MCM), ceramic hybrid, or printed-circuit board (PCB). These parameters include wiring

---

[3] The die model can be modified for other VLSI technologies, including bipolar and gallium arsenide, with relatively minor revisions [Bak90].

capacitance, resistance, pitch, characteristic impedance, and typical number of wiring layers, pin pitch and capacitance, and the substrate dielectric constant. Rent's coefficients are also determined by table look-up from user input of the system type (microprocessor, gate array, or supercomputer). Calculations are similar to those in the die model. SUSPENS assumes the total number of gates in a system is uniformly partitioned across a number of uniformly sized die. The footprint of each die is taken as the maximum of the die area itself, any user specified package size, and routing area. The latter is based on number of pins per die, default signal fan-out, average interconnect length, and module wiring parameters. Module area is simply the die footprint times the number of die. Line length for module signal propagation is taken as the edge dimension of the module, which is assumed to be square.

The die output-pad driver-chain delay is calculated and added to the module signal-propagation delay. The effective output resistance is set to be the characteristic impedance of the module. The number of stages is determined by the ratio of the minimum-sized CMOS transistor resistance and the driver output resistance, and the transistor size ratio between stages (assumed to be 5, a compromise between minimum delay and minimum area). The maximum frequency that can be supported by the module is based on the total of the driver chain delay, pad RC (resistance-capacitance) delay, interconnect RC delay, and interconnect time-of-flight. This assumes that the system is pipelined at each chip crossing. The module model calculates additional power for inter-chip communication using the product of capacitance per line and the number of lines. Capacitance per line includes the output driver chain transistors, two pads, and the interconnect capacitance at the average Rent's length. A default signal fan-out, the number of chips, and the number of pins per chip determines the number of signal lines. Additional power for module communication is then calculated as in the die model, with the final factor being the portion of pins that switch in an average cycle.

System Model

The system model assumes that no active devices exist between modules, simplifying the model. In SUSPENS, system frequency is the minimum of all the modules and all the die, power is the total for all modules adjusted for frequency, and area is the total of all modules.

3.2.3 Suspense Extensions

The Suspense early analysis tool developed as part of this dissertation extends the SUSPENS model.[4] SUSPENS as described in [Bak90] assumes that the total gates in a system are uniformly divided among some specified number of equal-sized chips all of the same type. Unlike SUSPENS, Suspense allows non-uniform system partitioning and introduces a "tile" into the hierarchy of models. The user can specify sub-circuits (tiles) of differing types to be combined on a given die, and different die types can be combined on a module. In addition to the general die models in SUSPENS, Suspense adds new models specific to RAM and CAM that exploit the regular topology and circuit characteristics to

---

[4] Suspense was written independently in C based on the equations in [Bak90], and is not a modification of any existing SUSPENS code.

improve accuracy and simplify user input. Suspense also adds models for differing packaging technologies, and permits mixtures of module technologies in a single analysis run. Suspense also extends SUSPENS by adding specific module interconnection topologies, such as a 2-d mesh for array processors, and a cost-metric model. Suspense does not include a system model, as each MISC array is a separate module with asynchronous communication between arrays; the SUSPENS system model is easily calculated manually. In addition to a tile-die-package-module analysis, the user can interactively access equations for VLSI area, wiring capacitance for a given area, propagation delays, and power as well as display intermediate variables (such as average gate-delay for a given tile) and directly input parameters for commercial packaged chips.

The following describes models and improvements specific to Suspense. Appendix B provides greater detail, including the new and revised equations and a more detailed summary of Suspense validation.

General Tile Model

A tile is sub-section of a chip with a particular function or characteristic, such as memory, microprocessor, or random logic. Suspense adds specific models form memory, and uses the SUSPENS die model for random logic, microprocessors, gate arrays, and supercomputers. A new general type, parallel processing element, uses the microprocessor parameters with increased effective logic depth (based on regression to numerous examples). Suspense improves the circuit delay and signal propagation calculations of SUSPENS and adjusts several of the VLSI parameters.

RAM and Register Tile Model

The RAM model covers SRAM, DRAM, and registers, and forms the basis of the CAM model. This model uses characteristics particular to memory to improve model accuracy. Area per memory cell is a user input, and support circuitry (such as address decoding) is based on the general tile model, with adjustments to gate characteristics. The memory can be divided into sub-arrays (reducing access times and power, at some increase in area as is done in commercial chips), and pin count is based on the memory size and organization. Delays are based on support circuitry logic depth and the known geometry of row, column, and other long-lines. Capacitance and in turn power are also based on the known geometry and the reduced bit-line voltage swing for during read in the case of DRAM.

CAM Tile Model

The bit-cell CAM tile model includes numerous additions and several modifications to the RAM tile model. Although the MISC architecture does not include bit-cell CAM, the model is important in determining cost-performance for comparison to the chosen alternatives. Memory cell areas are again a user input. In addition to adjusting memory array characteristics, such as line capacitances and the number of lines, the CAM model adds match flags and a multiple-match resolver tree. Times for different operations, such as load registers, match, and write are output. Resolver time is calculated based on the logic delay of each stage and estimated line length for each level of the tree.

Die Model

The Suspense die model combines tiles and input-output pads to calculate characteristics of unpackaged die. The user specifies several die by inputting the tiles to be assigned to a given die, and the number of each tile on the critical path. Area and power are the sum of the tiles, plus allowances for interconnect between the tiles. Delay calculations allow for clock and control signal skew across the die. If a die contains more than one tile, pincount is based on Rent's coefficients using a user-input die type. The user may optionally select low-powered drivers, and choose between perimeter, fine-pitch perimeter, and array pad placement. Unlike SUSPENS, Suspense adds pad area, and calculates the minimum area imposed by the inner pad-frame dimension and minimum array pad spacing. Suspense includes output driver delay and power at the die rather than the module level. Suspense also includes a cost-metric, estimating cost of the silicon area based on regressions to custom fabrication pricing, adjusted for yield.

Packaging Model

Suspense introduces a model for packaged die. The model uses typical pin-pitch and package form-factors for dual-inline-package, quad-flat-pack, ball-grid-array, C4 (and unpackaged), and pin-grid-array to determine package area. Propagation delay for the packaging is calculated, as is output driver power. Off-chip communication can be at a specified voltage swing. The resulting chip power-density is used to estimate the type cooling required (such as natural convection, fan, or liquid cooling) and the cost penalty. The cost model also adds a typical cost per pin for each package type to approximate the added packaging cost. The model includes calculations for stacked-die (3-D packaging) based on data in [Lei90, VL90].

Module Model

The Suspense module model uses the SUSPENS equations for module area, speed, and power, modified by the improved propagation delay calculation. Unlike SUSPENS, Suspense allows a mix of package sizes and specific package-interconnect topologies. Module-level interconnect is based on the weighted average of package areas and pin counts. Topologies include nearest-neighbor, random (Rent's rule), and bus. Normally, the bus communication required for instruction broadcast dominates signal-delay for parallel arrays. The module cost-metric is based on regressions to current printed circuit board and projected multi-chip module manufacturing costs, and approximates substrate, socket and module I/O, assembly, test, rework, and power supply costs.

3.3 2MISC Functional Simulator

The MISC architectural simulator, 2MISC, is a high-level functional simulator developed to speed and ease evaluation of architectural and algorithmic alternatives and to project run times. 2MISC runs on serial workstations. Overhead is kept to a minimum to provide fast simulations, support larger "applications," and encourage exploring more alternatives. Although investigating MISC programming methodology is not a primary goal of this dissertation, efforts to make "user programs" (simulator input) reasonably close to SEGUL pseudocode syntax allowed some investigation of programming methods and permitted a nearly direct manual translation from pseudocode to simulator input. 2MISC can be configured to simulate other machines as well as architectural alternatives for MISC.

3.3.1 Output

2MISC provides the following optional profiles, which can be switched off to reduce simulation time.

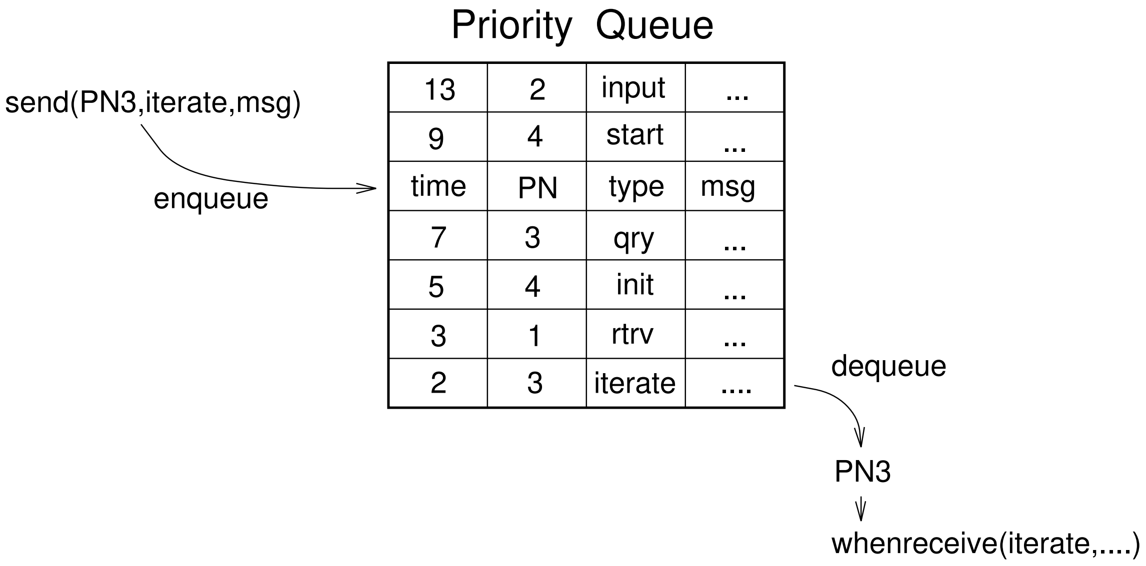
- MIMD communication (number of messages and total bytes sent by each PN and/or a file containing each message sent)
- SIMD communication (file containing each local communication)
- Number of calls to each SIMD macro function for each PN
- Memory references (by address bins) for each PN
- Estimates of array and supervisor processing times, along with message send time and receive wait time.

3.3.2 Simulation Methodology

2MISC is a discrete event driven simulator, with events being inter-PN messages. Multiple PNs are simulated using a priority queue, as shown in Figure 3.1. A send enqueues the message with an estimated arrival time and a receive dequeues the PN's message with the earliest estimated arrival time. Round-robin routine execution is not suitable as a PN later in the sequence could send a message to a PN already called that has an arrival time earlier than those previously enqueued. Further, a PN with short procedures could require several rounds to catch up. Instead, messages are dequeued in order of arrival time, activating the appropriate PN routine. A receive within an already executing routine poses two problems. First, it would require a different receive function than that which initiates a routine. Second, there won't be a message enqueued for reception until another PN has a chance to execute. To avoid complicating the simulator, 2MISC requires that each receive start a new routine. This introduces only minor complications (primarily adding global variables) in coding algorithms that receive data from other PNs while inside a loop or conditional, and no difficulties otherwise. Messages are structures to allow for compound data objects, with memory allocated on posting to the queue and deallocated when dequeued and copied to a PN variable. Arrival time is calculated as a function based on the number of PNs and a latency per router. Local communication is calculated by the number of array cycles for the type of communication pattern (nearest neighbor, general permutation, etc.). All synchronization and procedure activation is accomplished via user-defined messages. Host code is handled the same as a PN for communication uniformity.

SIMD array programming is by function calls equivalent to SEGUL operations. These functions compute output in an efficient manner for a uniprocessor, and log their call and memory references to profile files, without any behavioral simulation. Only those functions needed for the examples run are implemented. The simulator contains tables of machine cycles derived from manually coding the the high-level operations and uses these tables to calculate execution cycles and elapsed time. The tables permit configuring 2MISC to different architectural options and even completely different architectures. SIMD memory is handled as individual C structure declarations to reduce simulation overhead. The structures include the parallel data itself, size, and a simulated location in PE memory. Profiling memory referencing allows SIMD collections to span one or more bins.

Supervisor code is normal C code. Supervisor time is based on the simulator platform's real-time clock, adjusted by the ratio of the simulation platform's sustained MIPS to that projected for the MISC supervisor.



13	2	input	...
9	4	start	...
time	PN	type	msg
7	3	qry	...
5	4	init	...
3	1	rtrv	...
2	3	iterate	....

Figure 3.1: 2MISC MIMD Simulation

3.4 Summary

This chapter has summarized the tools developed to assist in designing and evaluating the MISC architecture. Suspense extends a prior early analysis tool to more flexibly and more accurately project the implementation ramifications of architectural and technology choices for parallel processors. Improvements include adding new models for memory, packaging, and cost metrics. Suspense also improves some of the calculations from its predecessor; accuracies are typically ± 20 to 3-%. Suspense should prove useful in other research and design projects. 2MISC is an event-driven functional simulator used to generate instruction profiles and project execution times for MISC and other architectures. These tools, together with the programming model presented in Chapter 4, support an innovative research methodology integrating algorithm, architecture, and implementation trade-offs.

4. The Associative Programming Model

Associative processing operations can serve as "knowledge processing primitives" for building efficient knowledge processing systems. Algorithmic efficiency must not be lost, however, in going to low-level massive parallelism. If processing elements go unused or do unnecessary work, the system loses cost-performance and may not be justifiable for many applications. The research proposes executing multiple levels of parallelism in knowledge processing codes. More specifically, this dissertation proposes data-parallel operations on collections of objects emphasizing access by content for fine-grain parallelism, while coarse-grain control (MIMD) parallelism assists in achieving parallelism, good processor utilization, control flexibility, and maximum parallelism. The combination may serve as the basis for building advanced retrieval, classification, machine learning, inferencing, and other systems.

Developing knowledge processing applications to exploit multiple levels of parallelism would be a daunting task without appropriate programming constructs. As discussed in Section 2.4, parallelizations of pre-existing knowledge processing languages typically support only one level (and often only one type) of parallelism, whereas existing multi-level parallel and content-addressable parallel processing languages do not directly support the complex pattern-matching and search operations crucial to knowledge processing. The MISC associative programming model combines features of multi-level collection oriented and associative languages, adding emphasis and direct support for complex and dynamic data objects.

This chapter establishes associative processing as a viable means of expressing massively parallel knowledge processing and introduces the MISC extensions to prior associative and collection-oriented programming models. These extensions emphasize matching and manipulating dynamic, complex data. Section 4.2 presents the specific pseudocoding notation, SEGUL, used in the remainder of this dissertation. SEGUL has proven useful in coding knowledge processing algorithms for simulation on the MISC architecture, but is not a complete and formal language. It remains an evolving notation and was developed only to the degree needed to meet the needs of the MISC research methodology. The final section highlights how the MISC associative model can be implemented within the framework of SIMD architectures

4.1 AI and Associative Processing

In this section we present arguments that the underlying operations of AI indicate that associative processing is a viable model for expressing AI algorithms. Many forms of knowledge processing can be viewed as having pattern matching as the single mechanism for accessing components of compound data structures, selecting procedures, and passing parameters [BF82]. This view is shared by Moldovan and the SNAP group (Section 2.3.4), who hold matching as the inner loop in reasoning, describing it as trivial for single keys but difficult for structured relations [M^{+}92]. Similarly, Fahlman characterizes most AI programs as a search in a large database followed by minimal processing, suggesting a massively parallel approach for AI primitives [Fah85]. Fahlman's NETL semantic network model was among the first proposed massively parallel AI systems. Kogge emphasizes matching, substitution, sets and relations, and Levinson builds a case for inference and intelligent behavior based on pattern associativity [Kog91, Lev92b, Lev91]. These views lend credence to the importance of an associative approach as a foundation for knowledge processing.

Associative processing can clearly serve as a basis of knowledge processing systems. Others have shown that associative processors and CAM-based systems can support functional programming, logic programming, production systems, knowledge bases, natural language systems, pattern and symbol recognition, semantic networks, and neural networks [Kog91, K^+89, O^+89, M^+90, Sto91, YZ90, WS89]. Content-addressing operations can indirectly support higher-level knowledge processing functions, including matching primitives in AI languages such as Prolog, and LISP, offering vital speed and coding improvements. Specific functions supported include backtracking, clause filtering, conflict resolution, demand driven reduction, dereferencing, garbage collection, indirection, inference, matching, the Rete algorithm, the A* search algorithm, unification, and variable binding [K^+89, Kog91, M^+90, NGC89, O^+89, RG91]. One system showed a 50 times speedup for the Rete algorithm over a system without CAM and 10 times over other Rete parallelizations, providing an important advantage for production system applications [Kog91]. Applications that especially benefit from content-addressing include natural language processing, neural networks, pattern and symbolic recognition, and semantic networks [Kog91, M^+90, Sto91, WL89, YZ90].

Unfortunately, CAM hardware implementations require uniform-length fixed-location data and exhibit poor utilization of hardware resources for non-matching operations. SIMD architectures have also historically been restricted to well-structured problems that feature regular patterns of control and repetitive operations on uniformly-structured data, suffering from poor hardware utilization when these restrictions are violated. MIMD architectures do not have these restrictions, but incur the penalty of much larger processors and higher synchronization costs. This dissertation develops means of extending the capabilities and performance of SIMD and content-addressing for knowledge processing, efficiently processing structured relational data with a scalable architecture that mitigates the difficult characteristics of knowledge processing codes and exploits the beneficial ones.

Having shown the viability of associative processing and the need for extending parallel programming models, the remainder of this section examines commonalities between several knowledge processing paradigms that can be implemented as associative and data-parallel operations, followed by a more specific discussion of these operations. The remainder of this chapter presents the syntax and processor requirements of the resulting SEGUL pseudocode notation. Chapter 5 analyzes the target applications, including their coding in SEGUL. After Chapter 7 discusses the resulting MISC architecture in detail, Chapter 8 further supports the claim of associative processing as a practical means of achieving significant speedups for massively parallel knowledge processing algorithms.

4.1.1 The Example Applications and AI

In finding commonalities between the target applications and the even broader range of other knowledge processing paradigms and algorithms, this dissertation can at best scratch the surface and discuss a few generalizations. It is nevertheless important to summarize the conceptual framework from which the MISC associative processing model is drawn, however skeletal. Section 2.5 and the above CAM survey characterize most work in providing massively-parallel support for AI applications. Here we extract a few principles, including emphasis on the knowledge side of the *knowledge-search tradeoff*.

The Pillars of AI

Figure 4.1 presents the "pillars" of symbolic AI. *Search*, particularly backtracking state-space search, is one of the unifying principles of problem solving and inference. Knowledge base *retrieval* is shown separately, the distinction being that search generates and/or gathers information without regard to structure whereas retrieval utilizes structure to access pre-existing knowledge [Lev94b]. Knowledge *representation* (KR) is a central component of knowledge processing; a representation must enable efficient access and operations in addition to just containing the key data. Applications often receive significant speedup with an alternative KR, and in learning the KR can make the difference between practicality and impossibility. *Learning* can be viewed as finding structure and order in raw data. This includes *subsymbolic* methods such as neural networks, pattern recognition, and classifiers, as well as *symbolic*-level machine learning. These pillars form the basis of reasoning, *inference*, problem-solving, and other high-level symbol-processing. This dissertation proposes relational structures (graphs) and associative memories as the common means of implementing the four pillars, with a basis in mathematical structures and operations.

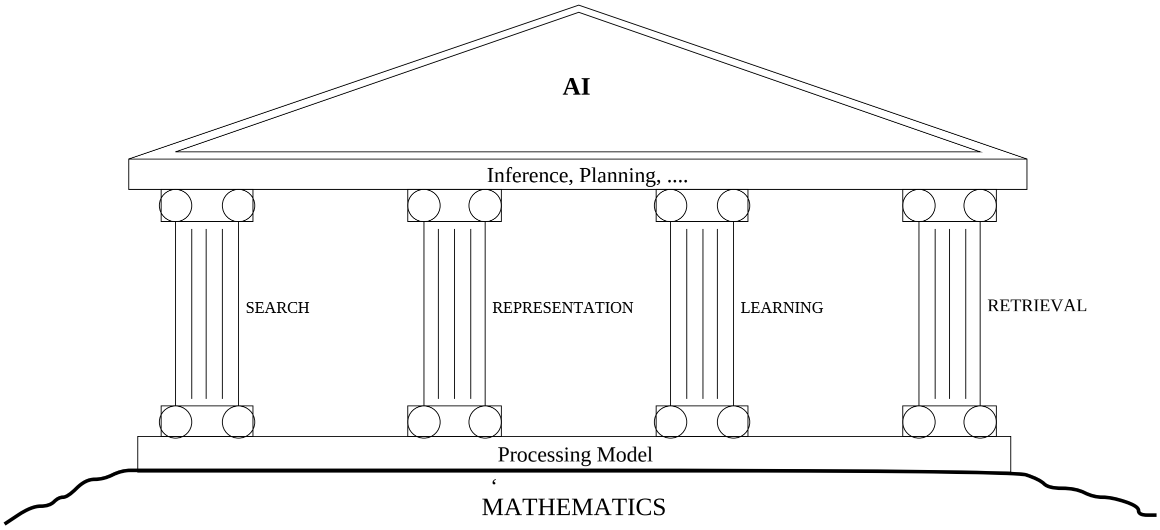


Figure 4.1: The Pillars of AI Processing

Machine learning and AI in its broadest sense can be viewed as converting *extrinsically* defined data (raw examples, explicitly defined facts) into *intrinsic* definitions (induced rules, relations) that are then further "compiled" into higher-order abstractions. The dissertation research relating to learning emphasizes SDM and closely related k-nearest neighbors methods of learning in the form of associative memories (map learning), classifiers, etc. Here the mathematical foundations are in extrinsic maps, attribute matching, and distance metrics. The other two example applications, semantic networks and graph retrieval can incorporate learning by adjusting values ("weights") associated with the nodes and graphs.

The MISC Retrieval Approach to AI

Figure 4.2 illustrates a proposed relationship between classical parallel AI and massively parallel AI. In the latter, retrieved knowledge forms the basis of AI approaches such as memory- and case-based reasoning. Increasing the sophistication of retrieval beyond address- or pointer-based access and simple matching, as well as greatly increasing the size of the knowledge base, greatly improves performance in terms speed and quality of results. Representation is important for speed and efficiency at all stages. The ultimate goal is to cover the full spectrum of massively parallel and classical AI.[1] In this dissertation the architectural development and associative model focus on retrieval, with some coverage of learning classification and inference.

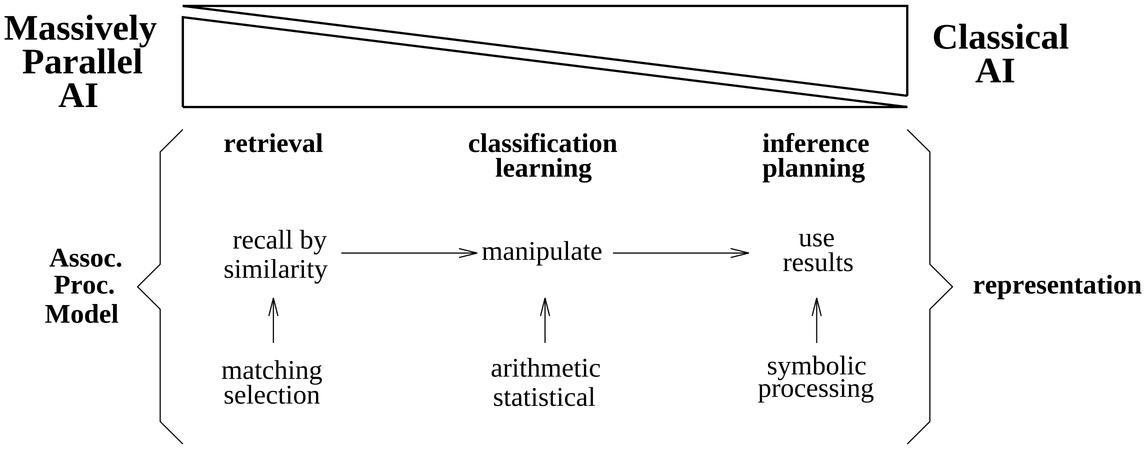


Figure 4.2: Massively Parallel and Classical AI

The MISC research emphasizes relational structures in "building" the pillars of AI for several reasons. Relational structures directly represent logical, structural, and causal relationships among objects and attributes, "pre-computing" connections and supporting operations such as generalization at a higher level of abstraction. Graph retrieval from a partial order is itself classification [ELR95]. Recent work by Levinson and Karplus demonstrates that state-space search can be represented and efficiently executed within this framework by representing the current state, the goal state, and the operators as graphs [Lev93, LK93]. Attribute inheritance and concept subsumption can be supported by a partial order, or more precisely in this context the *transitive closure* of the subsumption relations. Inference can also be viewed as the transitive closure over recursive relations. A relational structure KB can thus form the basis of classification, inferencing, reasoning, and planning systems. By replacing Prolog's terms with recursive labeled hypergraphs (CGs) such a KB becomes a generalization of Prolog and provides sub-linear indexing for first order logic formulas, independent of the knowledge representation or domain [ELR95]. The generalization hierarchy (partial order) of CGs can represent if-then productions with pointers to one another, with

---

[1] As part of this long-range goal, the MISC associative arrays are intended primarily for massively parallel algorithms while the supervisor (or network of multiple supervisors) serves serial or coarse-grain parallel algorithms.

the predecessor of a graph indicating all its possible productions [Lev94b]. Additionally, weights associated with graphs can form the basis of learning, as in Morph [AL93].

Semantic networks support several types of inference through marker or message propagation [Fah85, M^+92]. Semantic networks and relational structure processing can also be combined in supporting AI. Recent work by Levinson defines a "universal data structure" (UDS) combining the representations and benefits of relational data bases, semantic networks, conceptual graphs, the Rete algorithm, neural networks, and case based reasoning for a finite set of domain objects [Lev94b]. In UDS all data is represented by sets of tuples, and a partial order (poset) is maintained over data objects. It can be viewed as a three-tiered poset consisting of a node hierarchy, a relation hierarchy, a CG hierarchy, and additional pointers between objects at the various levels. Here the CG representation is organized by relations (edges) rather than nodes (concepts), reducing the amount of work required in subgraph isomorphism testing. Retrieval can be by marker passing, relation processing (RDB), or CG subsumption depending on the type of query. The method supports state space search, pattern recognition, machine learning, and inference through incremental updates of the hierarchy, further improving efficiency and speed.

SDM and related methodologies feature learning, including generalization from examples. Associative recall by similarity by either syntactic or semantic distance is a key component. In low-level processing of lists of attributes, a continuity bias in the input parameter space forms the basis of associative memory, nearest-neighbors classifiers, and other neural network-like behaviors. In semantic networks distances are often based on the number of links markers pass through, and nodes separated by a small number of links of particular types are considered similar. At higher levels, structural similarities including isomorphisms are important in reasoning by analogy and retrieving similar prior experiences for case-based reasoning. Such computationally derived distances are syntactic distances. By the Theorem of the Ugly Duckling, differences based on collections of attributes are insufficient [Wat85]. It is necessary to establish the relevance of particular features to an output such as a classification, and the criticality of the result to system goals. Semantic distances are thereby built upon syntactic distances, often by learning.

4.1.2 Operations for Supporting Knowledge Processing

From the above perspective the example algorithms in some way represent all the pillars of AI in Figure 4.1. These algorithms can be based on associative and SIMD processing functions such as content-addressing, set operations, maps, distance metrics, transitive closure, lattice encodings, and subgraph isomorphism and similar operations on relational structures.

Operation Classes

The resulting associative functions can be categorized as follows.

- **Matching operations** include equivalence, masked, and magnitude comparisons of elementary data types such as Boolean attributes, integers, and characters, and are the most basic operations of searching. More complex matching macro-operations such as substring search are also important.

- **Distance metrics** (computed syntactic distance) are crucial for nearest-neighbor functions and in supporting approximate matching as well as for building higher-level semantic distance measures. After thresholding, a distance operation becomes a matching predicate.
- **Partitioning** divides data objects into equivalence classes based on matching and distances, selecting sets of active objects for subsequent processing. In conjunction with distance metrics, partitioning can be used to extend basic match predicates into a variety of more sophisticated searches.
- **General data-parallel operations** include readout of search responses, summation, thresholding, and other arithmetic and logical operations.

These categories are the basis of an associative programming style similar to the Gamma model and the match-act cycles of production systems, or at a higher level, retrieve-infer. *Match* operations are used to form *sets* of objects followed by *data-parallel* operations on the sets so produced. The SDM algorithm epitomizes this style. As Banatre and Metayer point out, however, this "pure" associative model cannot support backtracking and other important algorithms [BM93]. This necessitates the ability to deal with global information and non-match-based control structures.

Nested Collections and Relational Structures

The associative model emphasizes collections such as sets and tuples. The concept of a set as "the ability to regard a collection of objects as a single entity" [Dev93] is fundamental to both abstraction in knowledge processing (problem conceptualization) and the inherent parallelism in collection-oriented programming languages (problem implementation). The model uses *nested collections*, such as sets of tuples, to represent relational structures. Allen *et al.* define a relational description of an object as "a set of 'primitives' each having its own attribute description and a set of named relations" [A^+93b], which coincides with the definition of a labeled graph.

The MISC associative model generalizes relational descriptions from graphs to nested hypergraphs. In a graph a 2-tuple represents a directed edge and a 2-element set an undirected edge; in a hypergraph an edge is an n-tuple or n-element set. As shown in Figure 4.3, in a nested graph any node can itself be a graph, recursively. A hyperedge can also be described as a subset of nodes. Graphs are an intuitive representation of complex objects, and can exploit ordering information in retrieval and other processing [ELR95].

Nested collections flexibly represent complex relational structures, as well as more conventional data structures such arrays and lists, using standard mathematical notation. Semantic networks are formally directed, labeled hypergraphs [Lev94b]. Both a directed hypergraph and a RDB relation instance correspond to a set of n-tuples. Peirce's existential graphs are hypergraphs in which an n-ary relation is a directed hyperedge of n ordered nodes (a label can be represented by an additional node) and a context is an undirected hyperedge. Existential graphs have the full expressive power of first-order predicate logic, indicating the expressiveness and generality of the nested collection notation. LISP lists are also notated as nested tuples. Partial orders further illustrate the expressiveness of nested collections. A partial order is itself a graph (the transitive kernel of the constituent relations) and nodes can themselves be relational structures such as nested hypergraphs. Applying *nested* *parallelism* to these objects as in the Paralation model maximizes parallelism, improves processing element utilization, and reduces execution time. Indeed, the term "paralation" is a contraction of "parallel relations" [Ble90].

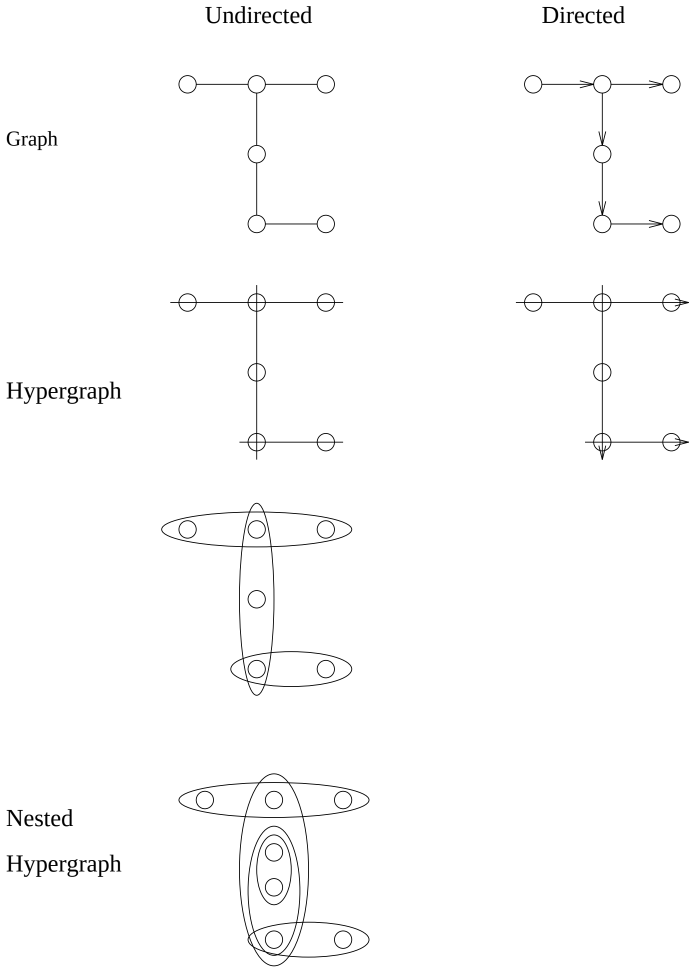


Figure 4.3: Graphs and Hypergraphs

4.2 SEGUL Notation and Terminology

This section presents the details of the MISC programming model through the pseudocode notation SEGUL, developed as part of the MISC research. The MISC associative programming model brings together issues of programmability, algorithm development, and hardware requirements and design. The primary purpose of the model is to provide a language and architecture-independent pseudocode for target algorithms; the pseudocoded algorithms are then profiled and analyzed to develop and evaluate architectural features of the MISC Machine. This methodology minimizes the "dusty deck" syndrome of using existing codes, written in existing languages for existing machines to design new architectures. Although an actual MISC Machine is likely to be programmed in an extension of an established programming language, the architecture is based on fundamental characteristics of knowledge processing methodologies with minimal bias toward prior hardware and software solutions. The resulting pseudocode notation proved to be an elegant and intuitive way of implicitly expressing the lower levels of parallelism while providing a straightforward method for explicitly specifying coarse-grain parallelism.

The programming model and notation informally draws from a number of collection oriented and parallel programming languages (the paralation model [Ble90], SETL [SDDS86], Gamma [BM93], and Linda [CG89]) and is called SEGUL as a mutilated acronym of these.[2] The notation is primarily that of naive set theory [Dev93], thus incorporating a widely understood formalism and resulting in many similarities with the set- and tuple-oriented language SETL. Low-level parallelism is implicit by treating manipulations of set-like collections as a single operation, and multi-level parallelism is implicit in nested collections as in the paralation model. Coarse-grained parallelism, including global communication, synchronization, and task activation on the MISC architecture is handled by matching operations on tuple-messages as in Linda. The associative model includes a programming style similar to the match-act cycles of production systems and the multiset-transformation model of Gamma. This combination of methodologies meets the goal of supporting dynamic complex data objects and exploiting multiple levels of parallelism in a programmer-friendly framework with a relatively direct mapping to hardware. In emphasizing associative processing, particularly nested-collection matching, SEGUL makes significant advances over prior methodologies.

SEGUL provides an application-programmer-level specification of the MISC architecture. The specific notation was developed *ad hoc* while pseudocoding the core algorithms of the target applications, and is documented here and somewhat extended to standardize and annotate the pseudocodes in this dissertation.

4.2.1 Data Objects

In keeping with the formal distinction between an element and a set of one element, a SEGUL data object can be either atomic (a single value) or a *collection* which generically refers to sets, tuples, multisets, and ordered sets. Collections can be nested, that is a member can be either atomic or a collection. Collections can be either *homogeneous* (all members of the same type) or *heterogeneous* (members of differing types). The standard set notation {} is used to indicate unordered collections whereas SETL's [] is used to indicate ordered collections, to avoid confusion with parenthesis used in precedence and function parameters. An unordered collection can be either a set or a multiset. An ordered collection can be either a tuple, where position indicates an attribute or variable "slot", or a specific sequence of elements.

Atomic (Scalar Primitives)

An atomic value can be Boolean (single bit), code (string of bits, typically used to indicate a unique identifier such as an ASCII character or pointer), byte (8-bit integer),

---

[2] The 'p' for *paralation* is "silent" despite its importance in the machine-level representation of SEGUL data objects.

integer, or floating point number. The latter types could be defined in terms of a tuple of bits (or tuple of tuples in the case of a floating point number), but are predefined for readability, ease of use, and to specify and restrict the types of operations on the value. A predefined **subsumptioncode** supports subsumption and lattice code operations. Atomic variables are declared as in Figure 4.4. They can be initialized by including an assignment (indicated by $\leftarrow$) in the declaration, and several declarations can be placed on a single line separated by commas.

```
boolean active
integer m
integer n ← 4096, alvin, bp3
code(16) chx, indexPtr
float radius
```

Figure 4.4: Scalar Declaration

Nested Collections

Collections are declared as **unique** or **nonunique**, and ordered (**[]**) or unordered ({}). A multiset is **nonunique** {}, a set is **unique** {}, a tuple is **nonunique** **[]**, and a sequence or ordered set is **unique** **[]**. Declarations have several options. Declarations are of the form **datatype name number-of-elements** (the number of elements defaults to 1, giving the scalar declarations in Table 4.4). If the collection is homogeneous, the element type can be declared inside the collection delimiters ({} or **[]**). If the collection is of fixed length, this can also be declared inside the delimiters by a constant or previously declared and initialized integer scalar. Nested collections can be defined with nested declarations if the structure is static. Collections can be initialized *extrinsically* by listing the members inside the delimiters, or *intrinsically* using the set convention of a vertical bar followed by the defining property. In the latter case, a colon is used after the element type to indicate that the following variable is a local variable for the defining property. Standard set notation can be used to specify that elements are to be from a specified set. Examples are given in Figure 4.5. The last two examples generate a 1024-bit Boolean vector initialized to random values and a 64-element integer vector initialized to 0.

Nested collections are normally flattened and uniformly distributed across processors as in the paralation model. The keyword **type** declares a unit, all of which is assigned to a single processor.

4.2.2 Operations

Control flow and operations involving only scalars are similar to those in C, with blocks indicated by indentation rather than by enclosure within braces.

```
nonunique bagt {}, bagu {integer}
unique someset{code(16)}, someotherset {}
nonunique statvec [float 128], statvec2 [n]
unique wilma []
nonunique 2-d_vector [ int 8 nonunique [int 8] ]
unique firstten ← {int 2, 1, 3, 4, 5, 6, 7, 8, 9, 10}
unique onetoten ← [int: x|1 <= x <= 10]
nonunique paddr ← [bit:1024 x|x = random01()]
nonunique myvec ← [int:64 x|x = 0]
unique G ← {[x,y] | x ∈ A, y ∈ B}
```

Figure 4.5: Example Collection Declarations

"Matrix" Operations

Numeric and logical operations involving collections are based on prior vector and data-parallel models, particularly Blelloch's Paralation [Ble90]. Most correspond to the element-wise "dot" operations of linear algebra. In the following examples the explicit collections can be replaced by the variable name of a collection, and operations can be numerical or logical.

An *expansion* operation applies the operator and scalar to each element of the collection and assigns the result to a collection:

$$A \leftarrow [2,9,4] + 3 \text{ assigns } \{5,12,7\} \text{ to A.}$$

A *reduction* applies the operator over all members of the collection and assigns the result to a scalar:

$$a \leftarrow +[2,9,4] \text{ assigns } 15 \text{ to a.}$$

An *Elementwise* operation between two collections applies the operator to corresponding pairs of elements and assigns the result to a collection:

$$A \leftarrow [2,9,4] + [3,6,8] \text{ assigns } \{5,15,12\} \text{ to A.}$$

The over-bar notation $\overline{A}$ flips the values of Boolean collections and codes. Relational operators such as $=$ and $\leq$ return a Boolean collection.

The operator $\times$ is *matrix* multiplication and A^T indicates matrix transpose. These operations apply only to ordered collections of uniform size.

Element pairing is arbitrary if the arguments for elementwise operations are not ordered (sets or multisets). Operations on nested collections are handled as in the paralation model; numeric operations are handled as if the sub-collection groupings did not exist. Type mismatches, including those between heterogeneous collections, are errors.

Set and Tuple Operations

Special operations are included for collections, and are based primarily on set operations. The notations for insertions and deletions from unordered collections are

$$X \leftarrow X + \{y\}$$

$$X \leftarrow X - \{Y\}$$

where the element y (which can be either atomic or a collection) is enclosed in curly braces to distinguish the notation from an elementwise operation, reduction, or expansion. For ordered collections, an insertion or deletion can specify which element to add or remove by position. If the position is not specified in a deletion, the first occurrence of the element is removed. If the position is not specified in an insertion, the element is appended to the end (concatenation if the element is itself a collection). The element may be omitted in a deletion, indicating that the element at the given position is to be removed. Position can also be specified by a constant, an integer scalar, **first** (synonymous with 0), or **last**. Figure 4.6 gives examples.

$$\begin{array}{l} X \leftarrow X + [2; y] \\ X \leftarrow X + [; y] \\ X \leftarrow X - [0;] \\ X \leftarrow X - [; y] \\ X \leftarrow X - [last;] \end{array}$$

Figure 4.6: Ordered Collection Insertions and Deletions

It is possible to select an element from an ordered collection, specified by position:

$$y \leftarrow X(3)$$

$$Y \leftarrow X(last).$$

Nested ordered collections are accessed similarly to arrays

$$y \leftarrow C(2, 1, a)$$

is equivalent to `y = C[1,2,a]` in the C language. The position can be specified as with insertion and deletions. Unless the type of the element at the specified position is known, the left-hand side must be a heterogeneous collection if the right-hand side is a heterogeneous or nested collection to avoid any type conflict.

The iterator **foreach** steps through each element of a collection (in arbitrary order if the collection is unordered), executing the body of the loop with the selected element assigned to the loop variable. Again, the left-hand side must be a heterogeneous collection if the right-hand side is a heterogeneous or nested collection. **Foreach** is included for convenience; it could be built from flow-control statements and selection.

```
foreach y in X
        /* loop body */
```

Collection operations include determining the size of a collection, which is taken as the number of elements in the outermost nesting:

$$sizex \leftarrow |X|.$$

Membership testing is the same as in set notation:

$$b \leftarrow y \in X,$$

where the LHS is assigned true (1) if y is in X, false (0) otherwise. The membership test can be used directly in a conditional, without the assignment. The set operators union, intersection, and set-difference can be applied to nested collections:

$$\begin{gathered} Z \leftarrow X \cup Y \\ X \leftarrow Y \cap X \\ Z \leftarrow X \backslash Y. \end{gathered}$$

Collection comparisons are an important addition to previous collection-oriented models. SEGUL generalizes the subset operator, $\subseteq$, to subsumption (generalization of), and includes equivalence, $\equiv$, defined as $(A \subseteq B) \wedge (B \subseteq A)$ for sets and all corresponding elements matching for tuples. For simple (unnested) sets, $\subseteq$ is the formal subset operator; for multisets the LHS must have no more occurrences of a given element than the RHS. For simple tuples, the default subsumption operator is *subsequence*; ordered sets are treated as tuples which happen to not have duplicated elements. The user can specify other operations, such as *substring* and $|A| \leq |B|$, for a specific collection. Comparing nested collections requires more complex pattern matching; each collection's nesting induces a tree structure to be compared with the other's. The recursive comparison bottoms out at subsumption tests of simple collections. This form of subgraph isomorphism test has polynomial time since both structures are trees [GJ79].

In addition to exact matching, subsumption operators can be specified for subsumptioncode atomic elements. SEGUL uses $\preceq$ to indicate atomic subsumption to maintain the distinction between an element and a set of one element. Other subsumptioncode operations include greatest lower bound and least upper bound. For fully ordered elements (Boolean, integer, floating point, and some codes depending on use) subsumption can be defined intrinsically by a comparison operator. Otherwise subsumption and equivalence sets must be extrinsically defined.

Distance

The distance between two collections is an important part of the associative processing model. SEGUL includes a distance operator for unnested collections. Distances for nested collections are built from SEGUL primitives by the user, for reasons of application specificity and efficiency.

For ordered collections, SEGUL uses the Minkowski distance metric (it is also referred to as the *ℓ-norms*) due to its generality, frequent usage, and efficient parallel computation. It is a true metric in the mathematical sense; the distance is always positive, symmetric, and obeys the triangle inequality, and the distance between an object and itself is zero. The metric's SEGUL notation is $||X,Y||^{\lambda}$ and its definition is:

$$||X,Y||^{\lambda} = \ell_{\lambda}(X,Y) = \rho_M(X,Y) = (\sum_{i=1}^{n} |X_i - Y_i|^{\lambda})^{1/\lambda}$$

where $\lambda \in \mathbf{R}$, and $n = |X| = |Y|$. For Booleans the difference operator $-$ is $\oplus$, exclusive-or. In practice λ is usually an integer, and the most common norms are ℓ_0, ℓ_1, ℓ_2, and ℓ_∞. The ℓ_2 norm is Euclidean distance, and the ℓ_1 norm on Boolean vectors is Hamming distance. Without the restriction to Booleans, ℓ_1 is the Manhattan distance (also called city-block or taxicab distance). The ℓ_∞ norm is defined as $\lim_{\lambda \to \infty} \parallel X, Y \parallel^\lambda$ and is also called the square metric. Note that ℓ_∞ is $\max(x, y)$ and ℓ_0 is $\min(x, y)$. For atomic Boolean, integer, and real values the SEGUL metric is simply $|x - y|$; for codes the atomic distance is $||x, y||^1$.

For unordered collections, SEGUL uses the set-difference measure $|X \backslash Y| + |Y \backslash X|$. This is interpreted as the number of elements differing between the two sets, and is equivalent to $|(X \cup Y) \backslash (X \cap Y)|$. the proof that this is also a true metric is straightforward. This metric also corresponds to Hamming distance when X and Y are boolean.

In combining or comparing distance measures taken on collections of different length, normalizing the distance is often desirable, and is denoted as $||X, Y||_n^\lambda$, where the n indicates that the ℓ-norm result is divided $|X \cup Y|$. For tuples (such as in a normalized Hamming distance), the norm is divided by the size of the tuple. The normalized distances may not be metrics.

Maps

Select-by-content is the basis of more general *content-addressed* retrieval, a crucial aspect of SEGUL and MISC. Extrinsically defined maps, represented as a set of 2-tuples (binary relations), are an important feature of SETL. In

$$Amap \leftarrow \{[x_1, y_1], [x_2, y_2], ...[x_n, y_n]\},$$

x_i is an element of the domain and y_i is an element of the range. This usage of the term map is not synonymous with a function; the extrinsic definition may omit elements of the domain, and the mapping to the range may not be unique. SEGUL further generalizes this to a set of n-tuples (n-ary relations) where the positions which correspond to domain and range depend on the select argument. Here the argument is an n-tuple with the special characters '_ 'and '?'. Any element value not preceded by a '?' is a domain element and any variable preceded by an '?' indicates a range element. The '_' character indicates a position not involved in this particular mapping ("don't care"). Thus

$$\begin{gathered} Amap \leftarrow \{[a_1, b_1, c_1, d_1, e_1]...[a_n, b_n, c_n, d_n, e_n]\} \\ R\{[y, x]\} \leftarrow Amap([7, ?x, \_, g, ?y]) \end{gathered}$$

is a mapping where the domain is $[a, d]$ and the range is $[e, b]$. In this example the left-hand side gets the values of e_i and b_i for each tuple $i \in Amap$ with $a_i = 7$ and $d_i = g'$. This operation corresponds to a masked (partial) match in CAM, as well as a relational database select and project instruction. In many cases it is desirable to mark those tuples matching the specified domain, without returning any values (such as in forming activation sets and equivalence classes). In this case the left-hand side is a Boolean tuple (of the same size as the 'map' set) and no range elements are specified:

$$R[boolean|Amap|] \leftarrow Amap([7, \_, \_, g, \_])$$

Communication Within Collections

Moving elements between and within collections[3] is handled by a partial-permutation assignment, expressed as a subscript on a collection.

$$B_J \leftarrow A_I$$

'permutes' the collection A according to the tuple of indices I, placing the result in B; $|A| = |I|$. The optional set J contains the indices of each element that receives a new value. Each element of I is an integer indicating the destination position for the corresponding value in A. A permutation can optionally include an operator for combining multiple elements sent to the same destination, a default to be supplied to any destination that does not receive a value from the source, and a Boolean select tuple where False indicates the corresponding element is not to be included in the destination. These are expressed as

$$A_{I,\oplus,D,S}$$

where $\oplus$ is an associative operator, D is the default tuple (if not specified, an element which does not receive a new value retains its prior value), S is the selection tuple indicating which elements are 'sending', and $|A| = |I| = |D| = |S|$. General permutations are not useful on unordered collections, simply changing one arbitrary ordering to another, but permutations with the options can perform useful transformations even on sets. Index tuples are predefined for common permutations. $A_{SL(1,d)}$ shifts elements left by one position with the right most element accepting the default d and $A_{SR(4,d)}$ shifts elements right by 4 positions filling the leftmost 4 elements with the default.

4.2.3 Communicating Processes

Fine-grain parallelism is implicit in the above operations on collections, and nesting can provide multiple low levels of parallelism. This does not capture all parallelism however, nor can a compiler automatically identify and efficiently implement all available parallelisms [Ble90]. SEGUL includes message passing as means of activating, synchronizing, and communicating data between parallel processes.[4] This has already been proven useful for coarse-grain user-defined parallelism in knowledge processing programs [A$^+$87]. The syntax and methodology is similar to that used in Linda, and is based on the associative tuple matching. A process is activated by *sending* a message tuple, whose first element is the name of the process and whose remaining tuples are the arguments: $send[procname, arg1, arg2, argn]$. Synchronization is achieved when the results are returned using a receive statement:

$$receive[procid, ?result1, ?result2, ?resultn].$$

The question marks indicate variable assignment; otherwise the tuple element must match for the receive to "activate." Other non-dependent operations can be performed while awaiting the results:

---

[3] This corresponds to hardware local communication within an array.

[4] In hardware, inter-processor communication can be either virtual within a given array or an actual message over the global network to another array.

```
send[mumblypeg, 4, 6]
      any other statements
receive['bigtoe',?fin]
```

A process is defined as follows. A procedure may update only local data objects, and unlike function calls local variables remain between invocations.

```
whenreceive['procname', ?arg1, ?arg2, ?arg3]:
      body of the procedure
      send[procid, result1, result2, resultn]
;
```

This methodology can be used to encapsulate operations in an object-oriented programming style, though without inheritance. At this stage of development, determining dependencies and linking calling send-receive pairs are the responsibility of the pseudocoder and the result of simultaneous (overlapped) operations on the same data object is undefined.

For convenience, inter-process communication can be expressed more implicitly using the following syntax, fully equivalent to the example immediately above.

```
procedure procname[arg1, arg2, arg3]:
      body of the procedure
      return[result1, result2, resultn]
;
```

4.2.4 Example Representations

Many common abstract data types (ADTs) are trivial in SEGUL. The ADT for sets is explicitly included in SEGUL, multi-dimensional arrays (matrices) are simply nested tuples of uniform size, and LISP lists can be directly notated as nested tuples. LIFO (stack) and FIFO queues can be implemented simply as in Figure 4.7. The stack operations can be encapsulated into an ADT as in Figure 4.8. Of course, pointer-based serial-processor implementations are just as fast. Advantages would accrue for 'articulated' queues where enques and deques occur at specified positions or as specified by content.

```
tuple Stack                          tuple Fifo
/* push */                           /* queue */
Stack ← Stack + [first;elm]          Fifo ← Fifo + [last;elm]
/* pop */                            /* dequeue */
elm ← Stack(first)                   Fifo ← Fifo(first)
stack ← stack - [first;]             Fifo ← Fifo - [first]
/* top */
elm ← Stack(first)
```

Figure 4.7: Example Stack and FIFO Queue

```
/* STACK ADT */
whenreceive['Stack', ?op, ?elm, ?Stack]:
      if op = 'push'
            Stack ← Stack + [first;elm]
      if op = 'pop'
            elm ← Stack(first)
            Stack ← Stack - [first;]
      if op = 'top'
            elm ← Stack(first)
      send[stackid, elem, stack]
;
```

Figure 4.8: Example “ADT”

A graph G can be represented in a manner identical to a standard definition of a graph

$$Graph \leftarrow \{[x, y] | x \in G \wedge y \in G \wedge connectedTo(x, y)\}$$

wherever the connectedTo can be defined either implicitly or as a selection from an explicit set of 2-tuples. More commonly, a graph would be defined explicitly as:

$$Graph \leftarrow \{[x_1, y_1], [x_2, y_2], ...[x_n, y_n]\}.$$

The latter representation also emphasizes the formal generalization of a graph as a set of binary relations, as well as suggesting the mapping implicit in a graph. Semantic nets and partial orders are graphs. Bringing out the mathematical structure of a data representation is an important benefit of SEGUL.

The use of tuples, as above, defines a directed graph whereas using the set doubleton $\{x, y\}$ defines an undirected graph. Labeled edges can be represented as $[l, x, y]$ or $[l, \{x, y\}]$, and hypergraphs can be represented by replacing the 2-tuples or set dubeltons by n-tuples or n-element sets. Other elements describing a set can also be included, such as the set of all vertices, the number of vertices or edges in the graph, the minimum cycle length, etc.

$$Graph \leftarrow \{verts, edges, mincyc, tree01, [x_1, y_1], [x_2, y_2], ...[x_n, y_n]\}$$

Applying the $\subseteq$ operator to two graphs represented in any of the above methods does *not* provide a subgraph-isomorphism test. Blelloch presents several graph algorithms in which graphs are represented as adjacency lists expressed as a nested tuple

$$[[y_{11}, y_{12}...], [y_{21}, y_{22}], ...]$$

with the source-node (x) identification implicit in the outer tuple's ordering [Ble90].

4.3 Processing Requirements

Blelloch has shown that the Paralation model with nested, ordered collections can be efficiently implemented on SIMD architectures. This section shows how *extension to unordered* *collections, set operations, and collection matching can also be efficiently implemented on a SIMD architecture.* Section 7.4.2 of Chapter 7 details how the ℓ_1 distance metric, code subsumption, and Boolean matrix multiplication are implemented on the MISC SIMD processors and Chapter 5 shows how associative memory, semantic networks, and relational structure matching are built from the SEGUL operations.

4.3.1 Numeric and Logic Operations on Nested Collections

Blelloch's dissertation demonstrated that the elementwise, scan, permute, insert, extract, expand (distribute), and length operations described above in terms of SEGUL's syntax can be efficiently executed with nested tuples on a generic SIMD array in conjunction with its controller or host [Ble90, BS90]. For these operations, data-parallelism is exploited simultaneously at all levels of the nesting and worst case slowdown for nested versus single-level collections is $O(n)$ where n is the depth of the nesting and is independent of the size of the collection and sub-collections whenever full data-parallelism is available. Slowdown for heterogeneous versus homogeneous nested collections is $O(a)$ where a is the number of atomic types. Constant factors are small in both cases. Reductions can be defined in terms of a permutation with a combining operator and all elements sent to the same destination, but are included separately in SEGUL for convenience and clarity.

Unordered Collections

Extending Blelloch's paralation methodologies to unordered collections, including nested combinations of ordered and unordered collections, does not introduce any additional processing requirements. This is a result of the arbitrary ordering provided by processing element and/or memory assignment. Only the description of the operations is modified, as summarized in Table 4.1.

Operation Type	Implications for Unordered
Reductions	Operation is associative, order does not affect result.
Scan	Operation is associative, ordering in result is arbitrary.
Expansion	All elements receive the same value, ordering irrelevant.
Elementwise	Parings and result order are arbitrary.
Permute	Position changes are irrelevant. Pairings of combined values and the position of the result are arbitrary. Position of defaults are arbitrary.
Insert	At arbitrary position.
Delete	Select by value rather than position. This requires matching nested collections.
Extract	Not relevant.
Length	Ordering irrelevant.

Table 4.1: Applying Ordered Collection Operations to Unordered

Dynamic Collections

The paralation model handles dynamic collections by generating a new collection of the appropriate size with distribute, then using permute and select to move the values into the new collection. In many cases it is preferable to use an *associatively-linked* representation similar to that proposed for Foster for list and string processing [Fos76]. Elements of both ordered and unordered collections are assigned to arbitrary, non-contiguous processing elements along with fields to indicate collection membership, ordinality and/or cardinality (formally, each atomic value becomes a tuple). Data movement is not required, so that insertions and deletions can be performed in constant time. Garbage collection can be performed in constant time by including a used/unused flag. These advantages are not without cost. Reductions and scans will take somewhat longer due to possibly non-contiguous locations, in which case data movement will be necessary to compact and align two collections for an elementwise operation.

4.3.2 Other Operations on Nested Collections

Although collection-oriented, the paralation model does not directly address set-like operations and non-atomic comparisons. These operations are not crucial for most conventional processing, but are a central part of the MISC associative model for knowledge processing processing. We first look at these operations on un-nested collections, and then consider them on nested collections.

Basis Set Operations

As with dynamic collections, an alternative representation can speed key operations. If the number of different elements present is reasonable in proportion to the number of processing elements available, a *basis-set* representation will support constant-time $\in$ $, \cup, \cap, \setminus$, and $\subseteq$ operations on non-nested collections. In the paralation representation these operations are $O(n/P + c \lg n)$ [Ble90], where n is the size of the smaller collection, P is the number of processing elements assigned, and c is the communication time to perform a permutation. The basis-set representation allocates a virtual processing element to each active element; each element has a flag for each active collection. This is very similar to the bit-vector set representation for serial computers. In addition to the set operations, insertion and deletion of basis-set members as well as the collection members can be performed in constant time. But like the associative-list representation, data movement is necessary for elementwise operations and reduction and scan operations require more time.

Transforming Representations

The choice between the contiguous (paralation), associative-list, and basis-set representations depends on what operations are to be applied to the collection. This is also the case for SETL implementations for serial machines. This research differs in developing representations specific to SIMD processing and efficient translations between the representations. Figure 4.9 summarizes the conversions. Conversion from a contiguous to a basis-set representation can be performed by converting the element values to a tuple of unique identifiers

(UIDs) then performing $A \leftarrow B_{UID}$. Conversion from a contiguous to an associative-list representation requires only the creation and initialization of the appropriate fields. Conversion from a basis-set to a contiguous representation requires a permutation $A \leftarrow B_{ORD}$ (where ORD is the tuple of the ordinality fields) for an ordered collection; the permutation index must be generated for unordered collections. Converting a basis-set to an associative-list representation only requires copying fields. Converting an attribute-list to a basis-set is the same as converting a contiguous to a basis-set representation. Converting an attribute-list to a contiguous representation is the same as the conversion from basis-set to contiguous.

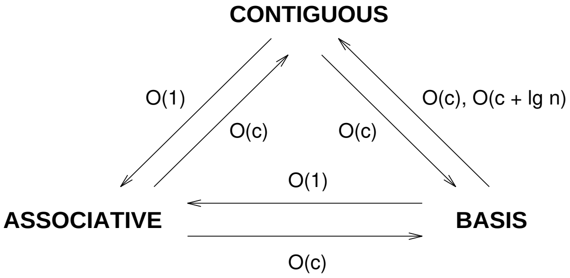


Figure 4.9: Representation Conversions

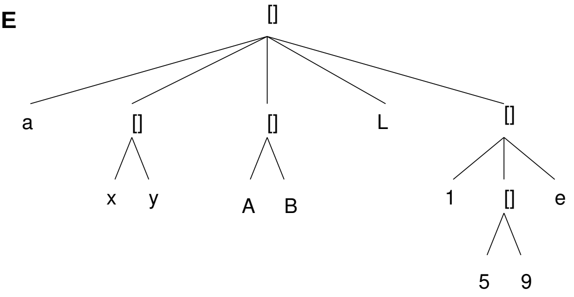
Membership and Matching Operations

The membership and matching operations can be defined on nested collections while remaining within the SIMD processing model. Since any element can be a nested collection, $\in$ requires $=$ unless 'compilation' to a basis-set UID is possible, and $X = Y$ is equivalent to $(|X| = |Y|) \wedge (X \subseteq Y)$ so that determining subsumption between collections is the key operation.

Nested collections, as well as Charles Peirce's existential graphs and LISP lists, can be conceptualized and represented as trees. Ordered nested collections can also be represented as a regular expression; Figure 4.10 shows an example. Comparison of these knowledge representations then becomes one of subgraph isomorphism, which can be done in polynomial time since both graphs are trees [GJ79]. The paralation internal representation for nested collections indicates groupings at each level of nesting, supporting both bottom-up and top-down methods, such as those presented by Hoffmann and O'Donnel [HO82]. More specifically, Lee presents SIMD systolic algorithms for matching regular expressions in time $O(n)$, where n is the length of the regular expression, using $O(n)$ processing elements. Willett *et al.* present algorithms for testing subgraph isomorphism on a single SIMD array and controller based on Ullmann's refinement, with expected time $O(n^2)$ on $O(n^2)$ processing elements for random graphs [WWR91, Ull76]. Since a tree-node can have at most one parent the number of nodes in the tree and symbols in the regular expression is $O(e)$, where e is the total number of atomic elements, provided 'empty' nestings such as $[[[[x]]]]$ are prohibited. Nested collections in the various representations can be converted into the adjacency matrices used by Willett et al.'s algorithms, or into a regular expression in time $O(e)$ by traversing the tree induced by the nesting, so $\subseteq$ and $=$ operations can be performed in polynomial time: $O(e^2)$ for general collections and $O(e)$ for nestings where all collections are ordered. In general membership testing, $\in$, requires the multi-level indexed search methods, giving time $O(e \lg N)$ where N is the number of elements in the right-hand collection; otherwise time is $O(N)$.

A NESTED TUPLE [a,[x,y],[A,B],L,[1,[5,9],e]]

THE INDUCED TREE



THE REGULAR EXPRESSION [](a[](xy)[](AB)L[](1[](59)e))

Figure 4.10: Representations for Nested Collection Matching

4.4 Summary

This chapter has presented arguments for associative processing as a basis for massively parallel processing and detailed an extended associative processing model and its pseudocode notation. The pseudocode notation is used in Chapter 5 to express example algorithms for analysis, avoiding the bias of existing codes and languages and emphasizing the formalisms of the underlying operations. Although subsidiary to the development of the MISC architecture, the model features several important innovations. Nested collections are the single notation used to define data structures for knowledge processing, extending associative processing from basic vector matching to processing complex relational structures. The extended model combines features of associative processing with set and collection oriented languages, extends prior work to more efficient processing of dynamic collections, and adds efficient collection matching. The programming model also combines implicit low-level data parallelism with explicit high-level parallelism.

5. Application Characteristics

With the SEGUL notation established, we can now characterize application processing requirements. This chapter presents new parallelizations and analysis, along with discussion of application characteristics for the example algorithms. Sections on the three example algorithms are followed by an analysis comparing exhaustive search with full data parallelism to multi-level indexed search. The next section presents common application characteristics derived from the analysis and manual profiles. These form the basis for architectural development in the next chapter. The example pseudocodes also serve to illustrate the utility and expressiveness of SEGUL and the associative programming model.

5.1 Sparse Distributed Memory

This section characterizes the SDM algorithm, beginning with expression of the basic algorithm in the SEGUL notation followed by a faster version described in the SDM literature but not immediately obvious from the most straightforward SEGUL description. We then discuss an analysis showing that exhaustive Hamming distance calculation is necessary in the basic algorithm due to the high dimensionality of the input. This section also summarizes alternative SDM algorithms, some of which may avoid the requirements of exhaustive distance calculation and closes with presentation of instruction profiles and discussion of processing characteristics.

5.1.1 Parallelizing the Basic Algorithm

Application characterization begins with a SEGUL programming lesson. SDM is a good example of the match-set-operations programming style, but the most compact expression of the data structure does not lead to a good implementation. Although it's doubtful that a compiler could identify and provide the proper data-to-processor mapping, the programmer could do so within the SEGUL syntax with only a general knowledge of SEGUL processing-element allocation and not necessarily any knowledge of the underlying hardware.

Straight Storage

As shown previously in Figure 2.2 (page 16), SDM uses a number of "physical locations," each of which contains the location address and a series of weights. Let m be the number of input bits, n be the number of output bits and N be the number of physical locations for an SDM associative memory. All of SDM's permanent storage, or physical locations (plocs) could then be defined concisely as `nonunique [code(m) addrs, nonunique [int weights n] ] plocs N`. Under the Paralation model, the data would be "flattened" to a tuple of addresses and weights and the correspondence between addresses and weight vectors would be lost and PE utilization would be poor. Instead, the pseudocode for the basic SDM algorithm in Figure 5.1 uses the `type` construct so that each element of the set of physical locations is a tuple of an address and a weight vector, and each tuple is assigned to a processor or virtual-processor. This is referred to as *straight storage* in the SDM literature; the physical location address and its associated weights are stored in the same PE.

For both write (learn) and read (recall) operations, the first step in the pseudocode forms the set of activated locations as all those with Hamming distance within the specified radius (Phase 1). This is followed by operations on the set (Phase 2). For write, weights in the activation set are incremented or decremented depending on the corresponding position in the desired output. The `for` loop could be expressed more succinctly as `code(m) output ← [boolean:x m | +(activePlocs(weights)<radius]`, although this obscures the iteration over the output bit positions. For read, the weights are summed for each corresponding position in the output. The SEGUL code is succinct, inherently parallel, and at the level of algorithm description.

```
/******** Data Declarations ********/
int m ← 1024, n ← 64, N ← 262144, radius ← 446
type nonunique [byte n] weights
type nonunique [code(m) addr , weights] ploc
nonunique ploc N plocs

/******** Write ********/
when receive ['SDMwrite', thisPN, ?code(m) input, ?code(m) output]:
    activePlocs ← { x ∈ plocs | hdistance(plocs(addr),input) ≤ radius }
    activePlocs(weights) ← activePlocs(weights) + (output∗2 − 1)
;

/******** Read ********/
when receive['SDMread',thisPN, ?code(m) input]:
    activePlocs ← { x ∈ plocs | hdistance(plocs(addr),input) ≤ radius }
    for i = 1 to m
        output(i) ← ( +activePlocs(weight,i)) ≥ 0
    send['SDMresult',thisPN, output]
;
```

Figure 5.1: SDM Pseudocode

Transposed Storage

Straight storage is usually not the fastest parallelization. Weights for each output position are log-summed (one weight in each PE) and iterated over the n output positions, giving Phase 2 time $O(n \lg a)$ where a is the number of activated locations. Log-sum is not the quickest approach when there is greater parallelism over the output bit positions. Exploiting full nested parallelism is impractical, as hundreds of thousands of PEs would be required for a single SDM module.

In the alternative *transposed storage* each PE is assigned addresses as above but n PEs are used, each containing all N weights for a given output-bit position. Pseudocode for transposed storage is given in Figure 5.2. In this case a values are serially summed, all n

positions in parallel, with time $O(a)$. As the PE with the storage location address does not contain the corresponding weights, it is necessary to communicate the indexes of the resulting active locations from Phase 1 to all PEs (this is unnecessary in straight storage). This is most quickly done by simply reading the activated locations into the controller, which then uses the locations as memory addresses into the weight array, taking $O(a)$ time. If a PE has more than one active location, they can be read out in sequence with no effect on the number of operations. It is also necessary to read the resulting sums from the various PEs, requiring $O(n)$ time (for straight storage this is part of the iteration over n). Phase 2 time for transposed storage is thereby $O(a + n)$.

Estimating constant factors from manual profiles of the pseudocode, it appears that a straight storage read operation is faster only with 32 or fewer outputs. Here only speed is considered; for SDM the cost of memory strongly dominates, and is the same for both cases. Very similar results are obtained assuming m PEs for straight storage. A write operation has time $O(n)$ for straight storage (iterate over the output bits) and $O(n+a)$ for transposed storage (sending the target output values to the appropriate PEs, then iterating over the active locations), excluding Phase 1. Estimating constant factors from manual profiles, the transposed storage has a speed advantage of 2.1 at Kanerva's standard example size ($m = 1024$, $n = 1024$, $N = 262144$) to 2.5 for a currently implementable size ($m = 256$, $n = 256$, $N = 32768$) for reads. For writes, time is nearly identical for the two storage methods, so that at 50% reads and 50% writes the speed advantage for transposed storage is reduced to about 1.5 to 1.8. These alternative weight allocations have been discussed in other studies as well [RN91, Rog90]. Since the transposed method is the fastest known mapping of the problem to a SIMD array, these speed penalties are not intolerable for a high-level expression of the algorithm. Further, the optimal method can be recognized as superior by considering the asymptotics, independent of low-level machine details, and expressed naturally in terms of linear algebra, readily expressible in SEGUL.

5.1.2 Analysis

The pseudocode in Figures 5.1 and 5.2 are for a complete associative memory module, allowing a full analysis of this "application" for determining processor needs and instruction profiles. An ambitious SDM associative memory might have a thousand input bits and 1/4 to 1 million physical locations. Most implementations to date have several hundred input bits and several thousand physical locations. Where p is the number of PEs available for a given SDM and p is less than N, each PE gets N/p location addresses and nN/p weights. Although the resulting VP ratio could be on the order of 10^3 for addresses and 10^6 for weights, SDM performance is still excellent. The Phase 1 exhaustive calculation can be implemented very efficiently in hardware, as discussed in Section 6, and dramatically prunes the number of weights (activated locations) processed in Phase 2.

Activated Locations

Kanerva gives the optimal probability of activation to maximize the signal-to-noise ratio in SDM recall as $p \approx (2NT)^{-1/3}$ where T is the total number of patterns to be stored [Kan92]. Keeler demonstrates that the capacity of SDM per storage element (weight) for

```
/******** Data Declarations ********/
int m $\leftarrow$ 1024, n $\leftarrow$ 1024, N $\leftarrow$ 262144, radius $\leftarrow$ 446
nonunique [code(m)] plocs N
type nonunique [byte n] weight
nonunique [weight N] weights
nonunique [int m] sum

/******** Write ********/
when receive['SDMwrite', thisPN, ?code(m) input, ?code(m) output]:
    activePlocs $\leftarrow$ { x $\in$ plocs | hdistance(plocs(addr),input) $\leq$ radius}
    foreach x $\in$ activePlocs
        weights(x) $\leftarrow$ weights(x) + (output$*$2 $-$ 1)
;

/******** Read ********/
when receive['SDMread', thisPN, ?code(m) input]:
    activePlocs $\leftarrow$ { x $\in$ plocs | hdistance(plocs(addr),input) $\leq$ radius}
    foreach x $\in$ activePlocs
        sum $\leftarrow$ sum + weights(x)
    code(m) output $\leftarrow$ sum $\geq$ 0
    send['SDMresult',thisPN, output]
;
```

Figure 5.2: SDM Pseudocode for Transposed Storage

a given output error rate ε is the same as the Hopfield net [Kee88], giving $T \approx .15N$ for $\varepsilon = .995$. Kanerva suggests that the number of patterns stored be reduced to allow for approximate input and greater output accuracy, and suggests 1% to 5% of N. Substituting and solving for the optimal number of activated locations, a, gives

$$a \approx 2.15N^{1/3}.$$

Thus, the number of active locations requiring processing grows asymptotically *much slower* than the number of physical locations. If a PEs are used, then the expected value of the number of active locations per PE is approximately 1; each PE would typically process only a single physical location for Phase 2. It may be desirable to have more than a PEs. Assuming a Poisson distribution[1] on the number of active locations per PE, if $p = a$ and $\lambda = 1$ then there is an approximately 26% probability of a PE having more than one active location. It is necessary to allow for 4 locations per PE to get the probability of a greater number below 1%. At $\lambda = .25$ ($p = 4a$), the probability of more than 1 location per PE is about 2% and the probability of more than 2 locations is about 0.2%. For transposed storage the probability of a PE having 2 or more active locations is also about 0.2%.

---

[1] The discrete Poisson distribution with mean λ is given by p[x successes] = $\sum_{k=0}^{x} \frac{\lambda^k e^{-\lambda}}{k!}$, so that the probability of y or more events is 1-p(y).

Processor Scaling

For further analysis and simulation, processor scaling is based on a combination of analysis and engineering judgment. Although in typical SDM implementations N is $O(m^2)$, there is no formal analysis to support this observation.[2] In a space-time analysis where space is taken as the number of processors, the space-time product Phase 1 is the same irregardless of the number of processors, so that the optimum number of PEs is $O(N^{1/3})$ (from the activated location analysis). This leads to memory requirements well over 1 MByte per PE and a higher virtual-processor ratio, slowing Phase 1. A system-level cost-performance analysis (including processor, memory, and other costs) indicates $1/4N$ processors, but this quickly leads to tens of thousands of processors and speed many times the real-time target. We thereby take the middle ground of $O(N^{1/2})$ processors, balancing the various concerns, and reducing the likelihood that any given PE has more than one or two active locations during Phase 2. In all cases the ideal number of processors scales much slower than the number of physical locations.

The Necessity of Exhaustive Hamming Distance

It is highly desirable to avoid the exhaustive Hamming distance used in finding the active locations. Phase 1 corresponds to a k-nearest neighbors search. Omohundro shows that the single closest match can be found in $O(\lg N)$ time using a K-d tree search and claims that at most k adjacent bins must be searched for a k-NN search [Omo87]. Here k refers to the number of nearest neighbors sought (a for SDM), K the dimensionality of the search parameter space (m for SDM) and N the number of stored points (N "physical locations" for SDM). A parallel implementation might appear particularly attractive, since a leaf-bin with a large number of nodes could be searched in parallel, so that with the number of points in each bin around k (reasonable for SDM) one might expect that on average very few adjacent bins would have to be searched. This is not the case, however; high dimensional spaces are often counter-intuitive.

Consider a real-valued hypersphere of sufficient radius to enclose k points. Even if a hyper-rectangular bin contains k or more points, the center of the sphere could be near an edge or corner and the number of bins the sphere could intersect is potentially exponential in the dimensionality of the space. In Boolean spaces, a point is always at a corner. Friedman *et al.* provide a rigorous analysis of generalized nearest neighbors search, deriving an expected value of the number of points that must be examined, independent on the underlying distribution of points and the number of points per bin [FBF77]. This expected value is based on the ℓ_∞ norm, and is a lower bound for all ℓ-norms (including the ℓ_1 Hamming distance and ℓ_2 Euclidean distance). Switching to this dissertation's notation, the expected number of points examined ([FBF77], eqn. 19) is

$$R_\infty = (k^{1/m} + 1)^m.$$

As m goes to infinity $k^{1/m}$ approaches 1 and the bound approaches 2^m; for a small SDM ($m = 256$, $a = 67$), $k^{1/m} = 1.017$ and for a large SDM ($m = 1024$, $a = 138$) $k^{1/m} = 1.005$.

---

[2] Storage capacity (number of uncorrupted patterns) has been shown to be $O(N)$ [Kan92, Kee88]; there may be an indirect relationship between the m and the number of patterns to be stored for good performance.

Therefore the expected-case number of points searched is $\Omega(2^m)$ and *exhaustive search is faster* where $N < \Omega(2^m)$.

5.1.3 Alternative SDM Algorithms

Alternatives to the base SDM algorithm promise reduced computation (possibly even avoiding exhaustive distance calculation) and increase capabilities. Jaeckel's variations, while reducing the number of bits to be compared in computing an individual location's distance, does not appear to offer speed benefits only for hard-wired special-purpose implementations [Jae89a, Jae89b]. For general-purpose parallel processing all input-bits must be broadcast to the array, requiring $O(m)$ time. Inverted indexing turns out not to be an option for Jaeckel's methods for either serial or parallel implementations. Jaeckel's work to indicate improved recall accuracy, but the analysis is restricted to uncorrelated inputs.

For inverted indexing, time is $O(qr)$ where q is the average number of attributes relevant to a given query and r is the average number of records having a given attribute [Luc89]. For the case of SDM Phase 1 search with uniform distribution, $q = .5m$ and $r = .5N$, so time is $O(Nm)$ and we are still no better off than exhaustive Hamming distance calculation unless the input vectors are sparse.

Other methods developed during this research, such as performing a closest-match k-NN search followed by iteratively traversing stored pointers to some number of the matches closest neighbors and partitioning the space by multiple dimensions at once (rather than one at a time as in K-D trees) also show worst-case performance $O(Nm)$. The k-d tree method (Section 5.1.2) with large bins might be useful if some error in selecting activated locations is tolerable. These alternatives require extensive analysis of expected case behavior and resulting errors.

5.1.4 SDM Characteristics

We are thereby left with the basic SDM algorithm. The dynamic execution breakdown in Table 5.1 is based on manual profiles of the pseudocode for transposed storage, over an equal number of reads and writes on an $m = n = 1024$, $N = 262144$ SDM. Global communication refers to collecting and broadcasting active physical location indexes, and global communication is receiving SDM inputs and sending outputs. Hamming distance clearly dominates total execution time for transposed storage.[3]

Multi-level indexed search techniques cannot be used to prune the Phase 1 search on SDM; exhaustive search (Hamming distance computation) is the best one can do due to the high-dimensionality. Phase 1 itself, however, extensively prunes the number of weights to be processed to $O(N^{1/3})$. Time for Phase 1 is $O(mN/p)$, where p is the number of processors, independent of the weight storage method. Total time thereby becomes $O(mN/p + n \lg a)$ for straight storage and $O(mN/p + n + a)$ for transposed storage. For transposed storage, it is convenient to use m processors. For simplification, assume $m = n = p$ (the first equality is typical of autoassociative memories and sequence generator/recognizer SDM,

---

[3] For straight storage summation time rises to some 20% for smaller SDMs, and Hamming distance and summation times are roughly equal for large SDMs.

Operation	time
Hamming distance	83%
summation	12%
supervisor communication	3%
increment-decrement	1%
conditionals	1%

Table 5.1: Projected Breakdown of SDM Execution Time

and the second is from the PE allocation discussed above) and $N = O(m^2)$ (appropriate for most SDM implementations). This simplifies the times to $O(N + N^{1/2} \lg N^{1/3})$ for straight storage and $O(N)$ for transposed. Under these assumptions the number of PEs scales as $O(N^{1/2})$. This indicates excellent scaling, with time linear or nearly linear and number of PEs sub-linear in problem size. Total memory requirements are linear in problem size, so that memory per PE grows with the problem size and becomes the only limiting factor in scalability.

Even with the resulting high virtual-processor (VP) ratio, early analysis projects that a single MISC PN can provide on the order of 10^{12} connections (weights) per second throughput with a storage capacity on the order of 10^9 weights for 10^6 "neurons". This is over 30 times faster than the Stanford application-specific SDM prototype and 200 times faster than a 16 K PE CM-2 executing SDM [FKB89], and represents "real-time" performance with thousands of input-output iterations per second.

Although no alternatives are clearly viable in reducing Phase 1 time on a parallel implementation, there are still numerous options whose investigation requires the flexibility of a general-purpose machine. These include Rogers' genetic learning of the physical location addresses, other methods proposed by the author for adjusting the address distribution in response to input distributions (particularly in minimizing the number of locations to mitigate SDM's large memory requirements) and dynamically adjusting the distance radius, the importance of weighting regions and/or dimensions to improve classification [Wat85], Prager and Fallside's adaptation to integer-valued inputs, and other input codings [PF89].

5.2 Semantic Networks

This section presents parallelizations of simple marker-passing semantic networks, approximating the behavior of more complex SN systems. After comparing edge-based and node-based network representations we introduce a new development in SNs, using inheritance codes to replace many propagation cycles with a single parallel code comparison. This section then presents an analytical model of the number of markers propagated. This section also covers analysis of the node-based network followed by instruction profiles and processing characteristics.

5.2.1 Edge-Based SN Representation

The edge-based representation, suggested by Foster's CAM shortest-path algorithm [Fos76], uses an edge-based graph description where each edge is represented as a tuple of **[source,destination]** with additional elements for node markers and to indicate if the tuple has been processed or not. In this algorithm, source and destination elements are symbolic node labels and the marker propagation proceeds by associative marking with no inter-PE communication, as shown in Figure 5.3. Markers are propagated as a "wave" from nodes marked in the prior cycle. The set of markers (edges) in the wave are associatively marked in parallel, and its elements are then processed iteratively. The destination of each edge in the wave is read, and all edges whose source matches that destination are marked. As Foster points out, parallelism is limited (only the wave formation and the destination marking) and the algorithm serializes over the elements of the wave; that is, over individual marker propagations. Time is thereby $O(LE_a)$ where L is the number of marker propagation cycles (length of the longest propagation path) and E_a is the average number of edges active per cycle. This is not as bad as it first seems. As is shown below, E_a is empirically $O(N^{3/5}/\lg N)$; time is asymptotically less than linear. The second method is nevertheless asymptotically better.

```
/******** Data Declaration and Creation ********/
type nonunique [code source, code destination, boolean marked, boolean processed] edge
nonunique {} graph

/******** Propagate Markers (1 cycle) ********/
wave ← {x in graph | marker=1 ∧ processed=0}
wave(processed) ← 1
foreach x in wave
    adjacent ← { y | graph(source)=x(dest) }
    adjacent(marker) ← 1
```

Figure 5.3: Edge-Based SN Marker "Propagation"

5.2.2 Node-Based SN Representation

The node-based representation uses an edge-list description where each tuple represents a particular node followed by a list of its outgoing edge destinations, as in Miranker's NETL simulator [MA90], along with additional elements to indicate if the tuple has been marked and whether it has been processed or not. In this algorithm, shown in Figures 5.4 and 5.5, each marker propagation iteration creates a local communication. Here the destinations are represented by the identifier of the PE they are allocated to. The wave of markers is generated as in the edge-based algorithm, but here the iteration is over the maximum arity of the marked nodes rather than the total number of markers being propagated, as shown in the **propagate** procedure pseudocode in Figure 5.5. As this is generally the preferred method, the figure shows code for a complete "application," allowing a full profiling and analysis. Time is $O(LaC)$ where a is the average arity of outgoing edges and C is the time for a random communication pattern.

For the node-representation algorithm, it is necessary to consider communication time and the possibility of collisions. Assuming a mesh communication topology, time for a random many-to-many communication is $O(\sqrt{P})$ where P is the number of PEs. Thus,

```
/******** Data Declaration and Creation ********/
integer maxArity ← 8, N ← 32768, maxCat ← 2
type nonunique [code(l) destination maxArity] outedges
type nonunique [subsumptioncode maxCat] taxonomy
type nonunique [code(l) label, taxonomy, code(m) nodetype,
                code(n) marked, boolean hassent, outedges] node
nonunique {} graph

/***** Procedures *****/
/*** Create a Node ***/
procedure createNode(code(l) label):
    graph ← graph + node[label, taxonomy[], outedges[]]
;
procedure addEdge(code(l) source, code(l) dest):
    graph(node(source), _ , _ , _ , _ , outedges] ←
        graph(node(source), _ , _ , _ , _ ,outedges]+ outedges[;dest]
;
procedure reset ():
    graph(marked) ← 0
    graph(hassent) ← FALSE
;
/*** Mark Nodes ***/
procedure search(subsumptioncode taxo, code(n) marker):
    wave ← { x ∈ graph | graph(nodetype) ⪯ taxo }
    wave(marked) ← marker
;
```

Figure 5.4: Node-Based SN, part 1

time for the node-representation algorithm $O(LaC) = O(\lg N a\sqrt{P})$. For a 32 K node SN, the average number of markers passed is about 100, so that with a 1 K PE MISC processor node only 1/10 of the PEs would be sending a message. Assuming a Poisson distribution on the destinations and $\lambda = 0.1$, the probability of a given destination receiving more than 2 markers is 0.0002. For early analysis, collisions can be comfortably ignored; even with 256 PEs ($\lambda = .39$), the probability is 0.007. Using manually generated profiles from the two algorithms, analysis shows that the node-representation is approximately 3 times faster for a 32 K node SN where the SN resides on a single MISC PN.

5.2.3 Subsumption Codes for Semantic Networks

Note that the pseudocode in Figure 5.4 includes a subsumption test in the initial node marking. Most SN topologies have sections of both tree and random-graph structure. The tree-like portions are usually the isa taxonomy (a dog is a mammal, a cat is a mammal, a mammal is an animal ...) and the random portions are the more specific relations,

```
/*** Propagate Markers (1 cycle) ***/
procedure propagate(/* criteria */):
    nextwave ← {}
    for i = 1 to maxArity
        allowed ← {x ∈ wave | (wave(hassent)=FALSE)
            ∧ /* edge or marker meets criteria */}
        $graph(marked)_{received} \leftarrow wave(marked)_{wave(outedge(i,dest)),\vee,\_,allowed}$
        allowed(hassent) ← TRUE
        nextwave ← nextwave + received
    wave ← nextwave
;

/*** Read Results ***/
procedure readResults(/* criteria */):
    answer ← {x ∈ graph mid /* meets criteria */}
    foreach x ∈ answer
        print x(nodelabel)
        print x(marked)


/******** Example Query ********/
reset()
search(A,1)
propagate(/* if has marker 1 */)
search(B,2)
propagate(/* if has marker 2 */)
readResults(/* has marker 1 and 2 */)
```

Figure 5.5: Node-Based SN, part 2

attributes and exceptions. By including a subsumption code with the node, a key transitive inheritance relation such as isa can be determined in constant time as part of the initial node marking rather than requiring marker propagation cycles.[4] A node might thereby include one or more subsumption codes in addition to the outgoing edges and label.[5] The network topology would then have few if any tree portions, and can be approximated as a random, disconnected directed graph.

---

[4] This approach was independently developed by Geller somewhat earlier [Gel94a, Gel94b].

[5] The labels are used as the destinations in the outgoing descriptions and could be arbitrary or they could specify the parallel node storage location. Either way, the label can also be used to associate the node with additional information (such as [088F7, 'beagle']).

5.2.4 Analysis of Marker Propagation

This research proposes an analytical model for the average number of markers propagated and the average number of propagation cycles for a query. DeMara proposes a similar model but with a separate term to adjust for marker fan-in [DeM92]. The MISC model is necessary to project SN behavior for networks much larger than those implemented to date.

Consider two extremes. In the first, a marker propagation is a single chain ($E_a = 1$, $L = N - 1$), and in the second an N-ary tree ($Ea = N$, $L = 1$). In between these extremes, consider the pattern of marker propagation as an A-ary tree. The average number of propagating markers is then

$$E_a = \frac{\sum_{i=0}^{L} A^L}{L} = \frac{A^{L+1}}{(A-1)L}.$$

More realistically, SNs begin by marking a small number of nodes. Markers then fan out from these nodes, and many markers converge to a smaller number destinations. As a simplifying assumption, consider propagation as starting from a single node, fanning out, then fanning back in to a single node. If fanout is for half the cycles, L would be replaced by $L/2$ in the above equation. L is bounded by $\log_A N$. Let f be the portion of fanout propagation cycles. Substitution into the above gives

$$E_a = \frac{A^{\frac{\log_A N+1}{f}}}{(A-1)\frac{\log_A N}{f}}.$$

Surveying several semantic networks, network arity A appears to asymptotically approach 4 for large networks, except for certain natural-language processing approaches [DM93, DeM92]. Propagation is usually restricted to certain types (labels) of edges and some node types "absorb" markers and do not further propagate. We fit parameters in the above equation to the two largest semantic networks for which we have the appropriate information. For a portion (40,000 nodes) of the cyc SN knowledge base [LG90], the average path length is 22. Working backwards, this gives an effective arity of 1.62, suggesting that markers propagate on slightly half the network links. For a SNAP application with $N = 32,000$, the average number of markers per cycle is 100 [DM93]. Substituting into the above equation with effective arity $A = 1.62$ gives $f = 1.68$, which includes the effect of starting and finishing with more than 1 node as well as the relative number of fanin versus fanout cycles. As an effort against overfitting, we reduce these coefficients to a single parameter, $\alpha = A = f \approx 5/3$, giving an error of about 5% for L and E_a for the above examples. Substituting and simplifying,

$$E_a = \frac{\alpha^{\frac{\log_A N+1}{\alpha}}}{(\alpha-1)\frac{\log_A N}{\alpha}} = N^{1/\alpha}\frac{\alpha^2}{(\alpha-1)\lg N} = O(\frac{N^{1/\alpha}}{\lg N}) \approx O(\frac{N^{3/5}}{\lg N}).$$

With these results, we see that only a small portion (less than 1%) of nodes are actively sending markers at any given iteration.

Figure 5.6 plots the analytical model against additional data from DeMara [DeM92] as well as 2MISC simulations with effective node arity (average numbers of markers sent from each activated node) of 1.4 on randomly generated semantic networks.[6] As expected from the analytical model, the average number of markers per cycle is highly dependent on effective node arity. The data from DeMara include actual applications related to natural language parsing and understanding, as well as synthetic examples with differing topologies and SN operations (such as tree, random graph, mesh, replicated random graph topologies and inheritance and intersection operations), with significantly different marker characteristics. From the limited examples, it is not clear if the average number of markers continues to grow as the number of nodes increases, or if they tend to level off at around 120 average markers per propagation cycle. The simulations fit the *mixture* of DeMara's examples and the analytical model when initially marking 10 nodes then propagating with the above rules and ρ, the probability that an outgoing edge of an active node sends a marker, of 0.35. This is close to those obtained from very limited actual SN application characteristics: average effective node arity (A) of 4, $\rho = 0.4$, and network diameter $\log_A N$ (where N is the total number of nodes in the network). DeMara's independent analysis assumed the same diameter and experimentally found ρ to range from 0.2 to 0.4.

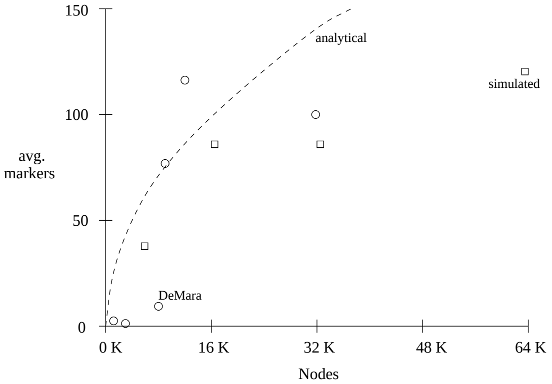


Figure 5.6: Average Markers per Propagation Cycle

5.2.5 SN Characteristics

It appears appropriate to have the number of processors scale with the average number of markers being passed, avoiding idle processors during marker passing cycles. SNs also exhibit significant pruning after the initial node marking marking. From the analysis in Section 5.2.3, given a 256 K node SN, on average less than 0.1% of the nodes are active at any given time. Analysis shows that despite the resulting high virtual-processor ratio during the marking phase, space-time product is better when the entire SN is placed on a single PN, avoiding propagating markers across the inter-PN communication network. Analysis

---

[6] The value 1.4 was chosen to most closely match behavior of the several example semantic networks. The value 5/3 was chosen to be closer to the larger examples and to provide a conservative (high) estimate of the number of markers propagating.

additionally shows that a large PN of 1 K PEs is appropriate even though there are only 100–200 active markers. The increased number of PEs reduces the virtual processor ratio during marking, reduces the probability that any processor will have more than one message to send, and reduces the number of sending processors during marker passing, improving routing speed. For certain types of node markings, techniques such as hashing or k-d trees can avoid iterating over the virtual-processor positions, allowing constant or logarithmic time initial node marking.

Table 5.2 summarizes dynamic execution times from manual profiles of the pseudocodes. The semantic network is assumed to have 256 K nodes and the profiling includes creating the network via 1 M edge-create calls, then performing 32 K queries. Matching operations dominate total time, followed by local communication which represents marker propagation. Global communication is receiving queries and returning results. This analysis underestimates local communication time, as the lower bound for mesh communication is used in lieu of any detailed knowledge of the MISC array network and routing algorithm.

Operation	time
matching	67%
local communication	25%
flow control	5%
logical operations	2%
global communication	1%

Table 5.2: Projected Breakdown of SN Execution Time

5.3 Graph Retrieval

The graph retrieval example involves several algorithms; this dissertation emphasizes subgraph isomorphism which occupies the majority of execution time. This section first details Ullmann's serial SI refinement algorithm and the author's generalizations of the algorithm. We then discuss Willett *et al*'.s parallelizations for single-bit processors followed by a new parallelization for multi-bit processors. This section then presents Levinson's Method III multi-level indexed search algorithm, and discusses integration with the multi-bit processor SI parallelization. We then discuss compiling hierarchies into subsumption codes and operations using subsumption codes. The section concludes by combining the various algorithms into an instruction profile and discussing application processing characteristics.

5.3.1 Subgraph Isomorphism

Work has focused on parallelizing subgraph isomorphism (SI) for search in large KBs of relational structures (graphs). SI is an NP-complete problem; given two graphs, G_a and G_b, we want to know if G_a is a graph that is properly contained in G_b. Graph G_a is a subgraph of G_b if all nodes and edges in G_a are present in G_b; G_a is isomorphic to a subgraph of G_b iff there exists a one-to-one correspondence between the set of nodes of this subgraph and those of G_a that preserves adjacency [Ull76]. Let relations (with

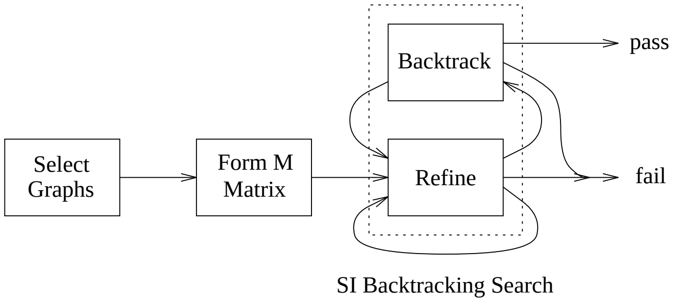


Figure 5.7: Flow Diagram of the Refinement Subgraph Isomorphism Algorithm

relation equivalence requiring exact matching) and node- or concept-types (with equivalence including concept-type subsumption and individual instantiations) be represented as nodes in a graph. SI then corresponds to the canonical CG operation of projection [Sow84], and G_b is a specialization of G_a. Determination of subsets of concept-types and relations and their immediate adjacencies is a necessary but not sufficient condition for testing generalization and specializations of graphs.

As an alternative to the brute force method of generating and testing all possible mapping of nodes in G_a to those in G_b, a backtracking tree search can be used to order and control the search. Ullmann presents a refinement technique to drastically prune the search tree [Ull76]. The algorithm represents the graphs and the binding between nodes as three Boolean matrices: A the adjacency matrix of G_a, B the adjacency matrix of G_b, and M the 'match' matrix. A bit $m_{ij} \in M$ is 1 if node $i \in G_a$ is a candidate for binding to node $j \in G_b$ (otherwise m_{ij} is 0). Figure 5.7 diagrams the overall structure of the algorithm. First graphs are selected for comparison; those selected could include all graphs (exhaustive comparison) [WWR91], those which pass filtering criteria (single-level indexing) [WWR91, Len88, Len92], or those passing multi-level indexing (initial graph SI tests filtering subsequent graphs) [LE92, HLR93]. Second, the match matrix M is formed. Third, the backtracking refinement search is performed. The refinement procedure is called at each branch of the backtracking search tree, dramatically pruning the number of branches explored.

Hendler *et al.* present a structure-matching algorithm for finding instances of a query graph within a semantic network [AHEK94]. This corresponds to finding multiple subgraph isomorphisms within a single large graph. Although time can be linear in the number of nodes, one processor per node and edge in the semantic network is evidently required. In this dissertation we investigate subgraph isomorphism in the context of a knowledge base of multiple graphs. As we will see, a multi-processor multi-bit parallelization of Ullmann's algorithm can also achieve expected-case time linear in the number of nodes in the individual graphs, with far fewer processors than the total number of nodes and edges in the knowledge base.

Refinement

For undirected graphs with unlabeled edges, the initial match matrix is formed by iterating over all i and j and setting m_{ij} to 1 if the node type of i is the same as that of j and the arity of i is less than or equal to the arity of j, otherwise m_{ij} is set to 0, thus forming the set of all possible bindings. Operations on the matrices then determine if the neighbors of each node also have candidate bindings, removing bindings if this is not the case. The algorithm thereby begins with an approximate solution, which is then refined to a more accurate solution. This procedure is not guaranteed to remove all invalid bindings; it is sufficient that that refinement eliminate some false node mappings while not removing any correct mappings.

The refinement algorithm keeps a candidate binding m_{ij} iff $\exists(x)\ a_{ix} \wedge \neg(\exists(y) m_{xy} \wedge b_{yj})$. Figure 5.8 presents Ullmann's original code. The logical criteria can be expressed by the matrix operations

$$M' = M \wedge \overline{A \times \overline{(M \times B)}}.$$

For simple graphs, refinement alone is often enough to determine SI. Determining if the smaller organic compound of Figure 2.4 (page 19) is a generalization (subgraph) of the larger compound demonstrates that the refinement procedure alone is insufficient; backtracking search is still required in many cases, particularly those involving graphs with cycles [HLR93].

```
/******** Refinement Procedure ********/
bitmatrix: A, B, M
    REPEAT
    Mcopy = M
        /* R = M x B */
        R = [ $r_{xj}$ | $\exists(y)$ st ($M(x,y)$ and $B(y,j)$) ]
        /* M = M and (not(A x not (R))) */
        M = [ $m_{ij}$ | ($M(i,j) = 1$ and
            $\exists(x)$ st ($A(i,x)$ and not($R(x,j)$))) ]
        if [ x in M | $\exists(row)$ st $M(row) = 1$ ] = []
        return "fail"
    UNTIL M=Mcopy
    return "unknown"
```

Figure 5.8: Ullmann's Refinement Algorithm Pseudocode [Ull76]

Backtracking

Ullman presents the psuedocode of Figure 5.9 for the full SI test, which uses the refinement procedure to prune a backtracking search. H is a vector of integers, of size equal to the number of nodes in G_a, indicating which column of M has been selected at which depth. F is a vector of bits recording which columns of M have been used, and d is the current search depth. If at any time there is a row in M consisting of all zeros the refinement fails, as a node in G_a does not have a corresponding node in G_b. If all rows in M have a single one, the refinement succeeds and the match matrix corresponds to one of the possibly multiple subgraph isomorphisms. Otherwise refinement iterates until the M

matrix has not changed from the prior iteration. In the latter case the backtracking portion of Ullmann's algorithm determines if M represents a valid subisomorphism, checking that each row of M has exactly one 1 and that no column has more than a single 1. If so, we're done. Otherwise the backtracking portion recursively continues the search by arbitrarily selecting one binding for a node in G_a and calling refinement again. The full algorithm terminates when either an isomorphism has been found or there are no more bindings to try. Hughey *et al.* present examples of the SI refinement algorithm on conceptual structures [HLR93]. Ullmann's algorithm must be modified if there are labeled nodes, concept- and relation-types, or directed edges as in CGs.

```
$$-----------SUBGRAPH ISOMORPHISM--------------------$$
$if succeed, M indicates an isomorphism$
1:   d=1; H(1)=0
     if REFINE(M) = fail then RETURN "fail" ;
2:   if {x | there_exists (j) st (M(d,j)
             AND NOT(F(j)))} = {}
             goto 7
     Md = M
     if d = 1 then k = H(1)
     else k = 0 ;
3:   k +1=
     if (M(d,k)=0) AND (F(k)=1) then goto 3
     forall j != k M(d,j) = 0
     if REFINE(M) = "fail" then goto 5 ;
4:   if d < Q(nodecount) then goto 6
     RETURN succeed ;
5:   if {x | there_exists (j>k) st ((M(d,j)=1)
             AND (F(j)=0))} = {}
             goto 7
     M = Md
     goto 3 ;
6:   H(d) = k; F(k) = 1; d +1=
     goto 2 ;
7:   if d = 1 RETURN fail
     F(k) = 0; d -1=; M = Md; k = H(d)
     goto 5 ;
;
```

Figure 5.9: Ullmann's Subgraph-Isomorphism Pseudocode [Ull76]

Extending Ullmann's SI

Directed and Labeled Edges Ullmann's refinement procedure requires that A and B are symmetric about the diagonal, which implies that the two graphs are undirected. Ullmann presents a modification for digraph isomorphism in which $m_{ij} = 1$ iff both the in and out degree of a_i and b_j are the same. This cannot be applied to subgraph isomorphism. Consider any edge $a_x \rightarrow a_y$. Reversing the edge direction increases the in degree of x, which may no longer be less than the in degree of the corresponding node in G_b. This is not necessarily the case; the in degree of a_x may have originally been more than 1 less than that of the corresponding b.

In conceptual structures, edge direction is determined by the relation type so that binary relations are treated as labeled edges and argument bindings must be checked in the backtracking portion of the algorithm to verify direction. Monadic relations can be handled as attributes attached to the concept-type node. Higher-order relations can be handled as graph nodes with appropriate correspondence criteria. Converting nodes to edges reduces the number of nodes in the graphs to be compared. In Sowa's "conceptual catalog" [Sow84], some 90% of the relations are binary and nearly all the remainder are monadic. Treating relations as labeled edges rather than nodes has a major speed advantage, as empirical time is $O(n^2)$ in parallel implementation [WWR91], where n is the number of nodes.

In Levinson's UDS, CS are transformed such that the relations and their adjacent concept-types become nodes in the UDS graph. Edges in the UDS graph represent the bindings between the concept-type arguments of their relations. A node has the form [relation, arg1, arg2,... argN], representing higher-order relations as a single node. Each node is in the UDS hierarchy, having a single subsumption code that can be used in forming candidate bindings. Representing CS at the relational level usually reduces the number of nodes, and the number of refinement iterations are usually reduced due to the greater specificity of the nodes.

Additional Criteria In many applications, including conceptual structures, nodes may bind if a concept node of G_a is subsumed by one of G_b. Node subsumption is determined by a separate concept-type hierarchy, typically based on class inheritance. We thereby set m_{ij} to 1 iff $a_i \preceq b_j$ and the number of relations adjacent to a_i is less than or equal to those of b_j.

Additional criteria and filters, such as minimum cycle and minimum self-cycle lengths[7], can also be applied in forming the initial match matrix. In testing projection between conceptual structures, there does not have to be a 1:1 mapping between the concept-type nodes and relations. This is readily handled by relaxing the mapping criteria applied in the backtracking portion of the algorithm.

Some criteria are better handled in the refinement portion of the algorithm rather than in forming M. For example, one could test that the relations of each node are a subset of those in the query graph. A subset algorithm on a serial machine could use hashing to test membership in the larger set, iterating over the nodes in the smaller set for time $O(a)$ where a is the size of the smaller set (the arity of a node in G_a in this case). We can not use this method on a data-parallel array and still process all nodes of all graphs simultaneously, and must instead iterate over all elements of the larger set (the arity of a node in G_b), giving time $O(a_m A_m)$ where a_m is the maximum arity of any node in any of the database graphs and A_m is the maximum arity of any node in the query graph. In the worst case time is

---

[7] These two loop lengths correct the problem encountered in the organic chemistry example (Figure 2.4, page 19), rejecting the SI in the refinement stage alone.

$O(n^3)$, asymptotically worse than the expected-case time of parallel SI. Here it is clearly better to check edge labels and directions in the backtracking portion of the code once a tentative SI mapping has been found. Rather than performing time-consuming operations for all possible node mappings, it is better to perform those operations only on the mappings in an otherwise subgraph isomorphism.

5.3.2 Parallel Subgraph Isomorphism

Prior Parallelizations for Single-Bit PE SIMD Arrays

Work by Willett, Wilson, and Reddaway forms the basis of our parallelization of Ullmann's algorithm [WWR91]. Their work details two parallel algorithms, the first performing pair-wise graph tests, the second comparing a query with all graphs in the database simultaneously. The matrix operations are readily parallelizable. Their parallelizations are for a bit-serial SIMD array processor, the DAP.

Willett *et al.*'s "Algorithm 1" uses the inner-loop parallelism of the refinement matrix operations. Algorithm 1 distributes the adjacency representation of the graphs across the SIMD array, each PE containing one bit from each of the matrices. The array control unit, corresponding to a MISC supervisor, executes the backtracking main body. On a sequential machine, Ullmann's refinement algorithm exploits bit-level parallelism in a conventional ALU, requiring $O(n^3)$ time to compare two graphs, where n is the number of nodes in the graphs, whenever the number of bits in the ALU is greater than or equal to n. The SIMD arrays perform only the refinement portion of the algorithm, processing the A, B, and M matrix elements simultaneously using n^2 PEs, reducing total time to $O(n)$. Although this comparison ignores the ability of the sequential version to take advantage of matrix sparseness, speedup is substantial even for small graphs.

Willett *et al.*'s "Algorithm 2" uses outer-loop parallelism in comparing the query against multiple graphs in a DB of graphs. One complete graph is stored in each PE and the SIMD PEs execute the full Ullmann algorithm by blocking operations in selected PEs. Comparison time is proportional to the size of the largest graph in the DB, so that a significant portion of the time is spent waiting for comparison of a small percentage of large graphs to be completed. Further, due to the single control stream, conditional operations must be executed by enabling PEs based on the conditional test. As a result, on average only 10% of the PEs are active at any given time [WWR91]. On a collection of 4,096 graphs (sampled from a chemical database) having an average size of 22 nodes, search of the full DB with query graphs averaging 12 nodes was two times slower for Algorithm 2 on N PEs (where N is the number of graphs) than for Algorithm 1 on n^2 PEs [WWR91].

Willett *et al.* also present modifications in which Algorithm 2 is used until only some number of graphs are not complete; these remaining graphs are then processed individually using Algorithm 1. This gives a total DB search time half that of Algorithm 1 alone. They assert that having fewer than N PEs, pulling in new graphs once a PE is free, and improvements in the looping and control functions would result in Algorithm 2 performing about ten times faster than Algorithm 1 on an exhaustive DB search. See [WWR91] for a more complete discussion of these implementations of Ullmann's refinement, with more detailed benchmark results.

A New Parallelization for Multi-Bit SIMD Processors

Recent machines feature wide data words in each processor, typically 16–32 bits. Since nearly all bit operations are independent of one another at any step of the refinement, it is possible to assign as many bits to a processor as it can handle simultaneously rather than place a single bit of each matrix in each processor. This increases speed by making much better use of the available resources and permits simultaneous comparison of many graphs even on a relatively small number of processors.

The following description involves a simplifying assumption for the purpose of discussion. We assume that the number of bits in the processor data word is at least as large as the number of nodes in a graph. Otherwise each row of the matrices must reside in multiple processors. The implementation details that arise can be dealt with in a straightforward manner, primarily by additional communication steps during matrix multiplication.

Subgraph Isomorphism Adapting the SI refinement algorithm to multi-bit processors, we combine the features of Willet *et al.*'s different methods and achieve additional benefits. In the new parallelization, we assign an entire row of each of the matrices to a processor and execute the refinement algorithm for many graphs on a single array, giving a group of processors a new graph once they have completed their prior SI test. We exploit the bit-parallelism of b-bit processors so that a given SI test requires $1/b$ as many processors with no slowdown. Having fewer processors and the full matrix row in a single processor greatly reduces communication time. As with Willett *et al.*'s first method, having multiple processors per graph speeds refinement. As with their second method, having several graphs on a single array minimizes wasted processors. A load-balancing approach, described in Section 5.3.4, is similar to their third and fourth methods and avoids the poor utilization which would result if the algorithm was to wait until all the graphs finish before proceeding to the next group of graphs. Using lattice codes of the concept-type hierarchy avoids the bottleneck of all processors consulting a single central hierarchy and allows forming all rows of the match matrices of all graphs simultaneously.

Figure 5.10 presents the SEGUL pseudocode for forming the M matrix and refinement.

Managing Graph Storage and Multiple Refinements Since only $O(\lg^2 N)$ graphs are actually tested, where N is the total number of graphs in the knowledge base, each processor stores a large number of graphs in off-chip memory but processes only a tiny fraction of them. Although during computation each processor is assigned a single row of a graph's matrices, each processor's memory holds the full adjacency matrix, node subsumption codes, and relation type labels for a complete graph. Graphs are assigned to processors for storage randomly, and the odds of a conflict in spreading the graph to nearby processors is small. If there is a conflict, the additional graph could remain on the queue of candidates or be moved to an array section that has no graph.

Executing multiple SI tests on a single array introduces new complexities. Multiple graphs must be assigned to different sub-sections of the array; the backtracking portion of the algorithm must be performed in parallel on the SIMD array rather than once in the supervisor, and load balancing is necessary. The latter is discussed in Sections 5.3.3 and 5.3.4.

```
/******** Data Declaration ********/
int a ← | $G_a$ |, b ← | $G_b$ |
nonunique [ nonunique[code(a)] a] A
nonunique [ int a] arels /* number of relations for each node in $G_a$
nonunique [ subsumptioncode a] acode /* for each node in $G_a$
nonunique [ nonunique[code(b)] b] B
/* brels in supervisor */
/* bcode in supervisor */
nonunique [ nonunique[code(b)] a] M
nonunique [ nonunique[code(b)] a] MC

/******** Form the M Matrix ********/
procedure FORMM($G_b$, arels, brels, acode, bcode):
    foreach nodex ∈ $G_b$
        M(;nodex) ← 1
        if arels(nodex) $\not\leq$ brels(nodex)
            M(;nodex) ← 0
        else if acode $\not\leq$ acode
            M(;nodex) ← 0
;
/******** Refinement Procedure ********/
procedure REFINE(A, B, M, MC):
    while TRUE
        MC ← M ∧ ¬(A × ¬(M × B))
        if MC = M
            return(M, UNKNOWN)
        if [x ∈ MC | M(x;) = 0] ≠ []
            return(MC, FALSE)
        M ← MC
;
```

Figure 5.10: SEGUL Pseudocode for Forming M and Refinement

In the new parallelization a KB graph is stored in a single PE as [node[subsumptionCode, adjMatrixRow]...] and each PE stores many graphs in off-chip memory. Graph loading and SI management proceed as follows, with each step done simultaneously for all graphs (array sections). Typically 16 to 64 graphs can be tested simultaneously (query graph 32 to 8 nodes, respectively).

1. Divide the array into equally sized subsections (coteries) based on the size of the query graph.[8] Each section is $\lceil |G_b|/b \rceil$ PEs wide and $|G_b|$ PEs tall, where $|G_b|$ is the size of the query graph.
2. Select one KB graph per section from the PEs contained in that section based on the

---

[8] The query graph is larger than any of the database graphs to be compared.

progress of the search through the partial order.[9]

3. Configure a coterie for each array section in the array local communication network. Distribute the KB graph nodes so that a PE has one node's subsumption label and the corresponding row of the adjacency matrix (in on-chip registers).[10]
4. Form the M match matrix for each graph, storing 1 row of the resulting matrix per PE.
5. Iterate refinement until all graphs have either failed or their M matrix has not changed since the last iteration.

5.3.3 Refinement Behavior

SI refinement behavior must be characterized in order to project characteristics of a full graph knowledge base as well as to consider load balancing and integration with multi-level indexed search. A probabilistic analytical model for predicting the likelihood of a bit in the M match matrix changing from 1 to 0 (and in turn the probability of an SI test failing or succeeding) based on the densities of the A, B, and M matrices would be overly complex due to the dependencies between the bits of the matrices. We have thereby run simulations (using 2MISC) to characterize refinement behavior, independent of any specific architecture. The knowledge base and query graphs are randomly generated connected graphs with no self-cycles.

The Experiments

Behavior is highly dependent on the synthetic data parameters. Results include several experiments varying parameters as shown in Table 5.3. As before G_a is the KB graph and G_b is the query graph and $p(X)$ indicates the probability that a bit in adjacency matrix X is one. The first experiment ("small") is for a small graph, the second ("cg") roughly approximates a small conceptual graph, the third ("chem") roughly approximates the chemical database used by Willett *et al.* [WWR91], and the fourth "large" a large graph. The second and third experiments are for relatively sparse graphs, the specified densities resulting in 1–2 edges per node; the others average 2–9 edges per node. Figure 5.11 plots the portion of graphs remaining to be processed (neither failed nor unchanging) at each iteration averaged over 4 queries and Figure 5.12 plots the cumulative number of graphs that have failed. All the examples show similar behavior in the number of iterations needed to complete most of the graphs, and only the small example shows a major difference in the total number of graphs that fail.

These experiments only approximate application behavior. The adjacency and match matrices are generated randomly using the same procedure used by Ullmann to evaluate his algorithm [Ull76] and do not capture any structures common to an application domain.[11]

---

[9] With random assignment of graphs the probability that any array section contains more than one graph to process is less than 5%. This is ignored in the analysis.

[10] In this description we ignore the possibility that the number of bits in the adjacency matrix row may require multiple words and would be stored on multiple PEs. This is included in the simulation.

[11] A probabilistic analysis of the evolution of the M matrix over refinement iterations failed to predict the observed behavior due to the assumption of independence.

Experiment	$\mid G_a \mid$	$\mid G_b \mid$	$p(A)$	$p(B)$	$p(M)$
small	8	12	0.20	0.30	0.50
cg	16	24	0.13	0.15	0.40
chem	8–19	19–25	0.10	0.10	0.45
large	24–26	30–31	0.15	0.30	0.25

Table 5.3: SI Refinement Experiment Conditions

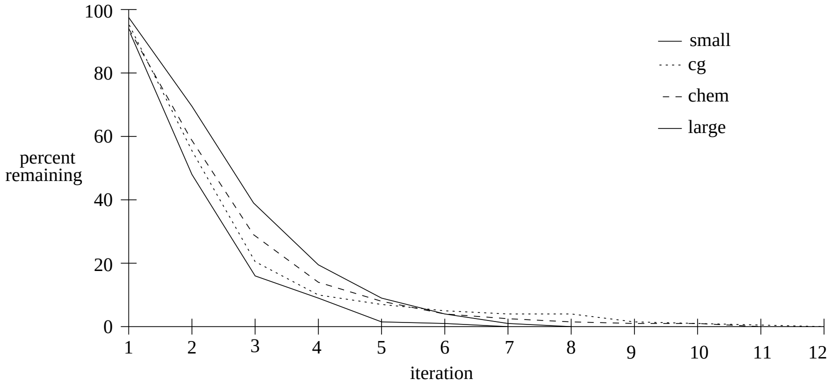


Figure 5.11: SI Refinement: Graphs Remaining

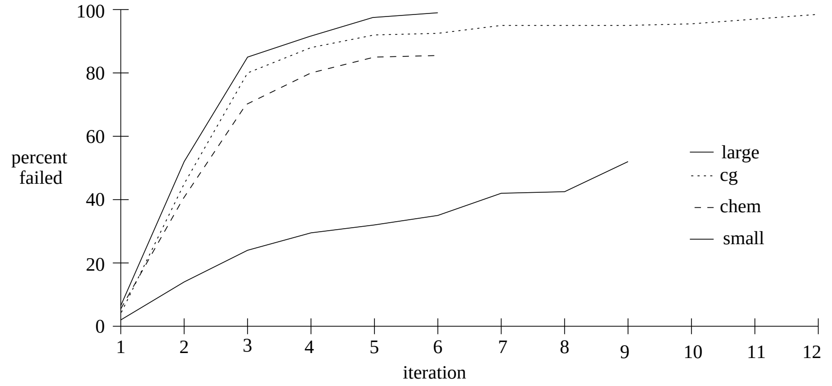


Figure 5.12: SI Refinement: Cumulative Graph Fails

Further, in the context of a Method III KB, the graphs selected for comparison are more likely to pass the SI test due to the filtering effect of the multi-level indexed search. The latter is approximated in the experiments by the large values for $p(M)$. This is particularly appropriate for the "chem" example, as most atoms in an organic chemistry database are either carbon or hydrogen.

Changes to the probabilities in Table 5.3 often result in all graphs failing within one or two refinement iterations. As examples, this is the case for the "small" experiment changing $p(A)$, $p(B)$, and $p(M)$ to 0.20, 0.30, 0.20 respectively (changing the probabilities to 0.10, 0.35, 0.50 has negligible effect) as well as for the "cg" example with probabilities changed

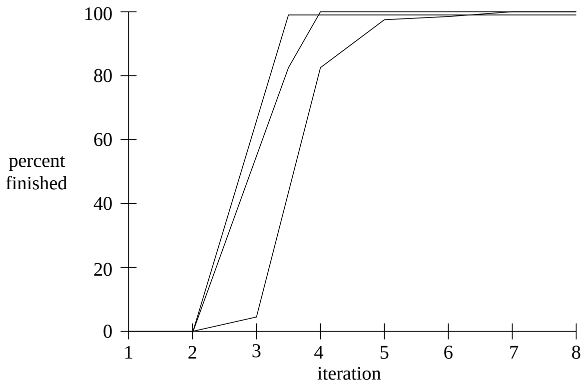


Figure 5.13: SI Calls to Refinement (after [WWR91])

to 0.13, 0.15, 0.20 and the "large" example with probabilities changed to 0.20, 0.20, 0.10. On the other hand, increasing $p(M)$ to 0.60 in the "cg" example results in as many of 17 iterations to complete all graphs and less than 50% (typically 30%) fail. For the "large" example, changing the probabilities to 0.10, 0.40, 0.40 results in none of the graphs failing and only 2–4 iterations required until all M matrices stop changing. The experiments of Table 5.3 are intended to represent a middle ground typical of applications.

Implications for a Graph Knowledge Base

Examining the experiments individually, the average portion of graphs being processed ranges from 17% to 46% (averaged over all iterations), and is typically about 30%. This is a direct indication of low PE utilization. Although this level of utilization is not unreasonable, it suggests that *load balancing* may be appropriate within refinement iterations (particularly after the first 3–4 iterations) rather than at just the backtracking or at Method III candidate graph queue updates. The simulations show, however, that a refinement iteration requires fewer than 1,000 cycles whereas distributing a graph and forming the M matrix requires roughly 2,000 cycles. PE utilization improvement can be effectively achieved by iterating until very few graphs remain then holding them over into the next round of graph comparisons. This may apply at the refinement, backtracking, and candidate graph queue levels.

SI Behavior

Thus far we have discussed refinement iterations. Willett et al.'s experimental data for exhaustive search in a chemical database shows that nearly all SI tests are resolved in 3 to 7 calls to refinement (each of which includes numerous iterations) with a few (less than 1%) requiring 20–50 calls. Load balancing at each SI completion is also attractive.

5.3.4 Hierarchical Retrieval

Here we detail multi-level indexed search and discuss how it may be integrated with the multi-bit multi-graph parallel SI algorithm.

Method III

Levinson's Method III multi-level hierarchical retrieval proceeds in two phases. In the first phase, the partial order of graphs is traversed seeking the immediate predecessors (closest generalizations) of the query graph. Smaller graphs are thereby used to avoid many comparisons with larger graphs, since for any graph that is not a generalization of the query, none of its specializations can be. In the second phase, immediate successors (closest specializations) of the query are found by intersecting the subhierarchies (immediate successors) of the query's immediate predecessors and performing a breadth first search. These two phases are also used to locate the position in the partial order for inserting a new graph. Empirically, only $O(\lg^2 N)$ of the total of N graphs in the database and analysis suggests that on average $O(\lg N)$ graphs are available for comparison at any time without risking unnecessary comparisons. Pseudocode for a processor-farm parallel implementation is presented in Figure 5.14.

Method III and Multi-Bit Parallel SI

With the multi-bit algorithm described in Sections 5.3.2 and 5.3.2 it is necessary to reconsider the approach to parallelizing MIS. Previously, a parallel array of a thousand processors could handle only one to a few graphs with many more ($O(\lg N)$) available at any stage in the MIS, comparisons would be farmed out to several processor arrays [HLR93] exactly as in Figure 5.14. This processor-farm model included implicit load balancing (each array requesting more work when it was done) and there were more graphs queued for comparison than the available processors could handle so that a graph need not be compared until it was certain that the comparison was necessary. On the minus side, communication is high as the full knowledge base graph and concept-type subsumption data had to be sent from wherever the MIS management was executing. Although having all graphs on a single parallel array significantly reduces communication outside the array and the use of concept-type subsusumption codes greatly reduces communication within the array, the new implementation affects the implementation of the multilevel indexed search itself. With 32-bit processors, an array of 1024 processors could handle thirty 32-node graphs simultaneously, but at any given time the MIS algorithm would have identified only one to two dozen graphs requiring comparison. One must thereby choose between not using all the processors or speculatively comparing graphs.

Performing speculative graph comparisons using an earlier version of Phase I of Levinson's Method III appears to be the better choice. In the more recent implementations of Method III, the queue of candidate graphs begins with only the top of the poset (generalization hierarchy) and candidates are added as the poset is traversed, limiting not only the number of SI tests but also the number of graphs "visited." By initially enqueueing all graphs smaller than the query graph and then dequeuing them if a predecessor fails the older version guarantees a large queue of candidate graphs. Although there will be some graph comparisons that are later determined to be unnecessary, this is expected to give a faster parallel implementation as far fewer processors would be idle. It should also be possible to integrate the MIS dequeuing with the refinement load balancing; if a currently executing graph is dequeued it can be abandoned and another graph started. The SIMD array can also handle much of the queue and other MIS management.

```
$$----------PHASE I:  FIND Qinsert(IPs)-----------$$
$$---SUPERVISING PN---$$
X = { x | x(dbGraph,size) <= Q(size) }
Qinsert(IPs) = {}
foreach x in X $enumerating in order small->large
     if SUBGRAPH(x,Q) $$send (x,Q) to a free PN
          Qinsert(IPs) = Qinsert(IPs) + x
                       - forall x(IPs)
     else X = X - forall x(ISs)
$$---PNs---$$
    whenreceive[:thisPN,?D,?Q]:
      if SUBGRAPH(D,Q) send[:supPN,'succeed',DId]
      else send[:supPN,'fail']
;

$$-----------PHASE II:  FIND Qinsert(ISs)---------$$
$---SUPERVISING PN---$
y = Q(IPs,1)   $$smallest
$$intersection of the subhierarchies of Qinsert(IPs):
foreach v>1 in Q(IPs)
     I = I INTERSECT foreach w in v(ISs)
$---BFS of successors of y---$
Q(ISs) = {}
foreach u in y(ISs)
     if (u in I) AND SUBGRAPH(Q,u)
          Qinsert(ISs) = Qinsert(ISs) + u
          foreach t in u(ISs) $$all parallel so no prune
               I = I - t
$PNs$
$same as in Phase II
```

Figure 5.14: Parallel Method III Pseudocode

5.3.5 Compiling Hierarchies

Partial orders, such as the concept-type (node) hierarchy in CGs and the generalization hierarchy in Method III, can be represented as graphs themselves, and pre-processed to encode the partial order in order to speedup operations on the hierarchy. The ordering relation is typically subsumption for the node-type hierarchy and more-general-than for the generalization hierarchy.

The resulting codes can be used in a variety of operations, avoiding link (relation) traversal. Subsumption testing between concept types is one of the most frequent operations in graph retrieval, performed numerous times in each projection test whenever node subsumption is allowed (such as when "animal" is allowed to match "dog"). If the above encoding is not performed, time is $O(N)$ in the worst case and $O(\lg N)$ in the expected case, in comparison $O(1)$ for the encoding or equivalently for hashing. Using partial-order encoding, greatest lower bound, least upper bound, and relative complementation between any two graphs can also be quickly determined. Ellis has shown that these operations can in turn be used to further prune graph comparisons, provided the "query" graph is known to be in the knowledge base, as described in detail in [Ell93]. Otherwise poset codes can prune the Method III search by considering only those graphs in the intersection of known predecessors of the query graph [ELR95].

On a parallel machine, a lattice code can be simultaneously compared against numerous others. This permits operations such as identifying all descendents of an element, or even the union or intersection of the descendents of several elements in time proportional to the length of the codes times times the number of root elements. With full data parallelism, time for these code operations are independent of the number of elements in the poset or the length of the paths.

Transitive-Closure and Related Codes

For transitive ordering relations (such as inheritance), the reflexive-transitive closure (RTC) of the adjacency matrix representation of the partial order encodes all the information. Each row of the RTC matrix can be used as a code for the corresponding node, as shown previously in Figure 2.5 (page 20), and the recursive application becomes a simple logical operation [AK$^+$89]. For example, subsumes(X,Y) becomes a masked match on the codes for X and Y. Since the transitive closure is the minimum information required to fully characterize the transitive relations, alternative codes can be viewed as compressing only the representation, not the underlying information.

Computing the transitive closure is often presented in terms of raising the adjacency matrix to successive powers. This can be revised to a reflexive-transitive closure by including the diagonal in the adjacency matrix. Although the matrix multiplications can be readily parallelized, this is not the most computationally efficient approach. An n-node graph results in an $n \times n$ adjacency matrix, and a *single* matrix multiplication requires $O(n^3)$ time using the simplest algorithm on a sequential machine. The best known sequential matrix multiplication algorithm requires $O(n^{2.376})$, but with higher constant factors [CLR90]. The Floyd-Washall Algorithm provides significant improvement. Floyd-Washall is an incremental dynamic-programming formulation of the all pairs shortest path graph problem, which can be modified to compute the transitive closure of the graph in total time $O(n^3)$ [CLR90]. The inner loop of the algorithm can be parallelized using n PEs, giving time $O(n^2)$.[12] Additionally, the inner loop requires only one row of the matrix and a global scalar. Thus it is possible to distribute computation of the transitive closure, as well as storage of the resulting codes, over multiple PNs. We modify the algorithm for connectivity rather than shortest paths; all paths are treated as length 1 and we need only mark that one node has been reached from another rather than tracking the path length.

It is often highly desirable to add new nodes without the computational expense of re-encoding the entire graph. This can be accomplished as follows, where input is a new node, all its parents, and all its children in the hierarchy. With SIMD parallel processing and

---

[12] The matrix multiplication method can be parallelized to give $O(n^2)$ total time for transitive closure, but this requires n^2 PEs.

RTC coding, ancestors can be marked and their codes logical-ored in parallel by matching a 1 in the bit position corresponding to a parent and performing the logical-or in all PEs that matched.

1. Assign a new bit position, b, in the codes for the new node
2. Logical-or the codes for all the children, to form the for the new node.
3. For each parent, locate all the ancestors and logical-or their codes (including the parent itself) with the code for the new node.

Code length is a major disadvantage for the basic RTC codes. Each code is N bits long, so that total storage in $O(N^2)$. Further since N will often be orders of magnitude larger than the machine word, time is $O(N)$. More advanced coding algorithms approach the lower bound of $\Omega(\lg N)$ length codes for ideal topologies. In addition to reducing storage, shorter codes reduce execution time. The time to compute subsumption, greatest lower bound, or least upper bound between two nodes in the poset is proportional to the length of the code.

It is straightforward to prove that where the encoded graphs are balanced binary trees, each code in the basic transitive closure coding has on average $O(\lg N)$ bits equal to 1, giving $O(N \lg N)$ space for a sparse-matrix coding. For concept-type subsumption tests, the sparse connectivity matrix can be converted into a hash table [CLR90].

Proof

1. A balanced binary tree of N nodes has $O(\lg N)$ levels. Number them $0 \ldots \lg N$.
2. A code at a given level of the tree inherits the 1s of its ancestors, and adds a single 1 to its code position, giving $L + 1$ ones for each code at level L.
3. Each level has 2^L nodes, so that the total number of 1 in all the node codes combined is

$$\sum_{L=0}^{\lg N - 1} (L+1)2^L = N \lg N - N + 1 = O(N \ln N).$$

□

Ait-Kaci presents an extension to transitive-closure coding that recursively groups and encodes clusters in the partial order [AK$^+$89]. In this method, codes are $O(\lg N)$ bits in length. The asymptotic space reduction is significant; with 64 K graphs, modulated encoding requires over 600 times less storage. Operations on the codes to obtain greatest lower bound (GLB), least upper bound (LUB), and other functions then require $O(\lg N)$ time. More importantly, there are restrictions on links to and from each group so that the method is not suitable for all applications and no empirical work has been done for knowledge base domains.

Tree Codes

Stormon presents an encoding for binary trees which can be used for determining subsumption or inheritance relations (connectivity), marking subtrees rooted at a particular node, and finding parents and siblings [Sto91]. As shown in Figure 5.15 for a complete balanced binary tree the codes correspond to an ordered numbering of the nodes, starting at the root. The resulting binary values are left-justified and padded on the right with "match-anything," a type of trinary logic equivalent to masked compare. Codes are $\lg N$ "trits" (trinary bits) long even in the case of unbalanced binary trees. Since the resulting codes are of $O(\lg N)$ length and functions are implementable by simple bit-wise operations, time is effectively $O(1)$.

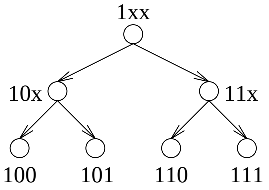


Figure 5.15: Tree Coding

Geller presents coding for trees in which each node is represented by an integer-valued interval [Gel94a, Gel94b]. A node subsumes its descendants by having an interval that covers those of all its descendents. These codes can be efficiently created, and Geller gives algorithms for incrementally updating the coded tree in constant time. Operations on the codes appear to be limited to inheritance, however, and like Stormon's coding have not been extended beyond tree structures.

Other Codings

Although useful in many applications and algorithms, including simple inheritance hierarchies found in some semantic network applications, tree coding schemes will not work on partial orders or lattices due to multiple inheritance: a given node may have more than one parent. Ellis presents a lattice encoding method using "match-all-below" variables and "constants" [ELR95]. Through the match-all and re-use of constants, code lengths are empirically of length $O(\lg N)$.

Operations Using Hierarchy Codes

For transitive closure codes, operations such as greatest-lower-bound (GLB) and least-upper-bound (LUB) are simply logical-and or logical-or operations between the codes, provided the GLB or LUB exist. Otherwise a decoding function can return the appropriate set of nodes [AK^{+}89]. In Stormon's coding a parent can be "named" by replacing the last 0 or 1 in the code with don't-care, and a sibling can be named by flipping the last 0 or 1.

Here we show that subsumption on Stormon's coding and the coding used by Ellis can be implemented in simple bit-wise logical operations in effectively constant time (where the machine word is at least as long as the code length). The truth table for the subsumption function is shown in Table 5.4. The match-all-below is shown as X and the constants as binary integers. Note that the function is not commutative. The values in parenthesis show a binary representation of the trinary digits X, 0, and 1. This can be used as the basis for representing the codes on a conventional machine. Let the first bit of A be denoted as a_1 and the second as a_2 and let the first bit of B be b_1 and the second b_2. We see that when a_1 is true the result is always true (acting as a match all below) and that when b_1

is true the result is always false (acting as match none above); otherwise the result is the equivalence operator exclusive-**nor**. Each code can now be represented as two binary words, the first concatenating a_1 or b_1 and the second a_2 or b_2. It is necessary to pad both so that the number of bits in each code position is sufficient to represent the largest constant. For example, the code X,1,3,X becomes 2 binary words $A_1 = [11000011]$ and $A_2 = [00011100]$. Simplifying the bitwise operations of the 1-bit truth table to $a_1 \vee a_2 b_2 \overline{b_1} \vee \overline{a_2 b_2 b_1}$, the logic equation can now be implemented in a conventional ALU as the standard logic operations

$$\begin{aligned} C &\leftarrow A_a \odot B_b \\ C &\leftarrow C \wedge \overline{B_\beta} \\ C &\leftarrow C \vee A_\alpha . \end{aligned}$$

It is now possible to form the intersection of the descendents of two nodes in a tree or partial order in effectively constant time on a SIMD array, as shown in the pseudocode of Figure 5.16; an intersection can be taken in constant time using a bit-set representation with full data parallelism over the basis element. Other parts of the partial order data structure are not shown.

A	B	$A \preceq B$
X (10)	X (10)	1
X (10)	0 (00)	1
X (10)	1 (01)	1
0 (00)	X (10)	0
0 (00)	0 (00)	1
0 (00)	1 (10)	0
1 (01)	X (10)	0
1 (01)	0 (00)	0
1 (01)	1 (01)	1

Table 5.4: Lattice Code Element Truth Table

```
/* type [code alph, code a] subsetcode */
subsetcode node1, node2
nonunique {subsetcode N} posetnodes

descendants1 ← {x ∈ posetnodes | $x \preceq node1$}
descendants2 ← {x ∈ posetnodes | $x \preceq node2$}
result ← descendants1 ∪ descendants2
```

Figure 5.16: Intersection of Descendents

5.3.6 Graph Retrieval Characteristics

The graph retrieval manual profiling is based on a synthetic "application" an order of magnitude larger than current implementations. The application is a 32 K graph associative database built by 32 K queries with inserts, followed by 32 K additional queries. Graphs average 16 nodes and 32 edges with 4 immediate predecessors and 4 immediate successors each in the generalization hierarchy. The profiling includes a node-type subsumption hierarchy consisting of 16 K types, the first half of which are generated by the Floyd-Washall algorithm, the second half by the incremental algorithm. The number of subgraph isomorphism tests per query is taken as $1.3 \lg^2 N$ (the constant factor empirical over a test database [ELR95]), so that the total number of SI tests is

$$1.3(\sum_{i=1}^{N} \lg^2 i + N \lg^2 N) \approx 17 \times 10^6$$

As Ullmann's algorithm is empirically $O(n^4)$ and refinement alone is $O(n^3)$ bit operations, the analysis assumes n calls to refinement for each SI test. Table 5.5 summarizes the results for associative array operations and global communication. Boolean matrix multiplication occupies the majority of execution time. Global communication is minimal; with the KB on a single array external communication is simply the query graph and identifiers for the matched graphs.

Operation	Time
Boolean matrix multiplication	67%
logical operations	19%
general ALU	11%
global communication	3%

Table 5.5: Projected Breakdown of Graph KB Execution Time

Multi-level indexed search directly prunes graph comparisons. With the hypothesis that $O(\lg N)$ SI tests can be performed simultaneously with negligible unnecessary work, the optimum number of processors is $O((n^2/b) \lg N)$, giving time per query also $O(n^2 \lg N)$. There is a logarithmic slowdown in N and a quadratic slowdown in n. This means that the number of processors required is only logarithmic in the number of graphs (or other objects) in the knowledge base; or from another perspective, the potential size of the KB is exponential in the number of PEs. On the other hand, the possible number of graphs is inversely exponential to the square of the average number of nodes in each graph. Due to the sub-linear scaling of P (the number of PEs), the memory per PE grows superlinearly, posing a significant design challenge.

5.4 Full Data Parallelism versus Pruned Search

This section compares the speed of exhaustive search with full data parallelism against search with pruning techniques, summarizing the author's prior work [HLR93]. Considering the massive data parallelism of SIMD processing, it at first seems intuitive that placing one graph per PE would be the fastest. All graphs are compared simultaneously, and any pruning heuristics would mask out certain PEs but would not reduce time to completion. On further thought, it appears better to distribute N graphs over N/p PEs, where p is the heuristic pruning factor, so that after pruning each PE has one graph to compare. Neither is the case, however, as an additional level of parallelism is available. The analysis is specific to the graph retrieval example, but the author believes the results are applicable in most cases where some form of pruning or filtering is available in conjunction with multiple levels of parallelism.

In the following analysis we ignore the number of calls to refinement required for a subgraph isomorphism test as well as the number of bits in the processors. Although exponential in the worst case, Ullmann's full subgraph isomorphism algorithm requires $O(n^4)$ bit operations in the expected case, where n is the number of graph nodes [Ull76] and each call to the refinement algorithm has time $O(n^3)$ on a serial machine [WWR91]. In comparing different parallelizations, the expected-case n calls to refinement appear in all cases and thereby cancels out. By processing multiple bits in a b-bit processor, a factor of n is reduced to $\lceil n/b \rceil$. Again, this affects all algorithms and can thereby be formally ignored.

In Willett *et al.*'s parallel benchmarks with graph sizes similar to those in CGs (mean values of 22 node graphs, 10 node subgraphs) refinement occupies 95% of execution time, and thus we only consider refinement time. Extrapolating the benchmark results for Algorithm 1 and 2 with their proposed improvements and MISC's SIMD enhancements[13] and adjusting for the relative parallelisms, constant factors are nearly identical and are ignored in our breakpoint estimates.

We now turn to the analyses, first considering a bounded number of processors and then an unbounded number.

5.4.1 Bounded Number of Processors

Pruning With a Single Level of Parallelism

On a sequential machine exhaustively searching a CG knowledge base, total refinement time is $O(Nn^3)$. Algorithm 1 uses n^2 parallelism whereas Algorithm 2 uses N parallelism. With similar constant factors, whichever algorithm has the greatest parallelism will be fastest, so Algorithm 2 (exhaustive search) is faster whenever $N > n$. For graphs of a dozen or so nodes, exhaustive search will be faster for knowledge bases as small as a few hundred graphs. Algorithm 1, however, can use pruning heuristics. We have presented search methods which have empirically been shown to reduce the number of SI tests in a knowledge base search to $O(\lg^2 N)$. With this pruning, Algorithm 1's time would be $O(\lg^2 Nn)$, a speedup of $O(Nn^2/\lg N)$. Algorithm 2's exhaustive search would now be better when

$$N > \frac{N}{\lg^2 N} n^2, \text{ or } \lg^2 N > n^2 \ .$$

Constant factors for the two algorithms are similar, so that with the above simplifications, Algorithm 2 is better whenever N is greater than 1,000 or so for 10-node graphs.

---

[13] Ignoring MISC's SIMD enhancements would further favor Algorithm 1 and pruning.

Pruning With Multiple Levels of Parallelism

It is possible to exploit both the inner-loop parallelism of Algorithm 1 and the outer-loop parallelism of Algorithm 2. We now take a closer look at three conditions, and consider differing assumptions of processor availability:

1. N PEs, permitting full data parallelism for Algorithm 2.
2. A number of PEs such that each has approximately one graph to compare after pruning.
3. Each method has the maximum number of PEs it can use effectively.

The results are summarized in Table 5.6.

For the first condition with N PEs, we compare exhaustive search using Algorithm 2 with a pruned search using Algorithm 1 and multiple grains of parallelism. Algorithm 2 parallelism remains N and execution time is $O(n^2)$. Algorithm 1 uses n^2 PEs for fine grain parallelism in matrix operations, so that as many as N/n^2 graph comparisons could be made simultaneously (coarse grain parallelism). With pruning, only $\lg^2 N$ comparisons are necessary, limiting coarse parallelism, and total speedup is

$$O(\min(\frac{N}{n^2}, \lg^2 N)\frac{N}{\lg^2 N}n^2).$$

This gives us the first two cases of Table 5.6. In Case 1a N/n^2 is the minimum, and in Case 1b $\lg^2 N$ is the minimum. Regardless of the minimum, exhaustive search (full data parallelism) is never faster. Note also that with both minima the size of the CGs and the exact pruning factor do not affect the conclusion in any practical circumstances.

For the second condition, given the pruning speedup applied to both Algorithm 1 and 2, and using $\lg^2 N$ rather than N PEs, Algorithm 2 would perform an average of one graph comparison per processor after pruning, at $O(n^3)$ time. Without coarse grain parallelism, Case 2a, Algorithm 1 would have $O(\lg Nn)$ time, giving the same result as the initial analysis which ignored the number of PEs and looked only at relative parallelism. When coarse grain parallelism is available, Case 2b, total speedup is $O(\lg^2 N/n^2)$, so that combined parallelism is $O(\lg^2 N)$ and time is the same for both approaches. Note that there are not enough PEs for Algorithm 1 when $n^2 > \lg^2 N$. This is not a concern for practical cases, but would be an issue were pruning asymptotically better than $O(\lg^2 N)$.

For the third condition, Case 3a has Algorithm 2 performing exhaustive search with $O(Nn^2)$ processors and Algorithm 1 performing pruned search with $O(\lg^2 Nn^2)$ processors. Both Algorithms have execution time $O(n)$ under these conditions. Algorithm 1 is preferable because it uses dramatically fewer processors, it is preferred. If pruning is applied to both algorithms, Algorithm 1 will have time $O(n)$ and Algorithm 2 time $O(n^3)$. Algorithm 1 with pruning will again always be faster than exhaustive search.

The above analysis, summarized in Table 5.6, makes the flaws in the original intuition apparent. Algorithm 1 can use n^2 additional processors for an additional n^2 speedup, and can employ an additional level of parallelism. Thus, at the expense of a factor of n^2 more PEs, pairwise comparison can always be faster than exhaustive search with full advantage taken of pruning and similar CG retrieval advances. When both algorithms are restricted to the same number of processing elements, Algorithm 1 with pruning is asymptotically

	Methods	Number of PEs	Exhaustive Better
Case 1a	exhaustive, pruned	N, N	$N > \frac{N^2}{\lg^2 N}$
1b			$N > Nn^2$
Case 2a	pruned, pruned	$\lg^2 N$, $\lg^2 N$	$\lg^2 N > n$
2b			equivalent
Case 3a	exhaustive, pruned	Nn^2, $\lg^2 Nn^2$	equivalent
3b	pruned, pruned	$\lg^2 N$, $\lg^2 Nn^2$	$\dot{n} > n^3$

Table 5.6: Data Parallelism versus Pruned Search Speed

better than the exhaustive search with one PE per graph. Even when pruning is used for both algorithms, Algorithm 1, exploiting both fine- and coarse-grain parallelism, is asymptotically as good as Algorithm 2.

5.4.2 Unbounded Number of Processors

Consider the space-time product of full data parallelism versus pruned search with an unbounded number of processors. For exhaustive search, we need at most Nn^2 PEs. Since edges must be represented, memory is also $O(Nn^2)$. Table 5.7 summarizes the space-time product, where space is the number of processors, and includes a comparison to serial execution of multi-level indexed search. Although the resulting parallel exhaustive search is much faster than serial MIS by a factor of $O(\lg^2 Nn^2)$, the $O(n^2)$ speedup is from parallelizing individual subgraph isomorphism tests and isn't relevant in evaluating MIS. The space-time product of exhaustive search is asymptotically much worse than for serial MI. Comparing parallel MIS to serial MIS, the parallel implementation has the same space-time product but is $O(\lg Nn^2)$ faster. More importantly, we now see that for parallel processing pruned search with an unbounded number of PEs is preferable over exhaustive search except for speed-critical applications; an $O(\lg N)$ speed penalty is paid but the number of PEs is reduced by $O(N/\lg N)$ for a net space-time improvement of $O(N/\lg^2 N)$. Where constant factors are not significantly different, full data parallelism with a million PEs will be slower than pruned search with a few thousand PEs.

Method	PEs	Time	Space-Time
Serial			
Exhaustive	$O(1)$	$O(Nn^3 \lceil n/b \rceil)$	$O(Nn^3 \lceil n/b \rceil)$
MIS	$O(1)$	$O(\lg^2 Nn^3 \lceil n/b \rceil)$	$O(\lg^2 Nn^3 \lceil n/b \rceil)$
Parallel			
Exhaustive	$O(Nn \lceil n/b \rceil)$	$O(n^2)$	$O(Nn^3 \lceil n/b \rceil)$
MIS	$O(\lg Nn \lceil n/b \rceil)$	$O(\lg Nn^2)$	$O(\lg^2 Nn^3 \lceil n/b \rceil)$

Table 5.7: Data Parallelism versus Pruned Search Space-Time (PEs)

Early analysis indicates that with MIS, memory will dominate cost in a MISC Machine. It is thereby appropriate to include memory in the space-time product, as shown in Table 5.8. In all cases memory asymptotically dominates the number of PEs, and space-time differences are purely the result of time differences. Here space-time for exhaustive parallel search is a factor of $O(\lg^2 Nn^2)$ better than serial MIS and a factor of $O(\lg N)$ better than parallel MIS. It is necessary to consider the practicality of machines with huge numbers of PEs. A CG database of 64 K graphs averaging 16 nodes each, a near-term application target, would require 16 million PEs for exhaustive search. In the long term, KBs with one million graphs are anticipated, and with 32-node graphs would require billions of PEs. Even with advanced machines of one million PEs, we can still expect high virtual-processor ratios in the long term. From the asymptotic analysis, the virtual-processor ratio for exhaustive search must be less than $O(\lg N)$ to have better space-time product than pruned search under the memory analysis.

Method	Memory	Time	Space-Time
Serial			
Exhaustive	$O(Nn^2)$	$O(Nn^4)$	$O(N^2n^6)$
MIS	$O(Nn^2)$	$O(\lg^2 Nn^4)$	$O(N\lg^2 Nn^6)$
Parallel			
Exhaustive	$O(Nn^2)$	$O(n^2)$	$O(Nn^4)$
MIS	$O(Nn^2)$	$O(\lg Nn^2)$	$O(N\lg^2 Nn^4)$

Table 5.8: Data Parallelism versus Pruned Search S-T (memory)

With an unbounded number of PEs, MIS has a smaller space-time product than exhaustive search with full data parallelism by a factor of $O(N/\lg N)$, with only a small $O(\lg N)$ speed penalty when the number of PEs is used as the space measure. When memory is used as the space measure, MIS is preferable whenever the virtual-processor ratio for exhaustive search is greater than $O(\lg N)$, which will be the case for practical machines except for very small KBs. MIS can thereby be used to asymptotically reduce the number of PEs and approach the ideal of a high virtual-processor ratio with the processing speed of full data parallelism.

5.5 Common Characteristics of the Example Algorithms

The pruning effects in all three target domains indicate fundamental commonalities between the applications. With MIS comparison pruning the optimal number of PEs results in a large memory per PE and re-introduces memory-PE bandwidth as an important SIMD design issue. Only a small portion of data objects are active at any given query, and memory requirements per PE grow asymptotically in the size of the problem despite increasing the number of PEs. Memory size and bandwidth become a key difficulty.

5.5.1 Scaling

The example algorithms exhibit scaling effects believed to be common to applications involving filtering and pruned search. As shown in Figure 5.17, the pruning has a major effect on how a parallel architecture should scale with larger application sizes. The figure shows how the appropriate number of PEs grows with the number of populated locations in a 1,000 bit input/output SDM, the number of nudes in a semantic network, and the number of graphs in the knowledge base of relational structures. In all cases, the appropriate number of processors grows sub-linearly in the size of the problem. This PE scaling improves system cost-effectiveness, but has a penalty. Figure 5.17 also shows that memory per PE must grow significantly as problem size is increased, to sizes much larger than most current parallel processor configurations. Even so, growth per PE is also sublinear. Commodity memory is less costly than custom or semi-custom processors.

Table 5.9 shows asymptotic and approximate memory requirements for the three targets; N is the total number of objects, n is the size of a graph, m is the length of an SDM input/output vector, and b is the number of bits in the processor ALU. The numeric values for object size are from typical applications. Increasing the number of processors for semantic networks would reduce execution time by avoiding marker communication congestion and reducing the virtual processor ratio in some initial marking criteria. Increasing the number of processors for graph retrieval would allow for speculative execution, also increasing speed. At 1 K processors for each application, memory per PE is 1152 KBytes for SDM, 128 KBytes for SN, and 512 KBytes for CG. For the given N, the portion of active objects is less than 0.1% for SDM and CGs, and less than 0.2% Even at $N \approx 2^{16}$ less than 0.5% of the objects are active for SDM and CG. Figure 5.17 shows how the number of PEs, memory per PE, and the average time per query grows with N for the 3 example applications. Memory requirements per PE were calculated for a variety of published and planned experiments and applications for the three example application. Requirements range from 153 K bytes to over 1.6 M byte per PE. Because inter-object access has little locality, traditionally organized memory hierarchies may not be effective. These memory issues begin to suggest the close interaction of algorithm, architecture, and implementation technology.

Domain	Objects	Number PEs	Object Size, Bytes	Mem per PE, K bytes
SDM	$N \approx 2^{20}$	$O(N^{1/2})$ 1024	$O(m)$ 1152	$O(N^{1/2}m)$ 1152
SN	$N \approx 2^{20}$	$O(N^{3/5}/\lg N)$ 850	$O(\lg N)$ 128	$O(N^{2/5}\lg^2 N)$ 153
CG	$N \approx 2^{20}$	$O(n^2/b\lg N)$ 320	$O(n^2)$ 512	$O(Nb/\lg N)$ 1678

Table 5.9: Processing Element and Memory Requirements

5.5.2 Instruction Profiles

Table 5.10 summarizes the combined macro profile which is the basis of the architectural analysis in Chapter 6. The combined profile is based on the manual profiles as described for each of the three example applications, weighted so that SDM represents 35% of total computation time, semantic network 20%, and graph retrieval 45%. Each application has a single macro-operation that occupies the majority of execution time, showing that tailoring to just a few operations can significantly improve performance.

Just three macro operations (Hamming distance, Boolean matrix multiplication, and match) represent nearly 3/4 of array computation time for the weighted mix. These macros and the primitives used in building them are important in a variety of AI operations. Hamming distance is representative of neighborhoods and continuity as an inductive bias,

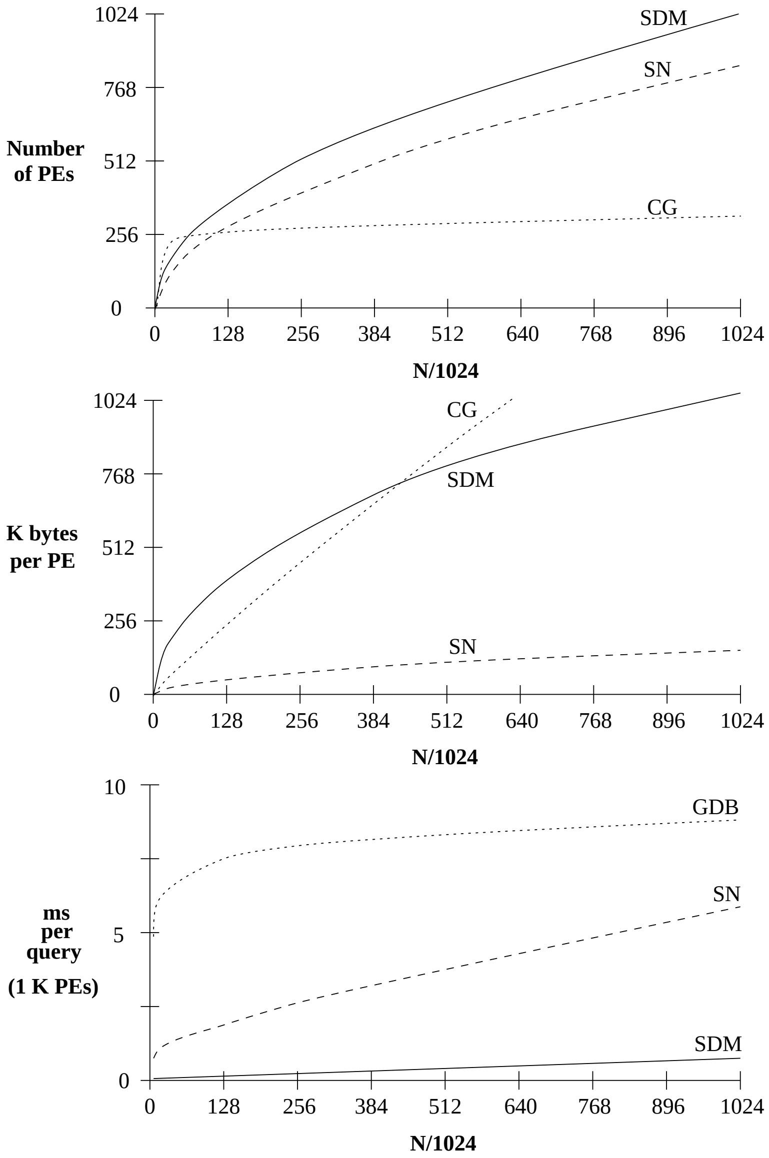


Figure 5.17: Example Application Scaling

matching is the lowest-level primitive of search and recognition and is based on primitive operations also useful in Boolean correlation, and Boolean matrix multiplication processes structural relationships, adjacencies and connectivities, and compiled recursive inferences.

Table 5.11 summarizes the results of manually breaking the associative array macros down into machine-level instructions, assuming a generic associative array PE with both word-wide and bit-wide general-purpose ALUs, local addressing, and mesh communication network. Word-wide computation represents nearly 50% of total execution time; the most common operations are exclusive-or and add/subtract (roughly 10% of total execution time each). Local (mesh) communication is important at 32% of total time. As the optimal number of PEs for target application sizes represent a reasonable size for a single PN's

Operation	time
Boolean matrix multiplication	30%
Hamming distance	29%
match	13%
general ALU	13%
logical	10%
local communication	5%
global communication	1%

Table 5.10: Projected Breakdown of Total Execution Time by Macros

associative array, each PN can be a processing cluster with a very high ratio of internal communication to external communication. Together, these indicate that the hierarchical MSIMD architecture will be highly scalable with manageable global communication requirements.

Operation	time
local communication	32%
word-wide logical operations	24%
word-wide arithmetic operations	21%
1-bit logical operations	15%
all other	8%

Table 5.11: Projected Breakdown of Total Array Execution Time by Instructions

5.6 Summary

This chapter has presented parallel pseudocodes, analysis, and manual instruction profiles for the example algorithms of SDM, semantic networks, and graph knowledge base retrieval. The analysis and instruction profiles characterize the algorithms as a basis for developing, refining, and evaluating the MISC architecture.

In several cases the analysis and parallelizations are new contributions. Analysis for SDM has shown that exhaustive Hamming distance calculation is required to be fully general, and the parallelization exploits a multiple-response resolver. An analytical model for semantic networks projects the average numbers of propagating markers and the parallelization exploits subsumption codes to speed inheritance operations. For graph databases, analysis shows that for bounded number of processors pruned search is as fast or faster than exhaustive search with full data parallelism, and that with an unbounded number of processors there is only a slight reduction in speed but a tremendous reduction in the number of processors required. This is likely to be the case whenever both filtering techniques and multiple levels of parallelism are available. This chapter presents a new multi-bit multi-graph parallelization of Ullmann's SI refinement algorithm, and shows how it can be adapted to conceptual structures. Simulation experiments characterize refinement behavior, and suggest approaches to load balancing and a new integration with Method III multi-level indexed search. The chapter shows that transitive-closure coding of partial orders can approach the lower bound when multiple inheritance is absent, and shows how codes for posets with multiple inheritance can be quickly processed to speed other operations.

The most important common characteristic influencing the MISC architecture is that of initial low-cycle operations significantly pruning subsequent time-consuming operations. Sub-linear scaling in the number of processors results in small SIMD arrays with unusually large memory per processor. Having fewer processors means that they could be made more powerful. The manual instruction profiles indicate that a small number of macro operations dominate execution time. As these are typical of knowledge processing requirements, tailoring the array processors to support these and other operations can have a significant effect on speed. The influence of application characteristics on the MISC architecture is analyzed in detail Chapter 6.

6. Design Development

Having established the design tools and application characteristics, we can now evaluate architectural choices and implementation issues. This chapter presents initial design analysis, illustrating the path taken from initial design ideas to the MISC architecture detailed in the next chapter. Section 6.1 of this chapter introduces architecture-technology interactions. It presents technology issues that must be faced by computer systems in general, many of which are exacerbated for parallel systems, and relates these to the special needs of AI processing. Section 6.2 then summarizes a series of specific design studies to select features for the MISC architecture. These studies are based primarily on the preliminary manual profiles and consider global and local communication networks, array memory, processing element architecture, and processor chip configuration. Section 6.3 presents a comparison of the projected cost-performance of several possible implementations in order to make a choice between high-level alternatives. Section 6.4 includes analysis of additional design alternatives based on simulations of the base MISC architecture.

6.1 Technology Characteristics

Considering implementation technology is important in developing a new architecture. Technology limitations may result in reduced performance or even render an architecture unimplementable. On the other hand, an understanding of technology capabilities and costs can influence architectural decisions, improving system cost-performance, and new technology capabilities may suggest architectural innovations. This section introduces general architecture-technology issues at the system and chip levels as a preface to the specific trade-offs analyzed for MISC presented in the next section.

6.1.1 Interconnection and Packaging

Interconnection is the ubiquitous limitation in VLSI system implementation and is implicit in many aspects of the Suspense early-analysis tool. Interconnection is crucial at both the die and system levels. At the die level, only a tiny fraction[1] of the total area is covered by active transistors; the remaining area is wiring. In packaging die, pin count is usually a limiting factor. At the system level, thermal contraction and expansion (TCE) and alignment accuracy limit packaging to about a hundred connections per edge [TH88]. The same pin count and wiring problems occur at each level in the system packaging and interconnection hierarchy. For a printed circuit board or multi-chip module interconnecting die, limited routing capacity is often the determining factor for area and mechanical factors limit the number of interconnections to the next level. These issues are particularly crucial for parallel systems, with their higher interconnect requirements between chip and packaging and between higher levels of packaging.

With few exceptions all packaging is planar with external connections around the periphery, so that while area scales as $O(N)$ (where N is the number of transistors or processing

---

[1] Typically less than 1% of a die area is transistors, even if several metal layers are used.

elements) the available periphery for connections is $O(\sqrt{N})$. Rent's rule empirically relates the number of devices to the number of connections. The number of external connections is cN^{β} where c and β are empirical constants. Since β is generally less than 0.5 (except for supercomputer die) [Bak90], the $O(\sqrt{N})$ scaling not a fundamental problem for uniprocessor systems. Practically, mechanical issues make packages with more than a few hundred pins difficult, and advanced microprocessors are pushing these packaging limits. Interconnection scaling poses a fundamental problem for parallel systems; a truly scalable parallel system requires $O(N)$ communication capacity. Massively parallel systems place several PEs per die. If memory is too large to be placed on-chip, $O(N)$ connections are required for off-chip memory. If these conflicts aren't resolved, a parallel system experiences an $O(\sqrt{N})$ slowdown once interconnect constant bounds are reached, and quickly becomes ineffective.

Other than increasing the constant factors, advanced technologies such as 3-dimensional VLSI, wafer-scale integration (WSI), and multi-chip modules (MCM) do not appear to offer relief. The prospects for 3-dimensional VLSI are not promising due to planarity problems in the fabrication process. Currently 5 or 6 metal layers is the practical limit for manufacturability and with area bounded by wiring there is little point in providing additional diffusion layers. Stacking multiple die is not generally attractive either. For true 3-d connections anywhere on the die surface it is necessary to diffuse metal completely through the die. Although there is existing work in this area, including a prototype SIMD processor composed of stacked wafers [L^{+}90], this has not proven commercially viable. Die can be stacked with interconnect on the die edges, as described in Section 3.2.2 for the Suspense model, but this gives the same connection constraints as planar packaging. It has, however, proven viable for high-density memories, where pin count requirements are low. Flip-chip packaging, described in Section 3.2.2, has been in use for over a decade in mainframes and supercomputers and is becoming more common in lower-end systems. Mechanical requirements, particularly joint reliability, again limit the total number of connections to a constant number, expected to be a few thousand interconnects per die. Although pins on both the die and receiving substrate form a 2-d array giving $O(N)$ pins, routing in the receiving substrate must still cross the die boundary so that the number of second-level interconnects remains $O(\sqrt{N})$.

Stacked and 3-d packaging for PCB, MCM, and WSI suffer from the same difficulties as VLSI die. Most PCB stacking methods use peripheral connections, but connections through the PCB are part of current manufacturing technology. There are several "3-d" board stacking technologies offering hundreds to thousands of interconnects between boards, such as the AMP Interposer [LAS89] and the "button board" [Kni91]. At the board level, mating force poses significant restrictions on the number of connections and layers.

The effective die edge-crossings can potentially be at less than the bonding pad pitch, giving advantage to array (versus peripheral) pad placement when multiple routing layers are available. Table 6.1 shows effective pitch (wire pitch divided by the number of wiring layers) for a variety of technologies at typical cost-effective numbers of routing layers; surprisingly all are about $25\,\mu$m. At less cost-effective extremes such as multi-layer WSI, MCM with 5 signal layers, and very fine-pitch PCB effective pitch can be $10\,\mu$m, and at the other extreme a low-cost PCB has an effective pitch of $110\,\mu$m.

MISC analysis assumes an upper bound of 1–2,000 pins per package. Packages with a hundred pins are inexpensive, while those with a several hundred pins are more costly on

Technology	Pad Pitch	Wiring Pitch	Typical Layers	Effective Pitch
WSI	40 μm	20 μm	1	20 μm
MCM	100 μm	50 μm	2	25 μm
PCB	1000 μm	200 μm	8	25 μm

Table 6.1: Effective Die-Crossing Connection Pitches

a per-pin basis. Packages with a thousand or more pins are currently feasible, but require expensive non-standard manufacturing. They should be more common in the near future.

6.1.2 VLSI Scaling

VLSI characteristics are also a major influence on system implementation and cost-performance. Many die-level scaling effects can be investigated by Suspense. Although reducing the minimum feature size λ increases the number of devices per chip and reduces gate delays this is not without a disadvantage: propagation delay can increase due to increased interconnect resistance. If the line is long enough such that line resistance is greater than driver resistance, delay will be quadratic in the line length. Suspense shows a linear reduction of gate delay in terms of λ, but increasing propagation delay below $1.2\,\mu$m for a 1 cm line even with a driver transistor 10 times the minimum and a 4 gate load. This is in agreement with other studies, such as Parhami's [Par90]. As λ is reduced die size for the same number of transistors will be proportionately smaller and the typical line length shorter. Normalizing to the size of a $0.5\,\text{cm}^2$ die at $1.2\,\mu$m, total delay falls roughly linearly. Figure 6.1 summarizes these simulations. Using the microprocessor model, Suspense shows that the net effect of reducing λ gives a better than linear speedup for PEs as illustrated in Figure 6.2. Reducing λ nonetheless exacerbates the off-chip interconnect problem for parallel systems. At half the minimum feature size, the number of possible interconnects remains the same but there would be four times as many PEs per chip and each would be operating at nearly twice the frequency.

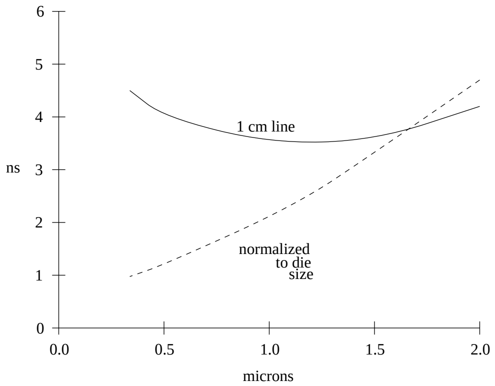


Figure 6.1: Combined Gate and Propagation Delays

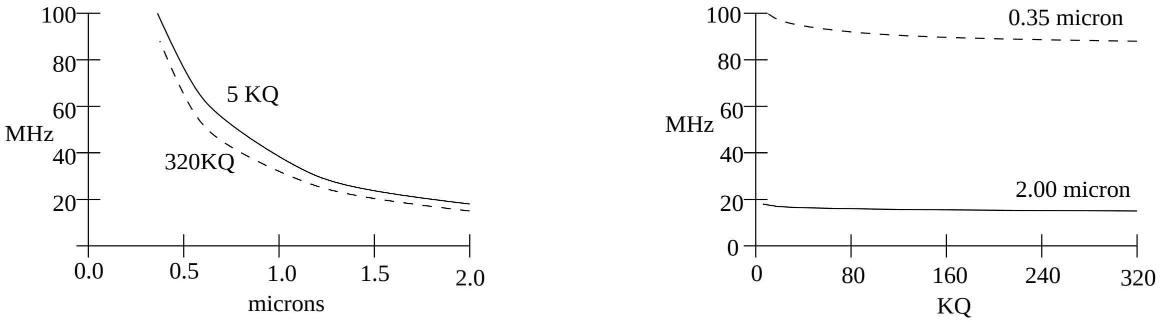


Figure 6.2: Microprocessor Speed Scaling

Figure 6.2 also shows that increasing the number of transistors in a microprocessor only slightly effects cycle time so that larger processors need only a small (5–30%) performance improvement to compensate for cycle time slowdown. Considering computational throughput per transistor, however, significant speedup is required. For contemporary microprocessors, the majority of the transistors are in on-chip cache. At a given transistor budget, parallel systems can achieve high speed by trading off processor complexity and size for an increased number of processors. Parallel systems can also make a trade-off between slower cycle time to avoid cache against a higher number of processors. Speedup is roughly proportional to word-size (the $\Omega(\lg b)$ time for carry propagate, where b is the number of bits in the ALU, is usually dominated by other factors) so that increasing word-width to be in line with typical application data sizes is appropriate for parallel systems, but other PE enhancements common in uniprocessors may not be desirable in a parallel system.

Historically, microprocessor speeds have increased much faster than memory speeds, resulting in the current "cache crisis" in microprocessor system design [Hig90]. This is largely due to the emphasis on increasing capacity per chip in new memory development. Suspense analysis shows that keeping the memory size constant, speed is roughly linear in λ for both SRAM and DRAM, so that the ratio of memory to microprocessor speed would be improving for a fixed memory size. The difficulty is that faster microprocessors need significantly larger cache and main memory.

Yield constrains the optimum die size for both serial and parallel systems. Although putting as much circuitry on a chip as possible reduces pin count when Rent's β is less than 0.5, cost per unit area of silicon is proportional to e^{Ad} due to yield, where A is area and d is the number of defects per unit area. For a given number of defects, silicon cost is exponential in the die area, as shown in Figure 6.3. At $0.6\,\mu$m and $d = 1$, Suspense shows cost per transistor increasing by a factor of four as die size rises from 0.1 to 1.5 cm^2. By reducing die size, smaller λ processes can reduce silicon cost despite the higher cost per unit area, as shown in Figure 6.3 for an 80,000-transistor die and $d = 1$.

6.2 Initial Design Studies

This section analyses the example applications using the manual profiles of their pseudocodes. Fortunately the target algorithms are highly iterative and only three macro operations occupy the majority of execution time. From the characteristics of the example

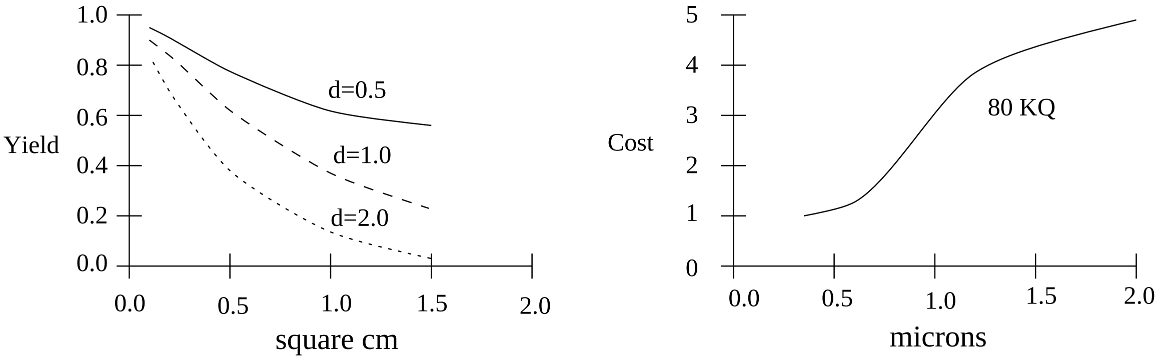


Figure 6.3: Yield Effects

algorithms identified in Chapter 5 and knowledge processing in general, we begin with the assumption of an MSIMD architecture with multiple small arrays with processors tailored to support the key operations, dynamic compound data structures, and data-dependent control.

We first consider the network for global communication between processor nodes, followed by the array memory, the array local communication network, and the processing element microarchitecture. Analysis of the PE architecture includes major alternatives such as CAM and extremely wide datapaths but concentrates on several special function units.

6.2.1 Global Communication

Consider global communication between N processor nodes (PNs). Even with true 3-d packaging communication bisection bandwidth is at best $O(N^{2/3})$, but uniform destination distribution requires $O(N)$ communication. For a truly scalable system with no communication slowdown as the system grows, advantage must be taken of communication locality. For all communications patterns with a planar topology, the required communication bisection width is $O(N^{1/2})$ [CGB91], a value that shows strong communication locality and permits operation within the bounds of planar packaging.

AI codes may not be so fortunate, with data-dependent “random” communication patterns. There has been little empirical work in this area. The MISC architecture provides scalability for AI codes by exploiting locality, particularly through computation clusters with a high ratio of internal computation to external communication. Much higher communication rates are possible within a cluster, due to a reduced number of PEs (as compared to the full machine) and closer physical proximity. Clustering requires making the array for each PN large enough for the largest expected routine. As shown in Figure 5.17 (page 112), this implies 128 to 1024 PEs per PN, well within the realm of current implementation technology. The following analyses estimate global communication versus internal processing for each of the example applications, verifying the feasibility of the hierarchical MISC architecture.

Processor Node Bandwidth Requirements

SDM is the most amenable to analytical treatment. For an $m = 1024$, $N = 256\,K$ SDM (where m is the number of input and output bits and N is the number of physical storage locations) Phase 1 processes $2^{10}/2^3 \times 2^{18} = 2^{25}$ bytes, assuming 1 operation per 8-bits of Hamming distance calculation. In Phase 2, the number of active locations a is about 500 so that $2^{10} \times 2^9 = 2^{19}$ byte-additions are be performed. Manual profiling shows 2000 iterations per second, giving 512 K bytes/s global communication for the PN. The ratio to bytes processed within the array to bytes communicated outside the array is extremely high. For each 2^{25} bytes processed, only 2^8 bytes are communicated.

Next consider a 32 K node semantic network on a single PN with 1 K PEs (P). For a simple query, we assume marking 4 four-byte node identifiers, followed by 16 propagation cycles with an average of 128 active markers per cycle. Marking thereby takes $2^2 \times 2^2 \times 2^{15} = 2^{19}$ byte operations and marker propagation moves $2^4 \times 2^7 = 2^{11}$ bytes. Assume a 2-d mesh for local communication with a random communication taking $3\sqrt{P}$ communication cycles [HCWS94]. With each cycle taking 12 instructions at 60 MHz, total time is $840\,\mu$s. With the query and response totaling approximately 128 bytes (some 28 node-ids as the returned result) communication is 128 K bytes/s. Again, there is a high ratio if intra-array computation to inter-array communication, with 2^{19} bytes processed for every 2^7 bytes global communication.

The profiled conceptual graph scenario executing on a single PN of 1024 PEs gives 2^{46} bytes processed. Building the 16 K node concept-type hierarchy requires messages of 16 bytes each and the total of 64 K queries requires messages of 64 to 256 bytes each, for a total of 2^{21} bytes. Manual profiles indicate a processing throughput of around 400 queries per second, giving 64 Kbyte/s global communication.

Table 6.2 summarizes results, assuming a 60 MHz PE clock speed. For all three applications the ratio of internal computation to communication is extremely high; at most 0.03% of bytes processed "generate" a byte of external communication. The example applications were not selected for having such high ratios, although communication clustering was expected. The author believes that similar high ratios will be found for other search paradigms and highly iterative codes, the open question being the number of PEs required for a cluster in other applications. Cheriton *et al.* claim that communication clustering occurs in many applications, particularly simulations and partial-differential equation solvers [CGB91]. For 2-dimensional problems region crossing analysis also shows that processor load increases quadratically with region size while communication increases only linearly [CGB91]. This gives the $O(\sqrt{N})$ communication scaling typical of engineering and scientific codes.

Example	Global Comm. M bytes/s per PN	Internal Processing Bytes	External Comm. Bytes	Internal to External Ratio
SDM	0.5	2^{25}	2^8	128 K
SN	0.1	2^{19}	2^7	4 K
CG	0.1	2^{46}	2^{21}	32 M

Table 6.2: Example Application Clustering

Communication Network

At the global communication bandwidths projected for the three example algorithms, a simple high-speed bus (100 MByte/s) or point-to-point network (10 MByte/s links) could handle a hundred PNs. Where latency is not crucial, a local-area network could serve for global communication. For long range application sizes, it may be necessary to divide "applications" across multiple PNs, resulting in higher communication rates more typical of existing MSIMD machines (such as the CM-5's 5 MByte/s global bandwidth per processor). Analysis shows that this is not the case for SDM. A partitioned semantic network, however, has 4 MByte/s inter-PN communication per node and the original processor farm model for the graph knowledge base has 17 MByte/s inter-PN communication per node.

More advanced networks would be necessary for scaling to more than 8–16 PNs at the higher communication rates. Even at 5–20 MByte/s per PN, MISC can scale to a hundred, perhaps thousands of nodes (millions of processors) using networks such as fat-trees, multi-buses, and 3-d meshes. With gigabyte/s networks becoming available, the difficulty would be more with application parallelism and programming than machine scalability and it is not necessary to consider the MISC global communication network in detail at this time.

6.2.2 Array Memory

Application characteristics allow initial analysis of the associative array memory in advance of memory access profiles from simulation. First, due to the large amount of memory per PE, array memory must be off-chip unless implementation incorporates the PEs into high-density DRAM chips (this option is analyzed later in Section 6.3). Off-chip memory introduces significant problems in bandwidth and pin count that the architecture must resolve. In the following analysis we assume a 17 ns (60 MHz) PE cycle time, based on Suspense analysis of the preliminary MISC architecture.

Bandwidth and Access-Time Requirements

We first calculate memory-PE bandwidth requirements for each of the key macro operations in the example applications. A word-wide Hamming distance can be calculated in one or two cycles with minimal custom hardware. For a large SDM, physical location addresses total 128 K bytes per PE. This is unlikely to fit on-chip, so that off-chip memory cycle times must be 17–34 ns. SDM weights require 1 MByte per PE and although the weights are sparsely accessed, they are locally addressed and cannot be pre-fetched. Matching for SNs can also be done in one PE cycle per word, but node information might fit in a small on-chip memory. Otherwise off-chip memory must have a 17 ns cycle time to feed the PE one operand per cycle. For Boolean matrix multiplication, a 16×16 matrix can be processed in 16 sixteen-bit PEs in as few as 50–100 cycles with hardware enhancements. Each refinement would typically require at least 4 matrix multiplications and each subgraph isomorphism test at least 4 refinement calls, giving at least 800 cycles per matrix load. The PEs each load only 3 words, so that 8 cycles per word loaded would cause less than 1% slowdown.

Although some operations such as integer multiplication and log-sum take many cycles per word fetched, it is clear that the system will be slowed if the memory subsystem cannot deliver one word per cycle to each PE.

Memory Solutions

At a 17 ns clock cycle, memory cycle time is a problem that requires an advanced memory design. Fast commodity DRAMS have access times of 60–80 ns, so even with interleaving cannot meet the required cycle times. Large SRAMs have similar speeds, and smaller fast SRAMs (15 ns or less access) are over 10 times more expensive per byte than DRAM. Rambus [Bur92] has a peak transfer rate of 1 byte per 2 ns, but the 212 ns set-up time gives effective speeds no faster than DRAM except for long contiguous transfers (atypical of the above MISC macros where fetches are typically only 2–32 bytes). Further, Rambus requires a large on-chip area per memory port. EDRAM [Bon92, Jon92, Har92, Ng92] extends DRAM with an on-chip SRAM "cache," taking advantage of the wide on-chip DRAM line size. Current EDRAM products have an effective cycle time of 20 ns at the estimated cache-hit ratio for the MISC example algorithms; commercial EDRAM should have no difficulty meeting the MISC 17 ns target 5 years from now.

A simple analysis shows that a high-bandwidth array memory subsystem based on EDRAM is the best choice for MISC. Estimates show that the cost of MISC will be primarily memory, so it is reasonable to first consider memory cost-performance. Here we use access times and typical costs for commodity products, giving a "dollars per M byte-MHz." For DRAM this is around 2, for fast SRAM 10, and for EDRAM 1. We thereby assume EDRAM. Even so, MISC needs a separate word-wide memory port for every PE to provide one operand per cycle. This incurs a significant pin count penalty. Table 6.3 presents the results of a more detailed analysis. "Length" is the typical length of consecutive transfers, and "necessary hit ratio" is for 1 PE per EDRAM. At the weighted average hit-rates and transfer lengths, current EDRAM has an effective cycle time of 20.6 ns, Rambus 68.8 ns.

Macro	Length (bytes)	Typical Hit ratio	Necessary Hit ratio	Maximum PEs per Memory
Boolean Matrix Multiplication	2	.50	0.00	40
Hamming Distance	32	0.96	0.45	1
Match	8	0.87	0.45	1
Logsum (pipelined)	256	0.99	0.45	1
Logsum (non-pipelined)	1	0.00	0.00	14
Integer multiply	1	0.00	0.00	8
Other word-wide	2	0.50	0.45	1
Bit-operations	on-chip	n/a	n/a	any

Table 6.3: Off-Chip Memory Analysis

Sharing Memories Between PEs

To simplify analyzing the effect of sharing a single memory port between several PEs, macro-operations can be put into four categories based on the manual profiles and the results in Table 6.3.

1. Long input operations giving high hit ratios but still requiring one memory per PE for maximum speed. 50% of operations.
2. Multi-cycle operations, allowing many PEs per memory. 35% of operations.
3. Single-bit operations (including response resolver) requiring no off-chip memory references. 10% of operations.
4. Single-word operations (zero hit ratio), requiring 1 memory per PE for maximum speed. 5% of operations.

Based on these categories, we can project the effect on system speed and cost-effectiveness of sharing a memory port with several PEs. Let P_m be the number of PEs per memory. We consider three cases. In the first case, the off-chip memory datapath width is the same as the ALU datapath. System speed relative to a system with 1 PE per memory is then given by $\eta = .5P_m + .35(1 + (2P_m/39) + .1 + .05(2P_m) = .62P_m + .45$. The factor 39 is the weighted average number of cycles for the second category. In the second case the off-chip datapath is 1/2 that of the ALU but 2 memory words can be transferred to registers during one PE cycle (such as with interleaved EDRAM), giving $\eta = .5P_m + .35(1 + (4P_m/39)) + .1 + .05(4P_m) = .74P_m + .45$. In the third case, the off-chip datapath is 1/2 that of the ALU at the same operating speeds, giving $\eta = 1.0P_m + .35(1 + (4P_m/39)) + .1 + .05(4P_m) = 1.14P_m + .45$. These speed adjustments are combined with the Suspense cost model for die area, typical memory sizes, and pin cost. Even at only \$12 per die (0.8μm, 8 PE per die) and pin cost increased to \$1 per pin, cost-performance is better at one PE per memory, as shown in Figure 6.4, with cost-performance 26% worse at 2 PEs per memory. The cost analysis is conservative. Custom 132-pin PGA packages are currently \$0.26 per pin [ISI93], and 447-pin PGA sockets are under \$0.04 per pin [Cor95]. At \$0.04 per pin, sharing a memory between two PEs has a 35% cost-performance penalty (the difference is not greater as nearly half the cost is in silicon and a large number of pins are needed for control and communication). Other parametric runs at 0.6μm, 16 PE per die and $.35\mu$m, 32 PE per die give the same conclusion. This gives estimated total package pin counts from 540 (8 PE per chip) to 1600 (32 PE per chip).

These results strongly indicate that providing an external memory interface for every one PE is worthwhile even with high pincost, requiring and justifying very high pin count packaging.

6.2.3 Array Communication

The most common communication topologies for SIMD machines are hypercubes and 2-dimensional meshes, opposing ends of a class of k-ary n-cube networks [Dal90]. Here k indicates the number of processors along each dimension, and n indicates the dimensionality . A 2-d mesh is a $\sqrt{P}$-ary 2-cube and a hypercube is a binary $\lg P$-cube. Although meshes have a network diameter $O(P^{1/n})$ versus $O(\lg P)$ for the hypercube, Dally shows that for a given amount of wiring resources and constant bisection bandwidth, low-dimensional k-ary n-cubes have lower latency, less contention, and higher hot-spot throughput than high-dimensional ones due to greater resource sharing [Dal90].

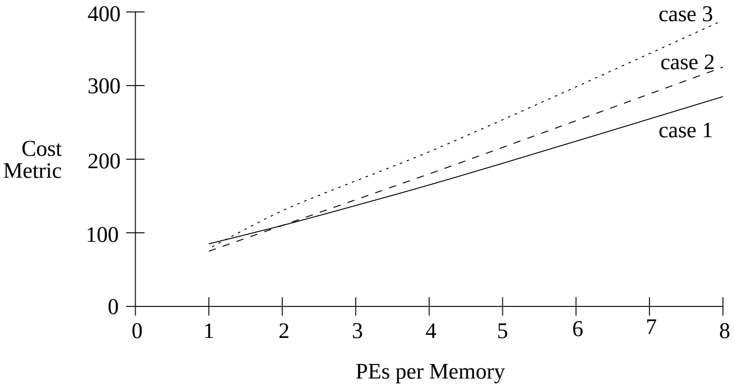


Figure 6.4: Optimal Number of PEs per Memory

Mesh

The 2-d mesh also has several advantages for implementation in planar substrates. Meshes have the advantage of a constant, small number of ports per processor and only nearest neighbor wiring. This decreases network interface costs and greatly reduces interconnect costs. The short interconnect lines also reduce signal delay and cross-talk as well as power and the resulting current-switching noise. The reduced wiring requirements also allow wider communication data paths. With a fixed array size, wider communication links can compensate for increased network diameter. At 1024 PEs and typical mesh routing algorithms, a 2-d mesh with 16-bit links is over 1.5 times faster than a hypercube with 1-bit links.[2]

For a more detailed analysis, we consider 4 routing methods: Foster's associative activation for graphs, Corbett *et al.*'s routing algorithm, the MP-2 multi-stage router, and a simple mesh-routing algorithm. Foster's method uses parallel matching operations rather than the local communication network, as described in Section 5.2; time is linear in the number of PEs sending. Corbett *et al.*'s algorithm forms coteries based on message destinations to skip over intermediate PEs and uses a 1-bit handshake line to signal blocking to other PEs in the coterie [HCWS94]. Their empirical tests give $2\sqrt{P}$ routing cycles for typical patterns, with small variance. Analytically, worst-case time is $\Theta(P)$. Each routing cycle is fairly complex, requiring at least 30 instructions each on MISC, giving $60\sqrt{P}$ cycles expected-case time. Empirical tests of the MP-2's global router give time linear in the number of processors sending, s, equal to $693 + 4283s$ for 32-bit data. This is in good agreement with the published average of 5,000 cycles for all PEs active. The simple mesh-routing algorithm developed for MISC in Section 7.4.2 page (172) turns out to be the fastest, despite not using coteries or the bypass mesh, as a result of having few instructions per routing cycle and reduced amortization of those instructions due to word-wide communication links.

Figure 6.5 plots processor cycles for the various routing methods on a 1 K PE array. The MP-2 times are from regression to empirical tests; all other times are analytical. Times for both Foster and the MP-2 are linear in s, whereas the MISC algorithm is sub-linear

---

[2] Here we favor the hypercube, including collisions for mesh routing but counting only the minimal lg N steps for hypercube routing.

approaching a maximum of 1864 processor cycles. MISC mesh general routing always requires fewer cycles than the MP-2 multistage router and the coterie, routing algorithm and is faster than Foster's method for more than75 PEs sending.

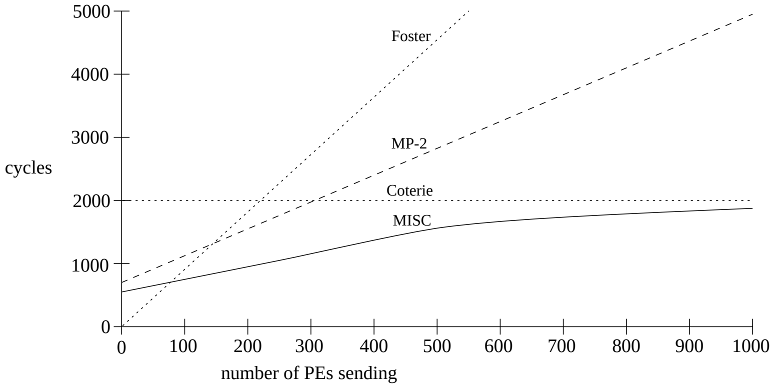


Figure 6.5: Comparison of Routing Speed, 1 K PE Array

Enhanced Mesh

From the above arguments, it appears best to use a 2-d mesh for the MISC arrays. We propose a *bypass-mesh* which allows an alternative connection skipping over a PE rather than being routed through it. PEs at any distance can thereby communicate with any other in the same row or column in a small number of cycles. This is the similar to the `xnetShift(distance, source, destination, length)` command on the MasPar except that time is nearly independent of distance. The bypass-mesh can also serve the same functions as the buses of the DAP.

Communication patterns in the example algorithms indicate the need for both nearby communication in regular patterns (Boolean matrix multiplication) and general long-distance routing (semantic network marker passing). The 2-d mesh supports the communications of matrix transpose and matrix multiplication, and the bypass features allow forming sub-arrays as required for processing several matrices in the same array. In conjunction with a handshaking line, the bypass also supports a coterie routing scheme which reduces the constant factor of the $O(\sqrt{P})$ mesh routing cycles to 3 with a very small standard deviation [HCWS94], although for MISC it appears faster to use a simpler routing algorithm (described in Section 7.4.2). The mesh bypass asymptotically improves times for parallel prefix operations such as log-sum. Although a reduction requires only $\lg N$ operations (such as add), $\sum_{i=0}^{\lg N} 2^i = N$ routing steps are required on a mesh as each step routes twice the distance as the prior step. With the bypass-mesh the distant processors can be directly connected so that reductions and scans can be performed with $\lg N$ communication steps.[3]

---

[3] Formally, the time for a bypass connection is linear in the distance, but as we will see this has negligible practical effect in the fixed-sized MISC arrays.

Siegel shows that a mesh with edge wrap-around (torus) can emulate a hypercube with $\sqrt{P}/2+1$ slowdown and a shuffle-exchange network with $2\sqrt{P}-1$ slowdown [Sie90]. It might first appear that the bypass-mesh could efficiently emulate a $\pm 2^i$ network, in which all PEs can simultaneously send to another PE a power of 2 distance away. This would be highly desirable, since Siegel also shows that a $\pm 2^i$ network can emulate a hypercube or mesh with only a small constant factor slowdown (2 and 1, respectively) and a shuffle-exchange with $\lg P+1$ slowdown. Unfortunately, this is not the case. Although all rows and columns in a bypass mesh could transmit simultaneously, the bypassed PEs cannot participate; the time to emulate a $\pm 2^i$ network is $O(P)$.

6.2.4 PE Microarchitecture

CAM versus Multi-bit ALU

We begin the processing element design analysis by evaluating significantly different architectures. Execution times for each are used to categorize operations, allowing a basic analysis from the manual profiles. The hardware analysis uses $.35\mu$m CMOS to anticipate future conditions. At minimum feature size, thousands of processors can be put on a single chip, and chip I/O becomes the key implementation constraint. The five alternatives are as follows.

- **Bit-cell CAM**, as described in Section 2.1.
- **Minimal bit-serial CAM**, as described in Section 2.1. Circuit estimates give 48 transistors per word for logic, with equivalent logic depth (fld) of 28, including memory access.
- **Foster's bit-serial CAM**, as detailed in Chapter 10 of [Fos76]. This design includes 3 flags and functions such as add, subtract, and flag move and complement.
- **Universal bit-serial ALU**, similar to the CM-1, consisting of a 3-input 2-output universal logic block, one flag, and response resolver. Circuit-level estimates indicate 80 transistors per PE, with $fld \simeq 22$ including memory access.
- **Word-wide ALU**, consisting of a bit-parallel ALU and accumulator, plus additional circuitry for accumulator zero-detect and response resolver. Circuit-level alternatives indicate 720 transistors per PE, with $fld \simeq 28$ including off-chip memory access.

Cost Comparison Suspense runs permit a detailed characterization of the 5 options. Let A and C be the area and cost metric of the alternative denoted by the above abbreviations. Let P be the number of PEs or CAM words in thousands, M be the total on-chip memory capacity in megabits, and b be the number of bits in a word-wide processors. Regression to Suspense parametric runs give the parameters of table 6.4. Foster's bit-serial design has more than 3 times the transistor count of the universal bit-serial ALU, so that Suspense runs were not made on this design. Surprisingly, minimal bit-serial CAM is larger and more costly than dynamic bit-cell CAM. Also from Suspense, commodity DRAM can be approximated as $A = .00025P + .051M$, $C = .06P + 5M$.

Option	Cycle Time	A	C
Dynamic bit-cell CAM	10 ns	$0.0026P + 0.398M$	$0.375P + 28M$
Minimal bit-serial CAM	11 ns	$0.0116P + 0.256M$	$0.94P + 19M$
Universal bit-serial ALU	11 ns	$0.013P + 0.256M$	$1.02P + 19M$
Word-wide ALU	11 ns	$0.0056bP + 0.256M$	$0.78bP + 19M$

Table 6.4: Option Parameters

Speed Comparison Table 6.5 gives cycle times for various macro operations. Cycle counts for bit-cell, minimum bit-serial, and Foster's CAM are taken from Foster [Fos76]. Here m is the number of bits in the data representation, m' is the number of relevant bits in a partial match, and b is the number of bits in a word for the word-wide ALU. Normalized values are at $m = m' = b = 16$, with all operations equally weighted.[4] The minimum/maximum operation assumes each value is in a separate CAM word or PE. With this parallelism, a bit-serial matching algorithm is faster than a reduction even for fairly small N.

We can make several conclusions from Table 6.5.

- Bit-cell CAM is faster only for matching, by a factor of m/b; bit-cell CAM is desirable only for wide, exhaustive search. Otherwise more conventional ALUs are preferred.
- Word-wide processing is always faster than bit-serial, even when increased cycle times are included. The benefits are somewhat smaller with long word width off due to single-bit and inherently bit-serial operations.

For a more detailed analysis, we now group the operations in Table 6.5 into four classes based on asymptotic behavior and operand type, and show the percentage breakdown from the profile of the weighted combination of the application examples. The preliminary manual profiles also indicate a weighted average operand length (m) of 16 bits ignoring single-bit operations (14 bits overall).

1. Matching: $O(1)$ on bit-cell CAM and $O(m)$ otherwise, (13%)
2. General: $O(m/b)$ for all alternatives examined, (66%)
3. Inherently bit-serial: $O(m)$ and showing no advantage with word-wide ALUs, (1%)
4. Single-bit: $O(1)$ for all alternatives, (20%)

The analysis can be generalized, with insights into the appropriateness of bit-cell CAM for other domains. We ignore Class 3 as it is only a tiny fraction of operations, and ignore Class 4 as its execution is the same for all alternatives. This leaves only Class 1 (matching operations) and Class 2 (general operations). As a worst case for the word-wide ALU, assume that all data objects must be processed (no pruning on any operations) and let α be the portion of operations that are matches. Bit-cell CAM will be asymptotically faster when

---

[4] The manual profiles indicate that 13% of all operations are matching, excluding the match operations required for local communication routing, 58% are the five multi-bit operations (compare, plus, etc.), and the remaining 29% are four single-bit and related operations (union, set-not-empty, etc.). The operations are thereby nearly evenly distributed over all operations; equally weighting the operations slightly favors CAM, as it over represents the single-bit operations (10% versus 7%).

Operation	Bit-Cell CAM	Minimum Bit-Serial CAM	Foster Bit-Serial CAM	Universal Bit-Serial ALU	Word-Wide ALU
exact match	1	$m+1$	$m+1$	$m+1$	m/b
partial match	1	$m'+1$	$m'+1$	$m'+1$	$2m/b$
Total	2	$m+m'+2$	$m+m'+2$	$m+m'+2$	$3m/b$
Normalized	0.7	11.3	11.3	11.3	1.0
magnitude compare	$3m+2$	$5m+4$	$m+1$	$m+1$	$2m/b$
maximum, minimum	$3m+2$	$4m+4$	$3m+1$	$2m+1$	$2m+1$
increment, decrement	$12m+5$	$11m+3$	$3m+1$	$2m+1$	$2m/b$
plus, minus	$12m+5$	$11m+3$	$5m$	$2m+1$	$2m/b$
one's compliment	$7m+1$	$11m+1$	$3m+1$	$2m+1$	$2m/b$
$\cup$, $\cap$	2	2	2	2	2
ϵ	1	1	1	1	1
$\neq\{\}$	2	2	2	2	2
and, or	2	2	2	2	2
Total	$37m+22$	$47m+20$	$15m+11$	$9m+12$	$2m+8m/b+8$
Normalized	12.8	16.1	5.2	3.3	1.0

Table 6.5: Cycles per Instruction

$$\alpha + (1-\alpha)m \leq \alpha\frac{m}{b} + (1-\alpha)\frac{m}{b}$$

Solving for α gives $(1-\alpha)m+m/b \leq \alpha$, and solving for b gives $(1-\alpha)+\alpha/m \leq 1/b$, indicating the necessary portion of match operations required to justify bit-cell CAM given data size and ALU size and the necessary ALU width given data size and portion of match operations. Including typical constant factors for instruction cycles favors the word-wide ALU even more strongly, giving approximately $9(1-\alpha)m + 1.75m/b \leq \alpha$ and $5.1(1-\alpha) + .57\alpha/m \leq 1/b$.

Tables 6.6 and 6.7 give some typical conditions. The important factor is the ratio of b to m. It is clear that even with exhaustive search, a very high percentage of matching operations are required to justify bit-cell CAM on speed alone, even ignoring the cost advantages of commodity RAM. This is true even when the ALU width is relatively small with respect to the size of the match operand. Further, if only some fraction β of values need to be matched bit-cell CAM is further disadvantaged by a factor of $1/\beta$. Even if all operations are matching a word-wide ALU is faster whenever $b \geq 1.5\beta m$ (pruning factor $> 1.5m/\beta$); if this condition is met the ALU will always be faster than bit-cell CAM.

Cost-Performance Comparison By including the Suspense results we can incorporate implementation effects, but remain general by considering only silicon area and cycle time. In Table 6.8, cycles per instruction is an average weighted as above, and entries are normalized to the $m = b = 16$ case. The word-wide ALU is the winner, even at conditions that strongly favor CAM and bit-serial implementations; area is based on on-chip memory for only one datum. With MIS pruning, there can be a large amount of memory per PE with no direct slowdown. With more memory per PE, differences in PE area have less of an effect. Bit-cell CAM cannot take advantage of lower-cost off-chip memory, as data transfer is $O(m)$, destroying the $O(1)$ matching advantage. At 64 K bytes of memory per PE, and allowing for chip-crossing delays, the universal bit-serial PEs have S-T 12 times worse than word-wide ALUs. As memory per PE grows the effect of PE area becomes negligible and

Conditions		Necessary bits for word-wide ALU (b)	
match width (m)	portion matched (α)	without constant factors	with constant factors
16	0.1	1.1	0.2
16	0.5	1.9	0.4
16	0.9	6.0	1.8
16	1.0	16.0	$> m$

Table 6.6: Necessary Bits for Word-Wide ALU over Bit-Cell CAM

Conditions		Necessary portion match for word-wide ALU (α)	
match width (m)	ALU bits (b)	without constant factors	with constant factors
16	8	0.94	0.98
16	16	1.00	.99
32	8	.89	.98
128	16	.95	.99

Table 6.7: Necessary Portion Matching for Word-Wide ALU over Bit-Cell CAM

S-T differences approach time alone. As we will see, pin connections are the major influence on chip area, active circuitry being only a few percent of die area. In the extreme, only the cycle counts are relevant, so that the word-wide ALU has an advantage of $4.4b$ over bit-cell CAM and b over bit-serial. We thereby conclude that a word-wide ALU is the appropriate choice for the MISC PE.

	Bit-Cell CAM	Minimum Bit-Serial CAM	Universal Bit-Serial ALU	Word-Wide ALU
ns per cycle	10	11	11	11
cycles per instruction	$6.6m + 3.2$	$7.4m + 2.5$	$1.5m + 1.2$	$1.5m/b + 0.4$
area, cm^2	0.28	0.50	0.54	2.99
space-time	$18.5m + 9.0$	$40.7m + 13.8$	$8.9m + 7.1$	$49.3m/b + 13.2$
normalized	4.8	10.6	2.4	1.0

Table 6.8: Space-Time of Alternatives

PE Functional Units

Evaluation Metric Before performing space-time evaluations of alternative features for the PE microarchitecture, it is necessary to consider the effect of area increases on system speed and cost in order to develop an appropriate measure for comparing alternatives. Early analysis puts PE bare silicon cost at 12% of system cost (the majority of MISC system cost is memory and system interconnect). Although the analysis also shows that only a small portion of the of die area is active circuitry, we judgementally consider design complexity and development cost, reliability, and other factors and include die area as 5% of the evaluation cost metric. Increasing area also increases signal propagation time within each PE. Suspense indicates that signal propagation is typically 4–8% of PE cycle time, and line length and propagation time is approximated by the square root of the area. Driver and propagation delays across chips are about 25% of a communication cycle time, and local communication accounts for some 35% of array time over the 3 example applications. We thereby give the cost penalty as $1+(0.05(A-1))\times(1+0.06(\sqrt{A}-1))\times(1+0.35\times0.25(\sqrt{A}-1)$ where the first term is the area cost, the second is the PE cycle penalty and the third is the communication penalty. This simplifies to

$$C_{penalty} \simeq (0.00026A^2 + 0.0069A^{1.5} + 0.048A + 0.13A^{.5} + 0.815).$$

For each functional unit considered, the evaluation time metric is the relative speedup for a given operation, t, adjusted by its portion of the total execution time, r. This gives

$$T = (1 - r) + rt$$

and the space-time metric is simply S-T $= TC_{penalty}$.

Analysis Assumptions Some engineering bias is included in decision making, as other factors such as power density's affect on system packaging and speed are difficult to quantify except with detailed simulation, and there is a general bias towards elegance and generality, particularly with respect to compiler effectiveness. Several alternatives are re-examined during subsequent simulation and system evaluation. Preliminary analysis considered numerous alternatives for improving speed on the three main macros. As a starting point, we assume an PE of about 3000 transistors (16-bit ALU with zero-detect, shifter/accumulator, local-address adder with accumulator, bit-serial unit, and communications) plus 32 registers for a base area of 0.0006 cm^2 per PE at 0.35 μm. Results are summarized below.

Features for Hamming Distance Hamming distance is one of the major components in the example application mix. In its most general form, on ordered sets of discrete valued elements, Hamming distance can be efficiently implemented as $|A \oplus B| = +[A \oplus B]$ where $\oplus$ is the appropriate comparison predicate and A and B are tuples. Hamming distance is most commonly applied to Boolean vectors, as in SDM, and the comparison operator is exclusive-or, $\otimes$. Rather than placing one element per PE, it is advantageous to exploit the bit-parallelism of the b-wide ALU. A single exclusive-or logical operation can process the comparison for b elements simultaneously.

Numerous functional unit (FU) alternatives are considered. For conventional microprocessors, computing an m-bit Hamming distance requires about $4m$ logical, shift, and add instructions based on an anonymous routine for performing bit-wise reduction on conventional ALUs. Computing Hamming distance by a bit-serial exclusive-or gate followed by an adder, as in PCAM, requires m cycles and allows each position to be weighted, *i.e.* Hamming distance can be modified to be $\sum_{i=1}^{m}(a_i \oplus b_i)w_i$. If weighting is not required, the word-wide ALU can compare b bits simultaneously in a single logical operation, and then count the 1s in the result. This could be done by table lookup in a total of $4m/b$ cycles with indirect addressing and 1-cycle off-chip memory access. This can be reduced to $2m/b$ with the addition of a *ones counter,* a tree of 1-bit adders. This tree would require 384 transistors for a 16-bit value. Linking the local-address adder to the ones-counter and pipelining could further reduce time to m/b. The last alternative has the best S-T product. Under the analysis assumptions Hamming distance alone is sufficient to justify the one's counter, local address adder, and related circuitry. The hardware also supports operations analogous to multiply-accumulate where multiply is replaced by bitwise logical operations. Other useful operations are similar to Hamming distance calculation. For example, replacing exclusive-or with and gives dot-product and correlation for Boolean vectors, and ℓ_1 distance metrics also apply a compare-accumulate computation .

Features for Boolean Matrix Multiplication Boolean matrix multiplication can pack b bits into a standard b-bit ALU operation once one of the matrices is transposed. The dot product multiplication is then simply a bit-wise and. The dot product summation is simply a zero-detect; the ones counter already justified for Hamming distance can be used if a non-Boolean result is needed. Cyclic-shifting intermediate values across the PEs favors a mesh topology. Comparison to the matrix multiplication algorithm on Lucas [FKS85] indicates that a multistage interconnect network would double total Boolean matrix multiplication time. A mesh topology is also beneficial to the required matrix transposition. With single-bit extraction and insertion and insertion at any position, transposition an $n \times n$ matrix on n^2/b PEs is $12n$ cycles. Supporting these single-bit operations requires most of the circuitry of a general 1-bit ALU (flags and data routing). Even allowing for two Boolean matrix multiplications per transposition, this S-T analysis alone justifies the entire 1-bit unit (ULB, flags, etc.) at an estimated 400 transistors.

Features for Matching Exact match can be performed by a conventional ALU in m/b cycles as in the previous analysis by a bit-wise exclusive-or. A masked match requires an twice the cycles, and-ing the mask (match true if $(\mathsf{A}\otimes\mathsf{B})\wedge\mathsf{Mask} = 0$, false otherwise). For multiple comparisons using a single mask, interposing b and gates between the ALU output and zero-detect could reduce time back to m/b cycles once a latch has been loaded with the mask. Assuming 32 matches per mask loaded and 5% of execution time as partial matches, the and gates and latch provide a slight improvement in system S-T.

An on-chip CAM can speed matching operations, provided that the data is not part of a pruned search and can reside "permanently" in the CAM. This would be the case for descriptors, search filters, and indices into large data objects, as well as the node labels and status for SNs. Otherwise, the time to load the data into CAM would equal or exceed that to process the matches word-serially. Here we consider a small 32×32-bit *addressable* (random read/write by address, allowing use as registers as well) dynamic CAM. The speed advantage is from the greater parallelism; 32 words can be matched simultaneously whereas the PE would have to do each in sequence. S-T analysis indicates that the large additional area cannot be justified even if *all* matches in the example application profiles can be "permanently" resident. CAM might be justifiable if it serves double duty as on-chip memory. At 32×32 bits it is only 22% larger than SRAM (though 77% larger than DRAM), but is around 3 ns slower (at $0.35\,\mu$m).

Array Interconnection Enhancements Increasing the bypass-mesh datapath from 8 to 16 bits adds approximately 320 transistors per PE but cuts local communication time in half assuming data is typically 16 bits or longer. Applying the standard FU analysis, but allowing an additional penalty for pin cost on the 4 communication-only links per chip indicates a significant S-T improvement for the wider communication datapath.

For log-sum and similar reductions using the bypass-mesh requires $3(\lg(N/32)+5)(m + .5\lg N)/b$ cycles, assuming a 32×32 mesh. Other options such as Foster's carry shower [Fos76] or Blelloch's scan-tree [Ble90] require more cycles under typical conditions. An adder tree would have $fld = 2\lg N$, and including chip crossings a reduction over a full PN would require about 8 PE cycle times to do plus and minimum reductions. The total number of full adders required is $\sum_{i=i}^{\lg N}(N/2^{i-1})(b'+i)$ where b' is the number of bits in the starting values. The first term is the number of adders per level and the second term is the number of bits per adder. Starting with 8-bit values 10,220 adders are required, totaling 327,040 transistors (including Manchester carry chains), and amortizing to 320 transistors per PE. At least 20 pins are added per die. Allowing these to be 1% of total execution time in plus and minus reductions, S-T is very slightly worse with this option. Replacing the adders with a more general-purpose ALU increases hardware to 730 transistors per PE. At 5% total time for all reductions there is a slight S-T advantage. With 16-bit starting values, reductions must exceed 8% of execution time to justify reduction tree hardware.

Features For Multiplication A basic ALU requires $3m$ cycles for multiplication. Using the single-bit unit to extract bits from the multiplier, avoiding shifting the multiplier, reduces this to $2m$. Using Booth's algorithm with 2-bit encoding further reduces this to $1.5m$. Requiring only 16 transistors per PE for the coding and some additional control circuitry, this can be justified even for as little as 1% of the execution time in 8×8 bit multiplies. To be justified by system S-T, multiplies must exceed 6% of the total execution time. A dedicated bit-serial multiplier still requires $2m$ cycles.

A parallel (array) multiplier has $fld = 3m$. For an 8-bit multiply this is about the same as the SIMD PE, but does not include memory access, so that 2 cycles are required. Over 2000 transistors are required. Multiplication would have to exceed 17% to justify a 16×16 array multiplier.

Summary of Alternatives Table 6.9 summarizes these results. As before, r is the portion of the macro in the combined example application profile, t is the relative time for the macro with the given option, T is the relative time of the combined applications with the given option, A is the relative PE area, and $C_{penalty}$ is the total system cost-penalty.

The space-time product is for the full system and weighted combination of the example applications. The space-time product advantages appear small as we are looking at system cost rather than PE area. If time is the only consideration, all options are viable under the given assumptions.

Macro	f	Option	t	T	A	$C_{penalty}$	Space-Time
Hamming Distance	0.3	table lookup	1.00	1.00	1.00	1.00	1.00
		ones counter	0.50	0.85	1.07	1.01	0.86
		ones counter pipelined adder	0.25	0.78	1.22	1.03	0.80
Boolean Matrix Multiply	0.17	bit-serial unit	0.56	0.93	1.08	1.01	0.93
Masked Match	0.05	mask and	0.52	0.98	1.03	1.00	0.98
Match	0.10	CAM	0.03	0.90	3.25	1.25	1.12
Communication	0.35	16-bit	0.50	0.83	1.06	1.03	0.85
± Reduction	0.01	adder tree	0.33	0.99	1.06	1.02	1.01
General Reduction	0.05	ALU tree	0.33	0.97	1.14	1.03	.99
Integer Multiply	0.01	Booth's	0.75	1.00	1.00	1.00	1.00
		Array Multiplier	0.13	0.99	1.41	1.05	1.04

Table 6.9: PE Alternatives

Additional Options with PE on Memory Die

We now consider additional alternatives which would be possible if the PEs were on the same die as their memory. DRAMS are composed of memory sub-arrays, each with typically 1024 columns (bits in a line) so that the datapath from memory to processor could be extremely wide. A 64 M bit DRAM could support 8–32 PEs with1/4 to 1 M byte of memory each. An advanced 256 M bit DRAM could support 32 to 128 PEs. Feeding column outputs to word-wide PEs would not involve any architectural changes, but would affect implementation cost and performance. This is analyzed in the next section.

Bit-Serial ALU First consider placing a universal bit-serial ALU at each column.[5] Although bit-serial, there can easily be many times more as there would otherwise be word-wide ALUs, and this ratio could be increased to compensate for speedups from special FUs on the word-wide ALUs. This is not, however, desirable for general operations. Local communication would have to be bit-serial, slowing processing by roughly a factor of 6 over 16-bit communication with 35% of the total execution time in communication. Also, with 16 times as many processors a 2-d mesh topology would require 4 times as many hops. The pruned versus full data parallelism analysis of Section 5.4 points to computational inefficiencies as well.

---

[5] This, and the following analyses ignore possibly significant aspect-ratio problems in placing PE logic on an already laid-out DRAM chip. It is important to avoid turns in the column-lines, as area becomes $(wp)^2$ where w is the number of wires and p is the wiring pitch.

Bit-Cell CAM The universal bit-serial units could operate only as bit-serial CAM, and be supplemented by word-wide ALUs for match operations. Since the universal-logic block layout is larger than even static bit-cell CAM, and performing 1024 thirty-two-bit matching operations in 32 bit-serial cycles has the same throughput as performing 32 thirty-two-bit matchings in one cycle, we instead consider the bit-cell CAM approach. This gives us the same analysis as for the bit-cell CAM "FU" analyzed above except that since it can be filled by the DRAM line in a single cycle, the objects to be matched do not have to be "permanent," and the objects can be selected from a pruned data set. Even so, the space-time analysis shows a significant disadvantage over letting the word-wide ALU do the matching. Matching operations alone would have to exceed 20% of execution time to justify a small on-chip bit-cell CAM as an additional functional unit with single-cycle refill.

Wide Hamming Distance DRAM line size is appropriate for computing Hamming distance for large SDMs. For the large SDM example, input is 1024 bits and each PE processes 256 physical locations. Thus the number of cycles to load the input address into a register is well amortized. Hardware would include a 1-bit input register and exclusive-or gate for each of the 1024 columns, followed by an adder tree to compute the ones count. This is similar to the approach taken by the Stanford SDM hardware prototype [FKB87]. The difficulty is with hardware cost. Despite starting with 1-bit adders, the adder tree requires over 65,000 transistors. Including the registers and gates, a total of nearly 84,000 transistors are needed. Hardware could be reduced somewhat by truncating the tree midway and using the "bypass-mesh" log sum. This, however, increases space-time, and an optimization gives the full reduction tree. SDM is sped up by a factor of 50 (t = 0.02, T = 0.701), but at nearly 24 times the area (A = 23.8, $C_{penalty}$ = 3.54); S-T is 2.52 times that of letting the ALU do all the work.

Boolean Matrix Multiplication Features The ability to quickly fill wide functional units could also speed Boolean matrix multiplication. So long as the word-width of the processors (b) is greater than n for an $n \times n$ matrix, n processors are used and time is $O(n)$. If this is not the case, time is $O(n^2/b)$. Here we analyze the case where $n = 64$ and $b = 16$, taking constant factors from manual algorithm profiles. Speedup is a factor of 3 ($t = .33, T = .89$), but again hardware is excessive. Logic circuitry is required to or each bit, perform a zero-detect across each matrix row, and shift rows. Matrix transposition requires a broadcast bus, and the ability for each matrix element to receive a bit from any of the bus' n lines. For 1024 bits, over 16,000 transistors are required and allowance must be made for the communication lines. This gives $A = 5.56$ and $C_{penalty} = 1.49$, for a space-time 1.33 times that of the ALU alone.

PE-on-Memory Conclusions It might be possible to share hardware and combine savings over these options. Even assuming maximum overlap and no control logic, over 100,000 transistors per PE are required, over ten times that of the PE including its register set. Combined speedups would halve total execution time for the example applications mix (T = 0.49), but A = 29.0 and $C_{penalty}$ = 7.27 giving S-T 3.6 times worse than the ALU alone. Although with the FUs on the large DRAM die, the memory I/O line perimeter crossing does not force a larger die area, a large number of pins are still needed for the mesh communication network. The large area of the additional functional units (0.8 to 3.2 million transistors for 8 to 32 PEs) may push size manufacturability limits so that $C_{penalty}$ is likely to be underestimated. Although intriguing, there does not appear to be any cost-effective means of architecturally utilizing the very wide data paths available by putting the PEs on the same DIE as the DRAM.

6.3 Preliminary Implementation Analysis

This section presents early analysis of several alternative MISC implementations. This is done to anticipate technology constraints (such as possible cooling problems), choose between separate PE chips or placing the PEs on a DRAM die, and to get a first look at system performance and cost-metrics. We first consider the appropriate number of PEs per chip, followed by comparison of system-level options.

6.3.1 The PE Chip

Having concluded that a separate off-chip memory for each PE is cost-justified despite the high pin count and that a wide-link 2-d mesh is most appropriate, the total number of pins per die is estimated as $N_p = (Pq + 4\sqrt{P}r + c)/0.7$ where P is the number of PEs per die, q is the number of lines in a memory port, r is the number of lines in a communication port, and c is the number of control lines. The 0.7 factor allows for additional ground and power lines. This creates a lower bound for the PE die area of max $\left[(pN_p/4)^2, (a\sqrt{P})^2\right]$, where p is the effective die-crossing line pitch and a is the array pad spacing. The first term is where area is determined by die-edge crossing in the packaging substrate and the second term is where area is determined by die array-pad spacing. Table 6.10 presents pin counts and areas for a 16-bit data path, $q = 29$ and $r = 17$, $c = 64$, $p = 25$ μm, $a = 200$ μm. Yield is at 1 defect per cm^2, typical of long-run logic. Array-pad spacing dominates only for 8 or fewer PEs per die and yield becomes unacceptable at around 32 PEs per die. The optimum die cost-metric is at 8 PEs per die, but the cost-metric at 16 is only slightly higher and is preferable from a systems standpoint, module area and the number of chips. *We conclude that 16 PEs per die is optimal.*

The packaging interconnect lower bounds of area are an order of magnitude larger than that of the PEs with selected FUs[6] and 32 registers, leaving the bulk of die area available. It is probably unreasonable to include additional FUs and features, for many of the reasons RISC PE architectures are preferred over CISC: minimal reduction in total cycle count while potentially increasing critical-path timing (and in turn time per cycle), inability of compilers to make good use of many features, and reduced reliability. It does seem reasonable to include more registers and perhaps a small on-chip memory. At 0.35μm 1 to 2 K bytes of registers could be added. This would, however, slow the PE cycle time excessively. Instead, 4 to 8 K bytes of on-chip DRAM, or 2 to 4 K bytes of CAM could be added. These sizes fall far short of AI application memory requirement, but are reasonable for other areas such as

---

[6] Ones counter and pipelined adder, bit-serial unit, mask and, and 16-bit communications, all improving system S-T.

PEs per Die	Pins	Pins per PE	Area cm^2	Area per PE cm^2	Yield	Cost Metric per PE
1	236	237	0.094	0.094	91%	5.0
2	320	160	0.128	0.064	88%	3.5
4	463	116	0.185	0.046	83%	2.7
8	714	89	0.286	0.036	75%	2.3
16	1166	73	0.513	0.033	59%	2.4
32	1999	62	1.561	0.049	21%	5.7
64	3566	56	4.967	0.078	1%	153.5

Table 6.10: PE per Die Optimization

image processing and computational biology. Profiles of memory references from simulations are necessary to determine the merits of small on-chip memory for MISC applications. One memory per PE capable of delivering one operand per cycle is cost-effective, so benefits of a small on-chip SRAM or DRAM are likely to be minimal; hardware support for numeric processing may be more advantageous and would result in a more general-purpose machine.

6.3.2 System Alternatives

We now take a first look at six system-level implementations to give further guidance to architectural choices.

1. **PCB**. A 256-PE PN on a printed circuit board, implemented in 0.6 μm with commodity SIMM EDRAM (4 M bit per die) and a commodity microprocessor for the supervisor. Due to high pin-count, the PE chips are packaged in ball grid arrays (BGA, arrays of solder balls).
2. **advanced PCB**. A 1024 PE PN on a printed circuit board, implemented in 0.35 μm. Here memory is wide data-path 64 M bit EDRAM. In addition to the supervisor, a 100 K transistor co-processor chip is included. All die are packaged in BGA.
3. **MCM**. Same as the 2, except the system is implemented in MCM rather then PCB.
4. **PE-on-memory**. Same as the 1, except PEs are placed on the same die as main memory.
5. **advanced PE-on-memory**. Same as the 2, except PEs are placed on the same die as main memory.
6. **MCM PE-on-memory**. Same as the 3, except PEs are placed on the same die as main memory.

Results are summarized in Table 6.11. All calculations are by Suspense, and the cost metric per PE includes the supervisor, packaging and interconnect, power, etc. amortized over the total number of PEs. Space-time is the cost per PE times the system cycle time. The more advanced implementations have better space-time product due to improved performance and reduced cost per pin. Each of these implementations is for a single PN. Interconnections between PNs would be solely for communication. Sustaining 20 M bytes/s at 20 MHz would require only 8 wires for the datapath leaving the PN (ignoring protocol).

As previously noted capacities required for the full global communication network can be met with reasonable hardware resources.

Implementation:	PCB	Adv. PCB	MCM	PE-on-Memory	Adv. PE-on-Memory	MCM PE-on-Memory
PEs per PN	256	1024	1024	256	1024	1024
PE λ, μm	0.6	0.35	0.35	0.6	0.35	0.35
PEs per Chip	16	16	16	2	8	8
On-chip Memory per PE, K bytes	0.5	2	2	0.5	2	2
PE Speed, MHz	40	67	67	43	76	76
System Speed, MHz	33	49	65	34	49	65
PN Area, cm^2	2090	2230	229	845	1210	291
PN Watts	294	522	109	325	278	74
W/cm^2	0.15	0.23	0.36	.38	.23	.25
Cost-metric, per PE	65	44	48	111	104	108
Space-Time	1.97	.89	.74	3.26	2.12	1.86

Table 6.11: Preliminary System Assessment

PE Chips with Off-Chip Memory The simulations confirm that 16 processors per PE are preferred at the system level. For the PCB implementation, 16 PEs per chip had an S-T product only 2% better than 8 and 10% better than 32. The advanced PCB implementation was more extreme, the optimum S-T 28% better than 8 and 19% better than 32.

The early analysis also indicates no major implementation constraints. Module-level densities are well within the realm of air cooling. The PE packages themselves may require more advanced methods, such as air-impingement cooling. This is included in the cost metrics. At over 2000 cm^2 (over 18 inches on a side), the PCB is quite large, but within the realm of current commercial products. For the advanced PCB implementation with more processors the PE package area is the same. Here the PCB must have 16 layers to prevent routing from determining board area and exceeding manufacturable limits.

Advanced packaging is clearly desirable. PE speed is for the advanced PCB implementation 67% faster, but system speed is only 48% faster, indicating that interconnect delays are having a significant effect. This suggests an MCM implementation to make the system compact and reduce delays and power consumption. Here we assume that 25 cm^2 is the maximum size for a commercially viable MCM, so that the 1024 PE PN is implemented on 16 MCM substrates. From Suspense, each substrate is 19 cm^2 (about 1 3/4 inches on a side) and contains 64 PEs (4 PE die and 8 memory die). The substrate for the supervisor, its memory, and the co-processor is only 3 cm^2. With flip-chip die mounting, chip areas are bare die; 0.2 cm^2 for the PEs and 2.0 cm^2 for memory. This packaging results in 30% higher speed at only 10% higher cost. The MCM space-time given in Table 6.11 does not include the cost of interconnecting the 17 substrates (16 PE sub-arrays plus the supervisor), however, whereas the PCB implementations include all PN packaging and interconnect cost, so the MCM version could loose its 20% space-time advantage, but still remain desirable for high-performance or highly compact application areas.

PEs on the Memory Chips Customizing commodity DRAM chips by adding MISC PEs has several system implementation advantages but does not appear cost effective. Placing the PEs on the same die as main memory significantly reduces pin requirements and avoids chip-crossing delays. The $O(P)$ memory links avoided and only $O(\sqrt{P})$ communication links are required, scaling within planar packaging limits. With 8 PEs placed on a 64 M bit DRAM (1 M byte per PE), 380 pins are required; more pins are currently used on leading-edge RISC chips. These savings offset some of the cost penalty of a semi-custom as opposed to a commodity DRAM.

The "PE-on-Memory" implementation places 2 PEs on a 16 Mbit DRAM and the "Adv. PE-on-Memory" implementation places 8 PEs on a 64 M bit DRAM. Both use PCB for chip interconnect. The last implementation is the same as the MCM implementation except 8 PEs are placed on a 64 M bit DRAM. Suspense results are summarized in Table 6.11. Placing PEs on the memory die produces some reduction in system power and better than halves the system area of PCB implementations. Although the PEs are a negligible addition to memory area, the reduced fabrication volumes and accompanying yield decreases[7] more than doubles cost per PE and overwhelms savings in pin cost. Even if pins are ten times as costly as estimated the cost disparity would only fall from about 2.3 to about 1.7.[8] System speeds are similar, so that placing PEs on the memory chips makes space-time more than twice as bad as when using commodity DRAM.

A Hypothetical Advanced Implementation Lee presents high-density protyping technology for stacking MCMs[9] which can dissipate up to 250 watts [LAS89]. Each MISC sub-array substrate would require about 900 interconnections (2-d mesh communication, control, power and ground). Using this technology, a 1 K PE PN, including 1 GByte memory, supervisor, and co-processor could be packaged within a 12 cm (5 in) cube. A MISC Machine with 8 PNs would fit within a 1 foot cube, consume about 1 KW of power, and offer nearly 500 sixteen-bit GOPS.

6.4 Design Alternatives

Having established a base architecture, this section reconsiders architectural options based on the simulations in Appendix C. The options include adding a significant amount of on-chip memory (increasing die area) and reducing bandwidth to off-chip memory, reducing

---

[7] 1.0 defects per cm^2 is used here rather than the normal 0.5 for commodity DRAM.

[8] At 8 to 16 PE per chip, the $O(\sqrt{P})$ advantage has little effect; memory capacity (and accompanying die area) prohibit significantly more PEs per chip.

[9] Die stacks are not a viable option due to the limited interconnect. Emerging products use lines with a 310 μm pitch one edge of the die stack [Lei90]. Even for large memory die this gives only 40–50 lines, and it is doubtful that more than 100 lines will become available. This is clearly insufficient for PE chips. A single DRAM chip is already large enough to serve a PE, and stacking several die to serve even a 4-PE chip would require more lines than would be available due to duplicating both data and address lines. Even without local addressing, multiple datapaths would limit a die stack to serving at most 4 PEs. If both PEs and memory are fabricated on the same die, inter-PE communication lines would also prohibit using die stacks. It would be better to add pad arrays to a single large DRAM chip.

communication link width to reduce pin count, increasing the width of the PE data path, and additional functional units.

6.4.1 Array Memory

Memory referencing profiles from the simulations indicate that one parallel memory port per PE is not necessary if there are 32 K bytes of on-chip memory per PE.[10] For SDM, only the physical location addresses are accessed exhaustively. They represent 1/8 of the memory locations but 99% of memory references. For the example SDM (1 K × 256 K), 32 K bytes are needed to hold all the physical location addresses. Analysis of the simulation profiles shows that with a 32 K byte on-chip memory per PE sharing one 1–4 M byte off-chip memory per 4 PEs slows the SDM algorithm by 3%. Sharing one off-chip memory per 16 PEs gives a 15% slowdown. Semantic networks do not exhibit any clustering of memory referencing, but the minimum data for marker propagation (node subsumption code, marker state, and outgoing edges) is quite small, 16 to 32 K bytes per PE for the 256 K node example. With this amount of memory on-chip, sharing off-chip memory ports has negligible slowdown. For refinement, off-chip memory is accessed only when loading a graph into the PE registers, after which thousands of cycles are executed. Overall, 1% of references are to off-chip memory and the remaining 99% are to the register set so that even with no on-chip memory sharing off-chip memory ports is only a 1% slowdown. Without any on-chip memory other than the register bank, SDM and SN would be slowed down by a factor nearly equal to the number of PEs sharing a memory port.

Is a large on-chip memory practical? Suspense shows that 32 K bytes per PE of DRAM using the MOSIS IC process and a 2-transistor DRAM cell occupying $190\lambda^2$ [Spe91] doubles die area and more than halves yield, increasing PE bare die cost by a factor of 5. This overwhelms any pin cost savings. Using a commercial DRAM memory process allowing a $60\,\lambda^2$ cell (advanced DRAMS have cell areas as small as $10\text{–}20\,\lambda^2$) does not increase die area. Reducing pin count by multiplexing memory ports reduces die area in addition to reducing pin costs. Combined, sharing 1 memory per 4 PEs reduces the total PE chip cost by 33% and total system cost by 12%.[11]

Combining slowdowns and cost savings, sharing a memory port between 4 PEs improves system cost-performance by 10%. This is insufficient to warrant revising the array architecture and re-analyzing the results at this stage of MISC's development, particularly in the light of potential difficulties. A large on-chip memory significantly increases design cost, and a memory IC process could compromise PE performance. Although many commercial products add significant logic to a memory chip, including custom logic in a commercial memory process may not be practical for prototype and small-volume systems. Relying on on-chip memory also limits scalability; larger SDMs and SNs require on-chip memory sizes that would overwhelm any pin cost savings. Off-chip memory sharing in conjunction with a large on-chip memory does provide a fall back should high pin-count packages (1–2,000

---

[10] As sharing memory ports reduces both data and address lines, it is preferable over reducing memory data-path width.

[11] Savings for additional sharing are negligible, as pins for local communication quickly dominates. For applications requiring less than 32 K bytes per PE, eliminating off-chip memory entirely (but retaining pins for later upgrade) reduces total system cost by 40%.

pins) not become a commercial reality or if major noise or power sag problems arise from switching such a large number of lines simultaneously.

6.4.2 Array Local Communication Network

Analysis has shown that there is little or no benefit to adding a bit-serial MIN, hypercube, or similar $O(\lg N)$ network to supplement the word-wide mesh array topology. Wider MIN, hypercube, or similar networks are cost prohibitive due to high pin-count across partitions between chips; MISC is already pushing the limits of projected high pin-count packages. Reducing mesh communication links from 16 to 8 data bits saves 10% in PE chip cost and 3% in system cost. SDM doesn't use the local communication network, but analysis of the simulations shows that 8-bit links increases array execution time by 42% for SN and 35% for refinement, for a weighted average slowdown of 25% over the three applications. Reducing link width significantly worsens system cost-performance Increasing link width to 32 bits serves no benefit since the 16-bit ALU would require 2 cycles to load the communication `in` or `out` registers, so this is discussed next in conjunction with ALU width.

6.4.3 PE Data Path

Analysis based on the simulations indicates a substantial speedup for a 32-bit versus a 16-bit PE, indicating that earlier calculations underestimated data widths. This is largely due to increased use of subsumption codes as the example algorithms were further developed. Table 6.12 gives the results of expanding just the PE and memory port widths as well as expanding PE, memory, and communication widths. SDM array execution time is 85% Hamming distance at the example size, and a 32-bit PE doubles speed on Hamming distance. For semantic networks, speed on the matching and other operations would also be doubled, and local communications (42% of execution time) is improved 50% if communication links are also widened. For refinement, a 32-bit PE can more than double execution speed for graphs larger than 16 nodes. Here, 92% of array time is in Boolean matrix multiplication, including the required matrix transpose, and most of this time is in communication. Widening the PE avoids iterating over the two columns of PEs required to store an adjacency matrix row of more than 16 bits, and doubles the number of graphs that can be processed simultaneously so that performance is increased by more than a factor of two. Table 6.12 includes an adjustment for the number of graphs processed.

Algorithm	ALU, memory	ALU, memory, communication
SDM	1.67	1.67
SN	1.40	1.75
Refinement	1.84	2.78
Weighted Avg.	1.7	2.2

Table 6.12: Speedups for 32-bit PE Over 16-bit PE

Despite available die area, a 32-bit ALU has significant cost penalties. Increasing the PE to 32 bits increases transistor count by 68%, including doubling the number of bytes in the register bank. Since area for chip pins still dominates, this does not increase the Suspense calculation of die area, and speed decrease is insignificant. Increasing only the memory port data paths, however, adds 512 pins resulting in a 1,600 pin package. This is well above the 1,200 pin packages projected to be economical several years from now and would increase manufacturing difficulty, so pins are counted at twice the cost of the smaller package. This increases total system cost by 36%, for an overall cost-performance improvement of 20%. Increasing the communication datapaths to 32 bits brings PE package pin count to over 2,000 pins, increasing system cost by 52% giving a 30% cost-performance improvement. If 4 PEs share a memory port, pin count with 32-bit memory and communication ports is reduced to just below the 1,200 pin estimate, with no difference in system cost. As discussed in Section 6.4.1, relying on a 32 KByte on-chip memory is as risky (if not riskier) than the higher pin-count packaging. Although there is the potential for doubling performance, this option is held in reserve as a "safety factor" due to technology risks (including potential implementation problems from even more simultaneously switching lines) and the limited cost-performance trade-off.

6.4.4 Functional Units

Additional functional units could be added to the PEs to enhance performance on a broader range of applications, including hardware multiply/divide or even a floating-point unit, a barrel shifter, a wider address unit, and more registers and flags. These have no obvious benefits for the example applications in this dissertation, and are not analyzed.

Additional local control is also outside the scope of this dissertation, but should be investigated in the future. This could be important for algorithms which include nested parallelism with complex control structures (such as the backtracking phase of subgraph isomorphism). There is sufficient room to include a substantial controller on each PE chip. As another option, simply latching a single instruction at each PE could allow heterogeneous systolic modes of operation at significantly reduced slowdown over PE masking. Either approach has the drawback of more complex compiler design or more difficult programming. Means of reducing the instruction count for each routing cycle of the coterie routing algorithm [HCWS94] should be investigated. This could reduce execution time, time variance, or both over the simple routing algorithm for general routing discussed in the Section 7.4.2.

6.5 Summary

This chapter has presented the analysis leading to the details of the MISC architecture, integrating characteristics of example algorithms and of implementation technologies. The MISC architecture itself is presented and discussed in the next chapter.

The most pervasive influence is the scaling introduced by filtering and pruning effects in the example algorithms. Sublinear scaling of the number of PEs in the problem size results in a global MSIMD architecture of multiple processor arrays. The arrays' high ratio of internal computation to external communication simplifies inter-array communication and greatly improves MISC scalability. Processor scaling also results in unusually large memories for each PE, and in conjunction with memory referencing patterns requires extremely high bandwidth interconnection to advanced off-chip memories.

Instruction profiles and intra-array communication patterns also have important implications. Local communications require both regular short-distance communication and non-uniform long distance communication, resulting in the choice of a 2-d mesh network with wide links and node-bypass circuitry. Despite being an associative processor, content-addressable memory hardware is inappropriate; the emphasis on pruning and processing complex data objects results in only a small portion of the instructions being vector-matching. The MISC PE is thereby a multi-bit ALU and register set, enhanced with special functional units. The enhancements include a one's counter, single-bit unit, and a local address adder, masking and a multiple-response resolver. Die area is available for additional features with negligible in crease in system cost.

Technology assessment shows that the high pin-count packaging required by the memory subsystem and local communication network is cost-justified despite pin costs and the large die area required for pin connections. Analysis also shows that although multi-chip module packaging is promising, other advanced techniques such as integrating process and memory die are not. Early analysis indicates that the proposed MISC architecture is an implementable, cost-effective, high-performance architecture.

7. The MISC Architecture

This chapter presents the MISC architecture itself. Section 7.1 describes the overall hierarchical architecture of global communication, supervisor processors, and multiple associative arrays. As this dissertation concentrates on the associative array and the higher levels require further application-related analysis, this first section presents viable options rather than a particular design. Section 7.2 presents details of the associative array, highlighting special features of the processing elements, array communication network, and array memory. This is followed by a description of proposed implementations in Section 7.3. The hypothetical implementations form the basis of early analysis of MISC system cost-performance, also presented in Section 7.3. Finally, Section 7.4 presents examples of machine-level programming for the key macro operations of the example algorithms in preparation for the evaluation of the MISC architecture in Chapter 8.

7.1 System Architecture

This section describes MISC's global architecture. Figures 7.1 and 7.2 along with the associative model (Chapter 4) form an informal specification for the MISC architectural family.

As seen from Figure 5.17 (page 112, a small array of a thousand or fewer PEs is appropriate for the class of target algorithms. The MISC global architecture is thereby MSIMD (multiple SIMD) with small, independent arrays, and scales by adding arrays rather than increasing array size, as shown in Figure 7.1. Each array is part of a processor node that also includes a RISC microprocessor for scalar operations and array instruction issue, an array control unit, and a global communication processor. An additional coprocessor to support special features of knowledge processing might also be incorporated. After discussing the advantages of the MSIMD approach, this section presents alternatives for implementing the global communication network and supervisors, and summarizes the associative arrays.

Although presented as a homogeneous architecture, with each PN the same, the MISC framework could support heterogeneous processing. Some PNs could have larger or smaller associative arrays or could substitute a numeric vector processor, a disk I/O subsystem, a large shared memory, graphics processors, or application-specific computational units for the associative array. High bandwidth system I/O can be achieved by tapping into the associative arrays' local communication network or by one or more PNs dedicated to input-output. Similarly, communication with a front-end workstation can be via a dedicated PN or via the global communication network.

7.1.1 Multiple-SIMD

With the processor scaling effects noted in Chapter 5 and communication requirements analyzed in Chapter 6, a full routine (such as one of the example algorithms) and all its data can reside on a single small MISC SIMD array. Despite a high rate of communication with the array, external communication is essentially the routine parameters (query

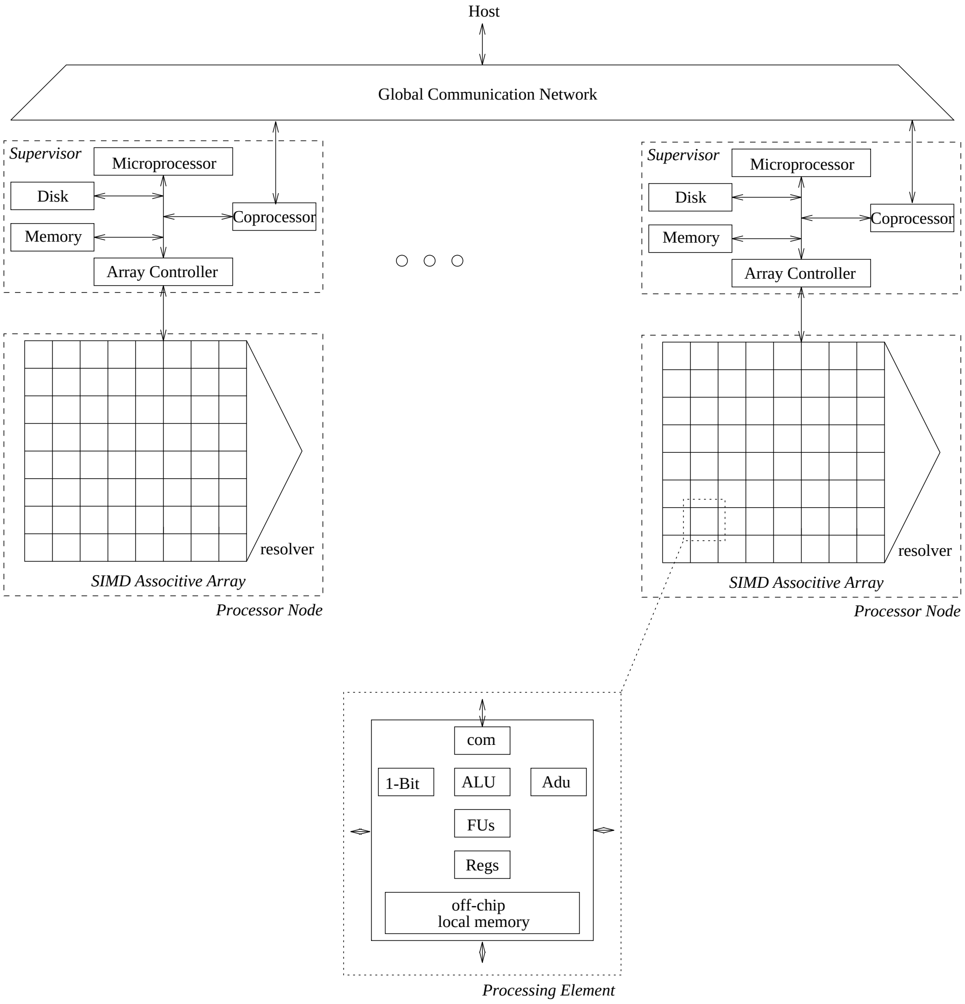


Figure 7.1: The MISC System Architecture

input/output) and the data rate is quite low.[1] These factors and other benefits strongly suggest an architecture employing SIMD arrays of bounded size, with the system scaling by adding arrays rather than increasing the size of a single array.

The multiple-PN MISC architecture combines MIMD and SIMD processing into MSIMD (multiple-SIMD). This has several advantages for AI processing, and parallel computing in general. MIMD processing is available for coarse-grained process-level parallelism. With each array sized for a typical knowledge-processing routine, multiple arrays would be used for intelligent systems featuring multiple routines or agents, or extremely large knowledge bases. Partitioning the PEs into small sub-arrays each with its own supervisor greatly increases control flexibility and improves PE utilization; each PN works on a different subtask without having to mask out large numbers of PEs. The MISC supervisor performs much of the work that would be performed in the front-end and is much more closely

---

[1] High-bandwidth input/output external to MISC, such as that required by vision processing, would be handled through the array edges as in the MPP.

integrated with the associative array, greatly reducing communication overhead between the front end and the array. Similar to the MasPar, the front-end, supervisor, and array control unit can run asynchronously.

Partitioning into multiple SIMD arrays greatly relieves inter-processor communication requirements. Each PN has an extremely high ratio of internal computation to external communication, which reduces global communication bandwidth needs. If all PNs were in a single array the otherwise inter-PN communication would be part of the array "local" communication and would have to pass through a large number of intermediate PEs. With the 2-d mesh, local communication time would be $O(\sqrt{N})$ rather than $O(\sqrt{N/P})$ where N is the total number of PEs in the system and P is the number of PNs; with a hierarchical inter-PN network, global communication time is $O(\lg P)$. This approach does, however, pose a communication bottleneck if it is necessary to combine multiple arrays into a single larger array. This could be remedied by wider bandwidth on the global network or by interconnecting array edges across PN boundaries.

The MSIMD architecture has other implementation benefits as well. The array local communication network is bounded in size, improving performance and minimizing scalability drawbacks for simple topologies. Communication between PNs can be asynchronous or self-clocking. A single clock distributed over the full system is not required; clock and instruction-skew and signal integrity problems are limited to a single PN, relieving a significant implementation difficulty in prior massively-parallel architectures and permitting higher array clocking speeds. Reduced global bandwidths resulting from communication clustering makes expanding systems to larger numbers of PNs easier and more cost-effective.

7.1.2 Global Communication

Communication between a large number of PNs does not appear to be a problem. With an entire routine in a single array, inter-array communication becomes simply input and results. The analysis of Chapter 6 established that global communication is less than 1 MByte/s per processor node despite peak performance of 60 GOPS per node. This is supported by the simulations described in Appendix C. At this communication rate, a single high-speed bus, such as the 100 MByte/s Futurebus II standard or the TMS320C40 [Tex91] DSP chip's parallel processor global bus, is sufficient for MISC systems of a hundred PNs. A 2-d mesh network with 10–20 MByte/s links, such as those in the TMS320C40s and the T9000 Transputer chip [Pou91], could also support a hundred PNs.

Higher global communication rates would require more advanced global communication networks. If one of the example algorithms is distributed over multiple PNs, as would be needed in the case of much larger data sets or smaller arrays, communication per PN would increase to 3–20 MBytes/s per PN. Other algorithms and applications may also have higher communication rates. The 200 MByte/s links of the Cray T3D [Oed93] processor nodes could support as many as a thousand MISC PNs in a mesh topology, even at 20 MByte/s per node. With gigabyte/s networks emerging [$B^{+}94$], higher bandwidth PNs or much larger MISC Machines should be feasible, with scaling limited more by available high-level parallelism than by global communication hardware capabilities.

The author hypothesizes that hierarchical networks, such as fat trees and hierarchical multibus topologies are preferable for MISC and AI applications. With hierarchical topologies, larger systems can be built by combining smaller, possibly pre-existing systems; only the additional level of communication needs to be added. The smaller systems have negligible burden for future expansion. Hierarchical networks also simplify partitioning a system among multiple users. Networks such as fat trees can better exploit communication locality and adapt to particular communication patterns [Lei85], especially important for AI applications, reducing hardware requirements and cost. In hierarchical global communication topologies, each node in the communication network could include processing, resulting in a multi-level hierarchy of "supervisor supervisors." Additions to basic router functionality could range from reduction (including message combining) and broadcast operations, to multi-level caching with coherency management, to high-performance microprocessors (possibly even including an associative array for support operations, *i.e.* a PN itself).

Where global communication remains small and increased latency can be tolerated, a MISC Machine could be implemented as a *distributed parallel processor* using a local area network. A MISC node would be a workstation with a single PN as an accelerator. In this incarnation MISC could be part of a heterogeneous parallel processing environment such as PVM 3 [Gei93].

As several viable options exists, a choice is not necessary at this time; this dissertation focuses on the associative arrays. The MISC analysis and evaluation in this dissertation is based on the simpler assumptions; the global communication net includes only basic router functionality. Inter-PE communication is exclusively message passing, with no shared memory. Considerable further analysis would be required to evaluate the system-level alternatives. Analysis must await profiles of large multi-PN application simulations, and is beyond the scope of this dissertation.

7.1.3 The Supervisor

The PN supervisor performs four major functions. It issues high-level instructions to the associative array, performs scalar/MIMD computation, handles disk and other I/O, and mediates global communication. Each PN includes a microprocessor, a co-processor for communication and accelerating certain AI operations, and an associative array controller. Figure 7.1 shows the supervisors.

The Supervisor Microprocessor

Each PN has a supervisor microprocessor executing the node program and issuing instructions to the associative array. These are high-performance off-the-shelf microprocessors to take advantage of large commercial design efforts and the economies of high-volume production and existing assembler, compiler, and operating system software. Although there are several excellent experimental research projects designing microprocessors specifically for parallel processing and AI applications (including [A$^+$93a, L$^+$89]), it is doubtful that their special features will provide speeds significantly greater than those of lower-cost commercial processors.

The simulations described in Appendix C show that a third of the execution time for SDM and semantic networks is executed by the supervisor, even with a 500 MIPS microprocessor. Although this ignores the potential of overlapping supervisor and array processing,[2] a high-performance supervisor chip is clearly needed to avoid throttling the associative array.

Performance for commercial chips has historically increased by 40% to 50% each year, so that 500–1000 MIPS processors should be commodity items within a 5-year time frame. Table 7.1 presents current microprocessors used as the basis of extrapolation. The performance figures for the Alpha microprocessors are SPECint benchmarks; the Alpha 21164 has a peak rate of 1,200 MIPS.

Microprocessor	Appx. MIPS	Release Date
PowerPC 601	70	1993
Pentium	100	1993
Alpha 21064A	175	1992
TMS320C40	200	1991
T9000	200	1991
Alpha 21164	330	1995

Table 7.1: Current High-Performance Microprocessors

Future generations of these processors could be used for the MISC supervisors. Two of the microprocessors in Table 7.1 are designed specifically for parallel applications, and include sophisticated communication support, both software and hardware. The TMS320C40, a Texas Instruments DSP chip designed for parallel operation, includes both shared-bus and serial point-to-point communications. The TMS320C40 supports shared memory over a single common bus. The T9000, an Inmos parallel processing chip, includes serial point-to-point communication. In both chips, communication duties are off-loaded from the CPU. The Alpha chips include hardware for shared memory and message passing, supporting parallel processor communication. The Alpha 21164 should move from 0.5 μm to 0.35 μm CMOS in 1995 [Rya94], improving sustained performance to 400 MIPS or more, and NEC has recently announced an experimental 500 MHz 32-bit CMOS RISC chip [S^{+}94].

This dissertation does not include an evaluation of the microprocessor alternatives, and instead concentrates on the associative array. System early analysis assumes a separate communication co-processor as a margin of safety.

The Co-Processor

Most off-the-shelf microprocessors lack many features needed for MISC, including interprocessor communication. Further, AI processing introduces requirements in addition to those of parallel processing. AI's subroutine- and recursion-intensive codes require rapid

---

[2] All of the example algorithms are essentially querying operations. While a given query is being processed, results from the prior query can be formatted and transmitted and input for the next query received and formatted.

task switching. Rapid task switching (typically accomplished through hardware stacks) is also important to general operation, hiding global communication and shared-memory reference latencies, long associative array operations, and cache refills by executing other operations while awaiting results. Mennemeier [Men91] details an external co-processor for performing rapid task switching and state saving for conventional microprocessors. Others have shown that 2–4 execution threads are usually sufficient to keep a processor busy in a multiprocessing environment [A^+90]. Multi-threading, in conjunction with a superscalar microprocessor would address medium-grain parallelism. Kogge and others [Kog91, Kog82, Ode91, PJK89, A^+87] have shown significant performance improvements from stack architectures for production systems. While highly parallel operations specific to AI processing such as search are handled in the associative array; those with limited parallelism may benefit from additional functional units as elements of a co-processor to the supervisor microprocessor.

Although the SEGUL programming model uses message-passing, it may prove desirable to implement inter-processor communication and other global operations through shared memory. In addition to the memory sharing itself, this would add the requirement of cache coherency, as the high-performance microprocessors require cache. Several studies demonstrate the practicality of multi-bus cache coherency without requiring any bus to be common to all processors [ASHH88, TD91].

In summary, a co-processor might provide the following functions.

- Managing inter-PN communication independently of the supervisor. This includes message routing and queuing incoming messages and tasks.
- Managing shared memory and cache coherency.
- Providing multiple hardware stacks for rapid task switching.
- Providing functional units for high-performance AI-related operations.

Designing a co-processor appropriate for a 500–1000 MIPS supervisor would be a daunting task. It is likely that the co-processor functions would be implemented using standard communication processors (such as ATM, asynchronous-transfer mode chips) and other off-the shelf components. The MISC simulations to date do not include any special AI support in the supervisor, giving conservative results.

The Array Controller

Finally, the PN needs an array control unit (ACU) to serve as an "instruction amplifier,"[3] off-loading array instruction sequencing from the supervisor. The controller accepts single high-level instructions from the supervisor microprocessor and converts them into a sequence of low-level microinstructions issued at a rate much higher than the supervisor itself could issue them while the supervisor continues with other tasks. Instruction amplification includes converting from single-word supervisor macroinstructions to much wider array microinstructions. The macro-expansion includes managing looping and conditionals as well as virtual-processorization and array variable and memory management. The controller is also responsible for queuing array input data and results for the supervisor and includes a writable control store so that highly optimized macro-instructions can be compiled for

---

[3] The term introduced by Hillis [Hil85].

particular applications. As a result, the ACU assumed in the system early analysis includes the following blocks. These blocks form the basis of transistor count estimates for projecting system cost and were not directly simulated.

- 1K x 64 bit or larger writable control store
- 32 x 32 bit dual-ported parameter and address registers
- 16 x 20 bit address stack
- 16 x 256 data stack (array data queue)
- 20-bit address ALU with shifter and zero detect
- one or more 16-bit loop counters
- instruction decode, microsequencer, control and multiplexing logic.

Alternatively, the sequencer could be a second high-performance RISC or DSP processor, possibly incorporating parallel communications as well. A 500 MIPS processor could execute 8 instructions for each cycle of the 60 MHz associative array. Sequencing could be handled through compiled microcode sent to the array via DMA (direct memory access)channels.

Without the controller, array performance would be throttled by the rate at which the supervisor could issue low-level instructions and the supervisor would be unable to perform any computation during lengthy array operations. A typical instruction issue by the supervisor would consist of placing input data on the data stack, issuing a macro instruction (consisting of an address in the microstore plus additional parameters such as the number of times to loop or other terminating conditions), potentially performing serial computation independent of the array, then collecting array results from the data stack.

7.1.4 The Associative Arrays

This dissertation demonstrates that it is worthwhile to customize SIMD arrays for AI. Small additions can reap significant performance improvements; the MISC SIMD PEs are small and relatively simple with a reasonable design time. Many SIMD chips are used in a single machine, greatly reducing the manufacturing cost differential, resulting in a cost-effective system. As shown in Figure 7.1, the associative array is a 2-d mesh of processing elements supporting general SIMD operations and specific AI primitives, and emulating word-parallel CAM. The next section discusses the array in more detail. Here we discuss a high-level view of its components.

MISC uses a 2-d mesh with toroidal wrap-around for communication within each array. To improve performance on long distance communications, MISC extends the mesh to a *bypass*-mesh in which communication signals are allowed to propagate through a PE without going through intermediate registers and routing steps. Long-distance communication occurs in time effectively independent of distance for regular patterns,[4] supplementing the adjacent-neighbor connections of a conventional mesh. The bypass switches also allow partitioning the array into sub-arrays and support a variety of communication patterns.

---

[4] The MISC architecture specifies bypass hardware, and a regular communication pattern requires at most 2 PE cycles (and typically 1 cycle) regardless of distance, in a 1024 PE array. Mesh communications in the MasPar MP-2 require at least 1 cycle per PE distance (up to 16 cycles in a 1024 PE array, plus at least 64 cycles set-up) [Mas92b].

Several prior massively parallel processors incorporate a nearly identical network for reconfigurability, providing switching for fault-tolerance and improving performance for a variety of algorithms, especially image processing algorithms [LS92]. A separate multiple-response resolver network determines if a particular flag is set in any PE, and iterates to the next PE with that flag set, in logarithmic time.

Array memory features a separate enhanced DRAM (EDRAM) for each PE. Access is pipelined, so that data from off-chip memory is fetched in the cycle before it is used. With the reduced cycle time provided by the EDRAM's on-chip row-cache, each PE can be supplied with a word of memory each cycle under most circumstances.

Figure 7.1 gives an overview of the PE microarchitecture. In addition to communication and memory interfaces, each PE has a general-purpose word-wide ALU with a register set supporting matching, logical, and numerical operations. A separate 1-bit ALU speeds single-bit manipulations and executes operations on 2 bits or flags. Additional functional units support operations specific to AI and associative primitives. An adder and additional registers support full local addressing. Memory can be addressed globally by the array controller or locally by its PE and the supervisor can access the array memory through the individual PEs.

7.2 The Associative Array

This section presents a register-transfer level (RTL) description of the base associative array design, MISC-1. Actual implementations might include variations such as a larger register set, additional functional units, *etc.* The design presented is the result of the algorithm, architecture, and technology analysis of this dissertation, and serves as the basis for evaluating the research results.

7.2.1 The PE Microarchitecture

The MISC array processing element consists of a standard SIMD PE "core" supplemented by functional units to speed the key operations of knowledge retrieval and processing, in many cases emulating CAM. The ALU and register set core form a conventional processor insuring a general-purpose architecture.

Figure 7.2 presents the array processing element. ALU communication is by 3 busses: A and B for input to and R for output from the word-wide ALU. Two operands are delivered to the ALU and the result stored in one clock cycle. Each register and most functional units can place data on either the A or B bus in the first phase of the clock cycle and accept a result from the R bus in the second phase. The PE microarchitecture description is grouped into the main ALU and register set, the 1-bit ALU, special-purpose functional units (FUs), and the address unit (AU). The PE description is followed by discussion of the array memory and local communications.

Word-Wide ALU

The conventional main ALU supports standard numeric and logic operations. Implementation details for analysis are based on the ALU in the Brown Systolic Array, which

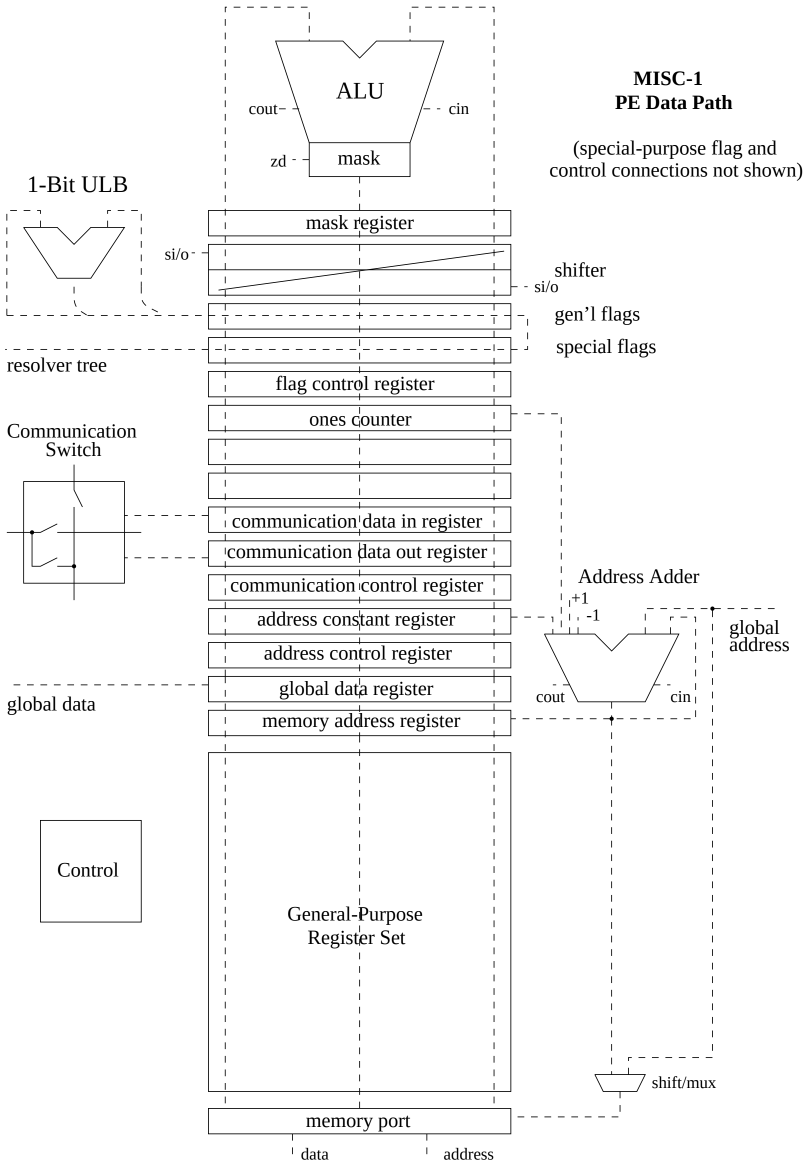


Figure 7.2: The MISC PE Microarchitecture

is in turn loosely based on the Mead and Conway's OM2 ALU [HL91, MC80]. The circuit features great flexibility at the penalty of slightly reduced speed and increased transistor count.[5] It includes "universal logic blocks" (ULBs) which are capable of generating all Boolean functions of 2 or 3 inputs, and a Manchester carry chain. Each bit position uses two 2-input ULBs to produce generate and propagate signals for the Manchester carry and one 3-input ULB to generate the result. Carry-in and carry-out are special-purpose flags included in the 1-bit section described below, and zero detect indicates if the result of any

---

[5] Appropriate for a prototype or experimental machine. A subsequent implementation would likely provide faster though less general logic.

ALU operation is all zeros. A shift register of length twice that of the ALU width is included. Each half can be accessed through the word-wide data busses, permitting manipulation of double-word length values. Two special-purpose flags, S1 and S2, serve as shift-in and shift-out flags to perform logical, arithmetic, and circular shifts. This is primarily to support multiplication, as would be needed for calculations such as the ℓ_2 (Euclidean) distance norm and integer-valued neural nets.

Table 7.2 partially lists the large number of single-cycle functions computable by the ALU; ALU inputs are a and b, and carry in is c. The table includes all logical operations on 2-bits and the most useful MISC ALU operations. Magnitude comparisons require the use of the zero detect, carry in and carry out flags. The "identity functions" are useful as constants and for transferring the contents of one register to another. Table 7.3 lists additional single-cycle operations.

Identity Functions	Logical Functions	Arithmetic Functions	Comparison Functions
0	$a \wedge b$	$a + b$	$a = 0$
1	$\overline{a \wedge b}$	$a + b + c$	$a \neq 0$
a	$a \vee b$	$a - b$	$a = b$
b	$\overline{a \vee b}$	$b - a$	$a \neq b$
c	$a \wedge c$	$a - b - c$	$a \leq b$
$-a$	$a \oplus b$	$b - a - c$	$a < b$
$-b$	$a \oplus c$	$a + 1$	$a \geq b$
$\overline{a}$	$a \oplus b \oplus c$	$b + 1$	$a > b$
$\overline{b}$	$a \odot b$	$a - 1$	
	$a \odot c$	$b - 1$	
	$\overline{a} \vee b$		
	$a \vee \overline{b}$		
	$\overline{a} \wedge b$		
	$a \wedge \overline{b}$		

Table 7.2: ALU Output Functions (based on [Hug91, MC80])

Register Set

The MISC-1 on-chip memory is a conventional multi-ported (1 read, 1 read/write) register set. For maximum throughput and flexibility the registers can supply 2 operands and store one result for each PE clock cycle. Later work may relax this requirement, as typically one of the operands is provided by the array controller (global data), a communications link, or one of the other special-purpose FUs.

There are thirty-two 16-bit registers as a balance between area and speed. Evaluation of application characteristics to date indicates that there is little or no benefit to having a large on-chip "data object buffer" as data objects are either small enough for the register bank, or are many kilobytes. Analysis of workload beyond the example applications in this dissertation may indicate a larger (or smaller) register set, several kilobytes of on-chip DRAM, or in application-specific configurations a bit-cell CAM.

Name	Function
zero detect	$R = 0$
logical shift	
circular shift	
arithmetic shift	
select a or b on c	$ac \wedge b\overline{c}$
select a or b on $\overline{c}$	$a\overline{c} \wedge bc$
multiply step	
divide step	
there exists matching bit(s)	$\exists i : a_i = b_i$
there exists mis-matching bit(s)	$\exists i : a_i \neq b_i$

Table 7.3: Additional ALU Operations (based on [Hug91, MC80])

1-Bit Unit

The 1-bit unit, a special feature of the MISC architecture, consists of a 2-input ULB, single-bit flags, and 2 up-down counters.[6] In MISC the 1-bit unit is used for single-bit manipulations and flag management rather than as a bit-serial basis of all computation, and is closely integrated with both the word-wide ALU and the PE control section. The ULB is capable of executing all the "identity" and logical operations on a and b listed in Table 7.2 for single-bit values.

The unit includes 32 flags. Sixteen of these are general-purpose and sixteen are special-purpose. Any two can be selected as inputs to the ULB and the result stored in any flag in a single PE clock cycle. This requires that the flags be implemented as 4-ported register cells (2 read, 2 read/write) to provide orthogonal access as both words and single bits. Flags are normally selected under global control, though the general-purpose flags can also be accessed as a two 1-bit wide stacks via the up/down counters. The general- and special-purpose flags, as well as the up/down counters are also accessible to the word-wide ALU for *en masse* transfers such as save/restore to memory and debugging visibility.

Table 7.4 shows the assignment of the 16 special-purpose flags. The write enable flag allows multi-write operations, conditional on prior results at each PE. This supplements dedicated flag signal paths used in PE control, including housekeeping bits for multi-word operations and communication handshaking bits. The ULB "identity" functions allow setting and resetting flags, selecting conditions for conditional instructions (such as zero detect and carry out), transferring the contents from one flag to another, and maintaining match, tag, and other flags for CAM emulation.

With the general-purpose flags on the A, B, and R buses, the 1-bit unit ULB can manipulate individual bits in regular words. Such logical instructions are important for set operations. The ability to independently select and store specific bits is also important in operations such as transpose and multiply on packed Boolean matrices. The up-down counters provide limited but important local control. In addition to stack operations in

---

[6] This is similar to the CM-1 PE (excluding the up-down counters), which uses a 3-input ULB connected to both flags and memory. The third ULB input and the direct connection to memory are unnecessary for MISC as the bit-serial operations operations they support are more quickly executed on the word-wide ALU.

ALU	1-Bit Unit	Memory Adder	Other
carry in carry out zero detect shift in shift out	global data communication stack overflow stack 1 empty stack 2 empty	carry in carry out	match resolver tag PE enable write enable

Table 7.4: Special-Purpose Flags

regular expression matching algorithms [Lee86], the counters could locally determine the number of instruction executions (or carry outs, *etc.*) before a PE sets its disable flag.

It may be desirable to perform 1-bit, counter, and word-wide ALU operations simultaneously, such as in PCAM's technique for weighted Hamming distance. This is briefly discussed in Section 7.4. Example profiles indicate that 15% of the operations are single-bit, placing a 15% upper bound on speedup.

Special-Purpose FUs

In addition to the 1-bit ALU and local address unit, other special-functional units are included in MISC-1 in response to AI application characteristics. The special-function units can also be used as regular registers.

Mask Unit The mask unit is a mask register and bank of and gates interposed between the ALU output and the R bus. The gates add little to data path delay as they form part of the ALU output bus driver chain. The gates serve to set specific bits in any result output to 0 as specified by the mask register, which otherwise is a normal register. This operation allows masked compares in a single cycle, more efficiently emulating a bit-cell CAM. It is useful in other bit manipulations, such as extracting individual bytes from a wider data word.

Ones Counter The ones counter computes the number of ones in any ALU output, tremendously speeding macro operations such as Hamming distance and Boolean inner products and correlation. Like a register, it receives input from the R bus and can place the computed result on the A or B bus 1/2 clock cycle later. Output can also be sent directly to the address adder for accumulating counts over multiple words. The circuit for the ones counter itself is a tree of 1-bit full adders, similar to the carry shower described by Foster for counting the number of responders in a CAM array [Fos76].

Resolver Tag The resolver tag interfaces the PE into the array-wide multiple response resolver. The tag can be reset in a single cycle directly by the resolver without the identifier from the controller or supervisor. Unlike many associative processors, the resolver tag can be set by a logical combination of any of the numerous general- and special-purpose flags. Also, possibly unique to MISC, the resolver PE-select vector (1 bit per PE) can be used to set any other flag.

Global Data The last special-purpose register receives global data from the array controller and integrates it into the MISC PE operand selection and result buses. In addition to providing broadcast operands, the supervisor can access any PE's memory through this register.

Address Unit

Each MISC PE includes an address unit (ADU) for true local addressing. This avoids the additional instruction cycle needed if the main ALU were used to calculate addresses. The ADU includes a separate adder, 3 special-purpose registers, and data multiplexers. The 12-bit adder allows individual word addressing within a 4096 word range when used as the lower bits of the complete address (the remainder provided globally by the array controller), or selection of one of 4096 pages (such as data objects) when used as the upper address bits with addressing within the page global by the array controller. Two of the registers, memory constant and memory address, are accessed as normal registers but have additional data paths directly to the address adder. The third, address control, sets the address unit multiplexers.

One multiplexer selects the between memory-constant register, the ones counter, +1, or -1 as one of the inputs to the adder and another multiplexer selects between global address bits or the memory-address register as the other input to the adder. The sum goes directly to the memory address register, which acts as an accumulator. The sum also goes to a third multiplexer, which determines whether the memory port receives the upper local address bits, the lower local address bits, or the global address.

The ADU has several addressing modes unavailable on most SIMD machines:

Indexed (local) The address is the memory-constant plus a global value. Particularly useful for table look up.

Indirect (local) Any register is transferred to the memory-address register (the adder is not used).

"Stride" (local) Incrementing the address for the next operation by a different value for each PE (the memory-constant).

Stack (local) With address increment/decrement each PE can push and pop at different times, provided the total numbers are equal over a given control structure (see [Ble90]).

The possible data-structure access modes are multiplied by the option of selecting the resulting address as the upper or lower address bits, offering important opportunities for more complex locally-manipulated data structures.

Local addressing is particularly useful in table look-up operations such as token matching distances and low-resolution functions. In addition to sequential access from a locally determined base, increment and decrement (±1) provides stack-mode addressing and use of the AU as an up-down counter. As in the 1-bit unit, this supports limited local control, allowing a PE to disable its operations after a locally-computed or locally-stored number of steps or conditions. Controlling the number of shifts in a floating point operation is a particularly important example.

Control

Each PE's control unit determines PE operation, combining global instructions with local state. It is responsible for selecting the condition for enabling PE operation (prior result zero, ALU carry-out, *etc.*) as well as for configuring adder carry-in, shifter shift-out, the response resolver, and others. MISC features a significant degree of *local control* via configuration registers shown in Figure 7.2.

- The communication control register configures the bypass-mesh switches and controls sends.
- The 1-bit unit control register controls the 1-bit stacks and instruction options.
- The address control unit selects the two inputs to the address adder from several sources and selects the source of the off-chip address.

Different PEs' control registers can be loaded with different data, resulting in different operations in different PEs. Additionally, the communication and 1-bit unit control registers select alternate operations based on the PE state for their respective units. This is discussed further in subsequent sections. In addition to selectively routing special-purpose flags and enabling different operations, each PE's control unit decodes its local control registers.

7.2.2 Local Communication

The MISC array uses a 2-dimensional mesh for inter-PE communication, taking advantage of wide data-path links and bounded array size. The network is enhanced to allow each PE to bypass signals through without any intermediate register transfers. The bypass is asynchronous rather than clocked as in the MasPar MP-2, significantly speeding long-distance transfers. The proposed implementation also permits forming independent sub-arrays. Array communications also includes a multiple response resolver tree typical of content-addressable processors.

Bypass-Mesh Network

The PEs' interface to the array local communication network through the communication data and control registers, which are accessed like other registers. The control register determines which communication paths are opened or closed at this PE by storing several patterns which are selected according to the global instruction. Communication paths can also be selected based on conditional flags, or controlled globally. As shown in Figure 7.2, connections for the X (east-west) and Y (north-south) can be either broken (with the possibility of data intercepted or sent by the PE) or closed, bypassing the PE (with the option of eavesdropping on values being transmitted). Additionally, the X and Y data paths can be connected for cut-through routing. The switches are bidirectional, and the PE can send and receive data from either side of the switch. There are 7 useful switch configurations for each direction plus "corner turning" as shown iconically in Figure 7.3. There is a 1-bit path in addition to the word-wide datapaths, which can be used for handshaking or as a second 1-bit channel for systolic algorithms.

The bypass-mesh network is highly flexible, providing numerous communication patterns. Figure 7.3 also shows several examples. Nearest-neighbor connections support local computations as well as systolic algorithms, and row/column broadcasting is important in operations such as matrix multiplication. The coterie pattern is useful in variable-distance communications, such as in reductions and inserting an element into a physically-ordered tuple. Corner turning, in conjunction with the coterie pattern and the 1-bit path, provide fast cut-through routing for general permutations. The switches permit dividing the array into sub-arrays for communication, providing communication support for nested parallelism and segmented versions of reductions and similar operations. Additionally, array edges are switched (off-chip) permitting overall configuration as a mesh, torus, and skewed torus, the latter supporting 1-d mesh and ring communication patterns.

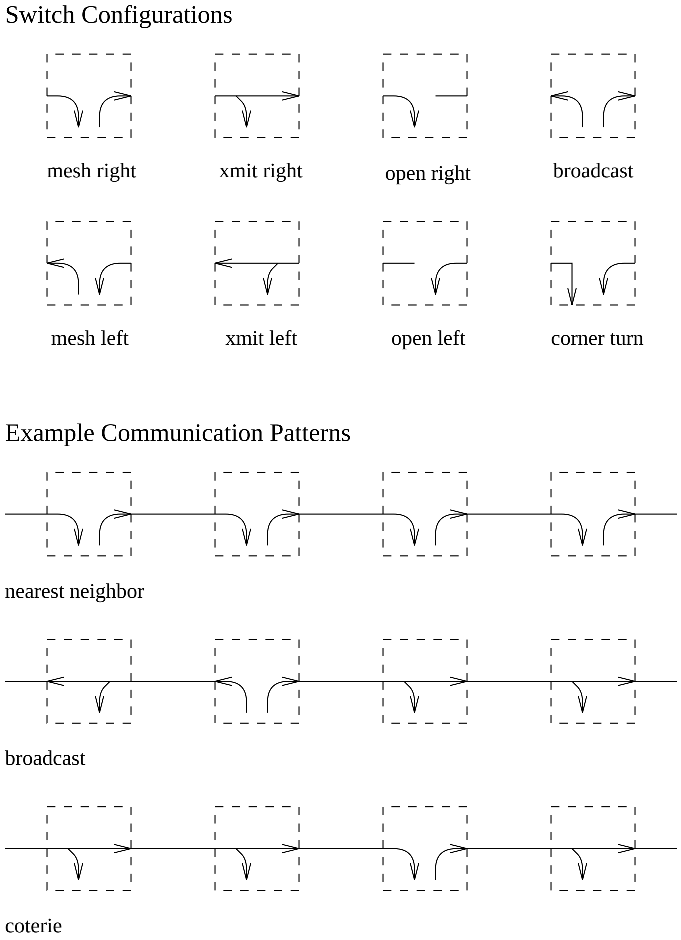


Figure 7.3: Local Communication Configurations

Communication patterns may be set up in one of three ways: global control, locally selecting pre-stored switch settings, and locally computing switch settings. Uniform patterns can be specified under global control. Non-uniform patterns can be set by storing different switch settings in the communication control register and selecting between them based on a local calculation. For example, inserting an element into a physically-ordered tuple can be based on the PE id at the point of insertion: all PEs with a lower id are set to "open" and those with equal or higher id are set to "open, send right" in order to shift all subsequent elements to the right. This provides insertion in constant time. The advantages of a 2-d mesh and the bypass feature are discussed in Section 6.2.3.

Switching the edges of the 2-d mesh allows configuring the communication network as a torus or skewed torus, the latter able to emulate a single 1-d interconnect for systolic algorithms without substantial slowdown. Although not included in MISC-1, further workload analysis may indicate a benefit of using one or more edges of the mesh for highly parallel I/O (as in the MPP) specific to each PN, and perhaps tiling multiple array meshes together to a single mesh for inter-PN communication in addition to the global network. The latter could be useful in dynamically combining smaller PNs to function as a single PN for large processes.

Multiple-Response Resolver

The resolver serves three functions: indicating if any of the resolver tags are set, generating the identifier of the first PE in an arbitrary ordering that has the tag set, and resetting the tag in the first PE in the arbitrary ordering. These operations are used to determine if data in any of the PEs meet some criteria, then optionally sequencing through those PEs. This is useful in general as well as associative processing. For example, examining if there are any PEs meeting an if-conditional before executing the conditional segment of code improves speed and processor utilization.

The multiple-response resolver is a modular look-ahead circuit based on Anderson's "P-generator" block and provides several operations [Fos76]. Given a 1-bit tag from each PE as input, the resolver generates the pointer vector (a vector of all 0s except for a single 1 at the first PE with the tag set in the physical ordering). In the nextResponder operation, the pointer vector resets the tag in the first responder and the resolver immediately generates a pointer to the next PE with the tag set. The resolver also generates a 1-bit output indicating if one or more PEs have the tag set (some), and a multi-bit output indicating the address (id) of the first active processor in the physical ordering. Since the tag can be set by any word-ALU result or flag manipulation, the some output can quickly (typically 2 PE cycles) determine if any PEs meet a particular criteria.

The some and nextResponder operations can be used to quickly iterate through the tagged PEs, using the simple construct in Figure 7.4. The operation performed in the iteration is often accumulating a list of ids. To support macros such as Boolean matrix multiplication, the resolver tree is partitioned such that the some test can be performed individually over each column and reported to the array controller (and supervisor) as a multi-bit word, in addition to being performed on the array as a whole.

Due to the tree structure, as shown in Figure 7.8, the multiple response resolution operates in time $O(\lg P)$ where P is the number of PEs. The resolver tree is distributed across the PE chips by including one extra block per chip for use in sub-tree combining, as in Leiserson's H-tree layout [AG89]. A circuit-level analysis including gate, and signal propagation and chip-crossing delays shows that resolution time for a MISC PN is slightly less than 3 PE clock cycles. These operations are likely to be faster than the supervisor can receive the results; in some algorithms the array controller can save the identifiers for use by the supervisor while subsequent array processing proceeds.

MISC-1 does not include reduction tree hardware. Although the transistor count per PE is relatively low reduction trees have significant pincount, layout and interconnect difficulties, and reductions are not a significant portion of the example applications.

```
while (some)
    /* perform operations */
    matchset ← matchset + {responderID}
    ...
    nextResponder
```

Figure 7.4: Responder Iteration

7.2.3 Array Memory

Array memory requirements for the MISC architecture are especially demanding. Application scaling results in extremely large memories per processor. Even though in most cases very few locations are accessed for a given task, crucial portions of the analyzed code require one operand per off-chip memory per processor cycle. With the early analysis tool projecting 60 MHz PE operation, these requirements make the array memory sub-system a crucial element. The MISC solution is two fold. First, off-chip memory is high-performance commodity DRAM. Second, each PE has its own off-chip memory port with full datapath width, local addressing, and write enable.[7] Although this incurs a high cost in high pin-count packaging and increased die area, this approach is cost-justified.

Off-chip memory must be fast to provide one word per PE clock cycle (16 ns). This is best met by EDRAM (extended DRAM incorporating a row cache)[8]. Typical data objects are no larger than the EDRAM row cache, so that even with low locality of object addressing the hit rate per word is high. Effective 20 ns EDRAMs[9] were commercially available several years ago at less than twice to cost of commodity 80 ns DRAMS; future versions five years from now are projected to easily the MISC cycle time given the projected hit rates. In the MISC architecture it is not necessary to detect misses or manage "cache" refill. A miss occurs only when the row address changes; in static-column mode all accesses within a row are hits. As with other DRAMS, row and column addresses are multiplexed, so that the address path is 10 bits for 1 M byte per PE (20 address bits total). The array controller or PE local address unit needs to provide only 10 bits of the address in a single cycle, most often the column address. When the row address changes the miss penalty must be paid, providing an extra PE cycle for transferring the new row address. The array control

---

[7] All other memory-chip control lines are global to the array.

[8] Other current high-speed memory technologies either have excessive cycle time or suffer from lengthy transfer set-up times.

[9] The Ramtrom EDRAM has 15 ns posted write cycle, 15 ns read hit access, and 35 ns read miss access times [Bon92].

unit pipelines off-chip memory addressing, fetching the next data word while the current instruction is executing.

The complications from row caching are easily handled and memory array details are straightforward. MISC-1 is not a load-store architecture. Off-chip memory is accessed as part of an ALU instruction through the memory port which routes memory access through the `A`, `B`, or `R` bus. Global addresses are routed through the PE chip. Although this increases pin count, it places multiplexing the global and local addresses on-chip, avoiding numerous additional chips at the next level of system interconnect. Output of the PE local address unit, which is wider than the external address bus, can be used as the lower or upper bits of the column and row addresses, as shown in Figure 7.5. The address bit routing can be implemented using transmission gates (Figure 7.6). Always sending the global address to the same output address bit positions reduces the number of transmission gates by nearly 25% but the ACU would then have to manage global address bit positions.

The supervisor can access an individual PE's memory in a single cycle by enabling only that PE, directly sending the global address to the PE's memory port, and sending or receiving data through the global address register and data bus. This permits rapid uploading and downloading of large data blocks. Unlike bit-serial CAM architectures, this transfer is word-wide so orthogonal ("turntable" bit-word translation) isn't required. A register load or store operation for off-chip memory can be done in a single cycle through the ALU identity operation.

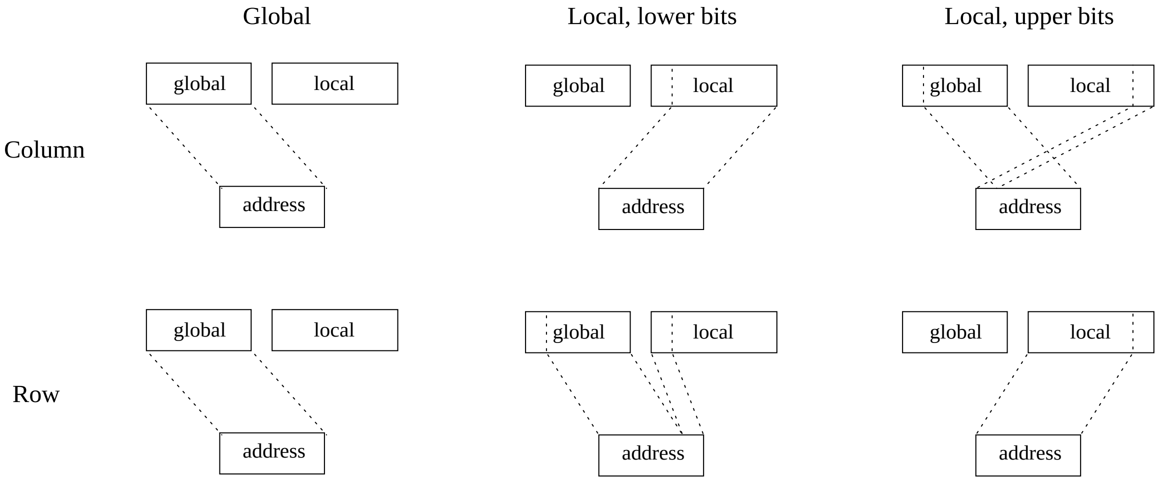


Figure 7.5: Address Multiplexing

The EDRAM row cache is equivalent to having the PE on the DRAM chip, except for the chip-crossing delay; an MCM implementation would reduce this disadvantage. EDRAM gives MISC-1 a 3-level memory hierarchy (DRAM, row cache, registers).

7.3 Proposed Implementation

A myriad of details must be worked out to refine an architectural-level or RTL description into an implementable design with gate, transistor, and chip-layout levels of description.

This dissertation takes the early-analysis approach by projecting cost-performance from high-level descriptions and estimations. This section presents key features of the architecture, particularly the array communication network, in greater detail in order to provide a more accurate early analysis. This section also presents the results of the early analysis, projecting processor and system characteristics and identifying implementation issues. Only those details presented in this chapter have been developed; there is no detailed design of a complete MISC Machine, nor has there been any prototyping.

7.3.1 Array Communication Details

Bypass Mesh

Communication time is a crucial part of many SIMD algorithms, so we present a practical implementation of the bypass-mesh in greater detail to justify flexibility claims and provide a basis to project communication times.

The communication lines are bidirectional, so the simplest circuit would be to use transmission gates as shown in Figure 7.6. The combined series resistance of 32 transistors, in conjunction with the distributed line capacitance, would result in excessive delays even with large transistor sizing. We therefore assume a unidirectional tristate driver, also shown in Figure 7.6. Here driving resistance never exceeds 2 transistors. Each switch thereby consists of a tristate driver for each direction plus a means for sending and receiving on each side of the switch. Figure 7.7 gives a logic-level diagram for one bit of a PE communication unit.

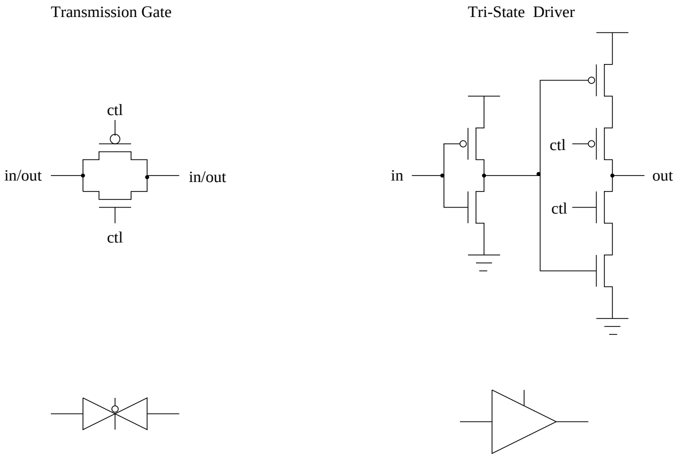


Figure 7.6: Transistor-Level Communication Switches

For timing we take the input inverter stage of the driver as having minimum sized transistors, and the tristate output stage as 8 times minimum transistor size. Logic delay is about 2 gate delays, 0.7 ns in 0.35 μm CMOS. Assuming a 0.7 cm die with 16 PEs, the line length between switches is 0.175 cm. The full switch circuit (Figure 7.7) shows a maximum of 4 inputs and 4 outputs loading a switch output. Under these conditions, Suspense calculates an on-chip propagation delay of 1.6 ns per stage. Chip-crossing delay, due primarily to

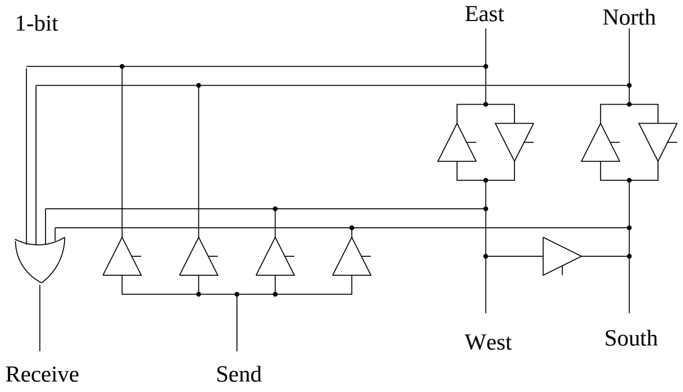


Figure 7.7: 1 Bit of Communication Unit

output driver chain and pad capacitances, is typically 1.5 ns per crossing. Table 7.5 presents delays for communications at various distances. A distance of 16 represents the longest likely communication time because communication at distances between 17 and 32 are better performed by using the torus and sending in the opposite direction. Even at a raw-die PE cycle time of 15 ns this gives any communication distance in only 2 PE cycles, and the average time per step of a reduction over the full array would be 1.2 cycles.

Distance	Chip Crossings	Delay, ns
1	1	3.1
2	1	4.7
4	1	7.9
8	1	14.4
16	3	30.3
32	7	62.0

Table 7.5: Bypass Mesh Communication Times

Multiple Response Resolver

Figure 7.8 is a gate-level diagram of the resolver network for 4 PEs. Unlike most CAM systems, the 1-bit ALUs add flexibility in setting and resetting the resolver tag flag for each PE. One additional cell is placed on each PE chip to complete the tree without any external logic. This increases the number of chip-crossing delays over having a single chip for the upper levels of the tree, but the effect on system performance is negligible. The CAM implementation includes the circuit of Figure 7.8 in 2.0 μm CMOS. Logic delay is 2 gate delays for each of the $\lg P$ levels. For a 0.7 cm chip with 16 PEs (0.35 μm CMOS), Suspense gives total signal propagation delay of 17 ns for the first 4 levels with minimal sized transistors, including signal propagation. For off-chip delay, there are 2 chip crossings per level (one going up the tree, the other going down the tree). For a 1 K PE processor node there are 6 additional levels for 18 ns of additional delay. This gives a total resolver time of 42 ns. Even at the raw-die PE cycle time of 15 ns this is at most 3 PE cycle times.

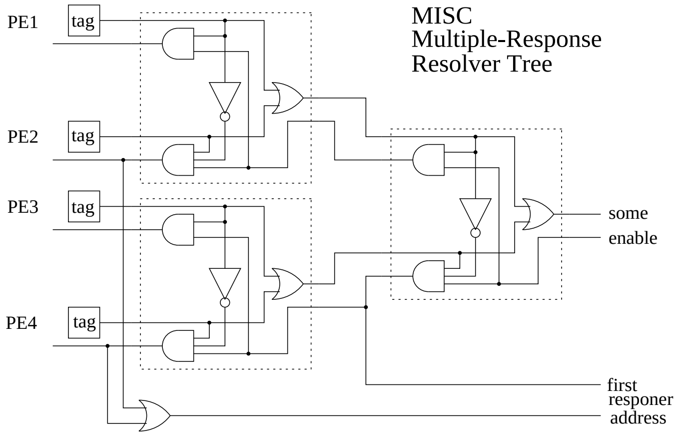


Figure 7.8: Resolver Network

7.3.2 PE Early Analysis

Table 7.6 summarizes estimates of transistor counts for the various sections of the 16-bit MISC-1 PE proposed implementation. Except for the PE control section, these estimates are based on other implementations such as BSYS and PCAM [Hug91, GR90], RTL, gate, or transistor level sketches of features unique to MISC. The additional control is estimated as 5% of the total PE transistor count. The transistor counts in the table are conservative, being the most straightforward and general approach. For example all registers, including the special-purpose functional units are taken as dual-ported (1 read, 1 read/write) with cell areas significantly larger than commercial implementations.

Section	Transistors	
Word ALU, zero-detect, shifter	1310	9%
Register set	5780	42%
Global data and constant registers	390	3%
Communication with registers	1140	8%
Local address unit	910	6%
Special function units	680	5%
Making special-purpose registers multi-ported	1700	12%
1-Bit unit	1370	9%
Control	710	5%
Total	14190	

Table 7.6: Estimated 16-Bit PE Transistor Counts

Like most microprocessors that exclude hardware floating point units, the ALU itself represents only a small portion of the total transistor count; the majority of the transistors are in the register set. Additional features represent only a minor portion of the device budget. Making the special-purpose registers multi-ported adds 12% to the flag's transistor count, and 1/3 of the transistors in the general-purpose register set are due to its being multi-ported. The multiple ports on the register set may not be necessary. Further refinement of the design will no doubt introduce additional transistors, but further optimization is expected to more than compensate.

MISC PE Area

Suspense results give $0.0012\,\mathrm{cm}^2$ for 16-bit MISC PE tile at $0.35\,\mu$m, 57% of this in the multi-ported register set.[10] Projected packaged die delay is slightly under 16 ns. Placing 16 on a die, pins represent over 60% of the PE chip cost for ball-grid array packaging, and over 40% for flip-chip preparation. Active silicon area is $0.0192\,\mathrm{cm}^2$. Due to the large pin count the minimum array pad pitch forces the PE die to $0.47\,\mathrm{cm}^2$, over 25 times larger! This surprising result of the optimizations in Chapter 6 is easily cross checked. PE silicon (at $0.47\,\mathrm{cm}^2$) is 12% of total system cost based on the Suspense cost-metrics. Reducing the PE silicon cost by a factor of 25 would only reduce system cost by about 10%, and accomplishing this would require dramatically reducing pin count. Halving pin count would reduce system cost 15% (PE silicon and packaging) but the mix of example algorithms would run 1.6 times slower.

Using the Available Die Area

Although leaving the additional die area unused would improve yield, it is attractive to use at least some of this area for active circuitry. PE enhancements rejected in Chapter 6 could be included provided critical path delay, floorplanning, and routing are not drawbacks.

The available area could be readily filled by additional memory and functional units including hardware multiply or even a floating point unit, further improving generality and flexibility. The left-right shifter could be replaced with a barrel shifter, the address unit could be made wider, or a hardware multiplier or other numeric support might be added. Alternative instructions could be stored and selected based on the PE state, providing further local control and improving PE utilization. PE microinstructions are on the order of a hundred bits, so that just a few of alternates would require an area approaching that of the entire register set. Even so, a single stored instruction would permit emulating a heterogeneous systolic array. Additional general-purpose flags and registers may prove useful depending on future application profiling.

As discussed in Chapter 6, on-chip RAM might permit narrower off-chip memory data paths or allow several PEs to share an off-chip memory for relatively small applications. The available area could support 2 K Bytes of additional mult-ported registers per PE (but at a 15% PE cycle time penalty), 2 K bytes of CAM (4 K bytes if dynamic), or 32 K bytes of DRAM.

An on-chip sub-array controller could improve flexibility and PE utilization. From the available area, Suspense calculations show that as over 1 million transistors could be added to the PE chip (64 thousand per PE).

---

[10] The PE is input as two tiles to allow for the larger register-cell area as compared to commercial microprocessor register layouts.

7.3.3 System Early Analysis

The system early analysis considers three PN configurations: 256 PEs with 64 K bytes each on-chip, 256 PEs with 512 K bytes each off-chip, and 1024 PEs with 512 K bytes each off-chip. Smaller mid-term application goals (256 x 64 K SDMs, 128 K node SNs, 64 K 16-node CGs) require only 256 PEs with 64 K bytes each or less; they are nonetheless an order of magnitude larger than those in current research projects. Long-term application goals (1 K x 1 M SDMs, 1-2 million node SNs, 1 million 32-node graph CG databases) require roughly 1 K PEs with 256 K to 1 M byte of memory each. System analysis includes the supervisors, array memory, array communication, and the processing elements.

For supervisors, the present analysis assumes a basic RISC chip, without a special-function AI coprocessor. The supervisor is taken as a 3 million transistor microprocessor and memories are based on commercial product literature supplemented by Suspense calculations. The communications interface is taken as 1/4 of a four-in one-out ATM packet switching component [B^+94] which includes virtual channels and a total throughput of 70 M bytes/s, giving 105,000 transistors. Lower-latency alternatives would be used in any MISC implementation; the ATM is used only as a guide to likely transistor counts. Like the supervisor and memories, it is assumed to be a commercial product. The array controller is estimated as 450,000 transistors, implemented using semi-custom microcontroller megacells (plus the writable control store). Only 4,000 transistors (1%) are logic, 40,000 (9%) are stacks and registers, and the remaining 90% are the writable control store.

The MISC architecture prohibits using large high-density DRAM. Industry targets 64 M bit DRAMS for 1995, sufficient for 8 to 32 PEs, but the commodity memory will include only a single address port and narrow data paths. It is reasonable to expect that current 4 M bit EDRAMS will be offered with 16-bit data paths, and that λ shrinks (from 0.6 μm down to 0.35 μm) will be attractive, providing a fast, high-yield low-cost memory. This provides a small area 512 K byte memory for each PE.[11] Suspense projects static column cycle times of 9 ns and read-miss times of 20 ns, allowing sufficient timing margins for a 17 ns array clock cycle. Memory die stacks are not feasible for MISC due to the multiple address and data ports.

The PE and communication switching is as presented in Section 7.3.2. Array wrap-around switching for a 32 x 32 array with 16-bit data paths requires at most 21,000 transistors[12] and 1600 pins for each of the four edges. This can be trivially partitioned onto multiple chips to distribute the circuitry along the edge length.

Early analysis compares single-chip packaging on printed circuit board (PCB), multi-chip module (MCM), and hybrid MCM and PCB implementation of the various PN configurations. Table 7.7 summarizes the Suspense analysis. All packages use ball-grid array packages. The high pin count packaging is available now, and should be cost-competetive within 5 years. In 1994, IBM was offering 700 pin ball-grid array packages and announced a 1,225 pin package; flip-chip bare die mounting for MCMs could provide even higher"pi" counts at reduced cost per connection.

---

[11] 8 M bit and 2 M bit sizes, providing 1 M byte or 256 K byte per PE are not common in industry.

[12] Using switches similar to that of the on-PE bypass mesh switches detailed in the next section, but with additional inverter chains for extremely large output transistors to drive the long wrap-around lines quickly.

In the event these packaging technologies don't become common, wafer-scale integration could be used. Early analysis indicates that a small (512 PE) MISC processor node with memory could be placed on a single 5 inch wafer, slightly increasing speed and reducing size, but at a projected factor of 3 cost penalty. The cost of custom high pincount packaging may be less than that of going to wafer-scale integration. The number of PEs per chip could also be reduced. At 8 PEs per chip, existing commodity pin-grid array packages could be used.

PEs:	256					1024	
Memory:	64 K byte on chip		512 K byte off chip			512 K byte off chip	
Substrate:	PCB	MCM	PCB	MCM	MCM & PCB	PCB	MCM & PCB
Size	11 x 11 in	1.7 x 1.7 in	12 x 12 in	3.2 x 3.2 in	6 x 6 in	20 x 20 in	9 x 9 in
Speed	55 MHz	64 MHz	55 MHz	69 MHz	60 MHz	43 MHz	60 MHz
Power	73 W	12 W	131 W	22 W	61 W	382 W	123 W
Cost/PE	14.9	15.2	27.0	62.9	28.0	25.5	26.6
S-T	0.27	0.24	0.49	0.91	0.47	0.59	0.44

Table 7.7: MISC Processor Node Early Analysis

Adding 64 K bytes of on-chip memory per PE using a MOSIS process 2-transistor DRAM ($190\,\lambda^2$ cell) doubles chip area and gives a cost greater than that with 512 K bytes off-chip commodity memory so that a commercial memory process[13] which does not increase die area is assumed. Without off-chip memory ports 400 pins are required as opposed to 1180 for off-chip memory. Even accounting for reduced production volume (tens of thousands versus tens of millions) the on-chip memory configuration has cost-performance twice as good as the off-chip configurations, *but at 1/8 the available memory.* Another advantage is that an entire PN can fit on a cost-effective sized MCM greatly reducing area and power and somewhat improving speed. For high-density application that do not require the additional memory an 8-PN system could be placed in the 100 cubic inch "prototyping stack" [LAS89].[14]

Adding large off-chip memory does not significantly affect the size or other characteristics of a 256 PE processor node using printed circuit board technology. It does, however, force a larger MCM substrate which is not cost-effective due to low MCM substrate yield. An MCM implementation might be useful in high-density, high-performance applications; MCM increases speed 25% and an 8-PN system would fit in a a modified form of the prototyping stack.

For a 1024-PE processor node, PCB area becomes excessive. Array speed is reduced by over 20%, possibly favoring 4 smaller PNs even with the accompanying communication overhead. Boards larger than about 1 square foot present manufacturing and reliability difficulties so that a 1 K PN would likely be implemented as 2 or 4 smaller boards, further reducing speed. Placing a PN on a single MCM would require an unusually large and expensive MCM substrate. A combination of MCM and PCB technology solves these

---

[13] Assuming a $60\,\lambda^2$ cell; current high-density DRAMS feature $20\,\lambda^2$ cells.

[14] This prototyping stack has been successfully implemented on other projects, and can dissipate 250 W with air cooling. The volume includes cooling; power supply would add another 80 cubic inches.

problems; 64 PEs and their memories fit on a cost-effective 1.6 inch square MCM. Each MCM package would require 935 pins for communication links, control, power, and ground. Substrate area dominates over pin pitch, giving a package size of about 1.75 inches square. This reduces a 1 K PN to a 9 inch PCB and cuts the area of a 256 PE node to 1/4 while significantly reducing power and slightly improving performance. With board stacking an 8 PN system would occupy roughly 300 cubic inches excluding power and cooling. Off-chip memory accounts for over 40% of total system cost (including interconnect, power, and cooling). Although it is less than 1/4 of PCB area, it accounts for nearly 2/3 of the area of MCM implementations.

7.3.4 Implementation Difficulties

Chip interconnection is the most significant difficulty expected in implementing MISC. Although the assumed pin counts and substrate routing density used are available today and should be cost-effective within a few years, switching many thousands of off-chip lines (the memory ports and communication links) at the projected 50–60 MHz is an important problem. Line lengths do not present unusual propagation delay difficulties; the problem is the switching currents when so many lines change logic states simultaneously. The current draw can induce switching transients and cause supply voltage sags, both of which reduce noise margin and can introduce errors. Switching can also induce transients in adjacent traces. It will be necessary to develop some combination of techniques such as line critical dampening, low-voltage signaling, and low ΔI drivers in addition to careful signal routing and extensive supply voltage decoupling. The PCB-MCM approach reduces these problems by reducing the length of MCM lines and the number and length of PCB lines. Additionally, MCM signals have much lower switching energy. These issues further favor an MCM approach. A proof-of-concept prototype could minimize these issues (and the cost of currently advanced packaging) by having fewer PEs per PN, operating at reduced frequency, reducing communication path widths, and sharing several off-chip memories per PE, but not without significant performance reductions.

Locating memory chips in the small-capacity multi-bit data path configuration may be difficult, but the configuration appears attractive for low-chip count, high-speed general applications. The MCM community appears to have developed the infrastructure for obtaining tested bare-die and performing MCM assembly tests. Nevertheless, reliability, fault tolerance, and testability are important issues that are not addressed in this study.

7.3.5 Summary

The early analysis indicates that a MISC PN has several cost-effective implementations and can operate at 40–70 MHz offering 20–35 thirty-two bit MIPS per PE. The total cost metric is on the order of $1 per MIPS, excluding development amortization, distribution, and support. A 1 K PN should have a peak performance of some 30,000 MIPs and hundreds to thousands of MFLOPS. Eight such PNs could be packaged in a few hundred cubic inches. There is the potential for terraops systems. At the opposite extreme, all implementations are small enough so that systems with one to a few PNs are suitable for embedded systems and as workstation accelerators.

7.4 Machine-Level Programming

The MISC PE incorporates numerous functional units with extensive processing flexibility so that programming will be more complex than for most SIMD arrays. Completely developing an instruction set is outside the goals of this of the dissertation. This section instead shows principles and several example instruction formats to establish MISC hardware level programming. In particular, machine-level codes are given for the key operations in the example application profiles. In addition to illustrating machine-level programming, these examples are the basis of cycle counts used in the simulations described in Appendix C.

7.4.1 Instruction Formats

The array requires microinstructions for the word-wide ALU, the 1-bit unit, communication, the special-purpose units, and the address unit. The ALU and 1-bit unit require three-address instructions, two for sources (A and B buses) and one for the destination (R bus), so that instructions must have a significant number of bits to specify operation of even a single unit. Control lines would require some 128 bits without special treatment.

- Word ALU control (16 lines)
- Select 2 word ALU operands and result (registers or main memory; 10 bits each for 1 M byte off-chip memory)
- 1-bit ALU control (4 lines)
- Select one of 32 flags for 2 1-bit ALU operands and result
- Enable one of 8 or more special function units
- Configure the address adder unit
  - Select one of 4 options for the first input
  - Select one of 2 options for the second input
  - Configure carry-in, enable or disable adder
  - Select address output, one of 3 options
- Configure 3 bi-directional communication switches plus enable/disable sends
- Other functions
  - Configure word ALU carry-in and carry-out
  - Configure shifter shift-in and shift-out, shift-left or shift-right
  - Select conditionals (If =0, for example)
  - Enable/disable resolver tree
  - *etc.*

It is important to minimize instruction size to avoid problems with wide buses as well as to reduce the size of the microsequencer in the ACU, while at the same time maintaining maximum flexibility. MISC accomplishes this in 3 ways. First, "vertical" microcode is used in which certain fields are encoded, to be decoded at the chip level. For example, the universal logic blocks in the ALU require a total of 16 bits but since fewer than 64 operations are useful, the ALU operand can be reduced to 6 bits. Second, and unique to MISC, many of the functional units are handled as special-purpose registers, controlled as a part of register addressing. Third, also unique to MISC, control registers are pre-loaded to set up particular operations that participate in subsequent instructions. This reduces microinstructions to at most 64 bits.

Figure 7.9 shows several example instruction formats.[15] The first format is for loading the communication, adder unit, and flag control registers. The second, the word-ALU format, is most common. The third format is used for conditional statements, combining an ALU operation to create a test condition with flag (1-bit unit) operations; the ALU result itself is not stored. Variations are created by deleting unused fields and adding fields for other functional units. Word ALU instructions not requiring immediate (global) data could specify a concurrent communication, 1-bit unit, or adder unit operation. Other instruction formats specify non-concurrent communication, 1-bit unit, and adder unit operations in 32- or 16-bit instructions. Additional 64-bit instruction formats can express various combinations of operations.

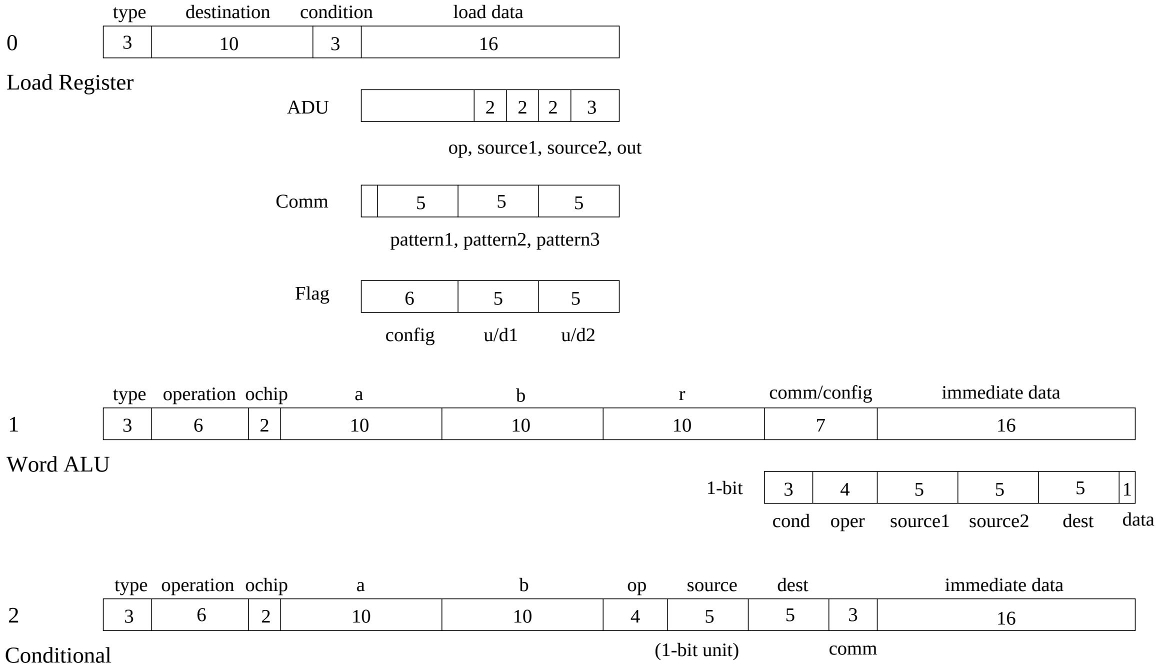


Figure 7.9: Primary Instruction Formats

7.4.2 Example Machine-Level Codes of Key Macros

Hamming Distance

Here we compute the Hamming distance between a single long (multi-word) Boolean vector stored in the supervisor and vectors stored in each PE, exactly as in the SDM

---

[15] Although multiple instructions violates the “RISC philosophy” the reason against them are relatively unimportant to MISC. Instruction decoding occurs in a single array control unit (ACU), not at each PE. The ACU can incorporate complex decoding logic and pipelining, and operate at a clock rate higher than the PEs.

algorithm. This is a good example of the intricacies possible in PE programming. The result of a logical operation in the ALU is sent to the ones counter special-function unit (ONES), and its results are accumulated by the address unit (ADU) functioning as an auxiliary adder. In high-level SEGUL notation, the operation for each word-wide iteration is $C \leftarrow C + |A \oplus B|$. In the machine-level coding of Figure 7.10, the memory address register (MAR) is C, the PE memory (**global_addr**) is A, and global immediate data (**global_data**) is B. Gdat in the first operand fields indicates that the operand comes from the immediate data field. Operation is pipelined. At iteration i, the ACU is prefetching data for step $i+1$ from off-chip memory (9 ns cycle time in static column mode), the ALU is performing exclusive-or for step i (15 ns), the ones counter is determining the number of bits in ALU result $i-1$ (6 ns), and the address adder is accumulating the ones count from step $i-1$ (8 ns). The MISC PE is thereby able to accumulate Hamming distance in a single cycle for each word width.

```
/* configure the address adder */
rload(always) adu ← (mar ← ADD(onesCounter, mar))
/* for each word */
      alu onesCounter ← XOR(b, gdat, global_address, global_data)
```

Figure 7.10: Hamming Distance Microcode

Matching Functions

Here we present three matching functions. The first is a simple one-word exact match, typical of SN node marking. The second complicates the operation by performing a masked match on multi-word comparands with multiple values stored in each PE. The third is the subsumption comparison on lattice codes used in CGs.

The SEGUL notation for the exact match is $F \leftarrow |A \oplus B| = 0$, where F is a one-bit flag (F1) and A and B are again immediate global data and a value stored in PE memory. The operation is a single line of microcode performing an exclusive-or and moving the ALU zero-detect (ZD) into the flag, as follows.

```
cond IF XOR(b, gdat, global_address, global_data) THEN flag1 ← zd
```

The multi-word masked-match requires loading the mask register, initializing the match result flag to 1, then iterating through each word comparison. At each iteration $F \leftarrow F \wedge ZD$; any mismatch clears the match flag. To avoid re-loading the mask register for each VP-slot, a separate flag is used for each slot. The microcode is given in Figure 7.11.

Figure 7.12 presents annotated microcode for a subsumption test on lattice codes. A is immediate global data and B is in register R1 and R2; the temporary value is stored in R0. Execution time is $3\lceil m/b \rceil$.

Interval subsumption codes require the same number of cycles, so it is unnecessary to distinguish between the two in evaluating and simulating algorithms. The operations $(x \geq a) \wedge (y \leq b)$ determine if the interval $[x, y]$ is contained within $[a, b]$; replacing the and with an or tests for overlap.

```
/* initialize the match flags*/
rload(always) gflags ← 0hFFFF
/* for each word-position $w$ */
    /* load the mask register */
    rload(always) mask ← $\text{mask}_w$
    /* for each data object $i$ */
        cond IF XOR(b, gdat, global_addr, $\text{data}_i$)
            THEN $\text{flag}_w$ ← AND($\text{flag}_w$,zd)
```

Figure 7.11: Multi-Word Length Masked-Mask Over VP Slots

```
/****flag ← $A \preceq B$*** */
/* $C \leftarrow A_1 \odot B_1$ */
alu r0 ← XNOR(x, gdata, r1, global_data)
/* $C \leftarrow C \wedge \overline{B_2}$ */
alu r0 ← ANDnotB(x, r0, r2)
/* flag ←| $C \vee A_2$ | */
cond IF OR(x, gdata, r0, global_data)
    THEN f1 ← zd
```

Figure 7.12: Subsumption Test Microcode

Bit-Packed Boolean Matrix Multiplication

Figure 7.13 presents the microcode for multiplying two Boolean matrices, $A \times B$, and permits concurrent operation on multiple graphs in a single array. The code is for the case where no matrix dimension is greater than the number of bits in the PE (*i.e.* $n \leq b$ and $n \leq \sqrt{P}$; if these condition is not met, communication between columns of the array are necessary). The code does allow for multiple graphs in each column of the array by forming a coterie for each graph. A PE stores one row each of the two matrices.

The transpose portion iterates through the rows of B, broadcasting each row to the other PEs in the coteries. Each PE selects the appropriate bit from the broadcast row and transfers it to the target bit position. In the multiplication code, the coteries perform a circular shift across PEs. The PEs compute the Boolean dot-product by and-ing the row and transposed column together and testing for an all-zero result. Total time is $7 + 6(r + c)$ cycles where r is the number of rows and c is the number of columns. This is $O(n)$ for $n \times n$ matrices using n^2/b processors per graph. When processing multiple graphs, time is determined by the largest graph.

The algorithm has several advantages. With one row of each matrix per PE, the "summing" portion of the Boolean vector product is replaced by a zero test on a single data word, avoiding inter-PE communication. This is particularly important as as each multiplication requires n iterations of this operation. The row assignment also allows quickly testing if any row of a matrix is all zeros or if the matrix has changed since the last iteration

```
/**** Transpose ****/
rload(topflag) comctl ← (sendS open open)
rload(midflag) comctl ← (broadcastNS xmitN xmitS)
rload(botflag) comctl ← (sendN open open)
/* for i = 1 to nrowsB */
    alu comput ← (x, Arow)
    cond IF EQL(x, peposition, i) THEN comset(1)
    cond IF GREATERTHAN(x, peposition, i) THEN comset(2)
    cond IF LESSTHAN(x, peposition, i) THEN comsetDo(3)
    alu AND(x, memmask, comin); flag$_i$ ← NOT(zd)
/**** Multiply ****/
rload(always) flagctl ← (modulocount)
rload(NOT edgeflag) comctl ← (meshS, xmitN)
rload(edgeflag) comctl ← (meshS meshN)
alu flagu/d1 ← (x, peposition)
alu comin ← flags
/* for i = 1 to ncolsB */
    alu AND(x, Arow, comin); flag$_{flagu/d1}$ ← NOT(zd)
    alu comout ← (x, comin)
    cond IF EQL(b, botflag, 0) THEN comsetDo(1)
    cond IF EQL(b, botflag, 1) THEN comsetDo(2)
```

Figure 7.13: Boolean Matrix Multiplication

in a single comparison and multiple response resolution. Further, with all rows of the matrices of a given graph allocated to a column of the array, the Boolean matrix transpose requires only a regular, efficient broadcast of each row of the matrix to the other processors in the subarray, which then select the appropriate bits for their destination matrix row.

General Routing

A simple mesh-routing algorithm developed for MISC turns out to be the fastest, despite not using coteries or the bypass mesh, as a result of having few instructions per routing cycle. The algorithm is a typical greedy mesh routing algorithm, routing in two phases and using toroidal edge connections. The first phase sends messages horizontally until they reach their destination column, where they are queued awaiting the next phase. The second phase routes messages vertically to their final destination. Figure 7.14 gives the microcode. Communication can be overlapped with ALU operations; negligible time is required for the communication unit to access the communication data registers as they have direct connections, and transmission time overlaps the ALU operation.

On MISC, each routing cycle of the algorithm requires 12 instructions per routing cycle for messages consisting of one destination word and two data words, as is the case for the SN simulation. As with Corbett *et al.*'s coterie-routing algorithm [HCWS94], worst case time is $\Theta(P)$ routing cycles; a queue could end up with as many as $\sqrt{P}$ messages, giving Phase

```
/**** Phase 1 ****/
rload(always) comctl ← (meshE, meshS, x)
rload(always) mar ← queuestart
rload(always) maskreg ← thecolmask
rload(always) adu ← (mar ← PLUS(mar, 1))
/* for i = 1 to arraycolumns */
    cond IF XOR(x, comout, PEid) THEN queueflag ← zd
        alu @mar ← comout IF queueflag marOP
        /* for j = 1 to messagewords */
            alu @mar ← reg$_j$ IF queueflag marOP
    /* else */
        alu temp ← commin, IF NOTqueueflag marNOP, commout
        alu commout ← reg$_{j=1}$, IF NOTqueueflag marNOP, commout
        /* for j= 2 to messagesqueued */
            alu comout ← reg$_j$, IF NOTqueueflag marNOP, commout
            alu reg$_j$ ← commin, IF NOTqueueflag marNOP, nocomm
        alu comput ← temp, IF NOTqueueflag marNOP, nocomm
/**** Phase 2 ****/
/* end of queue in mar */
alu questart ← temp2
cond IF NOTEQL(mar, queustart) THEN aflag ← 1
rload(allways) adu ← (mar ← MINUS(mar, 1))
/* for j = 1 to messagewords */
    alu reg$_j$ ← @mar, always marOP
alu comout ← @mar
alu mar ← temp
alu temp2 ← mar
/* same as phase 1, but mesh south ........ */
alu temp ← mar
```

Figure 7.14: MISC Mesh-Routing Algorithm

$2\sqrt{P}\sqrt{P}$ cycles. A probabilistic analysis of expected-case time where all destinations are equally likely shows much better behavior. For simulation, we want a time that is exceeded in only 1% of random routings; instances exceeding this time add at most a few percent to total time spent in routing. Given a 1 K PE array, we then want the queue size that will be exceeded less than $.99^{1/1024} = 99.999\%$ of the time in any PE. Using the Poisson distribution gives queue length of 6 for 100% of the PEs sending, 5 for 50%, 4 for 25%, and 3 for 10%. Adding 1 for Phase 1 gives the total number of times $\sqrt{P}$ routing cycles must be applied. Given the Poisson distribution and the minimum and maximum number of routing cycles gives $\lceil 5(1-e^{-2s})+2\rceil\sqrt{P}$ expected-case routing cycles where s is the portion of PEs sending. Expected-case general routing time is thereby $O(\sqrt{P})$.

7.5 Summary

This chapter has described the MISC architecture at several levels, focusing on the associative array. At the highest level, we discuss key components of the MSIMD global architecture. Rather than presenting a single design, this chapter presents several viable options for the global communication network and processor node supervisor, pending further analysis. Existing and emerging technologies should permit scaling MISC to a hundred, perhaps a thousand processor nodes (and a hundred thousand to a million processing elements), dependent on application parallelism.

The associative arrays are described primarily at a register-transfer level. The processing elements feature a general-purpose ALU and register set core with no cycle penalty for an off-chip operand. Special functional units support single-bit manipulation, distance metric calculation, full local addressing, improved local control, and CAM emulation. The array communication network features wide communication links and the ability to bypass intermediary processing elements, improving both nearby and long-distance communication performance, and permitting the formation of sub-arrays. The array memory features a high-bandwidth connection to a large off-chip memory for each processing element along with object caching, providing nearly the performance of on-chip memory at much lower cost.

Crucial features of the associative array are developed to the gate or even transistor level, with several features presented in this chapter, to improve the accuracy of early analysis of system cost-performance. Early analysis of the array processor chips shows that die area is greatly increased due to the required pin-count, but that the high pin count required by the MISC architecture is cost justifiable. The available area can be used to provide further tailoring to further knowledge processing as well as to make MISC more general-purpose. Early analysis also projects MISC system characteristics, showing that compact high-performance MISC implementations are both feasible and cost-effective, that advanced packaging is desirable, and that advanced cooling technology is not likely to be needed.

This chapter also discusses machine-level programming, including example microinstruction formats and microcodes for the key SEGUL macro operations of the example algorithms. In addition to establishing machine-level programmability, the microcodes in conjunction with the system early analysis, form the basis of the MISC cost-performance analysis of the next chapter.

8. Evaluation of Results

This chapter evaluates the end results of the dissertation: the MISC architecture presented in Chapter 7 and the MISC associative processing model presented in Chapter 4. The research is evaluated according to four criteria.

Feasibility Can the SIMD architectural framework support massively parallel processing of the complex, dynamic data (particularly relational structures) typical of knowledge processing? Can an associative processing model express knowledge processing data and control structures with their multiple levels of parallelism? Can this expression be readily converted to a computer program?

Generality Is the MISC architecture and programming model limited to the specific algorithms used in their development, or are they more widely useful? Can the architecture be extended to accommodate new features and applications?

Performance Does the MISC architecture significantly improve execution speed over prior architectures?

Efficiency Does MISC make efficient use of the processors and processor features as indicated by improved cost-performance?

Section 8.1 presents a qualitative evaluation of how the MISC processing model and architecture meet the needs of parallel knowledge processing in general and the example algorithms in particular. The following Section 8.2 discusses the generality of MISC, listing additional knowledge processing and related algorithms as suggested by architectures and processing models similar to MISC's. Section 8.3 then differentiates MISC from from the other architectures and processing models. Quantitative evaluation begins in Section 8.4. This section summarizes simulation results, confirming the manual instruction profiles and in turn the architectural choices, and presents projected execution times demonstrating high performance. Section 8.5 compares simulated performance of MISC and the MasPar MP-2, highlighting the benefits of particular MISC features, and closes with a comparison of cost-performance.

8.1 Supporting Massively Parallel AI

This section first reviews how the SEGUL pseudocode and processing model serves in expressing parallel data and control structures typical of knowledge processing and related algorithm, and its utility as a programming tool. This is followed by discussion of how SIMD architecture can provide massively parallel support for the associative processing model and how the MISC architecture provides improved support for knowledge processing. In so doing, this section establishes MISC as a feasible means of achieving the research goals.

8.1.1 The Associative Processing Model

The associative processing model and its pseudocode notation SEGUL are based on a single data structure, the nested collection. The model emphasizes matching operations to select elements of collections for further processing. In addition to succinctly defining data structures, the collection notation brings out their formal mathematical structure; a relation is an ordered collection and a graph is a set of relations, for example. Thus the model incorporates the most important and fundamental formalisms for knowledge representation: sets, order, and relations.

Expressing Knowledge Processing Algorithms

The examples throughout this dissertation show how the basic operations of matching and distance measure, set operations, and parallel operations (both symbolic and numeric) on selected elements in the MISC associative processing model as notated in SEGUL can be used to build high-level massively parallel knowledge processing algorithms. The algorithms include pattern matching, marker passing connectionist networks, subgraph isomorphism, partial order maintenance, and inheritance codes. These can in turn support the "pillars" of AI: retrieval from large knowledge bases, knowledge representation, learning, and heuristic search and inference. We now consider the central features of the MISC associative programming model and its notation, SEGUL.

Nested Collections as the Base Data Structure Nested collections are quite expressive. As an indication of their expressiveness, a nested collection is equivalent to the lists in LISP, but with two extensions: each collection (list) can be specified as ordered or unordered, and duplicates can be optionally prohibited. The MISC associative model adds direct element access and manipulation by content, which can avoid list-chasing in many circumstances. These extensions are not formally necessary, but are clearly helpful in expressing KP algorithms more succinctly and in speeding execution. SEGUL also features set operations and a default mapping to parallel processing of a collection's elements, and provides additional abstraction by encouraging operations on nested collections as though they were a single object. It does not, however, incorporate LISP's other key features such as lambda expressions and applications. Similarly, Prolog's predicates can be expressed as tuples, but SEGUL does not intrinsically include Prolog's resolution mechanisms.

Complex Dynamic Data Objects Chapters 4 and 5 demonstrate SEGUL's ability to express knowledge processing's compound and dynamic data structures. The data structures are explicitly included in the notation as nested collections with element insertion, deletion, membership, selection, and comparison operations. As a basic example, Section 4.2.4 shows how stack and queue ADTs can be readily expressed in SEGUL and Section 4.3.1 describes a linked list with constant-time garbage collection. The pseudocodes for the example further illustrate the variety of data structures easily represented in terms of SEGUL's primitives as well as SEGUL's ability to express complete algorithms comprising the major portion of full applications. There should be no difficulty in using the SEGUL pseudocode notation to describe full applications incorporating a wide variety of data structures and algorithms.

As to the more specific goals of this dissertation, relational structures such as graphs, partial orders, and classification hierarchies are expressed in the associative processing model's notation. Chapter 5 presents the SEGUL pseudocodes for the data structures as well as the algorithms for operations such as associative retrieval (SDM), graph edge traversal (SN), subgraph isomorphism and node subsumption. Additional graph representations are summarized in Chapter 4. Retrieval from large knowledge bases are also well supported. The example algorithms illustrate search and advanced retrieval paradigms (associative memory, semantic network, multi-level indexed search) expressed in SEGUL and "executing" on the 2MISC simulator.

Data-Dependent Control The associative processing model does not rely entirely on match predicates but also includes the standard declarative programming constructs (if, for, etc.) for supporting complex data-driven control structures. Blelloch's dissertation shows how conditionals can be efficiently applied to nested collections [Ble90], and Willett's parallelization of Ullmann's algorithm demonstrates that recursive backtracking search algorithms can be run with nested parallelism on a SIMD architecture [WWR91]. SEGUL also includes a control structure for iterating through collections.

Multiple Levels of Parallelism The SEGUL notation incorporates both implicit and explicit expression of multiple levels and grains of parallelism. Fine and medium grains of parallelism are implicit in treating the nested collections as a single object. A non-nested collection corresponds to the basic vector or data-parallel model of parallel programming and Blelloch has shown that he multiple levels of parallelism available at each level of nesting can be efficiently implemented on vector, MIMD, and SIMD architectures [BS90].[1] The hierarchical retrieval pseudocode in Section 5.3.4 demonstrates the implicit expression of even more complex nested parallelism, the simultaneous execution of subgraph isomorphism comparison of a set of graphs. The superiority of nested parallelism with pruning over exhaustive data parallelism is analyzed by the author [HLR93], and summarized in Section 5.4

Coarse-grain parallelism is explicit in the SEGUL notation, with independent parallel processes communicating by messages. A message is itself a nested collection, providing data communication between processes. Matching on certain elements of the message are used to activate and synchronize the processes.

The prior examples do not include inter-PN communication. Figure 8.1 is a simple example incorporating multiple PNs in implementing a "metanetwork" of multiple SDMs. The example is for Kanerva's k-fold method for associatively matching temporal sequences of patterns. Predicting the final symbol in the sequentially presented string ABC from BBD can not be done by a single SDM; B would always map to C (or if trained on both ABC and BBD, would return a corrupted mixture of C, B, and D). In the k-fold metanetwork, time-delayed copies of the input are fed to multiple SDMs that then vote on the output; k SDMs are needed to capture dependencies k steps back in time.

The examples presented in this dissertation show that programming multiple levels of parallelism over complex objects is feasible with minimal programmer burden, at least at the scale of the example application pseudocodes and simulator codes. In the examples in this dissertation, the SEGUL pseudocode notation provided a succinct high-level machine and programming language independent description of all the data structures and algorithms. SEGUL provides implicit expression of lower levels of parallelism, including nested

---

[1] The MISC associative model extends Blelloch's ordered collections to unordered collections with an emphasizes on matching, as detailed in Section 4.3.

```
/*** K-Fold Recall ***/
int k ← 4, m ← 1024, offset, i, x, done
nonunique tdpatterns ← [code(m):k]
code(m) result, cumresult

when receive['KfoldIn', thisPN, code(m) input]:
    tdpatterns(offset) ← input
    for i = 1 to k
        send['SDMRead', theSDM+offset, tdpatterns(offset)]
    when receive['SDMresult', ?x, result]
        cumresult ← cumresult + result
        done ← done +1
    when done = k
        cumresult ← cumresult > 0
        offset ← (offset+1) mod k
        send['KfoldOut',thisPN,cumresult]
;
```

Figure 8.1: Multiple SDMs (k-fold)

parallelism, and the notation is readily understandable, based primarily on standard set notation. SEGUL provides explicit expression of coarse-grain parallelism in its tuple-matching notation for inter-process and inter-PN message passing.

SEGUL Programming

Developing a high-level programming language is not part of this dissertation. The specific notation for the associative processing model, SEGUL, is designed specifically for pseudocoding algorithms used in the development and evaluation the MISC architecture. Nevertheless it has been a highly useful tool for developing algorithm input for the 2MISC simulator.

Comparing the SEGUL pseudocodes with the C code used in the 2MISC simulations gives a good indication of the promise of SEGUL as the basis for a programming language or program specification tool. The direct correspondence between the SEGUL notation and the simulator code enables manual translation, and automatic compilation may well be possible. Data declarations and type definitions correspond directly to SEGUL's described in Section 5.1 with additional data allocation function calls required to set up simulation parameters. A more specific discussion of mapping the example algorithms from SEGUL pseudocode to the 2MISC simulator and MISC microcode is included in Appendix C.

Difficulties for Machine-Level Programming In several cases, however, there is not an immediate and mechanical translation to efficient MISC machine-level code. For example, the most natural SEGUL expression of the SDM algorithm is not the fastest parallelization for most SIMD machines. It is presently unclear to what degree SEGUL might be automatically compiled (possibly incorporating hand coded extrinsics for difficult representations and macro operations) and to what degree it must remain an algorithm development and communication tool.

The example algorithms revealed several difficulties. The refinement algorithm shows a weakness in SEGUL's element communication construct. Data movement for distributing the graph nodes to PEs in forming the M matrix requires close coordination with local communication features of the hardware. It is questionable whether a high-level SEGUL description can briefly describe the necessary data movement or if standard routines would be sufficiently flexible. Low-level data movement and local communication may prove difficult. The paralation-based partial-permute construct may not be up to this task, requiring either too elaborate (and possibly machine-dependent) descriptions or insufficient specification for efficient implementation. As another example, this dissertation includes several different graph representations: edge list (SN), node list (SN alternate), paralation segmented vector, and partitioned adjacency matrix (refinement). The choice between them depends on what operations are to be performed. SEGUL's expressiveness and flexibility may turn out to be implementation liabilities.

The MISC architecture and programming model include numerous features for dealing with the special requirements and difficulties of knowledge processing algorithms. This results in an extremely rich set of functional units and options, exceptional in SIMD microarchitectures. This is not an unmixed blessing. It is not clear if a compiler can make effective use of all the possible combinations, or automatically translate from the SEGUL pseudocode to efficient microcode in the majority of instances. These issues are not within the scope of this dissertation. The simulations described in Section C do show that relatively straightforward programming, making only very limited and specific concurrent use of the special features, achieves dramatic speedup over architectures without the extensions.

Further research is needed to address how these problems might be overcome by some combination of programmer guidelines, collections of standard high- and low-level routines and data structures (including coterie communication constructs), and compiler optimizations based on usage analysis.

Future Potential Overall, experience implementing algorithms described in SEGUL on the 2MISC simulator indicates good potential for SEGUL to serve as a programming tool, and perhaps as the basis of a high-level programming language with nested parallelism but certain difficulties must be overcome. The most natural description of an algorithm may not have the highest performance, potentially burdening the programmer with including knowledge of the hardware in choosing an algorithm and its description. Performance loss on the examples ranges from under 20% for different ways of storing and distributing the KB graphs in refinement to a factor of 3 for straight- versus transposed-storage for SDM. SEGUL is not a complete fully defined language at this time, although future work may lead in these directions. What is most relevant to this dissertation is the associative processing model's expressiveness for parallel knowledge processing data structures and algorithms.

8.1.2 The MISC Architecture

The associative processing model can be executed on generic SIMD architectures; the MISC architecture includes numerous features to significantly improve hardware support for the example applications, the associative model, and the broader realm of knowledge processing. We first discuss how the proposed MISC system architecture supports the special characteristics of knowledge processing, followed by a somewhat more detailed discussion of the associative array and its processing elements

MSIMD

The analysis of Chapter 5 and the simulations below indicate that the example algorithms will scale to permit large knowledge bases of some 1,000,000 associative memory patterns of a thousand bits each, 1,000,000-node semantic networks or 1,000,000 CGs to reside on a single MISC processor node (PN) with query response times of 0.5 ms (SDM) to 12.5 ms (graph knowledge base). Multiple PNs permit knowledge processing systems with "metanetworks" of many SDMs, multiple knowledge representations, multiple knowledge domains, multiple agents, and extremely large knowledge bases.

Partitioning MISC into a MIMD network of multiple processor nodes each with an independent associative array goes a long way in supporting flexible control structures, improving concurrency and PE utilization and removing much of the burden from the SIMD associative arrays. The supervisor's array controller, and potentially sub-array controllers, further improves MISC's ability to handle the complex control structures and multiple levels of parallelism often encountered in KP codes.

Global Communication The literature suggests that global communications will be bursty with data-dependent "random" patterns. MISC minimizes this difficulty through the multiple PNs. Each PN exhibits a high degree of clustering (*i.e.* a high ratio of internal computation to external communication between PNs) as calculated in Chapter 5, dramatically reducing the total communication placed on the global network. Further, hierarchical communication topologies such as fat-trees respond well to non-uniform communication patterns and make the best use of available communication resources to minimize latency and contention [Lei85]. Elaborate topologies may not be necessary, as simulation results show that simple interconnections such as a shared bus may permit MISC systems as many as a hundred PNs.

Supervisors The high-performance supervisors execute scalar operations and code segments whose control structure or other characteristics prohibit using the associative array effectively, allowing each processor node to execute complete routines and applications.

The co-processor hardware-stack manager proposed for the supervisor supports recursion and subroutine intensive codes in coarse-grain parallelism through rapid task switching, and minimizes the impact of global communication delays and lengthy fine-grain parallel associative array operations. The co-processor may also support caching strategies specific to AI codes, emphasizing structural locality [WCHC89]. It may not prove practical to implement these features, however, as discussed in Section 7.1.3.

The Associative Arrays

Memory Memory scaling and referencing patterns show that an unusually large memory per PE is appropriate in comparison to current commercial SIMD machines in order to meet the needs needs of advanced knowledge retrieval methods. The proposed implementation meets this difficulty by using high pin-count packaging to provide single-cycle access to the full off-chip memory simultaneously by each processor. The off-chip memory takes advantage of beneficial referencing characteristics to support the extremely fast cycle times required. Although little locality is present in which data objects are referenced, considerable processing is usually performed on an object once it is accessed. The row-cache incorporated in enhanced DRAM chips is of the size of a typical data object, and functions as a separate object-cache for each array PE. This caching also functions well when iterating over virtual processor memory slots provided memory allocation anticipates this referencing pattern.

Communication The array local communication network must also support the characteristics of knowledge processing codes. Algorithm analysis shows that local communication is typically in regular patterns. The MISC architecture effectively supports this with the array's nearest-neighbor mesh, with wide data paths minimizing communication time which could otherwise become a bottleneck. The mesh bypass feature of the SIMD array improves performance for longer-distance communications.

Similar reconfigurable-mesh networks are used in a variety of systems, for purposes such as fault-tolerance, machine vision, signal processing, sorting, and general routing [LS92, JS94, HCWS94]. Here we develop and evaluate the network for multiple Boolean matrix multiplications and paralation operations with nested parallelism. The mesh bypass is beneficial for Boolean matrix operations, supporting broadcast and array partitioning. The network also provides flexible long-distance communication patterns important to dynamic collection maintenance and enabling log-time reduction and scan operations (including hardware support for segmented reductions and scans) as described in Chapter 4.

Many algorithms, such as marker passing, require data-driven "random" communication patterns (*i.e.* general routing). This is also well supported by the wide-link mesh network, and performance might be further improved by exploiting bypass-mesh configurations [HCWS94].

Processing Elements The MISC processing element microarchitecture includes numerous features that support compound, dynamic, and/or linked data structures. The local address unit provides considerable flexibility, including local data-dependency in selecting data objects and portions of objects. The indirect addressing may prove especially important in methods such as Martinez' "descriptors" for accessing large, variable-sized associative data structures [Mar90]. The local address unit can double as an adder pipelined to the PE's word-wide ALU for computing distance metrics. The flag unit provides single-bit manipulations concurrent to the ALU and other units, important to emulating CAM functionality and manipulating Boolean vectors and matrices in algorithms such as subgraph isomorphism on relational structures, including partial orders. It's also helpful in processing markers in semantic networks and related connectionist models. The flag unit is especially important in supporting constant-time set operations.

Other special-purpose units include the one's counter for Hamming distance and Boolean correlation and dot-product operations and the mask register for sub-word manipulations in the ALU. Several of these features, such as the resolver network, the register set, and the flag unit combine to provide efficient, high-speed emulation of CAM, but without the restrictions to fixed position, fixed length data.

MISC PE features additionally provide limited local data-driven control, extending the SIMD paradigm somewhat. The configuration registers (the mask register, flag control register, communication control register, memory constant register, and address control register) allow each PE to be configured differently, either by data local to the PE or under global control. Some of the benefits for the local address unit were just discussed. The communication control register allows significant local data-dependent control of communication patterns. The stacks and counters available through the flag control register, as well as the ability of the address unit to perform the same functions, allows further local control. As an example scenario, a data structure could include its length. With the length loaded into a counter, each PE could mask itself off once all elements of the structure are processed, independent of global control and even the PE's ALU. The resolver network can determine if one or more PEs are active in just 2–3 cycles, so that execution of unnecessary conditional code segments can be avoided. The resolver also supports the supervisor in rapidly iterating through accessing active PEs. Each of these uses improve PE utilization for various program control structures.

8.2 Generality of the MISC Architecture and Programming Model

MISC development focused on three example algorithms emphasizing knowledge retrieval and inference. This section establishes MISC's generality as the ability to efficiently execute a much larger list of algorithms through arguments of subsuming other architectures and processing models. In particular, Blelloch's analysis of the paralation model is the basis for the claim that MISC is asymptotically nearly optimal with respect to lower bounds on parallel processing for a variety of algorithms.

8.2.1 Architectural Equivalences

The MISC architecture is a superset of many massively parallel architectures, combining features of contemporary SIMD machines, prior associative processors, and elements of MIMD machines.

The core of the MISC PE is a fully general-purpose word-wide ALU and register set, enhanced with additional functional units based on KP algorithm characteristics. The flag unit is itself effectively a bit-serial SIMD PE, and the word-wide ALU and local addressing include most of the features of more recent SIMD machines. The enhanced mesh local communication network provides similar performance to other network topologies. Local communication also includes a multiple-response resolver. With these features, MISC subsumes the operations, if not the exact hardware, of most SIMD as well as many special-purpose architectures, such as bit-cell CAM, with at most a small slowdown and more often a significant speedup.

Although the proposed MISC-1 implementation does not include hardware multiply/divide or a floating point unit, these operations can be performed via software with only a small slowdown over most contemporary SIMD machines, or additional hardware could be added. The MISC PE does not include hardware explicitly for multiplication, division, or floating-point operations. These may be required in applications such as probabilistic reasoning and weight learning. The flag unit and shift-register can support the ALU in performing integer multiplication in b cycles where b is the length of the multiplier and ALU, in bits. Although not as fast as an array multiplier, MISC does perform integer multiplication in almost the same number of cycles as the MP-2, and should be able to execute floating point operations within a factor of 2 or 3 times that of the MP-2. The implementation analysis in Chapter 6 indicates that there is sufficient space available on the PE die to add hardware multiply or perhaps even a floating-point unit for each PE without significantly affecting system cost.

The overall MISC architecture combines MIMD and SIMD processing. From the perspective of MIMD processing, the associative arrays are co-processors for the supervisor. From the SIMD perspective, the network of supervisors improves problem partitioning, resulting in improved control flexibility and increased PE utilization.

We consider equivalences for two particular machines, the Connection Machine (widely used in massively parallel AI projects) and the LUCAS associative processor, as well as with systolic processing in order to establish that algorithms implemented on these and similar machines can be rapidly executed by MISC. Similar arguments hold for the MasPar MP-2, supplementing the more specific analysis later in this chapter.

CM-2

MISC's flag unit, with its 1-bit universal logic block, 1-bit registers and access to main memory, is nearly identical to the Connection Machine processing elements. The exception is that the ULB has 2 rather than 3 inputs. Operations requiring the third input bit are most often operations like addition, where the third bit is carry; the word-wide ALU can perform these operations much faster than a bit-serial processor. Otherwise, the flag unit can perform arbitrary 3-input Boolean operations in 2 cycles. Both the flag unit and word-wide ALU can access two inputs and store the result in the same cycle; the Connection Machine (and MasPar) PE requires a total of 3 cycles so that MISC requires fewer cycles even for 3-input functions.

MISC's restriction to 1024 PE arrays is not a limitation. Although for applications requiring more PEs communication between PNs might become a bottleneck, this is unlikely and not particularly relevant in establishing that MISC can perform algorithms already established for the Connection Machine. A single MISC PN has projected performance greater than the a Connection Machine with tens of thousands of PEs. The proposed MISC implementation features 8 times the clock speed, 16 times the data path width, and 1/3 the clock cycles per primitive operation in comparison to the CM-2. There is little doubt that a MISC PN can execute the same algorithms (except those depending specifically on a hypercube topology) as the older CM-2, at considerably greater speed.

Researchers and industry have developed a large number of algorithms and applications for the CM-2 and MasPar machines. Table 8.2 presents a small sampling taken from a single conference [AAA93]. This prior work is important in establishing state-space backtracking search and load balancing on SIMD architectures for AI applications. Backtracking search methods include state-space (tree) search and IDA*.

LUCAS

Similar arguments hold for LUCAS as for the CM-2. LUCAS is an associative processor whose PEs are also composed of bit-serial 3-input ULB with several 1-bit flags [FKS85]. Simple analysis shows that MISC's word-wide bypass-mesh network outperforms LUCAS's bit-serial shuffle-exchange network, and MISC includes a multiple-response resolver with the same capabilities as LUCAS's.

Fernstrom *et al.* include execution of numerous algorithms on LUCAS as part of its evaluation [FKS85]. Although LUCAS is presented as an associative array processor the algorithms include areas such as relational database operations, matrix operations, graph algorithms, and image processing. Table 8.2 lists the algorithms. The image processing algorithms include linear, mean, median, and Laplacian filtering, and the edge-detection algorithms include cross-difference, border finding, and object perimeter.

The relational database (RDB) operations are suggestive of MISC's performance on databases and regularly structured relations. Table 8.1 shows asymptotic performance, based on Fernstrom *et al.*'s analysis. For the table, N is the total number of relations, N_1 and N_2 are the number of the first and second argument's relations, N_{R2} is the number of result relations, N_{A1} is the number of different values the first attribute takes, and r is the ratio of the number of tuples of the second relation to the result. The projection figures are for the case where none of the attributes of the argument relation included in the result is a key. Otherwise, LUCAS time for projection is $O(1)$ as the result candidate tuples do not have to be checked for redundancies.

Operation	Sequential	SIMD	Speedup
Selection	$O(N)$	$O(1)$	$O(N)$
Intersection	$O(N \lg N)$	$O(N_1)$	$O(\frac{N}{N_1} \lg N)$
Difference	$O(N \lg N)$	$O(N_1)$	$O(\frac{N}{N_1} \lg N)$
Semi-Join	$O(N \lg N)$	$O(N_2)$	$O(\frac{N}{N_2} \lg N)$
Join	$O(N_1 \lg N_1)$	$O(N_{A2}(1 - r)N_1 N_2)$	$O(\frac{\lg N_1}{N_{A2}(1-r)N_2})$
Projection	$O(N \lg N_{R2})$	$O(N_{R2})$	$O(\frac{N}{N_{R2}} \lg N_{R2})$
Division	$O(N \lg N)$	$O(N)$	$O(\lg N)$
Union	$O(N \lg \frac{N}{2})$	$O(N_2 p)$	$O(\frac{N}{N_2 p} \lg \frac{N}{2})$
Product	$O(N_1 * N_2)$	$O(N_1 * N_2)$	$O(1)$

Table 8.1: Relational Database Parallelization (after [FKS85])

Other than the non-indexed select, the RDB asymptotic speedups in Table 8.1 are disappointing, typically not much better than a factor of lg N. In most cases this is the result of checking for duplicates serializing over the size of the result. Without this, operations such as projection, union, and intersection could be done in constant time. In some cases, a query optimizer could postpone removing duplicates until the final result. Alternate data representations and algorithms may asymptotically improve performance on RDB applications. This is particularly promising as the algorithms assumed were developed for LUCAS and do not take advantage of the special features of MISC.

Systolic Processors

It is possible to program SIMD machines in a systolic processing style; the local communication network serves as the systolic links. Consider BSYS as a specific example. The BSYS processing element is essentially the same ALU as MISC (but is 8 rather than 16 bits) plus a register set and bank of 1-bit flags [Hug91], so that MISC is for the most part a superset of the architecture. Its shared register architecture combines memory access and systolic communication, so that MISC requires additional cycles for communication. On the other hand, MISC incorporates features such as a 2-d mesh topology, large memory, and other features which extend the range of executable algorithms, but at a significant increase in cost over 1-d systolic arrays.

Similarity to BSYS indicates yet another set of algorithms that can be executed on MISC with only a small constant factor slowdown over the systolic hardware. These are also listed in Table 8.2. In addition to operation as a 1-d array like BSYS, we include additional algorithms made possible by MISC's 2-d mesh. In addition to several graph algorithms, the systolic algorithms feature a variety of sequence comparison algorithms. Algorithms such as convolution, data compression, FFT, DFT, and filtering as well as the image processing algorithms (executable in other programming paradigms) are important in input processing for robotic and other intelligent systems.

Category	LUCAS	CM-2, MP-1	Systolic Array
AI Related		backtracking search	
		branch and bound	
		frames and relations	
		semantic networks	
		neural network	
		genetic learning	
		induction	
Graph	shortest path		
			all-pairs shortest path
	MST		MST
			shortest cycle
			transitive closure
			bipartite matching
			bridges
Computational Geometry			convex hull
			nearest neighbor
			polygon intersection
Relational Database	select		select
	union, intersection		union, intersection
	difference		
	product, division		division
	join, semi-join		join, equi-join
	projection		
			duplicate removal
Image Processing	edge detection		
	filtering		
	gap filling		
	component labeling		object labeling
	shrink and expand		erosion and dilation
	tracking		
			stereo matching
Other			sort
			search
		load balancing	
			matrix inversion
			matrix transposition
	matrix multiply		matrix multiply
			inner product
	1-d, 2-d FFT		
			1-d, 2-d convolution
			deconvolution
			FIR, IIR filter
			sequence comparison
			priority queue
			data compression
			channel assignment
			alternate transcription

Table 8.2: Other Algorithms Efficiently Implementable on MISC (based on hardware models)

8.2.2 Processing Model Equivalences

MISC's processing model and its SEGUL notation draws from other programming models and languages which can be implemented on SIMD machines, indicating additional algorithms executable on the MISC architecture.

Gamma

Sections 2.4.3 and 4.1.2 discuss Gamma's "react-act" programming and the MISC associative model's match-act style, showing that the Gamma constructs can be readily translated into SEGUL. By this argument, MISC can execute the programming examples given in the Gamma literature [BM93, BM90, BCM88], listed in Table 8.3. SEGUL supplements this construct with many others, including global information and control structures which remove the Gamma restriction that prohibit its use for crucial algorithms such as backtracking.

SETL

As discussed in Section 2.4.6, SETL includes set operations and SEGUL incorporates the same basic data objects. SETL includes additional functions and input/output that can be readily added to SEGUL, and SEGUL includes additional functions not included in SETL. MISC can thereby efficiently execute the algorithms listed for SETL in Table 8.3. Other example algorithms include binary search and text processing. The SETL literature includes the classic introductory AI example "Towers of Hanoi," demonstrating backtracking search within the framework of a set-based language [SDDS86]. Topological sort is important in its use in Levinson's Method III multi-level indexed sort and in "untangling" hierarchies for encoding.

Paralation

Sections 4.3 and 8.2.2 demonstrate MISC's ability to efficiently incorporate Blelloch and Sabot's paralation model [BS90]. Their model has been implemented on the CM-2. MISC includes hardware (the bypass-mesh network) which speeds segmented scans, not present in the CM-2. Chapter 4 also showed various additions to the paralation model, including extension from ordered to both ordered and unordered collections, associative matching, and comparison between collections.

Blelloch's algorithms are listed in Table 8.3. The graph and computational geometry algorithms are particularly important due to their potential in supporting relational structures and complex searches. The sorting algorithms include quick sort, radix sort, and merge sort, and both quick hull and merge hull algorithms are given for finding the convex hull. Low-level graph routines also include merging vertices, deleting an edge or vertex, finding the maximum or sum of neighbors, and distributing excess. Low-level routines specific to trees include operations over descendents or ancestors, computing the depth of a vertex, size of subtree, passing a marker up or down the tree, splitting trees, and deleting vertices. These may prove useful in hierarchy and partial order maintenance. Other low-level routines include scans and extractions over multidimensional grids. The algorithms for building k-d trees and for closest-pairs are particularly relevant for nearest-neighbor classification. Blelloch also demonstrates load-balancing on a SIMD architecture using the paralation primitives.

Category	Gamma	SETL	Paralation
AI Related		backtracking search	A* search k-d tree building
Graph	shortest path MST connectivity	Euler path	* MST * bi-connected component ** max-flow ** maximum independent set
Computational Geometry	convex hull		* convex hull * closest pair
Relational Database			
Image Processing	edge detection		
Other	sort prime factorization longest increasing sequence	sort topological sort Huffman coding binary search	sort * matrix multiply ** vector-matrix multiply ** matrix inversion linear system solver simplex step

Table 8.3: Other Algorithms Efficiently Implementable on MISC (based on processing models)

MISC Near-Optimality

The ability to implement the paralation model serves to show that MISC is nearly time-optimal in the sense of the formal parallel-RAM (PRAM) models of parallel computing in which multiple processors share a common memory. Blelloch's analysis treats scan operations as constant-time, when in fact they are dependent on communication time. Making this correction, his results show that most scan operations execute within the communication-time factor of the lower bound of VLSI and circuit theory models for time, and with asymptotically less area [Ble90]. He also compares several important algorithms to the EREW (exclusive-read exclusive-write) and CRCW (concurrent-read concurrent write) PRAM models of concurrent computation. His analyses are based on 1 PE per edge for graph algorithms, 1 PE per key in sorting and merging, 1 PE per point in computational geometry algorithms, and 1 PE per matrix element in matrix multiplication and inversion.[2] In most cases, the paralation model speeds are at the EREW lower bounds, and within the communication-time factor of the faster (and more difficult to actually implement) CRCW lower bounds. These algorithms are noted in Table 8.3. The MISC associative array's 2-d mesh topology has communication time $O(\sqrt{N})$, asymptotically worse than the lower bound

---

[2] Blelloch shows instances where reducing the number of PEs from this full data parallelism does not increase processing time, namely halving merge, list ranking, and tree contraction.

of $O(\lg P)$. The partitioning to multiple PNs bounds the asymptotic difference, however, and analysis shows performance including constant factors to be similar to or better than $O(\lg P)$ networks such as Boolean hypercubes. Other machines using partionable meshes (and PEs simpler than MISC's) also approach the CRCW lower bound for several image processing algorithms [LS92].

Summary

To summarize, Tables 8.2 and 8.3 list the broad range of algorithms that could be demonstrated on the MISC architecture. Particularly noteworthy, those marked with an asterisk ('*') in the Paralation column are asymptotically within a factor of $O(\sqrt{N}/\lg N)$ of the lower-bound EREW PRAM model on MISC, and within an $O(\sqrt{N})$ factor of the tighter CRCW lower-bound. Those marked with '**' are also within a $O(\sqrt{N}/\lg N)$ factor of the CRCW lower-bound. With asymptotic factors this close, constant factors are clearly important and MISC's architectural and implementation choices result in a "practically optimal" design for key graph and computational geometry algorithms.

8.3 MISC Special Characteristics

Having discussed MISC's similarities with prior programming models and machine architectures, we now consider the differences which lead to MISC's improved performance and programmability.

8.3.1 Comparison to Other Processing Models

The MISC associative model draws from generic associative processing and data-parallel operations as well as from several specific languages: Paralation, SETL, Gamma, and Linda. In addition to being a unique combination of features, SEGUL incorporates support for complex dynamic data structures not present in any of its predecessors. Paralation is very influential in its emphasis on heterogeneous nested collections, but lacks the unordered collections and emphasis on matching and other associative operations featured in the MISC model. The MISC model also includes additional underlying mechanisms for data structure representation and management. SETL is important in its emphasis on set formalism and maps. It is, however, a serial language. The MISC model features parallel representations for sets and tuples, explicit communication between data objects, and additional low- and high-level matching operations. The Gamma model emphasizes match-act cycles, but lacks the mechanisms for global information and control necessary for crucial operations such as backtracking search. Linda features the same tuple-matching mechanism for parallel process communication and management as SEGUL, but lacks data parallel constructs, nested collections, and explicit communication. The MISC associative model extends these prior languages and methodologies by combining the nested collection data parallelism with inter-process (and inter-PN) communication by messages, a richer set of collection types, and mechanisms for dynamic collections and collection comparisons.

8.3.2 Comparison to Other Machines

CAM and Associative Processors MISC's innovations and unique combination of features are important differences from other SIMD architectures and make substantial improvements over prior associative processors. CAM systems, including PCAM, are highly specialized and as shown have poor S-T for general applications. These systems, and associative processors such as LUCAS and ASP, lack a full-function word-wide ALU. Combining a word-wide ALU with features typical of associative processors, MISC is a much more cost-effective and flexible system. MISC extends CAM operations from fixed format attribute lists to dynamic, compound structures, yet is faster than most CAM hardware.

SIMD Machines MISC includes many features of recent SIMD machines, with several important extensions. Early SIMD machines such as the CM-1, CM-2, and MPP are bit-serial and also lack MISC's wide communication links. A more recent implementation of the MPP, Blitzen, has local addressing that is limited to oring the lower address bits for accessing its on-chip memory. MISC features a full address calculation unit and its on-chip registers are nearly the size of Blitzen's on-chip memory. The MP-1 and MP-2 include full local addressing and a 32-bit wide ALU but multiplex access to off chip memory over 4-bit datapaths. The multiplexing and narrow data path introduce a major bottleneck for knowledge processing algorithms not present in MISC. The proposed MISC-1 does not include the floating point and multiply/divide support present in the MP-2, but this could be added with little effect on system cost. The SIMD processors, including the MasPars, omit many of the features important in associative processing incorporated in the MISC architecture. These include the concurrent flag unit, ALU mask, multiple-response resolver, ones counter, and pipelined general use capability for the address adder. MISC's unique flag unit, address unit, and communication configuration registers provide local control well beyond the MasPars', MPP's and Connection Machines' instruction masking and Blitzen's negated logic operation selection. MISC's registers are multi-ported, reducing most operations from 3 to 1 cycles. The SIMD systems discussed also lack MISC's asynchronous mesh-bypass feature which allows non-neighbor communications to be performed in time nearly independent of distance, typically faster than the Connection Machines' serial hypercube or the MasPars' serial multi-stage router.

MIMD Machines The MISC architecture is substantially different from that of current MIMD machines, offering benefits for knowledge-processing codes. The CM-5's global architecture is very similar to MISC's, but MISC is optimized for associative operations, replacing the CM-5 processor nodes' numeric vector processors with associative arrays. Similarly, the Intel Paragon and Cray T3D incorporate wide, high-bandwidth mesh networks but use standard microprocessors and as a result are most suitable for scientific and engineering applications. Unlike experimental MSIMD systems such as the partitionable 1 K PE array PASM, MISC promises a higher number of PEs with each array fully independent. As a result of application characteristics and high-performance PEs, algorithms exhibit a high degree of clustering on MISC and each routine resides on a single array. There is typically 10^5 bytes processed within an array for each 1 byte communicated outside the array.

This greatly alters communication requirements offering a simpler global communication network, simpler application partitioning, and potentially simpler programming.

AI Machines In comparison to the parallel AI machines surveyed, MISC supports a broader range of operations well beyond basic matching to support production systems. In lieu of MISC's custom high-performance PE and mesh network, DADO uses standard low-performance microprocessors for matching and sort operations. DADO's tree-structured communication network limits communication patterns and in turn applications, and reduces PE utilization. NON-VON incorporates both wide-ALU and bit-serial sections but these are simpler with reduced capabilities and are optimized for database-like operations in support of production systems. It does incorporate a combination of MIMD and SIMD processors but suffers similar restrictions from its tree-structured communication network. FAIM is similar to MISC at its top-level, incorporating multiple microprocessors each with co-processors for fast context switching, address management, and communications. Instead of a flexible associative array, each microprocessor is associated with a small CAM which performs only simple masked compares in support of logic programming. Other processors are special-purpose. SNAP is specific to semantic networks. Its current implementation uses standard digital signal processing processors, and future plans include multiple small CAMs for simple matching operations. IXM2 is also specifically for semantic network processing. H.A.S.P. incorporates a custom 16-bit ALU and local addressing, but is tailored specifically for database retrieval. WASP, originally intended for image and signal processing, is now being developed to support expert systems, LISP, and PROLOG. Unlike these machines, the MISC architecture has the power and flexibility required for a variety of evolving AI paradigms as well as other types of applications.

8.4 MISC Simulation Results

This section examines results from the 2MISC simulator for each of the three example algorithms: SDM, a minimal semantic network, and subgraph isomorphism (the most time-consuming portion of the graph knowledge base). Simulation instruction profiles confirm the manual analysis and architectural decisions and projected execution times demonstrate high-performance on the example applications.

The simulations use randomly-generated synthetic data sets. Although this does not directly tie the simulations to real-world examples, it avoids restricting results to one or a very few specific instances. Runs are parameterized, so that data sets of widely varying sizes can be generated. Parameterization also allows the investigation of the effects of various dataset characteristics on execution times and instruction profiles. MISC application target sizes are one to two orders of magnitude larger than those currently run on workstations. Due to memory and speed limitations of the workstation running 2MISC, it is necessary to perform runs of different sizes and interpolate to the target application size. Appendix C details each simulation.

8.4.1 Sparse Distributed Memory

Section 5.1 presents the SEGUL pseudocode for the basic SDM model of associative memory and Appendix C.2 presents the 2MISC simulator code for the straight- and transposed-storage implementations. The synthetic input does not pose any problems in MISC evaluation as the random input vectors correspond to uncorrelated patterns.[3] Correlated inputs reduce the number of patterns that can be stored accurately (or the recall accuracy of a given number of stored patterns) but do not affect execution time or instruction profiles.

The simulations brought three unexpected results. First, supervisor operations represent a substantial part of execution time even with a 500 MIPS supervisor. For transposed storage, supervisor time is 31% of the total at the target 1 K × 256 K SDM size and appears to be independent of N (the number of storage locations), and a (the number of activated locations). Much of this time could be overlapped with array operations and might be reduced through supervisor-coprocessor tailoring. The following analysis does not include these potential improvements. Second, some 10% of the array time is independent of SDM size and the number of activated locations. Third, the speed advantages of transposed over straight storage is somewhat greater than the factor of 2 from the manual profile analysis. Extrapolations of total execution time are given in Table 8.4. In conjunction with the direct simulation of smaller SDMs, it is clear that transposed storage is faster than straight storage on MISC.

Input Bits	Physical Locations	Straight Storage	Transposed Storage	Ratio
1 K	256 K	0.030	0.0099	3.0
256	256 K	0.013	0.0051	2.6
1 K	32 K	0.025	0.0045	5.5
256	32 K	0.008	0.0038	2.0

Table 8.4: SDM Total Execution Time (seconds, 10 reads and 10 writes)

Otherwise, simulation results are quite similar to those from the preliminary manual profiles. Table 8.5 presents a breakdown of array time by type of operation at the target SDM size for transposed storage. Global communication, calculated manually from the simulation in Appendix C.2 is 0.5 MByte/s. As expected from the analysis in Section 5.1, total array time is dominated by Hamming distance calculation, and is thereby $O(Nm)$ both empirically and analytically. Other components are constant time or linear in a. Addition time in the table is only for the summation phase (other additions are part of Hamming distance), giving only a tiny fraction of that listed in the manual profile. Combining with miscellaneous other operations, the percentage of non-Hamming distance operations is quite similar. Each operation averages 19,400 memory references per PE. Of these, 8% are to on-chip registers and the remaining is to off-chip memory. By analysis, 99% of the off-chip memory references are to the first 32 K bytes per PE, the active location address storage used in the exhaustive Hamming distance calculations. The remaining 1% ($O(N^{-2/3})$) is in

---

[3] Most analyses of SDM assume uncorrelated inputs.

the sparsely accessed 256 K bytes of weight storage. For an SDM with a million locations, 128 K bytes are needed to capture the bulk of the memory references.

Operation	Total	Array
Global Communication	2%	
Supervisor	30%	
Array	68%	100%
Hamming dist.		88%
addition		2%
other		10%

Table 8.5: Breakdown of SDM Simulated Execution Time

Given the 0.2 ms extrapolated execution time of the target 1 K x 256 K SDM, a single MISC PN executes 5000 iterations per second. At 2.56×10^8 weights, this gives 1.3×10^6 million connection (weight) updates per second (MCS), a common measure of neural network speeds.[4] With 1 M byte per PE, a single PN could execute 4 similarly sized SDMS for 5×10^6 MCS. Simulations show that reducing input/output to 256 bits doubles the number of MCS for a given number of PEs and memory allocation, and MISC machines with 16 PNs should be readily achievable. The MISC architecture thereby scales well into terra connections per second for SDM. Comparison to other SDM implementations are given in Section 8.5.1.

8.4.2 Semantic Networks

Whereas the SDM example emphasizes special functional units for Hamming distance, a semantic network emphases general routing (random, long-distance communication) on the associative array. General routing is also an important part of processing alternative graph and poset representations, and parallel processing in general. For example, Section 5.2 shows that parallelizations of algorithms using edge traversal (marker or message passing, such as Dijkstra's shortest-path algorithm) are faster using general routing whenever there is more than a few dozen active edges.

In an actual SN application the network topology is determined by the structure of the knowledge, initial node marking may be based on individual attribute fields or markers from prior operations, and marker propagation may be restricted to certain link types or by a node's marker state (and may include mathematical and logical operations). The 2MISC synthetic data and queries approximate these effects, as described in Appendix C.3.

Table 8.6 lists the instruction breakdowns from the simulation. The SN example spends roughly one fourth of its execution time in the supervisor. Unlike SDM, overlapping supervisor and array operations is less likely to be possible, or at least more difficult. Supervisor time is empirically $O(N)$ whereas total array time is $O(N^{3/5}/\lg N)$ so that scalability with respect to the supervisor is better than for SDM. Matching and "other" operations are also $O(N)$ time, and marker propagation time is $O(M)$ where M is the average number

---

[4] It should be noted that these figures are for Boolean inputs and outputs and integer valued "synaptic" weights.

of markers active each cycle.[5] As with SDM, there is a significant amount of overhead nearly independent of network size, which also shows in DeMara's analysis. Of the over 2 billion memory references, only 1% are to the on-chip registers. Global communication is calculated manually based on the simulations in Appendix C.3 as 0.7 MByte/s for adding nodes to the concept-type hierarchy and 0.2 MByte/s for SN queries.

The time spent in matching is an order of magnitude less that that of the manual profile (5% versus 67%), the result of four factors. First, many of the simple matches are replaced by subsumption tests that are tallied as "other" rather than matching. Second, the number of logical operation for updating markers and managing propagation ("other" in the table) was underestimated. Third, the matching operations in general routing are tallied as part of local communication time. Fourth, the manual analysis used the lower bound for communication on a mesh, $2\sqrt{P}$. Routing cycles per marker propagation cycle are $2\sqrt{P}$ only initially when only a few percent of the processors are sending; at the average number of markers (288 for the target application) it is $4\sqrt{P}$.

Operation	**Create SN**		**Queries**		**Combined**	
	Total	Array	Total	Array	Total	Array
Global Communication	10%		1%		2%	
Supervisor	61%		19%		22%	
Array	29%		80%		76%	
matching		98%		3%		5%
mesh comm.		0%		59%		57%
other		2%		38%		37%

Table 8.6: Simulated SN Execution Time Profiles

The extrapolated simulations show total elapsed time of 7.0 s to build the 256 K nodes with 1 M edges, 27 μs per SN node. Total time for the 32 K simple queries is 83.3 s (2.5 ms per query). This gives 390 queries per second, averaging 3540 marker propagation cycles per second. Section 8.5.2 compares performance to other machines.

8.4.3 Graph Knowledge Base

The graph knowledge base (KB) is a complex application, including forming and maintaining the concept-type hierarchy and lattice codes, Method III poset maintenance and traversal, and SI (subgraph isomorphism). Here we simulate the refinement portion of Ullmann's subgraph isomorphism algorithm as detailed in Appendix C.4, then add projections based on manual profiling for the remainder of the retrieval algorithm. Refinement represents the bulk of execution time in subgraph isomorphism, which in turn represents the bulk of time in Method III retrieval.

---

[5] Analytically, M is $O(N^{3/5}/\lg N)$. To be complete, another term should be added for routing time per cycle, $O(\sqrt{P})$ in this analysis; except for exceptionally large SNs (over a million nodes) P is fixed at 1,024 PEs.

Instruction profiles for refinement are presented in Table 8.7. Results are adjusted for the typical number of calls to refinement [WWR91] to amortize the time for retrieving the knowledge base graphs and form the M match matrices. Boolean matrix multiply clearly dominates runtime, even with multiple bits per processor. Of the matrix multiply time, roughly 60% is in the matrix transposition and 40% in the multiply itself. Local communication dominates both. Unlike SDM and SN, the supervisor does not perform any portion of refinement but merely passes the query graph and high-level instructions to the array controller. Over 99% of memory references are to on-chip registers. Global communication, calculated manually from the simulations in Appendix C.4, is 1 MByte/s even with a thousand generalizations and specializations returned (it is only 0.05 MByte/s returning a single graph identifier).

Operation	Total	Array
Global Communication	nil	
Supervisor	1%	
Array	99%	
distribute A, form M		4%
Boolean matrix multiply		91%
other		5%

Table 8.7: Refinement Execution Profile

Adjusted MISC execution time ranges from 0.25 ms to 0.63 ms per group of graphs given the graph sizes and characteristics described in Section 5, page 98. Amortized over the number of processed simultaneously graphs, and adding 14% for backtracking portion of subgraph isomorphism and 15% for the multi-level indexed search code and concept-type hierarchy maintenance (based on manual profiles), gives 5 to 51 μs per graph. With $\lg N$ groups of graphs to be processed and Method III executed on the supervisor simultaneously with the SI tests on the array, a single MISC PN can process from 74 queries per second for a 1 M graph KB (25 to 30 node graphs) to 249 queries for a 32 K graph KB (8 to 12 node graphs). Each query identifies generalizations and specializations of the query graph. Examining the simulations, for graphs of fewer than 32 nodes, average execution time iterating until all graphs are finished is linear in n. With $O(n)$ graphs processed simultaneously, time is $O(n^2)$ as expected. Section 8.5.3 presents limited comparison to other machines.

8.5 Comparison to Other Machines

This section compares MISC to other machines, primarily the MasPar MP-2. The MP-2, the highest-performance SIMD machine in widespread use, has a PE complexity and datapath width similar to MISC but is optimized for numeric operations. Additionally, most massively parallel AI research has been conducted on the older CM-2 due to (prior) commercial availability; the second most prevalent platform is the MasPar predecessor MP-1. Performance comparison is based on 2MISC simulations of both machines. The simulations also form the basis of further evaluation of MISC's architectural features and trade-offs. This section concludes with a cost-performance comparison. The latter shows that MISC makes efficient use of its processors and special features.

8.5.1 SDM

We first compare results of a 2MISC simulation of SDM on the MP-2 against an actual implementation on an MP-2204 as a limited validation of the 2MISC simulator. For small SDMs (up to 32 K physical locations), simulated execution time is as much as 50% high for straight storage and as much as 50% low for transposed storage. The differences are primarily in the fixed times independent of N, m, and a, and as such are primarily overhead. Figure 8.2 shows extrapolations from several smaller SDMs to a 1 K x 256 K SDM. Here, simulated time is 35% less than actual for straight storage, and 5% greater than actual for the preferred transposed storage.

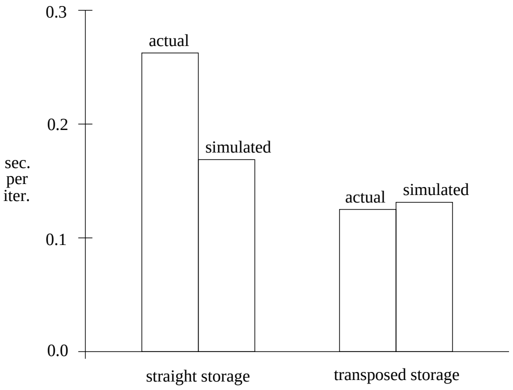


Figure 8.2: 2MISC Simulation Accuracies for 1 K × 256 K SDM

We simulate the MP-2 as well as MISC for a more fair comparison. Table 8.8 shows simulated execution (the final size is interpolated) for an average of one read and one write, with 1024-bit input and output vectors. At the target SDM size, MISC is 500 times faster than the MP-2. This is a combination of both architectural and technology advances. Adjusting the 12.5 MIPS MP-2 array control unit to the 500 MIPS supervisor makes supervisor time the same between the two architectures. Reducing the MP-2's 80 ns system clock to the 17 ns array clock speed used for the MISC analysis reduces the array time speedup from over 412 times to 86 times. Combining supervisor and array times, MISC speed up from architecture alone is about 50 times.

N	MP-2	MISC	Ratio
4 K	12.	0.19	61
16 K	18.	0.21	84
32 K	25.	0.32	110
256 K	150.	0.31	500

Table 8.8: MISC and MP-2 SDM Execution Times (ms)

Additional Architectural Support for SDM

Hamming Distance MISC's architectural advantages for SDM include the ones counter and simultaneous operation of the address unit, greatly speeding SDM's ubiquitous Hamming distance calculation. MISC's approach is faster than prior methods,[6] such as that used in PCAM, table look-up, and an anonymous algorithm from MIT and requires negligible additional hardware. Table 8.9 gives the amortized number of instruction cycles per bit for various the various methods of computing Hamming distance on MISC, the MP-2, PCAM, and conventional RISC microprocessors. The MISC ones counter configuration also permits bit-masking without any additional cycles once the mask register is loaded.

Of the methods analyzed, the special-purpose MISC hardware is clearly the fastest. The MP-2 requires many more cycles than a typical workstation due to its lack of a hardware multiply/divide unit (specifically, 95 cycles for a single 2-bit modulus operation) for the MIT hack. The MP-2 is also slower on table lookup due to the 40 cycle off-chip memory access and 50 cycle local addressing. The MISC PE architecture calculates Hamming distance nearly 5 times faster than a RISC microprocessor with a single-cycle hardware multiply/divide unit of equivalent cycle time. A 16-bit Transputer is capable of executing an unmasked Hamming distance calculation in 0.69 cycles per bit.

Method	Typical RISC	MP-2	PCAM	MISC
Bit-serial			1.0	
Table-lookup	0.4	6.2		0.25
MIT Hack	0.3	4.2		
Ones Counter				0.06

Table 8.9: Hamming Distance (cycles per bit)

Other Features MISC has other advantages beyond Hamming distance and single-cycle off-chip memory. Accessing the next active PE (the next physical location in the SDM transposed-storage implementation) is 80 times faster on MISC due to the multiple-response resolver network. Empirical tests on the MP-2 using the `selectone()` give time linear in the number of active processors, 240 cycles per processor, as opposed to the 3 cycle-delay estimated for the MISC resolver network. Additionally, the MP-2 is limited to 256 K bytes of memory per PE, severely limiting the size of an SDM. MISC allows at least 4 times as much memory (1 M byte per PE).

MISC has a similar advantage in computing log sums in the straight storage SDM implementation. For log sum, time is dependent on the total number of processors. Empirical tests on the MP-2 give time $O(\sqrt{P})$ (where P is the total number of PEs in the array) as communication with non-adjacent PEs has time linear in distance. Analysis of microcode for MISC gives time $O(\lg \sqrt{P})$ due to the bypass-mesh network. Including constant factors, MISC shows nearly a factor of 30 advantage for a 1 K PE array.

---

[6] Cray vector-processors and Control Data Corporation mainframes, as well as the Transputer, include a bit-count instruction. The Transputer instruction requires $B+2$ cycles where B is the number of ones. The author is unaware of any general-purpose SIMD processors including this instruction.

Performance Comparisons to Other Machines

Figure 8.3 shows storage capacities and speeds for a variety of SDM platforms including MISC. Execution times for SDM scale linearly in N and m, with parallel speedup linear in the number of processors. The direct implementation on the DEC Alpha achieved 1.2×10^7 MCS compared to the 5×10^{11} MCS on a single MISC PN, a factor of nearly 20,000. The improvement results from the combination of thousand-fold parallelism and special hardware for Hamming distance. More generally, the figure shows that MISC offers an order of magnitude increase in weight storage capacity over prior parallel machines and 2–3 orders of magnitude over current workstation implementations. MISC offers at least 2 orders of magnitude speed improvement over prior special- and general-purpose parallel architectures, including the Stanford application-specific SDM prototype, and 3 orders of magnitude over projected 500 MIPS uniprocessor workstations.

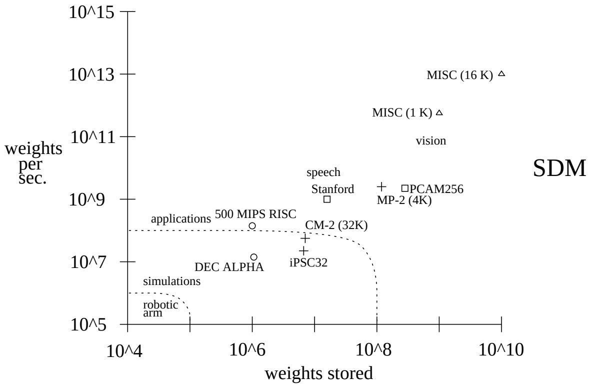


Figure 8.3: Comparison of MISC to Other SDM Implementations

8.5.2 Semantic Networks

Table 8.10 gives simulated MISC and MP-2 execution times for SNs. At the target application size, MISC builds the network and executes the queries in 90 seconds while the MP-2 requires 5870 seconds, a slowdown of 90 times. Array operations are 73 times faster on MISC than on the MP-2. Adjusting the clock period and supervisor speed, but retaining the architecture's multiplexed off-chip memory ports, the MISC array is still 15 times faster.

N	MP-2	MISC	Ratio
16 K	1270	19	67
32 K	1400	23	61
64 K	1900	26	73
256 K	5870	90	65

Table 8.10: Execution Times for Semantic Networks (seconds)

Architectural Advantages

Off-Chip Memory Access For the MP-2, 84% of execution time goes to accessing from off-chip memory at the target application size (256 K nodes). Although the expense of additional memory ports for the MP-2 cannot be justified for applications where the MP-2's many-cycle integer multiply/divide or floating-point operations represent even a small portion of the total instructions, this is clearly not the case for semantic networks where there are very few ALU cycles per marker passed.

Routing Depending on the number of processors sending, general routing on MISC's 2-d mesh ranges from about the same speed to nearly 3 times faster than on the MP-2 multistage router. Although MISC must do all routing through the mesh, its communication datapath is 16 times wider than the MP-2's bit-serial router, and the MP-2 multiplexes 16 PEs to a single router. In MISC, each PE has its own communication ports.

For the SN example application, the first marker propagation sends 14 markers and exceeds 50 markers by the fifth marker propagation. On average, 228 markers are sent, giving the MISC mesh a factor of 1.6 advantage over the MP-2 router even at equivalent array clock frequency.

The MISC PE Adjusting for cycle time, off-chip memory access, and routing speed leaves MISC with a factor of 2.5 advantage over the MP-2. This is primarily attributable to the multi-ported register set providing single-cycle operations rather than the three-cycle instructions on the MP-2. The MP-2 is a 32-bit machine whereas the MISC-1 is 16 bits, so that there is nearly a factor of two advantage from the various other MISC PE features.

Performance Comparison for Other Machines

The published literature provides little data for direct comparison of different semantic network implementations. Results are rarely presented in terms of marker propagation cycles per second, and the details of queries are often omitted in presenting query response times. The performances comparisons of *Figure 8.4 should be taken as only a general indication of relative performance.*

Unlike SDM, applications are more speed than memory limited, with total execution time $O(N^2)$ on both serial and parallel machines. Based on simulator execution time versus the simulated MISC times, MISC offers two to three orders of magnitude speedup over current workstations and at least an order of magnitude over projected 500 MIPS workstations for minimal SN models. Most current workstations have sufficient memory for a 512 K node SNs whereas a single MISC PN could hold as many as 4 M nodes at 128 bytes storage each. A 1 K PE MP-2 could hold 512 K nodes, and larger MP-2s proportionately more.

IXM2 (maximum 64 K node SNs) and SNAP-1 (maximum 32 K node SNs) are special-purpose hardware for semantic nets. Scaling on these systems is also exponential in N and MISC *appears* to offer significant speed benefits over SNAP but this may very well not be the case. First, the SNAP queries are more complex than the simple "mark all nodes then propagate" of the simulation's minimal model. Also, the comparison here is between a standard-component prototype (SNAP-1) versus a proposed custom associative array (MISC). The 144 DSP processors in the SNAP prototype total 1,700 MIPS compared to the projected 30,000 MIPS for a 1K PE single PN MISC. Moldovan proposes a larger SNAP with custom processing elements [M^{+}92]. Although a SNAP-1 successor will be more application-specific (with the attendant performance and resource-utilization advantages) in comparison to the MISC design, both are optimized for associative matching and local communication, so that MISC should not have a major speed disadvantage.

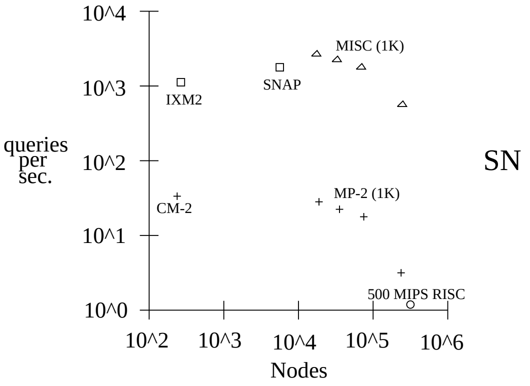


Figure 8.4: Comparison of SN Query Performance

8.5.3 Subgraph Isomorphism

Table 8.11 presents simulated times on MISC and the MP-2 for one graph distribution, one M matrix formation, and six refinement invocations. Each refinement invocation iterates until all graphs have either failed or their M matrix no longer changes. The MP-2 values are adjusted by the relative number of graphs; with its 32-bit PE a 1K MP-2 array can process twice as many graphs if there are more than 16 nodes each. Adjusting the MP-2 figures to the same array cycle time (supervisor time is negligible), MISC is from 3 to 19 times faster, averaging 8 times faster.

Experiment	MP-2	MISC	Ratio
small	22.3	0.25	89
cg	24.4	0.48	51
chem	18.3	0.53	35
large	24.5	0.63	39

Table 8.11: Subgraph Isomorphism Refinement Speeds, ms

Architectural Advantages

Unlike the other applications, off-chip memory reference is well below 1% of execution time on the MP-2 and MISC looses its 40:1 speed advantage for referencing off-chip memory.

The factor of 8 speedup is from only the PE microarchitecture and the local communication network and is in spite of the MP-2's 2:1 advantage in ALU width. MISC's 1-bit flag unit is used extensively in Boolean matrix transposition and multiplication, setting a specified bit in a special-purpose register based on a conditional test in a single cycle (the MISC microcode for matrix multiplication is given in Section 7.4.2). The MP-2 must use masking operations, including generating and maintaining the mask. The major savings are from MISC's bypass-mesh communication network, used in distributing the KB graph, Boolean matrix transpose, and Boolean matrix multiplication. The bypass allows distant communication on a given access to be performed in $2\lceil o/b\rceil + \left\lceil \frac{d}{10} \right\rceil$ cycles (where o is the operand width and b is the ALU and communication path width and d is the distance) whereas distant communication on the MP-2 is at least $5\lceil o/b\rceil + 2d + 24$ cycles where d is the distance (RHS `xnetp[ ]` instruction). Communication configuration establishes a coterie for each array section on MISC whereas this must be handled by conditionals and additional communications on the MP-2.

Performance Comparisons for Other Machines

Based on the simulation execution times, a single MISC PN is some 70 times faster than a 500 MIPS workstation, roughly the ratio of projected MIPS for 1 K MISC PEs to that of the workstation. The relatively low PE utilization (roughly 30%, compared to nearly 100% for the other example algorithms) largely mitigates the advantage of MISC's special features; future work with load balancing may significantly improve MISC performance for SI. Figure 8.5 compares MISC performance on retrieval from a graph database to that of other machines. *As with semantic networks (Figure 8.4) the conditions are not the same for all data points.* MISC is the only processor taking advantage of multi-level indexed search, and in the case of the CM-2 and Transputer examples, the query is a search for multiple subgraph isomorphisms in a single large graph.

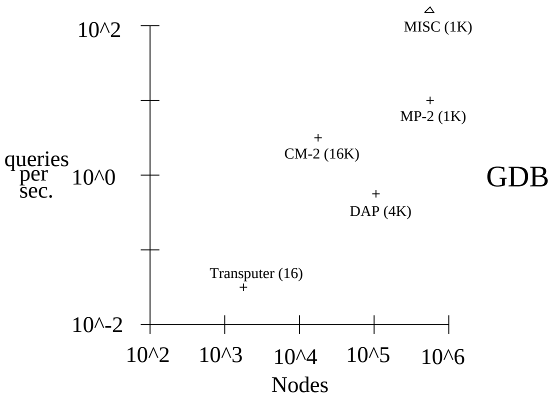


Figure 8.5: Comparison of Graph Database Query Performance

8.5.4 Cost-Performance

Overall Performance

MISC's performance as illustrated in Figures 8.3 and 8.4 and Table 8.11 is exceptional, typically two orders of magnitude over existing parallel machines. Table 8.12 summarizes MISC and MP-2 execution times for the three example applications, and includes adjustment to equivalent supervisor MIPS and array clock frequency for the MP-2. The final line of the table indicates a weighted combination of speedups (35% SDM, 20% SN, 45% refinement, as in the combined manual profiles). The MISC architecture averages well over an order of magnitude speedup, even with increasing the MP-2 array's speed by 4.8 and the MP-2 ACU's by 40 times. In comparison to projected 500 MIPS workstations, MISC shows a speedup of well over two orders of magnitude for the weighted combination of example algorithms. This is somewhat higher than the relative MIPS (30,000 32-bit integer MIPS for 1 K 16-bit PEs, excluding multiplication and division), indicating that the special PE architectural features more than compensate for communication and other parallel overhead for the knowledge processing codes simulated.

Algorithm	MP-2 existing technology	MP-2 extrapolated technology
SDM	500	63
SN	65	12
Graph KB	51	24
weighted	211	35

Table 8.12: MISC Speedup Over MP-2, Total Execution Time

MISC's architectural features can be better appreciated by factoring out MP-2 architectural effects on array execution time, as seen in Table 8.13. In the table, adjustments are cumulative (stepping from the existing MP-2 to the proposed MISC), and all columns include adjustment for processing a greater number of graphs in refinement on a 32-bit versus 16-bit PE. Removing off-chip memory reference times from the MP-2 simulated times removes MISC's implementation advantage of one memory port per PE and shows that off-chip memory referencing accounts for nearly half the MISC advantage over the MP-2. Adjusting MP-2 times for 1-cycle "minimal instructions" (such as integer add and logical operations) indicates the advantage of MISC's multi-port register set. MISC execution times are then adjusted for extending MISC to 32-bit ALUs and memory ports, and finally to a MISC with 32-bit ALUs, memory ports, and communication ports for a more fair comparison to the 32-bit MP-2. These final columns show MISC's architectural advantages irrespective of multi-port registers, advanced array memory, and implementation technology. Averaged over the example applications and allowing a safety factor,[7] MISC

---

[7] SUSPENS and Suspense early analysis projections have a typical error of 15%; the worst error observed in Suspense validation was 60%. More limited tests of 2MISC showed errors of 5–50%. Expected errors are thereby on the order of 30% for speed (50% for space-time). A safety factor of 2–4 allows for worst-case error based on validation to date.

shows a factor of 10 speed improvement as a result of tailoring the array architecture to AI and example application characteristics.

	MISC 16-bit ALU, mem., comm.				MISC 32-bit ALU, mem.	MISC 32-bit ALU, mem., comm.
Algorithm	Existing MP-2 Technology	Extrapolated MP-2 Technology	MP-2 0-cycle off-chip mem.	MP-2 1-cycle instr.	MP-2 1-cycle instr.	MP-2 1-cycle instr.
SDM	412	86	67.9	50.9	112.0	112.0
SN	73	15	2.4	1.2	1.8	2.4
Refine	35	7	3.6	3.6	6.2	9.3
Weighted	175	36	26	20	42	44

Table 8.13: MISC Speedup Over MP-2, Array Execution Time

System Cost Comparison

Figure 8.6 compares costs of the proposed MISC-1 implementation and the MP-2. Costs for both are the Suspense cost metric[8] avoiding any assumptions of manufacturing cost mark-up and monetary inflation on the MP-2. Balance-of-system (BOS) costs include the supervisor components and mesh wrap-around chips (router chips for the MP-2), the MCM and PCB, power, and cooling. The MISC implementation includes 1,200 pin PE die. Other alternatives, such as fewer PEs per die, reduced off-chip datapath widths and memory sharing, a PE on the memory chip, and wafer-scale integration are more costly under the Suspense cost model. Although a 32-bit MISC PE would have nearly twice the performance and promises improved cost-performance, the wider communication and memory data paths may pose significant signal-integrity problems so that MISC-1 costs are based on a 16-bit PE.[9] Compared to the existing MP-2, the MISC array is 189 times faster and costs 27% more giving a cost-performance nearly 150 times higher than that of the MP-2 on the example AI applications. This is, however, an unfair comparison. Extrapolating the MP-2 to the same manufacturing assumptions and including 512K bytes of memory per PE to coincide with the memory size used in the MISC system early analysis, but retaining the shared 4-bit memory ports, substantially lowers cost despite the increase in memory. Since MP-2 PE chips are not dominated by high pin-count they become small and inexpensive at 0.35μm CMOS and their cycle time is nearly equal MISC's. Under these conditions, the

---

[8] Assuming the combination multi-chip module and PCB described in Section 7.3. Suspense projects such an implementation of the MP-2 to have a system clock 10are used for comparison.

[9] Suspense analysis of PCB implementation with eight 32-bit PEs on a chip shows a 21% increase in the system cost-metric. Pin cost increases on a per PE basis, and the PCB is over 26% larger. There is also a 4% reduction in array clock frequency. Combined with the cycle-count adjustments for 32-bit PE, memory, and communication and including supervisor time there is nearly a factor of two improvement in speed and a 31% improvement in cost-performance. It is likely to be more desirable to improve speed in a single PN rather than distributing an algorithm over several PNs and dramatically increase communication time. For a hybrid MCM-PCB implementation, the MCM remains 1.6 inches square and there is negligible slowdown; system cost increases by 8%. This gives a 41% improvement in cost-performance and a factor of 1.8 speedup over the 16-bit MISC PE (16 per chip). This is an area for future simulation and analysis, but represents a tuning of the MISC implementation rather than more basic architectural issues.

MISC array is 40 times faster and costs 45% more giving MISC a cost-performance well over an order of magnitude better than the MP-2 for the weighted combination of example KP applications.

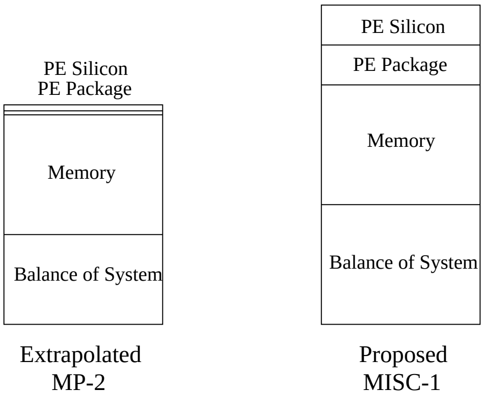


Figure 8.6: Comparison of MISC and MP-2 Cost Breakdown

Extensibility

With the available die area, the floating-point features of the MP-2 as well as a hardware multiplier could be incorporated into MISC along with additional knowledge processing features, or many of MISC's features could be added to a machine such as the MP-2. First-cut analysis indicates that the multiported register set and wide non-multiplexed memory would improve performance on numeric and general codes by a factor of 1 to 3 depending on the prevalence of load/store, and local addressing instructions. It appears likely that most if not all of the architectural features developed for knowledge processing applications could be incorporated into a numerically oriented machine while maintaining or even substantially improving cost-performance on scientific and engineering as well as AI codes. Such a machine would have an even wider range of viable applications than current "general-purpose" architectures.

8.6 Summary

This chapter has evaluated the MISC architecture and programming model. The evaluation establishes feasibility, showing that the associative programming model can express the complex, dynamic data and control structures characteristic of knowledge processing and that the MISC architecture supplies the necessary hardware support. We establish that the SEGUL notation is an effective programming tool, succinctly and clearly expressing knowledge processing algorithms, including multiple levels of parallelism. It is not clear, however, if SEGUL can be automatically converted into efficient machine-level code for MISC.

Generality is established by identifying parallel knowledge processing and related algorithms implemented on machines or using programming models less capable than those of MISC. These algorithms include areas such as common AI methodologies, graph algorithms, computational geometry, image processing, sorting and searching, linear algebra, and filtering and compression.

Simulation results show that MISC has at least an order of magnitude higher performance than existing parallel machines for the example algorithms. This includes a leading machine tailored to numeric computation, even when that machine is advanced to the same technology assumed for MISC. This chapter also notes that the simulations support the architectural choices based on manual profiles and discusses how the MISC features benefit each of the example algorithms.

Simulations and early analysis also establish efficiency and good utilization of the MISC's special features by showing that MISC has significantly greater performance than a machine with a similar cost-metric, and that execution speeds are often greater than would be expected by comparison of projected integer MIPS. Finally, this chapter notes that the MISC is economically extensible; the array processor die can readily accommodate additional functional units.

9. Conclusion

9.1 Research Summary

The MISC architecture and SEGUL pseudocode notation meet the goals set for the research, demonstrating that associative processing and an extended SIMD architecture can efficiently support massively parallel processing of complex structured data for knowledge retrieval. The following summarizes the research chronology.

Associative Processing and AI Research began by developing a programming model and pseudocode notation, SEGUL, emphasizing associatively processing collections of data objects. This work established that the associative model is capable of succinctly expressing knowledge processing data structures and algorithms with both implicit and explicit parallelism. The three example algorithms (sparse distributed associative memory, semantic networks, and graph knowledge base retrieval) used throughout this dissertation and pseudocoded in SEGUL either directly or indirectly cover the four "pillars" of AI: retrieval, learning, representation, and search.

Algorithms and Architecture The three examples, along with general characteristics of knowledge processing codes, formed the basis of MISC's development and evaluation. Manual analysis of the examples identified several characteristics important to MISC and massively parallel knowledge processing. Sub-linear scaling of the number of processing elements, resulting from the pruning effects of advanced retrieval algorithms, was the most pervasive. A thousand or fewer processors are sufficient even for unusually large knowledge bases so that each MISC processor node incorporates a relatively small SIMD array. There is only a small amount of communication between the arrays. The processor scaling in conjunction with memory referencing characteristics requires that each processing element have a high-bandwidth connection to a large memory. General AI characteristics require that this memory be locally addressed. Analysis of communication patterns within each associative array indicate the need for a mesh network with wide communication links and a means of bypassing intermediate processors. Instruction profiles show the expected "tight inner loop;" each example algorithm has a single macro operation that occupies the majority of execution time. Providing processor features to accelerate these operations substantially improves performance with only a small amount of added hardware.

Architecture and Implementation Technology An early analysis tool, Suspense, projected performance and cost for a proposed implementation, showing that MISC is both viable and cost-effective for the three example algorithms typical of memory-based approaches to massively parallel AI. Meeting the array memory and communication needs requires high connectivity. Under the Suspense cost model, a combination of high pad count die and advanced packaging is the better of several alternatives.

Evaluation The results fully support the thesis. The MISC architecture and associative programming model demonstrates the feasibility of SIMD processing dynamic, complex data structures in support of knowledge processing, including relational-structure processing, through simulated execution of the example algorithms. The MISC architecture is also general. In addition to the capacity for additional features[1] the research establishes that numerous algorithms other than the examples can be efficiently executed. The architectural insights are applicable to SIMD architecture and knowledge processing beyond MISC.

An architectural-level simulation showed that for the three example algorithms MISC has an order of magnitude higher performance than current numerically-oriented SIMD machines, even when the latter is adjusted to the same clock speed as MISC. MISC's projected cost is not significantly higher, indicating effective use of the special features.

9.2 Contributions

By considering applications, algorithms, and technology in concert with the development of the MISC architecture, this dissertation makes contributions in several areas. This section highlights those contributions; the following section describes them in greater detail.

- **Design tools**. The early analysis tool Suspense extends and improves upon Bakoglu's SUSPENS, providing more flexible input descriptions, additional models, and cost metrics.
- **Associative processing model**. The MISC associative processing model SEGUL provides a new programming model demonstrating the following.
    - The use of nested collections as a viable framework for expressing structured and dynamic data objects typical of knowledge processing.
    - SIMD associative processing of dynamic data structures and relational-objects.
    - The expression of multiple levels of parallelism, including lower levels implicitly in nested collections and upper levels explicitly in message passing, establishing the feasibility of programming MISC's MSIMD architecture.
- **Parallel Algorithms**. Parallelization and analysis of the three example applications include several results and innovations.
    - Pruned search is superior to data-parallel search when multiple levels of parallelism are available.
    - Search pruning effects result in sub-linear scaling of the number of processors in the knowledge base size, effectively bounding the processor array size.
    - k-d trees are not generally applicable for the high-dimensional near-neighbor search in SDM.
    - The incorporation of lattice codes into the type-hierarchy for graph representation, permitting the parallel formation of multiple candidate match matrices in subgraph isomorphism testing, improving performance and reducing communication.

---

[1] Analysis shows that pad spacing dominates PE die area in the proposed implementation, so that additional features (including additional support for numeric operations) can be added to the processing elements without significant additional cost.

  - Modifications to Ullmann's subgraph isomorphism algorithm for the characteristics of conceptual graphs.
  - A parallel Boolean matrix multiplication algorithm that exploits multi-bit processors and partitioning the associative array communication network to improve performance and process multiple matrices simultaneously.

- **Architecture**. The architecture incorporates many innovations into contemporary SIMD architecture, to meet the characteristics of knowledge processing and the example algorithms. They are generally applicable to knowledge retrieval and other applications. Additionally, analysis shows the promise of high-connectivity advanced packaging for SIMD architectures in memory-based AI applications.
  - An MSIMD global architecture of independent associative arrays, improving control flexibility and performance while reducing system cost.
  - An ALU-wide datapath and local addressing lines to off-chip memory for each PE to avoid a memory bottleneck and support knowledge processing data structures.
  - A memory hierarchy consisting of unusually large off-chip DRAM, a row-cache incorporated into the DRAM chip, and on-chip registers to meet required memory access time.
  - A 2-d mesh network incorporating ALU-wide links and circuitry to bypass data across a PE, speeding communication and allowing array partitioning.
  - Additional functional units incorporated into the standard PE to better support associative knowledge processing, providing wide coverage of both low-level and high-level operations.
  - Cost-justified use of emerging high pin count integrated circuit die and multi-chip modules.

9.3 Research Results

Design Tools

Specific enhancements to the predecessor SUSPENS include: new and more detailed models specifically for RAM and CAM, the ability to combine multiple circuit types on a single die, the ability to choose the system-level interconnection topology, the addition of yield and cost metrics, and improvement of several of the modeling equations. Verification shows increased accuracy for memory chips and parallel systems. Suspense verification shows a typical error of 15% for speed and area and 35% for power at die, packaged chip, and system levels. Design tool development also includes the 2MISC simulator. Its unique features are decoupling algorithm execution from instruction and memory profiling and calculating execution time (allowing easy simulation of other architectures in addition to MISC), and a combination of SIMD and MIMD parallelism.

Associative Processing Model

The associative model is an expressive notational framework for developing and communicating knowledge retrieval and other algorithms, emphasizing a set and relation representation of the underlying mathematical structure. The model combines prior work in collection oriented languages, multiset transformation, and associative and content-addressable memory processing. Most importantly, it extends associative processing to dynamic, structured data. The overall programming style is that of cycles of parallel match, set-formation, and operations on set members. The single data structure is a nested multiset; various restrictions give sets, tuples and sequences. The model supports non-uniform nesting, including mixed collection types (set or tuple for example) and member types (integer or Boolean vector, for example) as well as dynamic collections, resulting in significant flexibility in representing compound data objects. The data objects can be treated as a single object in high-level matching and element-wise operations and subelements are accessed either by content or position. Multiple levels of parallelism are implicit in the nested collections and coarse-grain parallelism is explicitly expressed using message passing.

Parallel Algorithms

Sparse Distributed Associative Memory Parallelization work includes a new SDM implementation on the MasPar MP-2 in addition to simulated execution on 2MISC. The MP-2 implementation is the fastest known to the author, but the published literature does not include sufficient detail on other implementations to determine whether this dissertation's implementation incorporates significant new techniques.

Semantic Network The semantic network example incorporates lattice codes for inheritance, which improves performance by reducing the size of the network and eliminating the marker propagation cycles otherwise needed to establish inheritance. This was first proposed by Geller [Gel94a, Gel94b] and is also used in the IXM2 semantic network processor. Our simulations provide a preliminary evaluation of performance using this technique on synthetic data sets.

Subgraph Isomorphism Simulation of the subgraph isomorphism refinement procedure substitutes lattice codes for node labels, avoiding consultation of a central node-type hierarchy and allowing the match matrix to be created for all nodes of all active graphs simultaneously. Performing subsumption tests on the lattice codes also has application in type hierarchy and partial-order maintenance, allowing simultaneous marking of all ancestors or descendents and in turn performing such operations as finding the intersection of descendent subtrees in constant time. Improvements over prior implementations of the refinement algorithm also include a packed representation of the Boolean matrices that allow each PE to process multiple elements simultaneously and a single associative array to process many graphs simultaneously. New algorithms for Boolean matrix transposition and multiplication exploit coteries formed in the bypass-mesh network and special features of the MISC processing elements.

Multi-Level Indexed Search Analysis shows that when nested parallelism is present, search methods with pruning are faster than exhaustive parallelism for a reasonably bounded number of processors and always have a better space-time product with an unbounded number of processors. This behavior is observable for SDM and semantic networks, and proven in a formal analysis for the graph knowledge base.

Architecture

System Each MISC processing node has an associative array accompanied by a supervising microprocessor and co-processor, array control unit, and interfacing to a global communication network. The hierarchy of MIMD and multiple SIMD arrays is intended to support all levels of parallelism and exploits clusters of high internal computation and communication with low external communication. With each node processing an entire task, global communication between nodes is sufficiently low that existing high-speed buses and point-to-point networks can support as many as a hundred nodes. Emerging communication networks and technologies should permit an order of magnitude more nodes, so that MISC scalability would most likely be limited by available application parallelism than hardware.

Array Memory And Communications To meet high bandwidth and low-access time requirements, each PE has its own port to row-cached DRAM. Analysis shows that sufficient on-chip memory to alleviate the need for one operand per PE per cycle from off-chip memory is not cost justifiable except for small application sizes. Communication between PEs is via a 2-d mesh with links of the full ALU-width in the proposed implementation. MISC enhances the mesh with the ability to bypass intermediate PEs, speeding communication to distant PEs and permitting array partitioning and true logarithmic time reductions. The bypass feature is not unique to MISC; this dissertation shows the advantages for the new Boolean matrix multiplication algorithm.

Processing Elements Initial design analysis established that bit-cell and bit-serial CAM have a poor space-time product in comparison to word-wide SIMD PEs even where matching and associative processing are emphasized. Analysis also established that enhancements to standard SIMD architectures, returning many of the features of associative processors, can extend fixed-format content-addressable parallel processing to efficient associative operations on the dynamic, structured data typical of knowledge processing applications and dramatically improve performance with a relative small increase in hardware resources.

A general-purpose 16 or 32-bit ALU and register set thereby forms the backbone of the MISC associative array processing elements. The register set is multiported, providing in single-cycle operations. Additional functional units complement the ALU, improving performance on associative knowledge processing operations. Each feature serves multiple purposes. The features include a 1-bit flag manipulation unit, ALU output ones counter, ALU output masking, and a multiple-response resolver network. A local address unit is crucial in providing the necessary flexibility in accessing data structures and also serves as an adder-accumulator. Communication, flag-unit, and address-unit configuration registers provide additional local control.

The MISC PE architecture is richer than typical SIMD PEs, and offers the potential for pipelined and superscalar operation within the SIMD framework. It retains the RISC philosophy in maintaining single-cycle instructions, including enhancements only for common operations and excluding enhancements that increase cycle time. To a significant degree, MISC supports the union of the instruction sets and programming models of most prior processors used for AI.

Proposed Implementation The high-bandwidth memory and communication ports require 1,200 connections per 16-processor chip. Under the Suspense cost model, emerging packaging techniques are more cost-effective in resolving implementation requirements than other implementation alternatives. Early analysis also shows that the minimum processor chip size, forced by the high pin count, leaves sufficient room for a number of additional functional units to further support knowledge as well as numeric processing. At the system level, Suspense shows that multi-chip modules can be expected to offer significant reductions in system speed, size, and power at a small increase in cost. Early analysis indicates that a proposed 1 K processor MISC node (of 16-bit processors with 512 K Bytes memory each) using multi-chip module packaging would operate at 60 MHz, occupy 80 square inches, and consume less than 150 watts. Suspense also projects that the current MasPar MP-2 architecture implemented in the same technology and the same memory size as MISC would have a production cost some 40% less (at equal manufacturing volume), but would be more than a factor of 10 slower than MISC on the suite of example knowledge processing algorithms.

9.4 Future Directions

Numerous items for future research have been mentioned throughout this dissertation. The following summarizes key areas.

A MISC Machine Prototyping a MISC machine would be an important next step, validating (or refuting) results of the early analysis. Any prototyping effort must be prefaced by simulations of a broad range of algorithms, including complete applications. Comparing MISC cost-performance to other machines, including MIMD architectures would also be important. Design refinement from architectural description to chip and board layout could then proceed. These efforts would likely identify further opportunities and features as well as difficulties to be overcome. Incorporating sub-controllers for array partitions is one opportunity, and signal integrity for a parallel system with high pin counts is a key concern.

Developing SEGUL and Suspense The SEGUL notation is promising as a programming tool and perhaps even as a programming language. Pursuing this avenue would involve elaborating and formalizing the current notation and investigating machine-level data structures and routines and their automatic extraction from high-level descriptions.

Implementing Applications Work on the parallelizations of the example algorithms could also be continued, including implementation on existing parallel machines. A complete parallel graph knowledge base, with analysis of load balancing, would be particularly important. The existing SDM implementation could form the basis of research in input coding, the effects of correlated input, and metanetworks of multiple associative memories. Work parallelizing other lattice coding techniques, particularly incremental updating and characterizing their application to semantic networks, could also be important.

Fundamentals Further long-term research could also lead to more fundamental issues. The investigation of algorithm-architecture-technology interplay could be extended into other technologies such as optical and molecular computing, potentially providing more fundamental understanding of the relationship between knowledge processing methodologies and their physical realization. The question of the ubiquity of the benefits of pruning and filtering might form an important part of this investigation.

Each of the example applications had a single macro operation accounting for the majority of execution. The dominance of a “tight inner loop” in execution time has been observed by others, suggesting the fundamental question “Is there a tractable set of such operations covering most massively-parallel AI methodologies” and the even more fundamental broad question “Can the operations be mathematically formalized into a set of universal primitives for knowledge representation and processing?” Interpretation of lattice codes as representations of inheritance and implication may provide insights, and in conjunction with SDM coding techniques, may suggest relationships between connectionist and symbolic representations.

This work could begin by strengthening the relationships between memory-based massively parallel AI, classical AI methodologies, the associative model, and parallel architecture in a formal framework as well as the exploration of additional algorithms. This could provide a more basic understanding of massively parallel algorithms and architectures for machine intelligence.

9.5 Implications

In meeting the research goals, this dissertation develops architectural innovations that promise high-performance cost-effective massively parallel processing for AI, either through the actual implementation of a MISC Machine or through application of the research results to other parallel machine development.

A High-Performance Platform for AI Recent work in massively parallel AI has primarily used the CM-2. On a MIPS basis, the MP-2 offers an order of magnitude increase in performance, and analysis shows that a MISC Machine could potentially offer another two orders of magnitude speed increase over the current MP-2 for knowledge retrieval and processing applications. It is reasonable to expect this tremendous increase in speed, and projected cost-effectiveness, to enable new AI research and applications as well as to improve cost-performance in other application areas.

Design Methodology This research has demonstrated early analysis tools and high-level simulation as a research and development technique. This approach, and the Suspense tool in particular, may be useful for other projects in anticipating constraints, in identifying viable opportunities, and in establishing and tracking performance and cost goals.

The Associative Programming Model Although developed only to aid in MISC's development, the associative model and its SEGUL notation has significant advantages as a parallel algorithm specification language, as the basis of a programming tool, and potentially even as a programming language for MISC and other architectures. SEGUL could be particularly important for programming applications with structured data and multiple levels of parallelism. The examples in this dissertation show the usefulness of nested collections as an intermediary between mathematical formalism in high-level algorithm descriptions and machine-level representation of knowledge, and suggest the viability of associative processing as a basis for massively parallel knowledge processing.

Principles for Parallel Knowledge Retrieval and Processing In addition to the specific features selected for the MISC architecture that may be incorporated into other machines, this dissertation has uncovered several principles important to massively parallel architectures for AI. Surprisingly, even with an associative programming model, content-addressable memory does not appear be cost-justified for general knowledge processing except where an extremely high percentage of instructions are matching operations and the majority of words must be matched against. Filtering and pruning techniques such as multi-level indexed search offer tremendous space-time advantages, with the important architectural implication of smaller processor arrays with accompanying communication clustering. As a related issue, the research identified the importance of extremely high-bandwidth access to large off-chip memory and extended mesh communication, and the cost-justification of the advanced packaging required to meet these needs. Additional functional units for the "tight inner loops" of key algorithms are also advantageous at negligible increase in system cost, potentially resulting in architectures supporting both knowledge and numeric processing.

9.6 Closing

The MISC architecture and associative processing model demonstrate the feasibility of massively parallel associative processing for knowledge representation, retrieval, and processing. By exploring the interaction between algorithm, architecture, and technology this dissertation develops a proposed machine with projected performance and cost-performance an order of magnitude greater than a numerically oriented massively parallel architecture for a suite of three knowledge-retrieval algorithms. The associative pseudocode notation succinctly expresses knowledge representations and processing algorithms and demonstrates the programming of the MISC architecture with multiple levels of parallelism. The proposed architecture is general-purpose, providing new insights into massively parallel AI and promising improved computational support for a variety of advanced applications.

Appendices

Appendix A. SUSPENS

This appendix presents the equations and parameters for the SUSPENS early analysis tool as presented in Chapter 9 of [Bak90] and summarized in this dissertation's Chapter 3. Tables A.1 and A.2 list the symbols used here and in Appendix B. Bakoglu's notation is changed in several places for consistency and clarity.

A		area	D		edge dimension
	A_{ad}	active die		D_c	chip
	A_{bc}	memory bit-cell		D_g	gate
	A_m	module		D_m	module
	A_r	total resolver		D_p	package
	A_{rl}	resolver logic	d		defects per square cm
	A_{rw}	resolver wire	e		efficiency
	A_t	total tile		e_w	wiring utilization efficiency
B		bits	F		footprint
	B_m	total bits in a memory		F_P	user package footprint
C		capacitance		F_c	chip fanout
	C_{ed}	effective die		F_p	package footprint
	C_{gin}	gate input	f		frequency
	C_{int}	interconnect		f_c	chip
	C_l	load		f_d	*fraction switching*
	C_m	total module		f_g	gate fanout
	C_{pad}	pad		f_{ld}	*effective logic depth*
	C_t	tile		f_m	module
	C_{tr}	transistor		f_s	system
	c_{ea}	effective address line	K		
	c_{eb}	effective bit line		K_p	Rent's coefficient
c		capacitance	k		
	c_{ec}	effective column line		k_i	gate input stage ratio
	c_{em}	effective match line		k_o	gate output stage ratio
	c_{ew}	effective word line	L		length
	c_{ls}	*number* of columns		$\overline{L}$	average length
	c_m	switched during match		$\overline{L_{ad}}$	address line
	c_{mt}	*cost metric*		L_c	column
	c_{mw}	switched during multi-write		$\overline{L_d}$	data line
	c_r	resolver		L_r	row

Table A.1: SUSPENS and Suspense Symbols

A.1 Chip-Level Equations

Average interconnect length in terms of gate size is given by

$$\overline{R} = \frac{2}{9}\left(7\frac{N_g^{p-0.5}-1}{4^{p-0.5}-1} - \frac{1-N_g^{p-1.5}}{1-4^{p-1.5}}\right)\frac{1-4^{p-1}}{1-N_g^{p-1}},$$

where N_g is the total number of gates and p is Rent's coefficient for on-chip interconnection. Gate dimension in μm is

$$D_g = \frac{f_g\overline{R}p_w}{e_w n_w},$$

N		number, output driver stages
	N_a	memory sub-arrays
	N_c	chips
	N_g	gates
	N_p	pads
	N_s	sockets
	N_t	tags per word
n		number of tiles on chip
	n_w	number wiring layers
O		
	O_b	memory output bits
P		number of pins
	$\overline{P}$	average pincount
	P_m	maximum pincount
p		Rent's coefficient
	p_p	pad pitch
	p_w	wiring pitch
Q		number of transistors
	Q_l	CAM latch
	Q_m	CAM match logic
R		resistance
	$\overline{R}$	Rent's average *length*
	R_d	effective driver
	R_{dr}	effective off-chip driver
	R_g	effective gate
	R_{int}	interconnect
	R_{tr}	transistor
r		
	r_{ws}	*number* of rows
s		
	s_r	row spacing
T		time, delay
	T_b	bus interconnect
	T_{cdc}	column decode and sense
	T_{cm}	CAM match
	T_{cmw}	CAM multi-write
	T_{cr}	CAM resolve
	T_d	total die
	T_g	gate
	T_l	CAM register load
	T_m	module edge interconnect
	T_{nn}	nearest neighbor
	T_p	general propagation
	T_r	random interconnect
	T_{rdc}	row decoder
	T_{rl}	resolver logic
	T_{rr}	total resolver
	T_{rw}	total read, write
	T_{ti}	delay of i^{th} tile
	T_w	word output
t		time
	t_{adr}	address line
	t_{bl}	CAM write line
	t_{blr}	bit-line, register
	t_{bls}	bit line, SRAM
	t_{blr}	bit line, CAM match
	t_w	word line
V		volts
	V_c	inter-chip communication
	V_{dd}	CMOS positive supply
v		
	v_m	electromag. prop. speed
	v_{mfr}	manufacturing volume
W		watts
	W_c	chip
	W_g	gate
	W_m	module
Z		
	Z_o	characteristic impedance
α		general coefficient
β		general coefficient
γ		general coefficient
η		Rent's coefficient
λ		minimum drawn feature

Table A.2: SUSPENS and Suspense Symbols, continued

where f_g is average gate fan-out, p_w is wiring pitch, e_w is wiring efficiency, and n_w is the number of wiring levels. Area of the chip is then simply

$$D_c = \sqrt{N_g}D_g$$

and average wiring length in μm is

$$\overline{L} = \overline{R}D_g.$$

Effective gate output resistance and input capacitance are calculated as:

$$R_g = f_g \frac{R_{tr}}{k_o}$$

$$C_{gin} = 3k_i C_{tr}$$

where R_{tr} is the effective on resistance of a minimum-sized nMOS transistor, C_{tr} is the input capacitance of a minimum-sized transistor, and k_o and k_i are the input and output stage sizing ratios. Average gate delay is

$$T_g = f_g R_g \overline{L} C_{int} + f_g R_q C_{gin} + R_{int} C_{int} \frac{\overline{L}^2}{2} + R_{int} C_{gin} \overline{L},$$

where C_{int} and R_{int} are interconnect resistance and capacitance per unit length.

The chip frequency is calculated as

$$f_c = \left(f_{ld} T_g + R_{int} C_{int} \frac{D_c}{2} + \frac{D_c}{v_m} \right)^{-1};$$

f_{ld} is the effective logic depth and v_m is the electromagnetic wave propagation speed of the dielectric. Chip capacitance is given by

$$C_{ed} = \frac{D_c^2 n_w e_w C_{int}}{p_w} + 3C_{tr} k_i N_g f_g$$

and total chip power by

$$W_c = \frac{1}{2} f_c f_d C_c V_{dd}^2,$$

where f_d is the fraction of simultaneously switching gates. Finally, the number of pins follows Rent's rule:

$$P = K_p N_b^{\beta}.$$

Table A.3 lists the λ-dependent coefficients and Table A.4 lists the chip-type dependent coefficients from [Bak90].

Parameter	7 μm nMOS	3 μm nMOS	2 μm CMOS	1.3 μm CMOS	0.7 μm CMOS	0.35 μm CMOS
p_w (μm)	28	10	6	4	2	1
n_w	1	1	2	3	4	5
R_{tr} (NENH) (Ω)	15000	15000	10000	8000	8000	8000
C_{tr} (fF)	20	6	4	3	2	1
R_{int} (Ω/cm)	21	120	200	375	1000	2400
C_{int} (pF$/cm$)	5	3.5	2.0	2.0	2.0	2.0

Table A.3: SUSPENS Chip-Technology Parameters

Chip Type	p	K_p	β	f_{ld}
Static Memory		6.0	0.12	
Microprocessor	0.4	0.8	0.45	~ 24
Gate Array	0.5	1.9	0.50	~ 28
Supercomputer	0.6	1.4	0.63	~ 10

Table A.4: Table Caption

A.2 Module-Level Equations

SUSPENS assumes that the system is uniformly partitioned into chips, so that the number of chips is simply

$$N_c = \frac{N_{gTOT}}{N_g}$$

(N_{gTOT} is the total number of gates in the system). Average wiring length in terms of chip size is

$$\overline{R} = \frac{2}{9}\left(7\frac{N_c^{\eta-0.5}-1}{4^{\eta-0.5}-1} - \frac{1-N_c^{\eta-0.75}}{1-4^{\eta-0.75}}\right)\frac{1-4^{\eta-1}}{1-N_c^{\eta-1}}$$

where η is Rent's coefficient for off-chip interconnections. The module area (foot print) per chip is given by

$$F_p = \max\left(D_c, \frac{F_c}{F_c+1}\frac{\overline{R}Pp_w}{e_w n_w}, \sqrt{N_p}p_p, FP\right)$$

where F_c is the chip fan-out and p_w, e_w, and n_w are the module-level wiring pitch, wiring utilization, and number of wiring layers. P_p is the pad pitch and FP is a user-supplied package size. Total module size is then simply

$$D_m = \sqrt{N_c}F_p$$

and average interconnect length $\overline{L}$ is taken to be equal to D_m.

Module frequency is calculated as follows

$$f_m = \left[15(N-1)R_{tr}C_{tr} + 2Z_oC_{pad} + R_{int}C_{int}\overline{L}^2 + \frac{\overline{L}}{v_m}\right]^{-1}$$

where N is the number of stages in the output-driver chain, Z_o the characteristic impedance of the module wiring, C_{pad} the pad capacitance, R_{int} and C_{int} the module wiring resistance and capacitance per unit length, and v_M is the electromagnetic wave propagation speed on the module. The number of output-driver stages is calculated as

$$N = 1 + \frac{\ln\left(\frac{R_{tc}}{Z_o}\right)}{\ln 5}.$$

Module capacitance is given by

$$C_m = \frac{F_c}{F_c+1}N_cN_p\left(3\frac{1-5^N}{1-5}C_{tr} + 3C_{pad} + \overline{R}F_pC_{int}\right)$$

and power by

$$W_m = \frac{1}{2} f_d f_m C_m V_{dd}^2.$$

Table A.5 list the module-technology dependent coefficients from [Bak90].

Parameter	WSI	MCM	Ceram.	PCB
p_w (μm)	20	50	500	160
n_w	2	2	16	4
p_p (μm)	40	100	250	1000
Z_o (Ω)	44	60	55	75
C_{pad} $(p\mathrm{F})$	0	0.25	0.5	10
R_{int} (Ω/cm)	15	3.4	0.4	0.5
C_{int} $(p\mathrm{F}/cm)$	1.5	1.0	1.9	0.8
v_m (cm/ns)	15	16	10	17

Table A.5: Module-Technology Parameters

A.3 System-Level Equations

The SUSPENS system model is straightforward. Total area is simply the sum of the module areas, and system frequency is

$$f_s = \min\left(f_c, f_m\right).$$

Total power is

$$W = W_g N_{gTOT} + W_m$$

evaluated at system frequency f_S.

Appendix B. Suspense

This appendix details the Suspense early analysis tool developed as part of this dissertation. The notation and symbols used are given in Tables A.1 and A.2 in Appendix A.

B.1 General Tile Model

CMOS die parameters for the tile and die models are calculated as a function of λ, with the function and its parameters derived from regression to Table 9.6 of [Bak90], with V_{dd} input by the user. The number of metal layers is reduced from that in Bakoglu's table to be more in line with commercial practice (2 layers at $2\,\mu$m, 3 at $0.6\,\mu$m, 4 at $0.3\,\mu$m), with one additional routing layer (polysilicon) used in calculating gate area D_g, giving the updated formula

$$D_g = \frac{f_g \overline{R} p_w}{e_w(n_w + 1)},$$

where f_g is the gate fanout (default of 3), $\overline{R}$ is the average interconnect length in terms of gate edge length, p_w is the wiring pitch (default of 3λ), e_w is the wiring utilization (default 0.5), and n_w is the number of metal wiring layers. Gate input capacitance is greater than in SUSPENSE; accounting for additional parasitic capacitances gives $C_{gin} = 4.5 k_i C_{tr}$. An improved signal propagation delay equation is developed by the author in [RD92] giving

$$T_p = \mathrm{prop}(L, R_d, C_l, C_{int}, R_{int}, v_m) = R_d C_l + R_d C_{int} L + R_{int} C_l L + 0.5 R_{int} C_{int} L^2 + L/v_m$$

where k_i is the input stage ratio (default of 3), C_{tr} is the capacitance of a minimum-sized transistor, R_d is the effective driver resistance, C_l is the load capacitance, L is the interconnect length, C_{int} is the distributed line capacitance, R_{int} is the distributed line resistance, and v_m is the electromagnetic wave propagation speed in the chip. This improved and generalized signal propagation equation is used at all levels of Suspense (tile through module); SUSPENS uses equations that are not appropriate under all conditions, such as nondissipative signal propagation at the module level.

Gate delay, T_g, is calculated as $\mathrm{prop}(\overline{L}, f_g R_g, C_{gin}, C_{int}, R_{int}, 15)$ where $\overline{L}$ is average on-tile interconnect length and R_g is effective gate output resistance. Tile capacitance is based on gate area and the number of gates rather than total die area

$$C_t = \frac{D_g^2 N_g n_w e_w C_{int}}{p_w} + C_{gin} N_g f_g,$$

where D_g^2 is the area of a gate and N_g is the number of gates. User input is in transistors, Q; based on surveys of layouts and validation results a gate has six transistors on average.

B.2 RAM and Register Model

The RAM and CAM tile models adjust gate output ratio, k_o, and fanout ratio to k_o to 15 and f_g to 4 to allow for the majority of gates driving long lines. Both models call the general tile mode to estimate the area of subsections such as row decoders, and use the following equation for pin count

$$P = 4 + O_b + \alpha \lg \frac{B_m}{O_b} + \beta(5 + \lg \frac{B_m}{c_{ls}})$$

where O_b is the number of output bits, B_m is the total number of bits in the memory, c_{ls} is the number of columns in a memory sub-array. The third term is for address lines, so that α is 1 if the memory is addressable and 0 otherwise. The fourth term is for inclusion of a first-responder identifier used in the CAM model, so that β is 1 if the memory is CAM and 0 otherwise.

The memory tile is divided into a number of sub-arrays, $N_a = B_m/(c_{ls}r_{ws})$, reducing times by shortening row and column line lengths at some area penalty (c_{ls} and r_{ws} are user inputs). The sub-arrays are assumed to be are laid out in two columns; this is typical of the memory layouts surveyed. The bit-cell is assumed to be square, giving row line length $L_r = \sqrt{A_{bc}}$ and column line lengths of $L_c = \sqrt{A_{bc}}$ where A_{bc} is a user-input area per bit-cell. The total area in bit-cells is simply $B_m A_{bc} \lambda^2$. The bit-cells represent numerous loads along the bit- (column) and word- (row) lines, as do the row- and column-decoder circuits for the address and data lines. These discrete loads are approximated by an effective distributed line capacitance as developed by the author in [RD92]. The effective capacitance for the bit-lines is

$$c_{eb} = C_{int} + \frac{r_{ws} C_{tr} R_{int}}{R_{tr} R_{int} L_c},$$

where R_{tr} is the resistance of a minimal-sized transistor; the resistance factors approximate the portion of discrete loading capacitance seen through the transistor resistance which connects it to the line. Bit-line propagation delays for registers, SRAM, and DRAM are calculated as follows.

$$\begin{aligned} t_{blr} &= \mathrm{prop}(L_c, R_{tr}, C_{gin}, c_{eb}, R_{int}, 15) \\ t_{bls} &= \mathrm{prop}(L_c, 2R_{tr}, C_{gin}, c_{eb}, R_{int}, 15) \\ t_{blr} &= \mathrm{prop}(L_c, R_{tr}, C_{gin}, c_{eb}, R_{int}, 15) \end{aligned}$$

The effective capacitance for the word lines is

$$c_{ew} = C_{int} + \frac{c_{ls} C_{tr}}{L_r}$$

and word-line propagation with 3 times normal final driver transistor sizing is

$$t_{wl} = prop(L_r, R_g/3, C_{tr}, c_{ew}, R_{int}, 15)$$

for registers, SRAM, and DRAM. Capacitance for calculating power for the bit-cell array is that of the switching transistors (2 per bit-cell for registers and SRAM, 1 for DRAM but at $3C_{tr}$) plus wiring capacitance

$$\mathrm{wcap}(A) = \frac{A n_w e_w C_{int}}{p_w}$$

where A is the total area. For DRAM voltage swing on the bit lines is taken as $0.02V_{dd}$ so that a factor of 0.02 adjustment is made to the bit-line capacitance in determining power. For addressed memories, row address decoder, row driver, column multiplex, and column sense amplifier areas are added based on estimated transistor count and the general tile model. The the row decoder and drivers are based on designs by Weste and Eshraghian [WE88] The number of transistors is $\lg r_{ws} + 11$; if there are two or more sub-arrays one row decoder is shared between each pair. Row decoder delay, T_{rdc}, is $3T_g$ and capacitance is

$$\mathrm{wcap}\,(A) + C_{tr} r_{ws} (\lg R_{ws} + 11) N_a/2.$$

The area and capacitance for the column multiplexers and sense amplifiers is calculated similarly, also based on designs by Weste and Eshraghian [WE88]. Equations for transistor counts are shown in Table B.1. If there is more than one sub-array, a column multiplexer and sense amplifier section is shared between 2 sub-arrays. Column multiplexer and sense amplifier delays include line propagation with effective capacitance $c_{ec} = C_{int} + c_{ls} C_{tr} / L_r$ and delay is

$$T_{cdc} = 4T_g + prop(L_r, R_{tr}, C_{gin}, c_{ec}, R_{int}, 15).$$

	$N_a = 1$	$N_a > 1$
$O_b \geq N_a$	$C_{ols}(6 + \lg(\frac{C_{ols}}{O_b/N_a}) + 9) + 12\frac{O_b}{N_a}$	$C_{ols}(12 + \lg(\frac{C_{ols}}{O_b/N_a}) + 9) + 6 + 12\frac{O_b}{N_a}$
$O_b < N_a$	$C_{ols}(6 + \lg(\frac{C_{ols}}{N_a/O_b}) + 9) + 12$	$C_{ols}(6 + \lg(\frac{C_{ols}}{N_a/O_b}) + 9) + 12$

Table B.1: Column Multiplex and Sense Amplifier Transistors

Additional data-path circuitry is included if the memory tile is addressable. If the number of output bits, O_b is less than the number of sub-arrays, N_a, then $O_b(\lg(N_a/O_b) + 9) + 11$ transistors are added, otherwise only 11. Word-output delay T_w is $6T_g$ if $O_b < N_a$, $3T_g$ otherwise. Again, area is based on the general tile model and capacitance for power is determined by wcap(A). If there is more than one sub-array, area and capacitance for the long data and address lines are added as follows

$$\begin{aligned}
\overline{L_{ad}} &= \min(1.3 L_c N_a/2, \sqrt{A_t}) \\
\overline{L_d} &= \overline{L_{ad}} - L_c \\
A &= (\overline{L_{ad}} \lg c_{ls} + \overline{L_d} \max(O_b, N_a)) \tfrac{p_w}{n_w} \\
c &= \overline{L_{ad}} \lg c_{ls} C_{int} + \overline{L_d} \max(O_b, N_a) C_{int}
\end{aligned}$$

where $\overline{L_{ad}}$ is the length of the address lines, $\overline{L_d}$ the length of the data lines, A_t the area of the tile without the lines, A the added area, and c the added capacitance. For delay, effective capacitance is

$$c_{ea} = C_{int} + \frac{\frac{N_a}{2} C_{gin}}{\overline{L_{ad}}}$$

and long address-line delay is given by

$$t_{adr} = \mathrm{prop}(\overline{L_{ad}}, R_g/\alpha, c_{ea}, R_{int}, 15)$$

where α is 3 for off-chip memory and 5 for on-chip.

Total read/write time is

$$T_{rw} = t_{adr} + \max(T_{cdc}, (T_{rdc} + T_{wl} + t_{bl})) + T_w + T_d$$

for static memory and

$$T_{rw} = t_{adr} + \max((t_{adr} + T_{cdc} + 4T_g), (T_{rdc} + t_{wl} + t_{bl})) + \max(t_{bl}, (T_w + T_d))$$

for dynamic memory. Total tile power assumes that only one SRAM sub-array is active and that 1/8 of the DRAM sub-arrays are active provided there are 8 or more sub-arrays.

B.3 CAM Model

The effective bit-line capacitance is

$$c_{eb} = C_{int} + \frac{\alpha r_{ws} C_{tr} R_{int}}{R_{tr} + R_{int} L_c}$$

where α is 1 for static CAM and 2 for dynamic CAM. The resistance terms approximate the portion of the load transistor capacitance seen through the transistor resistance, as in the RAM model. The write bit-line delay is t_{bl} =prop$(\beta L_c, R_{tr}/9, C_{tr}, c_{eb}, 15)$ where β is 2 if there is more than one sub-array, 1 otherwise. Delay for the word-match line t_{blm} and data-write line are as follows, where γ is 2 for static CAM and 1 for dynamic CAM.

$$\begin{aligned} c_{em} &= C_{int} + \tfrac{C_{ols} C_{tr} R_{int}}{R_{tr} + R_{int} L_r} \\ t_{blm} &= \text{prop}(L_r, \gamma R_{tr}, 9C_{tr}, c_{em}, R_{int}, 15) \\ c_{ew} &= C_{int} + \tfrac{\gamma C_{ols} C_{tr}}{L_r} \\ t_{wl} &= \text{prop}(L_r, R_{tr}/9, C_{tr}, c_{ew}, R_{int}, 15). \end{aligned}$$

The width of comparand and mask register circuitry is assumed to be the same as the number of columns. If $c_{ls} > O_b$, the number of transistors is given by

$$\frac{c_{ls}}{O_b}(\lg \frac{c_{ls}}{O_b} + 9) + 36 c_{ls},$$

otherwise it is $9 + 36c_{ls}$. The general tile model computes area (area(Q) where Q is the number of transistors in the random-logic circuit) and the registers are duplicated for each 2 sub-arrays. Delays for loading either register is

$$T_l = \alpha T_g + \text{prop}(\sqrt{A_t}, R_g/9, C_{tr}, C_{int}, R_{int}, 15)$$

where α is 6 if $O_b < c_{ls}$ and 3 otherwise. Sub-groups of transistors are used for computing power:

$$\begin{aligned} Q_l &= \beta(\lg \beta + 9) + 18\gamma \\ c_l &= \text{wcap}(\text{area}(Q_l) \max(1, \tfrac{N_a}{2})) + Q_l C_{tr}. \end{aligned}$$

The number of transistors in the match sense-amplifiers and tags, Q_m, is $r_{ws}(5 + 13N_t)$ where N_t is the number of tag bits per CAM word specified by the user. Power consumption and delay are calculated separately for different operations. For match, 10% of all words matching is assumed giving effective capacitance

$$c_m = N_a(\text{wcap}(L_c L_r) + 9 r_{ws} c_{ls} C_{tr}) + 0.1(\text{wcap}(\text{area}(16) r_{ws} N_a + 16 r_{ws} N_a C_{tr}).$$

Note that all cells participate, giving CAM high power dissipation. For multi-write (all matching words write a global value) capacitance is

$$c_{mw} = \beta r_{ws} N_a + \alpha N_a r_{ws} (\text{wcap}(\frac{L_c L_r}{r_{ws}}) - \beta + C_{tr}(9c_{ls} + \frac{Q_m}{r_{ws}})),$$

where α is 0.1 for static, 1.0 for dynamic, and $\beta = 4c_{ls}L_c C_{int}/r_{ws}$. Times to execute a match, T_cm, and a multiwrite, T_{cmw}, after the match and comparand mask registers have been loaded are

$$T_{cm} = t_{bl} + 3T_g + t_{blm}$$
$$T_{cmw} = \max(t_{bl}, (t_{wl} + 3T_g)) + T_g.$$

The CAM model's multiple-response resolver is based on Anderson's resolver tree with Koo's first-responder address generation [Fos76], which has a cell with 23 transistors. Resolver logic area is then A_{rl} =area$(23)r_{ws}N_a$ and logic delay is $T_{rl} = 4T_g(\lg r_{ws}N_a - 1)$. Logic switching capacitance is c_r =wcap$(A_r) + 23 r_{ws} N_a C_{tr}$. Long-wire calculations are complicated somewhat by the resolver's tree-structure. Since propagation delay can be quadratic in length, delay for each length must be computed separately. Average row spacing, s_r, is L_c/R_{ows}. Additional resolver area is then

$$A_{rw} = \sum_{i=1}^{\lg r_{ws}N_a} \frac{(2+i)r_{ws}N_a}{2^{i-1}} \frac{p_w}{n_w} s_r 2^{i-2} = \frac{1}{4} r_{ws}^2 N_a^2 p_w s_r \frac{5 + r_{ws}N_a}{n_w},$$

where the first term of the summation is the number of wires and the last term is the length of each wire and total resolver area is $A_r = A_{rl} + A_{rw}$. Total resolver delay is

$$T_{cr} = T_{rl} + 2 \sum_{i=1}^{\lg r_{ws}N_a} \text{prop}(s_r 2^{i-2}, R_g/9, C_{gin}, C_{int}, R_{int}, 15),$$

and total capacitance is

$$c_r = c_{rl} + \sum_{i=1}^{\log_2 r_{ws}N_a} \frac{(2+i)r_{ws}N_a}{2^{i-1}} s_r 2^{i-2} C_{int}.$$

Additional long lines are included if there is more than 2 sub-arrays. This increases area by

$$(2 + \lg r_{ws}N_a)\frac{p_w}{n_w}\sqrt{A_r},$$

resolver delay by

$$2\text{prop}(\sqrt{A_r}, R_g/9, C_{gin}, C_{int}, R_{int}, 15),$$

and resolver capacitance by

$$(2 + \lg r_{ws}N_a)\sqrt{A_r}C_{int}.$$

CAM tile power is then calculated based on the maximum of CAM times and total capacitances for the different operations: $\max(T_{cmw}, T_{cm}, T_l, T_{cr}), \max(c_{mw}, c_m, c_l, c_r)$. CAM results require cautions interpretation. For example, various citations present CAM time as the fastest operation, the slowest operation, and in some cases the total of all operations.

B.4 Die Model

If only a single tile is assigned to a die, the die area, speed, power, and other characteristics are simply that of the tile and pads. Otherwise, die area for active circuitry is calculated as

$$A_{ad} = \sum_{i=1}^{n} A_{t_i} + \lambda\alpha 0.01\sqrt{\sum_{i=1}^{n} A_{t_i}},$$

where n is the number of tiles, A_{t_i} is the area of the i^{th} tile, and α is 1 if n is greater than 5 and 0 otherwise. The model assumes that tiles are abutted with negligible routing area between them (hence the name tile). The second term allows for a total 64 minimum-width lines running the length of the active circuit area in 2 metal layers. Additional capacitance for this area is calculated as wcap$(A_{ad})/2$, assuming 1/2 of the lines are power or ground. Additional delay for clock skew and control distribution is added based on strongly driven lines of length 1/2 the edge of the active circuit area, giving total die delay as

$$\begin{gathered} C_{ed} = C_{int} + \frac{\sqrt{n}C_g}{\sqrt{A_{ad}}} \\ T_d = \sum_{i=1}^{n} T_{t_i} + \text{prop}(0.5\sqrt{A_{ad}}, R_g/9, C_{gin}, C_{ed}, R_{int}, 15) \end{gathered}$$

where n is the total number of tiles on the critical path and T_{t_i} is the delay of the i^{th} tile. Output driver delay and capacitance use the SUSPENS equations. The user may optionally specify low-powered drivers, for which effective output impedance is 500 Ω rather than the characteristic impedance of the typical interconnect substrate. Power is based on the total of the tile capacitances and the added interconnect area capacitance at the die speed, calculated according to the SUSPENS model.

If a die contains a single tile, the tile's pin count is used for the die; otherwise pin count is calculated as in SUSPENS. If there are more than 40 pins, Suspense adds 20% of the excess over 40 to allow for additional power and ground pins to minimize ground bounce. Unlike SUSPENS, Suspense explicitly adds area for the pads, although die are still assumed to be square. The user can select a padframe with regular pads ($200\,\mu$m × $200\,\mu$m per pad), a padframe with double rows of pads on each edge, a fine-pitch padframe ($100\,\mu$m × $100\,\mu$m pads), or a 2-d array of pads ($70\,\mu$m × $70\,\mu$m pads) distributed across the active die area. If a padframe is specified, a check is made to determine if the length of 1/4 of the pads (1/8 for the double padframe) exceeds $\sqrt{A_{ad}}$. If so, total die area A_d is determined by the edge dimension of the padframe and capacitance and delay are added corresponding to one line per pad of length $0.25\sqrt{A_d}$. Similarly, minimum pad spacing for 2-d array pads is 200 μm in each dimension. Otherwise die area is the active die area plus the total area of all the pads. The die cost-metric c_{mt} combines fixed-cost (masking charges), per-unit costs, and yield. Coefficients approximate dollar cost in 1993 dollars based on regression to sample custom-fab pricing [ISI93, Tan92].

$$c_{mt} = (1 + 1.8(2 - \lambda))(20000/v_{mfr} + 11A_d)e^{A_d d},$$

where v_{mfr} is the target quantity of good die and d is the defect rate per cm^2. The first term adjusts for minimum-feature size, the second term is the fixed and per-unit cost, and the third term is yield. The coefficients correspond to \$720 per 10 cm wafer at $\lambda = 2$ μm.

B.5 Packaging Model

The model includes calculations for stacked-die (3-D packaging) based on data in [Lei90, VL90]. As in the die model, the user can specify the number of each die to be included in the stack and the number on the critical-path for timing. Total die area, capacitance, delay (if on the critical-path), and costs are summed and a 10% allowance is added for redundancy. If a die is not active (not on the critical-path), only 1% of its capacitance is added for standby power. The average $\overline{P}$ and maximum P_m number of die pins are also accumulated and the number of interconnect traces for the stack is taken as $\min(1.2\overline{P}, P_m)$. If there are fewer than 50 traces it is assumed that all traces are on one edge of the stack; otherwise 2 edges are used. Trace pitch is $300\,\mu$m, determining the width of the stack. Stack height is total die area divided by the number of die times the width. Length is determined by the number of die, at 0.02 cm thickness each. The cost metric adds 1.5 'dollars' per die plus pin cost to the total die cost. A minimum of 21 pins per die is used to allow for heat transfer requirements.

The model adds signal propagation delay for both single-die packages and die stacks. For packaged die this is two lines at 1/2 the length of the package edge. For die stacks this is one line at the length of the stack. Interconnect parameters are taken from Table 9.7 in [Bak90], for a substrate typical of the package type. The delay caused by pad capacitances are included in the module model. The package model does include pad capacitance in determining package power so that power density (W/cm^2) is appropriate for estimating cooling requirements. The user can specify an off-chip communication voltage V_c different from the die V_{dd} to allow modeling reduced voltage-swing outputs. The package cost-metric also includes cooling, based on data published by Moresco [Mor90]. Natural convection is adequate below $0.05\,W/cm^2$ and no cost is added. Forced convection is needed between 0.05 and $0.15\,W/cm^2$ with cost approximated as $0.2W$ where W is the total package wattage. Forced convection with a package heat sink is adequate between 0.15 and $0.5\,W/cm^2$ at a cost adder of $0.5W$. Liquid cooling is typically required above this, at $0.9W$, and more advanced liquid cooling is necessary above $0.75\,W/cm^2$ with a cost allowance of $1.5W$.

B.6 Module Model

In calculating average package footprint F_p, SUSPENS' uniform chip-size is replaced by the weighted average of each package, D_p, plus minimum spacing, and the uniform chip-pin count is replaced by the weighted average of the packages. Package area already incorporates pad area and minimum footprint, and a chip fan-out F_c of 4 and wiring utilization e_w of 0.5 are assumed. Off-chip (module) propagation delays are calculated using the Suspense propagation equation for a variety of topologies by

$$T = \text{prop}(L, R_{dr}, C_l, c_e, R_{int}, 16)$$

and the parameters in Table B.2. Here $C_l = N_p C_{pad}$ At the module level, $\overline{R}$ is the average interconnect length (in terms of average package size) using system-level Rent's coefficients for the specified module architecture (microprocessor, memory, etc.), D_m is the calculated edge-length of the module (assumed to be square), and N_c is the total number of packaged chips. Output driver effective resistance, R_{dr}, is taken as that of the package type with the largest delay. For nearest-neighbor, Rents, and module edge topologies load capacitance $C_l = N_p(C_{pad} + C_{gin})$ where C_{pad} is pad capacitance and effective line capacitance $c_e = C_{int}$. For the bus topology pads are treated as a distributed capacitance as in [RD92]; $C_l = C_{pad} + C_{gin}$ and $c_e = C_{int} + (N_p - 1)(C_{pad} + C_{gin})/L$. Module delay (1/MHz) is $\max(T_{d_1}, T_{d_2}, ...T_{d_n}) + \max(T_b, \min(T_m, T_r, T_{nn}))$. Generally some signaling of each topology type is present; normally T_b dominates and corresponds to clock and control signal distribution.

Topology	Delay	Length, L	Pads Loading, N_p
Nearest Neighbor	T_{nn}	$F_p - Dc$	2
Random (Rent's)	T_r	$\overline{R}F_p$	4
Module Edge	T_m	D_m	4
Bus	T_b	D_m	$\sqrt{N_c}$

Table B.2: Module Interconnect Topology Parameters

Module interconnect capacitance is calculated to determine the additional power consumed by inter-chip communication. Its calculation incorporates two adjustments to the SUSPENS model. First, output driver transistor capacitances have already been included in the packaging model. Second, for interconnect we take the minimum of the module substrate wiring capacity and that from applying Rent's rule. Module interconnect capacitance becomes

$$c_{mi} = \frac{F_c}{F_c + 1} N_p N_c C_{pad} + \min(\frac{D_m^2 n_w e_w C_{int}}{p_w}, \frac{F_c}{F_c + 1} N_p N_c \overline{R} F_p C_{int}).$$

The number of pins for inter-module connections is determined by the module type specified. For the array type, the module is taken as a vertical slice of a 2-d mesh with each horizontal connection having 1/4 the average number of pins per package, plus an allowance for control, giving $P_m = 1.1(0.5\sqrt{N_p} D_p)$. Rents rule is applied for type supercomputer ($K_p = 82$ and $\beta = 0.25$ [Bak90]) and all other module types ($K_p = 7$ and $\beta = 0.21$ [MQF91]). Area and capacitance for inter-module connections area added based on typical pad pitch and capacitance. Inter-chip communication power is calculated from C_{mi}, V_c, and module frequency. Total power adds that of all packages, adjusted to the module frequency.

The module cost metric c_{mt} is based on *current* PCB manufacturing and *more mature* MCM and WSI manufacturing. The cost-metric totals that for all the packages and adds module substrate and assembly, power, and cooling costs using the equations below. For WSI the 'package' c_{mt} is multiplied by 1.5 to allow for redundancy, additional fixed-costs are amortized, and allowances for test and reconfiguration and input-output pins are added. The WSI 'tiles' are assumed to have a 0.05% failure after redundancy and re-routing. This gives

$$c'_{mt} = 0.9995^{N_c}(1.5c_{mt} + 10000/v_{mfr} + 100 + 0.26P_m)$$

where A_m is module area, $n_w$is the number of signal layers, and v_{mfr} is the target quantity of good modules. Where discrete chips are used, average cost per chip or package is used, along with an assembly cost of $\min(5, 250/v_{mfr} + 0.04)$. For MCM (thin-film hybrid) and ceramic hybrids a flat substrate cost per square centimeter is used due to lack of scaling information, and yield is based on an empirical formula for older IC processes with d selected empirically to give reasonable costs over the range of 1 to 5 cm MCMs. Set-up costs are amortized, an allowance is made for test and repair and pins are added. Allowance for assembly cost is added. Known-good die are assumed, with an after-assembly failure rate of 0.5%, and it is assumed that individual die can be replaced, giving

$$\begin{gathered} c_{mt} = A_m(n_w + 1)(1 + A_m d/2)^2 + 10000/v_{mfr} + 25 + 0.26P + \\ (C_a + 0.1)N_p + (1 - 0.995^{N_p})(C_p + 5) \end{gathered}$$

The resulting cost per square centimeter is in line with published figures ([Mal93, Bro91]). Although the cost of a 10 cm high-λ wafer was the starting point for the above coefficients, costs are in line with laminated MCMs when adjusted for the number of layers required at equivalent packaging densities. A multiplier of 2 is applied for ceramic hybrids. Cost for printed-circuit boards is based on established set-up and production charges, including bare-board test. Amortized assembled-board test and test-per board costs are added. Module pin costs are taken as typical for various PCB connectors, and assembly cost allows for sockets. Failure rates and repair cost are handled similarly to those for MCM, at a lower after-assembly package failure rate that gives results consistent with typical board failure rates after assembly. This cost-metric is calculated as

$$\begin{aligned} c_{mt} &= \tfrac{(350+0.6A_m)(n_w+2)}{Q_{tm}} + (0.1 + 0.04(n_w + 2))A_m \\ c'_{mt} &= c_{mt} + \tfrac{2000}{v_{mfr}} + 0.05C_aN_p + 0.04P + C_aN_pN_s) + (1 - 0.999^{N_p})(C_p + 5) \end{aligned}$$

where c_{mt} is the cost of the bare PCB and N_s is the number of sockets. Finally, $1.5W$ is added for power supply cost (where W is the total module power); cooling costs are included at the package level.

B.7 Suspense Validation

Table B.3 summarizes the results of Suspense validation against some 3 dozen test cases ranging from individual tiles to complete systems; avg is the net average error, $avg\,|x|$ is the average error, and σ is the standard deviation. Errors were typically around 15%, 3% to 30% for $\pm 1\sigma$, somewhat worse for power. Results were off by over 50% in some instances. Comparing errors at the tile, die, package, and system levels it appears that errors do not accumulate as results from one stage are used as input to the next; additions to tile area (other than pad area), speed, and power by the subsequent models do not dominate. Area and speed estimation are somewhat better for microprocessors, typically 1% to 15% (average $\pm 1\sigma$).

The tile models received the most extensive verification. Memory tiles were verified against register files presented by Mulder in [MQF91], compiled multiport SRAM in [S+91], and Don Speck's DRAM layout [Spe91]. All use standard fabrication rather than custom memory process. In addition to specific example register areas (MIPS-X, DEC3, HP1, and the 80860; the radiation-hardened examples were not used in validation), Mulder presents a simple on-chip memory area model including a scaling factor for additional bit-lines (multiport registers). Using the same memory-cell areas as Mulder, the Suspense memory

	samples	avg	$avg\|x\|$	σ	max	min
Area	32	-8%	14%	12%	51%	0%
Cycle Time	29	0%	14%	12%	57%	1%
Power	17	-4%	23%	15%	47%	6%
Pins	9	4%	17%	11%	38%	1%

Table B.3: Summary of Tile through Package Level Errors

model proved slightly more accurate. Calculated speed was also in line with [S+91]. The CAM was validated against 2 static chips [J+88, O+89] and one dynamic chip [WS89] and showed good agreement. The Suspense cell-area model was compared to numerous layouts [WE88, Hug91, Rob89b]. Average error was 24% with a standard deviation of 17%. This is particularly good in the light of published layouts varying by 30% to 60% for the same functional unit with the same number of transistors (such as a D flip-flop or full-adder); SUSPENS errors were on the order of 90% at this scale. The cell model also compares well with other estimation methods.

For die validation, only information on packaged microprocessor and memory die was available, combining tile, die, and package validation. Verification was limited by the few details publicly available; early low-transistor count microprocessors were nMOS or HMOS. Microprocessor results were evaluated against the 80386, 80486, 68020, and the experimental Bellmac-32A [Bak90]. Verification against commercial SRAMS and DRAMS required fitting λ, the number of sub-arrays, and the cell area simultaneously so the accuracy results are suspect. Detailed information on 3 experimental 16 M bit SRAM chips [S+93a, S+93b, U+93] and 3 experimental 256 M bit memories [T+93, K+93, H+93] allowed accurate validation against advanced memory designs. Suspense's accuracy was surprisingly good considering the numerous circuit 'tricks' used in these experimental memories.

Die validation included two SIMD processor chips (BLITZEN and the MasPar MP-2 [B+90a, Mas92a]), a systolic array (BSYS [Hug91]), a bit-serial CAM (PCAM [Rob89b]) and a CAM with a small 66 transistor bit-serial PE per word ([WS89]). A single SIMD PE is input as a tile (of type parallel if there is on-chip memory, or of type gateArray if there is a negligible registers and no memory) , then multiple PEs were combined on a single die (of type microprocessor). Accuracy was good, with typical (average $\pm 1\sigma$) errors 0% to 36% for area, 1% to 14% for speed, and 11% to 26% for pin count. Average power error was 44%.

Only one validation example was completed through the module (and system) level, due to lack of printed-circuit board information on existing systems. The MasPar MP-2 example was continued from the die and packaging validation. Chip parameters for processors, crossbars, and memory were from Suspense, with only pin count corrected. Neither memory nor the cross-bar influenced board frequency, but their package areas were important. The module was input as a PCB with 4 signal layers and system type as array. Suspense results were 13.6% high for system frequency, 10.3% low for board area, and 15% high for power. Backplane buffers, glue chips, and backplane propagation could easily account for the 11.4 ns difference in system cycle time and 176 cm^2 per board. Adjusted for speed, power error is near 0%. Such accuracy can only be attributed to luck rather than to the model, but does indicate that system-level metrics for parallel processors should be in line with the 5% to 30% errors of the other validations.

Appendix C. 2MISC Simulation Codes

This appendix presents the 2MISC simulation codes for the example algorithms after first describing 2MISC input and timing methods.

C.1 The Simulator

C.1.1 Creating a 2MISC Simulation

This section describes details of the 2MISC simulator "input" as annotation to the simulation codes presented in Sections C.2, C.3, and C.4. A user program is created as a C file which is compiled and linked into the simulator. Each PN is a subroutine, such as `PN0()`, `PN1()`. Code for each PN begins and ends with the C macros `beginPN` and `endPN`. The first macro checks the simulator queue for an appropriate message and opens a C switch and the second closes the switch. Supervisor routines are triggered by receiving a message, using the macro `whenrecieve(messageType, messageValue)` which accepts the message with the earliest arrival time with destination of the enclosing subroutine. This macro uses the message type to form a C case statement, and copies the dequeued message value into the second parameter. The message value can be any structure. A `whenrecieve()` must always start a new routine, and routines are ended with `break`; which closes the case statement.

Associative array operations are programmed by function calls within the supervisor routines. These include Boolean matrix transpose and multiplication, Hamming distance, and collection initialization. Functions to input and output collections are included. Two additional macros, `startCONC` and `stopCONC` indicate concurrent supervisor and array execution. For all code between these macros the maximum of supervisor and array time, rather than their sum, is used to update elapsed time. Doing this automatically would require analysis of data dependencies and incur additional overhead. Other macros allow excluding portions of the simulation, such as input and output, from the elapsed time.

Several datatypes have been pre-defined using C's `typedef` construct for both the supervisor and associative arrays. These include Boolean vectors and matrices, integer tuples, and sets. In addition to declaring a variable, an allocation function specific to each datatype must be called to initialize size and (simulated) location for use by the simulator.

C.1.2 Projecting Execution Times

Header files specify which profiles are to be collected and the cycle-counts for operations. The latter allows approximating the time-behavior of different architectures. Headers also specify the number of PNs and the number of PEs per PN.

Simulated execution times also include the MISC supervisor, assumed to be a commercial standard 500 MIPS RISC processor. Although this may at first seem optimistic, it is in fact a conservative estimate. Intel targets targets 1000 MIPS microprocessors for the year 2000 and DEC has announced a 300 MIPS superscalar RISC that peaks at 1,200 MIPS when all 4 functional units are fully utilized [Rya94]. Over the last twelve years microprocessor MIPS have increased an average of 48% per year. Extrapolating forward and backwards puts commodity microprocessors at 370–1200 MIPS (averaging 700 MIPS) 4 to 5 years from now when a full MISC machine could be manufactured. The example algorithm simulations assume all non-array work is done by the supervisor (some might be done in the array unit), and that there is no overlap between supervisor and array operations.

Array operations are timed at a 60 MHz clock, based on the SUSPENSE early analysis.

C.2 SDM

C.2.1 Simulation Description

As in the basic SDM model, "physical location addresses" are chosen at random, and for the purposes of simulation inputs and targets are random bit vectors for both straight and transposed storage simulations. Timing results are averages over 10 write and 10 read operations to minimize the variations introduced by the random physical locations, and the selected activation radius results in approximately $2N^{1/3}$ locations activated as required for good SDM recall and generalization.

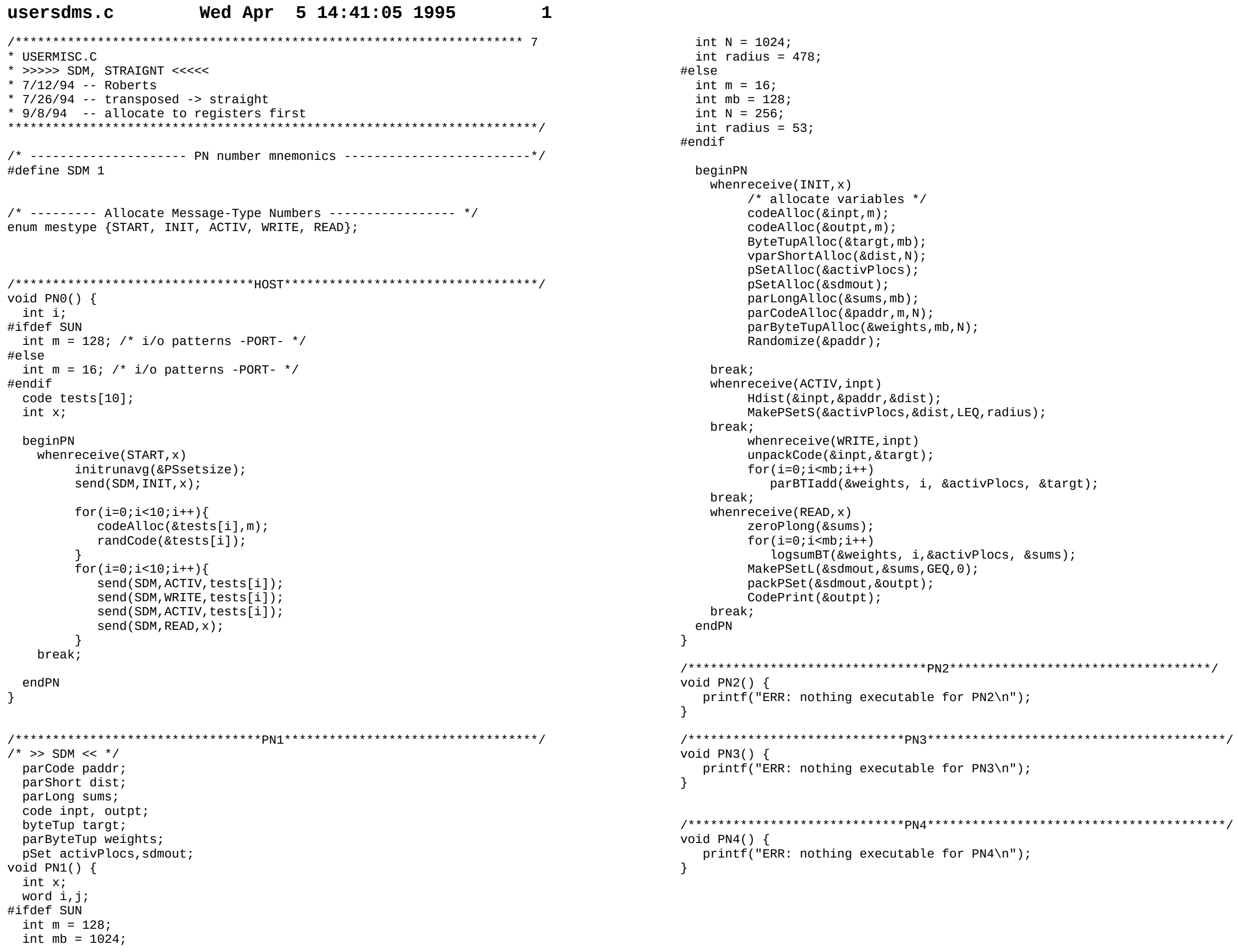
Memory requirements are the major limitation in the SDM simulations. At the target SDM sizes of 1 K input/output bits (m) and 256 K to 1 M physical locations (N), memory is 1/4 to 1 G byte, well above the 32–64 M bytes on the available workstations. As a result, simulations are run for combinations of 256 or 1 K bit input and 1 K, 4 K, 16 K, or 32 K physical locations and the results interpolated to the target sizes.

Global Communication

Extrapolated to the target size (1 Kbit input/output and 256 K storage locations), execution time averaged for one read and one write is 0.5 ms. With 128 bytes of input and output, this gives 0.5 Mbyte/s PN external communication

SEGUL and the 2MISC Code

The **when receive** and **send** operations are nearly identical to SEGUL, the SEGUL syntax replaced by the C function syntax. Hamming distance is invoked in nearly identical ways in both codes, and the general set notations in SEGUL correspond directly to various specific make-set C functions. Other SEGUL collection operations also have direct simulator equivalences. As discussed in Section 5.1, the most natural description of the SDM algorithm gives a "straight-storage" weight representation whereas "transposed storage" is much faster. Otherwise there is a relatively direct transformation of SEGUL pseudocode into the simulator's C function syntax. Figure C.1 presents the 2MISC code for straight storage, and Figure C.2 presents the code for transposed storage.



usersdms.c   Wed Apr 5 14:41:05 1995   1

```
/******************************************************************** 7
* USERMISC.C
* >>>>> SDM, STRAIGNT <<<<<
* 7/12/94 -- Roberts
* 7/26/94 -- transposed -> straight
* 9/8/94  -- allocate to registers first
***********************************************************************/

/* --------------------- PN number mnemonics --------------------------*/
#define SDM 1


/* --------- Allocate Message-Type Numbers ----------------- */
enum mestype {START, INIT, ACTIV, WRITE, READ};



/********************************HOST**********************************/
void PN0() {
  int i;
#ifdef SUN
  int m = 128; /* i/o patterns -PORT- */
#else
  int m = 16; /* i/o patterns -PORT- */
#endif
  code tests[10];
  int x;

  beginPN
    whenreceive(START,x)
         initrunavg(&PSsetsize);
         send(SDM,INIT,x);

         for(i=0;i<10;i++){
            codeAlloc(&tests[i],m);
            randCode(&tests[i]);
         }
         for(i=0;i<10;i++){
            send(SDM,ACTIV,tests[i]);
            send(SDM,WRITE,tests[i]);
            send(SDM,ACTIV,tests[i]);
            send(SDM,READ,x);
         }
    break;

  endPN
}

/*********************************PN1**********************************/
/* >> SDM << */
  parCode paddr;
  parShort dist;
  parLong sums;
  code inpt, outpt;
  byteTup targt;
  parByteTup weights;
  pSet activPlocs,sdmout;
void PN1() {
  int x;
  word i,j;
#ifdef SUN
  int m = 128;
  int mb = 1024;
  int N = 1024;
  int radius = 478;
#else
  int m = 16;
  int mb = 128;
  int N = 256;
  int radius = 53;
#endif

  beginPN
    whenreceive(INIT,x)
         /* allocate variables */
         codeAlloc(&inpt,m);
         codeAlloc(&outpt,m);
         ByteTupAlloc(&targt,mb);
         vparShortAlloc(&dist,N);
         pSetAlloc(&activPlocs);
         pSetAlloc(&sdmout);
         parLongAlloc(&sums,mb);
         parCodeAlloc(&paddr,m,N);
         parByteTupAlloc(&weights,mb,N);
         Randomize(&paddr);

    break;
    whenreceive(ACTIV,inpt)
         Hdist(&inpt,&paddr,&dist);
         MakePSetS(&activPlocs,&dist,LEQ,radius);
    break;
         whenreceive(WRITE,inpt)
         unpackCode(&inpt,&targt);
         for(i=0;i<mb;i++)
            parBTIadd(&weights, i, &activPlocs, &targt);
    break;
    whenreceive(READ,x)
         zeroPlong(&sums);
         for(i=0;i<mb;i++)
            logsumBT(&weights, i,&activPlocs, &sums);
         MakePSetL(&sdmout,&sums,GEQ,0);
         packPSet(&sdmout,&outpt);
         CodePrint(&outpt);
    break;
  endPN
}

/********************************PN2***********************************/
void PN2() {
   printf("ERR: nothing executable for PN2\n");
}

/*****************************PN3*****************************************/
void PN3() {
   printf("ERR: nothing executable for PN3\n");
}


/*****************************PN4*****************************************/
void PN4() {
   printf("ERR: nothing executable for PN4\n");
}
```

Figure C.1: The Straight-Storage SDM 2MISC Code

```
/*********************************************************** 7
* USERMISC.c
* >>>>> SDM, TRANSPOSED <<<<<
* 7/12/94 -- Roberts
* 7/22/94 -- sums made a long, unpacking code in SDM server PN
* 7/27/94 -- transposed routines for code pack and unpack
* 9/8/94  -- allocate to registers first
* 9/15/94 -- ifdef VERBOSE, number of active plocs printed
***********************************************************************/

/* -------------------- PN number mnemonics -------------------------*/
#define SDM 1


/* --------- Allocate Message-Type Numbers ----------------- */
enum mestype {START, INIT, ACTIV, WRITE, READ};



/********************************HOST**********************************/
void PN0() {
  int i;
#ifdef SUN
  int m = 128; /* i/o patterns  -PORT- */
#else
  int m = 16; /* i/o patterns  -PORT- */
#endif
  code tests[10];
  int x;

  beginPN
    whenreceive(START,x)
         initrunavg(&PSsetsize);
         send(SDM,INIT,x);

         for(i=0;i<10;i++){
            codeAlloc(&tests[i],m);
            randCode(&tests[i]);
         }
         for(i=0;i<10;i++){
            send(SDM,ACTIV,tests[i]);
            send(SDM,WRITE,tests[i]);
            send(SDM,ACTIV,tests[i]);
            send(SDM,READ,x);
         }
    break;

  endPN
}


/*********************************PN1**********************************/
/* >> SDM << */
  parCode paddr;
  parShort dist;
  parLong sums;
  code inpt, outpt;
  byteTup targt;
  parByteTup weights;
  pSet activPlocs,sdmout;
void PN1() {
  int x;
  word i,j;
#ifdef SUN
  int m = 128;
  int mb = 1024;
  int N = 1024;
  int radius = 478;
#else
  int m = 16;
  int mb = 128;
  int N = 256;
  int radius = 50;
#endif

  beginPN
    whenreceive(INIT,x)
         /* allocate variables */
         ByteTupAlloc(&targt,mb);
         codeAlloc(&inpt,m);
         codeAlloc(&outpt,m);
         vparShortAlloc(&dist,N);
         pSetAlloc(&activPlocs);
         pSetAlloc(&sdmout);
         parLongAlloc(&sums,mb);
         parCodeAlloc(&paddr,m,N);
         parByteTupAlloc(&weights,mb,N);
         Randomize(&paddr);

    break;
    whenreceive(ACTIV,inpt)
         Hdist(&inpt,&paddr,&dist);
         MakePSetS(&activPlocs,&dist,LEQ,radius);
    break;
      whenreceive(WRITE,inpt)
      unpackCodeT(&inpt,&targt);
         i = nextPset(&activPlocs);
         while( i  != EMPTY){
              parBTadd(&weights, i, &targt);
              i = nextPset(&activPlocs);
        }
    break;
    whenreceive(READ,x)
         zeroPlong(&sums);
         i = nextPset(&activPlocs);
         while( i  != EMPTY){
              parLsum(&sums, &weights, i);
              i = nextPset(&activPlocs);
         }
         MakePSetL(&sdmout,&sums,GEQ,0);
         packPSetT(&sdmout,&outpt);
         CodePrint(&outpt);
    break;
  endPN
}

/********************************PN2**********************************/
void PN2() {
   printf("ERR: nothing executable for PN2\n");
}

/*****************************PN3****************************************/
void PN3() {
   printf("ERR: nothing executable for PN3\n");
}


/*****************************PN4****************************************/
```

```
void PN4() {
   printf("ERR: nothing executable for PN4\n");
}
```

Figure C.2: The Transposed-Storage SDM 2MISC Code

C.3 Semantic Network

C.3.1 Simulation Description

The SN simulations are for a minimal semantic network model, similar to that used by Miranker [MA90], with operation approximated by subsumption node matching, a probability that an outgoing edge will be activated, and each query proceeding through a number of propagation cycles bounded by the network diameter. In keeping with the manual profile analysis, results are based on extrapolation to the target example size in which a 256 K node SN is built up through individual node insertions and deletions, followed by 32 K queries. Each query consists of initially marking 10 nodes, followed by at most $\log_A N$ marker propagation cycles, and a single result-set matching. Numerous runs with 0, 100, 500, and 1,000 queries and 8 K, 16 K, 32 K, and 64 K nodes provide the basis for extrapolation to the target application size. Here we are limited by execution time rather than memory on the simulation platform. Extrapolation also includes the average number of markers as determined by the analytical model.

For SNs, parameters and other simulation characteristics for the synthetic data set and queries are important as they affect performance through the number of markers, the number of propagation cycles, and the call frequencies to the various routines (createNode, addEdge, reset, search, propagate, described in Section 5.2). In the simulation, marker propagation is determined by four parameterized operations: initial node marking, outgoing edge (marker) activation, marker pass-through, and number of propagation cycles. The node representation described in Section 5.2 is used in conjunction with mesh-network communication, since on average significantly more than 40 markers will be propagating each cycle. Node labels are taken to be a concatenation of the PE number and memory location where the node is stored plus inheritance subsumption codes. The simulation begins with marking the initial nodes via subsumption test. Execution time and instruction count are the same as matching three 32-bit fields, so that simulation results are also applicable to SNs without the taxonomy coding, within the approximation of a fully random network topology. Each marked node then sends its own marker identifier. To approximate link-type and node-state restrictions, each outgoing edge of a node is activated with a constant probability ρ. When a node receives one or more markers in a propagation cycle, these markers are ored, saved at the node, and then sent over outgoing edges in the next propagation cycles. Once a node has sent, it does not send again but only ors incoming markers with its marker state. The simulation iterates marker propagation cycles until the network diameter has been reached or there are no markers sent in the prior cycle, after which a node result set is determined by matching the presence and absence of randomly specified markers.

Using subsumption codes for inheritance operations removes most if not all of the tree-like sections of the SN, so that a randomly generated disconnected graph is a good approximation to the network topology. For the purposes of evaluating the MISC architecture, the network topology itself is not crucial so long as number and pattern of marker propagations approximates that of the application. The 2MISC code assumes that a given PE will have no more than 1 node sending. A probabilistic analysis shows that the probability of one or more PEs having more than one sending node is slightly less than 3%. The simulation allows a node to receive any number of markers.

Global Communication

Although a highly simplified SN model, the extrapolated execution time is a reasonable basis for estimating PN external communication. Adding a node requires about 4 bytes to specify the node label and operations to be performed, and a node edge requires about 12 bytes to specify source, destination, and the operation. At 28 μs to insert a node with 4 edges, communication is 0.7 MBytes/s. Marker propagation is actually more like a short program. Assuming 120 bytes to specify the 10 initial node markings, 160 bytes to specify propagation rules, and 120 bytes to specify Boolean and math operations (based loosely on the SNAP instruction set [M^{+}92]) a total of 400 bytes are needed per query. At 530 queries per second, this gives 0.2 Mbyte/s external communication.

SEGUL and the Simulation Code

The 2MISC simulation is a modification of the basic algorithm as expressed in SEGUL (Section 5.2) and there is considerable additional code for managing the simulation experiments in the 2MISC code). Modifications include representing activation sets as a field in the node data structure rather than as independent sets. The latter was largely for convenience in writing the simulation. The original SEGUL pseudocode may be more efficient in this instance. Simulation data structure declarations correspond directly to the SEGUL pseudocode as with SDM, and the SEGUL `createnode` routine corresponds to the simulation's `addtuple` function. A simulator function `updatetuple` is used instead of the pseudocode's set operations for adding edges, resetting the SN, marking nodes, and propagating markers. This has performance advantages from avoiding recreating sets or associating them with multiple data objects, and could be expressed in SEGUL using the map (tuple) modification syntax. Although these operations aren't divided into functions, the comments in the simulator code shows how this would be done. As with SDM, the simulation is succinctly described in SEGUL, but consideration of implementation details results in a slightly different high-level expression from the original algorithm description. Figure C.3 presents the first part of the 2MISC code, the SN procedures, and Figure C.4 continues the code with a sample query.

```
usersn1         Tue Apr 18 19:50:07 1995         1

/******************************************************************** 7
* USERMISC.C
* Minimal Semantic Network
* 9/1/94 -- Roberts
* 9/22/94 -- max path log_base avg arity
***********************************************************************/


/* -------------------- PN number mnemonics -------------------------*/
#define OTHERPN 1



/* --------- Allocate Message-Type Numbers ----------------- */
enum mestype {START,DUMMY};



/********************************HOST**********************************/
int maxArity=6, nNodes=8192;
int lowArity=2, highArity=6, firstmark=10, propprob=35;
int query; int numquery = 0;
double logbase;
int numpcycles;
#define nodeLabel 0
#define nodeProc 1
#define nodeMarker 2
#define nodeEdges 3
#define wave    0x80000000
#define hassent 0x40000000
parCodeTup G;
word numnodes=0;

/* ---------------------- subroutines -------------------------- */




void printSN(parCodeTup *G, int first, int last){
  int i,j;
  STOPTIME
  STOPSIMT
  printf("     Node   Proc     Mark   OutEdges...\n");
  for(i=first;i<last;i++){
     printf("%6ld %6lx %8lx ",G->value[i][nodeLabel],G->value[i][nodeProc],
            G->value[i][nodeMarker]);
     j=nodeEdges;
     while((j<G->nelems) && (G->value[i][j]!=INFINITY)){
           printf("%6ld ",G->value[i][j]);
           j++;
     }
     printf("\n");
  }
  STARTTIME
  STARTSIMT
}

int propagate(parCodeTup *G, acode domarker, int prob){
    int i,j,nummarkers, maxoutedge, thismarkers;
    acode dest,temp;
    STOPTIME
    nummarkers=0; maxoutedge =0;
    for(i=0;i<numnodes;i++){
       thismarkers = 0;
```

```
     /** form the active wave **/
     /* andPsetT(activ,G,posToMatch,valToMatch=wave|~hassent|mark,mask=xxxxx);  */
       if ( ((G->value[i][nodeMarker]&wave) !=0) &&
            ((G->value[i][nodeMarker]&hassent) ==0) &&
         ( (domarker==0) || ((G->value[i][nodeMarker]&domarker) !=0) ) ){
          /** send edge if its there with given probability **/
          for(j=nodeEdges;j<nodeEdges+maxArity;j++){
             dest = G->value[i][j];
             if( (dest != INFINITY) && (randomI()%100 < prob) ){
             /* PROUTE(activ,G,sendfield,destaddr,OR,destfield,pset recieve) */
                nummarkers++; thismarkers++;
                G->value[dest][nodeProc] =
                    G->value[dest][nodeProc] | G->value[i][nodeMarker];
             }
          }
          maxoutedge = MAX(maxoutedge,thismarkers);
          /** update senders **/
          /* updatePsetT(activ,G,marker,~wave|hassent,xxx) */
            G->value[i][nodeMarker] = G->value[i][nodeMarker] & ~wave;
            G->value[i][nodeMarker] = G->value[i][nodeMarker] | hassent;
       } /* if active node */
    } /* for each node created */
    /** update receivers **/
    /* updatePsetT(receive,G,marker,G.Proc,xxxxx) */
    /* updatePsetT(receive,G,proc,0,xxxxx) */
      for(i=0;i<numnodes;i++){
         G->value[i][nodeMarker] =
            G->value[i][nodeMarker] | G->value[i][nodeProc];
         G->value[i][nodeProc] = 0;
      }
    /* activ=receive */
    STARTTIME
    if(nummarkers>0) markerPass(G,nummarkers,4,maxoutedge);
    return(nummarkers);
}

void printSNResult(parCodeTup *G, int tosend, int didsend, acode markerz){
   int i,numres,responder;
     /* makePsetT(ans,G,posToMatch,valToMatch=wave,EXCLUDE)  */
     /* andPsetT(ans,G,posToMatch,valToMatch=hassent,EXCLUDE)  */
     /* andPsetT(ans,G,posToMatch,valToMatch=domarker,INCLUDE)  */
     STOPTIME
     numres = 0;
     for(i=0;i<numnodes;i++){
      responder = TRUE;
      if ( (tosend == 1) && ((G->value[i][nodeMarker]&wave)==0) )
         responder = FALSE;
      if ( (didsend == 1) && ((G->value[i][nodeMarker]&hassent)==0) )
         responder = FALSE;
      if ( (markerz != 0x0) && ((G->value[i][nodeMarker]&markerz)==0) )
         responder = FALSE;
      if (responder == TRUE){
         numres++;
         /* printPsetT(ans) */
/*       printf("Node %ld meets criteria\n",G->value[i][nodeLabel]); */
      }
   }
#ifdef VERBOSE
   printf("%d nodes meet requirements\n",numres);
#endif
   STARTTIME
   matchCodeTuple(G,numres);
}
/* ------------------------------------------------------------- */
```

Figure C.3: The Semantic Network 2MISC Code, Procedures

```
usersn2        Tue Apr 18 19:50:34 1995        1

void PN0() {
int x;
  int i,j,temp,nout;
  word temp2;
  int ts,ds;
  acode mk;
  floatavg nummarkers,allmarkers;
  beginPN
    whenreceive(START,x)
/*       parCodeTupAlloc(&G,nodeEdges+maxArity,nNodes); */
       parCodeTupAlloc(&G,32,nNodes);
      srand(3);
       /* create random nodes and edges */
       for(i=0;i<nNodes;i++){
          /* create a node */
          addHTuple(&G, &numnodes, (acode) i, nodeEdges);
          temp = highArity-lowArity;
          nout = randomI()%temp + lowArity;
          for(j=0;j<nout;j++){
             temp2 = randomW()%nNodes;
             /* add an edge */
             appendHTuple(&G, nodeLabel, (acode) i, (acode) temp2, numnodes );
          }
       }
       /* DO QUERIES */
       STOPTIME
       STOPSIMT
       logbase=(double)(highArity+lowArity)/2;
       numpcycles = log((double)nNodes)/log(logbase);
       initrunavg(&allmarkers);
       STARTTIME
       STARTSIMT
    for(query=0;query<numquery;query++){
       /* reset the SN */
       updateTup(&G,nodeLabel,INFINITY,nodeProc,0,numnodes);
       updateTup(&G,nodeLabel,INFINITY,nodeMarker,0,numnodes);
       /* mark random nodes */
       for(i=0;i<firstmark;i++)
          updateTup(&G, nodeLabel, (acode) randomW()%numnodes,
                    nodeMarker, ((acode)pow(2,(double)i) | wave), numnodes);
       /* propagate */
       j=1; i=0;
       initrunavg(&nummarkers);
       while((j!=0)&&(i<numpcycles)){
          i++;
          j=propagate(&G, 0x0, propprob);
          runavg(&nummarkers, (float) j);
#ifdef VERBOSE
          STOPTIME
          STOPSIMT
          printf("Prop cycle %d: %d markers, %g cumulative avg markers\n",
                  i,j,nummarkers.runavg);
          STARTTIME
          STARTSIMT
#endif
       }
       /* query responders */
       printSNResult(&G,FALSE,FALSE,0x3);
       runavg(&allmarkers,nummarkers.runavg);
  } /* for each query */
  printf("Overall average markers %g\n",allmarkers.runavg);
    break;
  endPN
}
```

Figure C.4: The Semantic Network 2MISC Code Continued, A Query

C.4 Subgraph Isomorphism: Refinement

C.4.1 Simulation Description

Only the form M matrix and refinement portions of the graph knowledge base example application have been implemented in the simulator, and no effort has been made to express the code for forming the M matrix in terms of pseudocode as it is generated randomly for the simulation. Refinement corresponds directly to the pseudocode in Section 5.3.2.

Each graph requires a minimum of $\lceil |G_a|^2/8 \rceil$ bytes to store the adjacency matrix plus about $8|G_a|$ bytes for subsumption code and node arity information, so that the experiment KB graphs range from 72 to 294 bytes of key information (a typical KB would associate other information with each graph). Once the selected graphs are distributed to one node per PE, the required memory easily fits on-chip, requiring 8 bytes for code and node information (discardable after the M matrix is formed) plus 6 bytes (for a 16 bit PE) for the A, B, and M matrices.

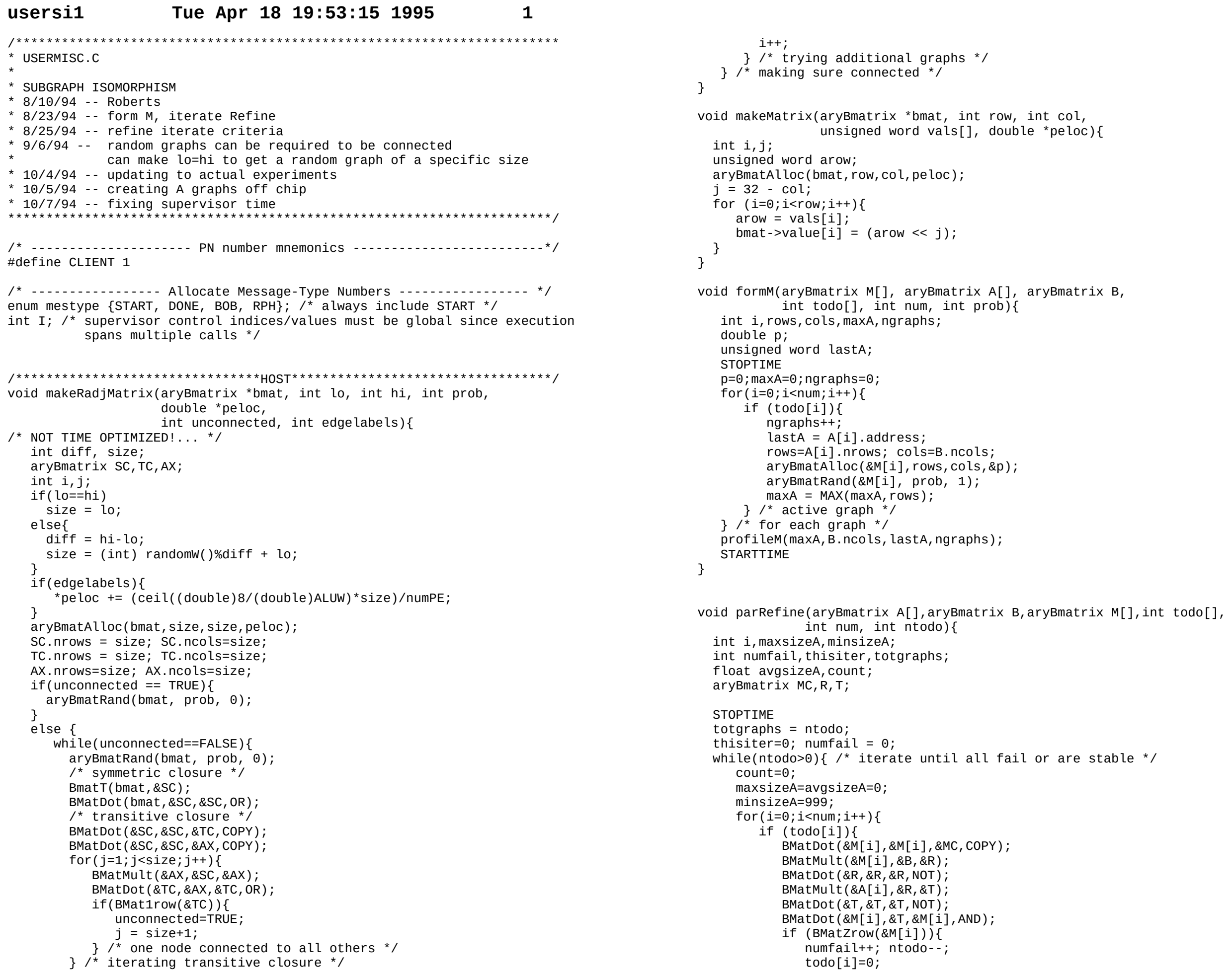
Instruction profiles from the simulations are averages over 4 queries for each experiment; variation between experiments is only a few percentage points. Extrapolation to larger sizes is unnecessary as we are directly simulating one group of graph comparisons. Each query in the simulation distributes the graphs and forms the M matrix then iterates the refinement algorithm until all graphs either fail or the M matrix no longer changes.

Global Communication

Since an entire KB can now be processed on a single PN without a central concept-type hierarchy, global communication plummets to that required to express the query graph and result notification. The number of bytes to express the query graph G_b is the same as the memory requirements for storing G_a. Assuming this same number of bytes to return results and the above estimates of queries per second, global communication is only about 0.05 M bytes/s. Even allowing for the return of 32-bit identifiers for a thousand generalizations and specializations, global communication is slightly less than 1 M byte/s. If the results are processed by the same PN there is no external communication for result notification.

SEGUL and the Simulation Code

The bulk of the 2MISC refinement code is for managing the simulation experiments. Again, simulation data declarations correspond directly to those of the pseudocode. The refinement step is implemented using the Boolean matrix operations described in Section 5.3.2. The matrix dot-operations correspond directly to the SEGUL elementwise operations on tuples of codes; matrix multiplication needs to be incorporated into the SEGUL notation. The simulation includes functions corresponding to `[code(n)...] = [code(n)...]` (matrix compare) and ∃x s.t. tupleY(x) = 0 (at least 1 row of a matrix all zeros). As with the other codes, the simulation is succinctly describable in SEGUL but different types of description (set versus matrix operations) have different low-level implementations. Figure C.5 and C.6 present the 2MISC code for subgraph isomorphism refinement. There is no simulation of the supervisor, so that here `PN0()` is the MISC SIMD array.



usersi1   Tue Apr 18 19:53:15 1995   1

```
/***********************************************************************
* USERMISC.C
*
* SUBGRAPH ISOMORPHISM
* 8/10/94 -- Roberts
* 8/23/94 -- form M, iterate Refine
* 8/25/94 -- refine iterate criteria
* 9/6/94 --  random graphs can be required to be connected
*            can make lo=hi to get a random graph of a specific size
* 10/4/94 -- updating to actual experiments
* 10/5/94 -- creating A graphs off chip
* 10/7/94 -- fixing supervisor time
***********************************************************************/

/* -------------------- PN number mnemonics -------------------------*/
#define CLIENT 1

/* ----------------- Allocate Message-Type Numbers ----------------- */
enum mestype {START, DONE, BOB, RPH}; /* always include START */
int I; /* supervisor control indices/values must be global since execution
          spans multiple calls */


/********************************HOST**********************************/
void makeRadjMatrix(aryBmatrix *bmat, int lo, int hi, int prob,
                    double *peloc,
                    int unconnected, int edgelabels){
/* NOT TIME OPTIMIZED!... */
   int diff, size;
   aryBmatrix SC,TC,AX;
   int i,j;
   if(lo==hi)
     size = lo;
   else{
     diff = hi-lo;
     size = (int) randomW()%diff + lo;
   }
   if(edgelabels){
      *peloc += (ceil((double)8/(double)ALUW)*size)/numPE;
   }
   aryBmatAlloc(bmat,size,size,peloc);
   SC.nrows = size; SC.ncols=size;
   TC.nrows = size; TC.ncols=size;
   AX.nrows=size; AX.ncols=size;
   if(unconnected == TRUE){
     aryBmatRand(bmat, prob, 0);
   }
   else {
      while(unconnected==FALSE){
        aryBmatRand(bmat, prob, 0);
        /* symmetric closure */
        BmatT(bmat,&SC);
        BMatDot(bmat,&SC,&SC,OR);
        /* transitive closure */
        BMatDot(&SC,&SC,&TC,COPY);
        BMatDot(&SC,&SC,&AX,COPY);
        for(j=1;j<size;j++){
           BMatMult(&AX,&SC,&AX);
           BMatDot(&TC,&AX,&TC,OR);
           if(BMat1row(&TC)){
              unconnected=TRUE;
              j = size+1;
           } /* one node connected to all others */
        } /* iterating transitive closure */
```

```
        i++;
      } /* trying additional graphs */
   } /* making sure connected */
}

void makeMatrix(aryBmatrix *bmat, int row, int col,
                unsigned word vals[], double *peloc){
  int i,j;
  unsigned word arow;
  aryBmatAlloc(bmat,row,col,peloc);
  j = 32 - col;
  for (i=0;i<row;i++){
     arow = vals[i];
     bmat->value[i] = (arow << j);
  }
}

void formM(aryBmatrix M[], aryBmatrix A[], aryBmatrix B,
           int todo[], int num, int prob){
   int i,rows,cols,maxA,ngraphs;
   double p;
   unsigned word lastA;
   STOPTIME
   p=0;maxA=0;ngraphs=0;
   for(i=0;i<num;i++){
      if (todo[i]){
         ngraphs++;
         lastA = A[i].address;
         rows=A[i].nrows; cols=B.ncols;
         aryBmatAlloc(&M[i],rows,cols,&p);
         aryBmatRand(&M[i], prob, 1);
         maxA = MAX(maxA,rows);
      } /* active graph */
   } /* for each graph */
   profileM(maxA,B.ncols,lastA,ngraphs);
   STARTTIME
}


void parRefine(aryBmatrix A[],aryBmatrix B,aryBmatrix M[],int todo[],
               int num, int ntodo){
  int i,maxsizeA,minsizeA;
  int numfail,thisiter,totgraphs;
  float avgsizeA,count;
  aryBmatrix MC,R,T;

  STOPTIME
  totgraphs = ntodo;
  thisiter=0; numfail = 0;
  while(ntodo>0){ /* iterate until all fail or are stable */
     count=0;
     maxsizeA=avgsizeA=0;
     minsizeA=999;
     for(i=0;i<num;i++){
        if (todo[i]){
           BMatDot(&M[i],&M[i],&MC,COPY);
           BMatMult(&M[i],&B,&R);
           BMatDot(&R,&R,&R,NOT);
           BMatMult(&A[i],&R,&T);
           BMatDot(&T,&T,&T,NOT);
           BMatDot(&M[i],&T,&M[i],AND);
           if (BMatZrow(&M[i])){
              numfail++; ntodo--;
              todo[i]=0;
```

Figure C.5: The Refinement 2MISC Code

```
usersi2         Tue Apr 18 19:53:50 1995        1
           }
           if (BMatComp(&MC,&M[i])){
              ntodo--;
              todo[i]=0;
           }
           maxsizeA = MAX(maxsizeA,A[i].nrows);
           minsizeA = MIN(minsizeA,A[i].nrows);
           avgsizeA = count/(count+1)*avgsizeA
                      + A[i].nrows/(count+1);
           count++;
        } /* if graph processed */
     } /* for each graph */
     if (count>0) profileRefine(maxsizeA,B.nrows);
#ifdef VERBOSE
     printf("Iter %d: %d cum. fail, %d cum. stbl, %d left\n",
             thisiter, numfail, totgraphs-numfail-ntodo, ntodo);
     /*
     printf("B = %d, maxA = %d, minA = %d, avgA = %g\n",
             B.nrows,maxsizeA,minsizeA,avgsizeA);
     */
#endif
     thisiter++;
  } /* iterate until all fail or stable */
#ifndef VERBOSE
     printf("Iter %d: %d cum. fail, %d cum. stbl, %d left\n",
             thisiter-1, numfail, totgraphs-numfail-ntodo, ntodo);
#else
     printf("\n");
#endif
  STARTTIME
}

/* ---------------------------------------------------------------- */
#define NQUERIES 4  /* number of random queries to do */
#define NGRAPHS 8192 /* total number of graphs in KB */
#define SGRAPH  24  /* minimum number of nodes in a graph */
#define LGRAPH  26 /* maximum nodes in A graph */
#define SGRAPHB 30  /* min nodes in B */
#define LGRAPHB 32  /* max nodes in B */
#define ADENS   20  /* % probability an edge in A adj matrix exists */
#define BDENS   20  /* as above, for B matrices */
#define MDENS   10 /* as above, for M matrices */

  aryBmatrix A[NGRAPHS],B,M[NGRAPHS],C;
  int doit[NGRAPHS];

void PN0() {
  int a,b,x,i,j, notdone, temp, nsimult,iters;
  double peloc;
  int xx;
  int numtodo, graphno;
/*  unsigned word thevals[32]; for makeMatrix */

  beginPN
    whenreceive(START,x)
         STOPTIME
         STOPSIMT
         /* create the ‘KB’ of graphs */
         peloc = 1024; /* OFF CHIP */
         for (i=0; i<NGRAPHS; i++){
              makeRadjMatrix(&A[i], SGRAPH, LGRAPH, ADENS, &peloc,
                            FALSE,TRUE);
         }
         STARTTIME
         STARTSIMT
         /***** do the queries *****/
         for(i=0;i<NQUERIES;i++){
             STOPTIME
             STOPSIMT
             /* create the ‘query’ graph */
             peloc = 0; /* MEM ALLOC! */
             makeRadjMatrix(&B, SGRAPHB, LGRAPHB, BDENS, &peloc, TRUE, FALSE);
             /* determine number simult */
             temp = B.nrows * (int) ceil((double)B.ncols/(8*ALUW));
             nsimult = (int) floor((double) numPE/temp);
             /** form doit vector **/
             graphno = nsimult;
             for(j=0;j<NGRAPHS;j++) doit[j] = 0;
             while(graphno>0){
                 xx = randomI()%NGRAPHS;
                 if(doit[xx] == 0){
                     doit[xx] = 1;
                     graphno--;
                 }
             }
             STARTTIME
             STARTSIMT
             /*** expand As and create Ms ***/
             formM(M,A,B,doit,NGRAPHS,MDENS);
             /*** REFINE ***/
             parRefine(A,B,M,doit,NGRAPHS,nsimult);
       } /* each query */

    break;

    whenreceive(DONE,x)
       printf("Done! Result = %d\n",x);
    break;

  endPN

}
```

Figure C.6: The Refinement 2MISC Code, Continued

References

[A$^+$87] Judy M. Anderson et al. The architecture of FAIM-1. *Computer*, pages 55–65, January 1987.

[A$^+$90] A. Agarwal et al. April: A processor architecture for multiprocessing. In *17th Int. Sym. on Computer Architectures*, pages 104–114, 1990.

[A$^+$93a] Anant Agarwal et al. Sparcle: An evolutionary processor design for large-scale multiprocessors. *IEEE Micro*, 13(3):48–61, June 1993.

[A$^+$93b] Robert Allen et al. A parallel algorithm for graph matching and its MasPar implementation. In *Computer Architectures for Machine Perception*, pages 13–18. IEEE Computer Society Press, 1993.

[AAA93] AAAI. Spring symposium series: Innovative applications of massive parallelism. Stanford University, March 1993.

[AB91] Selim Akyokus and P. Bruce Berra. Optical content-addressable memories for data/knowledge base processing. In V.K. Prasanna, editor, *Proc. 5th Int. Parallel Processing Symposium*, pages 202–207. IEEE Computer Society Press, 1991.

[AB93] Ed P. Andert Jr. and Thomas Bartolac. Parallel neural network training. In James Geller, editor, *Innovative Applications of Massive Parallelism*, pages 7–13. AAAI, 1993. Working Notes, Spring Symposium Series, Stanford University.

[ACP93] Jean-Marc Andreoli, Paolo Ciancarini, and Remo Pareschi. Parallel searching with multisets-as-agents. In James Geller, editor, *Innovative Applications of Massive Parallelism*, pages 14–25. AAAI, 1993. Working Notes, Spring Symposium Series, Stanford University.

[Adv88a] Advanced Micro Devices. *Am95C85 Content-Addressable Data Manager Product Description*, 1988.

[Adv88b] Advanced Micro Devices. *Am95C85 Content-Addressable Data Manager Technical Manual*, 1988.

[AG89] George S. Almasi and Allan Gottlieb. *Highly Parallel Computing.* Benjamin/Cummings Publishing Co. Inc., Redwood City, CA, 1989.

[AG93] Tito Autrey and Herbert Gelernter. Parallel heuristic search. In James Geller, editor, *Innovative Applications of Massive Parallelism*, pages 26–31. AAAI, 1993. Working Notes, Spring Symposium Series, Stanford University.

[AHEK94] William A. Anderson, James A. Hendler, Matthew Evett, and Brian Kettler. Massively parallel matching of knowledge structures. In Hiroaki Kitano and James A. Hendler, editors, *Massively Parallel Artificial Intelligence*, pages 52–73, Menlo Park, CA, 1994. AAAI Press / The MIT Press.

[AK$^+$89] Hassan Ait-Kaci et al. Efficient implementation of lattice operations. *ACM Transactions on Programming Languages and Systems*, 11(1):115–146, January 1989.

[AL93] J. Amenta and R. A. Levinson. Morph, an experience-based adaptive chess system: A demonstration report. *International Computer Chess Association Journal*, 16(1):51–53, March 1993.

[Alb91] James S. Albus. Outline for a theory of intelligence. *IEEE Transactions on Systems, Man, and Cybernetics*, 21(3):473–509, May/June 1991.

[AS83] Gregory R. Andrews and Fred B. Schneider. Concepts and notations for concurrent computing. *Computing Surveys*, 15(1):3–43, March 1983.

[ASHH88] A. Agarwal, R. Simoni, J. Hennessy, and M. Horowitz. An evaluation of directory schemes for cache coherence. In *15th International Symposium on Computer Architecture*, pages 280–289, 1988.

[Ass94] Ulrich Assmann. A model for parallel deduction. In V. Kumar et al., editors, *Parallel Processing for Artificial Intelligence*, pages 145–162, New York, 1994. North-Holland. volume 2.

[B+90a] D. W. Blevins et al. BLITZEN: A highly integrated massively parallel machine. *J. Parallel Distributed Computing*, 8:150–160, February 1990.

[B+90b] S. Bokar et al. Supporting systolic and memory communication in iWarp. In *17th Int. Sym. on Computer Architectures*, pages 70–81, 1990.

[B+94] Wolfgang O. Budde et al. An asynchronous high-speed packet switching component. *IEEE Design and Test of Computers*, pages 33–42, Summer 1994.

[Bak90] H. B. Bakoglu. *Circuits, Interconnections, and Packaging for VLSI.* Addison-Wesley, Reading, MA, 1990.

[Bar88] J. M. Barnard. Problems of substructure search and their solution. In Wendy Warr, editor, *Chemical Structures: The International Language of Chemistry.* Springer-Verlag, New York, 1988.

[Bat82] Kenneth E. Batcher. Bit-serial parallel processing systems. *IEEE Transactions on Computers*, C-31(5):377–384, May 1982.

[Bat86] Kenneth E. Batcher. The architecture of tomorrow's massively parallel computer. In *Proc. 1st Symp. on the Frontiers of Massively Parallel Scientific Computation*, pages 151–157, 1986.

[BCM88] J. P. Banatre, A. Coutant, and D. Le Metayer. A parallel machine for multiset transformation and its programming style. *Future Generations Computer Systems*, 4:133–144, 1988.

[BF82] Avron Barr and Edward A. Feigenbaum. *The Handbook of Artificial Intelligence.* HeurisTech Press, Stanford, CA, 1982.

[BHN+92] Franz Baader, Bernhard Hollunder, Bernhard Nebel, Hans-Jurgen Profitlich, and Enrico Franconi. An empirical analysis of optimization techniques for terminological representation systems. In *Proceedings of the 3rd International Conference on Principles of Knowledge Representation and Reasoning*, Cambridge, MA, October 1992.

[BKS93] Jon Bright, Simon Kasif, and Lewis Stiller. Exploiting algebraic structure in parallel state-space search (extended abstract). In James Geller, editor, *Innovative Applications of Massive Parallelism*, pages 32–39. AAAI, 1993. Working Notes, Spring Symposium Series, Stanford University.

[Ble90] Guy E. Blelloch. *Vector Models for Data-Parallel Computing.* MIT Press, Cambridge, MA, 1990.

[BM90] Jean-Pierre Banatre and Daniel Le Metayer. The Gamma model and its discipline of programming. *Science of Computer Programming*, 15:55–77, 1990.

[BM93] Jean-Pierre Banatre and Daniel Le Metayer. Programming by multiset transformation. *Communications of the ACM*, 36(1):98–111, January 1993.

[Bon92] David Bondurant. Enhanced dynamic ram. *IEEE Spectrum*, October 1992.

[Bow92] Bruce Bower. Rethinking the mind. *Science News*, 142:264, October 10 1992.

[Bro91] Chappell Brown. Polyimide gains favor. *Electronic Engineering Times*, page 110, May 13 1991.

[BS90] Guy E. Blelloch and Gary W. Sabot. Compiling collection-oriented languages onto massively parallel computers. *Journal of Parallel and Distributed Computing*, 8:119–134, 1990.

[BS93] Kenneth Baclawski and J. Elliot Smith. KEYNET: Fast indexing for semantically rich information retrieval. Unpublished, from kenb@ccs.neu.edu, December 1993.

[Bur88] Alan Burns. *Programming in Occam 2*. Addison-Wesley, New York, 1988.

[Bur92] Dave Bursky. Memory—CPU interface speeds up data transfers. *Electronic Design*, pages 137–142, March 19 1992.

[CD89] Lawrence Chisvin and R. James Duckworth. Content-addressable and associative memory: Alternatives to the ubiquitous RAM. *Computer*, 22(7):51–64, July 1989.

[CG89] Nicholas Carriero and David Gelernter. Linda in context. *Communications of the ACM*, 32(4):444–458, April 1989.

[CG94] Van-Dat Cung and Lucien Gotte. A first step towards the massively parallel game-tree search: A SIMD approach. In V. Kumar et al., editors, *Parallel Processing for Artificial Intelligence*, pages 117–130, New York, 1994. North-Holland. volume 2.

[CGA94] Chen-Chau Chu, Joydeep Gosh, and J. K. Aggarwal. On supporting rule-based image interpretation using a distributed memory multicomputer. In V. Kumar et al., editors, *Parallel Processing for Artificial Intelligence*, pages 21–44, New York, 1994. North-Holland. volume 1.

[CGB91] David R. Cheriton, Hendrik A. Goosen, and Patrick D. Boyle. Paradigm: A highly scalable shared-memory multicomputer architecture. *Computer*, 24(2):33–45, February 1991.

[CGK93] Daniel J. Challou, Maria Gini, and Vipin Kumar. Parallel search algorithms for robot motion planning. In James Geller, editor, *Innovative Applications of Massive Parallelism*, pages 40–47. AAAI, 1993. Working Notes, Spring Symposium Series, Stanford University.

[CGK94] Daniel J. Challou, Maria Gini, and Vipin Kumar. Toward real-time motion planning. In V. Kumar et al., editors, *Parallel Processing for Artificial Intelligence*, pages 163–176, New York, 1994. North-Holland. volume 2.

[CH67] T. M. Cover and P. E. Hart. Nearest neighbor pattern classification. *IEEE Transactions on Information Theory*, IT-13(1):21–27, January 1967.

[Chr90] Peter Christy. Software to support massively parallel computing on the MasPar MP-1. In *Proceedings of CompCon*, pages 29–33. IEEE Computer Society, 1990.

[CK80] D. G. Corneil and D. G. Kirkpatrick. A theoretical analysis of various heuristics for the graph isomorphism problem. *Siam Journal on Computing*, 9(2), May 1980.

[CLR90] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press, Cambridge, Mass., 1990.

[CM87] Nick Cercone and Gordon McCalla. What is knowledge representation. In *The Knowledge Frontier*, chapter 1, pages 1–43. Springler-Verlag, New York, 1987.

[CMD94] Sang-Hwa Chung, Dan I. Moldovan, and Ronald F. DeMara. A parallel computational model for integrated speech and natural language understanding. In Hiroaki Kitano and James A. Hendler, editors, *Massively Parallel Artificial Intelligence*, pages 138–170, Menlo Park, CA, 1994. AAAI Press / The MIT Press.

[Coo93] Diane J. Cook. Fast information distribution for massively parallel IDA* search. In James Geller, editor, *Innovative Applications of Massive Parallelism*, pages 48–53. AAAI, 1993. Working Notes, Spring Symposium Series, Stanford University.

[Cor95] Digi-Key Corp. *Distributor's catalog*. Thief River Falls, MN, 951 edition, January 1995.

[CR94] Alok Choudhary and Sanjay Ranka. A perspective on parallel processing in computer vision and image understanding. In V. Kumar et al., editors, *Parallel Processing for Artificial Intelligence*, pages 3–20, New York, 1994. North-Holland. volume 1.

[CS93] Scott Cost and Steven Salzberg. A weighted nearest neighbor algorithm for learning with symbolic features. *Machine Learning*, 10(1):57–78, 1993.

[CS94] Kevin J. Cherkauer and Jude W. Shavlik. Selecting salient features for machine learning from large candidate pools through parallel decision-tree construction. In Hiroaki Kitano and James A. Hendler, editors, *Massively Parallel Artificial Intelligence*, pages 102–137, Menlo Park, CA, 1994. AAAI Press / The MIT Press.

[Dal90] William J. Dally. Performance analysis of k-ary n-cube interconnection networks. In Patrick Henry Winston and Sara Alexandra Shellard, editors, *Artificial Intelligence at MIT: Expanding Frontiers*, pages 521–547. MIT Press, 1990.

[DB93] Rumi M. Dubash and Farokh B. Bastani. Decentralized massively parallel path planning and its application to process control and multi robot systems. In James Geller, editor, *Innovative Applications of Massive Parallelism*, pages 54–61. AAAI, 1993. Working Notes, Spring Symposium Series, Stanford University.

[DeM92] Ronald F. DeMara. Parallelism, design, and performance of a marker-propagation reasoning system. Technical Report PKPL 92-9, Department of Electrical Engineering Systems, USC, 1992.

[Dev93] Keith Devlin. *The Joy of Sets*. Springer-Verlag, New York, 1993.

[DFM89] Jose G. Delgado-Frias and Will R. Moore. *VLSI for Artificial Intelligence*. Kluwer Academic Publishers, Boston, Mass., 1989.

[DM93] R. F. DeMara and D. I. Moldovan. The SNAP-1 parallel AI prototype. *IEEE Transactions on Parallel and Distributed Systems*, 4(8):841–854, August 1993.

[DN94] Larry Davis and P. J. Narayanan. Massively parallel search for the interpretation of aerial images. In Hiroaki Kitano and James A. Hendler, editors, *Massively Parallel Artificial Intelligence*, pages 246–279, Menlo Park, CA, 1994. AAAI Press / The MIT Press.

[EAH93] Matthew P. Evett, William A. Anderson, and James A. Hendler. Massively parallel support for computationally effective recognition queries. In James Geller, editor, *Innovative Applications of Massive Parallelism*, pages 70–84. AAAI, 1993. Working Notes, Spring Symposium Series, Stanford University.

[EAH94] M. P. Evett, W. A. Anderson, and J. A. Hendler. Providing computationally effective knowledge representation via massive parallelism. In V. Kumar et al., editors, *Parallel Processing for Artificial Intelligence*, pages 115–138, New York, 1994. North-Holland. volume 1.

[EH90] Matthew P. Evett and James Hendler. PARKA: Frame-based property inheritance on the Connection Machine. In *Parallel Computing 89*, pages 381–386, 1990.

[EL92] Gerard Ellis and Robert Levinson, editors. *Proceedings of the First International Workshop on PEIRCE: A Conceptual Graphs Workbench.* Department of Computer Science, The University of Queensland, 1992.

[Ell92] Gerard Ellis. Compiling conceptual graphs. *IEEE Transactions on Knowledge and Data Engineering*, 1992. to appear.

[Ell93] Gerard Ellis. Efficient retrieval from hierarchies of objects using lattice operations. In *Conceptual Graphs for Knowledge Representations*, pages 274–293, New York, 1993. Springer-Verlag.

[ELR95] Gerard Ellis, Robert A. Levinson, and Peter J. Robinson. Managing complex objects in peirce. *International Journal on Man-Machine Studies*, 1995. To appear.

[Ert93] Wolfgang Ertel. Massively parallel search with random competetion. In James Geller, editor, *Innovative Applications of Massive Parallelism*, pages 62–69. AAAI, 1993. Working Notes, Spring Symposium Series, Stanford University.

[Fah85] Scott E. Fahlman. Extended abstract: Parallel processing in artificial intelligence. *Parallel Computing*, 2:283–286, 1985.

[Fah93] Scott E. Fahlman. Some thoughts on NETL 15 years later. In James Geller, editor, *Innovative Applications of Massive Parallelism*, pages 85–89. AAAI, 1993. Working Notes, Spring Symposium Series, Stanford University.

[FBF77] Jerome H. Friedman, Jon Louis Bentley, and Raphael Ari Finkel. An algorithm for finding best matches in logarithmic expected time. *ACM Transactions on Mathematical Software*, 3(3):209–226, September 1977.

[FFG94] Michael Factor, Scott J. Fertig, and David H. Gelernter. Process trellis and FGP: Software architectures for data filtering and mining. In V. Kumar et al., editors, *Parallel Processing for Artificial Intelligence*, pages 409–428, New York, 1994. North-Holland. volume 1.

[FKB87] M. J. Flynn, P. Kanerva, and N. Bhadkamkar. Sparse distributed memory: Principles of operation. Technical Report CSL-TR87-338, Stanford University, 1987.

[FKB89] M. J. Flynn, P. Kanerva, and N. Bhadkamkar. Sparse distributed memory principles of operation. Technical Report 89.53, NASA Ames Research Center, RIACS, December 1989.

[FKS85] Christer Fernstrom, Ivan Kruzela, and Bertil Svensson. *LUCAS Associative Array Processor: Design, Programming, and Application Studies.* Springer-Verlag, New York, 1985.

[FLR91] Jerome A. Feldman, Chu-Cheow Lim, and Thomas Rauber. The shared-memory language pSather on a distributed-memory multiprocessor. Technical report, International Computer Science Institute, University of California, Berkeley, 1991.

[FMD88] T. J. Fountain, K. N. Mathews, and M. J. B. Duff. The CLIP7A image processor. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, 10:310–319, March 1988.

[Fos76] Caxton C. Foster. *Content-Addressable Parallel Processors.* Van Nostrand Reinhold, New York, 1976.

[Gei93] G. A. Geist. PVM 3 beyond network computing. In *2nd International ACPC Conference*, pages 194–203, 1993.

[Gel93a] James Geller, editor. *Innovative Applications of Massive Parallelism.* AAAI, 1993. Working Notes, Spring Symposium Series, Stanford University.

[Gel93b] James Geller. Massively parallel knowledge representation. In James Geller, editor, *Innovative Applications of Massive Parallelism*, pages 90–97. AAAI, 1993. Working Notes, Spring Symposium Series, Stanford University.

[Gel94a] James Geller. Advanced update operations in massively parallel knowledge representation. In Hiroaki Kitano and James A. Hendler, editors, *Massively Parallel Artificial Intelligence*, pages 74–101, Menlo Park, CA, 1994. AAAI Press / The MIT Press.

[Gel94b] James Geller. Inheritance operations in massively parallel knowledge representation. In V. Kumar et al., editors, *Parallel Processing for Artificial Intelligence*, pages 95–114, New York, 1994. North-Holland. volume 1.

[Gen83] D. Gentner. Structure-mapping: a theoretical framework for analogy. *Cognitive Science*, 7(2), 1983.

[GJ79] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman and Company, San Francisco, 1979.

[GR90] Amit Gulati and Don Roberts. A cascadable content-addressable memory based database query accelerator with proximity resolution. (class project report), 1990.

[GW94] George Gusciora and Jon A. Webb. Parallel affine image warping. In V. Kumar et al., editors, *Parallel Processing for Artificial Intelligence*, pages 45–66, New York, 1994. North-Holland. volume 1.

[H$^+$86] J. P. Hayes et al. A microprocessor-based hypercube supercomputer. *IEEE Micro*, 6(5):6–17, October 1986.

[H$^+$91] Tetsuya Higuchi et al. IXM2: a parallel associative processor. *ACM SIGARCH*, 19(3):22–33, September 1991.

[H$^+$93] Takehiro Hasegawa et al. An experimental RAM with a NAND-structured cell. *IEEE Journal of Solid-State Circuits*, 28(11):1099–1103, November 1993.

[H$^+$94] Tetsuya Higuchi et al. The IMX2 parallel associative processor for AI. *Computer*, pages 53–63, November 1994.

[Har92] Charles A. Hart. Dynamic RAM as secondary cache. *IEEE Spectrum*, October 1992.

[HC91] Young-Sik Hong and Su-Shing Chen. Character recognition in a sparse distributed memory. *IEEE Transactions on Systems, Man, and Cybernetics*, 21(3):675–678, May/June 1991.

[HCWS94] Martin C. Herbordt, James C. Corbett, Charles C. Weems, and John Spalding. Practical algorithms for online routing on fixed and reconfigurable meshes. *Journal of Parallel and Distributed Computing*, 20:341–355, 1994.

[Hen94] Dominik Henrich. Initialization of parallel branch-and-bound algorithms. In V. Kumar et al., editors, *Parallel Processing for Artificial Intelligence*, pages 131–144, New York, 1994. North-Holland. volume 2.

[HGC87] Kai Hwang, Joydeep Ghosh, and Raymond Chowkwanyun. Computer architecture for artificial intelligence processing. *Computer*, pages 19–27, January 1987.

[Hig90] Lee Higbie. Quick and easy cache performance analysis. *ACM Computer Architecture News*, 18(2), June 1990.

[Hil85] W. Daniel Hillis. *The Connection Machine.* MIT Press, Cambridge, Mass., 1985.

[HIM94] Tetsuya Higuchi, Hitoshi Iba, and Bernard Manderick. Evolvable hardware with genetic learning. In Hiroaki Kitano and James A. Hendler, editors, *Massively Parallel Artificial Intelligence*, pages 398–422, Menlo Park, CA, 1994. AAAI Press / The MIT Press.

[HJ93] Walter Hower and Stephan Jacobi. Massively distributed constraint satisfaction. In James Geller, editor, *Innovative Applications of Massive Parallelism*, pages 99–105. AAAI, 1993. Working Notes, Spring Symposium Series, Stanford University.

[HJ94] Walter Hower and Stephan Jacobi. Parallel distributed realization for constraint satisfaction. In V. Kumar et al., editors, *Parallel Processing for Artificial Intelligence*, pages 107–116, New York, 1994. North-Holland. volume 2.

[HL91] Richard Hughey and Daniel P. Lopresti. B-SYS: A 470-processor programmable systolic array. In Chuan lin Wu, editor, *Proc. Int. Conf. Parallel Processing*, volume 1, pages 580–583. CRC Press, August 1991.

[HLR93] Richard Hughey, Robert Levinson, and James D. Roberts. Issues in parallel hardware for graph retrieval. In *Proceedings of the 1st International Conference on Conceptual Structures*, 1993.

[HO82] Christoph M. Hoffmann and Michael J. O'Donnel. Pattern matching in trees. *Journal of the ACM*, 29(1):68–95, January 1982.

[Hoa78] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–667, August 1978.

[Hop82] J. J. Hopfield. Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the National Academy of Sciences*, 79:2554–2558, 1982.

[Hug91] Richard P. Hughey. Programmable sytolic arrays. Technical Report CS-92-34, Department of Computer Science, Brown University, 1991.

[Hwa93] Kai Hwang. *Advanced Computer Architecture.* McGraw Hill, New York, 1993.

[IEE94] IEEE. *Computer*, November 1994. Special issue on associative processing.

[ISI93] ISI. MOSIS price list. email, MOSIS fabrication service, 1993.

[J^+88] Simon R. Jones et al. A 9-kbit associative memory for high-speed parallel processing applications. *IEEE Journal of Solid-State Circuits*, 23(2):543–548, 1988.

[Jae89a] L. A. Jaeckel. An alternative design for a sparse distributed memory. Technical report, Research Institute for Advanced Computer Science, NASA Ames Research Center, 1989.

[Jae89b] L. A. Jaeckel. A class of designs for a sparse distributed memory. Technical report, Research Institute for Advanced Computer Science, NASA Ames Research Center, 1989.

[JB94] Todd Jochem and Shumeet Baluja. Massively parallel, adaptive, color image processing for autonomous road following. In Hiroaki Kitano and James A. Hendler, editors, *Massively Parallel Artificial Intelligence*, pages 280–315, Menlo Park, CA, 1994. AAAI Press / The MIT Press.

[Jon92] Fred Jones. A new era of fast dynamic RAMs. *IEEE Spectrum*, October 1992.

[JS94] Jing-Fu Jenq and Sartaj Sahni. Image processing on reconfigurable meshes with buses. In V. Kumar et al., editors, *Parallel Processing for Artificial Intelligence*, pages 67–94, New York, 1994. North-Holland. volume 1.

[JSS93] Ken Jung, Evangelos Simoudis, and Ramesh Subramonian. Parallel induction systems based on branch and bound. In James Geller, editor, *Innovative Applications of Massive Parallelism*, pages 106–112. AAAI, 1993. Working Notes, Spring Symposium Series, Stanford University.

[K^+86] Toshiro Kondo et al. Pseudo MIMD array processor — AAP2. In *13th Conf. on Computer Architecture*, pages 330–377, 1986.

[K^+89] Peter Kogge et al. VLSI and rule-based systems. In Jose G. Delgado-Frias and Will R. Moore, editors, *VLSI for Artificial Intelligence.* Kluwer Academic Publishers, Boston, Mass., 1989.

[K^+93] Goro Kitsukawa et al. 256-Mb DRAM circuit technologies for file applications. *IEEE Journal of Solid-State Circuits*, 28(11):1105–1112, November 1993.

[K^+94a] V. Kumar et al., editors. *Parallel Processing for Artificial Intelligence*, volume 1. North-Holland, New York, 1994.

[K^+94b] V. Kumar et al., editors. *Parallel Processing for Artificial Intelligence*, volume 2. North-Holland, New York, 1994.

[Kan88] Pentti Kanerva. *Sparse Distributed Memory.* MIT Press, Cambridge, Mass., 1988.

[Kan92] Pentti Kanerva. Sparse distributed memory and related models. In H. M. Hassoun, editor, *Associative Neural Memories: Theory and Implementation.* Oxford University Press, New York, 1992. Pre-publication.

[Kee88] James D. Keeler. Comparison between Kanerva's SDM and Hopfield-type neural networks. *Cognitive Science*, 12:299–329, 1988.

[KF90] Dradley C. Kuszmaul and Jeff Fried. NAP (No ALU Processor): The great communicator. *J. Parallel and Distributed Computing*, 8:169–179, 1990.

[KH94] Hiroaki Kitano and James A. Hendler, editors. *Massively Parallel Artificial Intelligence*, Menlo Park, CA, 1994. AAAI Press / The MIT Press.

[Kit94] Hiroaki Kitano. The challenge of massive parallelism. In Hiroaki Kitano and James A. Hendler, editors, *Massively Parallel Artificial Intelligence*, pages 1–51, Menlo Park, CA, 1994. AAAI Press / The MIT Press.

[KK93] George Karypis and Vipin Kumar. Unstructured tree search on SIMD parallel computers: Experimental results. In James Geller, editor, *Innovative Applications of Massive Parallelism*, pages 113–119. AAAI, 1993. Working Notes, Spring Symposium Series, Stanford University.

[Kni91] Thomas F. Knight Jr. Technologies for low latency interconnection switches. *Computer Architecture News*, March 1991.

[Kog82] Peter M. Kogge. An architectural trail to threaded-code systems. *Computer*, March 1982.

[Kog91] Peter M. Kogge. *The Architecture of Symbolic Computers.* McGraw-Hill, New York, 1991.

[Koh80] Tuevo Kohonen. *Content-Addressable Memories.* Springer-Verlag, New York, 1980.

[Kol93] Janet Kolodner. *Case-Based Reasoning.* Morgan Kaufmann, San Mateo, CA, 1993.

[KT88] Masaru Kitsuregawa and Hidehiko Tanaka, editors. *Database Machines and Knowledge Base Machines.* Kluwer Academic Publishers, Boston, Mass., 1988.

[Kum93] Deepak Kumar. An AI architecture based on message passing. In James Geller, editor, *Innovative Applications of Massive Parallelism*, pages 127–132. AAAI, 1993. Working Notes, Spring Symposium Series, Stanford University.

[Kur90] Raymond Kurzweil. *The Age of Intelligent Machines.* MIT Press, Cambridge, Mass., 1990.

[Kur93] Franz Kurfess. Massive parallelism in logic. In James Geller, editor, *Innovative Applications of Massive Parallelism*, pages 132–143. AAAI, 1993. Working Notes, Spring Symposium Series, Stanford University.

[Kur94] Franz Kurfess. Massive parallelism in inference systems. In V. Kumar et al., editors, *Parallel Processing for Artificial Intelligence*, pages 259–278, New York, 1994. North-Holland. volume 1.

[L$^+$89] David D. Lee et al. A VLSI chip set for a multiprocesspr workstation — part I: A RISC microprocessor with coprocessor interface and support for symbolic processing. *IEEE Journal of Solid-State Circuits*, 24(6):1688–1697, December 1989.

[L+90] M. J. Little et al. The 3-d computer. *Journal of VLSI Signal Processing*, 2:79–87, 1990.

[Lan93] Trent E. Lange. Massively-parallel inferencing for natural language understanding and memory retrieval in structured spreading-activation networks. In James Geller, editor, *Innovative Applications of Massive Parallelism*, pages 144–149. AAAI, 1993. Working Notes, Spring Symposium Series, Stanford University.

[LAS89] Y. C. Lee, J. P. Avery, and R. Su. Quick prototyping and manufacturing for HWSI-based supercompact systems. In *Government Microcircuit Applications Conference*, 1989.

[LCV94] Yannick Lallement, Thierry Cornu, and Stephane Vialle. An abstract machine for implementing connectionist and hybrid systems on multi-processor architectures. In V. Kumar et al., editors, *Parallel Processing for Artificial Intelligence*, pages 11–28, New York, 1994. North-Holland. volume 2.

[LE92] Robert A. Levinson and Gerard Ellis. Multi-level hierarchical retrieval. *Knowledge-Based Systems Journal*, 5(3), September 1992.

[Lee86] Dik Lun Lee. ALTEP – a cellular processor for high-speed pattern matching. *New Generation Computing*, 4:225–244, 1986.

[Leh92] Fritz Lehmann. Semantic networks. In *Semantic Networks in Artificial Intelligence*, pages 1–50. Pergamon Press, 1992.

[Lei85] Charles E. Leiserson. Fat-trees: Universal networks for hardware-efficient supercomputing. *IEEE Transactions on Computers*, C-34(10):892–910, October 1985.

[Lei90] Michael R. Leibowitz. System makers eye 3D chip pack. *EDN News*, April 19 1990.

[Len88] George G. Lendaris. Representing conceptual graphs for parallel processing. In *Conceptual Graphs Workshop*, 1988.

[Len92] George G. Lendaris. A neural-network approach to implementing conceptual graphs. In Timothy E. Nagel et al., editors, *Conceptual Structures, Current Research and Practice*, chapter 8, pages 155–188. Ellis Horwood, New York, 1992.

[Lev91] R. A. Levinson. A self-organizing pattern retrieval system. *International J. of Intelligent Systems*, 6:717–738, 1991.

[Lev92a] R. A. Levinson. Pattern associativity and the retrieval of semantic networks. *Semantic Networks in AI*, 23:573–600, 1992.

[Lev92b] Robert Levinson. Pattern associativity and the retrieval of semantic networks. *Journal of Computers and Mathematics with Applications*, 23:573–600, 1992.

[Lev93] R. A. Levinson. Exploiting the physics of state-space search. In Susan Epstein and R. A. Levinson, editors, *Proceedings AAAI Fall Symposium on Games: Planning and Learning*, pages 157–165, Menlo Park, CA, 1993. AAAI Press.

[Lev94a] Robert Levinson. Morph II, a universal agent: Progress report and proposal. Technical Report UCSC-CRL-94-22, University of California, Santa Cruz, 1994.

[Lev94b] Robert Levinson. UDS: A universal data structure. In W. M. Tepfenhart, I. P. Dide, and J. F. Sowa, editors, *Conceptual Structures: Theory and Practice*, pages 230–250. Springer-Verlag, New York, 1994. Lecture Notes in AI 835.

[LG90] D. B. Lenat and R. V. Guha. *Building Large Knowledge-Based Systems.* Addison Wesley, Reading, Mass., 1990.

[LH94] Ahmed Louri and James A. Hatch Jr. An optical parallel associative processor for high-speed database processing. *Computer*, pages 65–72, November 1994.

[LHB93] Q. Y. Luo, P. G. Hendry, and J. T. Buchanan. Comparison of different approaches for solving distributed constraint satisfaction problems. In James Geller, editor, *Innovative Applications of Massive Parallelism*, pages 150–159. AAAI, 1993. Working Notes, Spring Symposium Series, Stanford University.

[LJ91] R. Mike Lea and Ian P. Jalowiecki. Associative massively parallel computers. *Proceedings of the IEEE*, 79(4):469–479, April 1991.

[LK93] R. Levinson and K. Karplus. Graph-isomorphism and experience-based planning. In D. Subramaniam, editor, *Proceedings Workshop on Knowledge Compilation and Speedup Learning*, Amherst, Mass., 1993. Machine Learning Conference.

[Lou91] Ahmed Loure. Design of an optical content-addressable parallel processor with applications to fast searching and information retrieval. In V. K. Prasanna Kumar, editor, *Proc. 5th International Parallel Processing Symposium*, pages 234–239. IEEE Computer Society Press, 1991.

[LP94] Wei-Ming Lin and Wiktor K. Prasanna. Parallel algorithms and architectures for consistent labeling. In V. Kumar et al., editors, *Parallel Processing for Artificial Intelligence*, pages 335–364, New York, 1994. North-Holland. volume 1.

[LS92] Hungwen Li and Quentin F. Stout, editors. *Reconfigurable Massively Parallel Computers.* Prentis Hall, Englewood Cliffs, New Jersey, 1992.

[Luc89] D. Lucarella. Heuristics to locate the best document set in information retrieval systems. In *International Phoenix Conference on Computers and Communications.* IEEE Communications Society, 1989.

[M+90] Masato Motomura et al. A 1.2 million transistor, 33-MHz, 20-b dictionary search (DISP) ULSI with a 160-kb CAM. *Journal of Solid-State Circuits*, 25(5):1158–1165, October 1990.

[M+92] Dan Moldovan et al. SNAP: Parallel processing applied to AI. *Computer*, 25(5):39–49, May 1992.

[MA90] Daniel P. Miranker and Archie D. Andrews. On balanced synchronous parallel computers for AI. In *International Conference on Parallel Processing*, pages I489–I493. CRC Press, 1990.

[Mal93] David Maliniak. Laminated-film MCMs aim for mass market. *Electronic Design*, pages 57–64, November 11 1993.

[Mar69] D. Marr. A theory of cerebellar cortex. *J. Physiology*, 202:437–470, 1969.

[Mar90] Tony R. Martinez. Smart memory architecture and methods. *Future Generation Computer Systems*, 6:145–162, 1990.

[Mas92a] MasPar Computer Corp., Sunnyvale, CA. *The Design of the MasPar MP-2: A Cost Effective Massively Parallel Computer*, 1992.

[Mas92b] MasPar Computer Corp., Sunnyvale, CA. *MasPar Programming Language Reference Manual*, 1992.

[MC80] Carver Mead and Lynn Conway. *Introduction to VLSI Systems.* Addison Wesley, Reading, Mass., 1980.

[Men91] Lawrence Mennemeier. Hardware mechanisms to support concurrent threads on RISC and superscalar multiprocessors. Master's thesis, University of California, Santa Cruz, 1991.

[Mic89] Micro Devices. *Micro Devices Application Notes, MD1210 Fuzzy Set Comparator*, December 1989.

[MLL90] Dan Moldovan, Wing Lee, and Changhwa Lin. SNAP: A marker-passing architecture for knowledge processing. Technical Report PKPL 90-4, University of Southern California, 1990.

[MNT94] Takao Mohri, Masaaki Nakamura, and Hidehiko Tanaka. Weather forecasting using memory-based reasoning. In V. Kumar et al., editors, *Parallel Processing for Artificial Intelligence*, pages 185–198, New York, 1994. North-Holland. volume 2.

[Mor90] Larry L. Moresco. Electronic system packaging: The search for manufacturing the optimum in a sea of constraints. *IEEE Transactions on Components, Hybrids, and Manufacturing Technology*, 13(3):494–508, September 1990.

[MQF91] Johannes M. Mulder, Nhon T. Quach, and Michael J. Flynn. An area model for on-chip memories and its application. *IEEE Journal of Solid-State Circuits*, 26(2):98–106, February 1991.

[MSS85] David G. Meyer, Howard Jay Siegel, and Thomas Schwederski. The PASM parallel system prototype. In *Proceedings of COMPCON*, pages 429 – 434. IEEE Computer Society Press, 1985.

[NCR87] NCR Corporation. *Geometric Arithmetic Parallel Processor Data Sheet*, 1987.

[ND94] Valeriy I. Nenov and Michael G. Dyer. Language learning via perceptual/motor association: A massively parallel model. In Hiroaki Kitano and James A. Hendler, editors, *Massively Parallel Artificial Intelligence*, pages 202–245, Menlo Park, CA, 1994. AAAI Press / The MIT Press.

[Ng92] Ray Ng. Fast computer memories. *IEEE Spectrum*, October 1992.

[NGC89] Yan Ng, Raymond Glover, and Chew-Lye Chng. Unify with active memory. In Jose G. Delgado-Frias and Will R. Moore, editors, *VLSI for Artificial Intelligence.* Kluwer Academic Publishers, Boston, Mass., 1989.

[Nic90] John R. Nickolls. The design of the MasPar MP-1: A cost effective massively parallel computer. In *IEEE Proceedings of Compcon*, pages 25–28, 1990.

[NM93] J. Thomas Ngo and Joe Marks. Massively parallel genetic algorithm for physically correct articulated figure locomotion. In James Geller, editor, *Innovative Applications of Massive Parallelism*, pages 167–173. AAAI, 1993. Working Notes, Spring Symposium Series, Stanford University.

[NW93] Katsumi Nitta and Stephen Wong. The role of parallelism in parallel inference applications. In James Geller, editor, *Innovative Applications of Massive Parallelism*, pages 174–180. AAAI, 1993. Working Notes, Spring Symposium Series, Stanford University.

[O$^+$89] Takeshi Ogura et al. A 20-kbit associative memory LSI for artificial intelligence machines. *IEEE Journal of Solid-State Circuits*, 24(4):1014–1020, August 1989.

[O$^+$94] Kozo Oi et al. Toward massively parallel spoken language translation. In V. Kumar et al., editors, *Parallel Processing for Artificial Intelligence*, pages 177–184, New York, 1994. North-Holland. volume 2.

[Ode91] Louis L. Odette. *Intelligent Embedded Systems.* Addison-Wesley, Reading, Mass., 1991.

[Oed93] Wilfried Oed. *The Cray Research Massively Parallel Processor System CRAY T3D.* Cray Research GmbH, Munich, Germany, November 15 1993.

[OL92] Stephen M. Omohundro and Chu-Chow Lim. The Sather language and libraries. Technical report, International Computer Science Institute, University of California, Berkeley, 1992.

[Omo87] Stephen M. Omohundro. Efficient algorithms with neural network behavior. Technical Report UIUCDCS-R-87-1331, Department of Computer Science, University of Illinois at Urbana-Champaign, 1987.

[Omo91] Stephen M. Omohundro. The Sather language. Technical report, International Computer Science Institute, University of California, Berkeley, 1991.

[OMT94] Gerald Ouvradou, Aymeric Poulain Maubant, and Andre Thepaut. Hybrid systems on a multi-grain parallel architecture. In V. Kumar et al., editors, *Parallel Processing for Artificial Intelligence*, pages 3–10, New York, 1994. North-Holland. volume 2.

[P$^+$94] Jerry Potter et al. ASC: An associative-computing paradigm. *Computer*, pages 19–25, November 1994.

[Par90] Behrooz Parhami. Scalable architectures for VLSI based associative memories. In N. Rishe et al., editors, *Parallel Architectures.* IEEE Computer Science Press, Washington, D.C., 1990.

[Pea94] Robert Pearson. A coarse grained parallel induction heuristic. In V. Kumar et al., editors, *Parallel Processing for Artificial Intelligence*, pages 207–226, New York, 1994. North-Holland. volume 2.

[PF89] R. W. Prager and F. Fallside. The modified Kanerva model for automatic speech recognition. *Computer Speech and Language*, 3(1):61–81, 1989.

[Pin94] Gadi Pinkas. Representing propositional logic and searching for satisfiability in connectionist networks. In V. Kumar et al., editors, *Parallel Processing for Artificial Intelligence*, pages 279–304, New York, 1994. North-Holland. volume 1.

[PJK89] Jr. Philip J. Koopman. *Stack Computers, the New Wave.* John Wiley and Sons, New York, 1989.

[PKF93] Curt Powley, Richard E. Korf, and Chris Ferfuson. Parallelization of tree-recursive algorithms on a SIMD machine. In James Geller, editor, *Innovative Applications of Massive Parallelism*, pages 181–186. AAAI, 1993. Working Notes, Spring Symposium Series, Stanford University.

[Pou91] Dick Pountain. The Transputer strikes back. *Byte*, pages 265–275, August 1991.

[PW94] Michael A. Palis and David S. L. Wei. Massively parallel parsing algorithms for natural language. In V. Kumar et al., editors, *Parallel Processing for Artificial Intelligence*, pages 365–408, New York, 1994. North-Holland. volume 1.

[RD92] James D. Roberts and W. W-M. Dai. Early system analysis of cache performance for RISC systems. In *IEEE Multi-Chip Module Conference*, pages 130–133, 1992.

[RG91] D. P. Rodohan and R. J. Glover. An overview of the A* architecture for optimization problem in a logic programming environment. *ACM Computer Architecture News*, 19(4), June 1991.

[RHW86] D. Rumelhart, G. Hinton, and R. Williams. Learning representations by back-propagating errors. *Nature*, 323:533–536, 1986.

[RN91] James D. Roberts and Claude Noshpitz. Implementing Kanerva's sparse distributed memory on the Connection Machine. Forthcoming technical report, UCSC, 1991.

[Rob89a] James D. Roberts. A CAM-based search accelerator. *IEEE Computing Futures*, Winter 1989.

[Rob89b] James D. Roberts. Proximity search via content-addressable memory PCAM. Technical Report UCSC-CRL-89-43, University of California, Santa Cruz, 1989.

[Rob90] James D. Roberts. Proximity content-addressable memory: an efficient extension to k-nearest neighbors search. Master's thesis, University of California Santa Cruz, September 1990.

[Rob93] Ian N. Robinson. PAM: Massive parallelism in support of run-time intelligence. In James Geller, editor, *Innovative Applications of Massive Parallelism*, pages 193–199. AAAI, 1993. Working Notes, Spring Symposium Series, Stanford University.

[Rog90] David Rogers. Predicting weather using a genetic memory: A combination of Kanerva's sparse distributed memory and Holland's genetic algorithms. In D. S. Touretzky, editor, *Advances in Neural Information Processing Systems 2*, pages 275–289. Morgan Kaufmann, San Mateo, CA, 1990.

[Rya94] Bob Ryan. Alpha rides high. *Byte*, pages 197–198, October 1994.

[S^+85] Lawrence Snyder et al., editors. *Algorithmically Specialized Parallel Computers.* Academic Press, San Diego, CA, 1985.

[S^+91] Hirofumi Shinohara et al. A flexible multiport RAM compiler for data path. *IEEE Journal of Solid-State Circuits*, 26(3):343–349, March 1991.

[S^+93a] Katsuro Sasaki et al. A 16-Mb CMOS SRAM with a 2.3-μm^2 single-bit-line memory cell. *IEEE Journal of Solid-State Circuits*, 28(11):1125–1130, November 1993.

[S^+93b] Katsunori Seno et al. A 9-ns 16-Mb CMOS SRAM with offset-compensated current sense amplifier. *IEEE Journal of Solid-State Circuits*, 28(11):1119–1123, November 1993.

[S^+93c] Salvatore J. Stollfo et al. A parallel and distributed environment for database rule processing: Open problems and future directions. In James Geller, editor, *Innovative Applications of Massive Parallelism*, pages 207–215. AAAI, 1993. Working Notes, Spring Symposium Series, Stanford University.

[S$^+$94] Kazumasa Suzuki et al. A 500 MHz, 32 bit, 0.4 micron CMOS RISC processor. *IEEE Journal of Solid-State Circuits*, 29(12):1464–1473, December 1994.

[Sab88] Gary Sabot. *The Paralation Model.* The MIT Press, Cambridge, MA, 1988.

[Sat94] Satoshi Sato. Example-based translation and its MIMD implementation. In Hiroaki Kitano and James A. Hendler, editors, *Massively Parallel Artificial Intelligence*, pages 171–201, Menlo Park, CA, 1994. AAAI Press / The MIT Press.

[Sch92] Heinz W. Schmidt. Data-parallel object-oriented programming. In *Fifth Australian Supercomputer Conference*, pages 263–272, 1992.

[SDDS86] J. T. Schwartz, R. B. K. Dewar, E. Dubinsky, and E. Schonberg. *Programming with Sets: An Introduction to SETL.* Springer-Verlag, New York, 1986.

[SFS77] R. J. Swan, S. H. Fuller, and D. P. Siewiorek. Cm* — a modular, multi-microprocessor. In *Proc. AFIPS National Computer Conference*, volume 46, pages 637–644, 1977.

[Sha88] David Elliot Shaw. Organization and operation of massively parallel machines. In Guy Rabbat, editor, *Handbook of Advanced Semiconductor Technology and Computer Systems.* Van Nostrand Reinhold, New York, 1988.

[Sha93] Lokendra Shastri. Leveraging massive parallelism for tractable reasoning — taking inspiration from cognition. In James Geller, editor, *Innovative Applications of Massive Parallelism*, pages 200–206. AAAI, 1993. Working Notes, Spring Symposium Series, Stanford University.

[Sie90] Howard Jay Siegel. *Interconnection Networks for Large-Scale Parallel Processing.* McGraw-Hill, New York, 1990.

[SJ94a] Johannes Schumann and Manfred Jobmann. Scalability of an OR-parallel theorem prover on a network of transputers — a modeling approach. In V. Kumar et al., editors, *Parallel Processing for Artificial Intelligence*, pages 199–206, New York, 1994. North-Holland. volume 2.

[SJ94b] Christian B. Suttner and Manfred R. Jobmann. Simulation analysis of static partitioning with slackness. In V. Kumar et al., editors, *Parallel Processing for Artificial Intelligence*, pages 93–106, New York, 1994. North-Holland. volume 2.

[SLH94] Kilian Stoffel, Ian Law, and Beat Hirsbrunner. Fuzzy logic controlled dynamic allocation system. In V. Kumar et al., editors, *Parallel Processing for Artificial Intelligence*, pages 227–239, New York, 1994. North-Holland. volume 2.

[Sow] John Sowa. Inference engines using peirce's rules. Email communication to the Conceptual Graphs Internet mailing list, November 11, 1992.

[Sow84] J. F. Sowa. *Conceptual Structures: Information Processing in Mind and Machine.* Addison-Wesley, Reading, Mass., 1984.

[Spe91] Don Speck. The MOSAIC fast 512K scalable CMOS DRAM. In Carlo Sequin, editor, *Advanced Research in VLSI: Proceedings of the 1991 University of California / Santa Cruz Conference*, pages 229–244. MIT Press, 1991.

[SS94a] James G. Schmolze and Wayne Snyder. Using confluence to control parallel production systems. In V. Kumar et al., editors, *Parallel Processing for Artificial Intelligence*, pages 41–56, New York, 1994. North-Holland. volume 2.

[SS94b] Christian B. Suttner and Johann M. Schumann. Parallel automated theorem proving. In V. Kumar et al., editors, *Parallel Processing for Artificial Intelligence*, pages 209–258, New York, 1994. North-Holland. volume 1.

[SSS81] Edmond Schonberg, Jacob T. Schwartz, and Micha Sharir. An automatic technique for selection of data representations in SETL programs. *ACM Transactions on Programming Languages and Systems*, 3(2):126–143, April 1981.

[Sto87] Salvatore Stolfo. Initial performance of the DADO2 prototype. *Computer*, pages 75–83, January 1987.

[Sto91] Charles D. Stormon. The Coherent processor, an associative processor architecture and applications. In *Proceedings of COMPCON*, pages 270–275. IEEE Press, 1991.

[Stu85] Heinrich J. Stuttgen. *A Hierarchical Associative Processing System.* Springer-Verlag, New York, 1985.

[Sus65] E. H. Sussenguth, Jr. A graph-theoretic algorithm for matching chemical structures. *Journal of Chemical Doc.*, 5:36–43, 1965.

[SW86] C. Stanfill and D. Waltz. Towards memory-based reasoning. *Communications of the ACM*, 29(12):1213–1228, December 1986.

[T^+93] Tadahiko et al. A 30-ns 256-Mb DRAM with a multidivided array structure. *IEEE Journal of Solid-State Circuits*, 28(11):1192–1097, November 1993.

[Tan92] Tanner Research. *Volume ASIC Costs*, 1992.

[TD91] Manu Thapar and Bruce Delagi. Cache coherence for large scale shared memory multiprocessors. *Computer Architecture News*, pages 114–119, March 1991.

[Tep93] William T. Tepfenhart. ICCS 94 call for papers. Internet, September 1993.

[Tex91] Texas Instruments. *TMS320C4x User's Guide*, 1991.

[TH88] Stuart K. Tewksbury and L. A. Hornak. Communication network issues and high-density interconnects in large-scale distributed computing systems. *IEEE Journal on Selected Areas in Communications*, 6(3), April 1988.

[Thi92] Thinking Machines Corporation. *CM-5 Technical Summary*, 1992.

[Tou88] David S. Touretzky. Beyond associative memory: Connectionists must search for other cognitive primitives. In *AAAI Spring Symposium Series: Parallel Models of Intelligence*, pages 231–236, March 1988.

[TR88] L. W. Tucker and G. G. Robertson. Architecture and applications of the Connection Machine. *Computer*, 21(8):26–38, August 1988.

[TW91] Arthur Trew and Greg Wilson, editors. *Past, Present, Parallel: A Survey of Available Computer Systems.* Springer-Verlag, New York, 1991.

[Twa94] Kirk Twardowski. An associative architecture for genetic algorithm-based machine learning. *Computer*, pages 27–38, November 1994.

[U^+93] Motomu Ukita et al. A single-bit-line cross-point cell activation (SCPA) architecture for ultra-low-power SRAMs. *IEEE Journal of Solid-State Circuits*, 28(11):1114–1118, November 1993.

[Ull76] J. R. Ullmann. An algorithm for subgraph isomorphism. *Journal of the ACM*, 23(1):31–42, January 1976.

[VL90] Christian Val and Thierry Lemoine. 3-D interconnection for ultra-dense multichip modules. *IEEE Transactions on Components, Hybrids, and Manufacturing Technology*, 13(4):814–821, December 1990.

[Wat85] Satoshi Watanabe. *Pattern Recognition Human and Mechanical.* John Wiley and Sons, New York, 1985.

[WCHC89] Jr. W. C. Hobart and H. G. Cragon. Locality characteristics of symbolic programs. In *Conference on Computer Design.* IEEE Computer Society, 1989.

[WD94] Gregory M. Werner and Michael G. Dyer. BioLand: A massively parallel simulation environment for evolving distributed forms of intelligent behavior. In Hiroaki Kitano and James A. Hendler, editors, *Massively Parallel Artificial Intelligence*, pages 316–349, Menlo Park, CA, 1994. AAAI Press / The MIT Press.

[WE88] Neil Weste and Kamran Eshraghian. *Principles of CMOS VLSI Design.* Addison-Wesley, Reading, MA, 1988.

[WG93] Stefan Winz and James Geller. Methods of large grammar representation in massively parallel parsing systems. In James Geller, editor, *Innovative Applications of Massive Parallelism*, pages 225–233. AAAI, 1993. Working Notes, Spring Symposium Series, Stanford University.

[Win79] P.H. Winston. Learning and reasoning by analogy. *Communications of the ACM*, 23(12):689–703, 1979.

[WL89] Benjamin W. Wah and Guo Jie Li. A survey on the design of multiprocessing systems for artificial intelligence applications. *IEEE Transactions on Systems, Man, and Cybernetics*, 19(4):667–692, July/August 1989.

[WLL90] Benjamin W. Wah, Matthew B. Lowrie, and Guo-Jie Li. Computers for symbolic processing. In Benjamin W. Wah and C. V. Ramamoorthy, editors, *Computers for Artificial Intelligence Processing*, pages 1–73. John Wiley & Sons, New York, 1990.

[WS89] Jon P. Wade and Charles G. Sodini. A ternary content addressable search engine. *IEEE Journal of Solid-State Circuits*, 24(4):1003–1013, August 1989.

[WWR91] Peter Willett, Terence Wilson, and Stewart F. Reddaway. Atom-by-atom searching using massive parallelism: Implementation of the Ullmann subgraph isomorphism algorithm on the distributed array processor. *Journal of the Chemical Information Computation Society*, 31:225–233, 1991.

[YK94] Moritoshi Yasunaga and Hiroaki Kitano. Wafer-scale integration for massively parallel AI. In Hiroaki Kitano and James A. Hendler, editors, *Massively Parallel Artificial Intelligence*, pages 350–397, Menlo Park, CA, 1994. AAAI Press / The MIT Press.

[YZ90] Joseph Yestrebsky and Michael Ziemacki. *Data Preprocessing for AI Systems: A Maximum Information Viewpoint.* Micro Devices, February 1990.

[ZM94] Ying Zhang and Alan K. Mackworth. Parallel and distributed finite constraint satisfaction: Complexity, algorithms and experiments. In V. Kumar et al., editors, *Parallel Processing for Artificial Intelligence*, pages 305–334, New York, 1994. North-Holland. volume 1.

[ZMHV93] Andreas Zell, Niels Mache, Markus Huttel, and Michael Vogt. Massive parallelism in neural network simulation. In James Geller, editor, *Innovative Applications of Massive Parallelism*, pages 234–241. AAAI, 1993. Working Notes, Spring Symposium Series, Stanford University.

[Zus93] Konrad Zuse. *The Computer – My Life*. Springer–Verlag, New York, 1993.