Compiler Support for Machine-Independent Parallelization of Irregular Problems

Reinhard von Hanxleden

CRPC-TR94494-S
December 1994

Center for Research on Parallel Computation
Rice University
P.O. Box 1892
Houston, TX 77251-1892

RICE UNIVERSITY

Compiler Support for Machine-Independent Parallelization of Irregular Problems

by

Reinhard von Hanxleden

A Thesis Submitted
in Partial Fulfillment of the
Requirements for the Degree

Doctor of Philosophy

Approved, Thesis Committee:

Ken Kennedy, Noah Harding Professor, Chair
Department of Computer Science
Rice University

John Mellor-Crummey, Faculty Fellow
Department of Computer Science
Rice University

Mary Wheeler, Noah Harding Professor
Department of Computational and Applied Mathematics
Rice University

Charles Koelbel, Research Scientist
Center for Research on Parallel Computation
Rice University

Ridgway Scott, Professor of
Computer Science and of Mathematics
University of Houston

Houston, Texas
December, 1994

Compiler Support for Machine-Independent Parallelization of Irregular Problems

Reinhard von Hanxleden

Abstract

Data-parallel languages, such as High Performance Fortran or Fortran D, provide a machine-independent data-parallel programming paradigm in which the applications programmer uses a dialect of a sequential language annotated with high-level data-distribution directives. Identifying parallelism in data-parallel applications typically is straightforward, but making efficient use of this parallelism for irregular applications, such as molecular dynamics or unstructured meshes, is a challenge due to the limited compile-time knowledge about data access patterns.

This dissertation establishes the thesis that spatial locality of the underlying problems can be used as a basis of compiler support for parallelizing such applications. The work done for supporting this thesis and for parallelizing applications in general can be divided into three parts, which correspond to different aspects of parallelizing compilers for different architectures. *Value-based mappings* express the spatial locality characteristics of an application and assist the compiler in computing a distribution with both a balanced computational workload and high data access locality. The Give-N-Take *data-flow framework* is an extension of Partial Redundancy Elimination particularly well suited to advanced code-placement tasks such as communication generation. *Loop flattening* is a code transformation to overcome SIMD specific control flow limitations when executing nested loops with varying inner loop bounds, which are typical for irregular problems.

To illustrate this thesis, the Fortran 77D compiler at Rice University has been extended with value-based alignments and distributions, a communication placement mechanism based on the Give-N-Take data-flow framework, and general infrastructure for handling irregular subscripts. This dissertation describes the techniques involved in these extensions and provides experimental results for various irregular applications compiled for a distributed-memory architecture.

Acknowledgments

This dissertation and the excellent research environment in which the underlying work was conducted would not exist without my advisor, Ken Kennedy. He was most supportive professionally and understanding in personal issues, and he gave advice when it counted. Ridgway Scott, whose seminar on parallel computation at Penn State got me in touch with this matter originally, not only was excellent in advising my M.Sc. thesis, along with my academic advisor, Georg Schnitger; but he and his family (and his pool) convinced me that Houston, Texas, is indeed a good place to live. Together with John Mellor-Crummey, Chuck Koelbel, and Mary Wheeler they constituted a critical, helpful, and uniquely stimulating thesis committee. I would also like to thank IBM corp. for providing me with a generous fellowship, and the National Aeronautics and Space Administration and the National Science Foundation, who supported this work under grant #ASC-9349459.

Terry Clark was the one who provided me with most insights on real-world scientific programs and proved to be an excellent collaborator. Joel Saltz and Raja Das were inspiring partners for discussing the parallelization of irregular applications and, together with other members of their group, were of critical importance for the Fortran D implementation efforts. Special thanks also go to Seema Hiranandani and Chau-Wen Tseng, who developed the original Fortran D compiler, to Paul Havlak, whose symbolic analysis proved extremely valuable, and to the other D System developers, who made this implementation possible. Scott Baden, whose thesis raised my first interest in irregular applications, has since then continuously influenced my thinking on this matter.

My cheerful office mates Nat McIntosh and Uli Kremer never let me down; Nat in particular gave countless insights along the path to Unix wisdom. Kevin Cureton saved the world whenever it was broken and proved an able master of monster makefiles. Debbie Campbell mercifully took up the cause of getting my technical writing in shape. Our staff was always competent and extremely helpful; in particular, Ivy Jorgensen, Ken Marshall, and Sean Starke never tired of answering questions and keeping things running smoothly.

All of these people have not only improved the research environment, but also enriched life at Rice in general. However, there is also a long list of other characters I am indebted to for making graduate school the experience it has been. A very incomplete subset includes Ervan, organizer of canoeing, stargazing, tubing, kayaking, hiking, sailing and road trips; Rosana and Pete, who will hopefully keep up the brewing in DC; Steph and Jerry, happy hot-tub hosts whenever it got chilly; Don, whose bike was very difficult to hang on to; Saniya and Karim, whose Tunisian food was simply the best; Gregor and Kay, fine hiking partners on those winter treks in northern Germany; and David, who was not discouraged by tents ripping away if we "were lucky with the weather." And there is, of course, the Valhalla crew.

My deepest gratitude, however, is reserved for those to whom this thesis is dedicated: my parents and my wife.

Für Imke, Hildegard und Volkhard

Contents

Illustrations

Chapter 1

Introduction

Data-parallel languages, such as High Performance Fortran (HPF) [KLS+94] and Fortran D [FHK+90], allow for a "machine independent parallel programming style," in which the applications programmer uses a dialect of a sequential language and annotates it with high-level data distribution information. From this annotated program, data-parallel compilers will generate codes in different native Fortran dialects for different parallel architectures. The target architectures of these compilers include both shared- and distributed-memory architectures and both MIMD and SIMD machines. The overall goal for the code generated by the data-parallel compiler is to have a performance which is, for most applications, relatively close to the performance of hand-written native code.

Section 7.2.2 briefly describes the main concepts of the Fortran D language. A prototype Fortran D compiler targeting distributed-memory architectures has had considerable success with regular problems [HHKT92, HKK+91, HKT91, HKT92b, Tse93]. The goal of this dissertation is to extend the applicability of data-parallel compilers into the domain of irregular problems.

1.1 Irregular Problems

The definition of when an application is irregular varies:

- A physicist might classify a computational problem by the degree of geometric simplicity and the variance of density of the underlying physical problem. An example of a regular problem under this metric is the calculation of a simple wave equation for a homogeneous rectangular problem domain, whereas mapping out the gravity potential for an expanding galaxy is certainly of irregular nature.

- An applied mathematician who sees the mathematical description of a problem may consider the sparsity of the data describing a particular instance of the problem. Typical examples are finite-difference methods, which have a dense (regular) description, *vs.* finite-element methods, whose description is sparse due to the varying element sizes.

- A computer scientist is typically interested in the complexity of the data structures and access mechanisms needed to efficiently solve a problem; here we say "efficiently" because even irregular problems can usually be captured by very simple data structures, but not without wasting memory and/or processing power. Simple arrays, accessed directly, are typical for regular problems, whereas irregular applications may employ arrays with indirection vectors, pointers, linked lists, or quad trees, for example [SLY90].

We consider a problem to be irregular if its data access patterns are hard to analyze at compile time; *i.e.*, there is no obvious, simple parallelization and data distribution that gives good speedups and makes efficient use of processing power and the memory hierarchy. With increasing processor power opening the door to solving scientific problems that were previously impractical to solve ("Grand Challenges"), the relative importance of the already widespread irregular applications is expected to increase even further. Examples for areas of high interest are molecular dynamics, galaxy simulation, gene decoding, climate modeling, and computational fluid dynamics. Typical difficulties that arise from parallelizing irregular problems are summarized below:

Difficulty 1 Poor load balance, for example when modeling a rapidly changing physical system.

Difficulty 2 Lack of compile-time knowledge about where and which data have to be communicated, for example in Monte-Carlo processes.

Difficulty 3 Limited locality, for example when computing long-range interactions between particles.

Difficulty 4 Large communication requirements, for example when simulating many timesteps of a relatively small, but dynamic system.

1.2 Previous Results

The compiler support level is the focus of this thesis. A principal reason for developing powerful compilers is to shift responsibilities for tedious low-level details away from the programmer. This is typically associated with a tradeoff between abstraction and performance, which is unfortunate but can be justified to some degree. However, there also seems to be a fine line between a compiler being powerful and helpful, for example by assisting the programmer in dealing with machine-specific details, and a compiler trying to be too smart and getting in the way of the programmer. The virtual-machine model used by CM FORTRAN [BHMS91] can be seen as a typical example of the latter [Chr91], as explained in more detail in Chapter 4.

A programmer should not have to make this tradeoff when choosing a compiler, especially in a performance-oriented field such as scientific parallel computing. A compiler should try to assist the user in making some decisions, but it should also provide the user with convenient mechanisms to guide or override the compiler; such mechanisms are particularly important considering that parallelizing compiler technology is still in its infancy. Citing from a study about parallelizing different applications (including a molecular dynamics simulation) using the FX/FORTRAN parallelizing compiler [SH91]:

> It is worth noting that the available directives were sometimes found to be restrictive or incapable of expressing the exact information we wished to convey to the compiler.

This observation has led to a trend away from completely automatic parallelizing compilers, which try to extract parallelism from a sequential program without any user assistance, towards the development of more annotation-oriented languages, which try to give the user a convenient interface for indicating parallelism. This approach is similar to the power steering paradigm [KMT91] used for loop transformations, where the compiler cannot always pick the best transformation, but it assists the user by (conservatively) testing correctness and performing the actual rewriting work. Developing annotations for conveying information to the compiler at a high level is a part of this dissertation.

1.2.1 The inspector-executor paradigm

An important concept associated with communication optimization for applications using irregular array subscripts is the *inspector-executor* paradigm [MSS+88, KMV90, KMSB90, WSBH91]. A loop that contains indirect accesses to a distributed array is processed in four steps:

1. The *inspector* runs through the loop and only records which array elements are accessed, without doing the actual computation. *Communication schedules* are computed that satisfy the communication requirements induced by these access patterns.

2. A *gather* operation fetches all referenced off-processor data from their owners and buffers them locally.

3. The *executor* runs through the loop and performs the computation.

4. A *scatter* writes all off-processor data that have been defined in the loop back to their owners.

The inspector-executor paradigm has been shown to be very effective under certain circumstances and recently has been extended to general patterns of control flow [DSvH93, Das94].

1.2.2 Compilation systems for irregular problems

Projects that have aimed at least to some degree towards compiler support for parallelizing irregular problems are the following.

Kali

KALI [KMV90, MV90, KM91] is the first compiler system that supports both regular and irregular computations on MIMD distributed-memory machines. Programs written for KALI must specify a virtual processor array and assign distributed arrays to `BLOCK`, `CYCLIC`, or user-specified decompositions. Instead of deriving a computational decomposition from the data decomposition, KALI requires that the programmer annotates each parallel loop with an `ON` clause that maps loop iterations onto the processor array. Communication is then generated automatically based on the `ON` clause and data decompositions. An inspector/executor strategy as described in Section 1.2.1 is used for run-time preprocessing of communication for irregularly distributed arrays [KMSB90]. Major differences between KALI and FORTRAN D include KALI's mandatory `ON` clauses for parallel loops and FORTRAN D's support for alignment, collective communication, and dynamic decomposition.

ARF

ARF is another compiler based on the inspector-executor paradigm. ARF is designed to interface FORTRAN application programs with the PARTI run-time routines described in Section 7.1.2 [WSHB91]. It supports `BLOCK`, `CYCLIC`, and user-defined irregular decompositions. The goal of ARF is to demonstrate that inspector/executors based on PARTI primitives can be automatically generated by the compiler.

Fortran S

FORTRAN S [BKP93] is a variation on FORTRAN 77 that contains directives for explicit parallelism. *Conditioned Iterations Loops* [HPE94] amortize the cost of irregular data accesses on distributed shared memory machines by first inspecting each iteration of a loop on whether a processor owns the page of data associated with it, and then looping explicitly over these iterations.

1.2.3 Mapping arrays and mapping functions

To specify irregular mappings in a data-parallel context, index-based mapping arrays [WSBH91] or mapping functions [CMZ92] have been proposed. However, such arrays or functions that explicitly map indices to processors have to be provided by the programmer, even though she or he may not be interested in what exactly these mappings look like.

Another alternative is to replicate the data and distribute just the computation itself and combine results at the end. This approach has the further advantage of simplicity and robustness, and for relatively small problem sizes and numbers of processors it may actually result in satisfactory performance [CM90]. However, one of the main advantages of distributed-memory machines, scalability to large problem and machine sizes, has to be compromised under this approach. Alternatively, one may distribute the data in a way that considers the actual data dependences specific to their application [CHMS94]. These "irregular" mappings are typically harder to debug and manage than regular mappings and present an additional level of complexity beyond general message-passing style programming.

1.2.4 Value-based mappings

Value-based distributions were initially proposed as an enhancement to FORTRAN 77D [Han92]. A variant of it, based on a *GeoCoL* (**Geo**metrical, **Co**nnectivity and/or **L**oad) data structure, has since then been implemented in a FORTRAN 90D prototype compiler by Ponnusamy et al. [PSC93a, PSC+93b]. However, the GeoCoL structure still has to be managed explicitly by the programmer.

1.2.5 Communication analysis

Determining communication requirements and satisfying them efficiently is critical for any parallel program running on a distributed-memory machine. Eliminating redundant communication, message blocking and hoisting, and hiding communication delays are important optimizations, all of which are particularly difficult to perform for irregular problems. Our strategy for effective communication placement is based on an extensive data-flow framework.

Data-flow analysis is a common technique for reasoning at compile time about the run-time behavior of the program concerning variable definitions and uses. The bulk of the work in this field has treated all variables as scalars, resulting in a very conservative analysis for array variables. More precise methods are based on representations of array subsets, such as *data access descriptors* [Bal90] or *regular sections* [HK91].

The W2 compiler [GS90] for the Warp multiprocessor gathers information such as the set of definitions reaching a basic block to exploit the fine-grain parallelism offered by the highly pipelined functional units. It is based on interval analysis [All70, Coc70] and computes information with array region granularity.

Granston and Veidenbaum combine flow and dependence analysis to detect redundant global memory accesses in parallelized and vectorized codes [GV91]. They assume that the program is already annotated with READ/WRITE operations. Their technique tries to eliminate these operations where possible, also across loop nests and in the presence of conditionals.

What appeared to be lacking so far is a general approach towards analyzing the communication needs of a given program and determining when communication statements can be combined and hoisted. This dissertation contributes a data-flow framework which provides this analysis and furthermore gives specific treatment for the access patterns induced by irregular applications (see Chapter 3).

1.2.6 Evaluation

The area of compiling regular applications onto distributed-memory machines has become very active, and much progress has been made. The formation of the HIGH PERFORMANCE FORTRAN Forum, an ongoing standardization effort for commercial parallel FORTRAN compilers, is certainly an indication for this progress. HPF derives many of its underlying concepts from FORTRAN D, which is the base language chosen for the extensions proposed here, and other languages, such as VIENNA FORTRAN [BCZ92].

The body of work that focuses on irregular applications is much smaller. In particular, there are few attempts to directly exploit the characteristics of underlying applications (such as the positions of atoms in a protein); previous approaches are commonly based on access patterns (such as a pairlist directly indicating the interaction partners for each atom) and try to determine data decompositions and communication optimizations *after* the access patterns of the program have been determined.

A compiler cannot reasonably be expected to derive all locality aspects of the application underlying a given program. However, there appears to be a considerable potential for optimizations if the user has a convenient way to express locality information to the compiler. The main objectives of this dissertation are the design and evaluation of language extensions that provide such an interface and the development of the compiler analysis necessary to support these extensions.

1.3 The Thesis

The thesis of this dissertation is the following:

> *It is feasible and profitable to provide compiler support for the parallelization of scientific applications of an irregular nature to* directly *exploit the spatial locality of the underlying problem. An important component of this compiler support is the concept of* value-based mappings *which are derived from snapshots of the spatial configuration of the application. In combination with the distribution and alignment mechanisms provided in data-parallel languages such as* FORTRAN D*, value-based mappings are a practical and convenient handle for expressing both spatial locality and data interdependence.*

Here, as in the rest of this dissertation, the term *spatial locality* refers to the physical locality in the problem domain of the application (as opposed to locality of reference in an array, for example). Furthermore, the term *data interdependence* is used in a high-level sense, such as "the force between two particles depends on their distance" (as opposed to the field of dependence analysis which derives statement-ordering constraints based on definitions and uses of the same variables).

The goal for this dissertation is to validate this thesis and to show its effectiveness for handling irregular applications by extending the support for irregular problems within the Fortran D framework.

The rest of this thesis is organized as follows. Chapter 2 introduces value-based mappings, which are an extension of Fortran D to express spatial locality when distributing data across processors. Chapter 3 describes Give-N-Take, a code placement framework which the Fortran D compiler uses to place communication statements. Chapter 4 addresses some SIMD-specific issues when parallelizing irregular applications and develops the *loop-flattening* transformation. Chapter 5 describes practical aspects of the Fortran D implementation, followed by experimental results in Chapter 6. Chapter 7 provides a more comprehensive treatment of background and related work. Chapter 8 presents conclusions, open problems, and future work. In addition, Appendix A proves the correctness of the Give-N-Take framework from Chapter 3.

He was to leave inhabited districts behind
and begin a trek tailor-made for disaster,
given his propensity for getting lost at the best of times.
His guides, on taking their leave,
cheerfully advised him to recite the sacred texts from time to time
to avoid being eaten by snow leopards.

— Scott Berry (A Stranger in Tibet – The Adventures of a Zen Monk)

Chapter 2

Value-Based Mappings

Let us assume that the value assigned to some array element $a(i)$ depends on some other array element $b(j)$. In a regular problem, there will typically be some simple relationship between i and j; for example, they may be related to each other via a linear function known at compile time. This usually implies that at least this dependence can be satisfied, with little or no communication, when a and b are distributed by first partitioning their index spaces in some regular fashion among processors and then aligning them to each other in a certain way. We will refer to this characteristic as *index-based locality*. Exactly how arrays should be mapped in the presence of index-based locality is by no means trivial and still an active field of research [KMCKC93]. However, one can generally assume that only regular mappings (such as `BLOCK`, `CYCLIC`, or `BLOCK_CYCLIC`) and remappings need to be considered.

For irregular problems, this assumption cannot be made. Here subscripts i and j may each be determined by some complicated function or an array lookup, whose outcome is unknown at compile time. Furthermore, even if the compiler knew the values of the is and js, there would often still be no way to achieve good locality (*i.e.*, low communication requirements) via regular, index-oriented mappings. For example, if i and j are different vertices in a mesh, then $a(i)$ and $b(j)$ might depend on each other if i and j are linked together by an edge. Most meshes number their vertices in a way that does not directly reflect their topology; *i.e.*, it is hard to predict whether two vertices are linked together by just looking at their indices. Therefore, distributing mesh points and the computation associated with them across processors by dividing their index space in some regular fashion typically results in many off-processor accesses; *i.e.*, if a processor is assigned some $a(i)$ and therefore has to know the value of $b(j)$, chances are high that $b(j)$ will be on a different processor. The potential speedup gained from distributing data and computation across processors is likely to be lost by exceedingly high communication costs when using simple mapping schemes.

The fact that the vertex numbering does not reflect the mesh topology is an example of poor index locality *within* a data structure, which we consider a data *distribution* problem. However, we may also have bad locality *across* data structures. Revisiting the mesh example, we are typically operating on vertices and edges, which have a certain interrelationship (each edge has two particular vertices as end points). This relationship can usually not be determined from the node and edge numbering. However, it would generally be advantageous if the data associated with an edge would be stored on the same processor as the data associated with its end points; we consider this a data *alignment* problem.

Fortunately, many scientific applications lacking any index-based locality that a compiler might take advantage of do offer another kind of locality, which we will refer to as *value-based locality.* This kind of locality, which will be introduced in more detail in the next section, naturally lends itself to *value-based mappings.* These mappings can be used to improve locality and also to increase load balance.

The rest of this chapter is organized as follows. Section 2.1 introduces value-based locality and illustrates the use of value-based mappings with kernels taken from a molecular dynamics code and an unstructured mesh application. (Experimental results obtained for these kernels are contained in Section 6.1.) Section 2.2 lists the implications of value-based mappings for message-passing node programs. Section 2.3 describes language extensions and compiler technology for generating such node programs. (A comparison of the effectiveness of index-based and value-based mappings for these applications is presented in Section 6.1 as well.)

2.1 Value-Based Locality

In the presence of value-based locality, two array references $a(i)$ and $b(j)$ may not be related to each other via their indices i and j, but instead by their values $a(i)$ and $b(j)$, or by the values of other variables, such as $x(k)$ and $x(l)$, that in turn are related to $a(i)$ and $b(j)$ by their indices (for example, $k = i$ and $l = j$). In this context, "related" refers to a preference towards residing close to each other with respect to the memory hierarchy, *e.g.*, on the same processor. Revisiting the mesh example, x might be a coordinate array storing the physical location of each mesh point (assuming 1-D for simplicity). Then the probability that vertices i and j are connected increases as $|x(i) - x(j)|$ decreases. Note that this is not a strict relationship; whether an edge actually exists or not still depends on other factors, such as mesh density and topology. However, since data mapping is not a correctness but only an efficiency issue, we are usually more interested in a fast heuristic for finding a reasonably good data mapping than in a strictly optimal, but expensive solution.

Value-based mappings allow the programmer to express value-based locality as a simple extension of the regular mapping mechanism employed in data-parallel languages. We distinguish between value-based distributions and value-based alignments. For example, let array `x` be aligned to a decomposition `arrayD`. Then the directive `DISTRIBUTE arrayD(VALUE(x))` is a *value-based distribution* specifying that decomposition `arrayD` (and with it array `x`) should be distributed such that the values of the elements of `x` assigned to each individual processor are from disjoint intervals. (See Section 2.3.1 for a more formal description of the syntax proposed for value-based mappings.) In other words, if the values of two elements of some array elements `x(i)` and `x(j)` are close, then `x(i)` and `x(j)` are likely to be mapped to the same processor, no matter what relationship `i` and `j` may have to each other.

2.1.1 Molecular dynamics - An example for value-based distributions

Examples where data are not related by indices, but by values, are molecular dynamics programs such as GROMOS [GB88], CHARMM [BBO+83], or ARGOS [SM90] that are used to simulate biomolecular systems. One important routine common to these programs is the non-bonded force (NBF) routine, which typically accounts for the bulk of the computational work (around 90%). Figure 2.1 shows an abstracted version of a sequential NBF calculation. An important characteristic of NBFs is that their intensities decay very rapidly with increasing distance between the atoms involved. This locality is exploited by using a *cutoff radius*, R_{cut}, beyond which the NBF interactions are ignored. This reduces the number of atom pairs for which the NBF has to be computed and significantly reduces overall computational costs. Furthermore, we can exploit this locality when distributing data in a parallel implementation according to the values of x, which stores the physical atom coordinates. The atom numbering itself is not related to the coordinates; instead, they are typically numbered according to some data bank standard, such as the Brookhaven protein data bank, which considers amino acid types, peptide bonds, solute/solvent classifications, etc. – an ordering which is rather difficult for a compiler to take direct advantage of without user assistance. Therefore, a traditional, index-based mapping would lose this locality. In fact, since each processor needs for each owned atom to

$$
\begin{array}{l}
\textbf{do } i = 1, N_{atom} \\
\quad \textbf{do } p = 1, inb(i) \\
\quad\quad j = partners(i,p) \\
\quad\quad force = nbf\_func(x(i), x(j)) \\
\quad\quad f(i) = f(i) + force \\
\quad\quad f(j) = f(j) - force \\
\quad \textbf{enddo} \\
\textbf{enddo}
\end{array}
$$

Figure 2.1 Sequential version of the Non-Bonded Force kernel `nbf`. N_{atom} is the total number of atoms, $inb(i)$ is the number of atom *partners* that are close enough to atom i to be considered for the NBF calculation. For simplicity, the coordinate and force arrays x and f are shown only one-dimensional.

know about all atoms within R_{cut} of that atom, a regular mapping typically results in each processor accessing all data, with accordingly high communication and storage requirements. For example, Figure 2.2 shows the mapping resulting from distributing an SOD* system ($N_{atom} = 6968$) across eight processors.

The actual use of a value-based distribution can be seen in Figure 2.3, which shows a Fortran D implementation of the `nbf` kernel outlined in Figure 2.1. This program is a condensed and abstracted version of the 340-line Gromos subroutine `nonbal.f`, enhanced with some data initialization. Although this program is very simplified, it still presents similar difficulties as the original code with respect to the compiler. Fortran D directives first declare a decomposition `atomD` (line 10), then align coordinates `x`, forces `f`, partner counts `inb`, and adjacency lists `partners` with `atomD` (line 11), and finally distribute `atomD`. Note that initially this is a regular, `BLOCK`-wise distribution (line 12). After reading in the initial coordinates and partner counts (line 15), `atomD` gets redistributed according to the values of coordinate array `x` (line 18). Note also that the strict owner-computes rule is overridden in the NBF calculation itself via an `ON_HOME` directive (line 29). Figures 2.4 and 2.5 show the SOD mappings resulting from one- and three-dimensional value-based mappings, respectively.

*SOD (superoxide dismutase) is a catalytic enzyme that converts the toxic free-radical, O_2^{-4}, a byproduct of aerobic respiration, to the neutral molecules O_2 and H_2O_2 [WCSM93].

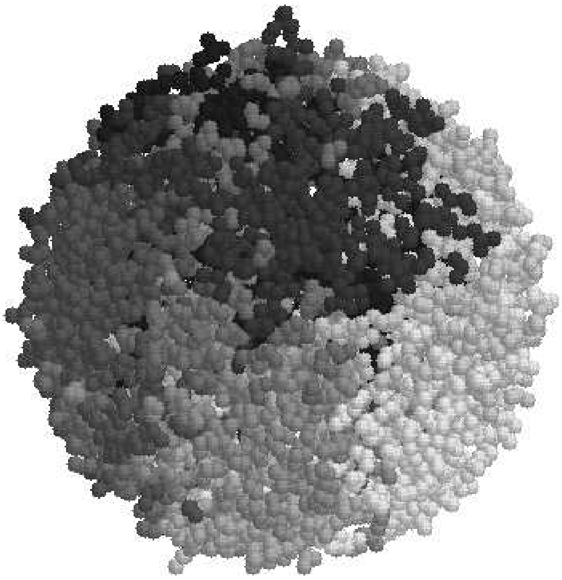


Figure 2.2 BLOCK mapping of an SOD system. Atoms are shaded according to the processor they are mapped to, for an eight-processor configuration. The regular, index-based mapping results in assigning each processor very irregular subdomains, with accordingly poor locality and load balance.

2.1.2 Unstructured meshes - An example for value-based alignments

An example of a *value-based alignment* is shown in Figure 2.6, which shows a FORTRAN D version of a sweep over the edges of an unstructured mesh [Mav91]. There are two decompositions, `nodeD` for the node data and `edgeD` for the edge data. After reading in the data (line 15), first `nodeD` gets distributed by value according to the node coordinates; that is, the values of `x` determine the mapping of node data indices onto processors (line 18). Secondly, `edgeD` gets aligned with `nodeD` according to the values of the topology arrays `ends1` and `ends2` (line 21); that is, the edge-index-to-node-index mappings given by `ends1` and `ends2` are combined with the node-index-to-processor mapping of `nodeD`. Note that since edges tend to connect nodes that are relatively close together, one would expect the two composed mappings, which correspond to the both endpoints of each edge, to agree in most cases. However, there may be conflicts in which case heuristics have to be used, which might for example keep load balancing in mind or simply toss a coin.

```
1        PROGRAM nbf
2
3        INTEGER i, j, p, t, n$proc, Natom, pMax, Nstep
4        PARAMETER (n$proc = 8)
5        PARAMETER (Natom = 8000, pMax = 250, Nstep = 30)
6        INTEGER inb(Natom), partners(Natom, pMax)
7        REAL x(Natom), f(Natom), force, nbf_func, delta_func
8
9   C    FORTRAN D directives
10       DECOMPOSITION atomD(Natom)
11       ALIGN inb, x, f, partners(i,j) WITH atomD(i)
12       DISTRIBUTE atomD(BLOCK)
13
14  C    Initialize data
15       CALL read_data(x, inb, partners)
16
17  C    Redistribute atomD according to coordinate values
18       DISTRIBUTE atomD(VALUE(DIM=1, VALS=x, WEIGHT=inb))
19
20  C    Loop over time steps
21       DO t = 1, Nstep
22
23  C       Reset forces to zero
24          DO i = 1, Natom
25             f(i) = 0
26          ENDDO
27
28  C       Computes forces
29          EXECUTE (i) ON_HOME f(i)
30          DO i = 1, Natom
31             DO p = 1, inb(i)
32                j = partners(i, p)
33                force = nbf_func(x(i), x(j))
34                f(i) = f(i) + force
35                f(j) = f(j) - force
36             ENDDO
37          ENDDO
38
39  C       Push atoms
40          DO i = 1, Natom
41             x(i) = x(i) + delta_func(f(i))
42          ENDDO
43       ENDDO
44       END
```

Figure 2.3 FORTRAN D version of the Non-Bonded Force kernel (with coordinates and forces shown 1-D for simplicity). During each of the `Nstep` time steps, forces are first reset to zero, then they are computed based on the distances between atoms (similar to the code in Figure 2.1), and finally the forces are used for updating coordinates. Note the index-based distribution directive in line 12, followed by a value-based redistribution in line 18.

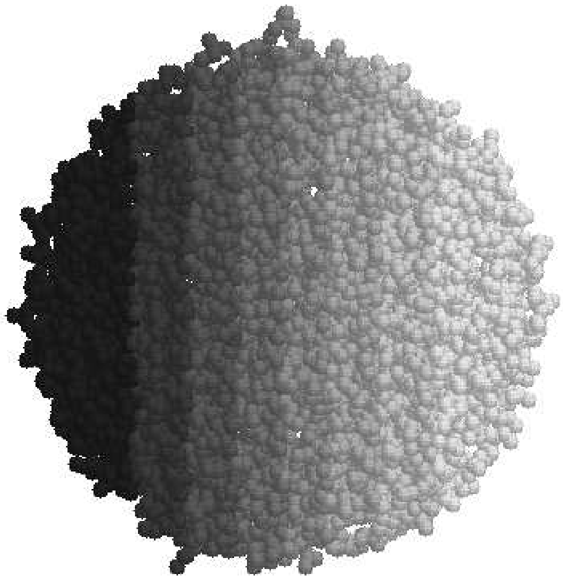


Figure 2.4 Value-based mapping of atoms along just one dimension. Locality is good, but communication may be expensive due to the high surface-to-volume ratio.

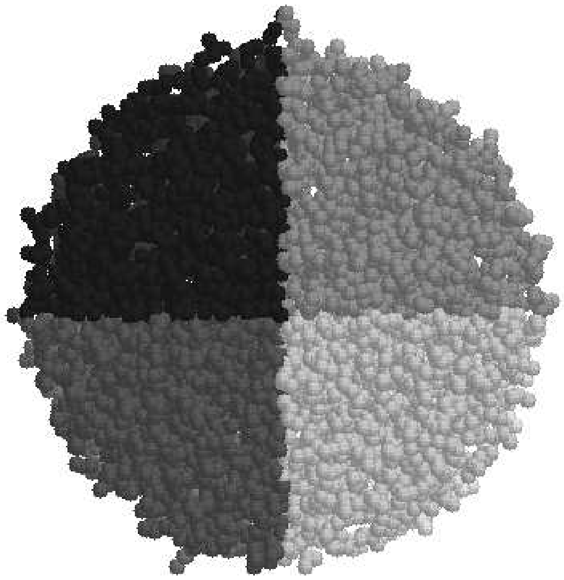
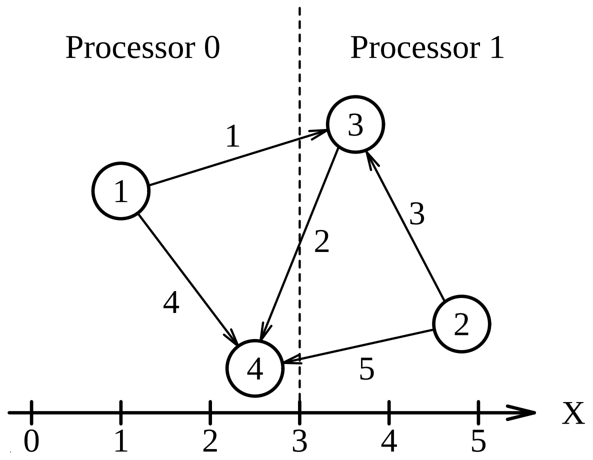


Figure 2.5 Value-based mapping of atoms along all three dimensions. Locality is good, and the compact subdomains minimize communication costs.

```
         PROGRAM mesh

         INTEGER i, n1, n2, n$proc, Nnodes, Nedges, Nstep, t
         PARAMETER (n$proc = 8, Nnodes = 10000, Nedges = 20000)
5        INTEGER ends1(Nedges), ends2(Nedges)
         REAL x(Nnodes), f(Nnodes), w(Nnodes), flux

   C     FORTRAN D directives
         DECOMPOSITION nodeD(Nnodes), edgeD(Nedges)
10       ALIGN f, w, x, WITH nodeD
         ALIGN ends1, ends2 WITH edgeD
         DISTRIBUTE nodeD(BLOCK), edgeD(BLOCK)

   C     Initialize data
15       CALL read_data(x, w, ends1, ends2)

   C     Redistribute nodeD according to coordinate values
         DISTRIBUTE nodeD(VALUE(DIM=1, VALS=x))

20 C     Redistribute edgeD according to nodeD
         ALIGN egdeD WITH nodeD(VALUE(DIM=2, VALS=ends1,ends2))

         Loop over time steps
         DO t = 1, Nstep
25
   C        Reset fluxes to zero
            DO i = 1, Nnodes
               f(i) = 0
            ENDDO
30
   C        Computes fluxes
            EXECUTE (i) ON_HOME ends1(i)
            DO i = 1, Nedges
               n1 = ends1(i)
35             n2 = ends2(i)
               flux = flux_func(w(n1),w(n2))
               f(n1) = f(n1) + flux
               f(n2) = f(n2) + flux
            ENDDO
40       ENDDO
         END
```

Figure 2.6 FORTRAN D kernel of a sweep over the edges of an unstructured mesh. The mesh coordinates are stored in the coordinate array `x` (again 1-D for simplicity), the topology is stored in the endpoint arrays `ends1` and `ends2`. After an initial index-based distribution of node and edge data in line 12, the node data are redistributed by value in line 18, and edge data are aligned by value with the redistributed node data in line 21.



i	x(i)	nodeD(i)	ends1(i)	ends2(i)	edgeD(i)
1	1.0	0	1	3	0
2	4.8	1	3	4	1
3	3.8	1	2	3	1
4	2.5	0	1	4	0
5	–	–	2	4	1

Figure 2.7 Example mesh with four nodes and five edges. The table shows its node coordinates (`x`), topology (`ends1`, `ends2`), and resulting mappings (`nodeD`, `edgeD`) for two processors.

A simple example where four nodes and five edges are mapped onto two processors is shown in Figure 2.7. Given the mesh code in Figure 2.6, the FORTRAN D compiler will generate code that at the location of the value-based mapping directive in line 18 will call a partitioner; see also Section 2.3.2. This partitioner will determine at run time a function which maps the values stored in `x`, which represent physical coordinates in space, to processors. A mapping function preserving physical locality and load balance would be, for example,

$$f_{\texttt{nodeD}}(x) = \begin{cases} 0 \text{ if } x \leq 3, \\ 1 \text{ if } x > 3. \end{cases}$$

This corresponds a distribution function which maps array indices to processors as follows:

$$\texttt{nodeD}(i) = \begin{cases} 0 \text{ if } \texttt{x}(i) \leq 3, \\ 1 \text{ if } \texttt{x}(i) > 3. \end{cases}$$

After mapping node data to processors, edge data are mapped (line 21), preferably to the same processors on which the node data of the endpoints of each edge are residing. As mentioned in the previous paragraph, there may be mapping conflicts if the endpoints of the same edge are on different processors. In Figure 2.7, edges are shown directed to distinguish between the endpoints given by `ends1` (tail) and `ends2` (head). In this example, let us assume for simplicity that such conflicts, which occur for edges 1, 2, and 5, are resolved by favoring the processor to which the tail node (`ends1`) is mapped to. This corresponds to a mapping function

$$\texttt{edgeD}(i) = \texttt{nodeD}(\texttt{ends1}(i)).$$

2.2 Implications of Value-Based Mappings

The prototype MIMD FORTRAN D compiler transforms a FORTRAN D program, written in a global name space and annotated with data mapping directives, into a message-passing program, which is a local name space node program including communication statements. This section describes the effect that distributing data by value has on a node program which may be generated by the compiler or coded by a programmer directly using message passing.

2.2.1 Specification and state

A regular distribution can be fully specified by a simple keyword (“`BLOCK`”), or a keyword enhanced with a small list of parameters (“`BLOCK_CYCLIC(blockSize)`”). There is very little state associated with a mapping, both at compile time and at run time. This already implies some simplicity for the programming assignment and for the resulting program. Many mapping-dependent decisions, such as the effect of an owner-computes rule (which links the mapping of computation to the mapping of the data involved), can already be resolved before run time. The amount of additional code and variables for computing and storing the mappings and for translating between global indices, local indices, and processor numbers is very small.

If an application has good index-based locality, then an irregular distribution may be used to improve load balance, but some or all of the mapping computation can still be done on the fly [BK93, Bia91, CHMS94]. Given a value-based distribution (*i.e.*, assuming *no* index-based locality), which for example distributes `x` according to its values, one could envision a scheme that also had very little state specifically devoted to representing the distribution. For example, one could compute the owner of some `x(i)` on the fly by sorting all elements of `x` and determining the position of `x(i)` in the sorted list. This, however, would clearly be impractical to do for each reference to an element of `x`. Instead, one should amortize the cost of determining ownership etc. by computing this information once and reusing it.

However, representing a value-based distribution explicitly requires a large amount of state. A *translation table* maps global indices i_{glob} into pairs (i_{loc}, p) of local indices and processor numbers. Often the translation table itself is too large to be fully replicated and is distributed instead [WSBH91]. Therefore, not only storing but also accessing the information adds complexity to the program. As also described in Section 7.1.2, run-time libraries such as CHAOS can take most of the complexity of this task from the programmer [DHU+93], but their use still requires explicit managing of the data structures associated with irregular distributions and communications.

2.2.2 Storing value-based distributed data

Again assuming that `x` is distributed according to its value, the number of elements of `x` assigned to each processor typically varies and is not known until run time. This poses particular problems when using a language that does not support dynamic memory allocation, such as FORTRAN 77. Common strategies for circumventing this restriction are to make arrays conservatively large or declare work arrays that are shared by several variables, both of which have obvious disadvantages.

Note that the same problem occurs when regularly distributed arrays are accessed irregularly and we wish to append buffer space for off-processor data at the end of the array [Han93].

2.2.3 Translating name spaces

Translating between the global name space of a FORTRAN D program and the local name space of the node program is an important component of the parallelization process. For regular mappings, most of this task can be performed before run time, or, if run-time translation is needed, the necessary code can be generated fairly easily; for example, a statement `x(i) = i`, where `i` is global, might be translated into something like `x(i) = i + my$proc*block_size`, with a local `i`.

For irregular mappings, the translation has to be delayed until run time, and computing the translation may be complicated.

2.2.4 Communication generation

Again due to the complicated relationship between global and local name spaces, generating correct communication statements in the presence of value-based mappings can be tricky for regular array accesses, such as `x(i) = x(i+1)`, and even more so for irregular references, such as `x(a(i)) = x(b(i))`.

Resolving such references requires several translation steps, some of which may themselves involve communication. To still generate efficient code, one should pre-compute and reuse as much of this information as possible. The *inspector-executor* paradigm (Section 1.2.1) allows us to message-vectorize low-locality data accesses, even in the absence of compile-time knowledge.

2.2.5 A bootstrapping problem

One characteristic of value-based mappings is that they may pose a certain bootstrapping problem to both the user and the compiler, as has also been identified by Ponnussamy *et al.* [PSC93a]. This problem occurs when an array is distributed based on its own values, which is considered perfectly legal, as is the case in the code shown in Figure 2.3. Here an array `x` is aligned to a decomposition `atomD`, which in turn gets distributed based on the values of `x`. When we start initializing `x`, we need to know its mapping function to assign each processor its share of array elements. This mapping function, however, cannot be determined until we know *all* values of `x`.

To resolve this problem, we start out with a different, typically regular mapping, which can be used for example to read in the data as is the case with the program shown in Figure 2.3. After the data relevant for the irregular mapping are known, the decomposition is remapped based on their values.

2.3 The Compiler's Perspective

The previous section outlined the general issues associated with distributing data based on values. This section addresses some of the implications of using value-based mappings in the context of a data-parallel language such as Fortran D. A more detailed discussion of the implementation aspects of value based mappings can be found in Chapter 5.

2.3.1 The input language

Since we implemented value-based mappings as part of a Fortran D compiler prototype, the Fortran D language constructs also serve as a basis for the syntax of value-based mappings. These extensions could also be applied directly to the HPF standard [KLS+94]. We were able to limit ourselves to a simple extension of the already existing `DISTRIBUTE` and `ALIGN` directives, as was also illustrated by the code in Figure 2.3. Here the directive `DISTRIBUTE atomD(VALUE(DIM=1, VALS=x, WEIGHT=inb))` was the only statement the programmer had to add in order to express the value-based locality of the application; the rest was done by the compiler. The value-based mapping syntax currently supported is shown in Figure 2.8.

Note that the range of available mapping strategy depends more on the available run-time support than on the compiler. In fact, the strategy specified by the user might be passed through verbatim to the run-time library. However, one might still require a certain minimal set of strategies to be always available [Han89, PSC93a].

Note also that the user may not select a specific strategy for distributing data explicitly, as shown in Figure 2.3. In this case the compiler chooses a default strategy, such as *recursive bisection* [BP90] or *spacefilling curves* [PB94]. Furthermore, since the default number of dimensions is one and some key words are optional, the `DISTRIBUTE atomD(VALUE(DIM=1, VALS=x, WEIGHT=inb))` could be abbreviated as `DISTRIBUTE atomD(VALUE(x,inb))`.

DistDirective	**is**	`DISTRIBUTE` *ValMapping*
AlignDirective	**is**	`ALIGN` *decomposition-name* `WITH` *ValMapping*
ValMapping	**is**	*decomposition-name* (`VALUE` (*ValArrays* [, *Weight*] [,*Strategy*]))
ValArrays	**is**	[`DIM =` *num-dims*,] [`VALS =`] *val-array-name* [, *val-array-name*] ...
Weight	**is**	[`WEIGHT =`] *weight-array-name*
Strategy	**is**	[`STRATEGY =`] *mapping-strategy-name*
Constraint:		Number of *val-array-names* must be one, or, if specified, *num-dims*.

Figure 2.8 Syntax of the value-based mapping directive.
Square brackets indicate optional components.

Naturally, there are several possible modifications/extensions for this syntax, which was held deliberately simple. For example, the value-based alignments could also be expressed as just another form of redistributions instead. Or, one could allow multidimensional value arrays instead of several one-dimensional arrays; for example, a program may store three-dimensional physical coordinates in one two-dimensional array, `x(3, n)`, instead of using three one-dimensional arrays, `x(n)`, `y(n)`, `z(n)`.

2.3.2 When to distribute and align

In general, mapping directives can be viewed as either static declarations (such as the HPF `DISTRIBUTE`), or as executable statements (such as the HPF `REDISTRIBUTE`). There are several reasons why value-based mappings should be viewed as executable, for example because of the bootstrapping problem described in Section 2.2.5, or because the values relevant for the mapping might change for dynamic problems and we might want to remap periodically.

A resulting question is *when* this mapping should be performed, whether it should be done when the execution reaches the directive, or instead at some other point in the program determined by the compiler. In the latter case, one might for example envision a scheme that lets the compiler analyze where relevant values are defined (or redefined) and where mapped data are used. However, the gain in programming convenience appears to be only marginal, and this kind of analysis seems to be fairly difficult and problem dependent; for example, we might not really want to redistribute whenever one relevant value changes. Therefore, the current implementation performs the value-based mappings at the location corresponding to the mapping directive (see also Section 5.3.2).

Value quietness, in which one has no wandering desires at all
but simply performs the acts of his life without desire,
that seems the hardest.

— Robert M. Pirsig (Zen and the Art of Motorcycle Maintenance)

Chapter 3

Balanced Code Placement with Give-N-Take

The previous chapter introduced the concept of value-based mappings to provide an efficient mechanism for specifying irregular data distributions with good locality and equalized work loads. However, in most cases even the best distribution results in some off-processor accesses; *i.e.*, the program requires communication. In the `nbf` example program in Figure 2.3, the communication steps are gathering coordinates and scattering forces. A performance-critical task of the compiler is to minimize the overheads associated with communication by generating and placing the communication statements judiciously. This requirement motivated the development of the GIVE-N-TAKE data-flow framework described in this chapter. As it turns out, however, communication placement is only one of the possible application of this framework, which for example can be used for any code placement task that is traditionally solved by Partial Redundancy Elimination (PRE).

The rest of this chapter is organized as follows. Section 3.1 revisits PRE and analyzes some of its current limitations. Section 3.2 introduces the communication generation problem, which will be used as an illustrating example application of GIVE-N-TAKE. Section 3.3 provides further intuition for the GIVE-N-TAKE framework and some background on the type of flow graph and neighbor relations used by the GIVE-N-TAKE equations. Section 3.4 states the actual equations and argues informally for their correctness and efficiency. Section 3.5 gives an efficient algorithm for solving the GIVE-N-TAKE equations. Section 3.6 concludes with a brief summary. Formal correctness proofs of GIVE-N-TAKE can be found in Appendix A.

3.1 Partial Redundancy Elimination

PRE is a classical optimization framework for moving and placing code in a program. Example applications include common subexpression elimination, loop invariant code motion, and strength reduction. The original PRE framework was developed by Morel and Renvoise [MR79] and has since then experienced various refinements [JD82, DS88, Dha88a, Dha91, DRZ92, KRS92]. However, the PRE frameworks developed to date still have certain limitations, which become apparent when trying to apply them to more complex code placement tasks.

Atomicity: PRE implicitly assumes that the code fragments it moves, generates, or modifies are atomic in that they need only a single location in the program to be executed. For example, when placing the computation of a common subexpression, PRE will specify only one location in the program, and code will be generated at that location to perform the entire computation. Later optimizations may then reschedule the individual instructions, for example to hide memory access delays, but PRE itself does not provide any such mechanism.

Ignoring side effects: Taking again the example of common subexpression elimination, classical PRE assumes that each common subexpression has to be computed somewhere; *i.e.*, nothing "comes for free." However, there are problems where side effects of other operations can eliminate the need for actual code placement. For example, when placing register loads and stores, certain loads may become redundant with previous definitions. This is generally treated as a special case, for example by developing different, but interdependent sets of equations for loads and stores [Dha88b].

Pessimistic loop handling: One difficulty with flow analysis has traditionally been the treatment of loop constructs that allow zero-trip instances, such as a Fortran **do** loop. Hoisting code out of such loops is generally considered *unsafe*, as it may introduce statements on paths where they have not existed before. However, unless the computation to be moved may change the meaning of the program, for example by introducing a division by zero, we often would like to hoist computation out of such loops even if the number of iterations is not known at compile time.

Give-N-Take aims to overcome these limitations in a general context. It is applicable to a broad class of code generation/placement problems, including the classical domains of PRE techniques as well as memory hierarchy related problems, such as prefetching and communication generation. Give-N-Take is subject to a set of correctness and optimality criteria as described in Section 3.3.2; for example, each consumption *must* be preceded by a production, and any generated code *should* be executed as infrequently as possible. However, the solutions computed by GIVE-N-TAKE vary depending on which kind of problem it is applied to. In a BEFORE problem, items have to be produced before they are needed (*e.g.*, for fetching an operand), whereas in an AFTER problem, they have to be produced afterwards (*e.g.*, for storing a result). Intuitively, one can think of an AFTER problem as a BEFORE problem with reversed flow of control.

Orthogonally we can classify a problem as EAGER when it asks for production as early as possible (*e.g.*, sending a message), or as LAZY when it wants production as late as possible (*e.g.*, receiving a message); this definition assumes a BEFORE problem. For an AFTER problem, "early" and "late" have to be interchanged since we are reversing the graph. Classical PRE, for example, can be classified as a LAZY-BEFORE problem. This means that the same framework can be used for different flavors of problems; there are no separate sets of equations for loads and stores [Dha88b], or for READs and WRITEs [GV91].

3.2 A Code Placement Example Problem: Communication Generation

An example of code placement is the generation of communication statements when compiling data-parallel languages, such as HIGH PERFORMANCE FORTRAN [KLS+94] or FORTRAN D [HKT92a]. For example, a processor of a distributed-memory machine may reference owned data, which by default reside on the processor, as well as non-owned data, which reside on other processors. Local references to non-owned data induce a need for communication, in this case a fetch of the referenced data from other processors. We will refer to such a fetch of non-owned data as a READ operation. Figure 3.1 shows another, simple example node code containing references to distributed data.

Since generating an individual message for each datum to be exchanged would be prohibitively expensive on most architectures, optimizations such as message vectorization, latency hiding, and avoiding redundant communication are crucial for achieving acceptable performance [HKT92b]. The profitability of such optimizations depends heavily on the actual machine characteristics; however, even for machines with low latencies or shared-memory architectures, the performance can benefit from maximizing reuse and minimizing the total number of shared data accesses.

do $i = 1, N$
    $y(i) = \ldots$
enddo
if *test* **then**
    do $j = 1, N$
        $z(j) = \ldots$
    enddo
    do $k = 1, N$
        $\ldots = x(a(k))$
    enddo
else
    do $l = 1, N$
        $\ldots = x(a(l))$
    enddo
endif

Figure 3.1 An instance of the communication placement problem, where the array x is assumed to be distributed or shared. Each reference to x in the k and l loops necessitates a global READ operation, whereby a processor referencing some element of x receives it from its owner. Possible communication placements are shown in Figure 3.2.

```
do i = 1, N
    y(i) = ...
enddo
if test then
    do j = 1, N
        z(j) = ...
    enddo
    do k = 1, N
        READ_Send{x(a(k))}
        READ_Recv{x(a(k))}
        ... = x(a(k))
    enddo
else
    do l = 1, N
        READ_Send{x(a(l))}
        READ_Recv{x(a(l))}
        ... = x(a(l))
    enddo
endif
```

```
READ_Send{x(a(1:N))}
do i = 1, N
    y(i) = ...
enddo
if test then
    do j = 1, N
        z(j) = ...
    enddo
    READ_Recv{x(a(1:N))}
    do k = 1, N
        ... = x(a(k))
    enddo
else
    READ_Recv{x(a(1:N))}
    do l = 1, N
        ... = x(a(l))
    enddo
endif
```

Figure 3.2 Possible communication placements for the code in Figure 3.1. A naïve code generation, shown on the left, results in a total of N messages to be exchanged, without any latency hiding. The solution provided by GIVE-N-TAKE, shown on the right, needs just one message and uses the i loop for latency hiding. $x(a(k))$ and $x(a(l))$ can be recognized as identical based on the subscript value numbers.

Figure 3.2 compares two possible communication placements for the example from Figure 3.1. The solution on the left places a $\text{READ}_{Send}/\text{READ}_{Recv}$ pair immediately before each reference. We will refer to such a solution as a *naïve* solution. Note that the GIVE-N-TAKE solution shown on the right would generally be considered unsafe, since for $N < 1$ the loops would not be executed. In the communication generation problem, however, we would generally rather accept the risk of slight over-communication than not hoist communication. Furthermore, it is often the case that non-execution of a loop also means that no communication needs to be performed; in the example, $N < 1$ implies $x(a(1:N)) = \emptyset$.

Note that the examples shown in this chapter do not include data declarations, initializations, distribution statements, etc. The communication statements are in a high level format that does not include any processor ids, schedule parameters, message tags, and so on. Communication schedule generation, which is a non-trivial problem in itself [HKK+92], and the conversion from global to local name space are also excluded. These and other implementation details on the usage of GIVE-N-TAKE for communication generation, such as the value number based data-flow universe, are described in Chapter 5.

3.2.1 The model

One must realize that both the hardware architecture and the operating system influence the exact nature of the communication problem and its solution. We will use the following model.

1. Communication can be one-to-one, one-to-many, many-to-one, or many-to-many.
2. Both the sending and the receiving processor(s) must issue matching communication operations.
3. If a processor p reaches a (lhs or rhs) reference requiring communication involving some other processor q, then q also has to reach this reference.

The last requirement facilitates fulfillment of the second requirement. For example, in Figure 3.3, assume that $x(10)$ is owned by processor 0, and that *test* evaluates to true on processor 1. Then processor 1 issues a receive statement, which processor

if *test* **then**
  $a = x(10)$
endif

if *test* **then**
  $\boxed{\text{READ}_{Send}\{x(10)\}}$
  $\boxed{\text{READ}_{Recv}\{x(10)\}}$
  $a = x(10)$
endif

Figure 3.3 A potentially illegal instance of the communication placement problem (left) and its solution (right). Array x is assumed to be distributed or shared. If on some processor *test* evaluates to true but not on the processor owning $x(10)$, then the third requirement of our communication model is violated.

0 has to match with a send statement. This is only guaranteed if and only if *test* evaluates to true on processor 0 as well.†

Another way of viewing the model is expressed in the following definition.

Definition 3.1 A program is a *valid instance of the communication generation problem* if its naïve solution results in a correct program; *i.e.*, the program annotated with $\text{READ}_{Send}/\text{READ}_{Recv}$ pair immediately preceding each non-local reference is deadlock-free.

Determining whether a program is a valid in the above sense is by no means trivial. In many cases the compiler could guarantee validity, but in other cases validity depends on run-time values. One obvious way of enforcing validity is to have all processors follow the same control flow, including that all conditionals have to evaluate identically across processors. This, however, would be overly restrictive; we do not require that all processors involved in a communication stemming from a particular reference reach that reference along the *same* path. Strictly speaking, even applying loop bounds reduction to a program can be viewed as carving up the control flow

†It actually gets even more complicated than this, since even though the receiver typically does not have to know whom to receive data from, the sender usually has to know where to send the data. An example for this is the NX operating system running on the intel iPSC series of MIMD distributed memory computers. Therefore, the processor owning $x(10)$ in the example from Figure 3.3 not only has to know whether *some* processor needs $x(10)$, but it also has to know *which* processors need this datum. However, we regard correctness in such cases to be outside of the scope of the GIVE-N-TAKE-framework and make it the responsibility of the code generation phase or the programmer.

between different processors and having processors following a different path through the program, since then each processor executes a different subset of the original set of iterations through the loop whose bounds are reduced even though the number of iterations may be the same on each processor.

If we do not use a strict owner-computes rule [CK88], then non-owned data may not only be locally referenced, but also locally defined. We assume that these data have to be written back to their owners before they can be used by other processors, as shown in Figure 3.4. (An alternative would be the direct exchange between a non-owner that writes data and another non-owner that reads them [Gup92, GS93]. This could also be accommodated by GIVE-N-TAKE, but especially in the presence of indirect references it would result in more complicated code generation.) A naïve solution would place a $\text{WRITE}_{Send}/\text{WRITE}_{Recv}$ pair immediately after each reference.

3.2.2 Previous work

Dependence analysis can guide communication optimizations, for example by guaranteeing the safety of hoisting communication out of a loop nest. However, dependence analysis alone is not powerful enough to take advantage of all optimization opportunities, since it only compares pairs of occurrences (*i.e.*, references or definitions) and does not take into account how control flow links them together. Therefore, combinations of dependence analysis and PRE have been used, for example for determining reaching definitions [GS90] or performing scalar replacement [CK92]. For example, Duesterwald *et al.* incorporate iteration distance vectors (assuming regular array references) into an array-reference data-flow framework, which is then applied to memory optimizations and controlled loop unrolling [DGS93].

Several researchers have addressed the communication generation problem, although often restricted to relatively simple array reference patterns. Amarasinghe and Lam optimize communication generation using Last Write Trees [AL93]. They assume affine loop bounds and array indices, they do not allow loops within conditionals as shown in Figure 3.1. Gupta and Schonberg use Available Section Descriptors, computed by interval based data-flow analysis, to determine the availability of data on a virtual processor grid [GS93]. They apply (regular) mapping functions to map this information to individual processors and list redundant communication elimination and communication generation as possible applications. Granston and Veidenbaum combine dependence analysis and PRE to detect redundant global memory accesses

```
if test then
   do i = 1, N
      x(a(i)) = ...
   enddo
   do j = 1, N
      ... = x(j + 5)
   enddo
endif
do k = 1, N
   ... = x(k + 5)
enddo
```

```
if test then
   do i = 1, N
      x(a(i)) = ...
   enddo
   WRITE_Send{x(a(1 : N))}
   WRITE_Recv{x(a(1 : N))}
   READ_Send{x(6 : N + 5)}
   READ_Recv{x(6 : N + 5)}
   do j = 1, N
      ... = x(j + 5)
   enddo
else
   READ_Send{x(6 : N + 5)}
   READ_Recv{x(6 : N + 5)}
endif
do k = 1, N
   ... = x(k + 5)
enddo
```

Figure 3.4 Example of a code with local definitions of potentially non-owned data (left), and a corresponding placement of global WRITEs (right).

in parallelized and vectorized codes [GV91]. Their technique tries to eliminate these operations where possible, also across loop nests and in the presence of conditionals, and they eliminate reads of non-owned variables if these variables have already been read or written locally. However, they assume atomicity, and they also assume that the program is already annotated with read/write operations; they do not try to hoist memory accesses to less frequently executed regions.

While these works address many important aspects of communication generation that are outside the scope of GIVE-N-TAKE itself, such as name space mappings or regular section analysis, they do not seem to be general and powerful enough with respect to communication *placement*. In the following, it is this aspect that we will focus on.

3.3 The Give-N-Take Framework

The basic idea behind the GIVE-N-TAKE framework is to view the given code generation problem as a producer-consumer process. In addition to being produced and consumed, data may also be destroyed before consumption. Furthermore, whatever has been produced can be consumed arbitrarily often, until it gets destroyed.

Data-flow frameworks are commonly characterized by a pair $\langle L, F \rangle$, where L is a meet semilattice and F is a class of functions; see Marlowe and Ryder [MR90] for a discussion of these and other general aspects of data-flow frameworks. Roughly speaking, L characterizes the solution space (or universe) of the framework, such as the set of common subexpressions or available constants, and their interrelationships. F contains functions that operate on L and compute the desired information about the program. Together with a flow graph consisting of nodes, edges, and a root and a mapping from graph nodes or edges to F, this framework constitutes a data-flow problem, which can be solved to analyze and optimize a certain aspect of a specific program. However, since GIVE-N-TAKE is not restricted to a specific lattice, we will focus mostly on F, the class of functions that we use to propagate information about consumption and production through a given program.

3.3.1 Communication placement with Give-N-Take

The problem of generating READs can be interpreted as a BEFORE problem as follows:

- Each reference to non-owned data *consumes* these data.

- Each READ operation, where a processor p sends data that it owns to another processor q that receives and references these data, *produces* the data sent.
- Each non-local definition (*i.e.*, a definition on another processor) of non-owned data *destroys* these data.

To split each READ into a READ_{Send}, the send issued at the owner, and a READ_{Recv}, the corresponding receive at the referencing processor, we need both the EAGER and the LAZY solution of the framework. We want to send as early as possible and receive as late as possible; since this is a BEFORE problem, the READ_{Send}s will be given by the EAGER solution, and the READ_{Recv}s will be the LAZY solution.

For placing global WRITEs, the non-owned definitions can be viewed as consumers, just as non-owned references, and we have to insert producers which in this case communicate data back to their owners instead of from their owners. Since we want to write data after they have been defined, this is an AFTER problem. Note that in this scenario, the previous problem of analyzing communication for non-owned references can be modified to take advantage of non-owned definitions if they are later locally referenced; *i.e.*, non-owned definitions can also be viewed as statements that produce non-owned references as a side effect "for free." potentially saving unnecessary communication to and from the owner. Again, we can split each WRITE into a WRITE_{Send}, given by the LAZY solution (since WRITE is an AFTER problem), and a WRITE_{Recv}, which is the EAGER solution.

3.3.2 Correctness and optimality

Given a program with some pattern of consumption and destruction, our framework has to determine a set of producers that meet certain correctness requirements and optimality criteria. The requirements that GIVE-N-TAKE has to meet to be correct are the following (with their specific implications when applied to communication generation):

(C1) *Balance*: If we compute both the EAGER and the LAZY solution for a given problem, then these solutions have to match each other; see Figure 3.5. (For each executed READ_{Send}, exactly one matching READ_{Recv} will be executed, and vice versa; similarly for WRITE_{Send}s and WRITE_{Recv}s.)

Figure 3.5 Left: unbalanced production, where one EAGER(X) production is followed by an arbitrary number of LAZY(X) productions. Right: possible solution obeying correctness criterion C1.

Figure 3.6 Left: unsafe production. Right: possible solution obeying C2.

Figure 3.7 Left: insufficient production.
Right: possible solution obeying C3.

(C2) *Safety*: Everything produced will be consumed; see Figure 3.6. (No unnecessary READs or WRITEs. In our specific case, this is more an optimization than a correctness issue.)

A special case are zero-trip loop constructs, such as a Fortran **do** loop. GIVE-N-TAKE tries to hoist items out of such loops, unless explicitly told otherwise on a general or case-by-case basis; see also Section 3.4.1.

(C3) *Sufficiency*: For each consumer at node n in the program, there must be a producer on each incoming path reaching n, without any destroyer in between; see Figure 3.7. (All references to non-owned data must be locally satisfiable due to preceding READs or local definitions, without intervening non-local definitions, and all definitions of non-owned data must be brought back to their owners by WRITEs before being referenced non-locally or communicated by a READ.)

The optimization criteria, subject to the correctness constraints stated above, are:

(O1) Nothing produced already (and not destroyed yet) will be produced again; see Figure 3.8. (Nothing will be recommunicated, unless it has been non-locally redefined.)

(O2) There are as few producers as possible; see Figure 3.9. (Communicate as little as possible.)

(O3) Things are produced as early as possible for EAGER-BEFORE and LAZY-AFTER problems; see Figure 3.10. (Send as early as possible.)

(O3′) Things are produced as late as possible for LAZY-BEFORE and EAGER-AFTER problems; see Figure 3.11. (Receive as late as possible.)

Note that while the correctness criteria are treated as strict requirements that GIVE-N-TAKE must fulfill, the optimality criteria are viewed more as general guidelines and are phrased correspondingly vaguely. A proof that GIVE-N-TAKE does indeed obey the correctness criteria can be found in Appendix A.

3.3.3 Zero-trip loop constructs

One difficulty with flow analysis has traditionally been the treatment of zero-trip loop constructs, such as a Fortran **do** loop. We are interested in hoisting computation

Figure 3.8 Left: redundant production.
Right: possible solution obeying O1.

Figure 3.9 Left: too many producers.
Right: possible solution obeying O2.

Figure 3.10 Left: too late production.
Right: possible solution obeying O3.

Figure 3.11 Left: too early production.
Right: possible solution obeying O3′.

out of such loops as well, but this may introduce statements on paths where they have not existed before, which is generally considered unsafe (criterion C2). Several techniques exist to circumvent this difficulty, for example adding an extra guard and a preheader node to each loop [Sor89], explicitly introducing zero-trip paths [DK83], or collapsing innermost loops [HKK+92]. These strategies, however, result in some loss of information, and they do not fully apply to nested loops. Therefore, the Give-N-Take framework generally treats a loop as if it will be executed at least once. In case this approach is not valid as such for a particular application of the framework, there are several relatively simple refinements to guarantee safety:

- The compiler can try to prove that a loop l will be executed at least once.
- Hoisting a statement S out of a loop l can be prohibited by adding S to STEAL(l); see Section 3.4.2.
- S may implicitly be void in case l does not execute; for example, if l has n iterations and S is a statement communicating $x(1{:}n)$, then $n \leq 0$ results in an empty statement.
- We can explicitly guard S by a test whether l will be executed.
- S might be a statement that results in some extra, but harmless computation, such as an unnecessary communication statement. In this case, we might be willing to pay that extra cost if it is amortized as soon as l is executed at least once.

- In case we rely on a statement S to be executed within l, for example, to bring in some not-owned data that are needed outside of l as well, we can add a test after the loop that explicitly executes S in case l is empty. These data are indicated in the framework by $\mathsf{GIVE}(l) - \mathsf{GIVE}_{init}(l)$.

3.3.4 The Interval-Flow Graph

A general data-flow analysis algorithm that considers loop nesting hierarchies is interval analysis. It can be used for forward problems, such as available expressions [All70, Coc70], and backward problems, such as live variables) [Ken71], and it has also been used for code motion [DP93] and incremental analysis [Bur90]. We are using a variant of interval analysis that is based on Tarjan intervals [Tar74]. Like Allen-Cocke intervals, a Tarjan interval $T(h)$ is a set of control-flow nodes that corresponds to a loop in the program text, entered through a unique header node h, where $h \notin T(h)$. However, Tarjan intervals include only nodes that are part of this loop; *i.e.*, together with their headers they form nested, strongly connected regions. Allen-Cocke intervals include in addition all nodes whose predecessors are all in $T(h)$; *i.e.*, they might include an acyclic structure dangling off the loop. In that sense, Tarjan intervals reflect the loop structure more closely than Allen-Cocke intervals [RP86]. Note that a node nested in multiple loops is a member of the Tarjan interval of the header of each enclosing loop.

Unlike in classical interval analysis, we do not explicitly construct a sequence of graphs in which intervals are recursively collapsed into single nodes. Instead, we operate on one *interval-flow graph* $G = (N, E)$, with nodes N and edges E. $\textsc{Root} \in N$ is the unique root of G, which is viewed as a header node for the entire program. For $n \in N$, $\textsc{Level}(n)$ is the loop nesting level of n, counted from the outside in; $\textsc{Level}(\textsc{Root}) = 0$.

We define $T(n) = \emptyset$ for all non-header nodes n, and $T^+(n) = T(n) \cup \{n\}$ for all nodes n. We also define $\textsc{Children}(n)$ to be the set of all nodes in $T(n)$ which are one level deeper than n; $\textsc{Children}(n) = \{c \mid c \in T(n), \textsc{Level}(c) = \textsc{Level}(n) + 1\}$. For each $m \in \textsc{Children}(n)$, we define $J(m)$ to be the immediately enclosing interval, $T(n)$.

One of the main differences between G and a standard control-flow graph is the way in which edges $e = (m, n) \in E$ are constructed and classified. In addition to edges that correspond to actual control-flow edges, E may also contain $\textsc{Synthetic}$

edges, which connect the header h of an interval $T(h)$ to all sinks (excluding $T^+(h)$) of edges originating within $T(h)$; *i.e.*, E will have SYNTHETIC edges if it contains jumps out of loops. Each non-SYNTHETIC edge (m, n) is classified as having one of the following types, as also illustrated by the example in Figure 3.13.

ENTRY: An edge from an interval header to a node within the interval; $n \in T(m)$.

CYCLE: An edge from a node in an interval to the header of the interval; $m \in T(n)$.

JUMP: An edge from a node in an interval to a node outside of the interval that is not the header node; $\exists h : m \in T(h), n \notin T^+(h)$. This corresponds to a jump out of a loop.

FLOW: An edge that is none of the above; $\forall h : m \in T(h) \Longleftrightarrow n \in T(h)$.

We also define HEADER$(n) = m$ if n is the sink of an ENTRY edge originating in m; otherwise, HEADER$(n) = \emptyset$.

Note that CYCLE and JUMP edges correspond to Tarjan's cycle and cross edges, respectively [Tar74]. However, we divide his forward edges into FLOW and ENTRY edges depending on whether they enter an interval or not (others divide them into forward and tree edges depending on whether they are part of an embedded tree or not [CLR90]). Note also that for each JUMP edge (m, n), G contains LEVEL(m) − LEVEL(n) SYNTHETIC edges, one from the header of each interval jumped out of.

GIVE-N-TAKE requires G to have the following properties:

- G is *reducible*; *i.e.*, each loop has a unique header node. This can be achieved, for example, by node splitting [CM69].
- For each non-empty interval $T(h)$, there exists a unique $n \in T(h)$ such that $(n, h) \in E$; *i.e.*, there is only one CYCLE edge out of $T(h)$. We will refer to node n as LASTCHILD(h).
- There are no *critical edges*, which connect a node with multiple outgoing edges to a node with multiple incoming edges. This can be achieved, for example, by inserting *synthetic nodes* [KRS92]. Code generated for synthetic nodes would reside in newly created basic blocks, for example a new **else** branch or a landing pad for a jump out of a loop.

```
   do i = 1, N
      y(a(i)) = ...
      if test(i) goto 77
   enddo
   do j = 1, N
      ...
   enddo
77 do k = 1, N
      ... = x(k + 10) + y(b(k))
   enddo
```

Figure 3.12 Example code. We wish to use the j loop for latency hiding in case the branch out of the i loop is not taken.

Intuitively, a critical edge might indicate a location in the program where we cannot place production without affecting paths that are not supposed to be affected by the production. The code shown in Figure 3.4 is a case of placing production at a synthetic node, namely the added **else** branch. Note that for the EAGER production on the **else** branch, which is the "$\text{READ}_{Send}\{x(6:N+5)\}$", a naïvely placed matching LAZY production (a "$\text{READ}_{Recv}\{x(6:N+5)\}$") might be located right before the k loop, since LAZY productions are generally delayed as far as possible. This, however, would violate balance, since on the **then** branch the corresponding EAGER production has already been matched by a LAZY production. Therefore, the LAZY production is moved up into the **else** branch.

Each of the requirements above can lead to a growth of G and can therefore slow GIVE-N-TAKE down. For example, inserting synthetic nodes makes $\mathcal{O}(N) = \mathcal{O}(E)$. However, it has been noted by several researchers that for typical programs, both the average out-degree of flow graph nodes and the maximal loop nesting depth can be assumed to be bounded by small constant independent of the size of the program [MR90]. Therefore, the increase of G should be fairly small for well structured programs.

Figure 3.13 shows the interval-flow graph for the code in Figure 3.12. The i loop, for example, corresponds to the interval $T(2)$ formed by nodes 3, 4, 5, with header 2; again, remember that the header itself is not part of the interval. Note that FLOW edges are the only non-SYNTHETIC edges that do not cross nesting level boundaries.

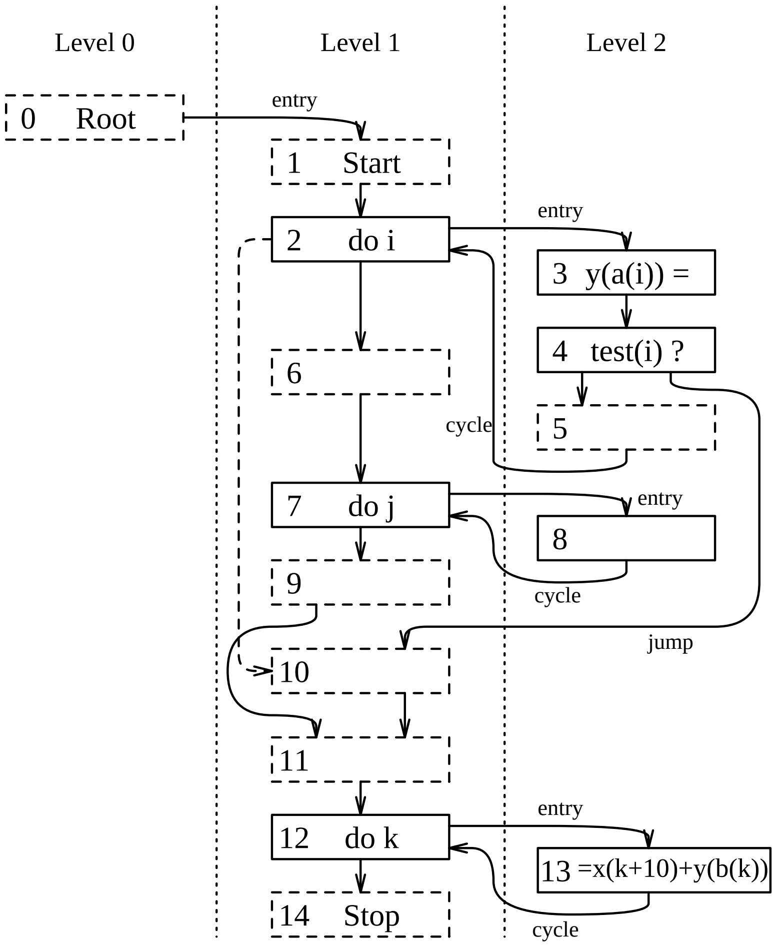


Figure 3.13 Flow graph for the code from Figure 3.12. The dashed nodes are synthetic nodes inserted to break critical edges, or to provide unique START and STOP nodes. The dashed edge $(2, 10)$ is a SYNTHETIC edge caused by JUMP edge $(4, 10)$ (since $4 \in T(2)$). All non-FLOW, non-SYNTHETIC edges are labeled as either ENTRY, CYCLE, or JUMP edges.

3.3.5 Traversal orders and neighbor relations

The order in which the nodes of the interval-flow graph are visited depends on the given problem type (BEFORE or AFTER, EAGER or LAZY) and on the pass of the GIVE-N-TAKE framework that is currently being solved (see Section 3.5). E induces two partial orderings on N:

Vertically: Given a FLOW/JUMP edge (m, n), a FORWARD order visits m before n, and a BACKWARD order visits m after n.

Horizontally: Given $m, n \in N$ such that $m \in T(n)$, an UPWARD order visits m before n, whereas a DOWNWARD order visits m after n.[‡]

Since these partial orderings are orthogonal, they can be combined without conflict into PREORDER (FORWARD and DOWNWARD), POSTORDER (FORWARD and UPWARD), and the corresponding reverse orderings, REVERSEPREORDER (BACKWARD and UPWARD) and REVERSEPOSTORDER (BACKWARD and DOWNWARD). For example, the nodes in Figure 3.13 are numbered in PREORDER.[§] Note that in a BEFORE problem, the flow of information when solving the data-flow equations is not necessarily in FORWARD order; this will become apparent in the discussion of the algorithm in Section 3.5.

A data-flow variable for some $n \in N$ might be defined in terms of variables of other nodes that are in some relation to n with respect to G. Therefore, we not only have to walk G in a certain order, but we also have to access for each $n \in N$ a subset of $N - \{n\}$ that has a certain relationship with n. In general, we are interested in information residing at predecessors or successors. However, we are also considering the type of the edge through which they are connected to n. The edge type carries information about how the neighboring nodes are related to each other, for example, whether moving production from one node to the other constitutes a hoist out of

[‡]Note that in the flow graph representation of Figure 3.13, the nesting depth increases left-to-right, not top-to-bottom. This gives rise to some terminology conflicts, since the terms for the partial orderings ("horizontal" and "vertical") are derived from this graph representation, whereas the terms for the two horizontal orderings ("UPWARD" and "DOWNWARD") correspond to decreasing and increasing loop nesting *depth*. An alternative would be to transpose the graph representation (*i.e.*, nesting depth increasing top-to-bottom and normal flow of control going left-to-right), but this was not chosen in order to adhere closer to the more common top-to-bottom representation of normal flow of control.

[§]This terminology is derived from viewing the flow graph as a tree, where CHILDREN(n) are subtrees of n, ordered according to flow of control.

a loop or not. The type also indicates whether this information has already been computed under the current node visiting order or not.

Let TYPE be a set of edge types, where the letters C, E, F, J, and S indicate CYCLE, ENTRY, FLOW, JUMP, and SYNTHETIC edges, respectively. GIVE-N-TAKE uses the following neighbor relations:

$\text{PREDS}^{\text{TYPE}}(n)$: The source nodes of edges reaching n of a type in TYPE.

$\text{SUCCS}^{\text{TYPE}}(n)$: The sink nodes of edges originating from n of a type in TYPE.

The conventional sets of "predecessors" and "successors" of n are $\text{PREDS}^{\text{CEFJ}}(n)$ and $\text{SUCCS}^{\text{CEFJ}}(n)$, respectively, which we will abbreviate as $\text{PREDS}(n)$ and $\text{SUCCS}(n)$, respectively. We will refer to the transitive closures of $\text{PREDS}^{\text{FJ}}(n)$ and $\text{SUCCS}^{\text{FJ}}(n)$ as the *ancestors* and *descendants* of n, respectively.

Note that $\text{PREDS}^{\text{C}}(n) = \{\text{LASTCHILD}(n)\}$, and $\text{PREDS}^{\text{E}}(n) = \{\text{HEADER}(n)\}$. Note also that the lack of critical edges has several implications for some of the sets defined above, two of which are stated in the following lemmata.

> **Lemma 3.1** The sink of a JUMP edge never has any predecessors besides the source of the JUMP edge.

Proof: First, remember that "predecessors" do not include SYNTHETIC edges. Let $e = (m, n)$ be a JUMP edge. Then there exists an $h \in N$ with $m \in T(h), n \notin T^+(h)$. Since $T^+(h)$ is by definition strongly connected, m must have successors within $T^+(h)$. Since n as well is a successor of m, m has multiple outgoing edges. However, G does not have critical edges, therefore n can only have one predecessor, which must be m; *i.e.*, $\text{PREDS}^{\text{CEF}}(n) = \emptyset$. □

> **Lemma 3.2** $\text{SUCCS}^{\text{EFJ}}(m) = \emptyset$ for each source m of a CYCLE edge.

Proof: Let $e = (m, n)$ be a CYCLE edge. Node n then is an interval header, which by definition has multiple predecessors. Since n is a successor of m, m may not have any other successors; otherwise e would be critical. However, it is $n \notin \text{SUCCS}^{\text{EFJ}}(m)$. □

Even though the equations and their correctness and effectiveness are the same for both BEFORE and AFTER problems, we will for simplicity assume in the following that we are solving a BEFORE problem unless noted otherwise.

3.4 Give-N-Take Equations

Given a set of initial variables for each node $n \in N$, which describe consumption, destruction, and side effects at the corresponding location in the program, GIVE-N-TAKE computes the production as a set of result variables for each node. Intermediate stages are the propagation and blocking of consumption, and the placing of production.

In the following, let $n \in N$, let $\perp$ denote the empty set, and let $\top$ be the whole data-flow universe. If an equation asks for certain neighbors, such as $\text{PREDS}^{\text{FJ}}(n)$, and there are no such neighbors, such as for a loop entry node, then an empty set results. Subscripts *in*, *out* denote variables for the entry and the exit of a node, respectively (reverse for AFTER problems). Subscript *loc* indicates information collected only from nodes within the same interval (*i.e.*, nodes in $J(n)$), and *init* identifies variables that are supplied as input to GIVE-N-TAKE.

Figure 3.14 contains the equations for the data-flow variables, which will be introduced in the following sections. We will provide example values from the READ instance for the graph in Figure 3.13, where x_k, y_a, and y_b correspond to references $x(k + 10)$, $y(a(i))$, and $y(b(k))$, respectively; values at ROOT are excluded for simplicity. In the following, we will refer to these array references as *array portions*; see Section 5.2.3 for a more formal definition.

3.4.1 Initial variables

The following variables get initialized depending on the problem to solve, where $\perp$ is the default value. Note that by default, these initializations are based on strictly local analysis. (Section 5.2.4 describes the initializations specific to generating READs, WRITEs and reductions for distributed memory accesses.)

$\mathsf{STEAL}_{init}(n)$: All elements whose production would be voided at n. This can also be used to prevent hoisting productions out of zero-trip loops, if so desired.

In our communication problem, this includes an array portion p if either the contents of this portion get partly modified at n, or if p itself gets changed, for example if p is an indirect array reference and n modifies the indirection array [HK93].

$$\mathsf{STEAL}(n) = \mathsf{STEAL}_{init}(n) \cup \mathsf{STEAL}_{loc}(\textsc{LastChild}(n)) \tag{3.1}$$

$$\mathsf{GIVE}(n) = \mathsf{GIVE}_{init}(n) \cup \mathsf{GIVE}_{loc}(\textsc{LastChild}(n)) \tag{3.2}$$

$$\mathsf{BLOCK}(n) = \mathsf{STEAL}(n) \cup \mathsf{GIVE}(n) \cup \bigcup_{s \in \textsc{Succs}^{\mathrm{E}}(n)} \mathsf{BLOCK}_{loc}(s) \tag{3.3}$$

$$\mathsf{TAKEN}_{out}(n) = \bigcap_{s \in \textsc{Succs}^{\mathrm{FJS}}(n)} \mathsf{TAKEN}_{in}(s) \tag{3.4}$$

$$\begin{aligned} \mathsf{TAKE}(n) = {} & \mathsf{TAKE}_{init}(n) \cup (\bigcup_{s \in \textsc{Succs}^{\mathrm{E}}(n)} \mathsf{TAKEN}_{in}(s) - \mathsf{STEAL}(n)) \\ & \cup((\mathsf{TAKEN}_{out}(n) \cap \bigcup_{s \in \textsc{Succs}^{\mathrm{E}}(n)} \mathsf{TAKE}_{loc}(s)) - \mathsf{BLOCK}(n)) \end{aligned} \tag{3.5}$$

$$\mathsf{TAKEN}_{in}(n) = \mathsf{TAKE}(n) \cup (\mathsf{TAKEN}_{out}(n) - \mathsf{BLOCK}(n)) \tag{3.6}$$

$$\mathsf{BLOCK}_{loc}(n) = (\mathsf{BLOCK}(n) \cup \bigcup_{s \in \textsc{Succs}^{\mathrm{F}}(n)} \mathsf{BLOCK}_{loc}(s)) - \mathsf{TAKE}(n) \tag{3.7}$$

$$\mathsf{TAKE}_{loc}(n) = \mathsf{TAKE}(n) \cup (\bigcup_{s \in \textsc{Succs}^{\mathrm{EF}}(n)} \mathsf{TAKE}_{loc}(s) - \mathsf{BLOCK}(n)) \tag{3.8}$$

$$\mathsf{GIVE}_{loc}(n) = (\mathsf{GIVE}(n) \cup \mathsf{TAKE}(n) \cup \bigcap_{p \in \textsc{Preds}^{\mathrm{FJ}}(n)} \mathsf{GIVE}_{loc}(p)) - \mathsf{STEAL}(n) \tag{3.9}$$

$$\begin{aligned} \mathsf{STEAL}_{loc}(n) = {} & \mathsf{STEAL}(n) \cup \bigcup_{p \in \textsc{Preds}^{\mathrm{FJ}}(n)} (\mathsf{STEAL}_{loc}(p) - \mathsf{GIVE}_{loc}(p)) \cup \\ & \bigcup_{p \in \textsc{Preds}^{\mathrm{S}}(n)} \mathsf{STEAL}_{loc}(p) \end{aligned} \tag{3.10}$$

$$\begin{aligned} \mathsf{GIVEN}_{in}(n) = {} & \mathsf{GIVEN}(\textsc{Header}(n)) \cup \bigcap_{p \in \textsc{Preds}^{\mathrm{FJ}}(n)} \mathsf{GIVEN}_{out}(p) \cup \\ & (\mathsf{TAKEN}_{in}(n) \cap \bigcup_{q \in \textsc{Preds}^{\mathrm{FJ}}(n)} \mathsf{GIVEN}_{out}(q)) \end{aligned} \tag{3.11}$$

$$\mathsf{GIVEN}(n) = \mathsf{GIVEN}_{in}(n) \cup \begin{cases} \mathsf{TAKEN}_{in}(n) & \text{for an } \textsc{Eager} \text{ Problem,} \\ \mathsf{TAKE}(n) & \text{for a } \textsc{Lazy} \text{ Problem.} \end{cases} \tag{3.12}$$

$$\mathsf{GIVEN}_{out}(n) = (\mathsf{GIVE}(n) \cup \mathsf{GIVEN}(n)) - \mathsf{STEAL}(n) \tag{3.13}$$

$$\mathsf{RES}_{in}(n) = \mathsf{GIVEN}(n) - \mathsf{GIVEN}_{in}(n) \tag{3.14}$$

$$\mathsf{RES}_{out}(n) = \bigcup_{s \in \textsc{Succs}^{\mathrm{FJ}}(n)} \mathsf{GIVEN}_{in}(s) - \mathsf{GIVEN}_{out}(n) \tag{3.15}$$

Figure 3.14 GIVE-N-TAKE equations.

For Figure 3.13, we have for example $y_b \in \mathsf{STEAL}_{init}(\{3\})$. (Read as: "For the READ problem, the variable STEAL_{init} at node 3 contains the array portion referenced by $y(b(k))$.")

$\mathsf{GIVE}_{init}(n)$: All elements that "come for free;" *i.e.*, elements that are already produced at n.

If we do not use the owner-computes rule in communication generation, then this includes local definitions of non-owned data, since a later reference to these data does not need to communicate them in any more.

$y_a \in \mathsf{GIVE}_{init}(\{3\})$.

$\mathsf{TAKE}_{init}(n)$: The set of consumers at n.

For communication generation, this is the set of non-owned array references.

$x_k, y_b \in \mathsf{TAKE}_{init}(\{13\})$.

3.4.2 Propagating consumption

The following variables, together with the variables defined in Section 3.4.3, analyze consumption.

$\mathsf{STEAL}(n)$: All elements whose production would be voided by n itself, as given by $\mathsf{STEAL}_{init}(n)$, or by some $m \in T(n)$ without being resupplied by a descendant of m within $T(n)$, which is given by $\mathsf{STEAL}_{loc}(\text{LASTCHILD}(n))$.

$y_b \in \mathsf{STEAL}(\{2\text{–}3\})$.

$\mathsf{GIVE}(n)$: All elements that are already produced at n, or at some node in $T(n)$ without being stolen later within $T(n)$.

$\mathsf{BLOCK}(n)$: Elements whose production is blocked by n; *i.e.*, elements whose production cannot be hoisted across n because they are stolen or already produced at n or a node in $T(n)$.

$y_a, y_b \in \mathsf{BLOCK}(\{2\text{–}3\})$.

$\mathsf{TAKEN}_{out}(n)$: Things guaranteed to be consumed before being stolen on all paths originating in n, excluding n itself. Here we have to consider not only FLOW and JUMP edges, but also SYNTHETIC edges; otherwise we might violate safety by producing something whose only consumer may be skipped due to a jump out of a loop.

$x_k, y_b \in$ TAKEN$_{out}(\{2, 6–7, 9–11\})$;
also, $x_k \in$ TAKEN$_{out}(\{1\})$.

TAKE(n): The set of consumers at n. This includes items that are guaranteed to be consumed by nodes in $T(n)$ (the TAKEN$_{in}$ term) and not stolen at n, and items that may be consumed by $T(n)$ (the TAKE$_{loc}$ term) and are guaranteed to be consumed on exit from n without being blocked by n.

$x_k, y_b \in$ TAKE($\{12–13\}$).

TAKEN$_{in}(n)$: Similar to TAKEN$_{out}$, except that the effects of n itself are included.

$x_k, y_b \in$ TAKEN$_{in}(\{6–7, 9–13\})$;
also, $x_k \in$ TAKEN$_{in}(\{1–2\})$.

BLOCK$_{loc}(n)$: Items blocked by n or by descendants of n within $J(n)$ without being consumed.

$y_a, y_b \in$ BLOCK$_{loc}(\{1–3\})$.

TAKE$_{loc}(n)$: Items taken by n, by descendants of n within $J(n)$, or by nodes within $T(n)$. Here, unlike for BLOCK$_{loc}$, we have to explicitly include successors on Entry edges, since they are not guaranteed to be reflected in TAKE, which has to be conservatively small, whereas they will always be considered by BLOCK(n), which is conservatively large.

$x_k, y_b \in$ TAKE$_{loc}(\{6–7, 9–13\})$;
also, $x_k \in$ TAKE$_{loc}(\{1–2\})$.

3.4.3 Blocking consumption

The following variables are used by the interval headers to determine whether items are stolen or taken within the interval.

GIVE$_{loc}(n)$: Items produced by n or by ancestors of n within the same interval. Here items are treated as produced also if they are consumed, since consumption is guaranteed to be satisfied by a production.

$y_a \in$ GIVE$_{loc}(\{2–7, 9–11\})$;
$x_k, y_b \in$ GIVE$_{loc}(\{12–14\})$.

$\mathsf{STEAL}_{loc}(n)$: Items stolen by n, or stolen by a predecessor p of n without being resupplied by p. Furthermore, if there exists a $p \in \text{P}_{\text{REDS}}^{\text{S}}(n)$ (*i.e.*, n is the sink of a JUMP edge, and p is the header of an interval enclosing the source of the JUMP edge but not n itself), then we also have to include items stolen by p. However, since taking the JUMP edge corresponds to a jump from within the interval, the interval headed by p is not guaranteed to be completed before n is reached; therefore, we cannot exclude items resupplied by p, which would be given by $\mathsf{GIVE}_{loc}(p)$.

$y_b \in \mathsf{STEAL}_{loc}(\{2\text{–}7,\ 9\text{–}12,\ 14\})$.

3.4.4 Placing production

After analyzing at each node what is consumed and not already produced, the production needed to satisfy all consumers is computed by the following variables. As described in Section 3.5, the following variables may differ for the EAGER and for the LAZY solution; this will be indicated in the examples by superscripts.

$\mathsf{GIVEN}_{in}(n)$: Things that are guaranteed to be available at the entry of n, or, for an AFTER problem, the exit of n. If n is a first child, then it has everything available that is available at its header, and it is $\text{PREDS}^{\text{FJ}} = \emptyset$. Otherwise, things are guaranteed to be produced if they are produced along all incoming paths, or if they are produced at least along some incoming paths and guaranteed to be consumed. In the latter case, the result variable RES_{out} will ensure that things will be produced also along the paths that originally did not have them available (see Equation 3.15).

$x_k \in \mathsf{GIVEN}_{in}^{eager}(\{2\text{–}14\})$;
$y_a \in \mathsf{GIVEN}_{in}^{eager}(\{4\text{–}14\})$;
$y_b \in \mathsf{GIVEN}_{in}^{eager}(\{7\text{–}9,\ 11\text{–}14\})$.
$x_k, y_b \in \mathsf{GIVEN}_{in}^{lazy}(\{13\text{–}14\})$;
$y_a \in \mathsf{GIVEN}_{in}^{lazy}(\{4\text{–}14\})$.

$\mathsf{GIVEN}(n)$: Items guaranteed to be available at n itself, either because they come from predecessors of n, or because they are consumed by n itself, or, for an EAGER problem, by a descendant of n.

$x_k \in \mathsf{GIVEN}^{eager}(\{1\text{–}14\})$;
$y_a \in \mathsf{GIVEN}^{eager}(\{4\text{–}14\})$;

$y_b \in \mathsf{GIVEN}^{eager}(\{6\text{–}14\})$.
$x_k, y_b \in \mathsf{GIVEN}^{lazy}(\{12\text{–}14\})$;
$y_a \in \mathsf{GIVEN}^{lazy}(\{4\text{–}14\})$.

$\mathsf{GIVEN}_{out}(n)$: Things that are available on exit from n. This includes whatever comes from at n itself, but it excludes things stolen by n.

$x_k \in \mathsf{GIVEN}^{eager}_{out}(\{1\text{–}14\})$;
$y_a \in \mathsf{GIVEN}^{eager}_{out}(\{2\text{–}14\})$;
$y_b \in \mathsf{GIVEN}^{eager}_{out}(\{6\text{–}14\})$.
$x_k, y_b \in \mathsf{GIVEN}^{lazy}_{out}(\{12\text{–}14\})$;
$y_a \in \mathsf{GIVEN}^{lazy}_{out}(\{2\text{–}14\})$.

3.4.5 Result variables

The result of Give-N-Take analysis is expressed by the following variables.

$\mathsf{RES}_{in}(n)$: The production generated at the entry of n. This includes everything that is guaranteed to be available at n itself but is not yet available at the entry of n.

The $\textsc{Read}_{Send}$s stem from $x_k \in \mathsf{RES}^{eager}_{in}(\{1\})$ and $y_b \in \mathsf{RES}^{eager}_{in}(\{6,\ 10\})$; the $\textsc{Read}_{Recv}$s are $x_k, y_b \in \mathsf{RES}^{lazy}_{in}(\{12\})$.

$\mathsf{RES}_{out}(n)$: The production at the exit of n. This includes items whose availability has been guaranteed to some successors of n and that are not already available on exit from n. See also the discussion of GIVEN_{in} in Section 3.4.4.

In Figure 3.13, there is no production needed on exit.

Note the following properties for production on exit:

Lemma 3.3

Let $n \in N$ be a node with $\mathsf{RES}_{out}(n) \neq \emptyset$. Then node n has exactly one successor $s \in \textsc{Succs}^{\text{FJ}}(n)$.

Proof: Equation 3.15 implies that $x \in \mathsf{RES}_{out}(n)$ requires $x \notin \mathsf{GIVEN}_{out}(n)$, but that for some $s \in \textsc{Succs}^{\text{FJ}}(n)$ and $p \in \textsc{Preds}^{\text{FJ}}(s) - \{n\}$, $x \in \mathsf{GIVEN}_{out}(p)$ must hold. In other words, n must have a successor s which in turn has a predecessor $p \neq n$ that produces an x which is consumed by s and not produced by n. The lack of critical edges then implies that s must be the only successor of n, and therefore it does not matter whether we use union or intersection in Equation 3.15. □

Corollary 3.4

Definition (3.15) is equivalent to the following:

$$\mathsf{RES}_{out}(n) \quad = \bigcap_{s \in \text{SUCCS}^{\text{FJ}}(n)} \mathsf{GIVEN}_{in}(s) - \mathsf{GIVEN}_{out}(n). \qquad (3.16)$$

Proof: Follows directly from Lemma 3.3. □

Figure 3.15 shows the code from Figure 3.12 annotated with communication generation as computed by GIVE-N-TAKE.

3.5 Solving the Equations

This section presents an algorithm called *GiveNTake* that can be used to solve a code placement problem via the GIVE-N-TAKE framework. Section 3.4 already listed the equations that lead from the initial data-flow variables to the result variables. What is left towards an actual algorithm is a recipe for evaluating these equations.

3.5.1 The constraints

The objective of the algorithm is to assign the flow variables at each node a value that is consistent with all equations; *i.e.*, we have to reach a *fixed point.* Note that the number of evaluation iterations to reach a fixed point may be constant, as is usually the case in interval analysis. In general, the evaluation order is also important for the convergence rate and, in some cases, termination behavior of the algorithm. For GIVE-N-TAKE, there actually exists an order where the right hand side of each equation to be evaluated is already fully known due to previous computation. Therefore, *GiveNTake* has to evaluate each equation only once for each node, which implies guaranteed termination and low computational complexity; it also implies *fastness* [GW76]. However, since the direction of the flow of information varies across the equations, we still need multiple passes over the control flow graph, solving a different set of equations during each pass.

An objective for *GiveNTake* is to minimize the number of passes; therefore, we partition the equations into different sets that can be evaluated concurrently, *i.e.*,

$$
\begin{array}{l}
\boxed{\textsc{Read}_{Send}\{x(11:N+10)\}} \\
\textbf{do } i = 1, N \\
\quad y(a(i)) = \ldots \\
\quad \textbf{if } test(i) \textbf{ then} \\
\qquad \boxed{\begin{array}{l} \textsc{Write}_{Send}\{y(a(1:i))\} \\ \textsc{Write}_{Recv}\{y(a(1:i))\} \\ \textsc{Read}_{Send}\{y(b(1:N))\} \end{array}} \\
\qquad \text{goto } 77 \\
\quad \textbf{endif} \\
\textbf{enddo} \\
\boxed{\begin{array}{l} \textsc{Write}_{Send}\{y(a(1:N))\} \\ \textsc{Write}_{Recv}\{y(a(1:N))\} \\ \textsc{Read}_{Send}\{y(b(1:N))\} \end{array}} \\
\textbf{do } j = 1, N \\
\quad \ldots \\
\textbf{enddo} \\
77\ \boxed{\textsc{Read}_{Recv}\{x(11:N+10), y(b(1:N))\}} \\
\textbf{do } k = 1, N \\
\quad \ldots = x(k+10) + y(b(k)) \\
\textbf{enddo}
\end{array}
$$

Figure 3.15 The code from Figure 3.12 annotated with communication statements.

within the same pass. It turns out that Sections 3.4.2, 3.4.3, 3.4.4, and 3.4.5 each define one set of equations that can be evaluated concurrently. We will refer to these sets as S_1 (Equations 3.1–3.8), S_2 (Equations 3.9, 3.10), S_3 (Equations 3.11–3.13), and S_4 (Equations 3.14 and 3.15), respectively. Since all equations except Equation 3.12 in S_3 are the same for EAGER and LAZY problems and S_1 and S_2 are computed before S_3, the variables defined in S_1 and S_2 are the same for both kinds of problems. Therefore, we need to differentiate between EAGER and LAZY only for variables defined in S_3 and S_4. We distinguish these variables by superscripts *eager* and *lazy*.

To determine an order for solving the GIVE-N-TAKE equations that yields a fixed point after evaluating each equation only once, we have to make sure that an equation is not evaluated before the right hand side is fully known. As also described in more detail in Appendix A.2, inspection of the equations yields the following constraints.

- S_1 should be evaluated in BACKWARD order, for example, because Equation 3.8 defines $\mathsf{TAKE}_{loc}(n)$ in terms of $\mathsf{TAKE}_{loc}(s)$, with $s \in \text{SUCCS}^{\text{EF}}(n)$.
- S_1 should also be evaluated in UPWARD order; *e.g.*, Equation 3.3.
- $S_1(n)$ ("the equations from S_1 for node n") should be computed before $S_2(n)$, but after $S_2(\text{CHILDREN}(n))$.
- S_2 should be evaluated in FORWARD order.
- S_3 must be computed in FORWARD, DOWNWARD fashion, *i.e.*, in PREORDER, after S_1.
- S_4 has to be evaluated after S_1 and S_3, in any order.

Intuitively, these constraints express that information about consumption is flowing up and back, whereas the availability of production gets propagated forward and down. The production to be inserted at a node, however, again depends on the successors of the node.

3.5.2 The algorithm

The resulting algorithm is shown in Figure 3.16. A formal proof that it does indeed obey all ordering constraints, as well as a proof that GIVE-N-TAKE meets the correctness constraints (C1), (C2), and (C3), can be found in Appendix A.

Procedure *GiveNTake*

Input: $G = (N, E)$; $\forall n \in N$:
$\mathsf{TAKE}_{init}(n)$, $\mathsf{STEAL}_{init}(n)$, $\mathsf{GIVE}_{init}(n)$
Output: $\forall n \in N$: $\mathsf{RES}^{eager}(n)$ and/or $\mathsf{RES}^{lazy}(n)$

forall $n \in N$, in ReversePreOrder
  forall $c \in$ Children(n), in Forward order
    Compute Equations 3.9, 3.10
  endforall
  Compute Equations 3.1…3.8
endforall
forall $n \in N$, in PreOrder
  Compute Equations 3.11…3.13 for Eager/Lazy
endforall
forall $n \in N$
  Compute Equations 3.14,3.15 for Eager/Lazy
endforall
end

Figure 3.16 Algorithm *GiveNTake* computing an Eager/Lazy code placement. RES without subscripts stands for both RES_{in} and RES_{out}.

As already noted, each equation is evaluated only once for each node in N. Furthermore, each equation depends only on a subset of neighbors. Therefore, the total complexity of GIVE-N-TAKE is $\mathcal{O}(E)$ steps where the cost of each step depends on the current lattice and its representation, for example bit vectors of a certain length. As already noted in Section 3.3.4, E can be assumed to be of a size in the order of the program size in most cases; under this assumption, GIVE-N-TAKE as well as other interval-based elimination methods have linear time complexity.

3.5.3 BEFORE vs. AFTER problems

We mentioned earlier that an AFTER problem can essentially be treated as a BEFORE problem with reversed flow of control. However, this also means that the reversed flow graph has to fulfill the same requirements from Section 3.3.4 as the original graph, which is not trivially the case. For example, ENTRY edges may become CYCLE edges and vice versa, but each loop may have only one CYCLE edge; this can be satisfied by adding nodes similar to the SYNTHETIC nodes. More severe is the requirement for G to be reducible, which will be violated if the original graph had any JUMP edges, since these will become jumps *into* loops. In fact, this would prevent us from determining a unique set of intervals for the reverse G. For example, consider the flow graph in Figure 3.17, which may be the result of solving an AFTER problem for a program containing a jump out of a loop. A consumption placed at node 4 might be hoisted into its header, node 3, which would be unsafe due to the path 1–2–5–3.

In our implementation, we handle this case by using the same interval structure as for the original graph, and preventing hoisting production out of loops that contain JUMP edges. This can be done by either accordingly initializing STEAL_{init} for each header of a loop containing a JUMP edge, or by ignoring for these headers the contributions to TAKE coming from the loop body (see Equation 3.5).

3.5.4 A note on synthetic nodes

Having computed the result variables with GIVE-N-TAKE, one still has to perform the actual program optimizations by modifying the analyzed code. This step might be complicated by having production placed at a synthetic node, which would require new basic blocks (see Figure 3.4). However, it may often be possible to shift production to a neighboring non-synthetic node. This can either be done at code generation time, or by post-processing the results of GIVE-N-TAKE, in a way that is similar to

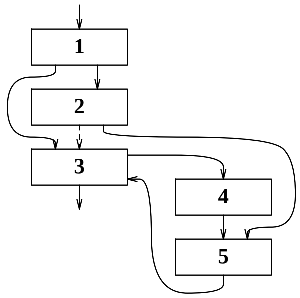


Figure 3.17 Flow graph containing a jump *into* a loop. Note the synthetic (dashed) edge between nodes 2 and 3.

a mechanism employed in edge placement [Dha88a] for avoiding code proliferation. Our implementation took the latter route, by running a backward pass on G which checks whether these movements can be done without conflicts.

3.6 Summary

This chapter has outlined a general code generation framework, based on Tarjan intervals, that handles several different classes of problems. Unlike previous approaches, it does not assume atomicity. Instead, Give-N-Take provides both Eager and Lazy solutions, and it guarantees their balance across arbitrary control flow. Furthermore, Give-N-Take can be applied to both Before and After problems, and it can take advantage of side effects to further eliminate unnecessary production without affecting balance. Other nice properties of Give-N-Take include the option to hoist code out of zero-trip loop constructs even for nested loops, and the natural handling of irregular loop bounds and access patterns.

Note, however, that as with code placement strategies in general, there may be conflicting goals in how far to separate production and consumption. Often the computations compete for resources, such as registers or message buffers, which could cause some "optimizations" to have a negative effect in practice. While Give-N-Take does not address this issue directly, certain extensions, such as a heuristic for inserting additional STEAL_{init}s which blocks production, could help to solve this conflict. Other possible extensions are the combination with dependence analysis citeKeNedeljkovic:DepDat, for example by refining the initial assignments to

TAKE$_{init}$ and STEAL$_{init}$, or a more thorough treatment of jumps out of loops for AFTER problems. While our current approach (Section 3.5.3) prevents unsafe code generation, it may miss some otherwise legal optimizations. Related to that is the issue of analyzing irreducible graphs in general.

As described in Chapter 5, the FORTRAN D compiler uses GIVE-N-TAKE to generate messages for distributed-memory machines. We generate READs, WRITEs, and WRITEs combined with different reduction operations, such as summation. All of these operations can be placed either atomically, for example, for a library call, or divided into sends and receives. The non-atomicity and balance attributes enables message latency hiding and other optimizations to be performed across arbitrary control flow. GIVE-N-TAKE's flexibility allowed us to apply the same algorithm to very different tasks that traditionally were solved with separate frameworks. This simplified the implementation in the FORTRAN D compiler significantly.

... the process of question and answer,
giving and taking,
talking at cross purposes
and seeing each other's point —
performs the communication ...

— Hans-Georg Gadamer (Truth and Method)

Chapter 4

Irregular Computations on SIMD architectures

On a MIMD machine, performing irregular computations does not impose particular problems once data have been distributed and can be properly communicated, as addressed in the previous two chapters. On SIMD machines, however, the situation is slightly different, due to the control-flow restrictions imposed by the *Single Instruction* Multiple Data model. There is only one program counter shared by all processors, and for each instruction issued by that counter a processor only has the choice between participating in it or sitting idle ("being masked out"). In other words, SIMD processors have to synchronize after every single instruction, instead of just at communication points. Every derivation from completely regular, balanced control flow increases average idle time. This tends to cause performance problems for naïve implementations of irregular applications.

An important special case are loop nests with an uneven number of iterations of the inner loop(s). When parallelizing the outer loop, different processors may end up with a work load that is not balanced between different iterations of the outer loop. Assuming no additional synchronization within the loop nest, this does not cause any problems on a MIMD machine, as long as the total work load assigned to each processor balances out. Under the SIMD model, however, each processor is required to wait at each iteration of the outer loop for the processor with the largest number of inner loop iterations, and idle time is likely to be high.

This chapter describes the technique of *loop flattening*, which aims at overcoming these control-flow constraints in the context of loop nests with an uneven number of inner iterations. The rest of this chapter is organized as follows. Section 4.1 describes the different variants of pseudo-FORTRAN used in the examples. Section 4.2 presents a small example to illustrate the kind of problem we are interested in and gives a first glance at loop flattening, which Section 4.3 elaborates on at a more general level. Section 4.4 evaluates loop flattening from the compiler perspective. (Experimental results are given in Section 6.2).

4.1 Languages

The concepts introduced here apply to a broad range of languages. We will give program examples in different variants of pseudo-FORTRAN:

F77 – Strictly sequential FORTRAN 77 (possibly a "dusty deck" program).

F77D – F77 enhanced with decomposition statements as proposed in FORTRAN D [FHK+90]. An important goal of F77D is to provide a basis for efficient compilation towards both MIMD and SIMD distributed-memory machines, so it does not contain any constructs that are specific to either architecture.

F77$_{MIMD}$ – A version of FORTRAN 77 designed to run on a MIMD machine, which assumes a separate name space for each processor.

F90$_{SIMD}$ – A version of FORTRAN 90 designed to run on a SIMD machine, similar to Connection Machine FORTRAN [Thi91] or MasPar FORTRAN [Mas91]. There are two important differences relative to the F77 variants:

- By default, scalars of F77 will be *replicated* in F90$_{SIMD}$; *i.e.*, they will be declared as vectors of size P, where processor p owns the p-th element.
- In keeping with FORTRAN 90 convention, omitted array indices refer to all elements of an array dimension, and an unsubscripted array reference refers to all array elements.

For enhancing readability of F90$_{SIMD}$ examples, we extend the language constructs that are typically implemented by vendors in several ways:

- The **forall** construct can be applied not only to single statements, but also to blocks, as is the case in HPF [Hig93]. The general form of this extension can be interpreted differently depending on the semantics chosen for the case where different iterations modify the same set of data; our examples, however, will avoid these access interferences.
- **do**-**enddo**s, **do**-**while**s, **if**s, **where**s, and **forall**s can be nested freely within each other. (HPF does not allow these control structures to be nested within **forall**s.)
- **while** loops can be controlled by an array of booleans (instead of just a scalar boolean), if the different array elements are guaranteed to have identical values.

4.2 Example of Loop Flattening

Consider the contrived F77 loop nest in Figure 4.1, henceforth called *Example*. This clearly is a dependence-free, parallelizable loop, where the number of inner loop iterations depends on the current iteration of the outer loop. Let K be 8 and let $L(1{:}8)$ have the values 4,1,2,1,1,3,1,3, respectively. Assuming $P = 2$ processors and the owner-computes rule, where in all assignment statements the right hand side expression is computed by the processor that "owns" the left hand side variable, we can in this case just distribute L and the rows of X blockwise to achieve perfect load balance. This is illustrated in the F77D program in Figure 4.2, which, for $Lmax = 4$, assigns $L(1{:}4)$, $X(1{:}4,1{:}4)$ to processor 0 and $L(5{:}8)$, $X(5{:}8,1{:}4)$ to processor 1. The owner-computes rule results in partitioning the iteration space among the two processors, so each processor executes only some iterations of the outer loop.

For a MIMD machine, the Fortran D compiler would derive the F77$_{MIMD}$ program shown in Figure 4.3. Each processor executes the loop nest independently, needing a total of

$$TIME_{MIMD} = \max_{p=1,2} \sum_{i=1}^{4} L(i + 4p) = 8 \tag{4.1}$$

inner loop iterations. This is illustrated in the trace in Figure 4.4.

A F90$_{SIMD}$ version could be derived from the F77D program by just changing the outer **do** loop to a **forall** loop. This would result in a partitioning of the iteration space, similar to the F77D version. For expository reasons, we will consider a slightly different but equivalent F90$_{SIMD}$ version that takes the data decomposition and the number of processors already into account and thus directly reflects the control flow for $K = 8$ and $P = 2$. As in the F77$_{MIMD}$ version, we change the upper bound of the outer loop from $K = 8$ to $K/P = 4$ and let each processor execute all iterations of the loop. We continue to use the loop index i in control-flow related statements; to enable the different processors to operate on different data, we introduce an auxiliary induction variable i', which replaces i in non-control-flow statements. The result is shown in Figure 4.5.

Note how we had to transform the inner **do** loop due to the single SIMD control flow. To make sure that each processor can perform all of its iterations, the upper bound $L(i')$ had to be changed into the maximum of $L(i')$ over all processors. This in turn necessitated a guard for the loop body that tests whether this processor is still involved in the current inner loop iteration or whether it is masked out and sits idle, possibly to participate again in later iterations.

```
C  P1 – sequential version
   do i = 1, K
      do j = 1, L(i)
         X(i,j) = i * j
      enddo
   enddo
```

Figure 4.1 Original loop nest *Example*.

```
C  P2 – Fortran D version
   decomposition XD(K, Lmax), LD(K)
   align X with XD, L with LD
   distribute XD(block,*), LD(block)

   do i = 1, K
      do j = 1, L(i)
         X(i,j) = i * j
      enddo
   enddo
```

Figure 4.2 *Example* in F77D.

```
C  P3 – MIMD version
   do i = 1, 4
      do j = 1, L'(i)
         X'(i,j) = i * j
      enddo
   enddo
```

Figure 4.3 *Example* in F77$_{MIMD}$. X and L are renamed to X' and L' to reflect that there is no common name space any more. On processor p, $p = 0, 1$, $L'(i)$ corresponds to $L(i + 4p)$, and $X'(i,j)$ corresponds to $X(i + 4p, j)$.

Time	1	2	3	4	5	6	7	8
i_0	1	1	1	1	2	3	3	4
j_0	1	2	3	4	1	1	2	1
i_1	1	2	2	2	3	4	4	4
j_1	1	1	2	3	1	1	2	3

Figure 4.4 MIMD execution trace for *Example* loop. Here i_p and j_p denote i and j on processor p.

C *P4 – naïve SIMD version*
do $i = 1, 4$
  $i' = i + [0, 4]$
  do $j = 1, \max(L(i'))$
    where $(j \leq L(i'))\ X(i', j) = i' * j$
  enddo
enddo

Figure 4.5 *Example* in F90$_{SIMD}$. [0,4] denotes the two-element vector containing 0 and 4.

We will refer to this transformation, which can be applied to other loop types as well, as *SIMDizing* a loop. It is a straightforward consequence of the SIMD restricted control flow and motivates the code transformation introduced in this chapter. The outer loop does not have to be SIMDized in this particular case because we know that each processor works on exactly four rows of X and therefore has to execute the outer loop the same number of times. Loop SIMDizing has the effect that our F90$_{SIMD}$ program has to execute

$$TIME_{SIMD} = \sum_{i=1}^{4} \max_{p=0,1} L(i + 4p) = 12 \tag{4.2}$$

iterations. Roughly speaking, our time bound has increased from a maximum over sums to a sum over maxima. This becomes apparent when considering the execution trace shown in Figure 4.6.

Since the equivalent MIMD implementation performs significantly better, this bad running time can not be explained by lack of parallelism or bad load balance.

Time	1	2	3	4	5	6	7	8	9	10	11	12
i_0	1	1	1	1	2			3	3	4		
j_0	1	2	3	4	1			1	2	1		
i_1	1				2	2	2	3		4	4	4
j_1	1				1	2	3	1		1	2	3

Figure 4.6 Execution trace for unflattened example loop; i_p, j_p denote the actual iteration counts of processor p, no entry means "idle."

To overcome this purely control-flow related problem, we apply *loop flattening*, which will be introduced at a more general level in the next section. The result is shown in Figure 4.7. Now we can achieve the same time bound as in the MIMD implementation, needing only eight steps as shown in the trace in Figure 4.4.

The reader might have noticed that the loop body shown in Figure 4.7 is now always executed at least once for each outer loop iteration, which is equivalent to assuming $L(i) \geq 1$ for all i. Even though this is correct in our example, a more general loop flattening does not rely on this assumption, as we will see in the next section.

4.3 General Loop Flattening

Assume that we are given two fully parallelizable nested loops such as in the previous section; an extension of the following to deeper loop nests is straightforward. Each of the loops might be structured as a **while** loop, a **do**-**while** loop, a simple **do** or **forall** loop, or it might use conditional **goto**s. The transformation described here can be done either at the F77/ F77D level or at the F90$_{SIMD}$ level. For simplicity and generality, we will present it here on the F77 level. A corresponding F90$_{SIMD}$ version can always be directly derived by SIMDizing loops and replacing **if**s with **where**s.

4.3.1 Loop normalization

As a first step, we *normalize* both loops by breaking their control pattern into three *phases* for each nesting level l; an initialization phase $init_l$, a guard $test_l$, and an incrementing step inc_l. For example, a control pattern such as **do** $var = lo,\ hi,\ stride$ would be broken into $init_l \equiv var = lo$, $test_l \equiv (var \leq hi)$, and $inc_l \equiv var = var$

```
C  P5 – flattened SIMD version
   i = [1, 5]
   K = [4, 8]
   j = 1
   while any (i ≤ K)
      where (i ≤ K)
         X(i, j) = i * j
         where (j = L(i))
            i = i + 1
            j = 1
         elsewhere
            j = j + 1
         endwhere
      endwhere
   endwhile
```

Figure 4.7 *Example* in flattened F90$_{SIMD}$.

+ *stride*. The resulting loop nest *GenNest* is shown in Figure 4.8, along with the corresponding version of the *Example* from the previous section; of course, we usually expect *BODY* to contain more computational work than in *Example*.

Since *GenNest* conservatively tests for loop completion before entering the loop body, all loops can be brought into this normal form. To estimate the running time of the above code on P processors, for processor p let K_p be the number of outer loop iterations and L_p^i be the number of inner loop iterations for the i-th outer loop iteration. A straightforward MIMD version would then finish after

$$TIME_{MIMD} = \max_{p=0,\ldots,P-1} \sum_{i=1}^{K_p} L_p^i \qquad (4.1')$$

iterations.

A F90$_{SIMD}$ version could be derived by SIMDizing both **while** loops and would execute

$$TIME_{SIMD} = \sum_{i=1}^{\max_{p=0}^{P-1} K_p} \max_{p=0,\ldots,P-1} L_p^i \qquad (4.2')$$

iterations. Again, if the number of iterations of the inner loop varies from one outer loop iteration to the next, then the restriction to a common program counter makes this SIMD implementation inefficient.

program *GenNest A*	program *Example A*
$init_1$	$i = 1$
while $test_1$	**while** $(i \leq K)$
$init_2$	$j = 1$
while $test_2$	**while** $(j \leq L(i))$
$BODY$	$X(i, j) = i * j$
inc_2	$j = j + 1$
endwhile	**endwhile**
inc_1	$i = i + 1$
endwhile	**endwhile**

Figure 4.8 Generic loop nest *GenNest* (left) and corresponding *Example* (right); original version after normalization.

4.3.2 The transformation

Since we do not know whether the evaluation of $test_l$ has any side effects, we introduce flags t_l to store the results of evaluating the conditions $test_l$ before we make any other transformations, as shown in Figure 4.9. So far, control flow is still unchanged.

The key idea of loop flattening is to make sure that each processor has a chance to advance to the next loop iteration where it participates in the execution of *BODY* before the control flow actually reaches *BODY*. One requirement that follows immediately is that control variables (iteration counts etc.) are replicated to enable individual processors to advance independently to the next outer loop iteration whenever they are done with the current inner loop. Furthermore, we have to take *BODY* out of the part of the loop nest that handles the transition between different iterations of the inner and outer loop. Each processor should be able to execute *BODY* whenever it has still work left to do in this loop nest and the control flow reaches *BODY*. In other words, *BODY* should be executed whenever t_1 is *true*, independent of t_2. The flattened loop version meeting these goals is shown in Figure 4.10.

As the reader might verify, we still execute exactly the same instructions in the same order and the same number of times as we did in the original loop nest. Figure 4.11 provides a step-by-step comparison of the two versions. We also still have two nested loops. However, *BODY* is lifted out of the inner loop. The inner loop now contains just the control structure to let each processor advance to the next

program *GenNest B*	**program** *Example B*
$init_1$	$i = 1$
$t_1 = test_1$	$t_1 = (i \leq K)$
while t_1	**while** t_1
$init_2$	$j = 1$
$t_2 = test_2$	$t_2 = (j \leq L(i))$
while t_2	**while** t_2
$BODY$	$X(i, j) = i * j$
inc_2	$j = j + 1$
$t_2 = test_2$	$t_2 = (j \leq L(i))$
endwhile	**endwhile**
inc_1	$i = i + 1$
$t_1 = test_1$	$t_1 = (i \leq K)$
endwhile	**endwhile**

Figure 4.9 *GenNest/Example*, with guard variables.

program *GenNest C*	**program** *Example C*
$init_1$	$i = 1$
$t_1 = test_1$	$t_1 = (i \leq K)$
if t_1 **then** $init_2$	**if** t_1 **then** $j = 1$
while t_1	**while** t_1
$t_2 = test_2$	$t_2 = (j \leq L(i))$
while $(t_1 \wedge \neg\, t_2)$	**while** $(t_1 \wedge \neg t_2)$
inc_1	$i = i + 1$
$t_1 = test_1$	$t_1 = (i \leq K)$
if t_1 **then**	**if** t_1 **then**
$init_2$	$j = 1$
$t_2 = test_2$	$t_2 = (j \leq L(i))$
endif	**endif**
endwhile	**endwhile**
if t_1 **then**	**if** t_1 **then**
$BODY$	$X(i, j) = i * j$
inc_2	$j = j + 1$
endif	**endif**
endwhile	**endwhile**

Figure 4.10 *GenNest/Example*, after flattening.

iteration in which it actually executes *BODY*. In other words, *the processors still have to run through* BODY *and the rest of the loop nest in lockstep, but now they may be executing effectively different loop iterations.*

4.3.3 Optimizations

The above transformation is the most general, conservative one. It can be optimized for several special cases; one common case is that

1. $test_1$, $test_2$, and $init_2$ have no side effects, and

2. For each outer loop iteration, the inner loop is executed at least once.

Then we can safely transform the code into the simpler version shown in Figure 4.12. An operational proof is found again in Figure 4.11.

If it is also the case that

3. We can replace the guard $test_2$ with a test $done_2$ whether we are in the last inner iteration (for example, in **do** $var = lo,\ hi,\ stride$, we can replace $test \equiv (var \leq hi)$ with $done \equiv (var = hi)$),

then we can save the last execution of inc_2, as shown in Figure 4.13. The SIMDized equivalent *Example* of this version was shown in Figure 4.7.

4.4 Loop Flattening from the Compiler's Perspective

The discussion so far seems to advocate a certain style of SIMD programming for applications that can benefit from loop flattening, just as a certain style of programming emerged when vector machines became popular. However, this would be contrary to existing efforts to make programming independent from machine idiosyncrasies, as for example the development of the Fortran D language. For non-SIMD machines, it still seems natural and efficient to have the inner loop bodies contained in the inner loops, even though flattened loops should run well on these machines also. Therefore, we suggest to make loop flattening part of the optimizing repertoire of SIMD compilers.

Applicability is ensured whenever there are multiple loops fully contained in each other; *i.e.*, there are not several loops on the same nesting level. This can be easily derived from the abstract syntax tree. Furthermore, the normalized version always

Operation	B	C	D	Comments
$init_1$	B1	C1	D1	
$t_1 = test_1$	B2	C2		
if t_1 **then**	(B3)			$t_1 = true$
$init_2$	B4	C3	D2	Entered B3 **while**, C3 **if**
$t_2 = test_2$	B5	(C4), C5		Entered C4 **while**
L1: **if** t_2 **then**	(B6)	(C6)	(D3)	$t_1 = true$, $t_2 = true$
$BODY$	B7	(C14), C15	D4	Skipped C6 **while**; entered B6/C14 **if**, D3 **while**
inc_2	B8	C16	D5	
$t_2 = test_2$	B9	(C4), C5	(D6)	Entered C4 **while**
goto L1				
else				$t_1 = true$, $t_2 = false$
inc_1	B11	C7	D7	Skipped B6 **while**; entered C6 **while**, D6 **if**
$t_1 = test_1$	B12	C8		
if t_1 **then**	(B3)	(C9)		$t_1 = true$, $t_2 = false$
$init_2$	B4	C10	D8	Entered B3 **while**, C9 **if**
$t_2 = test_2$	B5	C11		
goto L1				
else		(C6), (C14)	D8	$t_1 = false$, $t_2 = false$
STOP		(C4)	(D3)	Skipped B3/D3 **while**
endif				Skipped C9 **if**, C6 **while**, C14 **if**, C4 **while**
endif				Executed spurious D8 (no side effects)
else		(C3), (C4)	D2	$t_1 = false$
STOP			(D3)	Skipped B3/C4/D3 **while**
endif				Executed spurious D2 (no side effects)

Figure 4.11 Operational proof of equivalence of unflattened *GenNest B*, flattened *GenNest C*, and optimized *GenNest D*. Control-flow statement labels are parenthesized. Horizontal lines delineate basic blocks.

	program *GenNest D*	**program** *Example D*
D1	$init_1$	$i = 1$
D2	$init_2$	$j = 1$
D3	**while** $test_1$	**while** $(i \leq K)$
D4	$BODY$	$X(i, j) = i * j$
D5	inc_2	$j = j + 1$
D6	**if not** $test_2$ **then**	**if not** $(j \leq L(i))$
D7	inc_1	$i = i + 1$
D8	$init_2$	$j = 1$
D9	**endif**	**endif**
D10	**endwhile**	**endwhile**

Figure 4.12 *GenNest/Example*, flattened and optimized.

```
program GenNest E

init_1
init_2
while test_1
    BODY
    if done_2 then
        inc_1
        init_2
    else
        inc_2
    endif
endwhile
```

```
program Example E

i = 1
j = 1
while (i ≤ K)
    X(i, j) = i * j
    if (j = L(i))
        i = i + 1
        j = 1
    else
        j = j + 1
    endif
endwhile
```

Figure 4.13 *GenNest/Example* after further optimization.

tests the loop guard $test_l$ before executing *BODY*, so we cover all loop constructs. The transformation itself is relatively straightforward; for example, there are no parameters to adjust, unlike in loop skewing. The first step of the transformation is to identify the three phases *init*, *test*, and *inc*.

while/do-while loops: The relevant phases can be identified from their position between the **while** and **endwhile** keywords. Since inc_2 and *BODY* stay together throughout the transformation, we actually do not need to separate these two phases.

do/forall loops: The phases can be derived directly from the loop header, as exemplified earlier.

Reducible goto loops: Similar to **while** loops, we can identify the phases by their position between labels and jumps.

After normalization, the introduction of flags t_l and the actual code rearrangement follow straightforwardly. As described in Section 4.3, we also can often detect opportunities for further optimizations, for example when we are transforming simple **do/forall** loops.

In evaluating *profitability*, we note that the additional overhead caused by loop flattening is, in the worst case, to manipulate two flags and to perform two conditional jumps. So we can relatively safely assume profitability whenever the inner loop bounds may vary across the processors.

As with many code transformations, the hardest problem in automating loop flattening is to determine its *safety*. A sufficient condition is that the loop into which we lift an inner loop body can be parallelized, which might be hard to detect, especially if indirect addressing occurs. However, this is already a necessary condition for parallelizing loops in general, and therewith a standard problem for parallelizing compilers [HKT92a]. The same technology developed there can be applied here.

When safety is ensured, either by user information (such as a **forall** loop header) or by "heroic dependence analysis," we expect that the systematic loop flattening transformation, as described in Section 4.3, can be implemented efficiently into compilers like the FORTRAN D compiler. This implementation is not part of this dissertation; however, Section 6.2 contains a performance study on the improvements gained when applying loop flattening manually.

The correctness of the distribution
is founded on the justice
of the scheme of cooperation ...
the principle of utility requires us to maximize
the algebraic sums of expectations
taken over all relevant positions.

— John Rawls (A Theory of Justice)

Chapter 5

Implementation Experience

So far, this dissertation stated a thesis (Section 1.3) and discussed issues, problems, and technologies regarding compiler support for the parallelization of irregular problems. This chapter describes an extension of the Fortran D compiler prototype [Tse93] to handle irregular problems as part of the overall validation. Other validation components are experiments (Chapter 6) and additional theoretical proofs (Appendix A).

The rest of this chapter is organized as follows. Section 5.1 gives a general overview of the compiler. Section 5.2 and Section 5.3 describe the analysis and code-generation phases, respectively. Section 5.4 concludes with a short overview of the object-oriented design methodology employed.

5.1 Overview

The Fortran D compiler can be divided into two phases: the analysis phase and the code-generation phase. The *analysis phase* parses the program, builds internal data structures, and analyzes the program, but does not modify it yet. The *code-generation phase* performs the actual code transformation by modifying the Abstract Syntax Tree (AST) and generates the final program by unparsing the AST. This implies that the same AST can be used for both source and target language. On the one hand, this strict separation might in some cases slightly increase intermediate storage and run-time requirements. On the other hand, not touching the source code until achieving full knowledge about the transformation to be done lengthens the validity of intermediate analyses, such as the control flow graph (CFG), static single assignment information (SSA), value numbers, and their links into the AST. This potentially decreases the need for reanalysis after intermediate code transformations, which in turn can speed up the overall compilation process. Most importantly, however, the clean separation enhances modularity of the compiler itself. This benefits the development and debugging process, and it allows easy integration into an interactive environment where the user wishes to make queries about a program without actually modifying it.

The `nbf` kernel in Figure 2.3 will be used as a running example. The `nbf` program, annotated with output statements not shown here, has been compiled and run on both a Sun work station and, after feeding it through the Fortran D compiler, on an iPSC/860 hypercube where it was linked with the Chaos communication library (see Section 7.1.2) and a memory allocation library; see Section 6.1 for experimental results. The code generated by the Fortran D compiler is shown verbatim in Figures 5.7, 5.8, and 5.9. However, to facilitate the discussion of the concepts that are at a somewhat higher level than for example calling protocols for the run-time support and the emulation of dynamic memory in Fortran 77D, we also provide a slightly abstracted version of the Fortran D output in Figures 5.1 and 5.2, which we will be mostly referring to.

5.2 The Analysis Phase

5.2.1 Symbolic analysis

Even though the Fortran D compiler can be used in batch mode as a stand-alone tool, it is part of the ParaScope programming environment [KMT91]. Therefore the compiler not only can be used within ParaScope, but it also takes advantage of the information and utilities provided by this environment. Besides basic facilities such as file I/O, lexing, parsing, and symbol table management, the environment also provides more advanced features, such as a framework for deriving interprocedural symbolic analysis. One simple example of the use of symbolic analysis in the `nbf` program is the proper handling of the symbolic constants `Natom` and `pMax` in array declarations, Fortran D directives, and loop bounds. Of particular importance with respect to the algorithms presented here is the following information, which is derived from a Fortran D program P and handed to the compiler.

The control flow graph, $G := (N, E)$, with nodes N and edges E. Among other functions, G serves as a basis for the Give-N-Take data-flow analysis framework described in Chapter 3 and therefore has to meet certain criteria, such as reducibility and lack of critical edges (see Section 3.3.4). Furthermore, G has to be annotated with Tarjan interval information to guide the data-flow phase. Figure 5.3 shows G for `nbf`; note that the current implementation generates one node for each statement instead of summarizing information for basic blocks.

```
      PROGRAM nbf

      INTEGER i, j, p, t, n$proc, Natom, pMax, Nstep
      PARAMETER (n$proc=8)
      PARAMETER (Natom=8000, pMax=250, Nstep=30)
      INTEGER inb(1000), partners(1000, pMax)
      REAL x(1000), f(1000), force, nbf_func, delta_func

C     General FORTRAN D variable declarations
      COMMON /FortD/ n$p, my$p
      INTEGER n$p, my$p, numnodes, mynode

C     Irregular FORTRAN D variable declarations
      INTEGER atomD$cnt, atomD$sched, atomD$tab, atomD$loc2proc(Natom)
      INTEGER x$sched, x$offsize, j$cnt, $init, i$

C     FORTRAN D initializations
      my$p = mynode()
      n$p = numnodes()
      IF (n$p .NE. 8) STOP

C     Initialize data
      CALL read_data(x, inb, partners)

C     Redistribute atomD according to coordinate values
      atomD$cnt = Natom / n$p
      Compute atomD$loc2proc, atomD$cnt from inb, x, atomD$cnt
      Compute atomD$tab from atomD$loc2proc, atomD$cnt
      Allocate atomD$loc2glob(atomD$cnt)
      Compute atomD$loc2glob, atomD$sched from atomD$tab, atomD$cnt
      Delete atomD$tab
      Compute atomD$tab from atomD$loc2glob, atomD$cnt
      Resize inb(atomD$cnt), x(atomD$cnt), f(atomD$cnt), partners(atomD$cnt,pMax)
      Shuffle inb, x, partners according to atomD$sched

C     Counting slice for j$cnt
      j$cnt = 0
      DO i = 1, atomD$cnt
         j$cnt = j$cnt + inb(i)
      ENDDO
      Allocate j$glob(j$cnt), j$loc(j$cnt)
C     Compute j$glob
      j$cnt = 0
```

Figure 5.1 Slightly simplified output of FORTRAN D compiler for `nbf` (continued in Figure 5.2).

```
      DO i = 1, atomD$cnt
         DO p = 1, inb(i)
            j$cnt = j$cnt + 1
            j = partners(i, p)
            j$glob(j$cnt) = j
         ENDDO
      ENDDO
      Compute x$sched, j$loc, x$offsize from atomD$tab, j$glob, j$cnt, atomD$cnt
      Delete atomD$tab
      Resize f(atomD$cnt + x$offsize)
      Resize x(atomD$cnt + x$offsize)

C     Loop over time steps
      DO t = 1, Nstep
         Gather x using x$sched

C        Reset forces to zero
         DO i = 1, atomD$cnt
            f(i) = 0
         ENDDO

C        Initialize buffer for reduction
         DO $init = atomD$cnt + 1, atomD$cnt + x$offsize
            f($init) = 0
         ENDDO

C        Compute forces
         i$ = 0
         DO i = 1, atomD$cnt
            DO p = 1, inb(i)
               i$ = i$ + 1
               j = partners(i, p)
               force = nbf_func(x(i), x(j$loc(i$)))
               f(i) = f(i) + force
               f(j$loc(i$)) = f(j$loc(i$)) - force
            ENDDO
         ENDDO
         Scatter_add f using x$sched

C        Push atoms
         DO i = 1, atomD$cnt
            x(i) = x(i) + delta_func(f(i))
         ENDDO
      ENDDO
      END
```

Figure 5.2 Slightly simplified output of FORTRAN D compiler for `nbf`, continued from Figure 5.1.

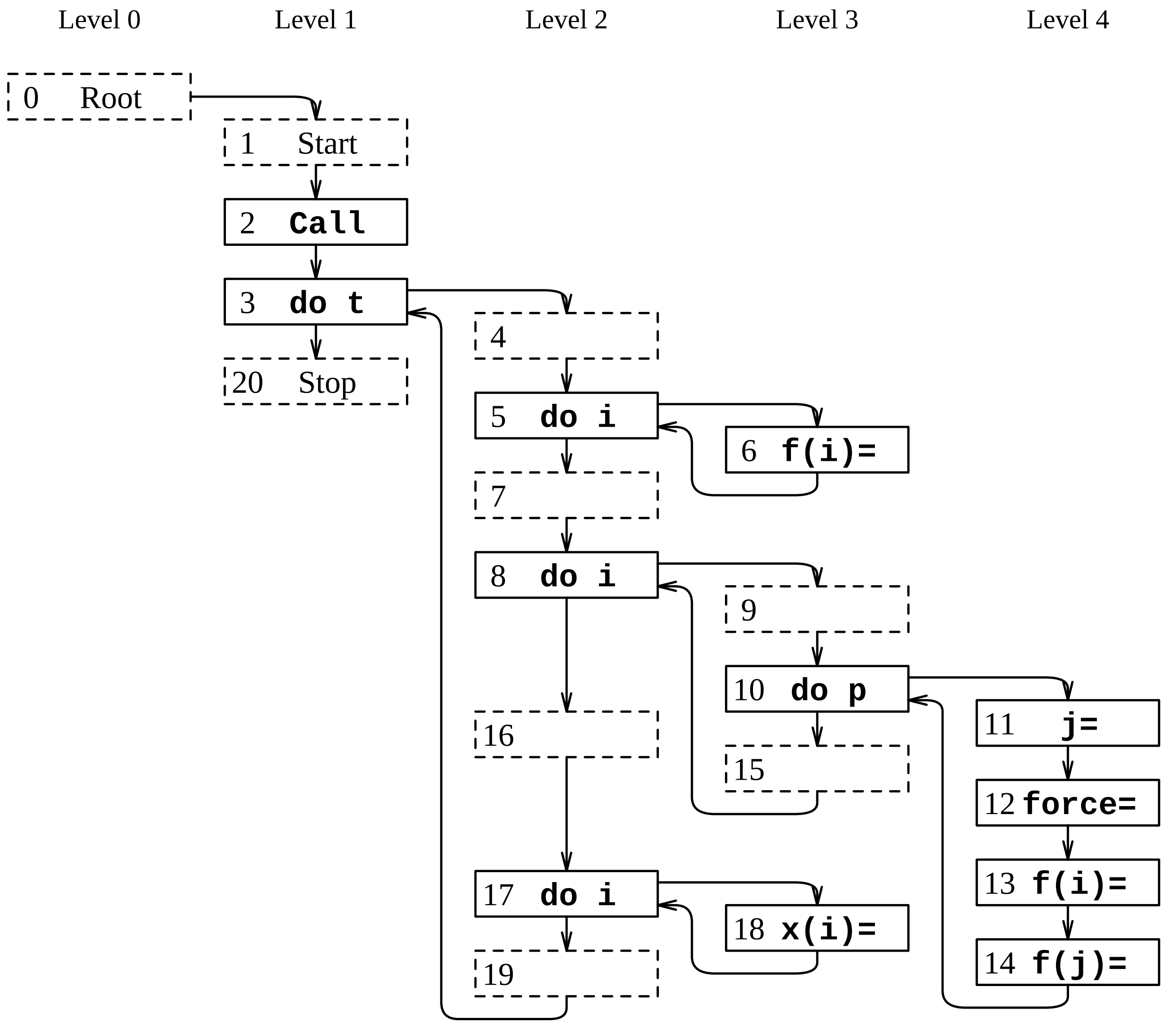


Figure 5.3 Flow graph G of `nbf` program. The loop nesting level in G increases from left to right. Node 0 is the root of G. Header nodes have their children attached to the right. Synthetic nodes, which do not directly correspond to statements in `nbf`, are dashed.

Value numbers, $VN := \{\ vn \mid e$ an expression in P, $vn = val(e)$, the value number of $e\}$. Value numbers are computed for both constant and non-constant expressions and are closely related to the SSA information [Hav93, Hav94]. They provide for example information about whether an expression is an immediate or auxiliary induction variable or a linear combination thereof. Each array reference's value number is a pair $\langle state, (sub_1, \ldots, sub_{rank})\rangle$, where $state$ represents the internal state of the array that gets altered with every modification of the array, and sub_i is the value number of the i-th subscript. This means that the array is viewed as a scalar with respect to modifications, but the actual subscripts are taken into account when comparing array expressions.

Note, however, that we do not only construct value numbers for individual subscripts, but also for whole subscript lists. Both types of subscript numbers are used; for example, inspector generation is mostly concerned with individual subscripts as described in Section 5.2.5, whereas communication (Section 5.2.4) depends on whole subscripts.

5.2.2 The regular part of Fortran D compiler

Just as there are distinct analysis and code-generation phases, there is also a fairly clean separation between those parts of the compiler that apply to both regular and irregular features of an application, and those parts that are specific to either kind. For brevity, the phases specific to irregular applications will be referred to as the "irregular compiler," whereas the rest of the Fortran D compiler that applies to all or just regular programs will be called the "regular compiler."

The irregular compiler performs its analysis after the regular compiler has performed interprocedural analysis, but before the regular compiler's intraprocedural analysis. An important class of interprocedural information provided by the regular compiler is reaching decomposition analysis, which propagates Fortran D specific decomposition information from callers to callees [HHKT92].

5.2.3 The data-flow universe for communication analysis

In preparation for analyzing the communication requirements of P, the irregular compiler first determines *IREFS*, the set of both regular and irregular references to arrays that are accessed irregularly somewhere, and then computes *KEYS*, the data-flow universe. Crucial to this phase is symbolic analysis, which can for example determine the equality of expressions even if they are syntactically different, across loops and other control flow constructs. An example of this can be seen in the code of Figure 3.2, where $x(a(k))$ and $x(a(l))$ can be recognized as identical based on the subscript value numbers. Symbolic analysis is also used to determine whether a subscript is irregular or not; the current heuristic classifies all subscripts that are not linear combinations of induction variables as irregular. A more formal specification of the computed information is shown in Figure 5.4. We will refer to the elements of *KEYS* also as *array portions*.

Note that to allow irregular references to multidimensional arrays, *SUB_VALS* contains value numbers of individual subscripts, whereas *KEYS* considers value numbers of whole subscript lists. Furthermore, in order to determine when messages can be combined, or when buffered data become stale because of non-local assignments, etc., the data-flow universe must be based on the comparison of actual memory locations, not values. Therefore, *KEYS* does not use value numbers for classifying data arrays, but instead their symbol table index. This is equivalent to syntactic comparisons plus aliasing and formal/actual parameter resolutions.

In `nbf`, the `j`s are considered irregular subscripts, resulting in $IARRS = \{\mathtt{x}, \mathtt{f}\}$. Therefore, all references to `x` and `f` are collected to build the data-flow universe.

5.2.4 Communication analysis

The communication model used by the Fortran D compiler is that each datum d has an owner $owner(d)$ (see Section 3.2.1). If a processor p referencing d owns d, it is assumed to have a valid copy of d at the point of reference. If p does not own d, it might either have to receive d from $owner(d)$, which together with the corresponding *Send* of $owner(d)$ is considered a global Read operation, or p might still have buffered a valid copy of d. An option not considered here is that p might receive d from any processor that has a valid copy of d [GS93].

Since the owner-computes rule is not strictly applied, any processor p may define d. If $p \neq owner(d)$ and d will be referenced by some processor q, $q \neq p$, then p must send d to $owner(d)$ after defining d. That, together with the corresponding receive of $owner(d)$, is considered a global Write.

Under the above constraints, the objective of communication placement is to communicate as little and as infrequently as possible (see Section 3.2). Small messages

$$
\begin{array}{rcl}
\mathit{IARRS} & := & \{x \mid x \text{ is referenced irregularly in } P.\} \\
\mathit{IREFS} & := & \{x(subs) \mid x \in \mathit{IARRS}, subs = (s_1, \ldots, s_r) \text{ a subscript list.}\} \\
\mathit{SUBS} & := & \{s \mid \exists x(\ldots, s, \ldots) \in \mathit{IREFS}.\} \\
\mathit{SUB\_VALS} & := & \{vn \mid vn \text{ is value number of some subscript } s \in \mathit{SUBS}.\} \\
\mathit{KEYS} & := & \{\langle st, vns, dist, iset\rangle \mid \text{ for some reference } x(subs) \in \mathit{IREFS}, \text{ it is} \\
& & \qquad st := \text{symbol table index of } x, \\
& & \qquad vns := \text{value number of subscript list } subs, \\
& & \qquad dist := \text{distribution of } x \text{ at point of reference}, \\
& & \qquad iset := \text{computational distribution of reference.} \\
& & \} \\
\mathit{REFS\_KEY}(key) & := & \text{set of references that match } key \in \mathit{KEYS}. \\
\mathit{REFS\_KEYS} & := & \text{set of } \mathit{REFS\_KEY}\text{s.}
\end{array}
$$

Figure 5.4 Description of the GIVE-N-TAKE-universe used for communication placement.

are combined into larger ones, and data buffered locally, due to proceeding READs or local definitions, are reused as long as they are valid.

The data-flow framework uses a binary lattice and represents its variables as bit vectors, one bit for each $key \in KEYS$. The position within the bit vector constitutes the *id* for each *key*. In the following, the "*id* of *ref*" denotes the *id* assigned to the key matching a reference *ref*. Similarly, the definition, use, etc. of an *id* refers to the definition, use, etc. of a reference with the key corresponding to *id*. Figure 5.5 describes the information that is computed for each node $n \in N$.

Here the *subscript dependence set* refers to the data that a subscript depends on. This can be thought of as the set of data that are referenced when inspecting for a subscript; a more detailed discussion can be found elsewhere [DSvH93]. For example, a subscript that is itself an indirection-array lookup depends on the indirection array. In `nbf`, it is `x(j)` $\in$ REF(12)¶, IND(2), and `f(j)` $\in$ REF(14), DEF(14), ADD(14), RED(14), IND(2); there are additional entries for the `x(i)`s and `f(i)`s.

¶Read as: "The key associated with the reference `x(j)` is contained in the REF bit vector at node 18, which corresponds to the assignment to `force` in `nbf`."

$$\begin{aligned}
\mathsf{REF}(n) &:= \{id \mid n \text{ references } id\},\\
\mathsf{DEF}(n) &:= \{id \mid n \text{ defines } id\},\\
\mathsf{ADD}(n) &:= \{id \mid n \text{ adds to } id\},\\
\mathsf{MULT}(n) &:= \{id \mid n \text{ multiplies to } id\},\\
\mathsf{RED}(n) &:= \mathsf{ADD}(n) \cup \mathsf{MULT}(n),\\
\mathsf{IND}(n) &:= \{id \mid n \text{ redefines the subscript dependence set of } id\}.
\end{aligned}$$

Figure 5.5 The information that is computed for each node $n \in N$.

Based on this local information, several instances of Give-N-Take are solved, one instance for each kind of communication. The communication types considered are global Reads (or GATHERs), global Writes (SCATTERs), and global Add and Mult reductions (SCATTER_ADD, SCATTER_MULT). The extension towards additional reduction types is straightforward. Separating *Send* and *Recv* operations for each type of communication can expose opportunities for hiding communication latencies by overlapping them with computation. The Give-N-Take mechanism accommodates this by providing both an Eager and a Lazy solution for all communication instances, where one solution indicates where to place the *Send*s and the other computes where to place *Recv*s. However, the Chaos communication library generates *Send*s and *Recv*s internally and presents them as monolithic entity; *i.e.*, a single Chaos call spawns complete *Send*/*Recv* pairs.

Each Give-N-Take instance is initialized by assigning bit-vector values to the variables TAKE_{init}, STEAL_{init}, and GIVE_{init} for each node $n \in N$. For example, the Read instance is initialized as shown in the first three equations of Figure 5.6. This initialization reflects the following: data that are referenced but not reduced to have to be buffered, redefining data or indirection arrays blocks Reads, and Reads come "for free" by local definitions. Here $\mathsf{DEF}(n)^{\circ}$ and $\mathsf{DEF}(n)^{\cap}$ are the data that are either "touched" or "contained" (*i.e.*, partially or fully enclosed) by references in $\mathsf{DEF}(n)$ [HKK+92]. For example, since `i` and `j` cannot be proven to be disjoint subscript ranges in `nbf`, it is $\{\mathtt{x(i)}\}^{\circ} = \{\mathtt{x(i)}, \mathtt{x(j)}\}$.

After initialization, each instance is solved as either a Forward Give-N-Take problem, which places communication before computation (as needed for Reads), or

$$\begin{aligned}
\textsc{Read}.\mathsf{TAKE}_{init} &:= \mathsf{REF} \setminus \mathsf{RED},\\
\textsc{Read}.\mathsf{STEAL}_{init} &:= \mathsf{IND} \cup \mathsf{DEF}^{\circ},\\
\textsc{Read}.\mathsf{GIVE}_{init} &:= \mathsf{DEF}^{\cap}.
\end{aligned}$$

$$\begin{aligned}
\textsc{Write}.\mathsf{TAKE}_{init} &:= \mathsf{DEF} \setminus \mathsf{RED},\\
\textsc{Write}.\mathsf{STEAL}_{init} &:= (\textsc{Read}.\mathsf{GEN}_{\mathrm{in}} \cup \mathsf{REF} \cup \mathsf{RED})^{\circ},\\
\textsc{Write}.\mathsf{GIVE}_{init} &:= \emptyset.
\end{aligned}$$

$$\begin{aligned}
\textsc{Add}.\mathsf{TAKE}_{init} &:= \mathsf{ADD},\\
\textsc{Add}.\mathsf{STEAL}_{init} &:= (\textsc{Read}.\mathsf{GEN}_{\mathrm{in}} \cup (\mathsf{REF} \setminus \mathsf{ADD}))^{\circ},\\
\textsc{Add}.\mathsf{GIVE}_{init} &:= \emptyset.
\end{aligned}$$

Figure 5.6 Initializations of TAKE_{init}, STEAL_{init}, and GIVE_{init} for the placement of READ, WRITE, and ADD communication operations.

as a BACKWARD problem, which places communication after computation (WRITEs and reductions). The results of the data-flow analysis reside for each node $n \in N$ in $\mathsf{RES}_{in}(n)$, which contains all references that have to be communicated on entry to n, and $\mathsf{RES}_{out}(n)$, on exit from n. The actual data-flow equations, formal correctness and optimality criteria, etc., are described in Chapter 3.

A minor, but in practice interesting point is that in some cases communication might be placed at *synthetic nodes*, *i.e.*, nodes that correspond to not-yet existing basic blocks (see Section 3.5.4). For example, if there is an **if-then** without a matching **else**, then breaking critical edges generates a synthetic node corresponding to an empty **else** branch. This may require creating new basic blocks during code generation. However, this can sometimes be avoided at no extra run-time cost by shifting communication from synthetic nodes to other nodes. For example, if a loop is guarded by a condition, then synthetic nodes are introduced for the **else** branch and between the loop and the merge after the condition; any subsequent READ that is blocked by the loop will be placed on both of these synthetic nodes instead of the merge after the guard. To take advantage of such opportunities for code simplification, an additional data-flow phase post-processes the $\mathsf{RES}_{in/out}$ results into new variables,

$\mathsf{GEN}_{in/out}$, that try to move results to non-synthetic nodes and to minimize the need for additional basic blocks.

Another interesting problem is that while global READs are shifted up and global WRITEs are shifted down, they should not be moved past each other if they communicate non-disjoint data. This can be satisfied by first solving the READ problem, and then initializing the WRITE problem. This and an example for a reduction initialization are shown in the middle and lower equations of Figure 5.6, respectively. In `nbf`, it is for example `x(j)` $\in$ READ.$\{\mathsf{TAKE}_{init}(12), \mathsf{STEAL}_{init}(2), \mathsf{RES}_{in}^{eager}(4), \mathsf{GEN}_{in}^{eager}(5)\}$; we also have `f(j)` $\in$ ADD.$\{\mathsf{TAKE}_{init}(14), \mathsf{STEAL}_{init}(2), \mathsf{STEAL}_{init}(6), \mathsf{RES}_{in}^{eager}(16), \mathsf{GEN}_{in}^{eager}(8)\}$.

5.2.5 Inspectors

Inspectors have to be generated whenever runtime resolution is required for determining which data have to be communicated.

The set of inspectors is computed as follows. For each control flow graph node that contains communication of a certain type, such as a READ_{Send} operation, let *Svals* be the set of all subscript value numbers that need to be inspected for this communication; let *SVALS* be the set of all *Svals* sets. For each $Svals \in SVALS$, one schedule *Sched*(*Svals*) will be created; let *SCHED* be the set of all *Sched*s. Each schedule $Sched \in SCHED$ will be computed in an inspector placed at $target(Sched)$, which is computed as the header of the outermost loop enclosing the least common ancestor of all nodes containing communication statements requiring *Sched*. Let $Insp(n)$ be the set of all schedules *Sched* for which $target(Sched) = n$. Let $INSP := \{(n, Insp(n)) \mid Insp(n) \neq \emptyset\}$; in `nbf`, it is $INSP = \{(3, \{\langle \textit{Schedule for } \{\texttt{j}\}\rangle\})\}$.

Note that this strategy imposes some restrictions that might result in unnecessary re-inspections. For a more general approach, see Das [Das94].

5.2.6 Executors

Executors are slightly modified versions of regions in P that contain irregular references. Part of the modifications is to replace irregular subscripts by references to trace arrays (see Section 5.3.3).

The use of trace arrays requires insertion of counters for indexing. For generating code to initialize and increment counters and for eliminating duplicate counters, executors are collected as follows. For each $s \in SUBS$, let $limit(s) = n \in N$ s.t.

n corresponds to the beginning of the inspector for s. This node may for example be the header of the outermost loop enclosing s that will be copied into the inspector. Let $Exec(n)$ be the set of all subscripts s with $limit(s) = n$. Let $EXEC :=$ $\{(n, Exec(n)) \mid Exec(n) \neq \emptyset\}$; in `nbf`, it is $EXEC = \{(8, \{$`js` in nodes 12 and 14$\})\}$.

5.3 The Code-Generation Phase

After the compiler has finished the analysis phase, the AST is modified to transform the Fortran D program into a message-passing node program.

5.3.1 The regular compiler

An important transformation performed by the regular compiler is the conversion of global name space references to distributed arrays into local-name-space references to local arrays. For `BLOCK` distributions and simple subscripts, this can be achieved by *loop bounds reduction*, which is also a convenient and efficient way to exploit data parallelism. In the `nfb` code, this transformation is applied to all `i` loops, whose upper bound is reduced from `Natom` (= `8000`) to `1000`. This corresponds to using eight processors, as indicated by the assignment to `n$proc` (line 4 in Figure 2.3). If no value is specified for `n$proc`, then four processors are used by default. Note how the use of induction variables outside of subscripts in reduced loops may necessitate the need for adding a processor dependent offset [Tse93].

An additional task currently performed by the regular compiler is the communication generation for regular subscripts (none such communication is needed in `nbf`). This process does not make use of the Give-N-Take analysis yet, one of the reasons being that Give-N-Take so far does not include dependence analysis. That alone could be changed relatively easily (see Section 3.6), but there are also other issues that deserve special attention when generating regular communications from the information provided by Give-N-Take. For example, message tags have to be generated for matching separate *Send*s and *Recv*s, which might require the construction of *Send*/*Recv* equivalence classes in case of complicated control flow. Furthermore, if the analysis indicates that data are sent together but received separately, data have to be grouped accordingly.

5.3.2 Value-based mappings

Each value-based mapping directive D results in certain tasks that the compiler, in the current implementation, performs at the location of the directive itself:

1. Generate code for calculating the new distribution as follows.

   (a) Compute an array that maps local indices (based on the initial distribution) to processor numbers, *D.loc2proc*, and the resulting number of owned data, *D.cnt*, by calling a partitioner (line 27 in Figure 5.1). This partitioner applies the strategy specified by the programmer or a default specified by the compiler to the values and, if specified, weights provided by the user and generates a distribution. The partitioner tries to distribute the data such that both high locality and good load balance are achieved; in the presence of weights, the overall weight assigned to each processor should be similar. Before partitioning, `atomD$cnt` is initialized (line 26) according to the initial, regular distribution.

   (b) Based on *D.loc2proc* and *D.cnt*, compute a translation table as described in Section 2.2.1, *D.tab* (line 28). *D.tab* is actually a pointer to a data structure internal to the run-time library; within FORTRAN it is declared as an integer. This mechanism is also used for other internal structures.

   (c) Allocate an array that maps local to global indices, *D.loc2glob*, of size *D.cnt* (line 29).

   (d) Call a remapper (line 30) to compute a mapping from local to global indices, *D.loc2glob*, and a communication schedule *D.sched* from *D.tab* and *D.cnt*. This schedule is used in the communication statements that reshuffle coordinates and pair list data according to the new distribution (line 34).

   (e) Based on *D.loc2glob* and *D.cnt*, recompute the translation table, *D.tab* (line 32).

2. Let ARR be the set of arrays that are aligned to the decomposition being distributed. (In our example, ARR can be determined at compile time; in general, however, this may require run-time resolution.) Resize each $arr \in ARR$ according to *D.cnt* (line 33).

As already indicated in Section 2.2.2, this implies a need for dynamic memory allocation capabilities. The FORTRAN D compiler emulates these by converting arrays that are dynamically allocated or resized to work array accesses. (However, for the sake of readability, Figures 5.1 and 5.2 show the code before memory allocation). The offsets into these work arrays are generated at run time by calls to a separate memory allocation library; see Section 5.3.9 for more details.

3. Let *LIVE* be the set of arrays $arr \in ARR$ that are live at the location of D. For each $arr \in LIVE$, generate code for communicating the elements of *arr* from their old owners to their new owners (line 34). In the example, we do not need to communicate `f` since forces have not been computed yet.

5.3.3 Trace arrays

The current strategy for collecting off-processor references is to record each individual subscript in a *trace array* [DSvH93]. These arrays contain traces of the subscripts encountered during inspection and will be localized from global to local name space.

In our example, a trace array `j$glob` is created for subscript `j` (lines 41–50 in Figure 5.2). This trace array is first generated in global name space and then converted into local name space (`j$loc`, line 51). For example, consider subscript `j` in the loop nest in lines 30–37 of the original, sequential program in Figure 2.3. Assume that for the original program, the value of `j` in the 15-th iteration of the inner loop (say, for `i = 1` and `p = 15`) is `38`, and that after parallelization this loop iteration will be executed on processor 2, as the third local iteration. Then the third *global trace element* on processor 2 will be `38`; *i.e.*, `j$glob(3) = 38`. Furthermore, assume that array `x` is distributed such that `x(38)` is owned by processor 2 and referred to by that processor as `x(12)`, then the third *local trace element* on processor 2 will be 12 (*i.e.*, `j$loc(3) = 12`). If instead `x` is distributed such that `x(38)` is owned by some other processor, then processor 2 has to communicate that element in and will therefore extend its local version of `x` to create *buffer space* for accommodating a copy of `x(38)`. For example, if processor 2 owns 20 elements of `x`, it will refer to these *local elements* as `x(1), ... , x(20)`. If `x(38)` is the 7-th non-local element that processor 2 has to buffer due to some reference to it, then it will referred to global `x(38)` as local `x(27)`, and it will be `j$loc(3) = 27`.

Trace arrays are one of several possible options to perform the name space conversion. In the presence of high subscript reuse, for example, hash tables would be a more complicated but also more space efficient alternative. Note that since the name-space translation is already implicit in the trace array, the trace-array approach results in a relatively fast executor. However, the space requirements for storing the traces are proportional to the total number of references, instead of just the number of elements referenced. Compile-time analysis can be used to shorten traces; in the example, the compiler determined that the reference pattern is the same in each time step and therefore did not include the time-stepping loop in the inspector. However, the traces may still become too space consuming, in which case more space saving alternatives, such as a hash table combined with name-space translations on the fly, may be used [DSvH93].

5.3.4 Inspectors

The inspector code has to perform the following tasks:

1. If the size of a subscript trace is not known at compile time, then a *counting slice* has to be generated and the trace array has to be allocated. A significant implication of this scenario is the need for dynamically allocatable memory. For example, in `nbf` the size of the pairlist storing all pairs of atoms within R_{cut} depends on the physical properties of the molecule to be simulated and is not known at compile time. Therefore, a counting slice is generated to compute `j$cnt` ($= \sum_{\mathtt{i=1}}^{\mathtt{2000}}$ `inb(i)`; see lines 37–40 in Figure 5.1) and to allocate trace arrays for both the global (`j$glob`) and local (`j$loc`) name space (line 41).

2. Collect subscript traces, in global name space (lines 43–50).

3. Generate CHAOS calls to `reglocalize()` for computing a communication schedule (`x$sched`), for counting the number of non-local elements (`x$offsize`), and for transforming the subscript trace from global to local name space (line 51).

4. Resize the data arrays subscripted irregularly (`x` and `f`) to accommodate the non-local data (lines 53–54), as computed by `reglocalize()`.

Apart from generating the code slices for computing the subscript traces (which in the current prototype is only implemented for relatively simple cases without complicated internal control flow), most of the code generation is relatively straightforward. A few implementation details are given below.

- When merging the code for generating multiple traces (which is not an issue in `nbf`), transformations such as loop fusion are applied if possible.

- As with all variables generated by the compiler (indicated by a `$` as part of the name), the identifiers for schedule, subscript trace, etc. must not collide with earlier declarations. Furthermore, on the one hand they should resemble the variables they are related to in the original program, and on the other hand, they should be reused as much as possible. For example, in `nbf` the communication schedule `x$sched` and off-processor count `x$offsize` are used for both `x` and `f`.

- Another issue related to compiler generated variables is the use of trace indices and counters providing the trace size for localization. A variable holding the size of the trace is needed if the number of iterations of the slice is not known. An auxiliary induction variable for indexing the trace array is required in case the loop is nested or not in normal form. The slice for `j$loc` needs both, since the size of the pairlist is not known, and since the slice is a nested loop; `j$cnt` is used to serve both purposes.

- Comments are generated to delineate the generated inspectors and the subscripts they are inspecting (similarly for counting slices, executors, and buffer initializations). This kind of information is not only useful for making the generated output more readable, but also for later debugging support. Subscripts occurring more than once are annotated with a count in brackets.

- If one schedule is computed for multiple subscripts, then each subscript needs its own subscript trace. If this is the case, then a 2-dimensional trace array will be constructed, where the first dimension selects a trace, and the second dimension is indexed by the counter. Figure 5.11 illustrates this for the mesh kernel (see also Section 2.1.2).

5.3.5 Communication statements

For each node $n \in N$, the GEN sets of the different GIVE-N-TAKE instances indicate the communication to be generated. Here we have to distinguish between communication to be prepended (indicated by GEN_{in} for a FORWARD problem and GEN_{out} for a BACKWARD problem) and communication to be appended (GEN_{out} and GEN_{in}, respectively). Furthermore, if we wish to generate separate *Send*s and *Recv*s, then we have to consider both GEN^{eager} and GEN^{lazy}. For example, the *Send*s to be prepended to n are given by the *communication set*

$$\textsc{Prepend}_{Send} = \textsc{Read}.\mathsf{GEN}_{in}^{eager} \cup \textsc{Write}.\mathsf{GEN}_{out}^{lazy} \cup \textsc{Add}.\mathsf{GEN}_{out}^{lazy} \cup \textsc{Mult}.\mathsf{GEN}_{out}^{lazy}.$$

For each generated communication set, the data to be actually communicated are indicated by the st and vn components of the keys contained in the bit vector, where st indicates the data array and vn gives a range of subscripts to communicate. For irregular communications, vn is annotated with the name of the communication schedule to use. Since irregular references are communicated using the Chaos routines, *Send*s and *Recv*s are not separated, and just the Eager solution is used for placing communication.

In `nbf`, the generated communications are $\textsc{Prepend}(5) = \textsc{Read}(\{$`x(j)`$\})$ and $\textsc{Append}(8) = \textsc{Add}(\{$`f(j)`$\})$. This translates into an `fgather` of `x(j$loc(1:j$cnt))` before the force initialization (line 58) and an `fscatter_add` of `f(j$loc(1:j$cnt))` after the executor (line 81).

An optimization that is not implemented yet but at least conceptionally fairly straightforward is to use *incremental schedules* for pruning messages in case at least some of the data covered by a reference are already locally available [DPSM91, HKK+92]. The information about what data are already available is stored for each node $n \in N$ in $\textsc{Read}.\mathsf{GIVEN}(n)$.

5.3.6 Reduction initialization

An issue specific to reduction communications (such as Add and Mult) is the need for initializing buffer space for non-local data (assigning `0` for Add, `1` for Mult). The heuristic used for placing this initialization code for a reduction instance of Give-N-Take is as follows. Let the "local reduction" of a node $n \in N$ be the variables that are affected by a reduction operation in n itself or in one of the children of n (*i.e.*, in a statement in a loop headed by n). The local reduction is computed in the Give-N-Take variable $\mathsf{TAKE}(n)$. Let $\mathsf{TAKE}_{header}(n)$ be the local reduction of the header node of the loop directly enclosing n, if n is in a loop, let it be $\emptyset$ otherwise. The set of data for which initialization code should be prepended to n is then given by $\mathsf{TAKE}(n) \setminus \mathsf{TAKE}_{header}(n)$.

In the `nbf` example, it is `f(j)` $\in \textsc{Add}.\mathsf{TAKE}(14)$. However, since `f(j)` is not "stolen" within the enclosing `p` loop, `f(j)` $\in \textsc{Add}.\mathsf{TAKE}_{header}(14) = \textsc{Add}.\mathsf{TAKE}(10)$

holds as well. Similarly, `f(j)` is in the local reduction sets of the header of the enclosing `i` loop: `f(j)` $\in$ ADD.TAKE(8). However, `f(j)` is "stolen" in nodes 6 and 5 enclosed in the `t` loop (forces are reset to zero at the beginning of each time step); therefore, `f(j)` $\notin$ ADD.TAKE(3). This implies `f(j)` $\in$ ADD.TAKE(8) $\setminus$ ADD.TAKE$_{header}$(8); therefore `f(atomD$cnt+1:atomD$cnt+x$offsize)` is initialized on entry to node 8, which is the header of the `i` loop. The corresponding initialization loop appears in lines 66–68.

5.3.7 The actual computation

After distributing data and prefetching off-processor references, the actual computation can be performed (lines 57–87 in Figure 5.2). The following are some of the issues arising here.

- When reducing loop bounds to parallelize a loop based on the owner-computes rule, the number of local elements, *D.cnt* (computed in line 27), has to be retrieved. In the `nbf` kernel, this is the case in the `i` loops.
- To regenerate the global iteration index from the local name space index, the array *D.loc2glob* (from line 30) has to be consulted.
- To map global indices to processor numbers and local indices, for example when printing a certain data element, a dereferencing call using the translation table *D.tab* must be generated.
- Like most compiler transformations, converting a section of code into an executor may expose opportunities for further optimizations. In our example, line 75 could be removed by Dead Code Elimination, since `j` is not used as a subscript any more (see above). We could also merge the two loops for initializing forces (lines 61–63) and for clearing the reduction buffer (lines 66–68).

5.3.8 Executors

The major tasks when generating executors are the conversion of irregular subscripts to trace-array lookups, and, related to that, the generation of counters (the *sub-subscripts*) to index the trace arrays. The current strategy for generating sub-subscripts is as follows. Pick an executor $Exec(n)$ from $EXEC$, and a subscript $s \in Exec(n)$. If n is the header of a loop in normal form and s is enclosed by n directly, then use the induction variable of n as a sub-subscript. Otherwise, an explicit counter is needed. If there already exists a counter at n for the loop directly enclosing s, then use it as a sub-subscript, otherwise generate a new one to use as a sub-subscript. After determining the sub-subscript, the value number of s determines which trace array to use, and the subscript can be converted. Note that we might need several counters for one executor, for example for different, imperfectly nested loops, or for references at different loop nesting levels. Note also that in order to allow the use of such indexing variables within loop headers, they have to be initialized to `1` instead of `0`. This in turn prohibits incrementing them at the beginning of the loop body; instead they have to be incremented at the end, which complicates code generation especially for loops with internal control flow.

In `nbf`, the executor for `x(j)` and `f(j)` (lines 71–80) needs a counter `i$`, because the references are nested two levels deep within the executor. The value number of `j` is the same throughout the executor and maps to the trace array `j$loc`. Consequently, `x(j)` and `f(j)` are converted to `x(j$loc(i$))` (line 76) and `f(j$loc(i$))` (line 78), respectively.

5.3.9 Dynamically allocated arrays

Dynamic array allocation is handled by calls to external library routines, such as `ialloc()`, `iresize()`, and `free()`. These routines manage space provided by some large work arrays and provide offsets into them for each allocated array. In the current prototype, the work arrays are of fixed size, specified either as a parameter when invoking the compiler or by a default value. This rather crude scheme could be replaced by a more advanced library that does not require fixed size work arrays. Another option not addressed here is to distinguish between a copying and a non-copying resizing operation.

In `nbf`, work arrays are the integer array `i$wrk` and the floating point array `f$wrk`. A reference such as `x(j$loc(i$))` becomes `f$wrk(x$ind+i$wrk(j$loc$ind+i$))`. This scheme also has to consider multidimensional arrays, such as `partners` in the example. In the prototype compiler, multidimensional arrays reference the work array section assigned to them by performing array arithmetic explicitly. For example, a reference to `partners(i, p)` becomes something like `i$wrk(partners$ind + p + (i - 1) * 1000)`. Note that this does not increase the overall instruction count; it only makes explicit the index calculations that are normally hidden from the programmer. Note how this explicit array arithmetic exposes the need for other classical optimizations, such as common subexpression elimination or loop invariant code motion.

In the FORTRAN D compiler, a separate pass, which can also be used as a standalone tool, converts all dynamic array references into work array references. It also introduces some additional bookkeeping code, for example to store the size of each dynamic array (the variables suffixed by `$size`). Separating explicit dynamic memory handling from the rest of code generation sometimes results in suboptimal code; for example, there are some redundant assignments to size variables if the same size is used for multiple arrays. This, however, is just one of several examples where the compiler relies on later, fairly well-understood optimizations, in order to achieve the modularity necessary for efficient use of high level transformations (see also Section 5.3.7).

5.3.10 Final notes on the compiler output

As a reference, Figures 5.7, 5.8, and 5.9 show the code generated by the FORTRAN D compiler, without any abstractions or cosmetic changes. Some comments on the differences between this code and the simplified version in Figures 5.1 and 5.2 follow.

- References to dynamically allocated arrays have been converted into work array accesses.

- Variable names containing a `$` and comments preceded by `--<<` and succeeded by `>>--` are generated by the compiler.

- A `Makefile` runs the FORTRAN D compiler and feeds its output through the native compiler of the target machine. However, we also want to be able to include statements in the original source that are visible to the native target compiler, but not to the parser of the FORTRAN D compiler. For example, we would like to use `implicit none`, which currently is not supported by PARASCOPE. For that purpose one can wrap the statements to be hidden into comments, which are then uncommented by a script in the `Makefile` before native compilation. (Actually, the compiler currently generates an `implicit none` comment automatically.) The same technique is used to pass FORTRAN D directives from the user to the compiler.

  There exist different kinds of "significant comments" in the current implementation:

```
       program nbf
C        --<<F77:implicit none
C        --<< OPTIONS: skip_irreg: 0, code_before_reg: 0, do_all_arra
C        ys: 0, split_comm: 0, save_irreg: 1, gen_high_level: 0 >>--

         integer i, j, p, t, n$proc, natom, pmax, nstep
         parameter (natom = 8000, pmax = 250, nstep = 30)
         integer inb(1), partners(1)
         real x(1), f(1), force, nbf_func, delta_func
C
C        --<< Fortran D variable declarations >>--
         common /FortD/ n$p, my$p, my$pid
         integer i$glo, n$p, my$p, my$pid, numnodes, mynode, mypid
C

C        --<< Fortran D/irreg variable declarations >>--
         integer j$loc(1), j$glob(1), atomD$loc2glob(1), atomD$loc2proc(n
     *atom), i$wrk(500000)
         integer x$sched, x$offsize, j$cnt, $init, i$, atomD$cnt, atomD$s
     *ched, atomD$tab, init_ttable_with_proc, build_translation_table, $
     *newsize, i$type, ialloc, iresize, f$type, falloc, fresize, atomD$l
     *oc2glob$ind, atomD$loc2glob$size, j$glob$ind, j$glob$size, j$loc$i
     *nd, j$loc$size, inb$ind, inb$size, x$ind, x$size, f$ind, f$size, p
     *artners$ind, partners$size
         parameter (i$type = 1, f$type = 2)
         real f$wrk(500000)
C        --<< END Fortran D/irreg variable declarations >>--

C        --<< Fortran D initializations >>--
         call PARTI_setup()
         call iputsize(500000, i$wrk)
         inb$size = 1000
         inb$ind = ialloc(i$type, inb$size) - 1
         partners$size = 250000
         partners$ind = ialloc(i$type, partners$size) - 1
         call fputsize(500000, f$wrk)
         x$size = 1000
         x$ind = ialloc(f$type, x$size) - 1
         f$size = 1000
         f$ind = ialloc(f$type, f$size) - 1
         n$p = numnodes()
         if (n$p .ne. 8) stop
         my$p = mynode()
         my$pid = mypid()

C        Fortran D directives

C        Initialize data
C        --<< gather Send/Recv {[x(1:atomD$cnt)]} >>--
         call read_data(x, inb, partners)

C        Redistribute atomD according to coordinate values
C        --<< Redistribute decomposition "atomD" >>--
         atomD$cnt = natom / n$p
         call fCoorWeighBisecMap(atomD$loc2proc(1), i$wrk(inb$ind + 1), a
     *tomD$cnt, 1, f$wrk(x$ind + 1))
```

Figure 5.7 The `nbf` program after compilation by the FORTRAN D compiler, Part 1 of 3.

```
      atomD$tab = init_ttable_with_proc(1, atomD$loc2proc(1), atomD$cn
     *t)
      atomD$loc2glob$size = atomD$cnt
      atomD$loc2glob$ind = ialloc(i$type, atomD$loc2glob$size) - 1
      call remap_reg(atomD$tab, 1, atomD$sched, i$wrk(atomD$loc2glob$i
     *nd + 1), atomD$cnt)
      call free_table(atomD$tab)
      atomD$tab = build_translation_table(1, i$wrk(atomD$loc2glob$ind
     *+ 1), atomD$cnt)
C     --<< Resize "inb", "x", "f", "partners" >>--
      $newsize = atomD$cnt
      inb$ind = iresize(i$type, inb$ind + 1, inb$size, $newsize) - 1
      inb$size = $newsize
      $newsize = atomD$cnt
      x$ind = iresize(f$type, x$ind + 1, x$size, $newsize) - 1
      x$size = $newsize
      $newsize = atomD$cnt
      f$ind = iresize(f$type, f$ind + 1, f$size, $newsize) - 1
      f$size = $newsize
      $newsize = atomD$cnt * 250
      partners$ind = iresize(i$type, partners$ind + 1, partners$size,
     *$newsize) - 1
      partners$size = $newsize
C     --<< Shuffle "inb", "x", "partners" >>--
      call igather(i$wrk(inb$ind + 1), i$wrk(inb$ind + 1), atomD$sched
     *)
      call fgather(f$wrk(x$ind + 1), f$wrk(x$ind + 1), atomD$sched)
      call ngather(i$wrk(partners$ind + 1), i$wrk(partners$ind + 1), a
     *tomD$sched, 1000)
C     --<< END Redistribute >>--

C     --<< Inspector for [3*]j$loc(1:j$cnt) >>--
C     --<< Counting slice for j$cnt >>--
      j$cnt = 0
      do i = 1, atomD$cnt
        j$cnt = j$cnt + i$wrk(inb$ind + i)
      enddo
      j$glob$size = j$cnt
      j$glob$ind = ialloc(i$type, j$glob$size) - 1
      j$loc$size = j$cnt
      j$loc$ind = ialloc(i$type, j$loc$size) - 1
C     --<< END Counting slice >>--
      j$cnt = 0
      do i = 1, atomD$cnt
        do p = 1, i$wrk(inb$ind + i)
          j$cnt = j$cnt + 1
          j = i$wrk(partners$ind + i + (p - 1) * 1000)
          i$wrk(j$glob$ind + j$cnt) = j
        enddo
      enddo
      call localize(atomD$tab, x$sched, i$wrk(j$glob$ind + 1), i$wrk(j
     *$loc$ind + 1), j$cnt, x$offsize, atomD$cnt, 1)
      call free_table(atomD$tab)
      $newsize = atomD$cnt + x$offsize
      x$ind = iresize(f$type, x$ind + 1, x$size, $newsize) - 1
      x$size = $newsize
      $newsize = atomD$cnt + x$offsize
      f$ind = iresize(f$type, f$ind + 1, f$size, $newsize) - 1
      f$size = $newsize
C     --<< END Inspector >>--
```

Figure 5.8 The `nbf` program after compilation by the FORTRAN D compiler, Part 2 of 3.

```
C     Loop over time steps
      do t = 1, nstep

C       Reset forces to zero
C       --<< gather Send/Recv {x(j$loc(1:j$cnt))} >>--
        call fgather(f$wrk(x$ind + atomD$cnt + 1), f$wrk(x$ind + 1), x
     *$sched)
        do i = 1, atomD$cnt
          f$wrk(f$ind + i) = 0
        enddo
C       --<< scatter Send/Recv {[f(1:atomD$cnt)]} >>--

C       --<< scatter_add initialization for {f(j$loc(1:j$cnt))} >>--
        do $init = atomD$cnt + 1, atomD$cnt + x$offsize
          f$wrk(f$ind + $init) = 0
        enddo
C       Compute forces
C       --<< Executor for x(j), [2*]f(j) >>--
        i$ = 0
C       execute (i) on_home f(i)
        do i = 1, atomD$cnt
          do p = 1, i$wrk(inb$ind + i)
            i$ = i$ + 1
            j = i$wrk(partners$ind + i + (p - 1) * 1000)
            force = nbf_func(f$wrk(x$ind + i), f$wrk(x$ind + i$wrk(j$l
     *oc$ind + i$)))
            f$wrk(f$ind + i) = f$wrk(f$ind + i) + force
            f$wrk(f$ind + i$wrk(j$loc$ind + i$)) = f$wrk(f$ind + i$wrk
     *(j$loc$ind + i$)) - force
          enddo
        enddo
C       --<< scatter_add Send/Recv {[f(1:atomD$cnt)], f(j$loc(1:j$cn
C       t))} >>--
        call fscatter_add(f$wrk(f$ind + atomD$cnt + 1), f$wrk(f$ind +
     *1), x$sched)

C       --<< scatter_add initialization for {} >>--
C       Push atoms
C       --<< gather Send/Recv {[f(1:atomD$cnt)]} >>--
        do i = 1, atomD$cnt
          f$wrk(x$ind + i) = f$wrk(x$ind + i) + delta_func(f$wrk(f$ind
     * + i))
        enddo
C       --<< scatter_add Send/Recv {[x(1:atomD$cnt)]} >>--
      enddo
      call free(i$type, atomD$loc2glob$ind + 1, atomD$loc2glob$size)
      call free(i$type, j$glob$ind + 1, j$glob$size)
      call free(i$type, j$loc$ind + 1, j$loc$size)
      call free(i$type, inb$ind + 1, inb$size)
      call free(f$type, x$ind + 1, x$size)
      call free(f$type, f$ind + 1, f$size)
      call free(i$type, partners$ind + 1, partners$size)
     end
```

Figure 5.9 The `nbf` program after compilation by the FORTRAN D compiler, Part 3 of 3.

- Fortran D comments, starting with a Fortran D keyword such as "`DISTRIBUTE`."
- Fortran 77 comments, starting with "`--<<F77`." Those are always uncommented before native compilation.
- Target-specific Fortran 77 comments, starting with "`--<<`*ArchPrefix*" (*e.g.*, "`--<<Sun:`," "`--<<iPSC:`"). Those will be uncommented when compiling for a specific architecture.

  These target-specific comments are used mainly to annotate a program machine-dependently with I/O and timing routines without having to use separate sources for different architectures.

- The declarations of arrays that are recognized as being dynamic are shrunk to minimal size; for example, `REAL x(Natom)` became `REAL x(1)`. We found this a useful help for analyzing the compiler, but instead the declarations could also be deleted altogether.

- The Give-N-Take framework decides on communication placement. The results of this analysis is shown in "`--<<` ..." communication comments. However, the decision on what to communicate or whether to communicate at all is left to the code generator. The current prototype uses the Give-N-Take analysis only for communicating for irregular references and rely on the "regular compiler" for placing the remaining communications. References considered regular are enclosed in square brackets.

- The current implementation handles programs with procedure calls conservatively and does not perform any interprocedural optimizations, such as interprocedural inspector or communication placement. For example, it assumes that all actual arguments may be used and defined. However, the underlying symbolic analysis does already provide some of the information that would be necessary for interprocedural placement [Hav94].

5.4 An Object-Oriented Design

An important aspect of the implementation is its object-oriented design, reflected in the encapsulation of most concepts and algorithms into separate classes. This section describes the relationship between the basic strategies outlined in this chapter and the actual C++ classes of the compiler.

5.4.1 Overview

In the FORTRAN D compiler, most components of both the original and the generated codes correspond to specific types of classes. In particular, array references, subscripts, communication statements, schedules, inspectors, executors, and various aggregate concepts (such as the sets described in Figure 5.4) correspond to their own classes.

Consider the main loop of the mesh kernel (Section 2.1.2), which is shown in Figure 5.10. There are several references to arrays `w`, `flux`, and `cc` which require communication. The code generated by the FORTRAN D compiler is shown in Figure 5.11. We see that even though there are a total of 14 references for which we communicate, there are only three communication statements (the `gather`s of `w` and `cc`, and the `dscatter_add` of `flux`), one schedule (`w$sched`), and one inspector. This is reflected by the objects that get created during the compilation process. Figures 5.12 and 5.13 show which instances of which classes are built for a small loop inspired by the mesh kernel. They also give an (incomplete) overview of their interdependences, leading from individual subscripts and array references to communication statements, inspectors, and executors.

5.4.2 The classes

This section gives an overview of the individual.

A single subscript: `Dim`

One instance of `Dim` will be created for each individual subscript node. The methods of this class deal mostly with low-level details, such as unparsing subscripts, but they also provide a test for whether a subscript is irregular, and they assist in code generation for the inspectors by giving access to SSA information and incoming def-use edges.

```
          do t = 1, niter
             do i = 1, nnode
                flux(i) = 0.d0
             enddo

c            execute (i) on_home ends1(i)
             do i = 1, nedge
                n1 = ends1(i)
                n2 = ends2(i)
                q1 = ec*(w(2,n1) + w(3,n1) + w(4,n1)) / w(1,n1)
                q2 = ec*(w(2,n2) + w(3,n2) + w(4,n2)) / w(1,n2)
                q = (q1+q2) / 2
                flux(n1) = flux(n1) + (q + cc(n1)*ec)
                flux(n2) = flux(n2) + (q + cc(n2)*ec)
             enddo
          enddo
```

Figure 5.10 Main loop nest of the mesh kernel.

A subscript value number: `SubVal`

An instance of `SubVal` will be created for each value number of an individual subscript (not for the whole subscript; see Section 5.2.1). All instances together form the *SUB_VALS* set from Figure 5.4. This class is mostly used for mapping back from value numbers to actual code at code-generation time.

Sorting keys for the data-flow universe: `Key`

This class constitutes the members of *KEYS* from Section 5.2.3. It is used mainly for comparing array references and sorting them according to their communication needs. One instance of `Key` will be created for each bit in the data-flow vectors.

An array reference: `Ref`

One instance of `Ref` will be created for each individual array reference node in the program. It mostly propagates information from the individual subscripts up, such as whether it is irregular (which depends on whether any of the subscripts are irregular). It also gives a handle on how the array of the reference is distributed and which data-flow bit this reference corresponds to.

```
C       --<< Inspector for [7*]n2n1$loc(1,1:n2n1$cnt), [7*]n2n1$loc(
C       2,1:n2n1$cnt) >>--

        ...

        do i = 1, edgeD$cnt
          n2 = ends2(i)
          n2n1$glob(1, i) = n2
          n1 = ends1(i)
          n2n1$glob(2, i) = n1
        enddo

        ...

C       --<< gather Send/Recv {w(2, n2n1$loc(2,1:n2n1$cnt)), w(3, n2
C       n1$loc(2,1:n2n1$cnt)), w(4, n2n1$loc(2,1:n2n1$cnt)), w(1, n2
C       n1$loc(2,1:n2n1$cnt)), w(2, n2n1$loc(1,1:n2n1$cnt)), w(3, n2
C       n1$loc(1,1:n2n1$cnt)), w(4, n2n1$loc(1,1:n2n1$cnt)), w(1, n2
C       n1$loc(1,1:n2n1$cnt))} >>--
        call ngather(w(1, nodeD$cnt + 1), w(1, 1), w$sched, 32)
C       --<< gather Send/Recv {cc(n2n1$loc(1,1:n2n1$cnt)), cc(n2n1$l
C       oc(2,1:n2n1$cnt))} >>--
        call dgather(cc(nodeD$cnt + 1), cc(1), w$sched)

        ...

        do t = 1, niter
          do i = 1, nodeD$cnt
            flux(i) = 0.d0
          enddo
C         --<< scatter Send/Recv {[flux(1:nodeD$cnt)]} >>--

C         --<< Executor for w(1, n1), w(4, n1), w(3, n1), w(2, n1), w(
C         1, n2), w(4, n2), w(3, n2), w(2, n2), cc(n1), [2*]flux(n1),
C         cc(n2), [2*]flux(n2) >>--
C         execute (i) on_home ends1(i)
          do i = 1, edgeD$cnt
            n1 = ends1(i)
            n2 = ends2(i)
            q1 = (ec * w(2, n2n1$loc(2, i)) + ec * w(3, n2n1$loc(2, i))
     *+ ec * w(4, n2n1$loc(2, i))) / w(1, n2n1$loc(2, i))
            q2 = (ec * w(2, n2n1$loc(1, i)) + ec * w(3, n2n1$loc(1, i))
     *+ ec * w(4, n2n1$loc(1, i))) / w(1, n2n1$loc(1, i))
            q = (q1 + q2) / 2
            flux(n2n1$loc(2, i)) = flux(n2n1$loc(2, i)) + (q + cc(n2n1$l
     *oc(2, i)) * ec)
            flux(n2n1$loc(1, i)) = flux(n2n1$loc(1, i)) + (q + cc(n2n1$l
     *oc(1, i)) * ec)
          enddo
        enddo

        call dscatter_add(flux(nodeD$cnt + 1), flux(1), w$sched)
```

Figure 5.11 Main loop nest of mesh kernel, output of FORTRAN D compiler with communication and inspection body.

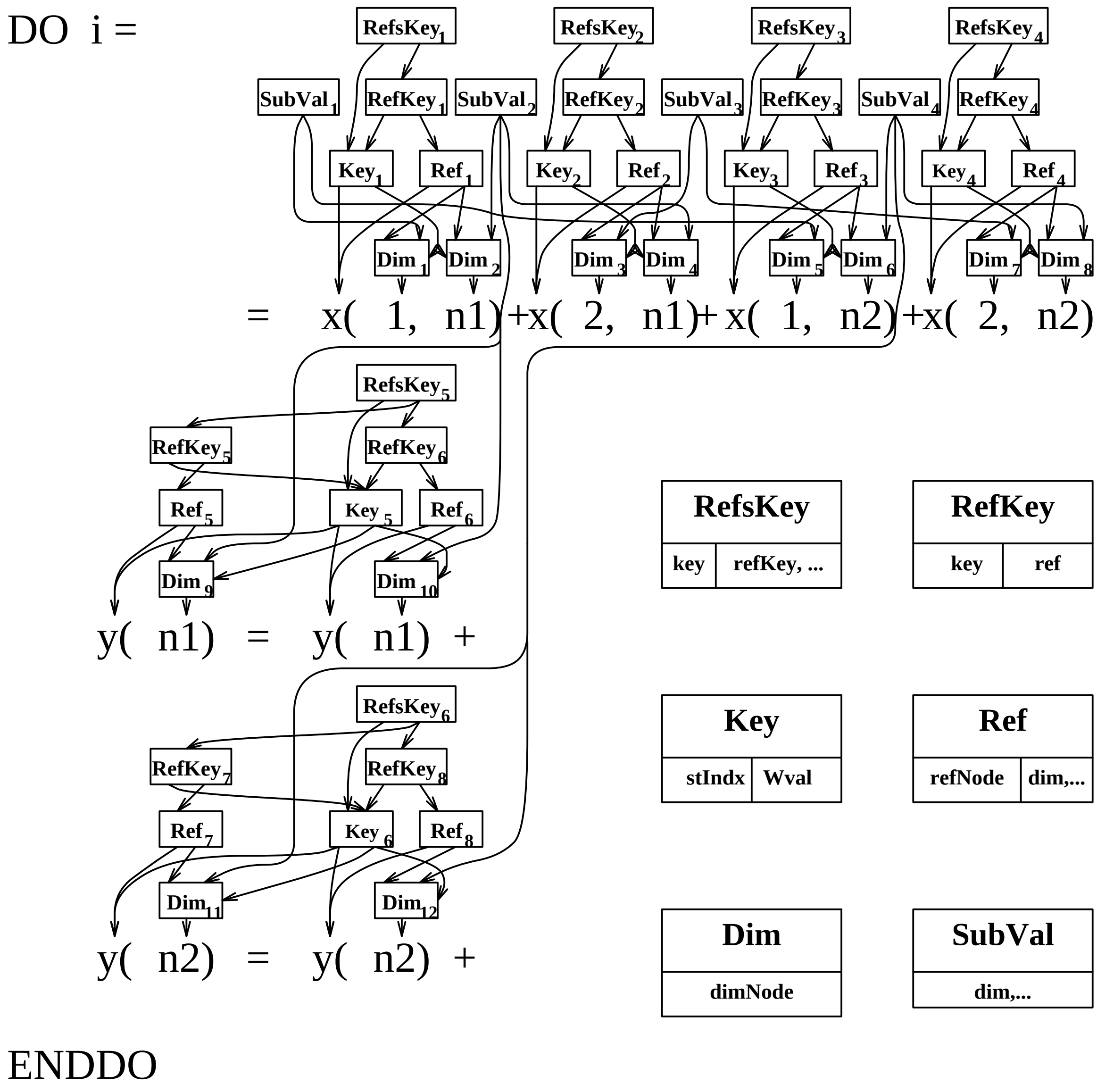


Figure 5.12 C++ classes for constructing the communication data-flow universe.

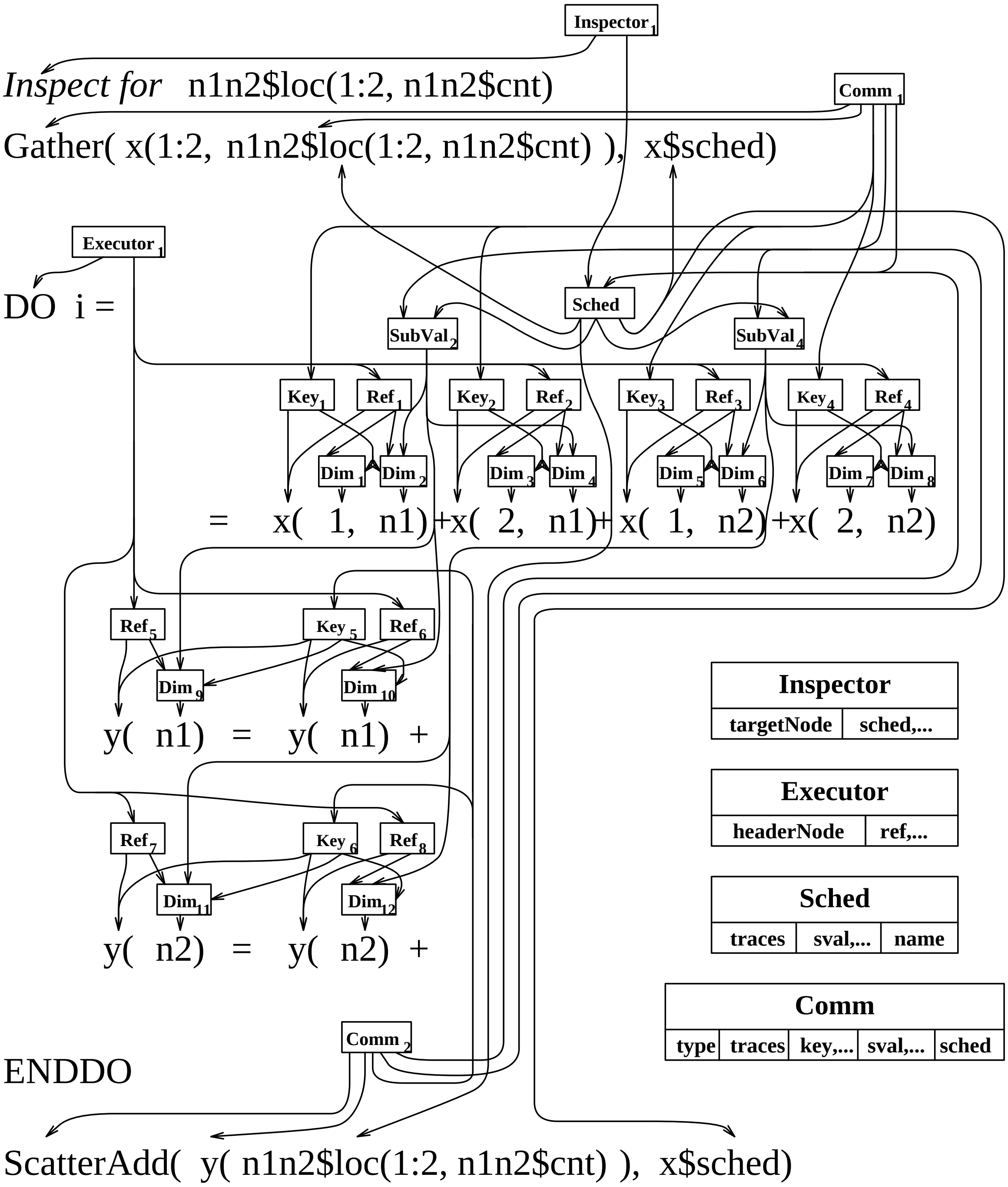


Figure 5.13 C++ classes for placing communication statements, inspectors, and executors.

A reference and its key: `RefKey`

When constructing the data-flow universe, one instance of `RefKey` will be created for each reference. This class provides some utilities for interfacing between the data flow universe and actual program text, such as dumping the universe.

All references for a key: `RefsKey`

This corresponds to the elements of *REFS_KEY* from Figure 5.4. For example, the references to `y(n1)` in Figure 5.12 have separate instances of `Ref` and `RefKey`, but they are combined into one `RefsKey`.

This also contains pointers to the value numbers of the individual subscripts for this key (not shown in Figure 5.12).

All keys: `RefsKeys`

This set, corresponding to *REFS_KEYS*, is the collection of RefsKey instances. It provides most of the interface between the universe and the rest of the compiler, such as mapping functions between value numbers, symbol table indices, keys, SSA nodes etc., many of which are non-unique. It also has methods for computing affected, contained, and touched closures of sets of keys (see Section 5.2.4).

A communication statement: `Comm`

An instance of `Comm` will be created for each statement that is sending or receiving messages or both. We are currently assuming that we can aggregate messages if and only if they are communicating the same array; this could easily be relaxed. This class serves mostly for determining which schedules are needed and for creating the actual statement at code-generation time (see also Section 5.2.5).

In Figure 5.12, there exist six different keys for the communication universe, four for `x` and two for `y`. However, all keys for `x` will be communicated at the same location, at the beginning of the loop. They also correspond to the same data array, and are therefore combined into one communication statement with one instance on `Comm`. Similarly, one `Comm` instance will be generated for reducing both `y(n1)` and `y(n2)`.

A communication schedule: `Sched`

We generate one instance of `Sched` for each set of value numbers of individual subscripts that are combined into one message somewhere.

In Figure 5.12, there are two communication statements. However, both statements need a schedule for the same set of subscript value numbers. Again, note that for schedule generation we are concerned with individual subscript value numbers, of which here two are interesting (`n1` and `n2`). For the communication generation, however, we need whole subscript value numbers, of which the first communication statement contains four and the second communication contains two.

The `Inspector` class

As outlined in Section 5.2.5, we generate an inspector whenever we have to generate a schedule. This class is mostly concerned with linking schedules that have to be generated at the same location together and with code-generation issues.

The `Executor` class

As described in Section 5.2.6, we create an executor for each loop which has a direct or auxiliary induction variable that is used for indexing trace arrays. Again, this class is used mostly for bookkeeping and code generation.

Relax – Don't worry – Have a home brew.

— Charlie Papazian (The Joy of Homebrewing)

Chapter 6

Experimental Results

This chapter describes the results of the practical experiments carried out for validating the overall thesis and concepts introduced earlier. The experiments fall into two categories. The first category, described in Section 6.1, uses the FORTRAN D prototype to compile and run FORTRAN D applications on MIMD machines and measure their performance. This allows us to compare the effectiveness of value-based mappings *vs.* index-based mappings. It also exercises the GIVE-N-TAKE framework for communication placement, and gives further insights into the overall feasibility of irregular data-parallelism. The second category of experiments, described in Section 6.2, is SIMD specific and does not use the FORTRAN D compiler; instead, we assess the value of the loop-flattening transformation by applying the transformation manually and comparing the results with the original version.

6.1 Value- vs. Index-Based Mappings

In evaluating the language extensions proposed in Section 2.3.1, we are mostly interested in the following questions:

1. How much does the extension improve the generated code?
2. How close is the generated code to a hand-coded implementation of the same approach?
3. How much convenience does the new extension provide over hand-coding?

To answer the first question, the output of the FORTRAN D compiler was compiled with `if77` onto an iPSC/860 with 32 nodes and 8 Megabytes of memory per node, and various performance aspects were measured as described in the following sections.

Regarding the second question, the FORTRAN D compiler-generated output was practically identical to a hand-coded parallel program using the same CHAOS runtime support. While this is certainly at least partly due to the simplicity and small size of most of the compiled kernels, it still gives reason to believe that there are no fundamental disadvantages when using the compiler to implement value-based mappings.

To answer the third question, one has to compare the complexities of original and compiled codes, with and without value-based distributions. While size alone is not a sufficient measure of complexity and programming difficulty, it still is an indication of the savings provided by the compiler. For example, one might compare the FORTRAN D version of the `nbf` program in Figure 2.3 with the compiler-generated code in Figures 5.7, 5.8, and 5.9. It turned out that for our test cases, the generated code, which in size was very close to hand-coded, was typically about twice as large as the initial code for index-based distributions. For value-based distributions, the code grew by another 25%.

The following sections describe the different benchmarks and experiments in detail.

6.1.1 The molecular dynamics kernel

Our experiments with this kernel were based on the FORTRAN D program from Figure 2.3 (but with 3-D instead of just 1-D coordinates and forces) with varying index- and value-based distribution directives as listed below. The parallel runs used pairlist data and physical coordinates from an SOD simulation (see Section 2.1.1) using an 8Å cutoff radius; SOD itself is of size $53 \times 55 \times 52\text{Å}^3$. We ran the simulation for 30 time steps, on 1, 2, 4, 8, 16, and 32 processors, with the following distribution strategies.

1. Index-based, `BLOCK`-wise.

   For N atoms and P processors, processor p gets assigned atoms $(p-1)N/P + 1, \ldots, pN/P$ (assuming P divides N). This corresponds to using the original `BLOCK` distribution (line 12 in Figure 2.3) throughout the run (*i.e.*, no redistribution in line 18), and is illustrated in Figure 2.2.

2. Value-based, 1-dimensional bisection, no load balancing.

   The physical problem domain gets divided along the x-axis, assigning each processor an equal number of atoms. This corresponds to redistributing data with a "`DISTRIBUTE atomD(VALUE(DIM=1, VALS=x))`" directive (no "`WEIGHT=inb`" parameter).

3. Value-based, 1-dimensional bisection, with load balancing.

   Each atom's work load is measured by the number of interaction partners, and the partitioner divides the physical problem domain such that each processor has an even workload. The appropriate directive is already shown in Figure 2.3, the resulting mapping can be seen in Figure 2.4.

4. Value-based, 3-dimensional bisection, no load balancing.

   The physical problem domain gets recursively divided along the x, y, and z-axes, assigning each processor an equal number of atoms. The directive is "`DISTRIBUTE atomD(VALUE(DIM=3, VALS=x,y,z))`."

5. Value-based, 3-dimensional recursive bisection, with load balancing.

   This is the version shown in Figure 2.5. The directive here is "`DISTRIBUTE atomD(VALUE(DIM=3, VALS=x,y,z, WEIGHT=inb))`."

We were able to run all strategies on all processors sizes, except for 1 and 2 processors, where the irregular distributions were too memory intensive.

Locality

The number of not-owned atoms accessed by the individual processors is a measure for the inter-processor access locality of the `nbf` kernel. We can estimate a theoretical lower bound on this number based on the total number of atoms (6968), the space they occupy (a sphere with a little over 50Å in diameter), and R_{cut} (8Å), which yields that at least about 950 atoms will have to be buffered.

Figure 6.1 compares the maximum number of buffered atoms for different machine sizes and distribution strategies. The dotted line with circles corresponds to the `BLOCK`-wise index-based distribution. For other, value-based distributions, the line style indicates dimensionality (dashed: 1-D, solid: 3-D), and the point style indicates balance (crosses: unbalanced, stars: balanced). We see that the 3-dimensional, value-based distribution performs best, with a slight additional advantage for the load balanced version, which comes very close to the predicted lower bound.

Balance

Figure 6.2 measures balance by comparing maximum and average workloads, expressed as floating point operation counts. The index-based distribution gets less

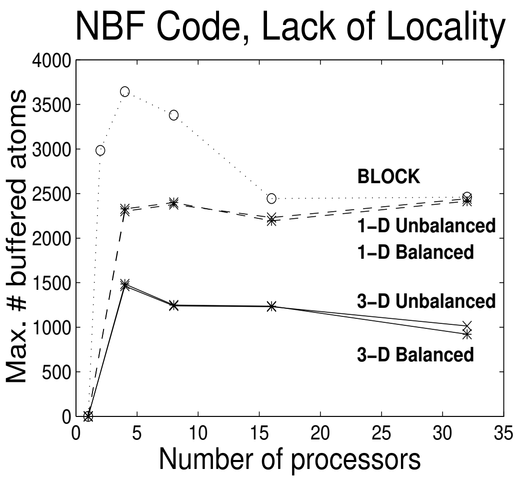


Figure 6.1 The number of communicated data (maximum across processors) for varying machine sizes and distribution strategies in the non-bonded force kernel.

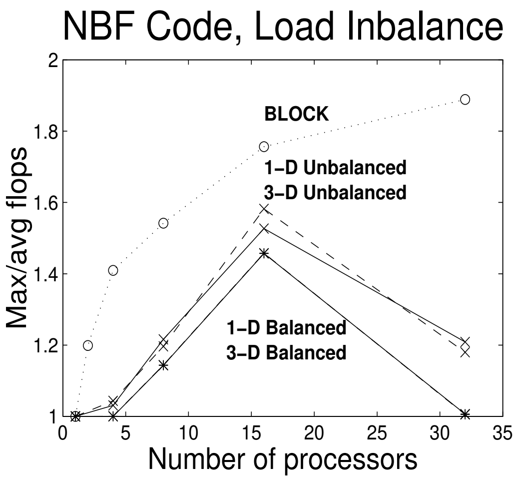


Figure 6.2 The fraction of maximum floating point operations (*i.e.*, workload of the slowest processor) and of the average across processors.

balanced as the number of processors goes up. Value-based distributions perform consistently better, and taking the work load into account provides a further improvement. For this application, the index-based `BLOCK` distribution is particularly unsuitable due to the diagonal nature of the adjacency matrix, which is actually symmetric according to Newton's Third Law; therefore, only one half of it is stored, which shifts the work load towards atoms with smaller indices.

Individual timings

Figure 6.3 shows the timings of the different code phases, using the 3-dimensional, balanced recursive bisection. In the processor ranges measured, the overheads associated with preprocessing, redistributing data, and communication are about an order of magnitude lower than the cost of the actual computation. The cost of preprocessing, including the inspector which enables message vectorization, gets well amortized for the given number of time steps (30). However, it is also apparent that the scalability of the run-time support may become critical for larger numbers of processors.

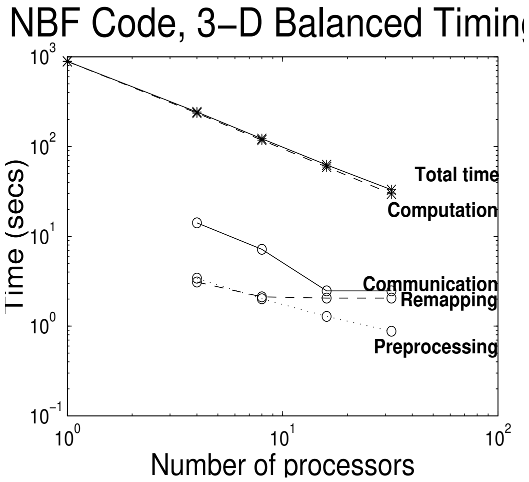


Figure 6.3 The timing breakdown (excluding I/O) for varying machine sizes, using a 3-dimensional, value-based balanced bisection.

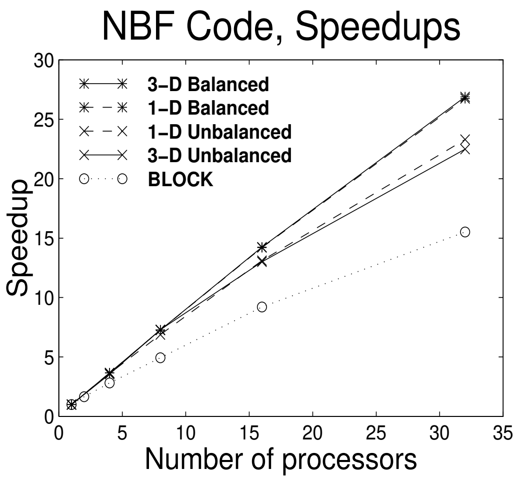


Figure 6.4 The speedup obtained with different processor numbers and distribution strategies.

Speedups

Figure 6.4 summarizes the speedups obtained for the various distribution methods, computed as the fraction of single-processor timing and parallel-processing time. The value-based distributions outperform the index-based distribution, and explicit load balancing provides an additional advantage.

6.1.2 The unstructured mesh kernel

We generated code for the Fortran D kernel from Figure 5.10, but again with 3-D instead of just 1-D coordinates and fluxes. Here we measured only the effect of varying alignments of `edgeD`; the distribution of `nodeD` was a fixed, 3-D value-based distribution. The parallel runs were done for a mesh with 9428 nodes and 59863 edges. We ran the simulation for 30 time steps, on 1, 2, 4, 8, and 16 processors, with the following alignment strategies.

1. Index-based, `BLOCK`-wise: for N edges and P processors, processor p gets assigned edges $(p-1)N/P+1, \dots, pN/P$ (assuming P divides N). This corre-

sponds to using the original `BLOCK` distribution (line 12) throughout the run (*i.e.*, no redistribution in line 21).

2. Value-based: decomposition `nodeD` and edge info `ends1`, `ends2` serve to align edge data with the node data. This corresponds to the value-based alignment directive shown in Figure 5.10.

Locality

Similar to the number of buffered atoms in the `nbf` kernel, here the number of not-owned nodes accessed by the individual processors is a measure for the inter-processor access locality of the mesh kernel. Figure 6.5 compares the maximum numbers of buffered atoms for different machine sizes and alignment strategies.

Considering that there are less than 10,000 atoms total, we see that the index-based mapping requires buffering a significant fraction of the whole data domain. Together with the owned edges we are effectively replicating the edge data for $P \leq 8$. The value-based alignment has a significantly better locality, typically requiring less than 500 nodes to be buffered.

Speedups

Figure 6.6 summarizes the speedups obtained for the two mapping methods. While the index-based mapping does not result in any speedup due to the bad locality, the value-based distribution provides about 40% efficiency for 16 processors.

6.1.3 A sparse matrix computation

The NAS CGM benchmark solves an unstructured sparse linear system by the conjugate gradient method [BBLS91]. The bulk of the computational time is spent in `matvec`, which multiplies a sparse matrix `a` with a vector `x`, as shown in Figure 6.7.

An interesting facet of this benchmark is the use of double indirection, since the inner loop not only accesses the `y` array through the `rowidx` indirection array, but also the loop bounds depend on the `colstr` array.

The Fortran D version parallelized this code by distributing `x` and `colstr` `BLOCK`-wise and using an `on_home` directive to parallelize the loop, since the owner-computes rule is difficult to apply to the innermost loop. Furthermore, since the current compiler prototype does not support interprocedural placement of inspectors

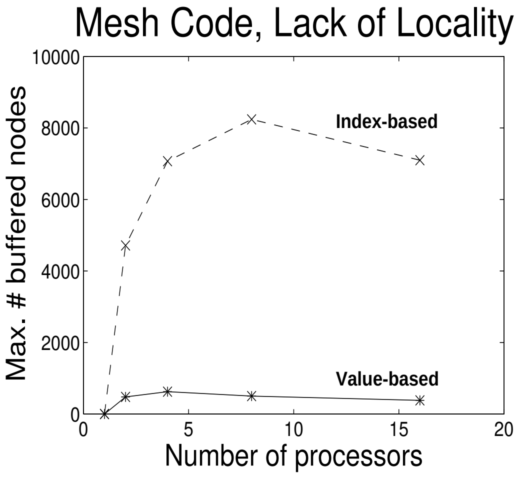


Figure 6.5 The number of communicated data (maximum across processors), for varying machine sizes and alignment strategies of the mesh kernel.

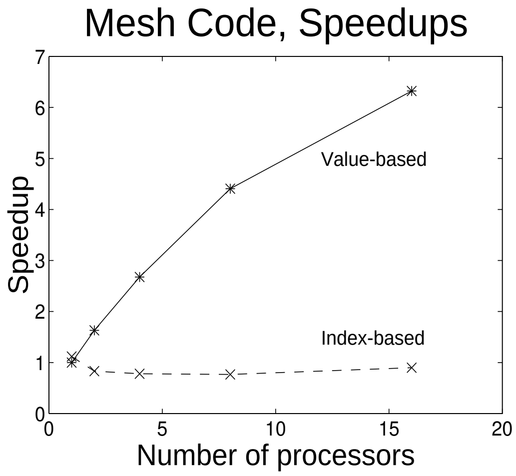


Figure 6.6 The speedup obtained with different processor numbers and mapping strategies.

```
      subroutine matvec(n, a, rowidx, colstr, x, y)
c     y = a * x.
c     a is a sparse matrix rep in a, rowidx, colstr form
      integer rowidx(1), colstr(1)
      integer n, i, j, k
      real*8  a(1), x(n), y(n), xj

      do 10 i = 1, n
         y(i) = 0.0d0
 10   continue
      do 200 j = 1, n
         xj = x(j)
         do 100 k = colstr(j) , colstr(j+1)-1
            y(rowidx(k)) = y(rowidx(k)) + a(k) * xj
 100     continue
 200  continue
      return
      end
```

Figure 6.7 NAS CGM benchmark, `subroutine matvec()`.

and communication, calls to `matvec` and to its caller, `cgsol`, where inlined, which resulted in a modest increase in code size; the original, sequential code had 13 subroutines and 855 lines, the FORTRAN D version had 11 subroutines and 892 lines. The performance of the compiled code is shown in Figure 6.8.

6.1.4 Full Gromos

As attested in Section 6.1.1, the main component of molecular dynamics performed by programs such as GROMOS, the non-bonded force computation, is inherently data parallel and amenable to efficient parallelizations using languages such as FORTRAN D. However, having a clean, data-parallel underlying algorithm alone is not sufficient; the compiler must also have a way to easily access this data parallelism.

Unfortunately, most of GROMOS is written in a way that makes this access hard, if not impossible; it is clearly not written in a "data-parallel programming style." For example, the compiler typically was unable to apply loop bounds reduction based on the owner-computes rule, since most loops did not iterate over data structures directly (*e.g.*, atom data were typically not accessed by looping over atoms directly, but instead by looping over whole charge groups). We parallelized this program, which consists of 7 subroutines and 4547 lines, in a very simple manner, by applying the `on_home` directive in association with a distributed dummy array. The corresponding speedups, shown in Figure 6.9, seem encouraging; however, one must realize that this is only for the main subroutine, with full data replication and correspondingly poor scalability.

The data-parallel community does not seem to promise users that they can easily transfer their old codes to the new paradigm, and the conversion of such "dusty decks" to parallel programs appears to be outside of the scope of most research interests. However, there is also the "dusty programmer" problem, which makes it worthwhile to analyze the difficulties of programs such as GROMOS in some more detail.

There were a number of difficulties that are probably neither limited to GROMOS nor do they seem likely to be addressed by improved compiler technology in the near future. The following lists a few of them, a more detailed discussion can be found elsewhere [CHK94].

- Multi-dimensional arrays that we would like to distribute only along a certain dimension are linearized.

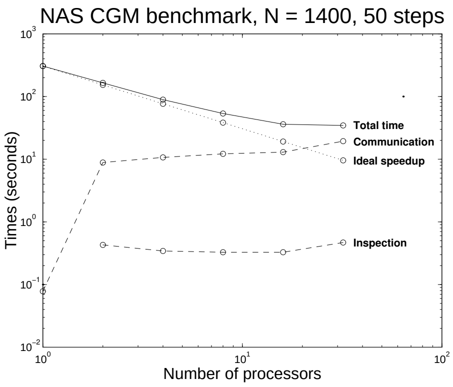


Figure 6.8 Performance of Fortran D for the NAS CGM benchmark.

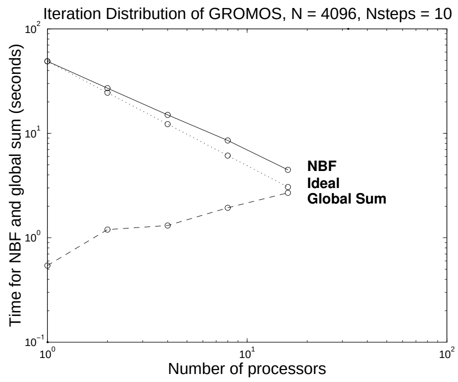


Figure 6.9 Performance of Gromos in Fortran D.

For example, 3-dimensional atom data, such as their coordinates stored in `x`, are not declared as a 2-dimensional array indexed by dimension and atom index, such as `x(3,Natom)`, but instead as 1-dimensional, linearized array, `x(3*Natom)`. It is possible to apply a `BLOCK_CYCLIC` distribution (in HPF: `CYCLIC(K)`) to distribute the linearized array, but analyzing accesses to these linearized arrays is challenging.

- Distributed arrays serve as work space for non-distributed data.

  Due to the lack of dynamic memory in Fortran 77, this seems to be a common practice disturbing advanced analysis.

- Distributed parameters are declared with different sizes in caller and callee.

- Unrelated computations are folded together.

  For example, the long-range force computation is performed together with the pair list recomputation. This complicates control flow and owner-computes analysis.

- No loops whose bounds could be reduced according to the owner-computes rule.

  As mentioned above, this typically evolved from the difference between the way data were organized (according to atom index) and the way computation was performed (on a charge group by charge group basis). This violates a major data-parallel principle, that computation should be distributed according to data distribution.

- **goto**s often substitute for the **do**-construct.

  This is another incarnation of the difficulty to apply loop bounds reduction.

- Jumps into loops result in irreducibilities.

  This is a classic problem for interval-based analysis techniques, which in our case can prevent hoisting communication out of such loops.

6.2 The Efficacy of Loop Flattening

On an SIMD architecture, the loop transformation described in Chapter 4 should be profitable whenever some processors sit idle in an inner loop and still have work to do in later iterations of the outer loop. This seems to be a situation potentially occurring in many scientific programs solving irregular problems [BSGM90, SPBR91, TP90, WLR90].

6.2.1 The application

One example of a loop nest with varying inner bounds is the NBF calculation as shown in Figure 2.1. However, this kernel takes advantage of Newton's Third Law and performs an indirect lhs assignment to $f(j)$, which results in irregular communication patterns. While it is possible to perform such communication operations on SIMD machines as well (see Section 7.1.2), they are generally more expensive than on MIMD machines. It appears that the performance gains from Newton's Third Law are likely to be small due to this computation/communication tradeoff. More importantly, however, is in this context that we are mostly interested in the relative effects of loop flattening, which does not change the communication requirements, and not so much in the communication costs itself.

For the experiments described in this section, we are using kernels derived from the F77 program shown in Figure 6.10. This code can be parallelized by partitioning the set of all atoms into P disjoint subsets and assigning one subset to each processor p. To achieve load balancing, the sum over the number of the partners of the atoms in a processor's subset should be roughly equal across the processors. Furthermore,

```
do At_1 = 1, N
   F(At_1) = 0
   do pr = 1, pCnt(At_1)
      At_2 = partners (At_1, pr)
      F(At_1) = F(At_1) + nbf_func(At_1, At_2)
   enddo
enddo
```

Figure 6.10 F77 version of the non-bonded force calculation `nbf`.

to achieve locality and scalability, the atoms within each subset should be closely together in space.

Figure 6.11 shows a $\mathrm{F90}_{SIMD}$ program which lays out the data in a cyclic fashion. If we assume for simplicity that P divides N, then each processor computes the non-bonded forces for N/P atoms. The uneven atom density results in varying values of $pCnt$; therefore, the inner loop with the (relatively expensive) force calculation often has to be executed with processors masked out even though they still have work to do in later iterations, just as it was the case in the *EXAMPLE* in Section 4.2. All processors have to go through

$$TIME_{SIMD} = \sum_{i=1}^{N/P} \max_{p=0,\ldots,P-1} pCnt(Atom_p^i) \tag{6.2$''$}$$

iterations, where $Atom_p^i$ is the i-th Atom of processor p.

This can be improved on by applying loop flattening, where we take into account that each atom has at least one interaction partner. The result is shown in Figure 6.12. Now each processor can loop through its atoms individually, so this code achieves the same time bound as a MIMD implementation:

$$TIME_{SIMD}^{flat} = \max_{p=0,\ldots,P-1} \sum_{i=1}^{N/P} pCnt(Atom_p^i), \tag{6.1$''$}$$

which is only limited by the quality of our workload distribution.

6.2.2 The hardware used

We implemented the non-bonded force kernel taken from the Gromos program suite on two SIMD machines and one work station. Our implementation models the behavior of the actual Gromos routine by reading in the arrays $pCnt$ and $partners$

$F = 0$
$At_1 = [1 : P]$
$lastAt = [N - P + 1 : N]$
while any $(At_1 \leq lastAt)$
  where $(At_1 \leq lastAt)$
    do $pr = 1, \max(pCnt(At_1))$
      where $(pr \leq pCnt(At_1))$
        $At_2 = partners(At_1, pr)$
        $F(At_1) = F(At_1) + \text{nbf\_func}(At_1, At_2)$
      endwhere
    enddo
    $At_1 = At_1 + P$
  endwhere
endwhile

Figure 6.11 F90$_{SIMD}$ version of `nbf`.

$F = 0$
$At_1 = [1 : P]$
$lastAt = [N - P + 1 : N]$
$pr = 1$
while any $(At_1 \leq lastAt)$
  where $(At_1 \leq lastAt)$
    $At_2 = partners\ (At_1, pr)$
    $F(At_1) = F(At_1) + \text{nbf\_func}(At_1, At_2)$
    where $(pr = pCnt(At_1))$
      $At_1 = At_1 + P$
      $pr = 1$
    elsewhere
      $pr = pr + 1$
    endwhere
  endwhere
endwhile

Figure 6.12 Flattened F90$_{SIMD}$ version of `nbf`. We take into account that $pCnt(i) \geq 1$ for all i.

as produced by GROMOS and then generating the calls to a force routine for each interaction pair. To exclude communication time from our measurements, we assume that the *pCnt* and *partners* arrays and the molecular configuration data (including the coordinates of atoms we are interacting with) are already locally available when calling the force routines.

The **DECmpp 12000 model 8B** (from Digital Equipment Corporation), which is identical to the MasPar MP-1200 series model, consists of 8192 processors (up to 16384 available), which are arranged in a mesh topology. It has 64 Kbytes main memory per processor, which gives 512 Mbytes total. Based on clock cycle counts, the individual processors are rated at 1.8 Mips. They are joined by an array control unit rated at 14 Mips. The MPFORTRAN version we had on site (1.0) did not allow the use of indirect array addressing in **forall** statements, so the timing results presented here are achieved using an α-version of the 2.0 compiler at MasPar which does not have this restriction.

The **CM-2** (from Thinking Machines Corporation) consists of 8192 one-bit processors (up to 65536 available), arranged in a hypercube topology. These are enhanced with 128 64-bit vector Floating Point Accelerators (FPAs) which use vector registers of length four. Each FPA is shared by two processor nodes of 32 processors each. The processors have 256 Kbits memory per processor, yielding a total of 268 Mbytes. The performance measured for a BYTE ADD is 500 Mips. We compiled our codes using the Slicewise 1.1 CMFORTRAN compiler which lays out the data "slicewise" across the one-bit processors and uses the FPAs directly.

We also implemented the kernel on Sun Microsystems' **Sparc 2**, which is rated at 28 Mips and whose 16 Mbytes memory allowed us to run the smaller test cases. We compiled our program with the Sun f77 compiler.

One additional interesting machine parameter is the *data granularity* which measures how small an array can be if we want to distribute it across *all* processors. This granularity, *Gran*, is particularly important on SIMD machines since whenever a certain array has to be manipulated by some processors, all processors have to step through the operation and they will be merely masked out if they do not actually own part of the array. Furthermore, this potential waste of processing time can not only occur for small arrays, but it is encountered whenever array sizes are not exact multiples of *Gran* [KLS90]. On the CM-2, using the slicewise compiler results in $Gran = P * 4/32 = P/8$ (32 processors per FPA, vector length 4); *i.e.*, we can economically use arrays whose total sizes are arbitrary multiples of $P/8$. This is a major advantage of the Slicewise model over the Paris model, which allocates data per one-bit processor. The corresponding data granularity on the DECmpp is simply $Gran = P$, and on the Sparc it is obviously $Gran = 1$.

Furthermore, the SIMD machines differ in the way they distribute data across the processors, which is significant if a dimension larger than *Gran* is distributed. The difference can be summarized as a cyclic ("cut-and-stack") data layout on the DECmpp and a blockwise layout on the CM-2.

6.2.3 Implementation experience

The DECmpp program and the CM-2 program used a single source, annotated with two sets of compiler directives, one for each machine. This worked relatively well; the only exception in our code was the `reshape` intrinsic. (The CMFortran convention for the argument order of this function is `mold` argument first, `source` argument second; MPFortran calls the `mold` argument `shape`, and has the order reversed. This combination of incompatibilities necessitated separate include files when using `reshape`; another option we tried was to replace the `reshape`s with explicit **forall** statements, which caused a slight performance degradation on both machines.) The Sparc implementation shared the code for performing I/O and gathering timing statistics.

On the DECmpp, a compiler switch is used to recompile for different machine sizes. No compiler switch is needed for CM-2 since it uses a *virtual processor* model which adjusts automatically to the actual machine size. However, we can still obtain significant performance improvements if compile time constants are used to adjust array dimensions to actual machine configurations.

The indirect addressing used in the flattened loop version frequently required resorting to **forall**s in the source code. For example, the statement

```
forall(i=1:P) at2(i) = partners(i,l(i),pr(i))
```

cannot be expressed with indirection vectors as

```
at2 = partner(:,l,pr)
```

since this expression would yield a three-dimensional array with `at2(i,j,k) = partners (i,l(j),pr(k))` instead of the desired one-dimensional array computed in the **forall** statement. However, implementing the flattened $F90_{SIMD}$ version from Figure 6.12 was still relatively straightforward. The derived code, L_f, ran well on both machines without further tuning; it is shown in Figure 6.13. `Lrs` is the number of *memory layers*

```
      subroutine AllFFlat()
C     Formal parameters omitted here;
C     F, pCnt, partners are distributed
C     in first dimension

      integer at1(P),at2(P),l(P),pr(P),m(P)
      real Force(P)
cmf$  layout Force,at1,at2,l,pr,m
cmpf  ondpu Force,at1,at2,l,pr,m

      F = 0
      l = 1
      pr = 1
      at1 = [1:P]
      do while(any(l.le.Lrs))
        forall(i=1:P)
     .       at2(i)= partners(i,l(i),pr(i))
        call OneFFlat(Force, at1, at2)
        forall(i=1:P, l(i).le.Lrs)
     .       F(i,l(i)) = F(i,l(i)) + Force(i)
        forall(i=1:P)
     .       m(i) = (pCnt(i,l(i)).ge.pr(i))
        where (m)
          pr = pr + 1
        elsewhere
          pr = 1
          l = l + 1
          at1 = at1 + P
        endwhere
      enddo
```

Figure 6.13 CMFortran/MPFortran version of flattened `nbf`.

(or *virtual processor slices*) which are in actual use; it is `Lrs` $= \lfloor 1 + (N-1)/Gran \rfloor$. The dimensions indexed with `1:Lrs` are of size `maxLrs` $= \lfloor 1 + (N_{max}-1)/P \rfloor$; for our implementation, the maximal number of atoms simulated is $N_{max} = 8192$. For example, for $Gran = 128$ and the $N = 6968$ atom test case described in Section 6.2.4, it is `Lrs` $= 55$ and `maxLrs` $= 64$; for $Gran = 8192$, we have `Lrs` $=$ `maxLrs` $= 1$.

Our experience with the implementation of the unflattened loop version was very different. The initial implementation of the pseudocode in Figure 6.11 was trivial to write, but its performance was roughly an order of magnitude worse than the flattened version on both machines and required significant performance debugging. We tried several different implementations using interface blocks, layout directives, inlining, different compiler switches, etc.; parameter arrays were automatic, fixed size, or passed in `COMMON` blocks; the dimension corresponding to different atom numbers was either left as a single dimension, as in `Force(1:Nmax)`, or split up into physical processor number and memory layer, as in `Force(1:P,1:maxLayers)`; the `:serial`-ized dimensions were rightmost or leftmost (the latter version recommended by the CMFORTRAN manuals); we tried **do**-**enddo** loops with precomputed loop bounds, such as `do pr = 1, maxPCnt`, and **do**-**while** loops, as in `do while(any(pr.le.pCnt))`; we also tried vectorizing the code in the dimension indexed by `pr`, but this was unfeasible due to the size of `partners`.

We here present timing results for two different unflattened versions; the first version, L_u^1, is shown in Figure 6.14. The other version, L_u^2, differs from L_u^1 in that all explicit "`1:Lrs`" indices are replaced with just a "`:`" referring to the whole dimension. Note that the dimension indexed with `1:Lrs` is laid out serially into local memory. Theoretically the machine front end could take advantage of the explicit subscripts of the L_u^1 version by pruning the number of processed memory layers. However, in practice it turns out that, at least on the CM-2, the processors will always cycle through *all* layers of memory. Doubling N_{max} (and therefore doubling `maxLrs`) and leaving all other parameters fixed results therefore not only in doubling execution time of the L_u^2 version on both machines, but on the CM-2, it also doubles running time of the L_u^1 version; on the DECmpp, the L_u^1 time increases by about 5%. The running time of L_f is independent of N_{max} on both machines, which is a nice side effect of loop flattening. Therefore, using the L_u^1 loops does not automatically result in savings by reducing the number of processed layers; however, we have to pay the additional overhead of checking on each layer whether it is active [Chr91]. This overhead is saved in the L_u^2 version.

```
      subroutine AllF()
C     Formal parameters omitted here;
C     F, pCnt, partners are distributed
C     in first dimension

      integer at1(P,maxLrs),at2(P,maxLrs)
      real Force(P,maxLrs)
cmf$  layout Force(:news,:serial)
cmf$  layout at1(:news,:serial)
cmf$  layout at2(:news,:serial)
cmpf  ondpu Force,at1,at2
cmpf  map Force(allbits,memory)
cmpf  map at1(allbits,memory)
cmpf  map at2(allbits,memory)
      integer pr

      F = 0
      at1 = reshape(shape = [P,maxLrs], source = [1:Nmax])
      maxPCnt = maxval(pCnt)
      do pr = 1, maxPCnt
        at2(:,1:Lrs) = partners(:,1:Lrs,pr)
        call OneF(Force,at1,at2)
        where (pCnt.ge.pr)
          F(:,1:Lrs) = F(:,1:Lrs) + Force(:,1:Lrs)
        endwhere
      enddo
```

Figure 6.14 CMFortran/MPFortran version of unflattened `nbf`.

6.2.4 The input data

We ran our test case for the bovine superoxide dismutase molecule (SOD), which has $N = 6968$ atoms. SOD is a catalytic enzyme composed of two identical subunits, each with 151 amino-acid residues and two metal atoms [SM91].

Figure 6.15 shows maximal and average numbers of interaction partners, $pCnt_{max}$ and $pCnt_{ave}$, which indicate the computational workloads for different cutoff radii. As expected, both values increase cubicly with the cutoff radius. As indicated in Equations 6.1″ and 6.2″, the difference between maximum and average number of partners gives us an upper bound on how much improvements we can expect from loop flattening.

6.2.5 The results

Table 6.1 gives performance results for the CM-2 and the DECmpp 12000, which are also displayed in Figure 6.16. For comparison, the running times on the Sparc were 3.86 seconds for the 4 Å case and 31.43 seconds for the 8 Å case. All runs were done several times, the differences in running times were usually less than 0.01%.

All loop versions were also timed with inlined calls to the force routine. On the CM-2, the effect was marginal; on the DECmpp, fluctuations were within 5%, roughly evenly distributed in both directions.

Table 6.2 gives the number of calls to Force routine for the flattened and the unflattened loop versions (the latter number scaled up by `Lrs` to account for the different argument sizes of `OneF()` and `OneFFlat()`) for different data granularities, along with their ratios. Note that the counts given in the last row are actually the maxima of $pCnt$ for the corresponding cutoff radii, as given in Figure 6.15. The given L_u/L_f ratios are bounded by the $pCnt_{max}/pCnt_{ave}$ ratios, which are 3.347, 2.689, 2.665, and 2.949 for cutoffs 4 Å, 8 Å, 12 Å, and 16 Å, respectively.

6.2.6 Interpretation

Considering the different computing powers per individual processor, the overall speedups of the parallel codes over the Sparc code version were satisfactory. However, we have to take into account that we excluded communication costs from our study. Due to the irregular nature of the problem, these communication costs might be relatively high; but as indicated earlier, the communication requirements are not changed by our transformation.

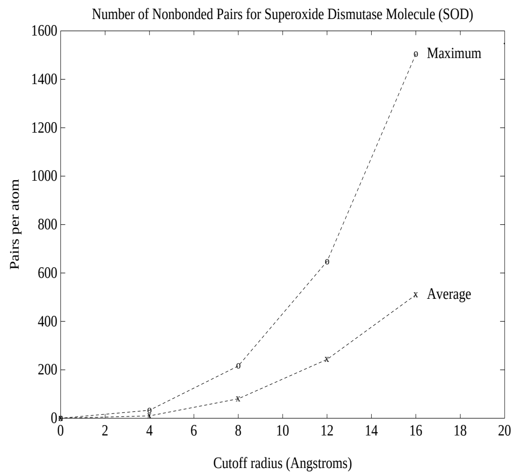


Figure 6.15 Maximum and average number of non-bonded force interaction partners per atom for the superoxide dismutase molecule, using different cutoff radii.

	4Å			8Å			12Å			16Å		
$P/Gran$	L_u^1	L_u^2	L_f	L_u^1	L_u^2	L_f	L_u^1	L_u^2	L_f	L_u^1	L_u^2	L_f
1024/128			3.89			27.03						
2048/256	6.57	3.86	2.13	42.91	25.13	14.72						
4096/512	3.22	1.83	1.11	21.02	11.95	7.65			24.78			
8192/1024	1.72	0.99	0.64	11.19	6.46	4.57			13.31			27.17
1024/1024	0.910	0.934	0.390	5.36	5.85	2.81	15.91	17.45	8.19	36.86	40.45	16.84
2048/2048	0.638	0.481	0.266	3.35	3.00	1.69	9.96	8.95	4.98	23.07	20.71	10.68
4096/4096	0.352	0.269	0.157	1.86	1.55	1.05	5.18	4.59	3.14	11.96	10.58	6.51
8192/8192	0.145	0.129	0.104	0.683	0.715	0.671	1.92	2.09	2.00	4.42	4.82	4.66

Table 6.1 Performance results for the CM-2 (upper half) and the DECmpp (lower half). Running times (in seconds) are listed for different cutoff radii and different loop versions. L_u^1 – unflattened loop selecting memory layers, L_u^2 – unflattened loop using all memory layers, L_f – flattened loop.

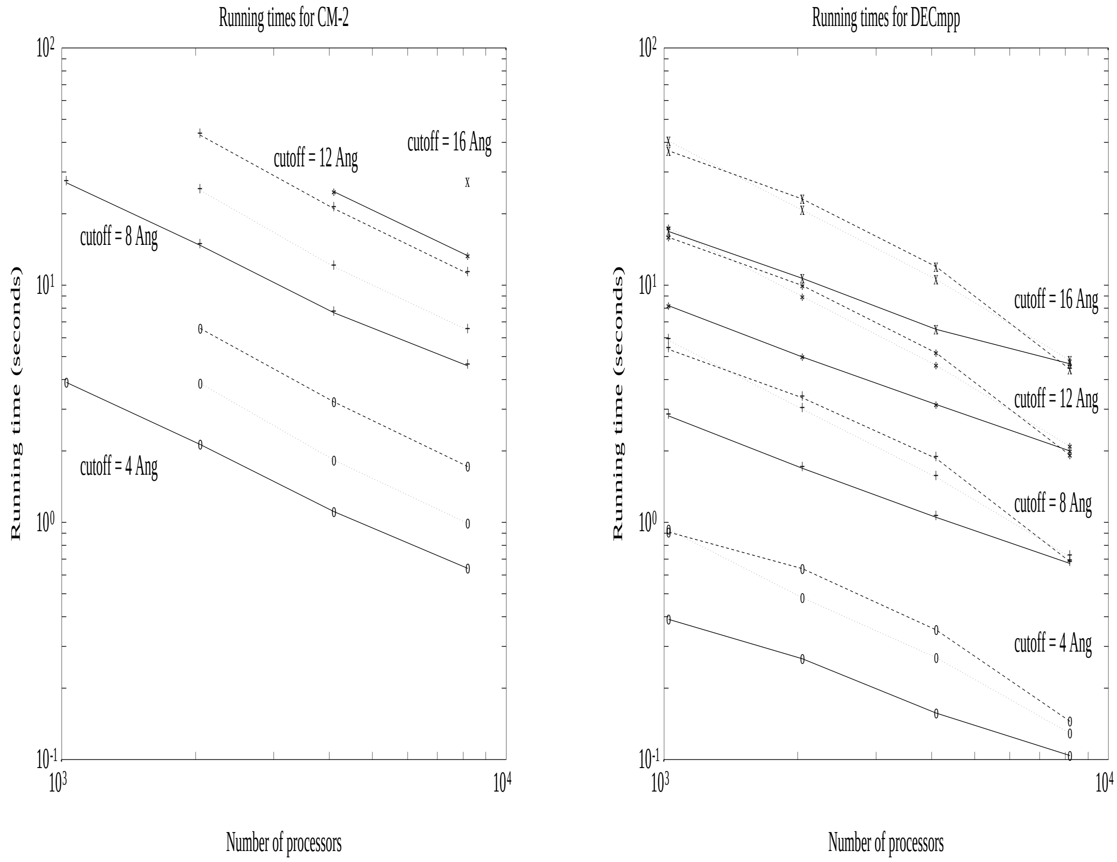


Figure 6.16 Performance results for the CM-2 and the DECmpp 12000. Different loop versions vary in line style; dashes: unflattened loop selecting memory layers; dots: unflattened loop using all memory layers; solid lines: flattened loop. Different cutoff radii are indicated by point styles; circles: 4 Å, plusses: 8 Å; stars: 12 Å; crosses: 16 Å. For judging speedups, note the log-log scale and the aspect ratio.

	4Å			8Å			12Å			16Å		
$Gran$	L_u	L_f	L_u/L_f	L_u	L_f	L_u/L_f	L_u	L_f	L_u/L_f	L_u	L_f	L_u/L_f
128		722			5076							
256	924	397	2.327	6048	2754	2.196						
512	462	224	2.063	3024	1559	1.940		4649				
1024	231	125	1.848	1512	906	1.669	4536	2642	1.717	10528	5436	1.937
2048	132	86	1.535	864	545	1.585	2592	1606	1.614	6016	3434	1.752
4096	66	51	1.210	432	357	1.210	1296	1069	1.212	3008	2222	1.354
8192	33	33	1	216	216	1	648	648	1	1504	1504	1

Table 6.2 Number of calls to Force routine, flattened/unflattened version. The data granularity, $Gran$, is equal to P for the DECmpp and $P/8$ for the CM-2. L_u counts are multiplied with `Lrs`.

When comparing Tables 6.1 and 6.2, loop flattening fulfills the expectations given by Equations 6.1″ and 6.2″. Despite the significant effort on speeding up the unflattened loop versions (as described in Section 6.2.3), the improvements of the flattened version often went beyond what we predicted from the $pCnt_{max}/pCnt_{ave}$ ratios, in particular on the DECmpp. We assume that this is largely due to the side effect mentioned in Section 6.2.3, namely that loop flattening makes actual running times less dependent on array sizes if we do not access all parts of the array; *i.e.*, we can increase N_{max} without automatically making L_f slower (unlike for L_u^2 and even L_u^1). This we consider a significant advantage in practice, since it allows compiling programs with provision for maximal problem sizes without paying a penalty on smaller sizes.

Moreover, it turned out that the effort of expressing array bounds in terms of actual machine sizes improved the unflattened loop versions as well. This was particularly beneficial for the virtual processor model of the CM-2.

The differences between the two unflattened loop versions L_u^1 and L_u^2 were larger on the CM-2 than on the DECmpp, as mentioned in Section 6.2.3. However, even on the DECmpp, L_u^2 performed better than L_u^1 when `Lrs` approached `maxLrs`.

It is important to keep in mind that the slicewise compiler for the CM-2 actually generates code with a data granularity of $Gran = P/8$, as discussed in Section 6.2.2. This coarser granularity results in more atoms per processor and therefore better applicability of loop flattening. As the table and the graph indicate for the CM-2, several cases could be run with the L_f version with reasonable performance while they could not be run at all in L_u^1 or L_u^2 because of stack overflows; large temporary arrays were needed in L_u^1 and L_u^2 even in loop versions which forward substituted intermediate results.

A Terraplane filled with Pinto beans
weighs 284 pounds and claims to have 6800 cu in ...
A claim made under obvious duress.
These volume figures provide usable volume,
not how much sand a pack will hold.

— Dana Design

Chapter 7

Background and Related Work

We can classify the steps towards the efficient parallel implementation of an irregular problem by the level at which they are taken: the algorithm development, the program text, supportive tools, the compiler, the operating system, or the underlying hardware. While this thesis has a strong focus on the compiler level, this section puts its contributions in perspective by examining related work also at higher and lower levels and in the domain of regular applications.

7.1 Tools

Tools can assist in load balancing and in communication, both of which can be particularly tedious and error prone when trying to parallelize an irregular problem efficiently [HS91].

The tools for parallelizing irregular problems can roughly be divided into two groups. The first group of tools provides an easier grip on the physical properties of the problem; *i.e.*, it takes advantage of *spatial* locality. The second group comes into play after the data structures have been laid out; these tools try to free the user from dealing with the access properties of the parallel program; *i.e.*, they examine *data* locality.

7.1.1 Tools based on spatial decomposition

The Generic Multiprocessor

The *Generic Multiprocessor* (GenMP) [Bad87, Bad91] aims at providing a machine-independent programming environment for a certain class of problems, namely scientific calculations that are spatially localized on a mesh. GenMP is a layer of software that can be thought of as a virtual machine that operates on a d-dimensional *work mesh* through a sequence of states. With the aid of application dependent routines written by the user, it repartitions the work mesh across processors to achieve a balanced work load and performs the necessary communication. Good results have been achieved with the implementation of the vorticity-stream function formulation of Euler's equation for incompressible flow in two dimensions in an infinite domain. The tests were performed on a 32-processor distributed-memory iPSC hypercube and a 4-processor shared-memory Cray X-MP vector machine. The limitation of this approach lies in its specificity towards applications that already use data structures reflecting locality characteristics; this we try to overcome by using general value-based decompositions as introduced in Chapter 2.

Lattice Parallelism

Lattice Parallelism (LPar) [Bad92, BK93, BKF94, FB95] is an SPMD programming model that supports coarse-grained parallelism based on the Fidil language [HC88] and the owner computes rule. It is intended for non-uniform computations that involve partial differential equations and have local structure. It explicitly excludes unstructured calculations such as sparse matrix linear algebra and finite element problems. Its main data type is the *Map*, whose elements are indexed by tuples just like array elements. However, the index set of a map, the *Domain*, is not necessarily rectangular, but can be arbitrarily sparse instead. Furthermore, a map declaration itself does not reserve any storage; this has to be allocated explicitly or by an initialization assignment. Maps are flexible; their domain can change at run time, and several domain constructors (for rectangular domains, also with arbitrary starting and ending points and strides) and operators (union, intersection) are available.

Parallelism is expressed by mapping a logical processing Domain onto a spatial processing Domain. LPar supports load balancing and *ghost regions*, in which each processor stores data within a certain proximity to its own data, similarly to overlap regions [Ger90]. It is currently implemented in C++ for the iPSC/860. LPar treats parallelism at a very high level, it manipulates the structure of the data, rather than the data itself. It enables very elegant formulations of a limited class of problems and can be seen as a potential user of an implementation of the value-based decompositions proposed in Chapter 2. Citing Baden [Bad92]:

> It is not an implementation-level system, and relies on application libraries or other run time systems to handle data partitioning or to handle machine-level optimizations, that could be provided for example by Dino or by Fortran D.

7.1.2 Tools based on access patterns

Chaos

The Chaos primitives, which succeed Parti (Parallel Automated Runtime Toolkit at ICASE), are a set of high level communication routines that provide convenient access to off-processor elements of arrays that are accessed (and distributed) irregularly [BS90, SBW90, DMS+92, DHU+93]. The authors of Parti were the first to propose and implement user-defined irregular distributions [MSS+88] and a hashed cache for nonlocal values [MSMB90]. They build on the inspector-executor paradigm described above; they

1. Coordinate interprocessor data movement,

2. Manage the storage of and access to copies of off-processor data, and

3. Support a shared name space, using a distributed translation table [SCMB90] to store the local address and processor number for each distributed array element.

High-level library routines, such as Chaos, can assist in tasks such as global-to-local name space mappings, communication schedule generation, and schedule based communication. Such libraries are essential both for keeping code complexity and programming difficulties in reasonable limits and also for the compiler writer; our Fortran 77D implementation also generates code that calls the Chaos library. However, since each application still has its own individual communication requirements, the proper usage of such routines and preprocessing for generating their arguments is still a non-trivial task one would rather leave to a compiler.

Communicating the right data at the right time and place is a difficult, yet crucial task for parallelizing irregular problems. The Chaos primitives are valuable tools for the first part of the problem, namely for determining where to find which data and for carrying out efficient data exchange. The data-flow framework presented as part of this thesis in Chapter 3 is designed for attacking the second part of the problem, namely enabling the compiler to make good use of these primitives without further advice by the user.

The Communication Compiler

The *Communication Compiler* [Dah90] is a software facility for scheduling general communications on the Connection Machine. It employs simulated annealing to find a data mapping with as low communication requirements as possible. It uses a recursive routing algorithm to determine an actual communication schedule. For fixed communication patterns, the cost of generating this schedule can be amortized by reuse, for example, over many time steps of a simulation. Its generality makes it highly applicable towards irregular communication structures. However, the communication patterns have to be known before using the Communication Compiler.

7.2 The Compiler

7.2.1 Parallel compilation systems

There have been and still are numerous research projects in the area of compiling for parallel architectures. Early work in the field of compiling for distributed-memory machines focussed on defining frameworks for nonlocal memory accesses [CK88] and data distributions [GB91, HA90, RS89]. For exploiting coarse-grained functional parallelism, high-level parallel languages such as LINDA [CG89], STRAND [FT90, FO90], and DELIRIUM [LS91] have been defined.

Several compilation systems for exploiting fine-grained parallelism have been and are being built, which include AL [Tse90], ASPAR [IFKF90], C*/DATAPARALLEL C [HQL+91, RS87], CRYSTAL [LC91], DINO [RSW91], ID NOUVEAU [RP89], MIMDIZER [SWW92], OXYGEN [RA90], P^{3}C [GAY91], PANDORE [APT90], PARAFRASE-2 [GB92], PARAGON [CCRS91], SPOT [SS90, Soc90], SUPERB [ZBG88], and VIENNA FORTRAN [BCZ92]. While there is still much work to be done in this field in general, there has already been considerable success in the field of regular applications, and "second generation parallelizing compiler" has become a common term.

7.2.2 FORTRAN D

FORTRAN D is an SPMD (Single-Program Multiple-Data) style language developed by the distributed-memory compiler group at Rice University and serves as a basis for this work. The following contains a very brief summary of its basic concepts, the complete language is described in detail elsewhere [FHK+90]. Citing Hiranandani *et al.* [HKT92b]:

> FORTRAN D is the first language to provide users with explicit control over data partitioning with both data *alignment* and *distribution* specifications. The `DECOMPOSITION` statement specifies an abstract problem or

index domain. The ALIGN statement specifies fine-grain parallelism, mapping each array element onto one or more elements of the decomposition. This provides the minimal requirement for reducing data movement for the program given an unlimited number of processors. The alignment of arrays to decompositions is determined by their subscript expressions in the statement; perfect alignment results if no subscripts are used.

The DISTRIBUTE statement specifies coarse-grain parallelism, grouping decomposition elements and mapping them and aligned array elements to the finite resources of the physical machine. Each dimension of the decomposition is distributed in a BLOCK, CYCLIC, or BLOCK_CYCLIC manner or replicated.

7.3 The Operating System

Virtual or hardware supported single-address space systems can ease the task of parallel programming by eliminating separate address spaces and explicit communications. Examples of these systems are Amber [CAL+89], Clouds [RAK88], Dash [LLG+90], Ivy [LH89], Midway [BZ91], Munin [CBZ91, KCZ92], Orca [BT88], and Platinum [CF89]. They preserve sequential semantics by enforcing a consistency protocol, which can be lazy or eager, based on invalidations or updates. Munin supports several such protocols, the choice between them for each individual shared variable is guided by access pattern annotations provided by the user. These systems, however, are demand-driven and therefore limited in how much they can hide memory latency (by prefetching data before they are needed) or reduce data movement costs (by fetching entire blocks at once). They are limited in that they can only react to accesses to nonlocal memory; at best, they can maintain a history of past accesses and try to guess future patterns.

One problem where operating systems can assist in the implementation of irregular applications in particular is the task of load balancing, since some spatial and temporal locality is usually associated with the workload. Here information about the utilization of different processors can be helpful. However, the work done in this area has focussed on thread-based parallelism [ELZ86, Luc88], typically even associated with distinct processes, instead of data parallelism. One also has to keep in mind that irregularities in the presence of race conditions might lead to system crashes [HtEBBW].

7.4 The Hardware

Some hardware facilities that can be particularly useful for irregular applications are the following.

7.4.1 Low latency

Due to the typically very irregular access patterns, message blocking becomes more complicated than for regular applications [SHG92]. A low latency communication system makes this difficulty less critical.

7.4.2 General routing facilities

Again, due to indirect addressing and the associated irregular access patterns it is often difficult to constrain the communication to nearest neighbor communication channels, so a fast general router is advantageous.

7.4.3 Decoupling of control flow on SIMD architectures

A problem similar to the potential load imbalance across processors is an uneven workload *within* processors. This is also a common characteristic of irregular problems, where the fraction which a processor spends of its total computation time on a certain part of its assigned workload may vary. This may cause additional idling when running irregular problems on SIMD machines instead of MIMD machines.

The restricted control flow of pure SIMD programming has been addressed by several researchers. General simulators of MIMD semantics on SIMD machines have been implemented by Kuszmaul [Kus86] and Hudak *et al.* [HM88] on the Connection Machine and by Biagioni [Bia91] and Dietz *et al.* [DC92] on the MasPar. These simulations are generally based on graph reduction interpreters for functional languages. Their performance tends to be scalable, but in absolute measures still below the speed of sequential work stations.

Philippsen *et al.* introduce two variants of a **forall** statement, a synchronous version and an asynchronous one [PT91]. The *asynchronous* **forall** enables multiple threads of control to coexist. This can either be emulated using stacks of MASK bits, or it can be implemented directly in an MSIMD machine which contains multiple program counters. In either case, their proposal is mainly concerned with enabling the concurrent execution of both branches in **if-then-else** constructs.

Loop flattening [HK92] is one technique to overcome this limitation for loop nests with varying loop bounds, as proposed in Chapter 4. Loop flattening can also be used to process multiple array *segments* of different lengths per processor, as introduced in Blelloch's *V-RAM model* [Ble90]. Thus it can be viewed as a generalization of substituting direct addressing with indirect addressing as Tomboulian and Pappas did for computing the Mandelbrot set [TP90].

7.4.4 Fast scan operations

The inhomogeneous workload across processors generally associated with irregular problems calls for load balancing. Scan operations are one efficient way for determining the total workload and its distribution [Bia91, Ble90]. On architectures providing an embedded reduction tree, this operation can be done in $\mathcal{O}(\log P)$ cycles.

Chapter 8

Summary & Open Issues

The projects described in Chapters 2, 3, and 4 focus on different aspects of the same problem, namely how to solve irregular applications efficiently with parallel machines. Furthermore, they all focus on the issue of how much support a compiler can give for these applications. The results so far seem to indicate that it is feasible to design high-level language support similar to the support existing for regular problems and to implement it at reasonable implementation and run-time costs.

An important example of this is the concept of the value-based decomposition based on the exploitation of spatial locality of the underlying physical problem domain, as introduced in Chapter 2. We extended the data-parallel paradigm by adding *value-based* enhancements to the already existing, so far index-based data distribution and alignment mechanisms. We illustrated their use with representative irregular kernels and compared the performance of these kernels with index-based and value-based distributions. We also compared a FORTRAN D version and a message-passing version, *i.e.*, a program using run-time support explicitly as generated by the compiler (or written by hand), of the same kernel. This gave an example for the added convenience provided by the proposed language extensions.

A principal difference between index-based and value-based mapping strategies is the departure from a model that derives data-to-processor mappings from static, sequential data-to-storage information (such as an array index). Instead, we consider run-time values to derive the mappings. This approach could also be used for data types other than arrays, for example for list structures in a version of data-parallel C, or for distributing spatial data structures such as oct-trees commonly used for N-body problems with very large N.

Another application of the value-based approach could be *value-based inspectors*. For example, the pair list used in the NBF computation is typically generated with a naive $\mathcal{O}(N^2)$ algorithm, where each atom is compared against every other atom, with correspondingly high computation, communication, and storage requirements. If we know that each processor has atoms only within a relatively small subdomain, we could restrict our attention to atoms that are either within or close to this subdomain, thereby reducing the amount of non-local data that have to be buffered.

Value-based mappings provide a specific kind of expressiveness that had been missing so far for data-parallel languages; they can give the compiler information that cannot be described in terms of array indices, and they provide a convenient mechanism to specify irregular mappings. While this is only one aspect of parallelizing irregular problems, we believe that it significantly widens the range of application that can be implemented efficiently in data-parallel languages.

After having decided on a certain decomposition, communicating the right data at the right time and place is still a difficult, yet crucial task for parallelizing irregular problems. The CHAOS primitives are valuable tools for the first part of the problem, namely for determining where to find which data and for carrying out efficient data exchange. The data-flow framework presented in Chapter 3 is designed to attack the second part of the problem, namely enabling the compiler to make good use of these primitives without further advice by the user. We expect GIVE-N-TAKE to have potential use in other areas as well, such as general memory hierarchy issues (cache prefetching, register allocation, parallel I/O) and classic partial redundancy elimination applications (common subexpression elimination, loop invariant code motion, etc.).

However, there is obviously still room for further improvement and generalization of the underlying theory. For example, the current framework does already support latency hiding by separating *Send*- and *Recv*-operations. However, there are some additional issues involved when generating simple *Send*s and *Recv*s from there (instead of using high-level communication routines such as CHAOS). In particular, the assignment of message id's and matching *Send* s and *Recv*s such that data which are sent together are also received together and vice versa appear to be both interesting and challenging problems.

A refinement that would be useful for block structured irregular problems, for example, is to allow data exchanges between just *subsets* of processors, instead of requiring a global coordination whenever the primitives are called as mentioned in the introduction.

One might also consider relaxing the static ownership concept for data accessed via indirection arrays. For example, it might occur that a processor p computes some data that are owned by processor q, but the next use of the data is on processor r. In the current owner-computes framework, we would first scatter the data from p to

q and then gather them from q to r, which could be replaced by a single "scatter-gather" from p to r. This, however, would add an additional degree of complexity to the theoretical framework and the underlying communication primitives.

Finally, one might consider pruning the framework down towards regular applications, where the same need for blocking and combining communication arises. So far, this is typically handled with local code transformations based on dependence analysis, but there is no inherent reason for not applying data-flow analysis here as well. This could be done by using the GIVE-N-TAKE framework over a lattice based on regular array sections.

Important questions not only for the GIVE-N-TAKE-based communication placement method presented here, but for compiling onto message passing architectures in general are how to determine whether a program is a valid instance of the communication generation problem (see Section 3.2.1), and how to transform an invalid program into a valid one. Answering these questions without being overly restrictive is a crucial step towards compiling correctly in the presence of arbitrary, non-SIMD control flow.

The loop flattening transformation described in Chapter 4 is an attempt to overcome certain limitations when using a SIMD computer for solving irregular problems without going as far as trying to achieve general MIMD semantics on SIMD machines. Loop flattening was designed to ease some particular SIMD restrictions without introducing any overhead; however, it supports a programming style that seems to be preferable on current SIMD machines even when running regular applications [HK92]. This rather surprising result suggests that flattened loops make it easier for compilers to derive the information they need for performing certain optimizations, such as pruning out virtual processor layers for individual statements whenever possible. This transformational approach might possibly be carried over into other situations where the restrictiveness of the SIMD model degrades overall performance.

So far, there does not seem to be a clear limit to the level of support that a compiler, when given the proper analysis, can give the scientist programming an irregular problem.

His "long range plan," he says,
is to "refine" these nerve-wrecking methods, somehow,
and eventually "create an entirely new form of journalism."

— The Editor (Dr. Hunter S. Thompson)

Bibliography

[AL93] S. P. Amarasinghe and M. S. Lam. Communication optimization and code generation for distributed memory machines. *ACM SIGPLAN Notices*, 28(6):126–138, June 1993. *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation.*

[All70] F. E. Allen. Control flow analysis. *ACM SIGPLAN Notices*, 5(7):1–19, 1970.

[APT90] F. André, J. Pazat, and H. Thomas. Pandore: A system to manage data distribution. In *Proceedings of the 1990 ACM International Conference on Supercomputing*, Amsterdam, The Netherlands, June 1990.

[Bad87] S. B. Baden. *Run-Time Partitioning of Scientific Continuum Calculations Running on Multiprocessors.* PhD thesis, Lawrence Berkeley Laboratory, University of California, 1987.

[Bad91] S. B. Baden. Programming abstractions for dynamically partitioning and coordinating localized scientific calculations running on multiprocessors. *SIAM Journal on Scientific and Statistical Computing*, 12(1):145–157, 1991.

[Bad92] S. B. Baden. Lattice parallelism: A parallel programming model for manipulating localized non-uniform scientific data structures. In *Intel Supercomputer University Partners Conference*, Timberline Lodge, Mt. Hood, OR, April 1992.

[Bal90] V. Balasundaram. A mechanism for keeping useful internal information in parallel programming tools: The data access descriptor. *Journal of Parallel and Distributed Computing*, 9(2):154–170, June 1990.

[BBLS91] D. Bailey, J. Barton, T. Lasinski, and H. Simon. The NAS parallel benchmarks. *International Journal of Supercomputing Applications*, 5(3):63–73, Fall 1991.

[BBO+83] B. R. Brooks, R. E. Bruccoleri, B. D. Olafson, D. J. States, S. Swaminathan, and M. Karplus. CHARMM: A program for macromolecular energy, minimization and dynamics calculations. *Journal of Computational Chemistry*, 4(2):187–217, 1983.

[BCZ92] S. Benkner, B. Chapman, and H. Zima. Vienna Fortran 90. In *Scalable High Performance Computing Conference*, Williamsburg, VA, April 1992.

[BHMS91] M. Bromley, S. Heller, T. McNerney, and G. Steele, Jr. Fortran at ten gigaflops: The Connection Machine convolution compiler. In *Proceedings of the ACM SIGPLAN '91 Conference on Program Language Design and Implementation*, Toronto, Canada, June 1991.

[Bia91] E. S. Biagioni. *Scan Directed Load Balancing.* PhD thesis, University of North Carolina at Chapel Hill, 1991.

[BK93] S. B. Baden and S. R. Kohn. Portable parallel programming of numerical problems under the LPAR system. Technical Report CS93-330, Department of Computer Science and Engineering, University of California, San Diego, 1993.

[BKF94] S. B. Baden, S. R. Kohn, and S. J. Fink. Programming with LPARX. In *Intel Supercomputer User's Group Meeting*, June 1994. Also available via anonymous ftp from `cs.ucsd.edu` as `pub/baden/tr/cs94-377.ps`.

[BKP93] F. Bodin, L. Kervella, and T. Priol. Fortran-S: A Fortran interface for shared virtual memory architectures. In *Proceedings of Supercomputing '93*, Portland, OR, November 1993.

[Ble90] G. E. Blelloch. *Vector Models for Data-Parallel Computing.* The MIT Press, 1990.

[BP90] K. P. Belkhale and P. Prithviraj. Recursive partitions on multiprocessors. In *Proceedings of the 5th Distributed Memory Computing Conference*, pages 930–938, 1990.

[BS90] H. Berryman and J. Saltz. A manual for PARTI runtime primitives. ICASE Interim Report 13, Institute for Computer Application in Science and Engineering, Hampton, VA, September 1990.

[BSGM90] H. Berryman, J. Saltz, W. Gropp, and R. Mirchandaney. Krylov methods preconditioned with incompletely factored matrices on the CM-2. *Journal of Parallel and Distributed Computing*, 8:186–190, 1990.

[BT88] Henri E. Bal and Andrew S. Tanenbaum. Distributed programming with shared data. In *Proceedings of the IEEE CS 1988 International Conference on Computer Languages*, pages 82–91, October 1988.

[Bur90] M. Burke. An interval-based approach to exhaustive and incremental interprocedural data-flow analysis. *ACM Transactions on Programming Languages and Systems*, 12(3):341–395, July 1990.

[BZ91] Brian N. Bershad and Matthew J. Zekauskas. Shared memory parallel programming with entry consistency for distributed memory multiprocessors. Technical Report CMU-CS-91-170, Carnegie-Mellon University, September 1991.

[CAL+89] J. Chase, F. Amador, E. Lazowska, H. Levy, and R. Littlefield. The Amber system: Parallel programming on a network of multiprocessors. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, pages 147–158, December 1989.

[CBZ91] J.B. Carter, J.K. Bennett, and W. Zwaenepoel. Implementation and performance of Munin. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, pages 152–164, October 1991.

[CCRS91] C. Chase, A. Cheung, A. Reeves, and M. Smith. Paragon: A parallel programming environment for scientific applications using communication structures. In *Proceedings of the 1991 International Conference on Parallel Processing*, St. Charles, IL, August 1991.

[CF89] A. Cox and R. Fowler. The implementation of a coherent memory abstraction on a NUMA multiprocessor: Experiences with Platinum. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, December 1989.

[CG89] N. Carriero and D. Gelernter. Linda in context. *Communications of the ACM*, 32(4):444–458, April 1989.

[CHK94] T. W. Clark, R. v. Hanxleden, and K. Kennedy. Experiences on data-parallel programming. Technical Report CRPC-TR94495-S, Center for Research on Parallel Computation, Rice University, December 1994. Available via anonymous ftp from `softlib.rice.edu` as `pub/CRPC-TRs/reports/CRPC-TR94495-S`.

[CHMS94] T. W. Clark, R. v. Hanxleden, J. A. McCammon, and L. R. Scott. Parallelization using spatial decomposition for molecular dynamics. In *Scalable High Performance Computing Conference*, Knoxville, TN, May 1994. Available via anonymous ftp from `softlib.rice.edu` as `pub/CRPC-TRs/reports/CRPC-TR93356-S`.

[Chr91] P. Christy. Virtual processors considered harmful. In *Proceedings of the 6th Distributed Memory Computing Conference*, Portland, OR, April 1991.

[CK88] D. Callahan and K. Kennedy. Compiling programs for distributed-memory multiprocessors. *Journal of Supercomputing*, 2:151–169, October 1988.

[CK92] S. Carr and K. Kennedy. Scalar replacement in the presence of conditional control flow. Technical Report TR92283, Rice University, CRPC, November 1992. To appear in *Software – Practice & Experience*.

[CLR90] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms*. The MIT Press, Cambridge, MA, 1990.

[CM69] J. Cocke and R. Miller. Some analysis techniques for optimizing computer programs. In *Proceedings of the 2nd Annual Hawaii International Conference on System Sciences*, pages 143–146, 1969.

[CM90] T. W. Clark and J. A. McCammon. Parallelization of a molecular dynamics non-bonded force algorithm for MIMD architectures. *Computers & Chemistry*, 14(3):219–224, 1990.

[CMZ92] B. Chapman, P. Mehrotra, and H. Zima. Programming in Vienna Fortran. *Scientific Programming*, 1(1):31–50, Fall 1992.

[Coc70] J. Cocke. Global common subexpression elimination. *ACM SIGPLAN Notices*, 5(7):20–24, 1970.

[Dah90] D. Dahl. Mapping and compiled communication on the connection machine system. In *Proceedings of the 5th Distributed Memory Computing Conference*, Charleston, SC, April 1990.

[Das94] R. Das. *Compilation Techniques for Irregular Problems on Parallel Machines*. PhD thesis, The College of William and Mary in Virginia, 1994.

[DC92] H. Dietz and W. Cohen. A control-parallel programming model implemented on SIMD hardware. In *Proceedings of the Fifth Workshop on Languages and*

Compilers for Parallel Computing, pages 311–325, New Haven, CT, August 1992.

[DGS93] E. Duesterwald, R. Gupta, and M. L. Soffa. A practical data flow framework for array reference analysis and its use in optimizations. *ACM SIGPLAN Notices*, 28(6):68–77, June 1993. *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation.*

[Dha88a] D.M. Dhamdhere. A fast algorithm for code movement optimization. *ACM SIGPLAN Notices*, 23(10):172–180, 1988.

[Dha88b] D.M. Dhamdhere. Register assignment using code placement techniques. *Computer Languages*, 13(2):75–93, 1988.

[Dha91] D.M. Dhamdhere. Practical adaptation of the global optimization algorithm of Morel and Renvoise. *ACM Transactions on Programming Languages and Systems*, 13(2):291–294, April 1991.

[DHU+93] R. Das, Y.-S. Hwang, M. Uysal, J. Saltz, and A. Sussman. Applying the CHAOS/PARTI library to irregular problems in computational chemistry and computational aerodynamics. In *Proceedings of the Scalable Parallel Libraries Conference, Mississippi State University, Starkville, MS.* IEEE Computer Society Press, October 1993.

[DK83] D.M. Dhamdhere and J.S. Keith. Characterization of program loops in code optimization. *Computer Languages*, 8:69–76, 1983.

[DK93] D. M. Dhamdhere and U. P. Khedker. Complexity of bidirectional data flow analysis. In *Conference Record of the Twentieth Annual ACM Symposium on Principles of Programming Languages*, pages 397–408, Charleston, South Carolina, January 1993.

[DMS+92] R. Das, D. Mavriplis, J. Saltz, S. Gupta, and R. Ponnusamy. The design and implementation of a parallel unstructured Euler solver using software primitives, AIAA-92-0562. In *Proceedings of the 30th Aerospace Sciences Meeting.* AIAA, January 1992.

[DP93] D.M. Dhamdhere and H. Patil. An elimination algorithm for bidirectional data flow problems using edge placement. *ACM Transactions on Programming Languages and Systems*, 15(2):312–336, April 1993.

[DPSM91] R. Das, R. Ponnusamy, J. Saltz, and D. Mavriplis. Distributed memory compiler methods for irregular problems — Data copy reuse and runtime partitioning. ICASE Report 91-73, Institute for Computer Application in Science and Engineering, Hampton, VA, September 1991.

[DRZ92] D.M. Dhamdhere, B.K. Rosen, and F.K. Zadeck. How to analyze large programs efficiently and informatively. In *Proceedings of the ACM SIGPLAN '92 Conference on Program Language Design and Implementation*, pages 212–223, San Francisco, CA, June 1992.

[DS88] K. Drechsler and M. Stadel. A solution to a problem with Morel and Renvoise's "Global optimization by suppression of partial redundancies". *ACM Transactions on Programming Languages and Systems*, 10(4):635–640, October 1988.

[DSvH93] R. Das, J. Saltz, and R. v. Hanxleden. Slicing analysis and indirect accesses to distributed arrays. In U. Banerjee et al., editor, *Lecture Notes in Computer Science*, volume 769, pages 152–168. Springer, Berlin, August 1993. From the *Proceedings of the Sixth Workshop on Languages and Compilers for Parallel Computing*, Portland, OR. Available via anonymous ftp from `softlib.rice.edu` as `pub/CRPC-TRs/reports/CRPC-TR93319-S`.

[ELZ86] D. Eager, E. D. Lazowska, and J. Zahorjan. A comparison of receiver-initiated and sender-initiated adaptive load sharing. *Performance Evaluation*, 6:53–68, 1986.

[FB95] S. J. Fink and S. B. Baden. Run-time data distribution for block-structured applications on distributed memory computers. In *Seventh SIAM Conf. on Parallel Proc. for Scientific Computing*, February 1995. Also available via anonymous ftp from `cs.ucsd.edu` as `pub/baden/tr/cs94-386.ps`.

[FHK+90] G. C. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C. Tseng, and M. Wu. Fortran D language specification. Technical Report TR90-141, Dept. of Computer Science, Rice University, December 1990. Revised April, 1991.

[FO90] I. Foster and R. Overbeek. Bilingual parallel programming. In *Advances in Languages and Compilers for Parallel Computing*, Irvine, CA, August 1990. The MIT Press.

[FT90] I. Foster and S. Taylor. *Strand: New Concepts in Parallel Programming*. Prentice-Hall, Englewood Cliffs, NJ, 1990.

[GAY91] E. Gabber, A. Averbuch, and A. Yehudai. Experience with a portable parallelizing Pascal compiler. In *Proceedings of the 1991 International Conference on Parallel Processing*, St. Charles, IL, August 1991.

[GB88] W. F. van Gunsteren and H. J. C. Berendsen. GROMOS: GROningen MOlecular Simulation software. Technical report, Laboratory of Physical Chemistry, University of Groningen, Nijenborgh, The Netherlands, 1988.

[GB91] M. Gupta and P. Banerjee. Automatic data partitioning on distributed memory multiprocessors. In *Proceedings of the 6th Distributed Memory Computing Conference*, Portland, OR, April 1991.

[GB92] M. Gupta and P. Banerjee. Compile-time estimation of communication costs on multicomputers. In *Proceedings of the 6th International Parallel Processing Symposium*, Beverly Hills, CA, March 1992.

[Ger90] M. Gerndt. Updating distributed variables in local computations. *Concurrency: Practice and Experience*, 2(3):171–193, September 1990.

[GS90] T. Gross and P. Steenkiste. Structured dataflow analysis for arrays and its use in an optimizing compiler. *Software—Practice and Experience*, 20(2):133–155, February 1990.

[GS93] M. Gupta and E. Schonberg. A framework for exploiting data availability to optimize communication. In *Proceedings of the Sixth Workshop on Languages and Compilers for Parallel Computing*, Portland, OR, August 1993.

[Gup92] M. Gupta. *Automatic Data Partitioning on Distributed Memory Multicomputers*. PhD thesis, College of Engineering, University of Illinois at Urbana-Champaign, September 1992.

[GV91] E. Granston and A. Veidenbaum. Detecting redundant accesses to array data. In *Proceedings of Supercomputing '91*, Albuquerque, NM, November 1991.

[GW76] S. Graham and M. Wegman. A fast and usually linear algorithm for global data flow analysis. *Journal of the ACM*, 23(1):172–202, January 1976.

[HA90] D. Hudak and S. Abraham. Compiler techniques for data partitioning of sequentially iterated parallel loops. In *Proceedings of the 1990 ACM International Conference on Supercomputing*, Amsterdam, The Netherlands, June 1990.

[Han89] R. v. Hanxleden. Parallelizing dynamic processes. Master's thesis, Dept. ofhpcnComputer Science, The Pennsylvania State University, August 1989.

[Han92] R. v. Hanxleden. Compiler support for machine independent parallelization of irregular problems. Technical Report CRPC-TR92301-S, Center for Research on Parallel Computation, Rice University, November 1992. Ph.D. Thesis Proposal.

[Han93] R. v. Hanxleden. Handling irregular problems with Fortran D — A preliminary report. In *Proceedings of the Fourth Workshop on Compilers for Parallel Computers*, pages 353–364, Delft, The Netherlands, December 1993. D Newsletter #9, available via anonymous ftp from `softlib.rice.edu` as `pub/CRPC-TRs/reports/CRPC-TR93339-S`.

[Hav93] P. Havlak. Construction of thinned gate single-assignment form. In *Proceedings of the Sixth Workshop on Languages and Compilers for Parallel Computing*, Portland, OR, August 1993.

[Hav94] P. Havlak. *Interprocedural Symbolic Analysis*. PhD thesis, Rice University, May 1994. Available as Technical Report CRPC-TR94451-S.

[HC88] P. N. Hilfinger and P. Colella. FIDIL: A language for scientific programming. Technical Report UCRL-98057, Lawrence Livermore National Laboratory, January 1988.

[HHKT92] M. W. Hall, S. Hiranandani, K. Kennedy, and C. Tseng. Interprocedural compilation of Fortran D for MIMD distributed-memory machines. In *Proceedings of Supercomputing '92*, Minneapolis, MN, November 1992.

[Hig93] High Performance Fortran Forum. High Performance Fortran language specification, version 1.0. Technical Report CRPC-TR92225, Center for Research on Parallel Computation, Rice University, Houston, TX, 1992 (revised Jan. 1993). To appear in *Scientific Programming*, July 1993.

[HK91] P. Havlak and K. Kennedy. An implementation of interprocedural bounded regular section analysis. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):350–360, July 1991.

[HK92] R. v. Hanxleden and K. Kennedy. Relaxing SIMD control flow constraints using loop transformations. In *Proceedings of the ACM SIGPLAN '92 Conference on Program Language Design and Implementation*, pages 188–199, San Francisco, CA, June 1992. ACM Press. Available via anonymous ftp from `softlib.rice.edu` as `pub/CRPC-TRs/reports/CRPC-TR92207-S`.

[HK93] R. v. Hanxleden and K. Kennedy. A code placement framework and its application to communication generation. Technical Report CRPC-TR93337-S, Center for Research on Parallel Computation, Rice University, October 1993. Available via anonymous ftp from `softlib.rice.edu` as `pub/CRPC-TRs/reports/CRPC-TR93337-S`.

[HKK+91] S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, and C. Tseng. An overview of the Fortran D programming system. In *Proceedings of the Fourth Workshop on Languages and Compilers for Parallel Computing*, Santa Clara, CA, August 1991.

[HKK+92] R. v. Hanxleden, K. Kennedy, C. Koelbel, R. Das, and J. Saltz. Compiler analysis for irregular problems in Fortran D. In U. Banerjee et al., editor, *Lecture Notes in Computer Science*, volume 757, pages 97–111. Springer, Berlin, August 1992. From the *Proceedings of the Fifth Workshop on Languages and Compilers for Parallel Computing*, New Haven, CT. Available via anonymous ftp from `softlib.rice.edu` as `pub/CRPC-TRs/reports/CRPC-TR92287-S`.

[HKT91] S. Hiranandani, K. Kennedy, and C. Tseng. Compiler optimizations for Fortran D on MIMD distributed-memory machines. In *Proceedings of Supercomputing '91*, Albuquerque, NM, November 1991.

[HKT92a] S. Hiranandani, K. Kennedy, and C. Tseng. Compiler support for machine-independent parallel programming in Fortran D. In J. Saltz and P. Mehrotra, editors, *Languages, Compilers, and Run-Time Environments for Distributed Memory Machines*. North-Holland, Amsterdam, The Netherlands, 1992.

[HKT92b] S. Hiranandani, K. Kennedy, and C. Tseng. Evaluation of compiler optimizations for Fortran D on MIMD distributed-memory machines. In *Proceedings of the 1992 ACM International Conference on Supercomputing*, Washington, DC, July 1992.

[HM88] P. Hudak and E. Mohr. Graphinators and the duality of SIMD and MIMD. In *Proceedings of the 1988 ACM Conference on Lisp and Functional Programming*, pages 224–234, July 1988.

[HPE94] M. Hahad, T. Priol, and J. Erhel. Irregular loop patterns compilation on distributed shared memory multiprocessors. Publication Interne 862, IRISA, Rennes, France, September 1994. Available via anonymous ftp from `ftp.irisa.fr` as `techreports/1994/PI-862.ps.Z`.

[HQL+91] P. Hatcher, M. Quinn, A. Lapadula, B. Seevers, R. Anderson, and R. Jones. Data-parallel programming on MIMD computers. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):377–383, July 1991.

[HS91] R. v. Hanxleden and L. R. Scott. Parallelizing dynamic processes on message passing architectures. In J. Dongorra et al., editor, *Proceedings of the Fifth SIAM Conference on Parallel Processing for Scientific Computing*, pages 451–455, March 1991.

[HtEBBW] Little Red Riding Hood and the Eight Big Bad Wolves. Evaluating the performance of the GSA system under BB race conditions. Valhalla Press. In preparation.

[IFKF90] K. Ikudome, G. Fox, A. Kolawa, and J. Flower. An automatic and symbolic parallelization system for distributed memory parallel computers. In *Proceedings of the 5th Distributed Memory Computing Conference*, Charleston, SC, April 1990.

[JD82] S.M. Joshi and D.M. Dhamdhere. A composite hoisting-strength reduction transformation for global program optimization, parts I & II. *International Journal of Computer Mathematics*, 11:21–41, 111–126, 1982.

[KCZ92] P. Keleher, A. Cox, and W. Zwaenepoel. Lazy consistency for software distributed shared memory. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 13–21, May 1992.

[Ken71] K. Kennedy. A global flow analysis algorithm. *International Journal of Computer Mathematics*, 3:5–15, 1971.

[KLS90] K. Knobe, J. Lukas, and G. Steele, Jr. Data optimization: Allocation of arrays to reduce communication on SIMD machines. *Journal of Parallel and Distributed Computing*, 8(2):102–118, February 1990.

[KLS+94] C. Koelbel, D. Loveman, R. Schreiber, G. Steele, Jr., and M. Zosel. *The High Performance Fortran Handbook.* The MIT Press, Cambridge, MA, 1994.

[KM91] C. Koelbel and P. Mehrotra. Programming data parallel algorithms on distributed memory machines using Kali. In *Proceedings of the 1991 ACM International Conference on Supercomputing*, Cologne, Germany, June 1991.

[KMCKC93] U. Kremer, J. Mellor-Crummey, K. Kennedy, and A. Carle. Automatic data layout for distributed-memory machines in the D programming environment. In Christoph W. Kessler, editor, *Automatic Parallelization — New Approaches to Code Generation, Data Distribution, and Performance Prediction*, pages 136–152. Vieweg Advanced Studies in Computer Science, Verlag Vieweg, Wiesbaden, Germany, 1993. Also available as technical report CRPC-TR93-298-S, Rice University.

[KMSB90] C. Koelbel, P. Mehrotra, J. Saltz, and S. Berryman. Parallel loops on distributed machines. In *Proceedings of the 5th Distributed Memory Computing Conference*, Charleston, SC, April 1990.

[KMT91] K. Kennedy, K. S. McKinley, and C. Tseng. Analysis and transformation in the ParaScope Editor. In *Proceedings of the 1991 ACM International Conference on Supercomputing*, Cologne, Germany, June 1991.

[KMV90] C. Koelbel, P. Mehrotra, and J. Van Rosendale. Supporting shared data structures on distributed memory machines. In *Proceedings of the Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Seattle, WA, March 1990.

[KRS92] J. Knoop, O. Rüthing, and B. Steffen. Lazy code motion. In *Proceedings of the ACM SIGPLAN '92 Conference on Program Language Design and Implementation*, San Francisco, CA, June 1992.

[Kus86] B. C. Kuszmaul. Simulating applicative architectures on the Connection Machine. Master's thesis, Massachusetts Institute of Technology, 1986.

[LC91] J. Li and M. Chen. Compiling communication-efficient programs for massively parallel machines. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):361–376, July 1991.

[LH89] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *IEEE Transactions on Computer Systems*, 7(4):321–359, November 1989.

[LLG+90] D. Lenoski, J. Laudon, K Gharachorloo, A. Gupta, and J. Hennessy. The directory-based cache coherence protocol for the DASH multiprocessor. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, May 1990.

[LS91] S. Lucco and O. Sharp. Parallel programming with coordination structures. In *Conference Record of the Eighteenth ACM Symposium on the Principles of Programming Languages*, Orlando, FL, January 1991.

[Luc88] B. J. Lucier. Performance evaluation for multiprocessors programmed using monitors. In *Proceedings of the 1988 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, volume 16 of *SIGMETRICS Performance Evaluation Review*, 1988.

[Mas91] MasPar Computer Corporation, Sunnyvale, CA. *MasPar Fortran Reference Manual*, 1991.

[Mav91] D. Mavriplis. Three dimensional unstructured multigrid for the euler equations. Technical Report 91-41, Institute for Computer Application in Science and Engineering, Hampton, VA, May 1991.

[MR79] E. Morel and C. Renvoise. Global optimization by suppression of partial redundancies. *Communications of the ACM*, 22(2):96–103, February 1979.

[MR90] T. Marlowe and B. Ryder. Properties of data flow frameworks. *Acta Informatica*, 28:121–163, 1990.

[MSMB90] S. Mirchandaney, J. Saltz, P. Mehrotra, and H. Berryman. A scheme for supporting automatic data migration on multicomputers. In *Proceedings of the 5th Distributed Memory Computing Conference*, Charleston, SC, April 1990.

[MSS+88] R. Mirchandaney, J. Saltz, R. Smith, D. Nicol, and K. Crowley. Principles of runtime support for parallel processors. In *Proceedings of the Second International Conference on Supercomputing*, pages 140–152, St. Malo, France, July 1988. ACM Press.

[MV90] P. Mehrotra and J. Van Rosendale. Programming distributed memory architectures using Kali. In *Advances in Languages and Compilers for Parallel Computing*, Irvine, CA, August 1990. The MIT Press.

[PB94] J. R. Pilkington and S. B. Baden. Partitioning with spacefilling curves. Technical Report CS94-349, Department of Computer Science and Engineering, University of California, San Diego, March 1994.

[PSC93a] R. Ponnusamy, J. Saltz, and A. Choudhary. Runtime compilation techniques for data partitioning and communication schedule reuse. In *Supercomputing*

'93, pages 361–370. IEEE Computer Society Press, November 1993. Technical Report CS-TR-3055 and UMIACS-TR-93-32, University of Maryland, April '93. Available via anonymous ftp from `hyena.cs.umd.edu`.

[PSC+93b] R. Ponnusamy, J. Saltz, A. Choudhary, Y.-S. Hwang, and G. Fox. Runtime support and compilation methods for user-specified data distributions. Technical Report CS-TR-3194 and UMIACS-TR-93-135, University of Maryland, November 1993. Available via anonymous ftp from `hyena.cs.umd.edu`.

[PT91] M. Philippsen and W. F. Tichy. Modula-2* and its compilation. In *First International Conference of the Austrian Center for Parallel Computation*, Salzburg, Austria, September 1991.

[RA90] R. Ruhl and M. Annaratone. Parallelization of FORTRAN code on distributed-memory parallel processors. In *Proceedings of the 1990 ACM International Conference on Supercomputing*, Amsterdam, The Netherlands, June 1990.

[RAK88] U. Ramachandran, M. Ahamad, and Y. Khalidi. Unifying synchronization and data transfer in maintaining coherence of distributed shared memory. Technical Report GIT-CS-88/23, Georgia Institute of Technology, June 1988.

[RP86] B.G. Ryder and M.C. Paull. Elimination algorithms for data flow analysis. *ACM Computing Surveys*, 18:77–316, 1986.

[RP89] A. Rogers and K. Pingali. Process decomposition through locality of reference. In *Proceedings of the ACM SIGPLAN '89 Conference on Program Language Design and Implementation*, Portland, OR, June 1989.

[RS87] J. Rose and G. Steele, Jr. C*: An extended C language for data parallel programming. In L. Kartashev and S. Kartashev, editors, *Proceedings of the Second International Conference on Supercomputing*, Santa Clara, CA, May 1987.

[RS89] J. Ramanujam and P. Sadayappan. A methodology for parallelizing programs for multicomputers and complex memory multiprocessors. In *Proceedings of Supercomputing '89*, Reno, NV, November 1989.

[RSW91] M. Rosing, R. Schnabel, and R. Weaver. The DINO parallel programming language. *Journal of Parallel and Distributed Computing*, 13(1):30–42, September 1991.

[SBW90] J. Saltz, H. Berryman, and J. Wu. Multiprocessors and runtime compilation. ICASE Report 90-59, Institute for Computer Application in Science and Engineering, Hampton, VA, September 1990.

[SCMB90] J. Saltz, K. Crowley, R. Mirchandaney, and H. Berryman. Run-time scheduling and execution of loops on message passing machines. *Journal of Parallel and Distributed Computing*, 8(2):303–312, 1990.

[SH91] J. P. Singh and J. L. Hennessy. An empirical investigation of the effectiveness and limitations of automatic parallelization. In *Proceedings of the International Symposium on Shared Memory Multiprocessing*, Tokyo, Japan, April 1991.

[SHG92] J. P. Singh, J. L. Hennessy, and A. Gupta. Implications of hierarchical N-body methods for multiprocessor architecture. Technical Report CSL-TR-92-506, Stanford University, 1992.

[SLY90] Z. Shen, Z. Li, and P. Yew. An empirical study of Fortran programs for parallelizing compilers. *IEEE Transactions on Parallel and Distributed Systems*, 1(3):356–364, July 1990.

[SM90] T. P. Straatsma and J. Andrew McCammon. ARGOS, a vectorized general molecular dynamics program. *Journal of Computational Chemistry*, II(8):943–951, 1990.

[SM91] J. Shen and J. A. McCammon. Molecular dynamics simulation of Superoxide interacting with Superoxide Dismutase. *Chemical Physics*, 158:191–198, 1991.

[Soc90] D. Socha. Compiling single-point iterative programs for distributed memory computers. In *Proceedings of the 5th Distributed Memory Computing Conference*, Charleston, SC, April 1990.

[Sor89] A. Sorkin. Some comments on "A solution to a problem with Morel and Renvoise's 'Global optimization by suppression of partial redundancies'". *ACM Transactions on Programming Languages and Systems*, 11(4):666–668, October 1989.

[SPBR91] J. Saltz, S. Petiton, H. Berryman, and A. Rifkin. Performance effects of irregular communication patterns on massively parallel multicomputers. ICASE Report 91-12, Institute for Computer Application in Science and Engineering, Hampton, VA, January 1991.

[SS90] L. Snyder and D. Socha. An algorithm producing balanced partitionings of data arrays. In *Proceedings of the 5th Distributed Memory Computing Conference*, Charleston, SC, April 1990.

[SWW92] R. Sawdayi, G. Wagenbreth, and J. Williamson. MIMDizer: Functional and data decomposition. In J. Saltz and P. Mehrotra, editors, *Languages, Compilers, and Run-Time Environments for Distributed Memory Machines.* Elsevier, Amsterdam, The Netherlands, 1992.

[Tar74] R. E. Tarjan. Testing flow graph reducibility. *Journal of Computer and System Sciences*, 9:355–365, 1974.

[Thi91] Thinking Machines Corporation, Cambridge, MA. *CM Fortran Reference Manual*, 1991.

[TP90] S. Tomboulian and M. Pappas. Indirect addressing and load balancing for faster solutions to the Mandelbrot set on SIMD architectures. In *Frontiers90: The 3rd Symposium on the Frontiers of Massively Parallel Computation*, pages 443–450, College Park, MD, October 1990.

[Tse90] P.-S. Tseng. A parallelizing compiler for distributed memory parallel computers. In *Proceedings of the ACM SIGPLAN '90 Conference on Program Language Design and Implementation*, White Plains, NY, June 1990.

[Tse93] C. Tseng. *An Optimizing Fortran D Compiler for MIMD Distributed-Memory Machines.* PhD thesis, Rice University, January 1993.

[WCSM93] Y.-T. Wong, T. W. Clark, J. Shen, and J. A. McCammon. Molecular dynamics simulation of substrate-enzyme interactions in the active site channel of superoxide dismutase. *Journal of Molecular Simulation*, 10(2–6):277–289, 1993.

[WLR90] M. Willebeek-LeMair and A. P. Reeves. Solving nonuniform problems on SIMD computers: Case study on region growing. *Journal of Parallel and Distributed Computing*, 8:135–149, 1990.

[WSBH91] J. Wu, J. Saltz, H. Berryman, and S. Hiranandani. Distributed memory compiler design for sparse problems. ICASE Report 91-13, Institute for Computer Application in Science and Engineering, Hampton, VA, January 1991.

[WSHB91] J. Wu, J. Saltz, S. Hiranandani, and H. Berryman. Runtime compilation methods for multicomputers. In *Proceedings of the 1991 International Conference on Parallel Processing*, St. Charles, IL, August 1991.

[ZBG88] H. Zima, H.-J. Bast, and M. Gerndt. SUPERB: A tool for semi-automatic MIMD/SIMD parallelization. *Parallel Computing*, 6:1–18, 1988.

Appendix A

Proofs of Correctness for Give-N-Take

A.1 Proof of correctness of the data-flow equations

To simplify the discussion of correctness for different cases of control flow, we expand a control flow path into its different *program points* [DK93], namely the entries and exits of nodes passed through. For

$$p : (\textsc{Start} =)\ n_0 \rightarrow n_1 \rightarrow \ldots \rightarrow n_s\ (= \textsc{Stop}),$$

we define the *expanded control flow path* to be

$$E(p) : ((\textsc{Start}, out) =)\ (q_0, r_0) \rightarrow (q_1, r_1) \rightarrow \ldots \rightarrow (q_t, r_t)\ (= (\textsc{Stop}, in)),$$

where $t \geq s$ and $\forall i, 0 \leq i \leq t : q_i \in N, r_i \in \{in, out\}$. (q_i, in) stands for the *entry* of q_i, and (q_i, out) denotes the *exit* of q_i. Of course, paths and expanded paths do not always have to originate at Start and terminate at Stop.

How exactly an edge $e = (m, n)$ is expanded depends on the type of edge (see Section 3.3.4).

- If e is a Forward or Jump edge, then $E(e) = (m, out) \rightarrow (n, in)$.
- If e is an Entry edge, then $E(e) = (m, in) \rightarrow (n, in)$.
- If e is a Cycle edge, then $E(e) = (m, out) \rightarrow (n, out)$.
- Note that edges of the form $(n, in) \rightarrow (n, out)$ do not correspond to actual edges in the control flow graph; we will refer to such edges as *internal edges*.
- Furthermore, there is no direct expanded equivalent for a control flow path that goes from the end of a loop through the loop header Cycle to the beginning of the loop, since it goes through the header without going through either its entry or its exit. This, however, does not affect our proofs, since we can take the full effect of each loop into account by traversing it once (we might also not execute it at all).

For example, a path (with a loop iteration)

$$p: 1 \to 2 \to 3 \to 2 \to 4$$

would be expanded into

$$E(p): \quad (1, out) \to (2, in) \to (3, in) \to (3, out) \to (2, out) \to (4, in)$$

Let x be an arbitrary entry in our data-flow universe. For a data-flow variable VAR and a node $n \in N$, let "$\mathsf{VAR}(n)$" denote $x \in \mathsf{VAR}$, and let "$\overline{\mathsf{VAR}(n)}$" denote $x \notin \mathsf{VAR}$.

A.1.1 Balance

The Balance Theorem

Balance is equivalent to matching, at run time (*i.e.*, along all possible control flow paths), all EAGER productions with succeeding LAZY productions, without any intervening EAGER production, and vice versa. This has to hold for BEFORE and AFTER solutions, but with different orientations relative to the flow of control. The following theorem expresses these constraints.

Theorem 1 *(C1: Balance.)*

Given: An expanded path

$$p: ((\textsc{Start}, out) =)\ (q_0, r_0) \to (q_1, r_1) \to \ldots \to (q_s, r_s)\ (= (\textsc{Stop}, in))$$

with an EAGER production for some e, $0 < e < s$:

$$\mathsf{RES}_{r_e}^{eager}(q_e). \tag{A.1}$$

Claim: In a BEFORE solution, there exists an $q_l \in N, e \leq l < s$, such that

$$\mathsf{RES}_{r_l}^{lazy}(q_l), \tag{A.2}$$

$$\forall q_k \in N, e < k \leq l : \overline{\mathsf{RES}_{r_k}^{eager}(q_k)}. \tag{A.3}$$

Similarly, a LAZY production $\mathsf{RES}_{r_e}^{lazy}(q_e)$ is balanced by an EAGER production. In an AFTER solution, the same holds with the flow of control reversed.

We will show the proof for balancing EAGER production in a BEFORE problem (in our communication generation application, this corresponds to guaranteeing that each READ$_{Send}$ will be matched by a READ$_{Recv}$, without any READ$_{Send}$ in between). The proofs for the other cases are analogous.

The underlying idea is to show that if an EAGER production has been placed on a path but no LAZY production has taken place yet, then this information is propagated forward (for a BEFORE problem) or backward (for an AFTER problem) by both GIVEN^{eager} and TAKEN being true and by GIVEN^{lazy} being false (control flow graph node numbers omitted here). The following contains some lemmata on which the inductive proof of Theorem 1 will be based (Section A.1.1).

Induction base lemmata

Intuitively, the different flavors of GIVEN indicate which productions are guaranteed to be available at n. The next lemma states that wherever the LAZY production of an item is available, its EAGER production must be available as well.

Lemma A.1 *(*EAGER *implies* LAZY.*)*

Claim: For all $n \in N$, the following holds:

$$\begin{aligned} \mathsf{GIVEN}_{in}^{eager}(l) &\supseteq \mathsf{GIVEN}_{in}^{lazy}(l), && \text{(A.4)} \\ \mathsf{GIVEN}^{eager}(l) &\supseteq \mathsf{GIVEN}^{lazy}(l), && \text{(A.5)} \\ \mathsf{GIVEN}_{out}^{eager}(l) &\supseteq \mathsf{GIVEN}_{out}^{lazy}(l). && \text{(A.6)} \end{aligned}$$

For the READ problem, this means intuitively the following: anything that is received at a certain point must already have been sent.

Proof:

The only difference between EAGER and LAZY is in Equation (3.12), where GIVEN^{eager} gets induced by TAKEN_{in} and GIVEN^{lazy} follows from TAKE. (3.6) implies that for all $n \in N$:

$$\mathsf{TAKEN}_{in}(n) \supseteq \mathsf{TAKE}(n). \qquad \text{(A.7)}$$

PREDS$^{\mathrm{FJ}}$(ROOT) $= \emptyset$ implies by (3.11) $\mathsf{GIVEN}_{in}^{eager}(\text{ROOT}) = \mathsf{GIVEN}_{in}^{lazy}(\text{ROOT}) = \perp$, which can serve as a base for a simple induction using (3.11), (3.12), (3.13), and (A.7) to prove (A.4), (A.5), and (A.6) for all $n \in N$.

□

The next lemma states that for each EAGER production at a node entry that is not matched yet by a LAZY production, the outstanding LAZY production is reflected by the values of GIVEN and TAKEN_{in}.

Lemma A.2 *(Production at entry.)*

Given: A node $n \in N$ such that

$$\mathsf{RES}_{in}^{eager}(n), \tag{A.8}$$

$$\overline{\mathsf{RES}_{in}^{lazy}(n)}. \tag{A.9}$$

Claim:

$$\mathsf{GIVEN}^{eager}(n), \tag{A.10}$$

$$\overline{\mathsf{GIVEN}^{lazy}(n)}, \tag{A.11}$$

$$\mathsf{TAKEN}_{in}(n). \tag{A.12}$$

Proof:

(Read the following as: "Fact (A.8) implies by rule (3.14) that (A.10) holds, and that (A.13), *i.e.*, $\overline{\mathsf{GIVEN}_{in}^{eager}(n)}$, holds as well. Fact (A.13) implies by rule (A.4) ...")

$$(A.8) \stackrel{(3.14)}{\longrightarrow} (A.10),\ \overline{\mathsf{GIVEN}_{in}^{eager}(n),} \tag{A.13}$$

$$(A.13) \stackrel{(A.4)}{\longrightarrow} \overline{\mathsf{GIVEN}_{in}^{lazy}(n)}. \tag{A.14}$$

From (A.10) and (A.13) follows (A.12) by (3.12). (A.11) can be derived from (A.14) and (A.9) via (3.14).

□

The following lemma ensures that for each EAGER production at a node exit that is not matched yet by a LAZY production, the outstanding LAZY production is reflected by the values of GIVEN and TAKEN_{in} at the successors.

Lemma A.3 *(Production at exit.)*

Given: A node $n \in N$ such that

$$\mathsf{RES}_{out}^{eager}(n), \tag{A.15}$$

$$\overline{\mathsf{RES}^{lazy}_{out}(n)}. \tag{A.16}$$

Claim: For all $s \in \textsc{Succs}^{\text{FJ}}(n)$:

$$\mathsf{GIVEN}^{eager}_{in}(s), \tag{A.17}$$

$$\overline{\mathsf{GIVEN}^{lazy}_{in}(s)}, \tag{A.18}$$

$$\mathsf{TAKEN}_{in}(s). \tag{A.19}$$

Proof:

$$(A.15) \xrightarrow{Lemma\ 3.4} (A.17), \overline{\mathsf{GIVEN}^{eager}_{out}(n)}, Lemma\ 3.4 \tag{A.20}$$

$$(A.20) \xrightarrow{(A.6)} \overline{\mathsf{GIVEN}^{lazy}_{out}(n)}. \tag{A.21}$$

(A.16), (A.21) imply (A.18) by (3.15), and (A.19) follows from (A.17) and (A.20) using (3.11).

□

Induction step lemmata

The following lemma states that whenever we enter a node n with an outstanding Lazy production, this will either be satisfied on entry of n or it will be preserved through n.

Lemma A.4 *(Balance entering a node.)*

Given: A node $n \in N$ such that

$$\mathsf{GIVEN}^{eager}_{in}(n), \tag{A.22}$$

$$\overline{\mathsf{GIVEN}^{lazy}_{in}(n)}, \tag{A.23}$$

$$\mathsf{TAKEN}_{in}(n). \tag{A.24}$$

Claim:

$$\overline{\mathsf{RES}^{eager}_{in}(n)}, \tag{A.25}$$

$$\mathsf{GIVEN}^{eager}(n). \tag{A.26}$$

Furthermore, it is

$$\mathsf{RES}_{in}^{lazy}(n) \tag{A.27}$$

or all of

$$\overline{\mathsf{GIVEN}^{lazy}(n)}, \tag{A.28}$$

$$\overline{\mathsf{TAKE}(n)}, \tag{A.29}$$

$$\mathsf{TAKEN}_{out}(n). \tag{A.30}$$

Proof:

(A.22) implies (A.25) by (3.14) and (A.26) by (3.12). Assume (A.27) does not hold. This together with (A.23) implies by (3.14) that (A.28) must hold, which in turn implies (A.29) by (3.12). (A.30) follows by (3.6) from (A.24) and (A.29).

□

The subsequent lemma ensures that local blocking and consumption are correctly propagated within each interval.

Lemma A.5 *(Local blocking and consumption.)*

Given: A node $n \in N$ such that

$$\overline{\mathsf{BLOCK}_{loc}(n)}, \tag{A.31}$$

$$\overline{\mathsf{TAKE}_{loc}(n)}. \tag{A.32}$$

Claim:

$$\overline{\mathsf{TAKE}(n)}, \tag{A.33}$$

$$\overline{\mathsf{BLOCK}(n)}, \tag{A.34}$$

$$\overline{\mathsf{GIVE}(n)}, \tag{A.35}$$

$$\overline{\mathsf{STEAL}(n)}. \tag{A.36}$$

Furthermore, (A.31), (A.32) (and therefore also (A.33), ... , (A.36)) hold for all $s \in \textsc{Succs}^{\text{EF}}(n)$ as well.

Proof:

First the claims for n itself: (A.32) implies (A.33) by (3.8); (A.33), (A.31) imply via (3.7) that (A.34) must hold, which in turn results in (A.35) and (A.36).

Now the recursive claim for $s \in \textsc{Succs}^{\text{EF}}(n)$. Let $s \in \textsc{Succs}^{\text{E}}(n)$. From (A.34) for n follows (A.31) for s by definition (3.3). (A.32) for s follows from (A.32) and (A.34) for n by Definition (3.8). Let $s \in \textsc{Succs}^{\text{F}}(n)$. (A.31) for s follows by definition (3.7) from (A.31), (A.33) for n. (A.32) for s follows again from (A.32) and (A.34) for n by Definition (3.8). □

The following corollary is based on the observation that the transitive closure of $\textsc{Succs}^{\text{EF}}(n)$ is the set of descendents of n that are in $T(\textsc{Header}(n))$, *i.e.*, within the interval immediately enclosing n.

Corollary A.6 *(Blocking and consumption within an interval)*

Given: A node $n \in N$ such that for all $s \in \textsc{Succs}^{\text{E}}(n)$,

$$\overline{\mathsf{BLOCK}_{loc}(s)}, \tag{A.37}$$

$$\overline{\mathsf{TAKE}_{loc}(s)}. \tag{A.38}$$

Claim: For all $m \in T(n)$, (A.31), … , (A.36) hold.

Proof: It is easy to see that for each $m \in T(n)$, there exists a path to m which originates in some $s \in \textsc{Succs}^{\text{E}}(n)$ and consist of only $\textsc{Entry}$ and $\textsc{Flow}$ edges. The corollary then follows from applying Lemma A.5 recursively along this path. □

Lemma A.7 *(Balance within a node and its interval.)*

Given: A node $n \in N$ such that

$$\mathsf{GIVEN}^{eager}(n), \tag{A.39}$$

$$\overline{\mathsf{GIVEN}^{lazy}(n)}, \tag{A.40}$$

$$\mathsf{TAKEN}_{in}(n). \tag{A.41}$$

Claim:

$$\overline{\mathsf{RES}_{out}^{eager}(n)}, \tag{A.42}$$

$$\mathsf{GIVEN}_{out}^{eager}(n), \tag{A.43}$$

$$\overline{\mathsf{GIVEN}_{out}^{lazy}(n)}, \tag{A.44}$$

$$\mathsf{TAKEN}_{out}(n). \tag{A.45}$$

Furthermore, for all $m \in T(n)$:

$$\mathsf{GIVEN}^{eager}(m), \tag{A.46}$$

$$\mathsf{GIVEN}_{out}^{eager}(m), \tag{A.47}$$

$$\overline{\mathsf{GIVEN}^{lazy}(m)}, \tag{A.48}$$

$$\overline{\mathsf{RES}_{in}^{eager}(m)}, \tag{A.49}$$

$$\overline{\mathsf{RES}_{in}^{lazy}(m)}, \tag{A.50}$$

$$\overline{\mathsf{RES}_{out}^{eager}(m)}, \tag{A.51}$$

$$\overline{\mathsf{RES}_{out}^{lazy}(m)}. \tag{A.52}$$

Finally, for all $s \in \textsc{Succs}^{\mathrm{s}}(n)$:

$$\overline{\mathsf{GIVEN}^{lazy}(s)}, \tag{A.53}$$

Proof:

We first prove the claims for n itself (claims (A.42) ... (A.45)). Then we will show that there is no consumption within $T(n)$. From there we will derive that EAGER and LAZY availability stay unchanged throughout $T(n)$ (claims (A.46), (A.48)). Proving that there will be no production of either type within $T(n)$ (claims (A.49), ... , (A.52)) concludes the proof.

$$(A.40) \stackrel{(3.12)}{\longrightarrow} \overline{\mathsf{TAKE}(n)}, \tag{A.54}$$

$$(A.41), (A.54) \stackrel{(3.6)}{\longrightarrow} (A.45), \overline{\mathsf{BLOCK}(n)}, \tag{A.55}$$

$$(A.55) \stackrel{(3.3)}{\longrightarrow} \overline{\mathsf{STEAL}(n)}, \tag{A.56}$$

$$\overline{\mathsf{GIVE}(n)}. \tag{A.57}$$

(A.44) follows by (3.13) from (A.40) and (A.57). (A.39), (A.56) imply via (3.13) that (A.43) holds, which in turn implies (A.42) by (3.15).

Having proven the claims for n itself, we will now apply Corollary A.6 to show that nothing is locally blocked or taken within $T(n)$. Let $s \in \textsc{Succs}^{\text{E}}(n)$; *i.e.*, $n = \textsc{Header}(s)$. (A.55) implies by (3.3) that (A.31) holds for s. (A.32) for s follows from (A.45), (A.54), (A.55) via (3.5). This fulfills the requirements for Corollary A.6; *i.e.*, (A.31), ... , (A.36) must hold for all $m \in T(n)$.

Let $l = \textsc{LastChild}(n)$. It follows:

$$\textsc{Succs}^{\text{FJS}}(l) = \emptyset \quad \xrightarrow{(3.4)} \quad \overline{\mathsf{TAKEN}_{out}(l)}, \tag{A.58}$$

$$(A.33), (A.58) \quad \xrightarrow{(3.6)} \quad \overline{\mathsf{TAKEN}_{in}(l)}. \tag{A.59}$$

Let $m \in \textsc{Preds}^{\text{FJ}}(l) \cap T(n)$. This implies $l \in \textsc{Succs}^{\text{FJS}}(m)$, and from (A.59) then follows by Equation (3.4) that (A.58) must hold for m as well. This together with (A.33) implies in turn (A.59) for m by (3.6). In this manner we can prove inductively that (A.58), (A.59) hold for all $t \in T(n)$, *i.e.*, there is no consumption within $T(n)$.

We proceed to prove inductively that Eager and Lazy availability stay unchanged throughout $T(n)$. First, the Eager availability (claims (A.46) and (A.47)). **Induction base**: Let $m \in \textsc{Succs}^{\text{E}}(n)$.

$$(A.39) \quad \xrightarrow{(3.11)} \quad \mathsf{GIVEN}_{in}^{eager}(m), \tag{A.60}$$

Induction step: Assume that (A.60) holds for some arbitrary $m \in T(n)$. Equation (3.12) implies (A.46) for m. It follows (A.47) from (A.46), (A.36) via (3.13). Let $s \in T(n)$ such that (A.47) holds for all $m \in \textsc{Preds}^{\text{FJ}}(s)$. Equation (3.11) then implies (A.60) for s, which concludes the induction. □ Now, the Lazy availability (claim (A.48)). **Induction base**: Let $m \in \textsc{Succs}^{\text{E}}(n)$.

$$(A.40) \quad \xrightarrow{(3.11)} \quad \overline{\mathsf{GIVEN}_{in}^{lazy}(m)}. \tag{A.61}$$

Induction step: Assume that (A.61) holds for some arbitrary $m \in T(n)$. (A.33) then implies via Equation (3.12) that (A.48) holds for m. It follows:

$$(A.48), (A.35) \quad \xrightarrow{(3.13)} \quad \overline{\mathsf{GIVEN}_{out}^{lazy}(m)}. \tag{A.62}$$

Similarly to the induction step for (A.47), we can assume to have inductively proven (A.62) for all $m \in \textsc{Preds}^{\text{FJ}}(s)$. Equation (3.11) then implies (A.61) for s, which concludes the induction. □

It remains to prove (A.53) for all $s \in \textsc{Succs}^{\text{S}}(n)$, which ensures that Jump edges do not push production back into $T(n)$. Let $s \in \textsc{Succs}^{\text{S}}(n)$; *i.e.*, there exists a Jump edge $e = (m, s)$ with $m \in T(n), s \notin T(n)$. From the lack of critical edges follows $\textsc{Preds}^{\text{FJ}}(s) = \{m\}$ (see Section 3.3.5). (A.62) for m then implies by Equation (3.11) that (A.53) holds for s. □

We conclude by proving that there is no production within $T(n)$ (claims (A.49), ... , (A.52)). Definition (3.14) implies (A.49) from (A.60), and (A.50) from (A.48). From (A.47) follows (A.51) via (3.15). Claim (A.52) is slightly more complicated, since we have to prove (A.61) to hold for all $s \in \textsc{Succs}^{\text{FJ}}(m)$, including successors connected to m through a Jump edge. This, however, corresponds to (A.53), which we just proved. Since (A.61) was already proven to hold for all $s \in T(n)$, we now know (A.61) to hold for all $s \in \textsc{Succs}^{\text{FJ}}(m)$, from which (A.52) follows by Equation (3.15).

□

Lemma A.8 *(Balance along* Forward/Jump *edges.)*

Given: Nodes $p, n \in N$ such that $p \in \textsc{Preds}^{\text{FJ}}(n)$ and

$$\mathsf{GIVEN}^{eager}_{out}(p), \tag{A.63}$$

$$\overline{\mathsf{GIVEN}^{lazy}_{out}(p)}, \tag{A.64}$$

$$\mathsf{TAKEN}_{out}(p), \tag{A.65}$$

$$\overline{\mathsf{RES}^{lazy}_{out}(p)}. \tag{A.66}$$

Claim:

$$\mathsf{GIVEN}^{eager}_{in}(n), \tag{A.67}$$

$$\overline{\mathsf{GIVEN}^{lazy}_{in}(n)}, \tag{A.68}$$

$$\mathsf{TAKEN}_{in}(n). \tag{A.69}$$

Proof:

(A.69) follows from (A.65) by (3.4). (A.63) and (A.69) imply (A.67) by (3.11). (A.64), (A.66) imply (A.68) by (3.15).

□

Proof of the Balance Theorem

Based on the lemmata developed so far, we now inductively prove Theorem 1 (as mentioned before, we will only show the EAGER, BEFORE case, the other cases are analogous).

Induction base.

Let the prerequisites for Theorem 1 be true; *i.e.*, (A.1) holds. $\mathsf{RES}^{lazy}_{r_e}(q_e)$ would correspond to (A.2) for $l = e$. (A.3) would be vacuously true, and we would be done. Let $n = q_e$. Suppose

$$\overline{\mathsf{RES}^{lazy}_{r_e}(n)}. \tag{A.70}$$

The following is the induction invariant that the induction base will prove to hold for n (if $r_e = in$) or $q_l + 1$ (for $r_e = out$):

$$\mathsf{GIVEN}^{eager}(n), \tag{A.71}$$

$$\overline{\mathsf{GIVEN}^{lazy}(n)}, \tag{A.72}$$

$$\mathsf{TAKEN}_{in}(n). \tag{A.73}$$

(The induction step will then prove that the invariant for some q_i implies that either the invariant holds for q_{i+1} as well, or that (A.70) cannot hold for q_{i+1}, in which case we would be done.) We have to differentiate between production at entry and production at exit of n. If $r_e = in$, then the invariant follows directly from (A.1), (A.70) through Lemma A.2. Suppose $r_e = out$; let $s = q_{e+1}$. We cannot prove the invariant for n itself ((A.1) actually contradicts (A.71) via (3.15) and (3.13)), but we will show that the invariant to hold for s. (A.1) implies by (3.15) that $\textsc{Succs}^{\text{FJ}}(n) \neq \emptyset$, therefore e cannot be a CYCLE edge (see Section 3.3.5). It follows $s \in \textsc{Succs}^{\text{FJ}}(n)$. (A.1), (A.70) then imply through Lemma A.3 that (A.17), (A.18), (A.19) hold for s. This implies (A.73) ($\equiv$ (A.19)) and also makes Lemma A.4 applicable ((A.17) $\equiv$ (A.22), (A.18) $\equiv$ (A.23), (A.19) $\equiv$ (A.24)). Invariant (A.71) then corresponds to (A.26). Assumption (A.70) contradicts (A.27), so the last invariant, (A.72), follows from (A.28).

Our induction has not only to prove that (A.2) will eventually come true for some q_l, $e \leq l < s$, (*i.e.*, that (A.70) will fail for q_l), but it also has to prove that (A.3) holds for all $q_k \in N, e < k \leq l$. For $r_e = in$, the induction base corresponds to $e = l$ (since we proved the invariant for q_e itself), and (A.3) is vacuously true again. For $r_e = out$, we have to prove (A.3) for $s = q_{e+1}$. This, however, is equivalent to result (A.25) of Lemma A.4 whose prerequisites were already fulfilled.

Induction step.

Let $q_k \in N$, $k > e$; let $m = q_k$, $n = q_{k+1}$. Assume (A.70), ... , (A.73) to hold for m. We want to prove that either (A.70) does not hold for n, or that (A.71), ... , (A.73) do hold for n. We also have to show (A.3) for n. We perform the induction along the edge $(m, r_k) \rightarrow (n, r_{k+1})$, based on the actual type of edge.

Case 1 (internal edge): $r_k = in$, $r_{k+1} = out$. Since this is an internal edge, it is $m = n$, and the invariant to prove is already part of the induction step assumptions. From the invariant also follows via Lemma A.7 that (A.3) ($\equiv$ (A.42)) holds for n. $\square$

Case 2 (Forward/Jump **edge):** $r_k = out$, $r_{k+1} = in$. We can apply Lemma A.7 for m ((A.39) $\equiv$ (A.71), (A.40) $\equiv$ (A.72), (A.41) $\equiv$ (A.73)). This lemma, together with (A.70), implies the prerequisites for Lemma A.8 (it is (A.43) $\equiv$ (A.63), (A.44) $\equiv$ (A.64), (A.45) $\equiv$ (A.65), (A.70) $\equiv$ (A.66)). Lemma A.8 in turn makes Lemma A.4 applicable ((A.67) $\equiv$ (A.22), (A.68) $\equiv$ (A.23), (A.69) $\equiv$ (A.24)). As in the induction base, it follows that for n either (A.70) does not hold or (A.71), (A.72), (A.73) hold; (A.3) ($\equiv$ (A.25)) follows as well. $\square$

Case 3 (Entry **edge):** $r_k = r_{k+1} = in$. We can apply Lemma A.7 for m, which then states in (A.49), ... , (A.52) that there will be no production anywhere within $T(m)$ (implying (A.3) for all nodes in $T(m)$). Therefore, we will perform an induction step that leads directly to the first q_f, $f > k$, such that $q_f \in T(m)$, $q_{f+1} \notin T(m)$. Let $p = q_f$, $s = q_{f+1}$. We want to prove the invariant (A.71), (A.72), (A.73) for s. Let $g = (p, s)$; g can be either a Cycle edge or a Jump edge.

If g is a Cycle edge, then we are exiting $T(m)$ as we entered it (*i.e.*, through it's header, m). Since in this case the induction invariant is already proven for s $(= m)$, we are done and can continue with Case 1 (with $f = k$).

If g is a Jump edge, it is $s \in \text{Succs}^{\text{S}}(m)$, $r_f = out$, $r_{f+1} = in$. Lemma A.7 states that (A.53) holds for s, and it implies in (A.45) via (3.4) that (A.73) holds for s. The same lemma states that (A.47) holds for p, which together with (A.73) implies via (3.11) that (A.60) holds for s. Now we can apply Lemma A.4 for s ((A.60) $\equiv$ (A.22), (A.61) $\equiv$ (A.23), (A.73) $\equiv$ (A.24)), and we are done. $\square$

Case 4 (Cycle **edge):** $r_k = r_{k+1} = out$. In this case, m is the last child of an interval. Since we do not allow critical edges, it is $\text{Succs}^{\text{FJS}}(m) = \emptyset$. From Equation (3.4) follows that (A.45) does not hold for n, which together with (A.73) implies via (3.6) that (A.54) does not hold. This, however, is by (3.12) a contradiction with (A.72), therefore we do not have to consider this case further. Note the implication that after an EAGER placement has occurred along p within some loop, we can never delay LAZY production past the exit of the loop. □

To conclude the proof of the theorem, we notice that since $\mathsf{TAKEN}_{out}(\text{STOP}) = \bot$, the induction invariant (A.73) eventually leads to a contradiction along p; therefore, (A.70) cannot hold for all nodes visited after q_e, and (A.1) becomes true.

□

A.1.2 Safety

Safety is guaranteed if each production is succeeded by a consumption specified in the initial input for the framework. This is equivalent to the following theorem:

Theorem 2 *(C2: Safety.)*

Given: An expanded path

$$p : ((\text{START}, out) =)\ (q_0, r_0) \rightarrow (q_1, r_1) \rightarrow \ldots \rightarrow (q_s, r_s)\ (= (\text{STOP}, in))$$

such that each loop is traversed at least once, and for some p, $0 < p < s$, it is

$$\mathsf{RES}_{r_p}. \tag{A.74}$$

Claim: there exists in a BEFORE solution a $q_c \in N$, $0 \leq p \leq c$, such that $r_c = in$ and

$$\mathsf{TAKE}_{init}(q_c). \tag{A.75}$$

In an AFTER solution, the flow of control is reversed.

Proof:

Let $m = q_p$. **Case 1:** $r_p = in$. It is

$$
\begin{aligned}
(A.74) &\overset{(3.14)}{\longrightarrow} & \mathsf{GIVEN}(m), & & \text{(A.76)}\\
&\text{and} & \overline{\mathsf{GIVEN}_{in}(m)}, & & \text{(A.77)}\\
(A.76), (A.77) &\overset{(3.12)}{\longrightarrow} & \mathsf{TAKEN}_{in}(m) & \quad \text{for an EAGER Problem,} & \text{(A.78)}\\
&\text{or} & \mathsf{TAKE}(m) & \quad \text{for a LAZY Problem.} & \text{(A.79)}
\end{aligned}
$$

The proof of the theorem follows by a straightforward induction from (A.78) and (A.79) using Equations (3.4), (3.5), (3.6), and (3.8). Note, however, that we rely on the fact that each loop is executed at least once (since GIVE takes through GIVE_{loc} also production within loop bodies into account); see Section 3.3.3 for a motivation/discussion. □

Case 2: $r_p = out$. Let $n = q_{p+1}$. From Equation (3.15) follows $\textsc{Succs}^{\text{FJ}}(m) \neq \emptyset$, the lack of critical edges subsequently implies that $n \in \textsc{Succs}^{\text{FJ}}(m)$. It is

$$(A.74) \quad \stackrel{(3.15),Lemma\ 3.4}{\longrightarrow} \quad \overline{\mathsf{GIVEN}_{out}(m)}, \qquad (A.80)$$

$$\text{and} \quad \overline{\mathsf{GIVEN}_{in}(n)}, \qquad (A.81)$$

$$(A.80),(A.81) \quad \stackrel{(3.11)}{\longrightarrow} \quad \mathsf{TAKEN}_{in}(n). \qquad (A.82)$$

Again the proof of the theorem follows by induction. □

A.1.3 Sufficiency

The Sufficiency Theorem

Sufficiency requires that each consumption is proceeded by production, without being destroyed before reaching the consumption. Furthermore, there has to be both an Eager and a Lazy production, in this order. This is implied by the following theorem.

Theorem 3 *(C3: Sufficiency.)*

Given: An expanded path

$$p : ((\textsc{Start}, out) =)\ (q_0, r_0) \rightarrow (q_1, r_1) \rightarrow \ldots \rightarrow (q_s, r_s)\ (= (\textsc{Stop}, in))$$

such that for some c, $0 < c < s$, $r_c = in$,

$$\mathsf{TAKE}_{init}(q_c). \qquad (A.83)$$

Claim: There exists in a Before solution a q_e, $0 \leq e < c$, such that

$$\mathsf{GIVE}_{init}(q_e). \qquad (A.84)$$

or

$$\mathsf{RES}^{eager}_{r_l}(q_e) \qquad (A.85)$$

In the latter case, there also exists an l, $e \leq l \leq c$, such that

$$\mathsf{RES}^{lazy}_{r_l}(q_l). \qquad (A.86)$$

Furthermore, for all i with $r_i = out$, $e \le i \le c$, it is

$$\overline{\mathsf{STEAL}_{init}(q_i)}. \tag{A.87}$$

In an AFTER solution, the flow of control is reversed.

Again, we will show the proof for a BEFORE problem (in our communication generation application, this corresponds to guaranteeing that each reference will be proceeded by a local definition or a READ_{Send} and a READ_{Recv}, without any non-local definition in between). The proof for an AFTER problem is analogous.

The basic idea of the proof is to backtrace on p from q_c until we reach a producer. While walking backwards, we keep the invariant GIVEN, and we also ensure $\overline{\mathsf{STEAL}}$ holds along all internal edges crossed. The following contains some lemmata on which an inductive proof of Theorem 3 will be based.

Induction step lemmata

Lemma A.9 *(Local availability implies global availability.)*

Claim: For all $n \in N$, the following holds.

$$\mathsf{GIVEN}_{out}(n) \supseteq \mathsf{GIVE}_{loc}(n). \tag{A.88}$$

Proof:

If $n = \text{ROOT}$, then $\mathsf{GIVEN}_{out}(\text{ROOT}) = \mathsf{GIVE}_{loc}(\text{ROOT}) = \perp$, and we are done. For $n \neq \text{ROOT}$, it is

$$\begin{aligned}
\mathsf{GIVEN}_{out}(n) &\overset{(3.13)}{=} (\mathsf{GIVE}(n) \cup \mathsf{GIVEN}(n)) - \mathsf{STEAL}(n) \\
&\overset{(3.12),(A.7)}{\supseteq} (\mathsf{GIVE}(n) \cup \mathsf{GIVEN}_{in}(n) \cup \mathsf{TAKE}(n)) - \mathsf{STEAL}(n) \quad \text{(A.89)} \\
&\overset{(A.89)}{\supseteq} (\mathsf{GIVE}(n) \cup \mathsf{TAKE}(n)) - \mathsf{STEAL}(n) \quad \text{(A.90)}
\end{aligned}$$

If $n \in \text{SUCCS}^{\text{E}}(m)$ for some $m \in N$, it is $\text{PREDS}^{\text{FJ}}(n) = \emptyset$, and (A.88) follows from (3.9).

Otherwise, assume (A.88) to hold for all $p \in \text{PREDS}^{\text{FJ}}(n)$. This implies

$$\bigcap_{p \in \text{PREDS}^{\text{FJ}}(n)} \mathsf{GIVEN}_{out}(p) \supseteq \bigcap_{p \in \text{PREDS}^{\text{FJ}}(n)} \mathsf{GIVE}_{loc}(p), \tag{A.91}$$

$$\begin{aligned}\mathsf{GIVEN}_{out}(n) &\overset{(A.89),(3.11)}{\supseteq} (\mathsf{GIVE}(n) \cup \bigcap_{p \in \textsc{Preds}^{\mathrm{FJ}}(n)} \mathsf{GIVEN}_{out}(p) \cup \mathsf{TAKE}(n)) \\ &\qquad -\mathsf{STEAL}(n) \\ &\overset{(3.9),(A.91)}{\supseteq} \mathsf{GIVE}_{loc}(n).\end{aligned}$$

□

Lemma A.10 *(Items are either propagated or stolen locally.)*

Given: A node $n \in N$ such that $\textsc{Preds}^{\mathrm{FJ}}(n) \neq \emptyset$, for all $p \in \textsc{Preds}^{\mathrm{FJ}}(n)$:

$$\mathsf{STEAL}_{loc}(p) \tag{A.92}$$

$$\text{or} \quad \mathsf{GIVEN}_{out}(p). \tag{A.93}$$

Claim:

$$\mathsf{STEAL}_{loc}(n), \tag{A.94}$$

$$\text{or} \quad \mathsf{GIVEN}_{out}(n) \tag{A.95}$$

$$\text{and} \quad \mathsf{GIVEN}(n). \tag{A.96}$$

Proof:
Assume

$$\overline{\mathsf{STEAL}_{loc}(n)}. \tag{A.97}$$

$$(A.97) \overset{(3.10)}{\longrightarrow} \overline{\mathsf{STEAL}(n)} \tag{A.98}$$

Case 1: Assume (A.93) holds for all $p \in \textsc{Preds}^{\mathrm{FJ}}(n)$.

$$(A.93) \overset{(3.11)}{\longrightarrow} \mathsf{GIVEN}_{in}(n). \tag{A.99}$$

(A.96) follows from (A.99) via (3.12), and (A.96), (A.98) together result by (3.13) in (A.95). □

Case 2: Assume $\exists p \in \textsc{Preds}^{\mathrm{FJ}}(n)$ such that (A.93) does not hold for p; this implies (A.92) for p.

$$(A.92), (A.97) \overset{(3.10)}{\longrightarrow} \mathsf{GIVE}_{loc}(p). \tag{A.100}$$

This, however, implies (A.93) for p by Lemma A.9, which is a contradiction.

□

Lemma A.11 *(Sufficiency throughout intervals.)*

Given: A node $n \in N$ such that

$$\mathsf{GIVEN}(n), \tag{A.101}$$

$$\mathsf{GIVEN}_{out}(n). \tag{A.102}$$

Claim: For $l = \textsc{LastChild}(n)$,

$$\mathsf{GIVEN}_{out}(l), \tag{A.103}$$

and for all $c \in \textsc{Children}(n)$:

$$\mathsf{STEAL}_{loc}(c), \tag{A.104}$$

$$\text{or both} \quad \mathsf{GIVEN}(c) \tag{A.105}$$

$$\text{and} \quad \mathsf{GIVEN}_{out}(c). \tag{A.106}$$

Proof:

The main result of this lemma that we are interested in is (A.103). However, we first prove inductively that (A.104) or both (A.105) and (A.106) hold for all children. Furthermore, this induction might lead us to deeper nesting levels of $T(n)$ than just the children of n.

Let $f \in \textsc{Succs}^{\text{E}}(n)$.

$$(A.101) \overset{(3.11)}{\longrightarrow} \mathsf{GIVEN}_{in}(f), \tag{A.107}$$

$$(A.107) \overset{(3.12)}{\longrightarrow} \mathsf{GIVEN}(f) \qquad (\equiv (A.105) \text{ for } f), \tag{A.108}$$

$$(A.108) \overset{(3.13)}{\longrightarrow} \mathsf{GIVEN}_{out}(f) \qquad (\equiv (A.106) \text{ for } f), \tag{A.109}$$

$$\mathit{or} \quad \mathsf{STEAL}(f), \tag{A.110}$$

$$(A.110) \overset{(3.10)}{\longrightarrow} \mathsf{STEAL}_{loc}(f) \qquad (\equiv (A.104) \text{ for } f). \tag{A.111}$$

If for all $c \in \textsc{Children}(n)$, $\textsc{Preds}^{\text{FJ}}(c) \subseteq \textsc{Children}(n)$ (*i.e.*, no $\textsc{Jump}$ edge from any loop nested within n is reaching any child of n), then we can apply Lemma A.10 to prove inductively for all $c \in \textsc{Children}(n)$ that (A.104) or both (A.106) and (A.105) hold.

Otherwise, let c be the first child of n such that there exists a JUMP edge $e = (p, c)$, and let $h \in \text{CHILDREN}(n)$ such that $p \in T(h)$ (*i.e.*, h is the header of the outermost loop exited by e). It is

$$h \in \text{PREDS}^{S}(c). \tag{A.112}$$

Since c was the first child of n reached by a JUMP edge, we can prove inductively by Lemma A.10 that (A.104) or both (A.105) and (A.106) hold for all ancestors of c, including h.

If (A.104) holds for h, then (A.112) implies (A.104) for c via (3.10). Assume (A.104) does not hold for h. Then (A.105) and (A.106) hold for h, and we can inductively apply Lemma A.11 for h. In this manner, we can perform an induction over the nesting level; due to the finite nesting depth of $T(n)$, (A.104) has eventually to hold for some header and we are done. □

This concludes the induction for (A.104) or both (A.106) and (A.105). We also have:

$$(A.102) \stackrel{(3.13)}{\longrightarrow} \overline{\mathsf{STEAL}(n)} \tag{A.113}$$

$$(A.113) \stackrel{(3.1)}{\longrightarrow} \overline{\mathsf{STEAL}_{loc}(l)}. \tag{A.114}$$

In other words, (A.104) does not hold for l, which implies (A.103) for l ($\equiv$ (A.106)) and concludes the proof of the lemma.

□

Proof of the Sufficiency Theorem

Based on the lemmata developed so far, we now prove Theorem 3 (as mentioned before, we will only show the EAGER, BEFORE case, the other cases are analogous).

Induction base. We will prove that for both flavors (EAGER and LAZY) there exists a producer q_p reaching consumer q_c. Balance then implies that they are actually in the right order.

Let $n = q_c$.

$$(A.83) \stackrel{(3.5)}{\longrightarrow} \mathsf{TAKE}(n), \tag{A.115}$$

$$(A.115) \stackrel{(3.6)}{\longrightarrow} \mathsf{TAKEN}_{in}(n). \tag{A.116}$$

This implies for both EAGER and LAZY:

$$(A.115), (A.116) \stackrel{(3.12)}{\longrightarrow} \mathsf{GIVEN}(n), \tag{A.117}$$

$$(A.117) \xrightarrow{(3.14)} \mathsf{RES}_{in}(n) \quad (A.118)$$

$$\text{or} \quad \mathsf{GIVEN}_{in}(n). \quad (A.119)$$

If (A.118) were true, then we would have reached production and would be done. Assume (A.119); this will be the induction invariant we will use for node entries.

Induction step.

Let $q_i \in N$, $i < c$; let $m = q_{i-1}$, $n = q_i$. We perform the induction along the edge $(m, r_{i-1}) \rightarrow (n, r_i)$, based on the actual type of edge. We have to prove for m that we either reach production ((A.118) at entry, or (A.120) at exit), or that availability is preserved (the invariant, (A.119) at entry and (A.121) at exit). The exit invariant (A.121) then implies via (3.13) and (3.1) that nothing is stolen ((A.87) at exit). A special case arises when traversing internal edges of interval headers, which corresponds to skipping the interval (*e.g.*, not executing a loop; see Section 3.3.3).

Case 1 (FORWARD/JUMP edge): $r_{i-1} = out$, $r_i = in$. From the invariant (A.119) for n follows for m directly from (3.15) either

$$\mathsf{RES}_{out}(m) \quad (A.120)$$

or the invariant

$$\mathsf{GIVEN}_{out}(m). \quad (A.121)$$

□

Case 2 (ENTRY edge): $r_{i-1} = r_i = in$. Since in this case $m = \text{HEADER}(n)$, it follows from the invariant (A.119) for n via (3.11) that (A.117) must hold for m. We can proceed as in the induction base to prove that at m we either reached production or preserve the invariant. □

Case 3 (internal edge): $r_{i-1} = in$, $r_i = out$. Since this is an internal edge, it is $m = n$. The invariant (A.121) for n implies by (3.13) that either (A.117) must hold for n, in which case we could proceed as in Case 2 and would be done, or that the following holds:

$$\mathsf{GIVE}(n), \quad (A.122)$$

$$(A.122) \xrightarrow{(3.2)} \mathsf{GIVE}_{init}(n), \quad (A.123)$$

$$\text{or} \quad \mathsf{GIVE}_{loc}(\text{LASTCHILD}(n)). \quad (A.124)$$

(A.123) would be equivalent to (A.84), and we would be done. If (A.123) does not hold, then we have encountered the special case mentioned above, where data are produced in $T(n)$ and not in n itself or a predecessor of n. As described in Section 3.3.3,

$\mathsf{GIVE}(n) - \mathsf{GIVE}_{init}(n)$ are the data for which we rely on them being produced in the loop, or being produced in a separate node that gets executed if the loop does not get executed. □

Case 4 (CYCLE edge): $r_{i-1} = r_i = out$. As in Case 3, (A.123) or (A.124) must hold. If (A.123) were true, we would be done. Otherwise, (A.124) would imply for m (= LASTCHILD(n)) by (A.88) that the invariant (A.121) for m, and we would be done in this case as well. □

This completes the proof of the Sufficiency Theorem.

A.2 Proof of correctness of the algorithm

After proving in Appendix A.1 the correctness of the equations form Section 3.4, this section proves the correctness of the *GiveNTake* algorithm presented in Section 3.5 by demonstrating that it computes a fixed point for these equations. We are guaranteed to reach a fixed point if each equation gets evaluated after its right hand side is fully known.

For the discussion of how the equations are linked to each other, we define the $E(e, n)$ as a shorthand for the variable defined by Equation e for node n; for example, $E(e_1, n_1) \longleftarrow E(e_2, n_2)$ expresses that $E(e_1, n_1)$ depends on $E(e_2, n_2)$. We let $S_i(n)$ denote the evaluation of the equations in set S_i for node n. We assume that each set is evaluated in increasing equation number. Therefore, constraints of the form $E(e_1, n) \longleftarrow E(e_2, n)$ are satisfied for all e_1, e_2 from the same set with $e_1 > e_2$ and will be omitted from further discussion. For example, the dependence of $\mathsf{GIVE}(n)$ on $\mathsf{STEAL}(N)$ (*i.e.*, $E(3.2, n) \longleftarrow E(3.1, n)$) is already satisfied under this assumption.

S_1 depends on itself as follows:

- $E(3.4, n) \longleftarrow E(3.6, \textsc{Succs}^{\mathrm{FJ}}(n))$;
  $E(3.7, n) \longleftarrow E(3.7, \textsc{Succs}^{\mathrm{FJ}}(n))$;
  $E(3.8, n) \longleftarrow E(3.8, \textsc{Succs}^{\mathrm{FJ}}(n))$.

  This implies that S_1 should be evaluated in BACKWARD order to make sure all the successors of n are processed before n itself.

- $E(3.3, n) \longleftarrow E(3.7, \textsc{Succs}^{\mathrm{E}}(n))$;
  $E(3.5, n) \longleftarrow E(3.6, \textsc{Succs}^{\mathrm{E}}(n))$, $E(3.8, \textsc{Succs}^{\mathrm{E}}(n))$.

  S_1 should be evaluated in an UPWARD fashion to make sure that each header can access the values of its children.

The above constraints could be satisfied by evaluating S_1 in ReversePreOrder. However, S_1 also depends on S_2 as follows:

- $E(3.1, n) \longleftarrow E(3.10, \textsc{LastChild}(n))$;
  $E(3.2, n) \longleftarrow E(3.9, \textsc{LastChild}(n))$.

  This can be satisfied by evaluating $S_1(n)$ after $S_2(\textsc{Children}(n))$.

Furthermore, S_2 depends on S_1:

- $E(3.9, n) \longleftarrow E(3.1, n),\ E(3.2, n),\ E(3.5, n)$;
  $E(3.10, n) \longleftarrow E(3.1, n),\ E(3.2, n),\ E(3.5, n)$.

  $S_2(n)$ should be computed after $S_1(n)$.

Finally, S_2 depends on itself:

- $E(3.9, n) \longleftarrow E(3.9, \textsc{Preds}^{\text{FJ}}(n))$;
  $E(3.10, n) \longleftarrow E(3.10, \textsc{Preds}^{\text{FJ}}(n))$.

  S_2 should be evaluated in Forward order.

The dependencies for S_3 and S_4 are somewhat simpler:

- $E(3.11, n) \longleftarrow E(3.6, n),\ E(3.12, \textsc{Header}(n)),\ E(3.13, \textsc{Preds}^{\text{FJ}}(n))$;
  $E(3.12, n) \longleftarrow E(3.6, n),\ E(3.5, n)$;
  $E(3.13, n) \longleftarrow E(3.2, n),\ E(3.1, n)$.

  These constraints are satisfied when evaluating S_3 in Forward, Downward fashion (*i.e.*, PreOrder) after S_1.

- $E(3.14, n) \longleftarrow E(3.6, n),\ E(3.12, n),\ E(3.11, n)$;
  $E(3.15, n) \longleftarrow E(3.4, n),\ E(3.11, n),\ E(3.11, \textsc{Succs}^{\text{FJ}}(n)),\ E(3.13, n)$.

  S_4 has to be evaluated after S_1 and S_3, in any order.

It can easily be verified that the algorithm in Figure 3.16 observes all the constraints.

The story ends in a bloodbath,
like most conjugal complications
in 10th-century Iceland.

— Rowlinson Carter (Insight Guides Iceland)

Vita

Originally from Neustadt in Holstein, Germany, Reinhard v. Hanxleden began his studies at the Christian Albrechts University in Kiel, where he received the Vordiplom in computer science and physics in 1987. He completed his M.S. in computer science at Pennsylvania State University in 1989. In 1990, he joined the Center for Research on Parallel Computation at Rice University in Houston, Texas. He was a member of the GSA Beer Bike Team in 1991, 1992 (captain), 1993, and 1994.

After completing his dissertation, R. v. Hanxleden will join the Research and Technology division of Daimler-Benz AG, Berlin.