Characterizing Shared Memory and Communication Performance: A Case Study of the Convex SPP–1000*

Gheith A. Abandah Edward S. Davidson
Advanced Computer Architecture Laboratory, Department of EECS
University of Michigan
1301 Beal Avenue, Ann Arbor, MI 48109–2122
TEL: (313) 936–2917, FAX: (313) 763–4617
gabandah,davidson@eecs.umich.edu

January 8, 1996

Keywords: Shared-memory Multiprocessor, SPP–1000, Memory Performance, Communication Performance, and Performance Evaluation.

Abstract

The objective of this paper is to develop models that characterize the memory and communication performance of shared-memory multiprocessors which is crucial for developing efficient parallel applications for them. The Convex SPP–1000, a modern scalable distributed-memory multiprocessor that supports the shared-memory programming paradigm, is used throughout as a case study. The paper evaluates and models four aspects of SPP–1000 performance: scheduling, local-memory, shared-memory, and synchronization. Our evaluation and modeling are intended to supply useful information for application and compiler development.

1 Introduction

A *distributed-memory multiprocessor* is a scalable shared-memory parallel processor that uses a high-bandwidth, low-latency interconnection network to connect processing nodes which contain processors and memory [1]. The interconnection network provides the communication channels through which nodes exchange data and coordinate their work in solving a parallel application. Different types of interconnection networks vary in throughput, number of communication links per node, and topology. Mesh, ring, and multistage interconnection network (MIN) are three of the commonly used topologies [2, 3, 4, 5].

*The University of Michigan Center for Parallel Computing, site of the SPP–1000, is partially funded by NSF grant CDA–92–14296.

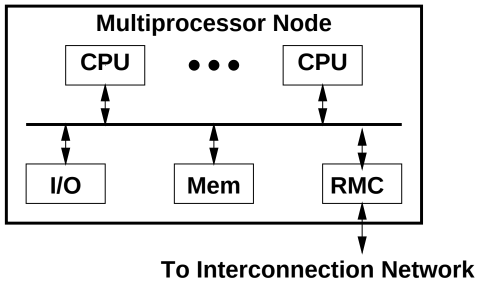

Figure 1: A multiprocessor node.

In a distributed-memory multiprocessor, the physical memory is distributed among the nodes but forms one global address space. A node is essentially a symmetric multiprocessor with a bus, or crossbar, interconnecting one or more processors, local memory, a remote memory controller (RMC), and optionally I/O. All or a large fraction of the node memory is shared so that processors from remote nodes can access it, though with a higher latency than the local access latency. The RMC handles internode memory access using point-to-point transactions. The RMC may have an Interconnect Cache (IC), as in the Convex SPP–1000, to reduce remote-memory accesses by copying referenced remote data into the IC and thus enabling future accesses of the referenced remote data to be handled locally from the IC. The RMC usually maintains cache coherence using a write-invalidate, distributed directory-based protocol.

In addition to supporting the message-passing programming paradigm, multiprocessors support the shared-memory programming paradigm. Although the latter model is simpler for developing parallel applications, programmers need to give special attention to data partitioning among processors in order to get good scalability. A parallel application with heavy remote access can run faster if its data can be rearranged to decrease remote accesses.

The achieved performance of a parallel application is a function of the application itself, the performance of the parallel computer, and the compiler and supporting libraries used. More specifically, the performance of a parallel computer is a function of its components: operating system, processors, memory subsystem, and communication subsystem. Our objective is to model the different aspects of a parallel computer's performance to enable estimating the execution time of an application given its high-level source code. This characterization would supply information that is useful for development and tuning of parallel applications and compilers.

In this paper, we present an experimental methodology and use it to characterize the SPP–1000 scheduling, memory, communication, and synchronization performance. The rest of this Section gives an overview of the SPP–1000. Section 2 characterizes the scheduling overhead of the parallel environment in managing processor allocation for parallel tasks. Section 3 presents a characterization of data cache and local-memory performance. Shared-memory performance is treated in Section 4, synchronization overhead in Section 5 and conclusions are presented in Section 6.

The experiments of this paper were carried out on the University of Michigan Center for Parallel Computing (CPC) SPP–1000 which has 4 Hypernodes with a total of 32 CPUs. All experiments were carried out during exclusive reservation (no processes running for other users). Table 1 summarizes the configuration of the nodes evaluated in this paper.

Feature	SPP–1000 data
Number of processors	32 in 4 Hypernodes
Processor	PA7100 @ 100 MHz
Instruction cache/processor (KB)	1024
Data cache/processor (KB)	1024
Main memory (MB)	1024
Memory bus (Bits)	32
Interconnect Cache (MB)	128 per Hypernode
OS version	SPP-UX 3.1.134
Fortran compiler	Convex FC 9.3

Table 1: Node configuration.

Convex SPP–1000

The Convex Exemplar SPP–1000 consists of 1 to 16 *Hypernodes* [6]. Each Hypernode contains 4 *Functional Blocks* and an I/O Interface interconnected by a 5-port crossbar, rather than a bus to achieve higher throughput. Each Functional Block contains 2 Hewlett-Packard PA-RISC 7100 processors, Memory, and some control devices (see Figures 2 and 3). The Functional Blocks communicate across the Hypernodes via four CTI (Coherent Toroidal Interconnect) Rings.

The CTI supports an extended version of the Scalable Coherent Interface (SCI) standard [7]. The CTI supports global memory accesses by providing or invalidating one cache line in response to a global memory access. Each CTI Ring is a pair of unidirectional links with a peak transfer rate of 600 MB/sec for each link.

Each processor has direct access to its instruction and data caches which are direct-mapped and virtually-addressed. The processor pair of the Functional Block share one CPU Agent to communicate with the rest of the machine. The Memory has two physical banks that are configured into three logical sections; *Hypernode Local*, *Subcomplex Global*, and *Interconnect Cache* (IC).

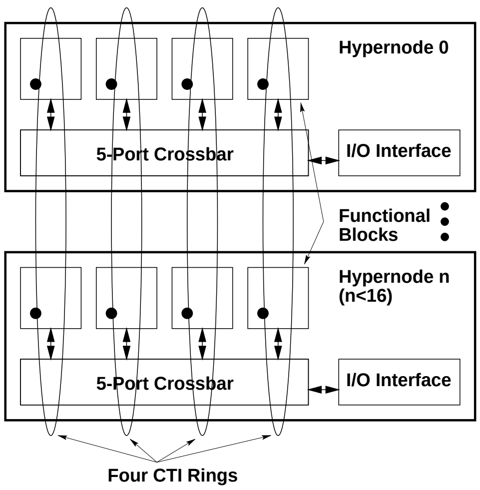

Figure 2: Convex SPP–1000 Hypernodes.

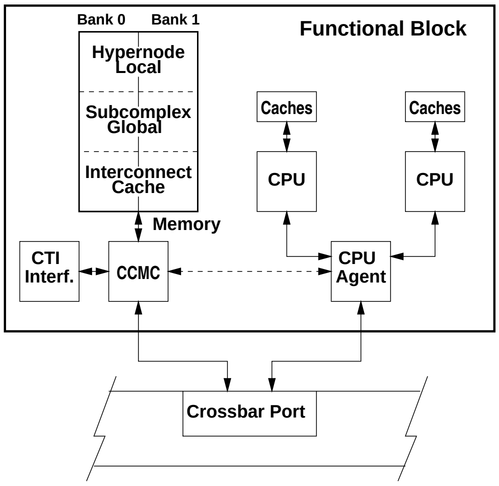

Figure 3: Convex Exemplar Functional Block.

The Hypernode Local section is used for thread and Hypernode private data. On the SPP–1000, processes run on virtual machines called *Subcomplexes*, which are arbitrary collections of processors. The Subcomplex Global section is used for shared-data and might be interleaved across the Subcomplex Hypernodes. The IC is used for holding copies of shared data that is referenced by the Hypernode processors but has home addresses in the Subcomplex Global memory of other Hypernodes.

Physical memory pages are 8-way interleaved by IC lines across the Memory banks of the four Functional Blocks of each Hypernode. Consecutive IC lines are assigned in round robin fashion, first to the four even banks, then to the four odd banks. A processor cache line is 32 Bytes wide, whereas an IC line is 64 Bytes wide, containing a pair of processor cache lines.

The Convex Coherent Memory Controller (CCMC) provides the interface between the Memory and the rest of the machine. When a processor has a cache line miss, its Agent generates a memory request to one of the four CCMCs associated with the processor's Hypernode. The CCMC accesses its Memory if it has a valid copy (in any of the three sections) and contacts other Agents if their processors have a cached copy. Otherwise it contacts a remote CCMC for service through its CTI Ring.

The SPP–1000 supports the shared-memory programming model. Its Fortran and C compilers can automatically parallelize simple loops. The compilers feature some directives that enable programmers to assist in parallelizing more difficult loops and to exploit task parallelism. The SPP–1000 also supports the PVM [8] and MPI [9] message-passing libraries.

2 Scheduling Overhead

The scheduling time is the time needed by the parallel environment to start and finish parallel tasks on p processors. This time may include the overhead of allocating processors for the parallel task, distributing the task executable code, starting the task on the processors, and freeing the allocated processors at the task completion. In this section, we present the overhead of two aspects of SPP–1000 scheduling: static scheduling and parallel-loop scheduling.

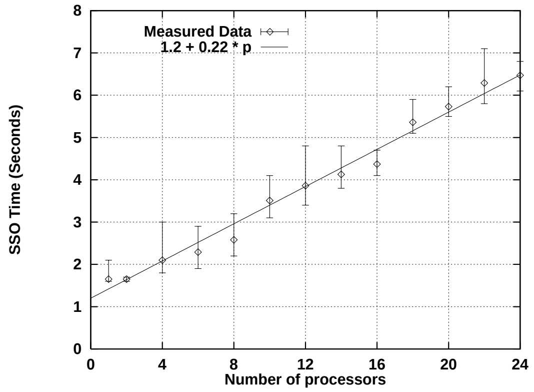

Figure 4: Static Scheduling Overhead.

The *Static Scheduling Overhead* (SSO) is for scheduling a fixed number of processors that does not change during run time. It is incurred once and is significant for short programs. To evaluate this overhead, we run a simple program on a varying number of processors where each processor prints its task id. Measuring the execution wall time is a good approximation for the SSO. Figure 4 shows the range and average of the SSO for 10 runs.

Using curve fitting, the SSO in seconds can be roughly approximated by:

$$\mathrm{SSO}(p) = 1.2 + 0.22p$$

Compared with multicomputers [10], the SPP–1000 has relatively short SSO. The SPP–1000 advantage stems from having one operating system image with central control that swiftly allocates and starts parallel tasks. Moreover, multiple processors can share the same executable binaries.

The Convex Fortran and C parallelizing compilers enable parallelizing loops. During program execution, processor 0 is always active and other available processors become active when entering a parallel loop. When processor 0 is ready to enter a parallel loop, it activates the other processors and they become idle once again at the parallel-loop completion. The overhead of processor activation and deactivation for a parallel loop is the *Parallel-Loop Scheduling Overhead* (PLSO).

To evaluate this overhead, we time a parallel loop that has one call to a subroutine consisting simply of a return statement (Null Subroutine). The loop is run many times and the average time

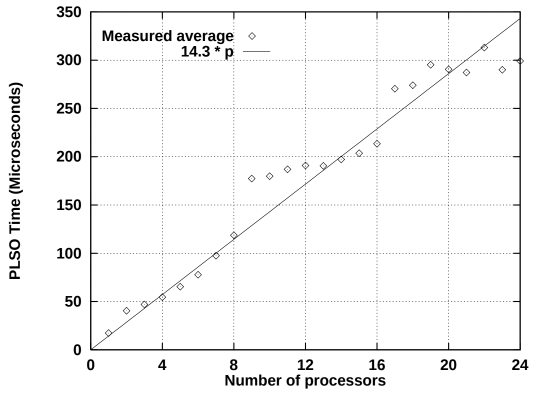

Figure 5: SPP–1000 Parallel-loop Scheduling Overhead.

for a varying number of processors is shown in Figure 5. The PLSO is approximately proportional to the number of processors and has sudden increases when the additional processor is from a new Hypernode. Using curve fitting, the PLSO can be roughly approximated by:

$$\text{PLSO}(p) = 14.3p$$

3 Local-memory Performance

We have used Load/Store kernels [11, 12, 13] to characterize the performance of the local-memory. Figure 6 shows the average time per access operation for SPP–1000 processor. The Load kernel is a serial program with an inner loop that loads double-precision (8 Bytes) 1-dimensional array elements into the floating-point registers. The array size is varied from 1 KBytes to 3 times the data cache size. The experiment is repeated for strides 1, 2, 4, ..., S, and 2S; where 2S is the first stride with the same time per load as the previous stride. One cache line thus contains S elements and results are shown for strides 1 through S. For the Store kernel the load instructions are replaced with stores. The Figure shows three regions:

1. *Hit Region*, where the array size is smaller than the cache size, and every access is a hit taking T_{H} time.

2. *Transition Region*, where some of the accesses are hits and others are misses taking T_{T} average time per reference which is a function of the stride. The width of this region equals the cache size divided by the degree of the cache associativity.

3. *Miss Region*, where the array size is big enough that every access to a new cache line is a miss taking T_{M} average time per reference which is a function of the stride.

From these simple kernels and graphs we get the information shown in Table 2 for the SPP–1000. The Memory Load and Store Bandwidths are found by dividing the number of bytes in one double-precision element by the stride 1 access time in the Miss Region.

T_{M} for strides 4 or higher is shown in Table 2. T_{M} for strides 1 and 2 can be approximated as a function of the stride 4 time which is purely a miss time since there is one reference per line. For

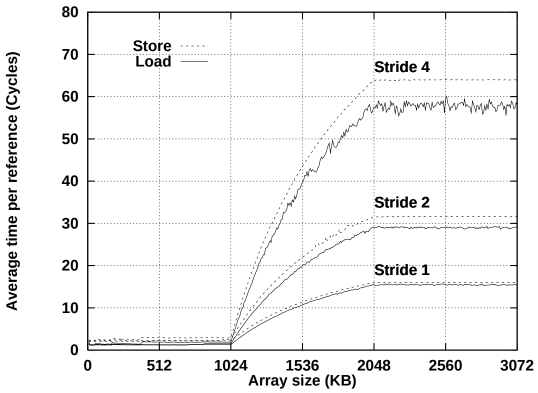

Figure 6: Access time for varying strides.

Feature	SPP–1000
Cache size (C) in KB	1024
Cache Associativity (A)	1
Cache Line size in Bytes (Elements)	32(4)
Load Hit time (T_{HL}) in cycles (nsec)	1(10)
Store Hit time (T_{HS}) in cycles (nsec)	2(20)
Load Miss time (T_{ML}) in cycles (nsec)	55.4(554)
Store Miss time (T_{MS}) in cycles (nsec)	63.3(633)
Memory Load Bandwidth (MB/sec)	52
Memory Store Bandwidth (MB/sec)	50

Table 2: Local-memory performance.

store, T_{M} is the time for stride 4 divided by the number of stores per miss (4 for stride 1 and 2 for stride 2). For load, the trailing edge effect must be taken into consideration. On a load miss, the line elements arrive one per two cycles (the one word wide memory bus takes 2 cycles to transfer a double-word element). Hence, T_{M} for stride 1 is one fourth the sum of stride 4 time and the bus transfer time for 3 elements (2×3), and T_{M} for stride 2 is one half the sum of stride 4 time and the bus transfer time to get the third element in the cache line (4×1).

The access time in the Transition Region (T_{T}) can be found as a function of T_{H} and T_{M} for the corresponding stride, namely

$$T_{\text{T}} = \frac{T_{\text{H}} \times \text{(resident lines)} + T_{\text{M}} \times \text{(nonresident lines)}}{\text{total lines}}$$

Figure 7 illustrates T_{T} for a cache with size C and associativity A. For an array of size W, the segment $W - C$ shown to the right of the A cache sets is the excess segment. When the array is accessed repetitively, assuming LRU replacement strategy, the resident lines are proportional to $A(C/A - (W - C))$ and the nonresident lines are proportional to $(A + 1)(W - C)$. Hence for a given cache system, T_{T} is given by the following non-linear function of W:

$$T_{\text{T}} = T_{\text{H}} \times \frac{C - A(W - C)}{W} + T_{\text{M}} \times \frac{(A + 1)(W - C)}{W}$$

Note that $T_{\text{T}} = T_{\text{H}}$ at $W = C$ and $T_{\text{T}} = T_{\text{M}}$ at $W = C(1 + 1/A)$.

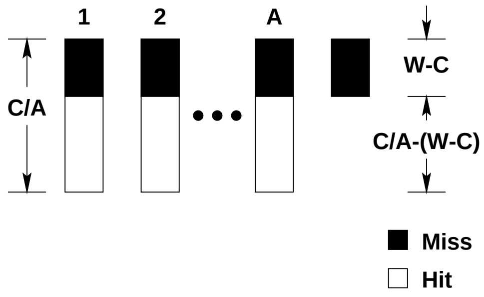

Figure 7: Cache misses in the Transition Region.

4 Shared-memory Performance

In SPP–1000 programs when a data structure is declared as shared, then multiple processors can access it directly at run-time. Since the SPP–1000 employs caching in the processor data cache and the Interconnect Cache to reduce the average access latency, there can be more than one copy of a data item. The SPP–1000 uses the Scalable Coherent Interface protocol to ensure that a processor always sees the latest update of a data item.

SCI uses write-invalidate, write-back coherence protocol where multiple processors can have a copy of a data item for read access. When a processor writes into a data item, all other copies are invalidated. So subsequent reads must get the current copy from the writer's cache. When a processor needs to replace a written cache line, it writes back the cache line to the memory. The SPP–1000 keeps track of who has copies of a cache line using distributed linked-list directories.

In this Section we present our evaluation methodology and results on SPP–1000 shared-memory performance. Subsection 4.1 evaluates the Interconnect Cache performance; 4.2 evaluates shared-memory performance when 2 processors interact in a producer-consumer access pattern; 4.3 evaluates the overhead of maintaining coherence when multiple processors are involved in shared-data access.

4.1 Interconnect Cache Performance

The Interconnect Cache (IC) is a dedicated section of the Hypernode Memory. The IC size is configurable by the system administrator, and is selected to achieve the best performance for applications that are frequently executed.

The IC in each Hypernode exploits locality of reference for the remote shared-memory data (shared data with a home memory location in some other Hypernode). Whenever remote shared-memory data is referenced by a processor, if there is a miss in the processor's data cache, followed by a miss in the Hypernode's IC, a 64-byte IC line is retrieved over the CTI through its home Hypernode. This line is stored in the IC, and the referenced 32-byte portion is stored in the processor's data cache. Hence additional references to this line that miss in the data cache can be satisfied locally from the IC until this line is replaced or invalidated due to an update by a remote Hypernode.

To evaluate the performance of the IC, we used an experiment similar to the one used for evaluating the local-memory performance. We have used a program that is run on two processors from distinct Hypernodes. The first processor allocates an array of some size and initializes it. The second processor keeps accessing this array repetitively form top to bottom with some stride. Figure 8 shows the average latency of the second processor for load and store with a variety of strides as a function of the array size.

For array sizes up through 128 MB, the array fits in the IC and we get access times similar to the local-memory access times as reported in Section 3. For larger arrays, we enter a transition region that is 128 MB wide indicating that the IC is direct mapped. For array sizes larger than 256 MB, no part of the array remains in the cache between two iterations, so every access to a new IC line generates an IC miss that is satisfied from the remote Hypernode.

When we go from stride 1 to stride 8, the average latency increases due to the increase in the number of misses per access. For stride 8 or larger, every access is a miss. Our experiments have shown that the maximum latency is for stride 32, because in addition to the fact that every access is a miss, fewer CTI Rings are used resulting in CTI congestion; strides up through 8 use 4 Rings,

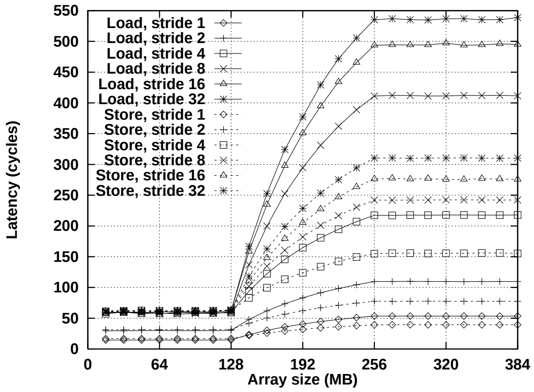

Figure 8: Interconnect Cache performance.

stride 16 uses 2 Rings, and strides of 32 or more use 1 Ring. No noticeable increase in the average latency was observed beyond stride 32.

Peak transfer rate between a remote memory and a processor is measured by the stride 1 average latency in the Miss Region (8 Bytes divided by the latency). This rate is 15 MB/sec for loads and 21 MB/sec for stores. Remote store is faster than remote load because the CTI protocol simply sends the address with the new data for stores, but sends the address and waits for the response data for loads.

4.2 Shared Read/Write Performance

In this subsection we present our results for evaluating the shared-memory performance on the SPP–1000 when 2 processors interact in a producer-consumer access pattern for shared data. For this purpose we use a program that has the following pseudo-code:

```
shared A[N]
repeat {
        proc 0 writes into A[] with stride S
        wait_barrier()
        proc 1 reads from A[] with stride S
        wait_barrier()
}
```

This program is run on two processors and the outer loop is repeated many times. This program simulates the case when one processor *produces* data and another processor *consumes* it. For an N element array with stride S, in each iteration Processor 0 does N/S write accesses and Processor 1 does N/S read accesses. N is selected such that the array fits in the processor data cache. The time spent in doing these accesses is measured for the two processors and divided by the number of accesses to get the average access time. The time of Processor 0 is the Write-after-read (WAR) access time, shown in Figure 9. The time of Processor 1 is the Read-after-write (RAW) access time, shown in Figure 10.

When Processor 1 writes into the array instead of reading, we get a third access pattern with Write-after-write (WAW) access time, shown in Figure 11. This is a less common access pattern,

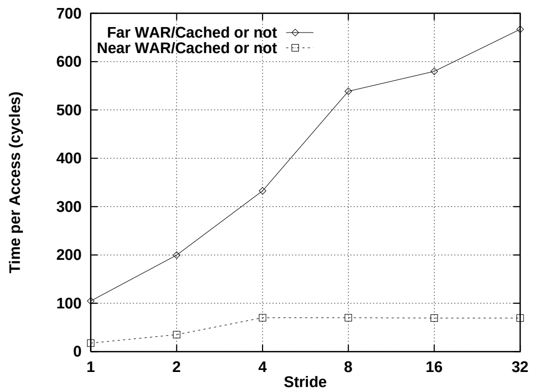

Figure 9: Write-after-read access time.

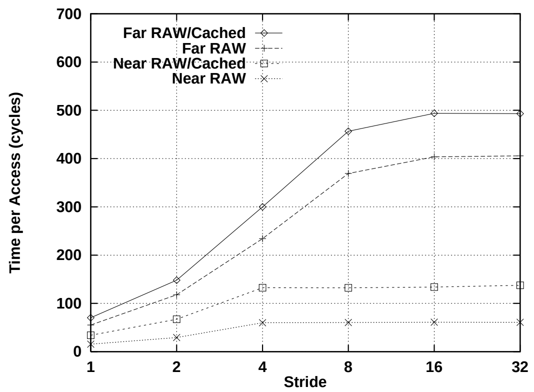

Figure 10: Read-after-write access time.

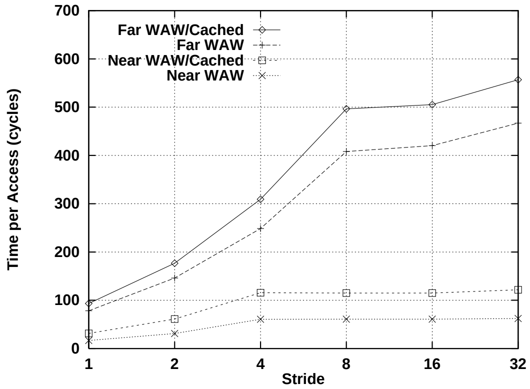

Figure 11: Write-after-write access time.

Type	Distance	Cached	Stride					
			1	2	4	8	16	32
WAR	Near	No	18	35	70	70	70	70
		Yes	18	35	70	70	70	69
	Far	No	105	199	332	539	580	667
		Yes	105	199	333	539	580	667
RAW	Near	No	16	29	60	60	61	61
		Yes	34	67	132	132	134	138
	Far	No	55	118	234	369	404	406
		Yes	70	148	300	457	494	493
WAW	Near	No	17	31	61	61	61	62
		Yes	31	61	116	115	115	122
	Far	No	78	146	249	408	420	467
		Yes	93	177	309	497	505	557

Table 3: Producer-consumer access time in cycles.

but may occur in false-sharing situations where two processors write into two different variables that happen to be located in the same cache line. The three access times are summarized in Table 3.

For the fourth access pattern, Read-after-read (RAR), each processor acquires a copy of the data. Hence we get access times similar to the local-memory access times when the two processors are from the same Hypernode or when the array size fits in the Interconnect Cache. Otherwise, we get access times similar to the load times as reported in Subsection 4.1.

For the first three access patterns, the access time depends on the access stride, the distance between the two processors, and whether the data is cached in the other processor's cache. Since the array fits in the data cache, it is cached whenever a processor accesses it. To measure the not-cached case, we add to the program code to flush the cache just before the barrier. In general, the access time increases as the stride increases due to the increase in the number of misses per reference or the decrease in the number of CTI Rings used. The access time when the two processors are from different Hypernodes (Far) is from 2 to 10 times larger than the access time when the two processors are from the same Hypernode (Near).

Distance	Cached	Latency in microseconds	Transfer Rate in MB/sec
Near	No	1.3	23.5
	Yes	2.0	15.4
Far	No	9.1	5.0
	Yes	10.0	4.6

Table 4: Shared-memory point-to-point communication performance.

In WAR, the access time is higher than the IC store time due to the need to invalidate the copy in the remote Hypernode IC (Far access). It is higher than the local store time due to the need to invalidate the copy in the Hypernode Memory (Near access). This invalidation time is the same regardless of whether the data is in the other processor's cache.

In RAW, a read access is done to the local memory (Near access) or the remote memory (Far access). When the memory has a valid copy, the read access is satisfied from the memory. Otherwise, when the data is invalid, then the current copy is in the cache of the other processor. In the latter case the read access is satisfied from the other processor cache with a higher latency.

The WAW access is similar to the RAW access and starts by reading the current copy with invalidation. Once the data is in the processor's cache, it is updated.

The WAR and RAW access times can be used to find the shared-memory point-to-point communication latency and transfer rate. The latency is the sum of WAR and RAW access times for stride 8. The transfer rate is 8 Bytes divided by the sum of WAR and RAW access times for stride 1. These derived parameters are shown in Table 4. For the cached case, Far communication has about 5 times the Near communication latency with about one third the transfer rate.

4.3 Coherency Overhead

In this subsection we present the evaluation results for the shared-memory performance on the SPP–1000 when 2 or more processors perform read and write accesses to shared data. The main objective here is to evaluate the coherency overhead as a function of the number of processors involved in a shared-memory access. For this purpose we use a program that has the following pseudo-code:

```
shared A[N]
repeat {
        proc 0 writes into A[] with stride S
        wait_barrier()
        foreach proc ≠ 0 {
                begin_critical_section
                     read from A[] with stride S
                end_critical_section
        }
        wait_barrier()
}
```

This program is run on varying number of processors and its loop is repeated many times. This program simulates the case when one processor *produces* data and other processors *consume* it. In

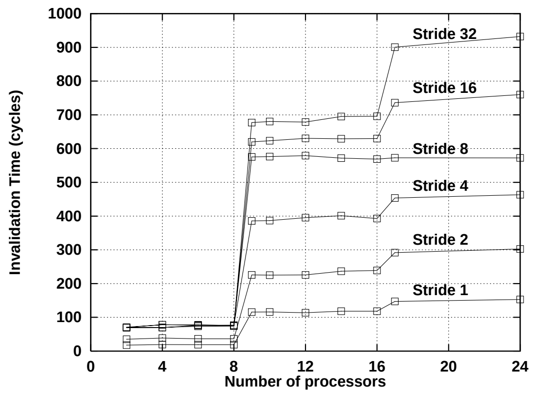

Figure 12: Invalidation time as a function of sharing processors.

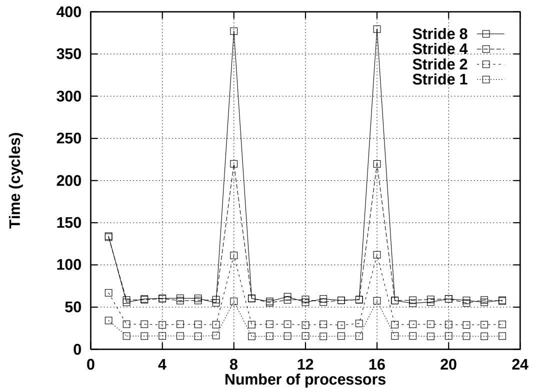

Figure 13: Read time for the n-th processor.

this program Processor 0 does N/S write accesses per iteration, and each of the other processors do N/S read accesses per iteration inside a critical section. Notice that no more than one processor is active in the critical section at any time so the reads are totally ordered. The time spent in doing these accesses is measured for each processor and divided by the number of accesses to get the average access time. The time of Processor 0 is the *Invalidation time*, shown in Figure 12 as a function of the total number of processors. The other processors' time depends on the order in which the processors enter the critical section. Figure 13 shows the read time as a function of the processor read order for experiments with 24 processors.

The Invalidation time increases with increasing stride for the same reasons as described in Subsection 4.1. Invalidation depends on the number of Hypernodes that the processors span, and is the fastest within one Hypernode (8 or fewer processors). In general, the Invalidation time increases in steps, it remains almost constant when the new processor is from the same Hypernode, and increases when the new processor is from a new Hypernode. Invalidation time for stride 8 jumps from 74 cycles for 8 processors of one Hypernode to 575 cycles for 9 processors of two Hypernodes and, opposed to other strides, it does not increase for three Hypernodes.

The first processor to read from the writer's cache causes the memory to get updated as a side effect. The second processor's read time is thus less since it is satisfied from the local-memory. This is also true for processors 3 through 7. The read time is higher for processor 8 since the data is not in its Hypernode and must be provided remotely. When processors 9 through 15 read, they

Type	Stride			
	1	2	4	8
First read	34	67	133	134
Local read	15	29	55	55
Far read	56	111	220	377

Table 5: Read time for shared data in cycles.

find the data in their Hypernode's IC and their read time is similar to processors 2 through 7. This sequence repeats for each Hypernode. Table 5 summarizes these read times.

5 Synchronization Time

In a shared-memory multiprocessor explicit synchronization subroutines are frequently used. A call to a synchronization routine is often needed between code segments that access shared data. When a processor reads shared data that is modified by another processor a synchronization call before the read is needed to ensure that some other processor has completed its update.

The SPP–1000 has several synchronization subroutines, the commonly used subroutines are `WAIT_BARRIER` and `CPS_BARRIER`. We have done experiments to evaluate the overhead of these two subroutines where we measured the time to call a subroutine when all the processors enter the barrier simultaneously. This experiment was implemented by making every processor call the subroutine for many iterations. We have found that the two subroutines have similar performance. Figure 14 shows the average `WAIT_BARRIER` synchronization time for a varying number of processors.

This time shoots up to more than 1500 microseconds for 8 processors implying high contention for the synchronization variables. For 9 or more processors, the processors are spread over two or more Hypernodes with less contention, but the synchronization time increases steadily. This time can be approximated by:

$$T_{\text{sync}}(p) = 7.1p^2$$

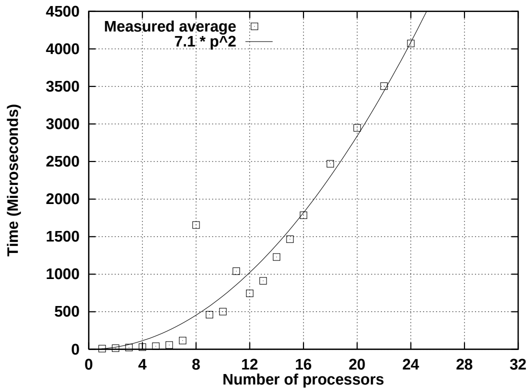

Figure 14: Synchronization time.

Clearly, an inefficient implementation of the barrier was used yielding large synchronization overhead when more than 7 processors are being synchronized.

6 Conclusions

In this paper we have presented an experimental method for systematically measuring the memory and communication performance of a distributed-memory multiprocessor and then modeling it analytically via simple curve fitting. We illustrated this method by carrying out a case study of four aspects of the Convex SPP–1000 performance: scheduling, local-memory, shared-memory, and synchronization.

The scheduling overhead of the SPP–1000 is directly proportional to the number of processors and is relatively small. The parallel-loop scheduling overhead is also proportional to the number of processors and takes 14.3 microseconds to schedule each additional processor. Since the PLSO is in the order of hundreds of microseconds for tens of processors, it might not be rewarding to parallelize a short loop. For a serial loop that takes T_0 microseconds, parallelizing it with perfect load balance gives a parallel loop that takes $T_0/p + \text{PLSO}(p)$ microseconds. Hence a loop can have a faster parallel version if $T_0 > 14.3p/(1 - 1/p)$.

For local-memory access, the SPP-1000 processors depend on their large data caches, that are 1 cycle away for loads, to reduce the cache miss rates. The local-memory bandwidth is only about 50 MB/sec due to the small cache line (32 Bytes), long miss time (55.4 cycles) and narrow memory bus (32 Bits).

Our methodology for characterizing the shared-memory performance of the SPP–1000 reveals that the IC miss time is 410 cycles, i.e. 7.4 times longer than a miss satisfied from the local memory. Each CTI Ring has a peak transfer rate of 600 MB/sec. However the coherence protocol limits the actual load transfer rate between a remote memory and a processor to 15 MB/sec (only 2.5% of the peak). The remote shared-memory point-to-point transfer rate is limited to 5.0 MB/sec in a producer-consumer situation. The big difference between near and far access performance, as presented in this paper, sheds light on the performance gains that can be achieved by localizing the data structures of SPP–1000 applications.

The implementation of the synchronization barrier subroutines for the SPP–1000 is inefficient and better algorithms are available [14].

We suggest that the methodology presented in this paper can be applied to other shared-memory multiprocessors and that the resulting characterization is useful for developing and tuning shared-memory applications and compilers. We have shown that the corresponding characterization of message-passing multicomputer communication performance [10, 15] can also be systematically carried out.

References

[1] K. Hwang, *Advanced computer architecture: parallelism, scalability, programmability*. McGraw-Hill, 1993.

[2] D. Lenoski *et al.*, "The Stanford DASH Multiprocessor," *Computer*, vol. 25, pp. 63–79, Mar. 1992.

[3] J. Kuskin *et al.*, "The Stanford FLASH Multiprocessor," in *International Symposium on Computer Architecture*, pp. 302–313, 1994.

[4] E. Boyd and E. Davidson, "Communication in the KSR1 MPP: performance evaluation using synthetic workload experiments," in *International Conference on Supercomputing*, pp. 166–175, 1994.

[5] T. Agerwala *et al.*, "SP2 system architecture," *IBM Systems Journal*, vol. 34, no. 2, pp. 152–184, 1995.

[6] Convex Computer Corporation, P.O. Box 833851, Richardson, TX 75083-3851, *Convex Exemplar Programming Guide*, x3.0.0.2 ed., June 1994.

[7] "Scalable Coherent Interface (SCI)." ANSI/IEEE Std 1596, 1992.

[8] A. Geist *et al.*, *PVM 3 User's Guide and Reference Manual*. Oak Ridge National Laboratory, Oak Ridge, Tennessee 37831, Sept. 1994. ORNL/TM-12187.

[9] E. Anderson *et al.*, "MPI: a message-passing interface standard," Technical Report, Message Passing Interface Forum, May 1994.

[10] G. Abandah and E. Davidson, "Modeling the communication performance of the IBM SP2," in *10th International Parallel Processing symposium (IPPS'96)*, (Honolulu, Hawaii), April 1996.

[11] K. Gallivan, D. Gannon, W. Jalby, A. Malony, and H. Wijshoff, "Experimentally characterizing the behavior of multiprocessor memory systems: A case study," *IEEE Trans. Software Engineering*, vol. 16, pp. 216–223, Feb. 1990.

[12] W. H. Mangione-Smith, T. P. Shih, S. G. Abraham, and E. S. Davidson, "Approaching a machine-application bound in delivered performance on scientific code," *IEEE Proceedings*, vol. 81, pp. 1166–1178, Aug. 1993.

[13] R. Saavedra, R. Gaines, and M. Carlton, "Micro benchmark analysis of the KSR1," in *Supercomputing*, pp. 202–213, 1993.

[14] J. Mellor-Crummey and M. Scott, "Algorithms for scalable synchronization on shared-memory multiprocessors," *ACM Transactions on Computer Systems*, vol. 9, no. 1, pp. 21–65, 1991.

[15] G. A. Abandah and E. S. Davidson, "Modeling computation and communication performance of the IBM SP2," Technical Report CSE-TR-258-95, University of Michigan, Rm. 3402, 1301 Beal Ave., Ann Arbor, MI 48109, May 1995.