Assessing Programming Costs of Explicit Memory Localization on a Large Scale Shared Memory Multiprocessor*

SILVIO PICANO[1,†], EUGENE D. BROOKS III[2], AND JOSEPH E. HOAG[2]

[1]*School of Electrical Engineering, Purdue University, West Lafayette, IN 47907*

[2]*Massively Parallel Computing Initiative (MPCI), Lawrence Livermore National Laboratory (LLNL), Livermore, CA 94550*

ABSTRACT

We present detailed experimental work involving a commercially available large scale shared memory multiple instruction stream-multiple data stream (MIMD) parallel computer having a software controlled cache coherence mechanism. To make effective use of such an architecture, the programmer is responsible for designing the program's structure to match the underlying multiprocessor's capabilities. We describe the techniques used to exploit our multiprocessor (the BBN TC2000) on a network simulation program, showing the resulting performance gains and the associated programming costs. We show that an efficient implementation relies heavily on the user's ability to explicitly manage the memory system. © 1992 by John Wiley & Sons, Inc.

1 INTRODUCTION

1.1 Overview

Large scale general purpose machines, such as the BBN TC2000 [1] and the nCUBE [2] hypercube machines, are achieving wide performance advantages (and cost advantages) over the traditional mainframe technology. Yet these new parallel computers come with their own unique design and implementation problems with regards to shared and nonshared memory paradigms, [3] scalability, and cache coherence mechanisms [4–6].

In this paper, we present experimental results of a network simulation program mapped onto the BBN TC2000 multiprocessor. The program development is divided into phases, where each phase attempts to exploit a feature of the machine's architecture. Performance measurements are made at each phase of experimentation. Mapping the network simulator efficiently onto the BBN multiprocessor requires considerable modification because it is the programmer's responsibility to explicitly manage data locality.

1.2 Motivation

Detailed mappings of real-world application programs onto specific parallel machines often uncover the most desirable features and also the critical weaknesses inherent in the computer's design. The network simulator program is one of

* Work performed under the auspices of the U.S. Department of Energy by the Lawrence Livermore National Laboratory under contract no. W-7405-ENG-48.

† Author performed work during a summer visit at LLNL and at Purdue University.

Received February 1992.
Revised April 1992.

© 1992 by John Wiley & Sons, Inc.

Scientific Programming, Vol. 1, pp. 67–78 (1992)
CCC 1058-9244/92/010067-12$04.00

these applications, and by examining the resulting performance gains and associated programming costs, we make progress in the development of parallel processing.

We perform this study in phases to determine the software costs and the resulting performance gains achieved by exploiting specific architectural features of the BBN TC2000. The program modifications begin at a general level and progressively become more architecturally dependent. This study revolves about the effective use of the memory hierarchy, including the programmer's maintenance of the caches to avoid coherence problems.

We organize the remaining sections of this paper as follows. In Section 2, we describe past work on the network simulator and give an overview of the BBN multiprocessor used by the Massively Parallel Computing Initiative (MPCI) at Lawrence Livermore National Laboratory (LLNL). We introduce the network simulator algorithm in Section 3 and present the parallel implementations and performance results in Section 4. In Section 5, we discuss the software costs associated with overcoming the lack of hardware support for coherent shared memory caches. In Section 6, we give a summary and discuss open issues.

2 BACKGROUND

2.1 Recent and Past Work

The network simulator program [7] determines the performance of a scalable multiprocessor in a vector processing environment. It is also a vital component of the Cerberus multiprocessor simulator [8], which performs simulation of parallel programs at the instruction level. The network simulator also executes on Sequent Symmetry [9] and Alliant FX/8 [10] multiprocessors.

Programmers of the above multiprocessors often regard these machines as easy to program because of the benefits of (1) using a single, shared address space, and (2) having hardware support for coherent caches. We, however, are interested in large scale systems on the order of hundreds or thousands of processors, where cache coherent shared memory systems currently do not exist.

2.2 BBN TC2000 Multiprocessor

The TC2000 multiprocessor [1] is a commercial parallel processor designed by BBN Advanced Computers Inc. It is a scalable shared memory, *multiple instruction stream-multiple data stream* (MIMD) multiprocessor having up to 512 high performance processing elements (PEs) and memory modules. Processors and memories communicate through a variant of a multistage cube network [11] in a *PE-to-PE* model.*

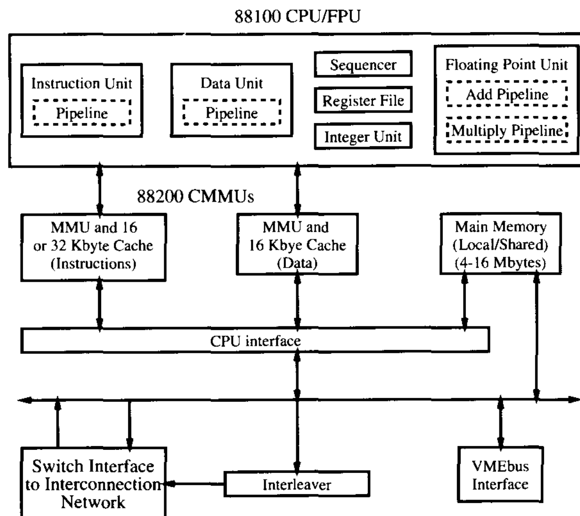


FIGURE 1 BBN TC2000 processing element.

Each PE (Figure 1) consists of a processor, memory, memory management units, and a network switch interface. The processor is a Motorola 88100 *reduced instruction set computer* (RISC) operating at 20 MHz [12]. The 88100 combines the *central processing unit* (CPU) and the *floating point unit* (FPU) on a single integrated circuit. Shared memory per PE consists of from 4 to 16 megabytes and has local cache memories.† The Motorola 88200 *cache memory management unit* (CMMU) [13] facilitates both virtual memory management and cache memory support. Separate instruction and data caches help reduce the access times required for local and remote memory references. The cache organization is a 16 kilobyte, four-way set associative design,‡ and each PE has two instruction caches and one data cache. There is no hardware mechanism for

* The "PE-to-PE" model specifies that a cpu port and a memory port share a common port into the multistage network.

† The LLNL MPCI's TC2000 has 128 PEs, each having 16 megabytes of shared memory.

‡ A four-way set associative design allows memory references to be placed in one of four dedicated locations in the cache.

Table 1. Hierarchical Memory Access Times (Microseconds)

Cache		Memory Response Delays			
Mode	Activity	Local Read	Local Write	Remote Read	Remote Write
Inhibit	None	0.550	0.600	1.913	1.889
Write-through	Hit	0.150	0.600	0.150	1.889
Copyback	Hit	1.150	0.150	0.150	1.150
Write-through	Miss	0.850	1.200	2.529	4.168
Copyback	Miss with no writeback	0.850	1.200	2.529	4.168
	Miss with local write	1.500	1.850	3.179	4.818
	Miss with remote write	2.905	3.255	4.534	6.173

maintaining data cache coherence for shared memory. The application programmer must explicitly control the data caches to ensure correct program operation. Similar to the IBM RP3 memory design concept [14], regions of shared memory are marked as either cacheable or noncacheable. However, the memory design in the BBN machine can further specify each cacheable region as having either copyback or write-through operation. Copyback caching mode is a memory update policy that allows memory updates to occur directly to cache memory, and not necessarily to main memory at the same points in time. Write-through caching mode always enforces a strict update policy such that cache memory and main memory are always consistent. In Table 1 [1], we show the access times for the TC2000 memory hierarchy. The large difference in access times between cache memory references (0.150 microseconds) and shared memory references (1.913 microseconds) causes severe performance degradation if a relatively large fraction of dynamic instructions reference shared memory.

Each PE has a bidirectional switch interface to the network responsible for satisfying remote memory requests. The Motorola 88200 CMMU on each PE is responsible for performing the virtual to physical address translation. Specific portions of the physical address determine the access route to either local memory (and local caches) or to the switch interface (remote memory). The interconnection network design is based on a multistage cube variant called the *butterfly switch.* This network has bidirectional paths and supports circuit switched transmissions to memory modules on other PEs. Individual switch nodes are composed of 8 × 8 crossbar modules, having 8 bit wide data paths clocked at 38 MHz.

3 THE NETWORK SIMULATOR

3.1 Preliminaries

The network simulator program[7] simulates and computes performance statistics of a proposed communications network for a scalable multiprocessor. Simulated CPUs interface to this network and communicate with other CPUs via packet requests and responses. The network organization is based on the multistage cube network, which has equidistant paths to other resources.

The multistage cube interconnection networks provide the communication vehicles for any number of simulated cpu and memory ports. Specifically, the cpu and memory ports are connected to a request network per the *dance hall* model.* A separate response network returns the words of memory to the requesting cpu port (Figure 2). Furthermore, the networks are packet switched, allowing packets to be buffered in the presence of network conflicts. The design of the individual switch nodes in the network provides efficient packet switched communications. The design of a κ-input port, κ-output port switch node uses κ^2 buffers to perform necessary storage and also has control logic to perform packet routing. In Figure 3, we show examples of simplified switch node designs.

The proposed network and switch node designs achieve virtually ideal performance in a vector processing environment [7]. Briefly, each simulated cpu port will fetch a number of memory words from consecutive memory ports (a vector fetch). For example, if we simulate a 2,048 word

* The "dance hall" model specifies that the cpu ports and the memory ports lie at opposite sides of a multistage network.

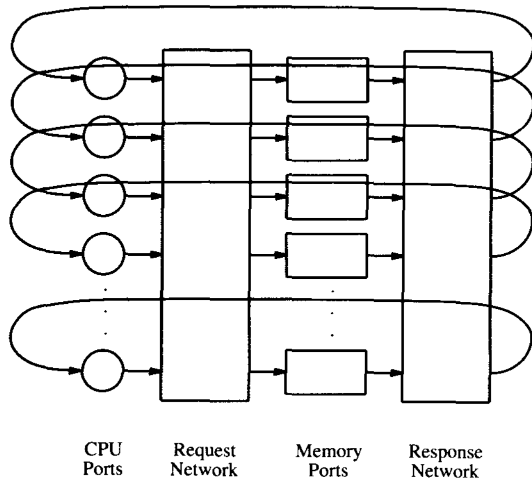


FIGURE 2 Network simulator architecture.

vector fetch in a system using 2-ary 10-cube networks, we are effectively simulating a 1,024 cpu system that uses 10-stage networks having 2 × 2 switch nodes (2^{10} = 1,024), where each cpu performs 2,048 memory fetches. These memory fetches begin at a random memory port and cycle consecutively through the memory ports in a modulo fashion. For example, simulated cpu (i) would fetch its first word from memory port (j), its second fetch from memory port (j + 1) mod 1024, its third fetch from memory port (j + 2) mod 1024, etc. until all the fetches from all the memories are received. Vector stride refers to the incrementing value between consecutive, regular references. In the above example, the vector stride is one element.

The network simulator program input parameters include the following:

1. The vector length
2. The vector stride
3. The buffer length in a switch node
4. The order (number of stages in the networks)
5. The base (number of CPUs wired to one switch node)

The network simulator program output statistics include the following:

1. The average bandwidth
2. The average packet latency
3. The number of simulation cycles
4. The standard deviation among the individual finish times

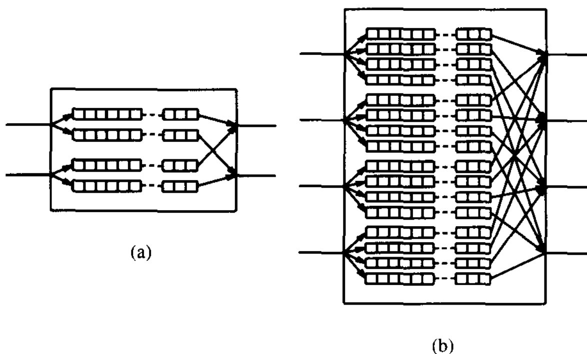


FIGURE 3 Examples of packet switched, node designs: (a) 2 × 2 design, (b) 4 × 4 design.

3.2 The Algorithm

The network simulator is a *logic* type of program, performing a large number of boolean operations to submit, forward, and receive packets, resolve conflicts, and compute packet statistics. In the phases of interest, the program executes no floating point instructions and performs only a small amount of work per packet move operation. The resulting performance is strongly dependent on data placement, movement, and allocation strategies.

The simulator program consists of four major phases on each simulation clock cycle: (1) a cpu, memory set phase, (2) a network set phase, (3) a network move phase, and (4) a cpu, memory move phase. The main loop of the network simulator is shown below:

```
while (any more packets to move) {
        cpu-memory_set()
        network_set()
        barrier_synchronization()
        network_move()
        cpu-memory_move()
        barrier_synchronization()
        serial_code()
        barrier_synchronization()
}
```

The cpu, memory set phase sets and resets boolean flags depending on the following states:

1. cpu ports are able to submit packets into the request network
2. Memory requests have propagated back to cpu ports
3. Memory response ports can accept packets

The network set phase sets and resets boolean flags depending on the following states:

1. cpu ports in the request network can advance packets and there is available buffer space on the corresponding switch node input ports
2. Memory ports in the response network can advance packets and there is available buffer space on the corresponding switch node input ports
3. Switch nodes can advance packets and there is available buffer space on the corresponding switch node, cpu, or memory input ports
4. Determine packet *winners* when conflicts arise

The network move phase performs the following actions based on the flag states determined in the first two phases:

1. Packets move from cpu ports into the corresponding switch nodes in the request network
2. Packets move from switch node output ports into the corresponding switch node or memory input ports in the request network
3. Packets move from memory ports into the corresponding switch nodes in the response network
4. Packets move from switch node output ports into the corresponding switch node or cpu input ports in the response network

The cpu, memory move phase performs the following actions based on the flag states determined in the first two phases:

1. Advance packets in memory request ports to the memory response ports
2. Advance packets in cpu response ports, count packets received, and record network latencies
3. Initialize information for packets entering the network and increment the counters of packets sent

Barrier synchronization primitives force all processors to wait for all other processors at points specified in the program. After all processors reach such synchronization points, they continue executing useful instructions. Barrier synchronization is necessary in this program to ensure the proper ordering of memory accesses among the asynchronous processors to maintain program correctness.

4 PARALLEL IMPLEMENTATIONS

4.1 Phase 0 Parallel Program

The above algorithm decomposition represents the original serial implementation and an immediate parallel implementation is easy to see: given a multiprocessor with N PEs, PE i would be responsible for maintaining and computing information on the simulated cpu ports $[i, i + N, i + 2N, \ldots]$, memory ports $[i, i + N, i + 2N, \ldots]$, and switch nodes $[(i,j), (i + N,j), (i + 2N,j), \ldots]$ for $0 \leq j < order$ (Figure 4).

The original parallel program implementation mentioned above is the starting point for measuring performance of the BBN TC2000 on the network simulator program. Because of the lack of hardware support for coherent caches, this implementation is unable to make effective use of the local data caches in the TC2000. This program uses interleaved shared memory for all shared data accesses. Interleaved shared memory design is a memory mapping technique that automatically places consecutive memory references (or groups of references) in successive memory modules that require different communication paths. Interleaving data structures throughout shared memory allows multiple PE access to different elements of the data structure without incurring unnecessary network conflict and delay. In Figure 5, we show parallel and serial execution time results.

Recall that the vector length corresponds to the

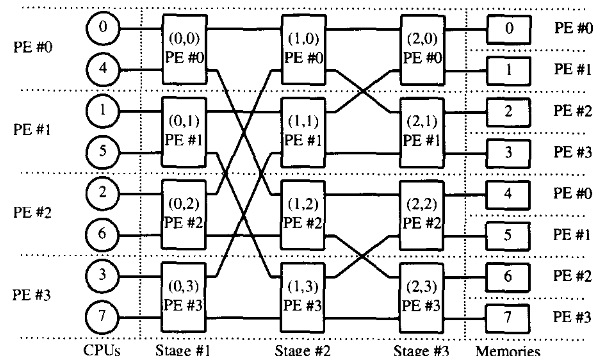


FIGURE 4 Phase 0 parallel program decomposition of a 2-ary, 3-cube network simulator using four TC2000 PEs.

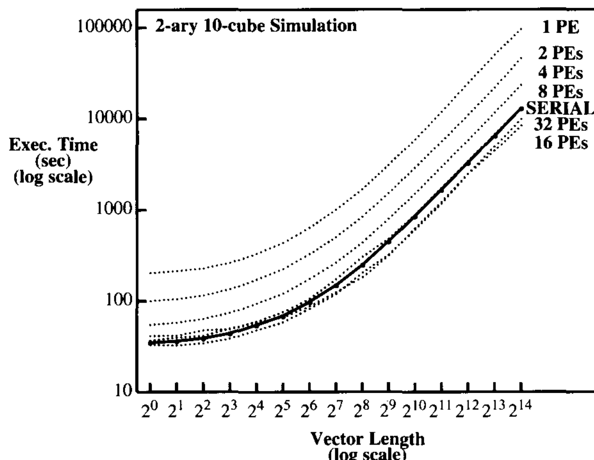


FIGURE 5 Phase 0 execution time requirements.

number of memory words fetched by each simulated cpu. The 2-ary 10-cube network parameters specify the simulation of a 10-stage network using 2×2 switch nodes (N = number of PEs = 2^{10}). The serial program is the parallel program compiled in a manner that strips all synchronization operations and uses only local memory with copy-back cache mode enabled. (The performance of the serial program is marked "SERIAL" in Figure 5.) In Figure 5, we show that eight processors, working in parallel, are required to match the serial program requirements. While the serial program simply uses local and cache memory accesses, the parallel program uses interleaved shared memory accesses. Note that the 32-processor results actually perform worse than the 16-processor results, possibly due to (1) increasing initialization time, (2) increasing communication time with respect to computation time, and (3) increasing amounts of network conflicts. In particular, *read-only hot spot* [15] accesses are potentially significant because severe network blockage and excessive communication delays result. "Hot spot" accesses are memory references made to a single shared location by several processors at similar points in time.

This parallel program also executes on the Alliant FX/8 and Sequent Symmetry multiprocessors, obtaining excellent speedup results [16]. Recall that these multiprocessors have hardware support for cache coherence. For the execution of large scale problems, however, the above systems lack the required performance. The BBN TC2000 promises a scalable solution to finding that performance. However, the BBN machine lacks hardware support for coherent shared memory caches. We need to recover from the performance penalty (Figure 5) resulting from the dependence of the Phase 0 implementation on coherent cache support.

4.2 Phase 1 Parallel Program

Our second attempt at a parallel implementation makes use of the data caches whenever possible. From Table 1, cache memory hit references are about 4 times faster than local memory references (150 vs. 600 nsec) and about 13 times faster than remote memory references (150 vs. 1,889 nsec). There are two ways to maintain cache coherence on the TC2000. One method is to cache all references and explicitly flush them when another PE requires access to the cached data. The alternative method is to find data references that only a single PE will access while the remaining shared data references are simply interleaved throughout the memory system.

We performed separate, simple experiments to determine which of the above two methods would be better for our application. Explicit cache flushes require 11 microseconds per cache line using a BBN-specific routine. This coherence scheme was not used because we determined that the 11 microseconds and the additional software required would be intolerable for our application. (Recall that our application performs very small quantities of work between communication operations.) Using this technique, cache flushing could dominate execution time due to its high overhead. Therefore, we chose to explicitly and carefully allocate all data that was not shared as local cacheable memory, restructuring the program to make use of the cacheable regions as much as possible. We also manually identified and removed write-once, read-many hot spot accesses by shadowing this data with copies in each PE. In Figure 6, we show a sample parallel decomposition for the request network (the response network is done similarly).

The decomposition shown in Figure 6 differs slightly from the one shown in Figure 4. We altered the program to handle the switch node, memory module interface more efficiently. In Figure 6, we see that the last stage of the network and the corresponding memory modules are handled by the same physical processor. This one processor is sharing some* data with itself so no coher-

* Part of the memory port data structure can now be safely cached, while other parts must still reside in shared memory for correct operation.

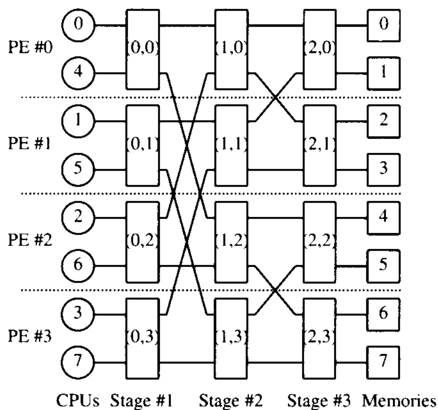


FIGURE 6 Phase 1 parallel program decomposition of a 2-ary, 3-cube network simulator using four TC2000 PEs.

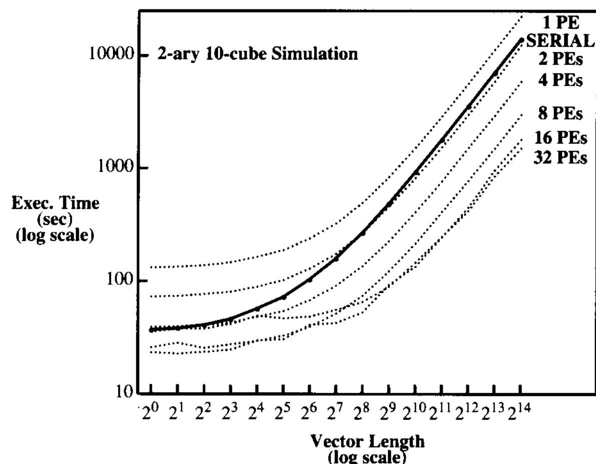


FIGURE 7 Phase 1 execution time requirements.

ence problems will occur. Therefore, we can safely cache part of these data structures used in this part of the network simulator program. In Section 4.3, we find more examples of data sharing having similar behavior as above.

The resulting program has a message passing flavor. In the original version of the program, cpu, memory, and switch node structures explicitly need information from other structures in the next stage (buffer space availability and packet buffer access) for correct operation. For example (see Figure 6): before a packet can propagate on the link between switch node (0, 0) and switch node (1, 2), switch node (0, 0) must be certain that buffer space is available in switch node (1, 2). Because these switch node structures reside on different physical PEs, they cannot be cached and must reside in shared memory. The new version of the program extracts the shared data necessary for interprocess communications from each cpu, memory, and switch node structure so that we can allocate these structures in local cache memory. This removed shared data is left in shared memory so that inter-PE communication can proceed, maintaining program correctness. We use simple *static scheduling** methods in the program because the individual instruction streams are similar in execution time. The static scheduling scheme also forces a PE to control the same data structures on each iteration to prevent extraneous and expensive data movement. We show parallel and serial execution time results in Figure 7 and show speedup results in Figures 8 and 9.

We see an impressive improvement in parallel execution time with respect to Phase 0 execution times (Figure 5). The large effort in restructuring the program also resulted in marginal improvements to the serial execution time, thereby producing unambiguous speedup results. At larger vector lengths, two parallel processors now surpass the serial execution time in contrast to the eight processors required in Phase 0.

For ideal speedup behavior, the curves in Figures 8 and 9 should shadow the horizontal lines for the appropriate number of PEs used. Asymptotically, we see good performance because the relatively large amount of initialization time (memory allocation and initialization, cache transients, and network transients), when compared to actual processing time, becomes less signifi-

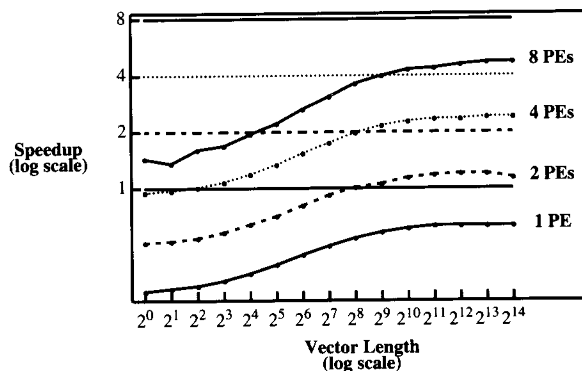


FIGURE 8 Phase 1 speedup results (one-, two-, four-, and eight-processor results).

* We use the term "static scheduling" to refer to a fixed decomposition of the work load, independent of any execution time parameters.

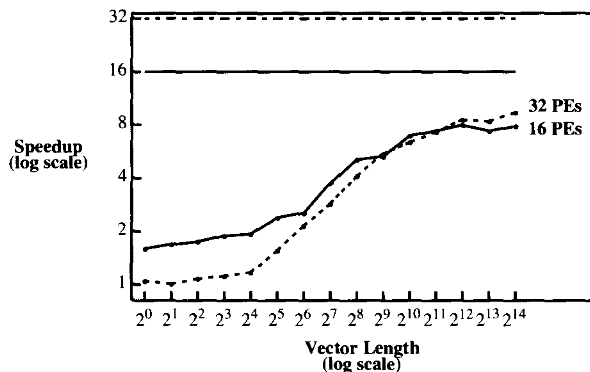


FIGURE 9 Phase 1 speedup results (16- and 32-processor results).

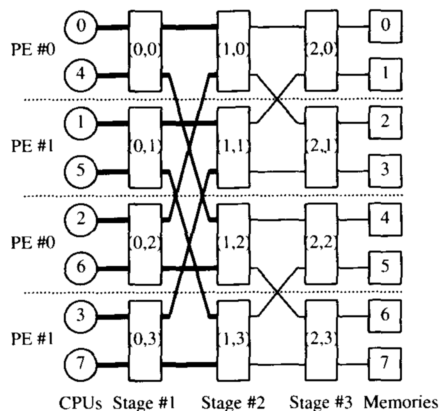


FIGURE 10 A 2-ary 3-cube decomposition with two processors.

cant. From the details of the network simulation program, we note that initialization time is a constant term, independent of the vector length parameter, for a given processor count. Because initialization time is a constant as the problem size increases (as the vector length increases), longer computation time amortizes the effects of initialization time. In Figure 7, we can see that at short vector lengths (less than 2^6), small increases in the vector length result in negligible increases in execution time. However, at vector lengths beyond 2^8, execution time becomes a linear function of the vector length, resulting in amortized initialization time with respect to large computation time. Shared memory reference delay always degrades performance on such a multiprocessor. In the next phase, we deal with identification and elimination of unnecessary shared memory references.

4.3 Phase 2 Parallel Program

In Phase 1, we use shared memory locations to handle the data communication occurring on the links between the cpu ports, memory ports, and the switch nodes, ignoring the actual location of these data structures. This point of view is the *data parallel* one, programming as if each port or switch node data structure needs its own *virtual* processor. In Phase 2, we determine if communication occurs between cpu, memory, and switch node ports represented by the same physical processor and arrange that local cache memory be used for this purpose.

For example, suppose we are simulating a 2-ary 3-cube network. In Figure 10, one can see the decomposition of the request network using two TC2000 PEs. When moving a packet or checking buffer space availability for cpus (0), (4) with switch nodes (0, 0), (1, 0), and (1, 2), we notice that only physical PE#0 accesses the necessary data. (PE#1 operates in a similar manner with a different subset of structures in the same stages of the network model.) When moving packets or checking buffer space availability in the cpu ports and in network stage 1, the PE communicates with itself using shared memory. Also, when moving data between network stages 2 and 3, and the memory ports, we must use shared data references because different PEs are responsible for moving packets and checking buffer space availability.* As the network order grows for a fixed number of physical processors applied, more consecutive stages of the network will be able to exploit local cache memory in this manner. Blind use of the data parallel model (not taking into account the actual physical processor and the data it contains) leads to extraneous use of shared memory when a processor communicates with itself.

In Phase 2, we remove much of the extraneous use of shared memory by shadowing the shared arrays with arrays allocated in local cache memory. When we operate in a stage of the network simulator program that would normally suffer from extraneous sharing of data, we simply access local cache memory data because we know no other PE will ever access this data. When we operate in a region of the network that requires sharing of data, we access shared data because we know other PEs will access this data. We show the parallel and serial execution time results in Figure 11

* Although the memory ports for this example appear to be cacheable, the decomposition of the corresponding response network (not shown) prohibits it.

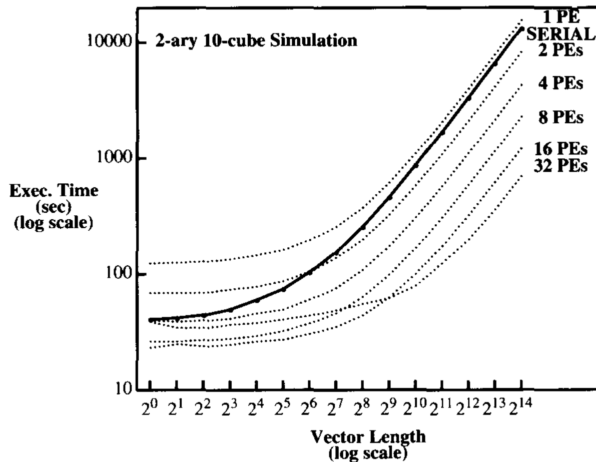


FIGURE 11 Phase 2 execution time requirements.

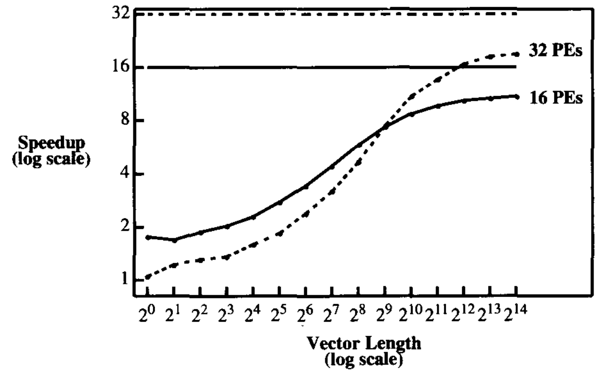


FIGURE 13 Phase 2 speedup results (16- and 32-processor results).

and show the speedup results in Figures 12 and 13.

To eliminate extraneous use of shared data in the network simulator program, we perform a simple test at the stage of the network from which we are attempting to move packets. Factors including the order of the network, the number of inputs to a switch node, and the number of physical PEs determine the stages where extraneous sharing occurs. Note that a shared memory multiprocessor possessing hardware enforced cache coherence does not have this extraneous shared data problem because this data is always cached and never invalidated by another PE.

Through the elimination of extraneous use of shared memory, execution time results show significant improvements. The greatest benefits of this phase occur when the simulated network is large with respect to number of parallel processors employed because more stages of the network will be able to use local cache arrays instead of shared arrays. In particular, note the significant performance gains using 32 processors on the longer vector length problems in the 2-ary 10-cube simulations (compare Figure 13 to Figure 9). Note that for vector lengths of less than 2^9, 16-processor results are consistently better than 32-processor results because we are dealing with relatively small execution times on the order of 20 seconds. As noted earlier, run time is not sufficient in such cases to amortize initialization time on these relatively small problem sizes.

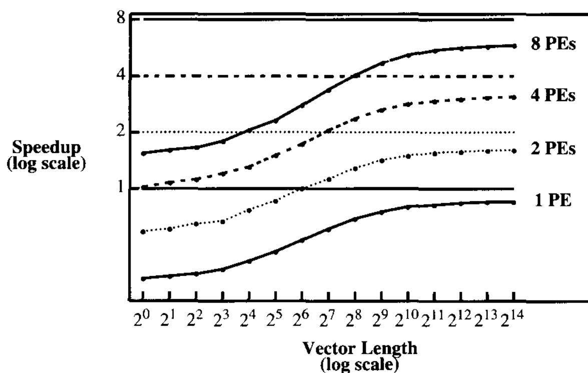


FIGURE 12 Phase 2 speedup results (one-, two-, four-, and eight-processor results).

4.4 Phase 3 Parallel Program

In the previous phases, we blindly interleave shared data references throughout the system. For example, when a packet advances through the network, the program transfers a pointer to a packet structure in shared memory between communicating structures. The program performs this packet transfer in two steps (on the same simulation clock cycle): (1) the cpu port, memory port, or switch node that contains the packet writes this pointer into interleaved shared memory, then (2) the communicating neighbor simply reads this pointer from shared memory, thereby simulating the moving of the packet. This method requires two remote shared references: one for the write operation and one for the read operation. For example (refer to Figure 6), when a packet propagates from switch node (0, 0) (on physical PE#0) to switch node (1, 2) (on physical PE#2), the operation requires two remote shared references: switch node (0, 0) writes the packet pointer to an

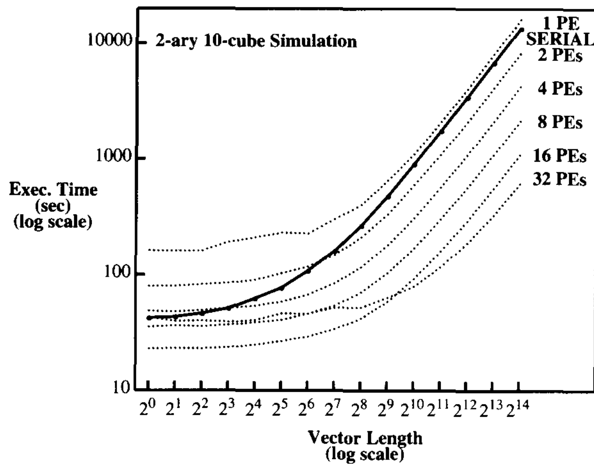


FIGURE 14 Phase 3 execution time requirements.

interleaved shared memory location while switch node (1, 2) reads this packet pointer from this location. The BBN TC2000 supports explicit placement of shared data on specific PEs. By mapping shared data arrays favorably, one can arrange for faster accesses.

In Phase 3, we take advantage of this BBN-specific architectural feature by replacing the remote read operation (1,913 nsec, Table 1) with a local read operation (550 nsec). We accomplish this replacement by having the writer explicitly place a packet pointer into the local (shared, non-cached) memory of the PE which will read it. Each packet movement operation now requires approximately 2,439 nsec (550 local read + 1,889 remote write) instead of 3,802 nsec (1,889 remote write + 1,913 remote read). The program uses this mapping scheme for the packet pointer movement routines only, however, this scheme could also be applied to the buffer space availability

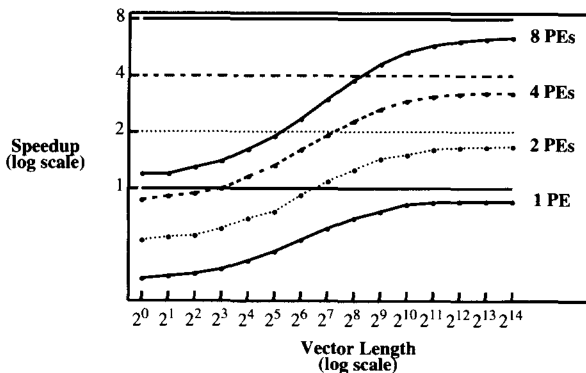


FIGURE 15 Phase 3 speedup results (one-, two-, four-, and eight-processor results).

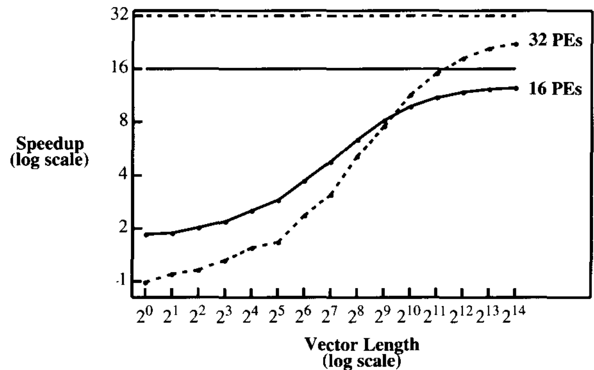


FIGURE 16 Phase 3 speedup results (16- and 32-processor results).

variables if the resulting gains are significant. We show parallel and serial execution time results in Figure 14 and show speedup results in Figures 15 and 16.

These speedup results show marginal performance increases with respect to the previous phases. Once again, the serial execution time did not increase, thereby yielding unambiguous speedup results. The programming effort to perform this faster packet pointer movement is onerous. We perform explicit allocation and initialization of several global and local tables to facilitate the additional translations of the packet pointer's destination. We expect similar (marginal) gains if the same idea is applied to the buffer space availability variables.

5 PROGRAMMING COSTS

As one measure of programming complexity, we document the total number of actual program lines necessary in the original program and its successive modifications. In Table 2, we show each program phase of interest with its respective number of program lines. The "% Incremental In-

Table 2. Programming Costs Summary

Program	Number of Lines	% Incremental Increase	% Total Increase
Serial	1,451	—	—
Phase 0	1,501	3.4	3.4
Phase 1	1,814	20.8	25.0
Phase 2	2,224	22.6	53.3
Phase 3	2,327	4.6	60.4

crease" entries indicate the series of consecutive additions to the program. For example, the "Phase 3" costs with respect to "Phase 2" costs resulted in a 4.6% incremental increase. The "% Total Increase" entries indicate the increase with respect to the serial program. For example, the "Phase 3" costs with respect to "Serial" costs resulted in a 60.4% total increase.

The costs associated with the original parallelization of the program resulted in only a 3.4% increase in program size. To make effective use of the TC2000 multiprocessor, Phase 1 and 2 program modifications added significant amounts of code to extract the desired performance. The development, debugging, and performance evaluation of Phase 1, 2, and 3 programs required approximately 4 man-months to complete. The availability of the original code developer and the advanced parallel programming tools significantly accelerated this development work.

The resulting source programs are difficult to understand if the reader has little knowledge of the BBN TC2000 multiprocessor. The code modifications are necessary to explicitly place data in the multiprocessor's memory hierarchy to provide the fastest access possible. Furthermore, the successive program modifications have left the program somewhat dependent on the multiprocessor. The Phase 0 parallel program is applicable to all shared memory multiprocessors, while the Phase 1 and 2 programs are applicable only to those architectures supporting a shared memory and a fast local memory. The Phase 3 parallel program is restricted to architectures that support explicit placement of shared data. The availability of a hardware enforced coherent cache system would free the programmer from the coding costs required in Phases 1 and 2, and at the same time, would maintain program portability. The enhancements performed in Phase 3 would not be done automatically, however, such enhancements would also improve this system's performance.

It is difficult to ascertain the programming costs that would be incurred during the development of significantly large production codes (100,000 line programs). From this experiment, we note that a significant amount of work was done to identify and initialize data regions with the proper protection (for example, enabling selective caching and manually identifying write-once, read-many hot spot references in Phase 1). We duplicated several data structures in an attempt to separate private and shared regions, so that faster access can be obtained when necessary (adding private and shared variables in Phase 2). In addition, we separated some routines to facilitate locality and synchronization when accessing data structures (done in Phase 2). Although the "% Total Increase" numbers shown in Table 2 may not hold true for production codes, we believe such codes would have to undergo similar analysis stages as we have done on the network simulator program to extract the desired performance level.

While this paper analyzes only one specialized application mapping on the TC2000, interested readers are also referred to an annual report[17] detailing similar work. This reference contains the MPCI's executive summary, TC2000 parallel programming support and scheduling, and several application mappings and resulting performance on the TC2000 and other computer systems.

6 SUMMARY AND OPEN ISSUES

The network simulation program makes critical demands on parallel computers because of the relatively high communication requirements it poses. The resulting performance depends heavily on data placement, movement, and allocation strategies. We have demonstrated effective mapping techniques to extract high performance from a commercially available large scale shared memory multiprocessor. Using incremental program modifications, we obtain significant performance gains through the effective use of the memory hierarchy, eventually increasing program size by 60%. When there is more potential performance to be had, it is reasonable to expect programmers to work harder and longer to attain it. However, we believe that the above costs are overly unyielding and arduous for prospective users.

At this point, however, it is unclear exactly how close the network simulator parallel program can approach ideal speedup even if all unnecessary shared references are eliminated. This question could easily be answered if we had at our disposal a scalable cache coherent multiprocessor. It is also unclear how much better a scalable cache coherent system would perform on this problem (currently under study). Certainly the programmer would be relieved of the burden of identifying data structure allocation to cacheable memory segments. Quantifying the degradations due to shared memory latencies and the increased network traffic for automatic cache coherence maintenance is still an open question.

Whether or not we can build a scalable, hard-

ware enforced coherent cache multiprocessor is still an open question. Although no such systems are currently offered by vendors, considerable progress on the design and performance of such systems has been made in the research community [5, 18, 19]. We are also investigating more aggressive CPU designs to tolerate long memory access delays based on multithreaded and von Neumann dataflow hybrid processors.[20–23]

The authors wish to thank Brent Gorda, Tammy Welcome, and Linda Woods of the MPCI at LLNL and Ken Sedgwick for their assistance with the parallel programming support for the BBN multiprocessor. We also thank the anonymous referees for their helpful criticisms and suggestions on this paper.

REFERENCES

[1] BBN Advanced Computers Inc., *Inside the TC2000.* Cambridge, MA, 1989.

[2] nCUBE Corp., *nCUBE 2 Processor Manual,* PN 101636, r. 2.0 edition, Beaverton, OR, Dec. 1990.

[3] S. Picano and T. L. Casavant, "An experimental analysis of image correlation on shared vs. non-shared memory MIMD parallel computers," *Proc. of the Int. Conf. on Parallel Processing,* St. Charles, IL: Penn State University Press, 1990, pp. 92–96.

[4] L. M. Censier and P. Feautrier, "A new solution to coherence problems in multicache systems," *IEEE Trans. Comput.,* vol. C-27, no. 12, pp. 1112–1118, Dec. 1978.

[5] E. D. Brooks III and J. E. Hoag, "A scalable coherent cache system with incomplete directory state," *Proc. of the Int. Conf. on Parallel Processing.* St. Charles, IL: Penn State University Press, 1990, pp. 553–554.

[6] C. Warner and D. G. Meyer, "Directory Based Cache Coherence Protocols for Shared Memory Multiprocessors," *Technical Report TR-EE 90-33,* Purdue University, West Lafayette, IN, May 1990.

[7] E. D. Brooks III, "The indirect K-ary N-cube for a vector processing environment," *Parallel Comput.,* vol. 6, no. 3, 1988, pp. 339–348.

[8] E. D. Brooks III, T. S. Axelrod, and G. A. Darmohray, *Parallel Processing for Scientific Computing.* Philadelphia: SIAM, 1989, pp. 384–390.

[9] Sequent Computer Systems, *Sequent Technical Summary,* Beaverton, OR, 1987.

[10] Alliant Computer Systems Corp., *FX/SERIES Architecture Manual,* PN 300-00001-B edition, Jan. 1986.

[11] H. J. Siegel, *Interconnection Networks for Large-Scale Parallel Processing: Theory and Case Studies,* 2nd ed. New York: McGraw-Hill, 1990.

[12] Motorola, *MC88100 RISC Microprocessor User's Manual,* 2nd ed. Englewood Cliffs, NJ: Prentice-Hall, 1990.

[13] Motorola, *MC88200 Cache/Memory Management Unit User's Manual,* 2nd ed. Englewood Cliffs, NJ: Prentice-Hall, 1990.

[14] W. C. Brantley, K. P. McAuliffe, and J. Weiss, "RP3 processor-memory element," *Proc. of the Int. Conf. on Parallel Processing.* St. Charles, IL: IEEE Computer Society Press, 1985, pp. 782–789.

[15] G. F. Pfister and V. A. Norton, "Hot Spot" contention and combining in multistage interconnection networks, *Proc. of the Int. Conf. on Parallel Processing.* St. Charles, IL: IEEE Computer Society Press, 1985, pp. 790–797.

[16] E. D. Brooks III, "Effective use of shared memory multiprocessors," *Proc. of the Int. Conf. on Supercomputing.* Boston, MA: International Supercomputing Institute Inc., 1988, pp. 365–371.

[17] E. D. Brooks III and K. H. Warren, "The 1991 MPCI Yearly Report: The Attack of the Killer Micros," *Technical Report UCRL-ID-107022,* Lawrence Livermore National Laboratory, Livermore, CA, Mar. 1991.

[18] A. Gupta, W. Weber, and T. Mowry, "Reducing memory and traffic requirements for scalable directory-based cache coherence schemes," *Proc. of the Int. Conf. on Parallel Processing.* St. Charles, IL: Penn State University Press, 1990, pp. 312–321.

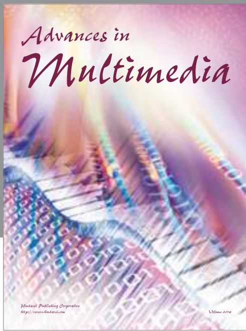
[19] P1596 Working Group, "P1596/Part IIIA- SCI Cache Coherence Overview," *Technical Report 0.33,* IEEE Computer Society, Nov. 1989.

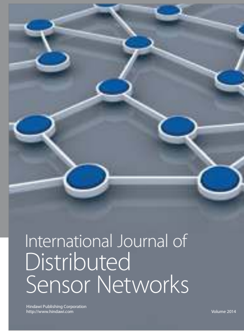
[20] A. Agarwal, B. H. Lim, D. Kranz, and J. Kubiatowicz, "APRIL: A processor architecture for multiprocessing," *Proc. of the Ann. Int. Symp. on Computer Arch.* Seattle, WA: IEEE Computer Society Press, May 1990, pp. 104–114.

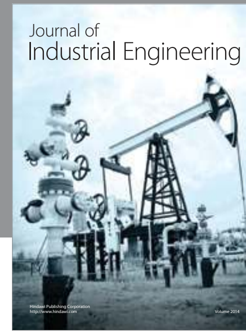
[21] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith, "The TERA computer system," *Proc. of the Int. Conf. on Supercomputing.* New York: ACM, June 1990, pp. 1–6.

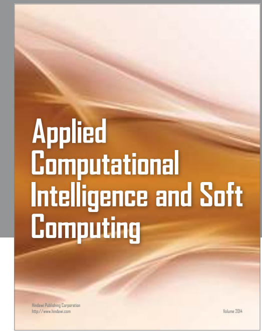
[22] R. Buehrer and K. Ekanadham, "Incorporating data flow ideas into von Neumann processors for parallel execution," *IEEE Trans. Comput.,* vol. C-36, no. 12, pp. 1515–1522, Dec. 1987.

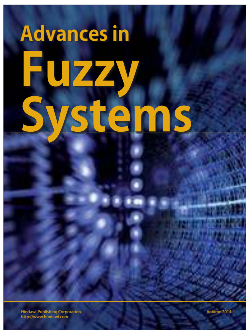
[23] R. S. Nikhil, G. P. Papadopoulos, and Arvind, "*T: A Multithreaded Massively Parallel Architecture," *Technical Report Computation Structures Group* Memo 325-1, MIT, Nov. 1991.











Hindawi

Submit your manuscripts at
http://www.hindawi.com