[7] T.J. LeBlanc. "Shared Memory Versus Message Passing in a Tightly-Coupled Multiprocessor: A Case Study". In *Proceedings of the 1986 International Conference on Parallel Processing*, pages 463–466, August 1986.

[8] Kai Li and Paul Hudak. "Memory Coherence in Shared Virtual Memory Systems". *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.

[9] C. Lin and L. Snyder. "A Comparison of Programming Models for Shared Memory Multiprocessors". In *Proceedings of the 1990 International Conference on Parallel Processing*, pages 163–170, August 1990.

[10] J. M. Mellor-Crummey and M. L. Scott. "Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors". *ACM Transactions on Computer Systems*, 9(1):21–65, 1991.

[11] T. Ngo and L. Snyder. "On the Influence of Programming Models on Shared Memory Computer Performance". In *Scalable High Performance Computing Conference (SHPCC '92)*, Williamsburg, VA, April 1992.

[12] R.H. Thomas and W. Crowther. "The Uniform System: An Approach to Runtime Support for Large Scale Shared Memory Parallel Processors". In *Proceedings of the 1988 International Conference on Parallel Processing*, pages 245–254, August 1988.

[13] A. Tucker and A. Gupta. "Process Control and Scheduling Issues for Multiprogrammed Shared-Memory Multiprocessors". In *Proceedings of the 12th Symposium on Operating Systems Principles*, pages 159–166, December 1989.

[14] X. Zhang and P. Srinivasan. "Distributed Task Processing Performance on a NUMA Shared Memory Multiprocessor". In *Proceedings of the 2nd IEEE Symposium on Parallel and Distribut ed Processing*, pages 786–789, Dec 1990.

scheduling the threads of a shared-memory program using the same principles of locality employed in the message-passing model, it is possible to both exploit locality and balance the load. Programs based on this new model perform well on a wide range of architectures, regardless of the degree of load imbalance in the application.

References

[1] T. E. Anderson. "The Performance of Spin Lock Alternatives for Shared-Memory Multiprocessors". *IEEETPDS*, 1(1):6–16, Jan 1990.

[2] T. E. Anderson, E. D. Lazowska, and H. M. Levy. "The Performance Implications of Thread Management Alternatives for Shared Memory Multiprocessors". *IEEE Transactions on Computers*, 38(12):1631–1644, December 1989.

[3] B.N. Bershad, E.D. Lazowska, H.M. Levy, and D.B. Wagner. "An Open Environment for Building Parallel Programming Systems". In *Proceedings of the ACM/SIGPLAN PPEALS 1988 Symposium on Parallel Programming: Experience with Applications, Languages, and Systems*, pages 1–9, July 1988.

[4] S. Dandamudi. "A Comparison of Task Scheduling Strategies for Multiprocessor Systems". In *Proceedings of the Third IEEE Symposium on Parallel and Distributed Processing*, December 1991.

[5] T. W. Doeppner Jr. "Threads: A System for the Support of Concurrent Programming". Technical Report CS-87-11, Department of Computer Science, Brown University, 1987.

[6] T.J. LeBlanc. "Problem Decomposition and Communication Tradeoffs in a Shared-Memory Multiprocessor". In Martin Schultz, editor, *Numerical Algorithms for Modern Parallel Computer Architectures, IMA Volumes in Mathematics and its Applications Volume 13*, pages 145–163. Springer-Verlag, 1988.

locality considerations dominate, and SMP is comparable to message passing. We should note however that SMP does not have the same conceptual simplicity of the shared-memory model, since the programmer must effectively partition the lightweight threads of an application into equivalence classes, so that the scheduler can assign threads that access the same data to the same processor.

7 Conclusions

In this paper we compared two popular programming models for shared-memory multiprocessors: the shared-memory model and the message-passing model. We showed that each model has performance advantages, and that any performance comparison of the two models depends on both the application and the underlying architecture. The load balancing properties of the shared-memory model are beneficial for applications with inherent load imbalance, while the locality properties of the message-passing model are beneficial on machines with relatively expensive communication.

Our experiments using applications with and without load imbalance on a wide range of architectures provides insights into the relative utility of the two models. Based on these experiments we conclude:

- *The relative performance of the two models depends on the cost of communication (relative to computation) as provided by the hardware, and the inherent load imbalance of the application.*
- *Shared memory is preferable on multiprocessors where communication is relatively cheap* (like the Sequent Symmetry). In these machines, the load balancing advantages of shared memory tend to outweigh the locality advantages of message passing. Even in cases where there is no load imbalance, if communication is sufficiently cheap, message passing is only marginally better than shared memory.
- *As the cost of communication increases in shared-memory multiprocessors, message passing is becoming an increasingly attractive alternative to shared memory.* The locality advantages of message passing increase with a rise in the cost of communication, while the load balancing advantages of shared memory are not influenced by a change in the relative cost of communication.
- *It is possible to combine the advantages of both models into a single programming model.* By

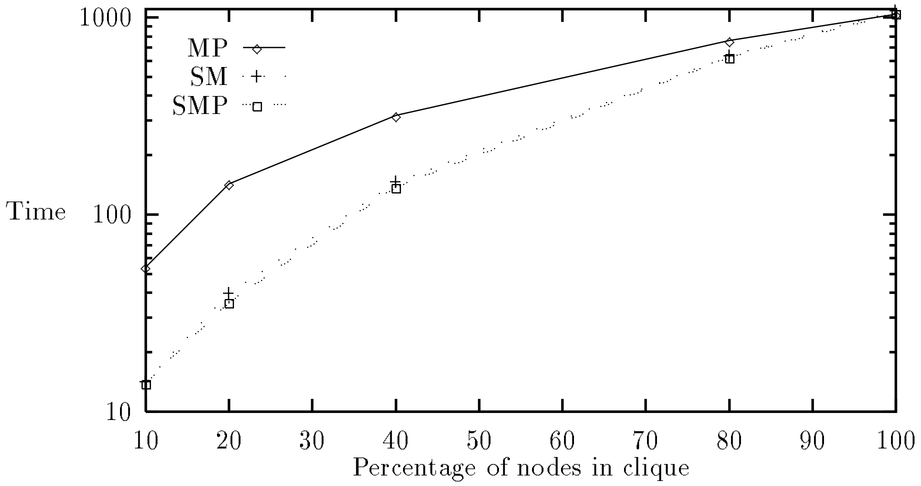

Figure 13: Transitive closure on the Symmetry, as a function of load imbalance.

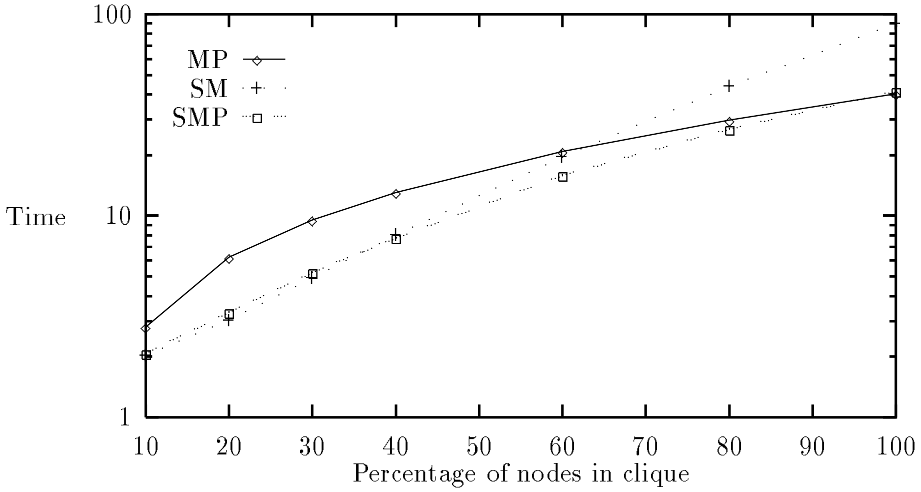

Figure 14: Transitive closure on the Iris, as a function of load imbalance.

That is, we assume that a subset of threads in the fine-grain decomposition of our new model correspond to a single process in the coarse-grain decomposition used in message passing, and we schedule that set of threads on the same processor. The resulting program has the same locality properties as the message-passing implementation. If load imbalance should arise during execution, we reassign threads to idle processors dynamically. Of course any such reassignment would destroy the locality of a thread, but would occur only in response to load imbalance.

This model combines the properties of the shared-memory and message-passing models in that:

- We create a large number of threads, as in the shared-memory model.
- The set of threads in the shared-memory model that correspond to a single process in the message-passing model are scheduled for execution on the same processor, as in the message-passing model.
- If a processor is idle, it removes a thread from another processor's queue, and executes it.

Note that we do not propose a specific policy for thread reassignment. In our current implementation, an idle processor examines the queues of the other processors and removes the first thread from the queue with the most threads. This implementation suffices on small-scale machines, but would not be efficient on a large-scale machine, where a scalable or randomized policy [4] would be more appropriate.

To illustrate the performance of the new model under various load imbalance conditions and communication costs, we repeated the experiments from the previous section, adding an implementation based on this new model. The results appear in figures 13-14.

On the Symmetry (figure 13) SMP closely approximates the performance of shared memory. When there is very little load imbalance, SMP performs slightly better than shared memory, and is comparable to message passing. On the Iris, SMP is comparable to shared memory when there is significant load imbalance, and is comparable to message passing when there is little load imbalance. SMP actually performs the best of all three alternatives near the crossover point for shared memory and message passing, since it combines the advantages of each.

These results show that SMP does in fact have the performance advantages of each model. Where communication is cheap (the Symmetry) and load balancing considerations dominate, SMP is comparable to shared memory. Where communication is expensive (the Iris) or load imbalance is insignificant,

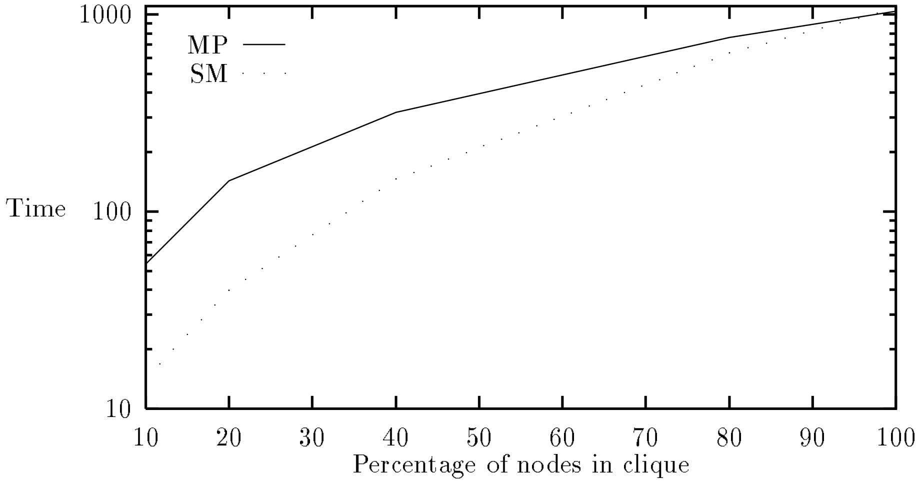

Figure 11: Transitive closure on the Symmetry.

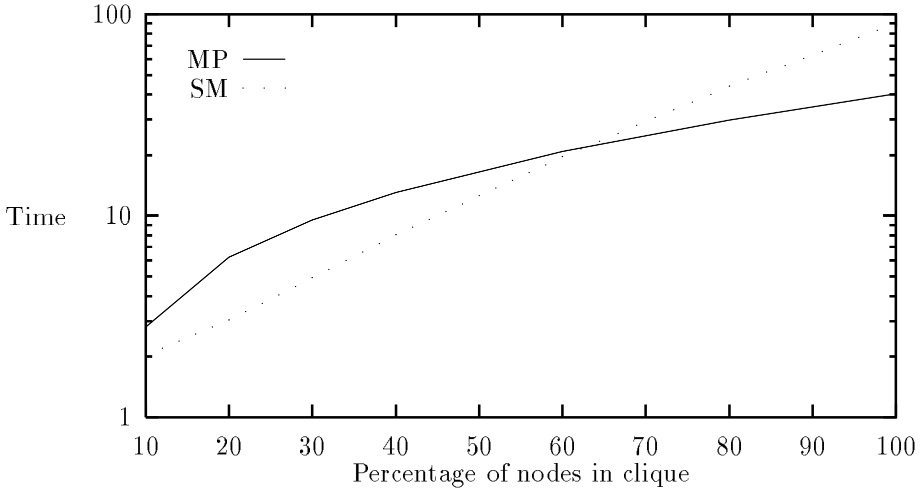

Figure 12: Transitive closure on the Iris.

required twice as much time as the shared-memory implementation.

Even though the Symmetry and Iris are both bus-based, cache-coherent multiprocessors, the results for the Symmetry do not apply to the Iris and vice versa. The cost of communication on the Iris is much higher (relative to the speed of the processors) than it is on the Symmetry, and communication is the dominant factor is a comparison of the two models. We expect future multiprocessors to behave more like the Iris and less like the Symmetry because processors are getting faster at a much higher rate than memories and interconnection networks. If this trend continues, then the advantages of message passing for applications with no imbalance will continue to increase. Even if an application exhibits some load imbalance, it will take a greater degree of load imbalance for shared memory to perform better than message passing on the machines of the future. As a result, there will be very few applications for which there is a performance argument in favor of shared memory.

6 Combining Shared Memory and Message Passing

The experiments in the previous sections suggest that there are always cases where one model will perform significantly better than the other. For a given application, the best choice of model varies across machines. For a given machine, the best choice of model varies across applications. The decision is especially difficult in cases where the correct choice of model depends on unpredictable factors, such as the input values. Rather than force programmers to make a difficult choice between the advantages of each model, we would like to know whether it is possible to combine the load balancing properties of shared memory with the locality properties of message passing.

The main problem with the shared-memory model is the use of a central ready queue, which ignores locality in the scheduling decision. The problem with this approach is that it balances load at the expense of locality *even when there is no load imbalance.* If we combine the lightweight threads of the shared-memory model (essential to load balancing) with the scheduling policy of the message-passing model (essential to locality), we can realize the advantages of each.

In the new model (which we will refer to as SMP), applications create a large number of threads that express the natural parallelism in the problem. Threads are scheduled as in the message-passing model.

different inputs to vary the amount of load imbalance. We used as input a graph of 1000 nodes, where the first N nodes form a clique, and the remaining nodes have no connections. We varied N between 100 and 1000.

In the message-passing implementation, each of the P processors works on $1000/P$ consecutive iterations. In this implementation, most of the work is assigned to the processors working on the first N iterations of the parallel loop; the other processors are idle most of the time. In the shared-memory implementation, work moves freely among processors at the expense of locality, so there is no load imbalance.

Figure 11 plots the completion time of the two programs on an 8-processor Sequent Symmetry as a function of the amount of load imbalance caused by the input. As we increase the percentage of nodes in the clique along the x axis, we increase the distribution of work among processors, and decrease the extent of load imbalance. When 50% of all nodes are in the clique, the vast majority of work is assigned to only 4 processors. When 95% of all nodes are in the clique, the load is fairly evenly balanced among the 8 processors.

We can see from figure 11 that the shared-memory implementation performs best over a wide range of load imbalance. The message-passing implementation performs best only when the load is almost perfectly balanced, and then only performs about 2% better. This experiment confirms the fact that the Symmetry is an excellent vehicle for shared-memory programs, and suggests one reason why shared memory is the most popular programming model for the Symmetry and other bus-based, cache-coherent multiprocessors.

We performed the same experiments on the Iris, which has significantly faster processors than the Symmetry, but only a slightly faster bus (for the number of processors it employs). Figure 12 plots the results for the Iris. On this machine the message-passing model is better than the shared-memory model for a wide range of load imbalance. When there is almost no imbalance, message passing is about twice as fast as shared memory. Shared memory is better than message passing only when the imbalance is such that all the work is assigned to four or fewer processors. Even in the presence of extreme load imbalance, the message-passing implementation is only about 25% worse than the shared-memory implementation. Under the same conditions on the Symmetry, the message-passing implementation for shared-memory programs precisely because the conceptual simplicity of the shared-memory model can be exploited without any significant loss in performance. However, on newer machines, like the Iris and TC2000, which have extremely fast RISC processors and a relatively slow communication medium, the performance penalty associated with the shared-memory model becomes quite substantial. On these machines, the conceptual simplicity of the shared-memory model may not be justified by the cost.

These results illustrate a trend in multiprocessor architecture. Over the last ten years, we have witnessed dramatic improvements in microprocessor speeds due to recent advances in VLSI and RISC technology. These improvements in processor speed have not been accompanied by a corresponding improvement in bus or interconnection network bandwidth. Since processor speeds have improved by two orders of magnitude, while the communication bandwidth has increased by only one order of magnitude, communication is much more expensive (relative to computation) in modern multiprocessors. This trend clearly favors the message-passing model.

If the performance penalty associated with the shared-memory model for a given application is 5% on the Symmetry or Multimax, it will grow to 35% or more on the Iris. The disparity between the models will grow even larger on the next generation of multiprocessors, if current trends continue. Even though the number of communication operations does not increase with changes in processor and memory speeds, if the relative cost of those operations continues to increase, the performance gap between the models will increase as well.

5 Load Balancing vs. Locality

The experiments in the previous section suggest that the message-passing model is clearly preferable if no load imbalance exists, particularly on state-of-the-art multiprocessors, which have relatively expensive communication. Our earlier experiments with transitive closure suggest that under conditions of extreme load imbalance, the shared-memory model is preferable. We will now examine how the degree of load imbalance affects the choice between the models on two machines, the Symmetry and Iris, which differ primarily in the relative cost of communication.

To explore the role of load imbalance, we repeatedly executed the transitive closure application using

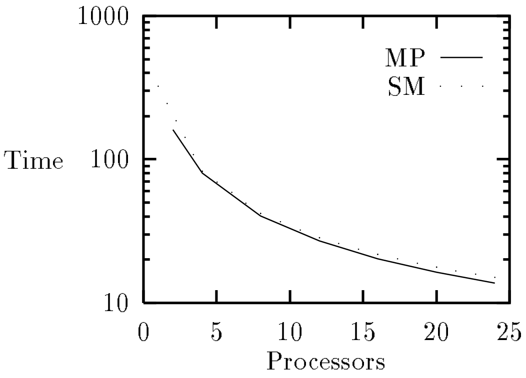

Figure 5: Symmetry

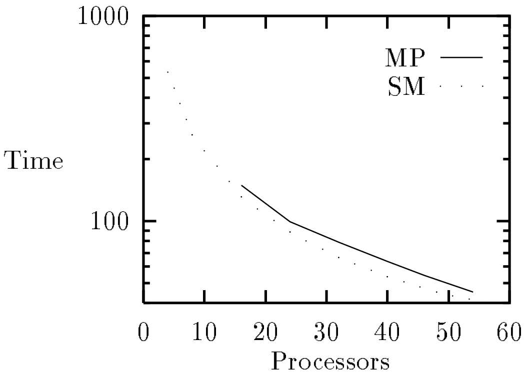

Figure 6: Butterfly I

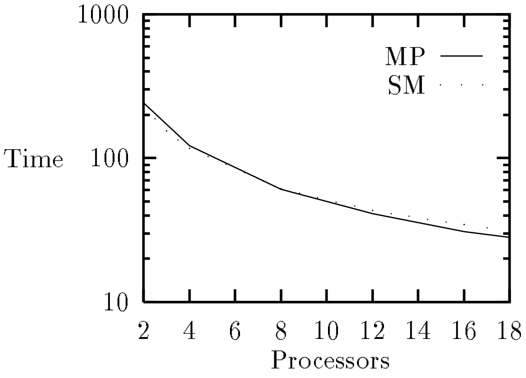

Figure 7: Multimax

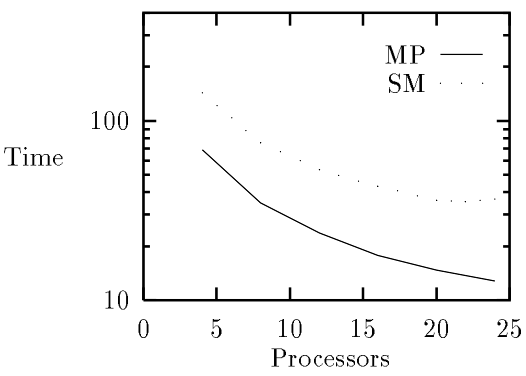

Figure 8: Butterfly Plus

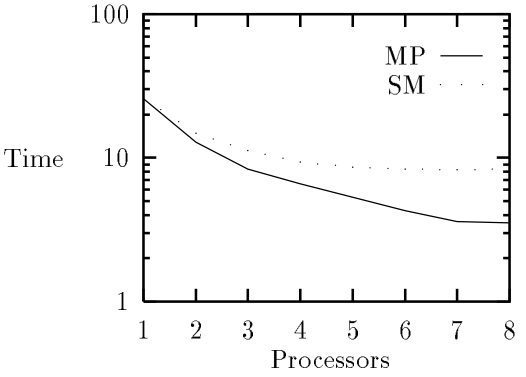

Figure 9: Iris

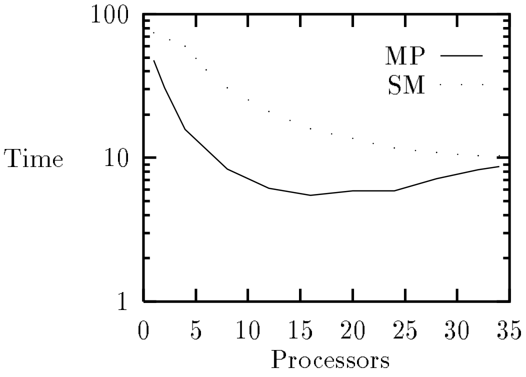

Figure 10: TC2000

and execute it to completion. The pseudo-code for this implementation is as follows:

```
for k = 2 to N do
      forall i = k to N do
            for j = k-1 to N+1 do
                  A[i][j] = A[i][j] - A[k-1][j] * A[i][k-1]/A[k-1][k-1]
```

In the message-passing implementation (MP), each process is responsible for the elimination of all elements in the subset of matrix rows statically assigned to the process. The only communication between processes consists of the broadcasting of pivot rows. The pseudo-code for a process in this implementation is as follows:

```
for k = 2 to N do {
    receive_pivot(pivot,k)
    for i = k to N step P do
           for j = k-1 to N+1 do
                  A[i][j] = A[i][j] - pivot[j] * A[i][k-1]/pivot[k-1]
    }
    if (I_have_row(k+1)) broadcast_pivot(A[k+1])
}
```

We executed these two programs on six different shared-memory machines, including the four machines from the previous section, the BBN Butterfly Plus (the hardware base of the GP1000 product line), and the Encore Multimax (a bus-based cache-coherent machine similar to the Sequent). The results of our experiments are presented in figures 5-10.

It is not surprising that the message-passing implementation performs best in nearly all cases. The overhead associated with communication and process management in the share-memory implementation dominates, because this application has very little load imbalance. These results confirm the observations made in previous studies [6, 7, 11]. It is interesting to note however that the performance advantages of the message-passing implementation vary significantly across architectures, depending on the relative cost of communication as implemented by the hardware.

On older machines, like the Butterfly, Multimax, and Symmetry, there is no discernible difference in the performance of the two implementations. In fact, these machines have been very popular platforms machines have enormous bandwidth for communication, and relatively slow processors for computation. Since our input graph produced quite a bit of load imbalance, the advantages of the shared-memory model dominate on these machines.

In figure 3, the exact opposite occurs. On this machine, the message-passing implementation is better than the shared-memory implementation. The Iris has less bandwidth than the Symmetry, and much faster processors. As a result, communication is relatively expensive on the Iris, and the locality benefits associated with the message-passing implementation dominate any performance improvements due to load balancing, even when the application exhibits significant load imbalance.

The results are less clear on the TC2000, as can be seen in figure 4. On 7 or fewer processors, the shared-memory implementation performs best, while on 8 or more processors the message-passing implementation is clearly better. As more and more processors are used in the message-passing implementation, the amount of load imbalance due to the static assignment of rows to processors decreases. Eventually, the performance benefits of locality begin to dominate, as the need for load balancing decreases. On this machine, which has relatively expensive communication, and for this application, which has a significant amount of load imbalance, the crossover point is around 7 processors.

We can conclude from these experiments that both programming models offer performance advantages. In situations where communication is cheap and load imbalance significant, the shared-memory model is probably the best choice. Where communication is expensive, or where load imbalance is unlikely, the message-passing model is probably better. In the next two sections, we will explore in more detail the roles of the architecture and load imbalance in this tradeoff.

4 Role of the Architecture

In order to focus on the role of the architecture, and in particular on the cost of communication, we will consider an application that does not introduce any significant load imbalance. We will use Gaussian elimination, which was used in the earlier studies comparing these two programming models [6, 7, 11].

The shared-memory implementation creates a thread for each element of the matrix to be eliminated. All threads are placed in a central ready queue. Idle processors take a thread from the ready queue,

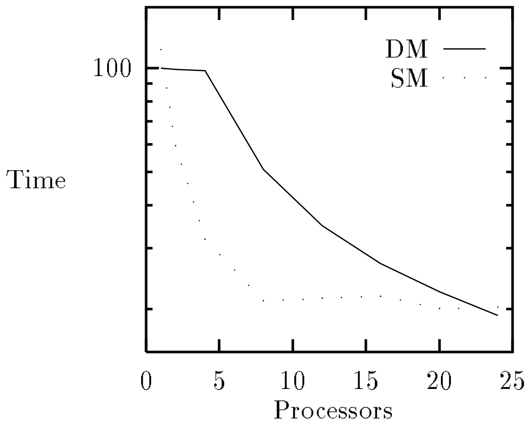

Figure 1: Transitive closure on the Symmetry

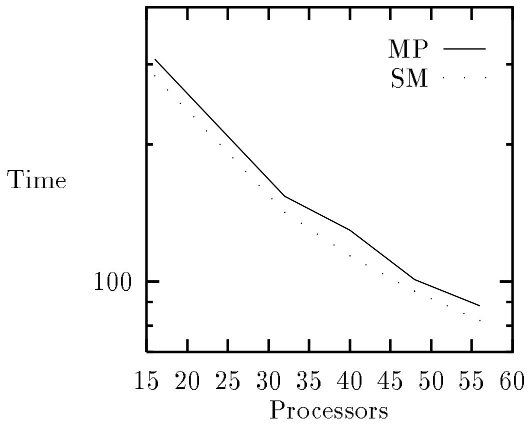

Figure 2: Transitive closure on the Butterfly I

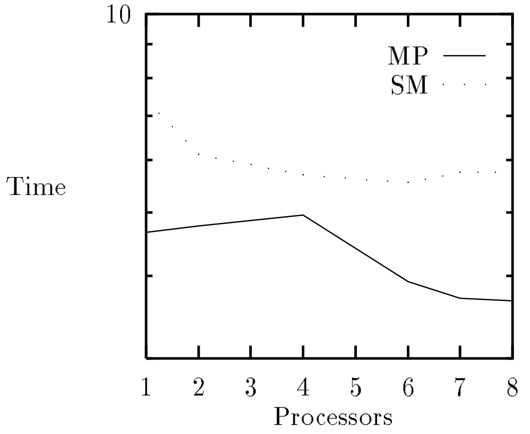

Figure 3: Transitive closure on the Iris

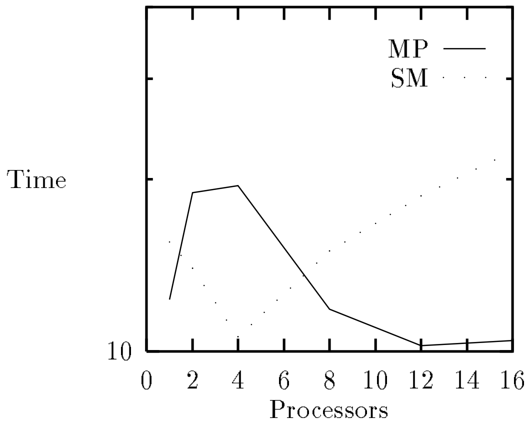

Figure 4: Transitive closure on the TC2000

```
for i = 1 to N do
      forall j = 1 to N do
            if (M(j,i)) then
                  for k = 1 to N do
                        if (M(i,k)) M(j,k) := true
```

An implementation based on message passing would assign a number of rows to each processor statically. In this implementation, each processor is responsible for updating its own rows. During the i_{th} iteration of the outmost loop, the processor containing the i_{th} row broadcasts the contents of the row to the other processors. This implementation has what we call *inherent load imbalance*, in that the amount of work performed by each process depends on the input values. Under any static assignment of processes to processors, the input determines how much load imbalance will occur. In our case, the input to the program is a graph of N nodes, with the first $N/4$ nodes forming a clique and the rest of the nodes having no connections at all, so most of the computation is contained in the first $N/4$ iterations of the parallel loop.

An implementation based on shared memory creates a thread for each iteration of the `forall` loop. Since all threads are scheduled using a central ready queue, no processor is idle while there are threads on the queue. As a result, all processors finish executing within one iteration of each other, and load imbalance doesn't arise. However, each thread must load the data it will need into the local memory or cache when it begins execution, so there is substantial communication overhead associated with the shared-memory implementation.

To quantify the relative importance of communication and load imbalance, we executed the shared-memory (SM) and message-passing (MP) implementations of transitive closure on four shared-memory multiprocessors. We used two bus-based, cache-coherent machines: the Sequent Symmetry S81, and the Silicon Graphics 4D/480GTX Iris multiprocessor workstation. We also used two large-scale distributed shared-memory machines: the BBN Butterfly and the TC2000 from BBN Advanced Computers Inc. These machines span several generations of multiprocessor architecture over the last ten years. The results of our experiments are presented in figures 1-4.

As can be seen in figures 1 and 2, the shared-memory implementation performs slightly better than the message-passing implementation on the Butterfly, and much better on the Symmetry. Both of these thread executes to completion on a single processor, it may spend a substantial percentage of its lifetime bringing the data it needs into the local cache or memory. Threads that access the same data may not execute on the same processor, causing the data to move back and forth between processors. The resulting communication overhead can be substantial, and is incurred whether or not load imbalance exists.

In summary, the shared-memory model employs cheap threads that may execute on any processor and access data anywhere in the machine. The message-passing model employs heavyweight processes that are statically assigned to a processor and access local data only; communication among processors is done through message passing. The shared-memory model has superior load balancing properties, while the message-passing model offers better locality of reference. A performance comparison between the two models depends therefore on the relative cost associated with communication and load imbalance. If communication is extremely expensive in a given architecture, message-passing programs will likely perform better than their shared-memory counterparts. If an application exhibits tremendous load imbalance, the shared-memory implementation will likely perform better. The precise tradeoff depends on the architecture and application.

We should note that any performance comparison ignores an important qualitative difference in the two models: the shared-memory programming model is widely believed to be easier to use than the message-passing model. The conceptual simplicity of the shared-memory model derives from similarities with sequential programming. Evidence in favor of the shared-memory model is the overwhelming dominance of shared-memory multiprocessors for general-purpose parallel programming, and the considerable effort in software development designed to provide the illusion of shared memory on multicomputers.

3 The Effects of Load Imbalance

In order to explore the performance implications of load imbalance, and thereby quantify the performance advantages offered by the shared-memory model, we will consider an application that has the potential for both load imbalance and locality of reference. The following parallel algorithm to compute the transitive closure exhibits both of these properties:

cesses are expensive, and are therefore used sparingly (typically one process per processor). Since all the threads in a shared-memory program share a single address space (resulting in so called *lightweight* processes), threads can be implemented in user-space, and process management overhead is negligible. As a result, threads can be used to represent the parallelism in the application without regard to the physical parallelism provided by the hardware. Although there are conceptual advantages to having more threads than processors in a shared-memory program, a fine-grain decomposition also helps in avoiding load imbalance, which can severely impact performance.

Load imbalance occurs whenever processors are idle and there is work still to be performed. There are two distinct sources of load imbalance in a parallel program: (1) an uneven assignment of computation to processes (or threads) and (2) an uneven assignment of processes (or threads) to processors. A fine-grain decomposition using lightweight threads minimizes the effects of an uneven assignment of computation to threads, since the variance in thread execution times is small. To avoid an uneven assignment of threads to processors, most shared-memory programming systems use a central ready queue from which idle processors remove threads for execution [3, 5, 12, 13]. A single ready queue evenly distributes the load among processors, and ensures that no processor remains idle while there is work to be done. This scheme is effective because, in a shared-memory program on a shared-memory multiprocessor, a thread can execute on any processor and still access the shared data.

In the message-passing model, the static partitioning of data among processes, and the static assignment of processes to processors, can produce load imbalance. Although a static assignment may work well in many cases, there are applications where the work a processor performs depends on factors that are unknown at compile-time, such as the input values. These cases require an assignment strategy that can adjust the load dynamically [14], like the central work queue. Process migration is one alternative for message-passing programs, but migration of heavyweight processes is expensive.

In general, dynamic load balancing policies introduce a lot of communication overhead. Even the central work queue mechanism used in the shared-memory model increases communication. Threads are constantly moved to different processors, always executing on the next available processor so as to avoid load imbalance. Each time a thread is scheduled for execution, it is usually executed on a different processor, although its data may still reside on the processor where it executed previously. Even if a wait for something to happen, cycles are wasted and the hardware is under-utilized. Synchronization can be a dominant source of overhead, but this overhead can be minimized through the use of scalable synchronization primitives [1, 10]. As a result, synchronization overhead need not be significant in either model.

Process management overhead refers to the time required to create, destroy, and schedule processes. Parallel programs in the message-passing model typically create one process per processor, so the overhead associated with process management is negligible. Shared-memory programs, on the other hand, usually employ a large number of fine-grain processes (called *threads*), whose number depends primarily on the amount of parallelism exhibited by the algorithm. A large number of threads would be very expensive if implemented inside the operating system kernel, but the current trend is to move thread management out of the kernel and into a thread library [3, 5]. Given an appropriate user-level implementation of threads [2], most shared-memory applications exhibit no noticeable overhead associated with process management.

Communication overhead is introduced by interaction between processors. Communication manifests itself as cache misses in multiprocessors with caches, as non-local memory accesses in distributed shared-memory machines, and as the sending or receiving of messages in a distributed-memory machine or multicomputer.

In the shared-memory model, threads communicate using references to shared data. Conceptually, the shared data resides in a global memory accessible to all threads. In reality, some data references will be more expensive than others, since only a fraction of the shared data will reside in local memory. The global memory is typically implemented using a single address space, which is shared by all the threads in a program.

In the message-passing model on the other hand, processes communicate using messages; each process resides within its own address space, and there is no data shared between processes. All data must be associated with a process statically, and is accessible only to that process. Thus, all data references are to local memory.

Since the message-passing model allocates an address space for each process (resulting in so called *heavyweight* processes), the operating system is involved in process management. In this model, pro-

All of these experiments were performed on shared-memory multiprocessors, and therefore lead one to conclude that there are *no* performance advantages to using the shared-memory model on any machine. This conclusion implies that shared memory is a luxury, providing conceptual simplicity at the expense of application performance. Since there are no metrics that measure conceptual simplicity during the programming process, any quantitative comparison of the two models favors message passing.

In this paper we argue that previous comparisons of the two models ignored an important dimension that favors shared memory, namely load imbalance. We show that applications with inherent load imbalance often execute more efficiently under the shared-memory model than under the message-passing model, and therefore any quantitative comparison of the two models must balance the relative advantages of locality of reference in the message-passing model against the load balancing properties of the shared-memory model. Our experiments on a wide range of shared-memory multiprocessors quantify this tradeoff, while varying the relative cost of communication. Our results show that the relatively cheap communication in the previous generation of multiprocessors favors the shared-memory model (assuming some load imbalance), while the recent increase in the cost of communication (relative to the cost of computation) favors the message-passing model. Since an accurate comparison of the two models depends on factors that may be unknown until runtime, we propose an alternative programming model, which has the load balancing properties of the shared-memory model and the locality properties of the message-passing model. We show that this new model performs better than the other two alternatives, regardless of the relative cost of communication or the load imbalance inherent in the application.

2 Comparing Programming Models

There are several dimensions along which a performance comparison between the two models can be made, where each dimension is related to an important source of overhead in parallel programs. We will consider four sources of overhead that arise in parallel processing: *synchronization*, *process management*, *communication*, and *load imbalance*.

Synchronization overhead stems from the fact that a processor often has nothing to do while it waits for another processor to relinquish a critical region or provide the required data. If a processor must

1 Introduction

Two of the most popular programming models for parallel processors are the shared-memory (SM) and the message-passing (MP) models. Each of these models was developed for a different class of multiprocessor, and each model exhibits characteristics that reflect assumptions about the underlying machine. The SM model uses shared memory for inter-process communication and is typically associated with shared-memory multiprocessors like the Sequent Symmetry. The MP model uses message passing for inter-process communication, and is typically associated with distributed-memory multiprocessors like the Intel Hypercube, or distributed multicomputers made up of a network of workstations. The defining characteristic for these two models is the communication mechanism provided by the underlying hardware: the SM model assumes hardware support for shared-memory, while the MP model makes no such assumption.

Recent advances in both hardware and software technology have blurred the distinction between shared-memory multiprocessors and distributed-memory multicomputers. The (relative) cost of communication has increased enormously on shared-memory multiprocessors, due to fundamental breakthroughs in processor technology (like the advent of RISC processors) that were not accompanied by similar breakthroughs in memory and switch technology. At the same time, sophisticated software techniques have been developed that implement shared memory in software on top of multicomputers [8]. As a result of these two developments, both programming models are now in use on both classes of machine.

In earlier work [6, 7] we argued that a message-passing implementation performs better than a shared-memory implementation of the same program on distributed shared-memory machines like the BBN Butterfly. Lin and Snyder [9] came to the same conclusion based on their experiments on both the Butterfly and the Sequent Symmetry, a bus-based, cache-coherent multiprocessor. Subsequent work by Ngo and Snyder [11] verified these conclusions using a more sophisticated set of applications, and a wider range of shared-memory machines. In each case, the conclusion favoring the message-passing model was attributed to the locality of reference implicit in the message-passing model, and sorely lacking in the shared-memory model.

Shared Memory vs. Message Passing in Shared-Memory Multiprocessors

Thomas J. LeBlanc
leblanc@cs.rochester.edu

Evangelos P. Markatos
markatos@cs.rochester.edu

Computer Science Department
University of Rochester
Rochester, NY 14627

April 30, 1992

Abstract

Shared-memory multiprocessors can support a model of parallel computation based on either shared memory or message passing. Since the programming model dictates so many aspects of the implementation of an application, including process decomposition and scheduling, the choice of programming model is a dominant factor in application performance. In this paper we argue that the choice between the shared-memory and message-passing models depends on two factors: the relative cost of communication and computation as implemented by the hardware, and the degree of load imbalance inherent in the application. We use two representative applications to illustrate the performance advantages of each programming model on several different shared-memory machines, including the BBN Butterfly, Sequent Symmetry, Encore Multimax, and Silicon Graphics Iris multiprocessors. We show that applications implemented in the shared-memory model perform better on the previous generation of multiprocessors, while applications implemented in the message-passing model perform better on modern multiprocessors. We argue that both models have performance advantages, and that the factors that influence the choice of model may not be known at compile-time. As a compromise solution, we propose an alternative programming model, which has the load balancing properties of the shared-memory model and the locality properties of the message-passing model, and show that this new model performs better than the other two alternatives.