A Comparison of Programming Models for Shared Memory Multiprocessors*

Calvin Lin
Lawrence Snyder

University of Washington
Department of Computer Science and Engineering, FR-35
Seattle, WA 98195

Abstract

Shared memory machines can be programmed using any of several models of parallel computation: The shared memory model compiles directly, and the nonshared memory model can be implemented simply by simulating message passing. In this paper preliminary evidence is presented suggesting that the nonshared memory programming model is actually *better* for shared memory machines than the shared memory model. Possible explanations for this observation are offered. Locality and granularity of parallelism, which are encouraged by the nonshared memory model, seem to explain the reported results. To understand these and other confounding features on the issue of "best programming model for shared memory machines," a broader study, involving more researchers, programs, and machines is proposed.

1 Introduction

There currently exist a wide range of parallel programming languages and parallel machines. Rather than evaluate a specific instance, the goal of this paper is to identify broad principles applicable to a variety of machines and languages. By inquiring into a particular question concerning parallel programming, this paper seeks to motivate others to join in conducting comparative experiments.

The question addressed here is: *How do programs written in the shared memory and nonshared (distributed) memory models of parallel computation compare in performance on shared memory multiprocessors?* This question may seem odd, but it is motivated by the goal of portability. The reasoning proceeds as follows: (1) Parallel programs for distributed memory parallel machines should probably be written using the distributed memory model; the shared memory model is precluded because it is difficult for compilers to hide the large latencies of message passing. (2) On the other hand, distributed memory programs should execute passably on shared memory machines since message passing is easily emulated in shared memory at small cost. If the suppositions are correct they would seem to suggest that programs written in distributed memory model languages are likely to be more portable[1] than programs written in shared memory languages. Though it appears difficult to test either antecedent thoroughly given the present state of parallel computation technology, the second statement does admit some analysis.

The results were unexpected. Rather than showing programs of the two models to be essentially equivalent, with the nonshared memory versions being sightly inferior, as the "emulation" reasoning of statement (2) would suggest, *the programs written in the distributed memory model were substantially faster.* The possible reasons for this will be considered below, but the intuition is that programs based on the nonshared memory model exploit locality well and emphasize the more efficient large grain size – two features which are beneficial for shared memory machines as well as nonshared memory machines.

The results presented are preliminary in a variety of ways. Only two problems were solved on two machines, advanced compilation techniques were not available to us, and several potential differences between the two models couldn't even be tested on the available hardware. Moreover, there is a possibility we have been biased in some way, though we have tried to avoid it. *Remedying these shortcomings will require broad, objective empirical studies involving many researchers. It is in the spirit of proposing a community-wide challenge rather than offering tidy conclusions that this paper is written.*

The next section presents some background and explains our methodology. We then describe the sample programs, and in subsequent sections describe and interpret the experiments. Concluding remarks are presented at the end.

2 Background and Methodology

The general topic is to better understand the portability of parallel programs by studying how a programming language's memory model interacts with the object machine. The specific question is to identify differences between the shared and distributed memory programs executing on a shared memory machine. (We clarify the terminology below.)

The first (nonexperimental) study of the interaction between programming model and machine [14] focused principally on the programming model for nonshared memory machines, but it supports statements (1) and (2) of the introduction.

Baillie [3] assumed that the architecture dictates the choice of programming model, i.e. a shared memory machine implies a shared memory model, and a nonshared memory machine implies a nonshared memory model. LeBlanc [8] compared the shared and nonshared memory models by studying the implementation and performance of Gaussian elimination programs on the Butterfly. LeBlanc concludes that "the particular model of computation in use is less important than how well it is matched to the application," but also mentions that message passing may "leave the programmer fewer opportunities to introduce inefficiencies" and has the advantage of encouraging the programmer to exploit data locality.

Agreed upon definitions for shared memory and distributed memory programming models do not exist, nor does the conceptual basis needed to give fully precise definitions. The following explains our understanding of the prevailing use of these terms. The primary difference between shared and distributed memory programming models is the assumed memory reference time: Shared memory models do not differentiate among memory reference times, implicitly making all references unit-cost. Examples include the FORTRAN-based languages, data flow and functional languages, etc. Nonshared memory models distinguish between local references, which are unit-cost, and nonlocal references, which have substantially larger reference times. Exam-

*This research was supported in part by AFOSR Grant 89-0282, in part by ONR Contract N00014-89-J-1368.

[1] Portable here means runs *well* on a variety of architectures. The emphasis on performance is critical, since presumably any reasonable parallel computer is universal in the sense of computability, making it *possible* to host the programs.

ples include CSP derivative languages, cube-specific languages, etc. (Finer cost distinctions are also possible [2].) To be sure, many shared memory languages allow the programmer to declare data as "shared," "local," or belonging to some other storage class, but it is precisely because these declarations are not needed that shared memory languages have become widely accepted as being easier to program.

Another feature that seems to distinguish the two models is the use of explicit specification of parallelism. In distributed memory languages the parallel threads of control are usually given explicitly as process declarations. The need to perform the nonlocal references explicitly (with sends and receives) and the implication that these operations are expensive, apparently encourage programmers to write "coarse grain" processes that maximize local computation, i.e. exploit locality. Shared memory languages tend not to require explicit specification of parallelism, or if they do the undifferentiated memory costs promote the use of fine grain threads to maximize concurrency. In either case locality is not generally available and the only possibility of finding it is through advanced compilation techniques. Again, differences exist among shared models, but the requirement of explicitly specifying parallelism will be taken as a property of nonshared memory models.

Why would these characteristics be beneficial to a shared memory machine?

First, exploiting locality is useful once one observes that shared memory machines do not have a flat RAM structure, but rather they generally possess some type of memory hierarchy.[2] Local caches on the processors, cluster memory as in the Cedar machine, split global and local memory as on the Butterfly and the RP3 – all are examples of structures that provide a performance improvement for local data. Furthermore, knowing that data is used locally enables a compiler to exploit these features.

Second, the coarser grain of distributed memory programs reduces the overhead of process creation, management and multiplexing. Even with hardware assistance these operations can be expensive, independent of the machine's memory structure. Although it is possible for a compiler to create coarse grain computation from fine grain specifications, it is likely to be more effective if provided explicitly by the programmer.

Our methodology was to write shared and nonshared memory programs and then compare their performance in solving the same set of problems on a shared memory machine. Our shared memory language was the C-based language available from the manufacturer. For distributed memory programming we used Poker [13]. Since this language was not always available on the shared memory machines, we wrote subroutines for operations such as "send" and "receive" and then hand translated the program into a form acceptable to the C compiler. Thus, both languages employ the same "low level" compiler facilities, e.g. code generation, register allocation, etc.

3 Setting

The problems to be solved are the Jacobi method and matrix multiplication. The machines are the BBN Butterfly and the Sequent Symmetry.

The Jacobi method is an iterative technique to solve partial differential equations [7]. In our case we use the Jacobi method to solve Laplace's equation on a rectangle, where the rectangle is represented as a 2D array of integers, V. The problem is governed by the equation

$$\frac{\partial^2 V}{\partial x^2} + \frac{\partial^2 V}{\partial y^2} = 0,$$

$$V(x, y) = 0 \text{ on the boundary.}$$

The algorithm involves three steps (see Figure 1): initialize the matrix, **V**, repeatedly compute local averages for each element of **V**, and return the results. Figure 2 shows how the local averages are based on the 5 point stencil. Note that the algorithm is presented here in a generic form. The way in which this algorithm is parallelized will depend on the implementation.

(1) Allocate and initialize matrix V;

(2) repeat
  for each point in V
    $V_{i+1}[x,y] = \frac{V_i[x+1,y]+V_i[x-1,y]+V_i[x,y+1]+V_i[x,y-1]}{4}$;
  until convergence == true;

(3) print results;

Convergence test: $|V_{i+1}[x,y]V_i[x,y]| < \delta$ for all $V[x,y] \in V$.

Figure 1: High Level Jacobi Algorithm

In our particular problem all data points are initialized to a constant integer value. We also use constant boundary conditions.

The matrix multiply problem is to find the product, **C**, of two dense input matrices, **A** and **B**. In one experiment, we use the straightforward approach in which each element of **C** is computed as a dot product of a row of **A** and a column of **B**. There is much room for parallelism, as each dot product can be computed independently. In our programs, we spawn multiple tasks, and each task computes a different subset of the solution matrix. As input, we use dense matrices with integer elements. The dimensions of **A** and **B** are 200×400 and 400×300, respectively.

Our BBN Butterfly GP1000 has 32 nodes.[3] In addition to a Motorola 68020 processor, each node has 4 Mbytes of local memory and a processor node controller which interacts with an omega network to make remote references when needed. Together, the 32 memory modules, the process node controllers and the network form a single shared memory which all processors may access. Local memory access is about 5 times faster than remote access [9].

In contrast to the Butterfly, our Sequent Symmetry Model A has 20 Intel 80386 processors connected by a shared bus. Each processor has a 64K cache which holds both data and instructions. A modified Illinois ownership scheme is used to maintain cache coherency [11]. The Symmetry currently has 32Mb of main memory.

Our programs were all written in C. For the Butterfly, the BBN Uniform System [4] was used to provide task creation, synchronization, and memory management routines, while on the Symmetry, these facilities were provided by the DYNIX operating system. The shared memory programs were written from scratch, with the programming model being that of a parallel machine with symmetric processors,

NW_i	N_i	NE_i
W_i	v_i	E_i
SW_i	S_i	SE_i

$$v_{i+1} = \frac{N_i+E_i+W_i+S_i}{4}$$

Figure 2: Five Point Stencil for Computing Local Average

[2] It is this feature that motivates some architects to see shared and distributed memory architectures converging.

[3] We currently have only 24 nodes available, and only 19 available for general purpose use.

cheap task creation, and a single shared memory. The nonshared memory programs used the Poker model [13], a distributed model in which processes communicate by sending synchronous messages through well defined ports. To support this model we used C to implement Poker-style message passing routines for both the Butterfly and the Symmetry.

4 Jacobi

With either programming model there is much room for variation in implementing a particular program. To see this, note that one can simulate either of the two programming models with the other. For our first Jacobi experiment on the Butterfly we chose to implement those programs that require the least intellectual effort with respect to their particular model, since these programs will best reflect the biases of the respective programming models. We dub these two implementations of least intellectual effort **sm1** and **nsm1** because they are of primary importance in comparing the two models.

4.1 Description of sm1 and nsm1

We now describe **sm1** and **nsm1** in terms of N, the number of data points, and P, the number of processors.

```
(1) constant tolerance = δ;
    global int delta = 0;

    Allocate global current[] and global next[] matrices;
    Initialize current[];

(2) repeat
    {
        spawn N tasks: jacobi(x,y);
        Swap (current, next);
    } until delta < tolerance;

(3) print results;

jacobi(x,y)
    next[x,y] = (current[x+1,y]+current[x−1,y]+
                 current[x,y+1]+current[x,y−1])/4;
    diff = Max (delta, |next[x,y] − current[x,y]|);

    /* update delta */
    lock delta;
        if (delta < diff)
            delta = diff;
    unlock delta;
```

Figure 3: Algorithm for **sm1**

In **sm1** two 2D arrays are allocated in shared memory. One array holds the values for the **current** iteration. The other holds values of the **next** iteration. To reduce memory contention these arrays are allocated in a "scattered" fashion [4]: each row of the matrix is allocated to a different memory module. After each point of the **current** matrix is initialized, N tasks are spawned, one for each data point (see Figure 3). Each task computes one local average, updates **delta**, and terminates. We repeatedly spawn N tasks until the tolerance is met.

In **nsm1** only P tasks are spawned, one per processor (see Figure 5). As in **sm1**, we use two arrays, **current** and **next**. Unlike **sm1**, these arrays are logically distributed, so each task owns square submatrices of **current** and **next**.

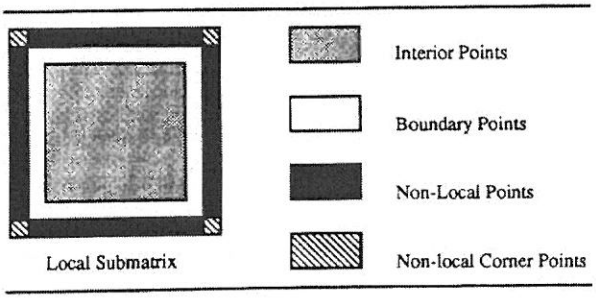

Figure 4: Local Submatrix

Note that in **nsm1** each task operates on multiple data points, but the local averages for the edges of the submatrix (the "Boundary Points" in Figure 4) cannot be computed until their neighbor values are sent from remote tasks. So the **nsm1** algorithm requires an explicit communication step before all local data points can be computed. Once these are computed and the local maximum **delta** value has been found, the P tasks form a logical tree to find the global maximum value of **delta**, and then to broadcast this maximum value to all tasks. Each task can then test for termination.

4.2 Discussion of sm1 and nsm1

The results show **nsm1** to be 23 times faster than **sm1** for 16 processors and 8 times faster for 4 processors (see Figure 6 and Tables 1 and 2). The inferior performance of **sm1** compared to **nsm1** is due presumably to poor data locality. Our measurements show that the cost of spawning additional tasks in **sm1** is negligible. So we see that while single point code and shared memory access to array elements are each great programming conveniences, together they tend to result in programs which make too many remote references.

4.3 Optimized Jacobi

Our next experiment involves **smo** and **nsmo**, so named because they are optimized ('o' for optimized) versions of **sm1** and **nsm1**. These programs are interesting for two reasons. First, as will be discussed below, the optimized shared memory program can confirm our explanation of **sm1**'s poor performance. And second, the results may hint at some inherent performance advantage of one model over the other for the Butterfly.

4.3.1 Description of smo and nsmo

To avoid the inefficiencies of its predecessor, **smo** attempts to minimize its number of remote references. **Smo** spawns P tasks, one for each processor (see Figure 7). Like **nsm1**, each task allocates a contiguous portion of the 2D arrays, **current** and **next**. Although these arrays are local to a particular processor, they are still in shared memory. However, the arrays are no longer logically contiguous, so tasks must have some way to access the remote elements of the matrices. To do this, a directory at a globally known address holds the addresses of each of the P submatrices. Once the address of a remote matrix is known, it's individual elements can be accessed as well. Because interior points and exterior points are accessed differently, we actually allocate an array large enough to hold the local submatrix and all of its adjacent Non-local points (see Figure 4). This eliminates the need to check for the boundary condition when computing the local neighbor averages.

The algorithm for **nsmo** is identical to that of **nsm1**. The differences are that **nsmo** uses variable length messages, and that tasks pass only one message for each edge instead of passing multiple small messages, as was done in **nsm1**. For large messages, the message passing routines use the Mach **btransfer()** procedure, which is optimized for transferring large blocks of data. Notice that such a feature would likely be part of a good implementation of Poker on the Butterfly and probably wouldn't require programmer intervention as in the **smo** improvements. Also note that for East-West neighbors additional work is needed to bundle each vertical column of data into a contiguous block of data which can be passed as a single message. In this case, we're trading additional computation for more efficient interprocess communication.

4.3.2 Discussion of smo and nsmo

The results of Table 1 and Table 2 show that **smo** is a vast improvement over **sm1**, but is still slower than both **nsm1** and **nsmo**. **Nsmo** is 10.5% faster than it's predecessor (for N=1M and P=4) and about 21.5% faster than **smo** (for N=1M and P=4).

Note that in **smo** the array of data is now logically distributed as well as physically distributed. We no longer have the convenient location-independent array addressing of **sm1** because non-local array elements must be accessed indirectly through the directory. *Thus, in order to achieve comparable performance we have lost one of the major advantages of the shared memory model.*

In fact, the **smo** algorithm is very similar to the nonshared memory algorithm, the main difference being the way edge values are transferred, and the manner in which the maximum **delta** value is calculated. Finally, note that the differences between **nsm1** and **nsmo** are small optimizations while the changes from **sm1** to **smo** involve more fundamental changes to the program.

4.4 Nine Point Jacobi

Having found the optimized nonshared memory program to be somewhat faster than its shared memory counterpart, it is natural to ask how the two models compare for a problem with different communication costs. One easy way to explore this question is to compare the Jacobi method using the 9 point stencil. This variation of our earlier problem increases the amount of interprocess communication, yet requires little additional programming effort. Our programs using the 9 point stencil are named **sm9** and **nsm9**.

4.4.1 Description of sm9 and nsm9

Recall that in the 5 point stencil the value of a data point is computed as the average of its North, East, West, and South neighbors. In the 9 point stencil, a data point is the average of its eight neighbors: North, NorthEast, East, SouthEast, South, SouthWest, West, and NorthWest.

The **sm9** program is derived from **smo**, with two differences: The main computation loop becomes

$$v_{i+1} = \frac{N_i + E_i + W_i + S_i + NE_i + SE_i + SW_i + NW_i}{8}$$

and the Corner Points of the local arrays (see Figure 4) must now be filled with values from neighboring tasks. Likewise, **nsm9** is a descendant of **nsmo**, with the differences being four additional messages to receive corner values, and the compute loop modified as shown above.

4.4.2 Discussion of sm9 and nsm9

The results show that while **nsm9** still outperforms **sm9**, the discrepancy is not as large as in the 5 point problem (see Tables 1 and 2). This is no surprise. The 9 point problem favors the shared memory implementation because each task must obtain data from four additional processors. In the shared memory case this is four more indirect memory references, while in the nonshared memory case each task passes four additional messages. But these messages are short, so because of the overhead of message passing, the additional communication is more efficient in the shared memory case.

4.5 Jacobi by Rows

At one extreme, **sm1** was programmed without regard to locality. At the other extreme, **nsm1** maximizes locality and minimizes communication by using square submatrices whenever possible. A compromise, and a technique which is often found in parallel programs, is to allocate local data in terms of rows, rather than in terms of square submatrices (see Figure 9). Allocation by rows is typically easier to program than allocation by square (or almost square) submatrices, but it essentially increases the interprocess communication requirements of the resulting program. Our next experiment involves **smr** and **nsmr**, in which data is allocated by rows.

4.5.1 Description of smr and nsmr

The **smr** program is a modification of **smo**. The program is unchanged except for two details. Each task allocates a contiguous group of rows to hold rectangular submatrices of **current** and **next**. And secondly, there is only communication between North-South neighbors since East-West neighbors don't exist. Similarly, **nsmr** differs from **nsmo** in its data allocation and in its communication behavior.

4.5.2 Discussion of smr and nsmr

In the worst case, for P=16 processors, allocation by rows causes 2.5 times as much data to move between processors. In the best case (not including P=1,2, or 3) allocation by rows causes 1.5 times as much data to be moved. Surprisingly, **smr** and **nsmr** performed as well as their predecessors, **smo** and **nsmo** (see Tables 1 and 2). These results can be explained: Although **smr** and **nsmr** require more data movement, each process also has fewer neighbors and thus passes fewer messages. For example, for P=16 a total of 30 messages are passed per iteration, while **smo** and **nsmo** each pass 48 messages per iteration. Saltz, Naik and Nicol [12] discuss in more detail the tradeoffs between rows vs. squares for distributed memory machines.

4.6 Jacobi on the Symmetry

The shared memory programming model matches the Symmetry much better than it does the Butterfly. We would expect data locality to be less important on this shared bus machine. We ported **sm** and **nsm** from the Butterfly, resulting in $\mathbf{sm}_s$ and $\mathbf{nsm}_s$, respectively (the subscript standing for "Symmetry"). The main difference between the ported code and the original code was that on the Symmetry matrices are always allocated in shared memory, i.e. we can only exploit data locality implicitly by repeatedly accessing the same data because there is no way to explicitly load the caches.

Surprisingly, we found that the nonshared memory was slightly faster (see Tables 3 and 4). We conjecture that in the nonshared memory program the overhead of message passing is offset by the use of **bcopy()** to load "external" data points; thus the caches are loaded more efficiently in the nonshared memory program.

5 Matrix Multiply

For the matrix multiply problem we took existing shared memory implementations from Harrison [5] and compared it to our own nonshared

memory implementations written using the same high level algorithm. The shared memory programs allocate matrices **A**, **B**, and **C** so that they're "scattered" among the available processors. Then, a number of tasks are spawned which compute the various dot products. Each task computes a portion of a row of **C**, depending on a runtime parameter. For our comparison, we chose the value known to give best results for the given program input[6].

Three shared memory implementations were used. The first program, **matrixSM1**, references elements of all three matrices without regard to physical location. The second program, **matrixSM2**, is an optimized version of the first, in which tasks use block transfers to cache the necessary rows and columns before they compute dot products. Finally, **matrixSM3** is the best of Harrison's various programs. It is optimized to assign contiguous groups of rows of matrix **A** to the same processor, to assign threads to where "their" data is, to have tasks migrate to other portions of the matrix when they finish with their own allotment, and to store the 2D matrix **B** as a single 1D array in column-major order so that multiple columns of the matrix may be transferred and cached with a single block transfer.

The nonshared memory version, **matrixNSM**, spawns one task per processor. Each task owns some rows of **A** and **C**, and some columns of **B**. For example, *task 0* owns rows 0 to m of matrix **A**, and it calculates and owns rows 0 to m of matrix **C**, where the value of m depends on the number of processors and the dimensions of **A**. This task also owns columns 0 to n of matrix **B**, so the upper left $m \times n$ portion of **C** can be computed by *task 0* without communication. To compute the remaining values for rows 0 to m this task must access the remote columns of **B**. This is done by passing a series of messages. The P tasks form a logical ring. For the kth iteration through the "shift" loop (see Figure 10), each task sends its columns of matrix **B** to the task on the ring which is k tasks to the left, so that after $P-1$ iterations, each task has received each column of matrix **B**.

Figure 11 shows that the nonshared memory program is faster than all but the best of the shared memory programs.[4] As with the Jacobi programs, the results can be explained by examining data locality. The **matrixSM1** program makes a large number of remote data references. Because the rows of all arrays are scattered, on the average, only $\frac{1}{P}$ of the matrix references are local, where P is the number of processors used in the computation. In contrast, the nonshared memory program and the best shared memory program only reference local values of matrix **A** and **C**, and each element of **B** is sent to any given processor at most once. The **matrixNSM2** program falls in between **matrixSM1** and **matrixNSM** in terms of remote references, as its references to **A** and **C** will sometimes ($\frac{P-1}{P}$, on average) require block transfers.

Offsetting the performance benefit of the nonshared memory model is the fact that the source code for the shared memory programs are much shorter than the nonshared memory program and appear to have been significantly easier to write.

5.1 Matrix Multiply on the Symmetry

To compare matrix multiply implementations on the Symmetry we ported **matrixNSM** and **matrixSM** from the Butterfly. As with the Jacobi programs, the ported programs can perform none of the data scattering optimizations that were done on the Butterfly.

The shared memory program required some modifications: Whereas the Butterfly's Uniform System provides the means to spawn many tasks cheaply, thread support is different on the Symmetry, so we use a slightly different work queue model of computation. The unit of work is still **taskSize** columns of the solution matrix, where **taskSize** is a runtime parameter. However, only P tasks are spawned which repeatedly obtain and perform a unit of work until the queue is empty. For this problem the work queue is implemented as a counter, initially 0. Work is obtained by atomically adding **taskSize** to the integer. Atomicity is guaranteed by protecting the work queue with a lock. The work queue is exhausted when the value of the integer equals the number of units of work to be performed.

Various values of **taskSize** were tested and the best (50) was chosen. The choice of **taskSize** did not have a significant effect on performance. The nonshared memory program exhibits better speedup (see Figure 12). The nonshared memory program is more efficient in two respects: (1) It does not have the overhead of the work queue, and (2) it has better locality of data since each thread operates on the same data throughout the life of the program. The effects of (2) are difficult to measure directly because we have no way to turn off the cache. The effects of (1) can likely be reduced by using multiple queues.

5.2 Divide and Conquer Matrix Multiply

As an alternative to the naive matrix multiply approach we chose to implement a divide and conquer algorithm. The basic idea is that the product of N×N matrices can be thought of as the sums of 8 $\frac{N}{2} \times \frac{N}{2}$ matrix products (see Figure 13). We recursively apply this reasoning until we have nothing left to multiply except scalars. This divide and conquer algorithm makes better use of data locality than the algorithm presented in the previous section.

The shared memory implementation is straightforward. We spawn one task per recursive call. The recursion bottoms out when 2×2 matrices are multiplied.

The nonshared memory algorithm is conceptually identical but the details are nontrivial. Nelson's algorithm [10] spanws one task per processor and decomposes the matrices in a manner analogous to Strassen's algorithm [1], to the point at which all processors have been assigned two square submatrices, one per input matrix.[5] Nelson's algorithm uses a two-level approach. At one level the processes send and receive matrices from neighbors: Communication proceeds as if the processes were arranged as a hypercube. At another level a sequential algorithm is used to compute the product of m×m matrices, where $m = \frac{N}{P}$ and m×m is the size of the subarray initially allocated to each processor.

Table 5 shows the difference between the shared memory and nonshared memory programs. Most of the difference in performance appears to be due to the increased overhead of thread creation in the shared memory program.

6 Conclusion

We have presented solutions to two problems on two machines in an effort to understand how the shared and distributed memory programming models differ in performance. The results, though preliminary, show substantial benefit for the distributed memory model over the shared model. The observations seem to be explained by the exploitation of locality and the large granularity of the distributed memory programs. Indeed, as the shared memory model programs were improved, they acquired some of the same characteristics that the nonshared memory programs exhibited. Clearly, more problems and more machines must be considered before these results can be considered conclusive.

There are important methodological issues that must be considered, too. The phenomena under study – memory models of parallel computation – have not been rigorously defined; greater precision is

[4] The **matrixNSM** program can be optimized in ways similar to **matrixSM3**.

[5] Note that unlike Strassen's algorithm, Nelson's algorithm does *not* reduce the number of actual multiplications.

needed. There are questions of when the observed differences are a result of programming differences and when they simply reflect different algorithms; the Jacobi example appears to reflect only programming differences, but the situation is less clear for matrix multiplication. The problem is likely to get much more complex as the programs do. These and other methodological issues are worthy of deeper consideration.

Acknowledgements It is a pleasure to thank Hans Mandt, Stu Stern, and the Advanced Systems Laboratory of Boeing Computing Services for their help in providing access to a Butterfly multiprocessor. We wish to thank Gail Harrison for a dozen "assists" in various aspects of this research.

References

[1] A. Aho, J. Hopcroft, and J. Ullman. *The Design and Analysis of Computer Algorithms*, Addison-Wesley, (1974) pp 230-232.

[2] A. Aggarwal, A. Chandra, and M. Snir. Hierarchical Memory with Block Transfers, *Proceedings 28th FOCS*, IEEE, (1987) pp 204-216.

[3] C. Baillie. Comparing Shared and Distributed Memory Computers, *Parallel Computing* vol 8, (1987) pp. 267-279.

[4] BBN. Programming in C with the Uniform System. (1986) pp. 11-50.

[5] G. Harrison and D. Notkin. Effective Portability, TR 89-09-08, Department of Computer Science and Engineering, University of Washington, (September 1989).

[6] G. Harrison. Personal communication, (1990).

[7] T. Holman. Processor Element Architecture for Non-shared Memory Parallel Computers, Ph.D. Thesis, Department of Computer Science, University of Washington, (1988).

[8] T. LeBlanc. Shared Memory Versus Message-Passing in a Tightly-Coupled Multiprocessor: A Case Study, *International Conference on Parallel Processing*, (1986) pp. 463-466.

[9] T. LeBlanc, M. Scott, and C. Brown. Large-Scale Parallel Programming: Experience with the BBN Butterfly Parallel Processor. *Proceedings of the ACM/SIGPLAN PPEALS*, (July 1988) pp. 161-172.

[10] P. Nelson. Parallel Programming Paradigms, Ph.D. Thesis, Department of Computer Science, University of Washington, (1987).

[11] T. Lovett and S. Thakkar. The Symmetry Multiprocessor System, *International Conference on Parallel Processing*, (1988) pp. 303-310.

[12] J.Saltz, V. Naik, and D. Nicol. Reduction of the Effects of the Communication Delays in Scientific Algorithms on Message Passing MIMD Architectures," *SIAM J. Sci. Statist. Computing*, vol 8, number 1, (January 1987) s118-s134.

[13] L. Snyder. Parallel Programming and the Poker Programming Environment. *Computer*, (July 1984) pp. 27-36.

[14] L. Snyder. Type Architectures, Shared Memory, and the Corollary of Modest Potential, *Annual Review of Computer Science*, (1986).

```
Spawn P tasks:
task(i,j) executes:
    init(i,j);
    jacobi(i,j);
    printResults(i,j);

init(i,j)
    int diff, delta = 0;
    Allocate current_{i,j}[] and next_{i,j}[] matrices;
    Initialize current_{i,j}[];

jacobi(i,j)
    repeat
    {
        /* send and receive edge values */
        send edge values to N, E, W, S neighbors;
        receive edge values from N, E, W, S neighbors;

        /* compute local averages */
        for each element in next_{i,j}[]
        {
            next_{i,j}[x,y] = (current_{i,j}[x+1,y]+ current_{i,j}[x-1,y]+
                               current_{i,j}[x,y+1]+ current_{i,j}[x,y-1])/4;
            diff = |next_{i,j}[x,y] - current_{i,j}[x,y]|;

            if (delta < diff)
                delta = diff;
        }

        /* find maximum delta */
        if not leaf
            receive delta from children;
        if not root
            send max(delta, children's delta) to parent;

        /* broadcast maximum delta */
        if not root
            receive delta from parent;
        if not leaf
            send delta to children;

        Swap (current_{i,j}, next_{i,j});
    } until delta < tolerance;
```

Figure 5: Algorithm for nsm1

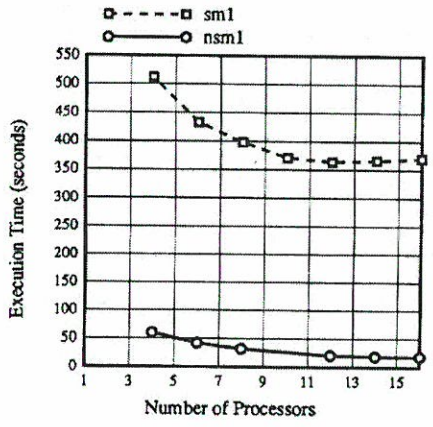

Figure 6: Jacobi on the Butterfly (N=1M)

```
constant tolerance = δ;
global int delta = 0;

Spawn P tasks:
task(i,j) executes:
    init(i,j);
    jacobi(i,j);
    printResults(i,j);

init(i,j)
    int diff = 0;
    Allocate current_{i,j}[] and next_{i,j}[] matrices;
    Initialize current_{i,j}[];

jacobi(i,j)
    repeat
    {
        Acquire non-local values for current_{i,j}[];

        /* compute local averages */
        for each (x,y) in next_{i,j}[];
        {
            next_{i,j}[x,y] = (current_{i,j}[x+1,y]+ current_{i,j}[x-1,y]+
                               current_{i,j}[x,y+1]+ current_{i,j}[x,y-1])/4;
            diff = Max (delta, |next_{i,j}[x,y] - current_{i,j}[x,y]|);
        }

        /* update delta */
        lock delta;
        if (delta < diff)
            delta = diff;
        unlock delta;

        Swap (current_{i,j}, next_{i,j});
    } until delta < tolerance;
```

Figure 7: Algorithm for smo

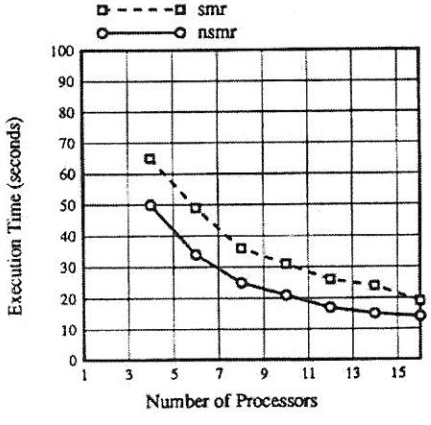

Figure 8: Jacobi on the Butterfly (N=1M)

Figure 9: Allocation By Row vs. Allocation by Square Submatrices

```
Spawn P tasks:
task(i,j) executes:
    init(i,j);
    multiply(i,j);
    printResults(i,j);

init(i,j)
    int diff = 0;
    Allocate rows of A_{i,j}[];
    Allocate columns of B_{i,j}[];
    Allocate columns of B_{src}[];
    Allocate rows of C_{i,j}[];
    Initialize A_{i,j}[] using "bucket brigade";

multiply(i,j)
    for each row r in A_{i,j}[]
        for each column c in B_{i,j}[]
            C_{i,j}[r,c] = dot product (row r, column c);

        /* form logical ring */
        for (k=1; k<P; k++)
        {           /* shift loop */
            send columns of B_{i,j}[] to task_{dest},
                where RingPosition(task(i,j)) - RingPosition(task_{dest}) = k
            receive columns of B_{src}[] from task_{src},
                where RingPosition(task_{src}) - RingPosition(task(i,j)) = k;

            for each row r in A_{i,j}[]
                for each column c in B_{src}[]
                    C_{i,j}[r, c'] = dot product(row r, column c);
        }
```

Figure 10: Algorithm for matrixNSM

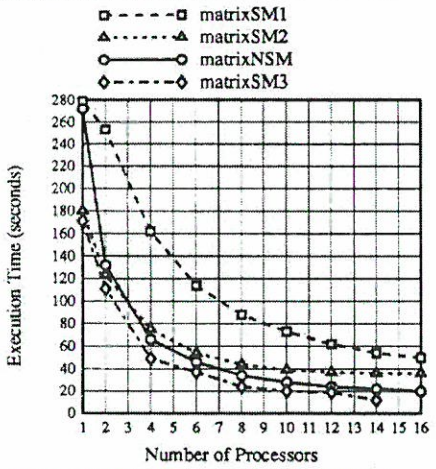

Figure 11: Matrix Multiply on the Butterfly

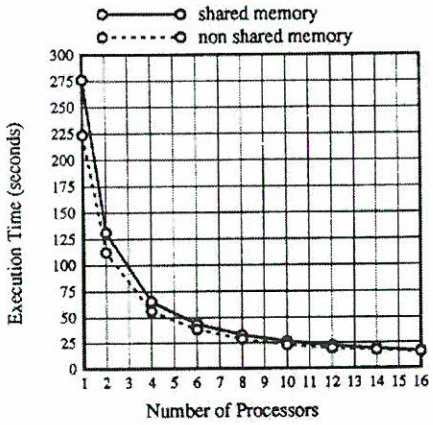

Figure 12: Matrix Multiply on the Symmetry

A1	A2	x	B1	B2	=	A1xB1 + A2xB3	A1xB2 + A2xB4
A3	A4		B3	B4		A3xB1 + A4xB3	A3xB2 + A4xB4

Figure 13: Divide and Conquer Matrix Multiply

Table 1
Jacobi on the Butterfly - 4 Processors
Time in seconds

Program	N = 256K	N = 1M
nsm1	15	57
sm1	128	512
nsmo	13	51
smo	16	65
nsm9	22	88
sm9	26	101
nsmr	13	50
smr	17	65

Table 2
Jacobi on the Butterfly - 16 Processors
Time in seconds

Program	N = 256K	N = 1M
nsm1	4	15
sm1	94	370
nsmo	4	14
smo	5	19
nsm9	6	24
sm9	8	27
nsmr	4	14
smr	5	19

Table 3
Jacobi on the Sequent - 4 Processors
Time in seconds

Program	N = 256K	N = 1M
sm_s	12.5	50.3
nsm_s	10.2	40.6
$nsm1_s$	10.5	41.9

Table 4
Jacobi on the Sequent - 16 Processors
Time in seconds

Program	N = 256K	N = 1M
sm_s	3.2	12.6
nsm_s	2.6	10.3
$nsm1_s$	2.6	10.6

Table 5
Divide and Conquer Matrix Multiply on the Butterfly
16 Processors
Time in seconds

Program	8 x 8	16 x 16	32 x 32	64 x 64
sm	0.46	0.82	3.86	did not finish
nsm	0.59	0.56	0.60	1.15