Viability of Multithreading on Networks of Workstations

Hantak Kwak[1], Ben Lee[1], and Ali R. Hurson[2]

[1] hantak, benl@ece.orst.edu
Department of Electrical and Computer Engineering
Oregon State University
Corvallis, OR 97331

[2] hurson@cse.psu.edu
Computer Science and Engineering Department
The Pennsylvania State University
University Park, PA 16802

Abstract. Recent trend in high-performance computing focuses on networks of workstations (NOWs) as a way of realizing cost-effective parallel machines. This has been due to the availability of powerful wide-issue processors, high-speed networks, and software infrastructure systems. Due to its distributed nature, message-passing has been the choice of communication model for NOWs. This paper, however, examines the viability of using multithreading on NOWs. A matrix multiplication algorithm was studied by simulating a shared-memory abstraction on top of Parallel Virtual Machine (PVM) to characterize the behavior of multithreading. Our experiments indicate the performance of multithreading, with a small number of threads per processor, is very comparable to that of programs written using message-passing. Our studies also show multithreading has an added advantage over message-passing in that it is relatively insensitive to initial data distribution.

1 Introduction

Over the past several years, distributed computing using networks of workstations (NOWs) has gained wide acceptance for both scientific and general purpose applications. Distributed computing employs powerful workstations, or nodes, connected by high-speed local area networks (LANs). A software infrastructure system provides the capability to emulate virtual parallel machines with efficiency ranging from moderate to high. Therefore, it is possible to build low-cost parallel machines as an alternative to more expensive Massively Parallel Processors (MPPs). For example, Parallel Virtual Machine (PVM) supports a heterogeneous parallel computing environment with message-passing [7]. In the message-passing model, a parallel machine is viewed as a collection of processors, or nodes, where the memory is disjoint and distributed among the nodes. Although the message-passing model is suitable and efficient for many applications,

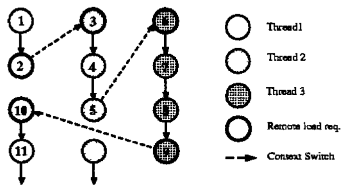

Fig. 1. An example of multithreading. Numbers indicate the order of execution. It is assumed that the remote load operation at node 2 is completed before the context-switch occurs from node 9 to node 10.

```
Matrix_Multiplication(A,B,C)
for i=0 to n-1
    for  j=0 to n-1
        C[i,j]=0
        for k=0 to n-1
            C[i,j]=C[i,j]+A[i,k]×B[k,j]
        end
    end
end
```

Fig. 2. A serial matrix multiplication algorithm for two matrices.

it is often hard for a programmer to manage a large number of concurrent tasks in a parallel program. Moreover, because the distributed nature of the memory is exposed to the programmer, the programmer has to be fully aware of when and how to communicate with other nodes.

To alleviate this problem, many researchers have turned to distributed shared memory (DSM) as a way of providing a shared-memory abstraction on top of message-passing. In DSM, memory is physically distributed among the nodes, but a software system provides a virtual global memory space for ease of programming by eliminating the need to explicitly specify synchronization and communication requirements among nodes.

Although the ease of programming is one of DSM's primary assets, the shared-memory abstraction inevitably leads to performance degradation due to long and unpredictable memory latency [1]. A memory latency occurs when a miss in the local memory requires a request/reply to/from the remote node. To tolerate memory latency, multithreading can be used where a processor maintains a pool of ready threads and a context-switch occurs to a new thread of computation after a remote memory request is sent out as shown in Figure 1. This effectively masks long and unpredictable latencies due to remote loads. However, in order for multithreading to be effective, a number of interrelated issues must be carefully considered. Issues such as the number of contexts, thread run-length, thread scheduling, and granularity of threads have to be carefully considered to provide the best performance for a given architecture. Therefore, this paper studies the viability and effectiveness of multithreading in a networked computing environment, and its performance is compared to the message-passing model.

The paper is organized as follows: Section 2 provides a brief discussion of multithreading. Section 3 discusses parallel matrix multiplication algorithms and presents analytical models that characterize their performance using both message-passing and multithreaded execution models. Section 4 provides simulation results of the two execution models. Finally, Section 5 provides a brief conclusion.

2 Analytical Models for Matrix Multiplication

Matrix multiplication is a simple yet widely used algorithm in many scientific and engineering applications. Matrix multiplication algorithm is well structured in the sense that elements of the matrices can be evenly distributed to the nodes and communications among the nodes have a regular pattern. Therefore, exploiting data parallelism based on message-passing is very suitable for solving the matrix multiplication problem.

Figure 2 shows a sequential version of an $n \times n$ matrix multiplication algorithm consisting of three nested loops. As can be seen, the complexity of the algorithm is n^3. Thus, assuming each pair of multiplication/addition in the inner-loop requires a time of c, the total sequential execution time is given by $T_s = cn^3$. Our motivation for studying the matrix multiplication algorithm is to see how its multithreaded version compares to the message-passing counterpart in a networked computing environment. Therefore, the following two subsections derive the analytical models for both message-passing and multithreaded versions of the matrix multiplication algorithm.

2.1 Matrix Multiplication using Message-Passing

For message-passing, nodes need to communicate data among different parts of the program. For a large parallel system, this exchange of data introduces large communication delays during the execution of a program. Thus, proper implementation of communication operations is important to achieve an efficient execution based on message-passing. There are a few basic communication patterns that frequently appear in various parallel algorithms, e.g., one-to-one, one-to-all broadcast, all-to-all broadcast, and shift with wrap-around [4]. To simplify the development of a communication model of one-to-one communication used in matrix multiplication, we focus only on two major components of the communication cost: *startup cost*, t_s, and *transmission cost*, t_w. Startup cost consists of the time to setup a network channel between the source and destination nodes, the time to allocate a buffer space, and the time to package the messages. Transmission cost is the time required for a message of unit length to travel from one node to the other (i.e., t_w =1/bandwidth). Based on this, one-to-one communication between two processors takes $t_s + t_w m$, where m is the message length.

Data distribution for matrix multiplication can be divided into three classes: element-wise, row or column-wise, and block distribution. In this paper, we implement a simple row/column-wise distribution to study the performance of both message-passing and multithreaded execution models. We have experimented with other algorithms such as Cannon's and Fox's algorithms [4], but found that their communication requirements do not map well to a LAN environment and thus resulted in inferior performance. Assume that the matrices A and B are partitioned into p number of $\frac{n}{p} \times n$ row-stripped and column-stripped submatrices respectively, where p is the number of processors. The processors are labeled from P_0 to P_{p-1} and the submatrices A_i and B_i are initially assigned to P_i (for

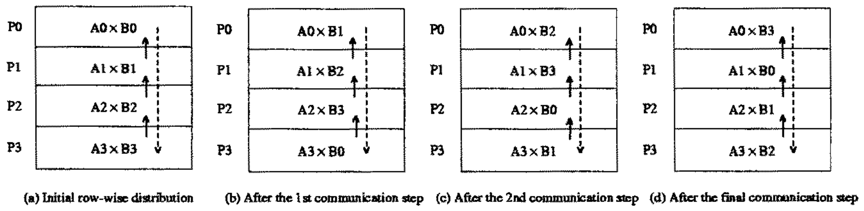
P0
P1
P2
P3
A0×B0
A1×B1
A2×B2
A3×B3
A0×B1
A1×B2
A2×B3
A3×B0
A0×B2
A1×B3
A2×B0
A3×B1
A0×B3
A1×B0
A2×B1
A3×B2
(a) Initial row-wise distribution
(b) After the 1st communication step
(c) After the 2nd communication step
(d) After the final communication step

Fig. 3. Data distribution among four processors. Each processor has submatrices of A and B. Submatrices of B are shifted with a wrap-around in each communication step.

$0 \leq i \leq p-1$) as shown in Figure 3(a). To compute C_i, every processor requires all p submatrices B_k (for $0 \leq k \leq p-1, k \neq i$), which requires excessive memory if all the submatrices are duplicated in each processor. To avoid this problem, each row of submatrices B_k is systematically shifted so that every processor gets a new B_k from its neighbors on each communication step. Therefore, the basic communication step involved in the simple row/column-wise matrix multiplication is to rotate all B_k (for $0 \leq k \leq p-1$) by one step up with a wrap-around in each communication step as depicted on four processors shown in Figure 3. Since the communication in a static network is done sequentially, each communication step has to be done in p sequences.

The algorithm proceeds by having every processor perform multiplication/addition on its local submatrices before each communication step. Each processor also has to perform $p-1$ number of communication steps before completing the matrix multiplication, requiring a total of $p(p-1)(t_s + t_w \frac{n^2}{p})$ time for communication. It is possible to reduce the total number of communication steps to $p-1$ if one-to-all broadcast is used. However, this means the exchange of all submatrices has to be completed before the computation can proceed, and therefore extra memory has to be provided on each node to temporarily store all the submatrices. For the computation part, every processor performs multiplications on its local $\frac{n}{p} \times n$ submatrix, which takes $c\frac{n^3}{p^2}$ time, and the computation has to be repeated p times. Therefore, the total computation time for each processor is $c\frac{n^3}{p}$. The overall parallel run-time for the message-passing version is then given by

$$T_{m-p} = c\frac{n^3}{p} + p\,(p-1)\left(t_s + t_w\frac{n^2}{p}\right). \tag{1}$$

2.2 Matrix Multiplication using Multithreading

For the multithreaded version, we assume the initial distribution of matrices A and B is the same as in the message-passing case. Also, each $\frac{n}{p} \times n$ submatrix of A_i and B_i are further partitioned into n_{th} threads per processor. Therefore, each thread consists of an $\frac{n}{n_{th}p} \times n$ submatrix. There are four basic operations performed in a thread: *request send*, *request service*, *computation*, and *context-switch*. First, each processor P_i sends out a request for $B_{j,k}$ (for $0 \leq j \leq p-1$,

$j \neq i$ and $0 \leq k \leq n_{th} - 1$) of size $\frac{n}{n_{th}p} \times n$ to the remote processor P_j, where j is the processor number and k represents the k^{th} thread (i.e., request send). While this remote access is pending, each processor performs the following operations (see Figure 4): (1) polls to see if a request from other processors has arrived and if there are any, services it (first request service); (2) performs multiplication/addition on a thread that resides on its local memory (computation); (3) once again, polls to see if a remote request has arrived during the computation portion and services it if there are any (second request service); (4) polls to determine if its own requested data has arrived; otherwise, processor idles until the data arrives; finally, context-switches to the next thread (context-switch).

For simplicity, assume that t_{cs} is the context-switching cost, and t_r is the time required to send out a request plus the time to check for a remote request. Note that t_r will be very close to $t_s + t_w$ because the request message is of unit length, and detecting a remote request by polling takes minimal time. The optimal execution time for multithreaded matrix multiplication occurs when the communication time is completely masked by the computation. In this case, we can assume that the time to service a request is approximately equal to the startup time, t_s, since the network transfer time (i.e., $t_w \frac{n^2}{n_{th}p}$ will be hidden by the computation as depicted by P_0 in Figure 4. In other words, the total time required for request send and first or second request service will be approximately equal to $t_r + t_s$. Because each thread consists of $\frac{n^2}{n_{th}p}$ elements, and the computation requires $\frac{n}{p}$ number of multiplications/additions for each element in a thread, the total computation time is $c\frac{n^3}{n_{th}p^2}$. Therefore, the run-time of a thread (i.e., granularity) is given as $t_r + t_s + c\frac{n^3}{n_{th}p^2} + t_{cs}$. Every processor will eventually repeat the above step $n_{th}p$ times before the matrix multiplication completes. Considering the fact that remote request and service are not necessary after $n_{th}(p-1)$ steps, the run-time of the multithreaded version can be expressed as

$$\begin{aligned} T_{mt}^{ideal} &= n_{th}(p-1)\left(t_r + t_s + c\frac{n^3}{n_{th}p^2} + t_{cs}\right) + n_{th}\left(c\frac{n^3}{n_{th}p^2} + t_{cs}\right) \\ &= c\frac{n^3}{p} + n_{th}pt_{cs} + n_{th}(p-1)(t_r + t_s) \\ &\approx c\frac{n^3}{p} + n_{th}p(t_r + t_s + t_{cs}). \end{aligned} \qquad (2)$$

The approximation for Equation (2) is obtained assuming $p \gg 1$. Note that Equation (2) is valid only if the granularity is optimal so that the computation time is long enough to mask remote latencies. However, it is more likely that the round-trip time for the remote access takes longer than the computation time since the network speed is very slow compared to the processor speed and/or servicing of the remote request by the remote processor is delayed (i.e., the request is serviced after the local computation has been completed). This can be modeled by considering two different cases. In the first case, the request

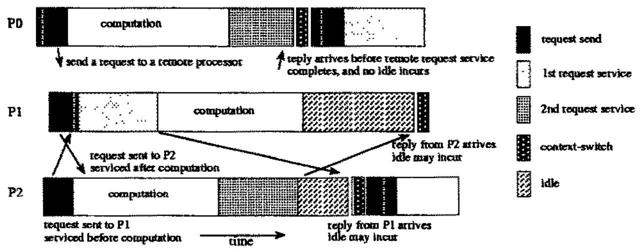
P0
computation
send a request to a remote processor
reply arrives before remote request service completes, and no idle incurs
P1
computation
request sent to P2 serviced after computation
reply from P2 arrives idle may incur
P2
computation
request sent to P1 serviced before computation
time
reply from P1 arrives idle may incur
request send
1st request service
2nd request service
context-switch
idle

Fig. 4. An example of multithreaded execution as a function of time. A thread execution is composed of request-send, request-service, computation, remote receive, and context-switch. Note that when remote access takes longer than the computation idling occurs.

is serviced immediately by the remote processor. This situation is depicted in Figure 4 where P_2 sends a request to P_1 and the request is immediately serviced by P_1 (i.e., the remote processor services the request before performing its local computation). The round-trip time for the remote access can be estimated as $t_r + t_s + t_w \frac{n^2}{n_{th}p}$, where t_r is the time to send/detect a request to/from the remote processor, and $t_s + t_w \frac{n^2}{n_{th}p}$ is the time spent by the remote processor to service the request plus the network transfer time—note that network transfer time may be unbound in reality, but we use the communication model based on $t_s + t_w m$ for simplicity. For this case, the run-time of a thread is composed of the time to send out a request, the round-trip time for the remote access, and the context-switching cost, requiring a time of $t_r + t_s + t_w \frac{n^2}{n_{th}p} + t_{cs}$. Figure 4 shows another case where P_1 sends a request to P_2 and P_2 services the request after its local computation. This situation gives the worst performance and can be estimated by adding computation time to remote access time. Therefore, each thread will require a time of $t_r + t_s + t_w \frac{n^2}{n_{th}p} + c\frac{n^3}{n_{th}p^2} + t_{cs}$. Assuming $p \gg 1$, the total run-time is given as

$$\begin{aligned}
T_{mt} &= n_{th}\,(p-1)\left(t_r + t_s + t_w \frac{n^2}{n_{th}p} + c\frac{n^3}{n_{th}p^2} + t_{cs}\right) + n_{th}\left(c\frac{n^3}{n_{th}p^2} + t_{cs}\right) \\
&= c\frac{n^3}{p} + n_{th}\,(p-1)\left(t_r + t_s + t_w \frac{n^2}{n_{th}p} + t_{cs}\right) + n_{th}t_{cs} \\
&\approx c\frac{n^3}{p} + n_{th}p\,(t_r + t_s + t_{cs}) + t_w n^2 . \qquad (3)
\end{aligned}$$

Equation (3) shows that the overall execution time depends on the communication time rather than the computation time in the worst case. This situation will occur when the number of threads is too large or the granularity is too small to effectively mask the remote latency. Therefore, the granularity has to be properly determined such that the computation and communication are well balanced to achieve optimum performance.

Table 1. Speedup of multithreaded version vs. sequential version.

Dimension	200	400	600	800	1000
Speedup	2.18	2.81	3.08	3.27	3.45

3 Experimental Results

In the previous section, several key characteristics of message-passing and multithreaded versions of the matrix multiplication algorithm were identified using analytical models. This section presents experimental results of the two versions running on four 100MHz Pentium-based LINUX workstations connected via Ethernet. To compare performance, the simulation programs were developed using PVM. Since PVM does not support multithreading, a run-time system was implemented to provide a virtual global memory space and thread scheduling.

Our simulation program for the multithreaded version allows granularity of threads to be varied by assigning an arbitrary number of threads to each node. The scheduling of threads is software-controlled to exploit locality as much as possible rather than relying the dynamic behavior of the run-time scheduler provided by PVM or TPVM [2,7]. The measured execution time of each simulation includes the time taken for PVM processes to initialize as well as the time to execute the matrix multiplication routine. In addition, the thread context-switching cost is assumed to be 50 μsec [2].

In the message-passing implementation, each processor proceeds first with the multiplication of submatrices and then communicates among all four nodes. For the multithreading implementation, we simulate the shared-memory abstraction by sending out an explicit remote request to prefetch the next submatrices required by the computation part. Also, each processor checks to see if there are any incoming requests before and after the computation step on local submatrices. If a request is detected before the computation, the processor immediately prepares the requested data and sends them out to the requesting nodes. Otherwise, incoming requests are serviced after the computation. After servicing the remote requests and performing computation, each processor checks to see if its own requested data has arrived. If the remote data has arrived, a context-switch occurs to the next thread. If not, the processor waits for the requested data. Table 1 shows the speedup of the multithreaded version over the sequential version as a function of matrix dimension. These results indicate the speedup increases from 2.18 to 3.45 as the matrix dimension increases. A low speedup was obtained for $n = 200$ because each node experiences a large portion of remote latencies due to its relatively fine granularity. On the other hand, the speedup factor was higher for $n = 1000$, indicating that its coarse granularity allows a larger portion of the remote access time to be tolerated.

Figure 5 shows the overall execution time of matrix multiplication for the matrices of size 200×200 through 1000×1000. The results for `MT` were based on five threads per each processor, and the results for `MT-opt` were obtained

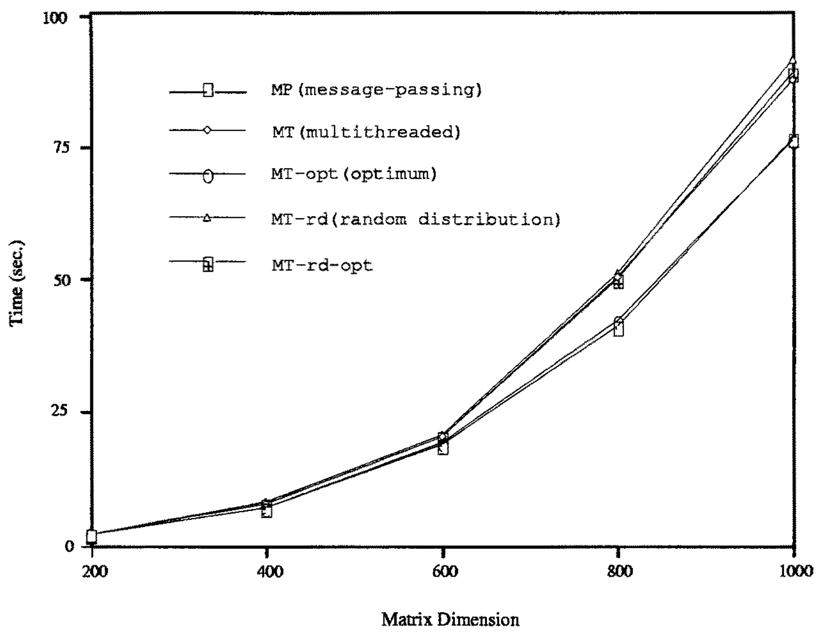

Fig. 5. Execution times of message-passing and multithreaded versions of the row-wise stripped matrix algorithm.

when each processor has an optimum number of threads relative to matrix dimensions, i.e., $n_{th} = 5$ for $n = 200$ and 400, $n_{th} = 20$ for $n = 600$, $n_{th} = 25$ for $n = 800$, and $n_{th} = 50$ for $n = 1000$. It can be seen that message-passing version, MP, gives slightly better performance compared to MT. However, when n_{th} is chosen properly, as in the case of MT-opt, performance is comparable to MP. For example, 50 threads per processor for 1000×1000 matrix multiplication resulted in very close or even better performance than MP. We have also experimented with random distribution of submatrices among nodes for the purpose of investigating the sensitivity of MT or MT-opt to initial data distribution. The resulting graphs MT-rd and MT-rd-opt show that performance suffered less than 5 percent compared to MT. This indicates that matrix multiplication algorithm using multithreading is somewhat insensitive to the initial data distribution.

Figure 6 shows the execution time increases when the number of threads is either very small or large. These effects can be explained as follows: When n_{th} is small, the processor has to idle because more data is needed per thread thus requiring more communication time (i.e., $t_w n^2$ term in Equation 3 becomes dominant and the computation time cannot mask the remote access delay). As n_{th} becomes large, the number of context-switches required increases thereby degrading the performance as indicated by the term $n_{th}p(t_r+t_s+t_{cs})$ in Equation 3. The experimental results indicate that depending on the sizes of the matrices, approximately 5 to 25 threads per processor is enough to achieve the best possible performance.

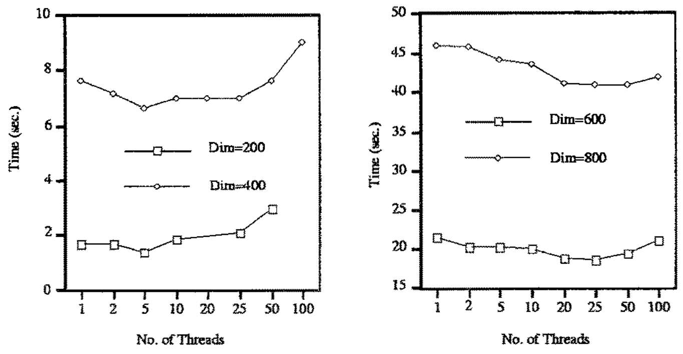

Fig. 6. Execution time vs. number of threads per processor.

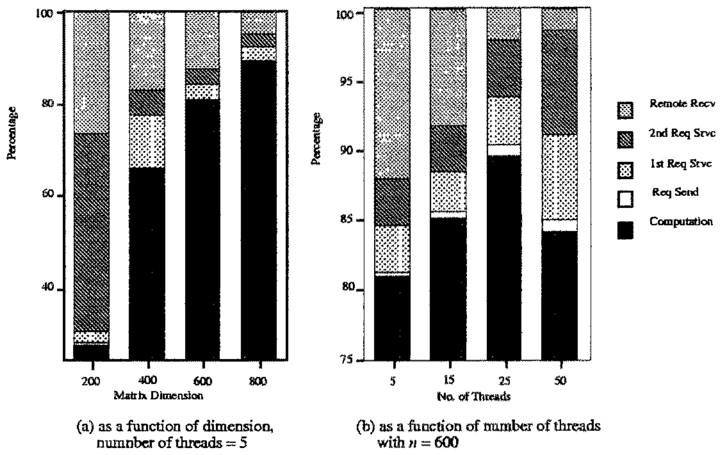

Fig. 7. Percentages of various components of a thread execution.

Figure 7(a) shows the percentages of time spent on each component of a thread as a function of n for $n_{th} = 5$. For illustration purpose, a thread execution is subdivided into the following five components: **`Computation`** is the time spent on multiplications/additions; **`Req Send`** is the time taken to send a request to a remote processor; **`1st Req Srvc`** is the time spent to service the remote request before the computation; **`2nd Req Srvc`** is the time required to service the remote request after the computation (a remote request will be serviced either in **`1st Req Srvc`** portion or **`2nd Req Srvc`** portion, but not both); and **`Remote Recv`** represents the idle time incurred due to remote access. It can also be seen that **`Computation`** portion increases as n increases, thereby tolerating longer remote latencies (e.g., this can be seen by the decrease in **`Remote Recv`** as a function of n). **`Req Send`** takes almost constant amount of time because t_r is independent of matrix dimension. However, the total amount of time spent on sending out a remote request will increase if the number of threads becomes large.

Our experiments also showed there is no definite pattern in terms of whether a remote request is more likely to fall in the `1st Req Srvc` portion or in the `2nd Req Srvc` portion—our experiments show that the chances of an incoming remote request to be serviced before or after the computation is about the same. Figure 7(b) shows the changes in the various components of a thread execution as a function of n_{th} for $n = 600$. It shows that only the components related to remote operation increases as n_{th} increases. Therefore, each processor has to spend more time servicing remote requests, and the performance will degrade.

4 Conclusion

The paper examined the matrix multiplication algorithms to see how the multithreaded execution compares to the message-passing counterpart in a networked computing environment. Our findings indicate that the message-passing execution outperforms its multithreaded counterpart when thread computation cannot effectively mask the remote latency. Also, context-switching and interruptions within a thread to service remote requests adds to the overhead of implementing a multithreaded system. However, the overhead can be reduced if the granularity of threads is properly chosen so that thread computation effectively masks the remote memory latency. We also found that the multithreaded execution of the matrix multiplication is relatively insensitive to initial data distribution. Our future plan is to further study the issues such as thread granularity, thread scheduling, and the effects of non-deterministic data distribution on multithreading.

References

1. Agarwal, A., "*Performance Tradeoffs in Multithreaded Processors,*" IEEE Transactions on Parallel and Distributed Systems, Vol. 3, No. 5, Sept. 1991, pp. 525–539.
2. Ferrari, A., Sunderam, V. S., "*TPVM: Distributed Concurrent Computing with Lightweight Processes,*" can be obtained from http://uvacs.cs.virginia.edu/~ajf2j /tpvm.html.
3. Klaiber, A. C., and Levy, H. M., "*A Comparison of Message Passing and Shared Memory Architectures for Data Parallel Programs,*" IEEE 21st Annual International Symposium on Computer Architecture, 1994, pp. 94–105.
4. Kumar, V., *et al.*, "*Introduction to Parallel Computing,*" The Benjamine Cummings Publishing Company, Inc. 1994.
5. Kodama, Y., *et al.*, "*The EM-X Parallel Computer: Architecture and Basic Performance,*" IEEE International Symposium on Computer Architecture, 1995.
6. Sakane, H., *et al.*, "*Dynamic Characteristics of Multithreaded Execution in the EM-X Multiprocessor,*" Proceedings of 1995 International Workshop on Computer Performance Measurement and Analysis (PERMEAN '95), Beppu Ohita Japan, pp. 14-22.
7. Sunderam, V. S., *et al.*, "*The PVM Concurrent Computing System: Evolution, Experiences, and Trends,*" can be obtained from http://www.ornl.gov:80/pvm.