Data and Workload Distribution in a Multithreaded Architecture

Andrew Sohn[1], Mitsuhisa Sato[2], Namhoon Yoo[3], and Jean-Luc Gaudiot[3]

Abstract

Matching data distribution to workload distribution is important to improve the performance of distributed-memory multiprocessors. While data and workload distribution can be tailored to fit a particular problem to a particular distributed-memory architecture, it is often difficult to do so for various reasons including complexity of address computation, runtime data movement, and irregular resource usage. This report presents our study on multithreading for distributed-memory multiprocessors. Specifically, we investigate the effects of multithreading on *data* distribution and *workload* distribution with *variable* thread granularity. Various types of workload distribution strategies are defined along with thread granularity. Several types of data distribution strategies are investigated. These include row-wise cyclic, *k*-way partial-row cyclic, and blocked distribution. To investigate the performance of multithreading, two problems are selected: highly sequential Gaussian Elimination with Partial Pivoting and highly parallel Matrix Multiplication. Execution results on the 80-processor EM-4 distributed-memory multiprocessor indicate that multithreading can offset the loss that is due to the mismatch between data distribution and workload distribution even for sequential and irregular problems while giving high absolute performance.

1 Dept. of Computer and Information Science, New Jersey Institute of Technology, Newark, NJ 07102-1982, sohn@cis.njit.edu

2 Computer Architecture Section, Electrotechnical Laboratory, 1-1-4 Umezono, Tsukuba-shi, Ibaraki 305, Japan, msato@etl.go.jp

3 Dept. of EE-Systems, EEB336, University of Southern California, Los Angeles, California 90089-2563, {nyoo, gaudiot}@usc.edu

1 Introduction

Distributed-memory multiprocessors give the designer the flexibility of increasing the machine's capability by merely adding more processors to the system. Several research machines have been designed around this distributed-memory/message-passing paradigm: AP1000+ [10], EM-4 [20,21] and EM-X [14,15], FLASH [16], *T:NG [7,18], EARTH [12] and Alewife [1]. Also some commercial machines have been or are being built around this scheme: T3E, SP-2 [2] and Tera [3]. However, their scalability for general-purpose computations remains an open issue. It is indeed known that the major problem which hinders the performance of distributed-memory machines is remote memory latency.

In a distributed-memory machine, data should be distributed so there is no overlapping or copying of *major* data. Typical distributed-memory machines incur much latency, ranging approximately from a few to hundreds of micro seconds for a single remote read operation. The gap between processor cycle and remote memory access time becomes wider, as the processor technology uses more advanced techniques such as pipelining or superscalar and the machine size becomes larger. The latest message-passing machine SP-2 incurs approximately 40 micro seconds to read data allocated to remote processors. Considering that the microprocessors are running at over 66.5 MHz1(5 nano seconds cycle time) for theSP2 590 model, the loss due to a remote read operation is enormous: a single remote read operation would cost 40 micro sec/15 nsec, which is 2667 cycles. If the machine executes one instruction per clock, a single remote read operation would thus amount to the equivalent of 2667 instructions. This does not even take into consideration such architectural schemes as the multiple instruction issue superscalar capability, and yet it shows a tremendous amount of potential computational loss.

Numerous approaches have been reported to reduce/hide/tolerate this communication overhead. The Cilk Project builds a software-based distributed shared memory programming environment using a multithreaded runtime system [4,5]. Threads specified in the high-level language Cilk are automatically scheduled by the runtime system and execute in a machine-independent multithreaded fashion. Data partitioning and alignment [9] is another typical method to reduce communication overhead. Analyzing the behavior of the program, data can be partitioned and allocated to processors such that runtime data movement (read/write) can be minimized. The dynamic migration technique reported in [11] moves computation or data based on runtime-determined statistics. The heuristic used in the approach consists in migrating the computation to where data is whenever the computation requires a remote write. Data migration takes place when there are more than two repeated remote reads.

While data distribution can be carefully designed to minimize the number of remote reads in the course of computation for the given problem, this approach is only applicable to the specific application to which the data partitioning is well tuned. Applications such as computational fluid dynamics change their computational behavior at runtime. The initial data distribution is often found invalid and inefficient after some computations. Problems such as Gaussian Elimination with Partial Pivoting (GEPP) can also be problematic in terms of initial data distribution. Pivot row search in GEPP can often result in irregular data access or movement. Should such situations occur, runtime remote read operations are unavoidable.

Multithreading aims at tolerating remote memory latency through a split-phase read mechanism and context switch [1,3,7,12,15,19]. Threads are usually ended by remote read instructions since those may incur long latencies if the requested data is located in a remote processor [8,13,17]. Through a split-phase read mechanism, a processor switches to another thread instead of waiting for the requested data to arrive, thereby masking the detrimental effect of latency [17]. Performance modeling of multithreading based on an analytical study is presented in [18]. The abstract machine TAM was introduced which exploits parallelism in multiple threads using implicit switching [6]. The Alewife

Multiprocessor provides a hardware support for multithreading [1]. Together with prefetching, block multithreading with four hardware contexts has been shown to be effective in tolerating the latency caused on cache misses for shared-memory applications such as MP3D. Simulated results on the effectiveness of multiple hardware contexts indicated that multithreading is effective for programs which are optimized for data locality by programmers or compilers [23]. The study based on simulated multithreading further indicated that multiple hardware contexts have limited effects on unoptimized programs. Our preliminary study, however, indicated that simple-minded data distribution can give performance comparable to that of the best performing algorithms with hand-crafted data distribution but no threading [21,22].

This paper investigates the effectiveness of the latency tolerance approach. We define various types of data distribution and workload distribution strategies to explicate their performance. Two widely used problems were selected for performance verification: Gaussian Elimination with Partial Pivoting and matrix multiplication. The two problems with all the distribution strategies are implemented on our target machine, the 80-processor EM-4 distributed-memory multiprocessor [19,20]. We begin our discussion in section 2 by giving a brief introduction to multithreading and architectural support for multithreading. Section 3 introduces data distribution methods used in this study with a brief description of Gaussian Elimination with Partial Pivoting and matrix multiplication. Section 4 describes various workload distribution strategies for the two problems. Section 5 lists execution results and evaluates the performance of multithreading. The last section concludes this report.

2 Multithreading

2.1 The principle

A thread is a set of instructions which are executed in sequence. A multithreaded execution model exploits parallelism across threads to improve the performance of multiprocessors [6,13,17]. Threads are usually delimited by remote read instructions which may incur long latency if the requested data is located in a remote processor. Through a split-phase read mechanism, a processor switches to another thread instead of waiting for the requested data to arrive, thereby masking the detrimental effect of latency. Figure 1 pictorially describes the basic principle of multithreading.

Processor 0, P_0, has three threads, T0, T1,and T2, ready to execute in the ready queue. P_0 indicates that T0 is currently being executed which is indicated by a thick dark line. P_0 starts executing the first thread, T0. As T0 is executed, a remote read operation is reached, denoted by a dotted line. The processor switches to T1 while the remote memory

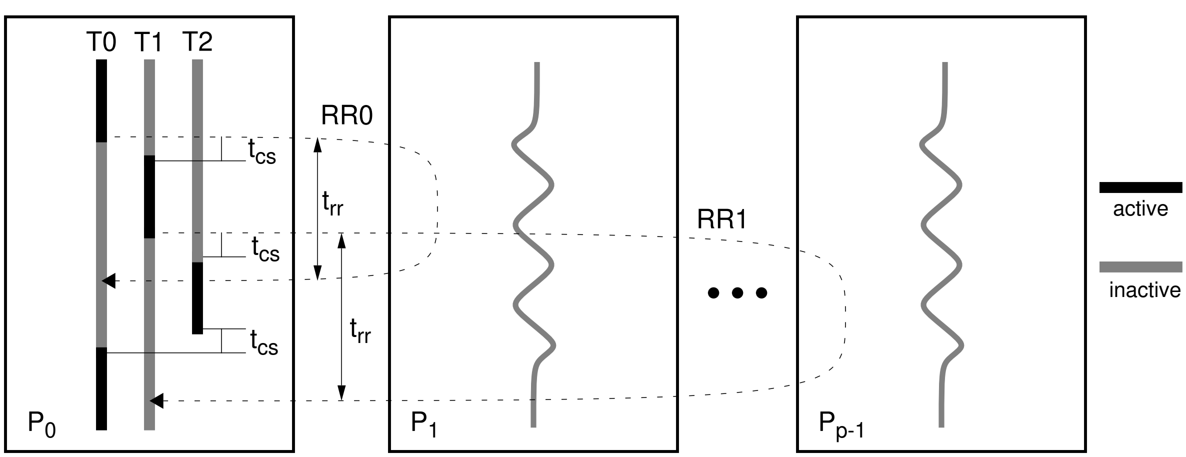

Figure 1: Multithreading on p processors. t_{cs} = context switch time, t_{rr} = remote read time, RR = remote read.

read request RR0 is pending. The processor again switches to T2 when another remote memory read occurs in T1. After T2 completes, T0 can resume its execution assuming the requested data has arrived.

There are several important parameters which characterize the performance of multithreading: (1) the number of threads per processor, (2) the number of remote reads per thread, and (3) the number of instructions in a thread, or thread granularity, (4) context switch cost, (5) remote read latency, and (6) remote read servicing mechanism. The number of active threads determines the amount of parallelism and is often bound by hardware. The number of remote read operations can determine the frequency of thread switching. Thread granularity is determined by the number of instructions per thread. While there is no clear agreement on thread granularity, fine-grain threading typically refers to a thread of tens of instructions while coarse-grain threading refers to a thousand or more instructions per thread.

Two types of context switch are possible: *explicit* switching and *implicit* switching. Explicit switching uses a thread per activation frame while implicit switching uses multiple threads per activation frame. Explicit switching does not require register sharing while implicit switching does since multiple threads may simultaneously exist in an activation frame. EM-4 supports explicit switching. A remote read operation causes the suspension of a thread and in turn context switches if there is any. This thread switching requires saving registers to memory as no register sharing across threads is allowed. The remote read servicing mechanism can be an important factor. While machines such as the T3E and the AP1000+ service remote read requests concurrently with program execution, the EM-4 treats a remote read as another 1-instruction thread which consumes processor cycles. This consumption adversely affects the performance. Details on remote read requests as well as context switching can be found in [20].

2.2 Architectural support for multithreading in the EM-4

The EM-4 distributed-memory multiprocessor supports multithreading both in hardware and software at the instruction level. The machine adopts two models of execution. The high-level is based on data-flow principles of execution while the lower level operates with conventional register-based control-flow principles of execution. The unit on which the two levels of execution operate is a two-word-sized packet. The first word of a packet contains the destination address whereas the second contains the return address (often called *continuation*) for a remote memory read or data for a remote memory write. A remote read operation generates a packet with a destination address and a return address while a remote write operation generates a packet with a destination address and data. Architectural supports for multithreading include autonomous function spawning by packets and hardware scheduling of function invocation.

Packets arriving from the network are buffered in the packet queue. As a packet is read from the packet queue, a thread of instructions specified by the address portion of the packet is invoked along with the one-word data. Once a thread is invoked, it will run to completion if it encounters no remote memory operations. If the thread encounters a remote memory operation, it will be immediately suspended after the remote read request is sent out. Should this suspension occur, any register values currently being used for the thread will be saved in an activation frame associated with the thread for resumption upon the return of the outstanding remote memory operation. The completion or suspension of a thread causes the next packet to be automatically dequeued from the packet queue, resulting in a thread switch. Network packets can be interpreted as dataflow tokens. Dataflow tokens have the option of *matching* with another token on arrival. Unless both tokens have arrived, the token will be saved in memory, which allow other packets in the FIFO queue to be processed.

Compiler supports for multithreading include explicit context switch and global address space. The current compiler supports C with thread library. Programs written in C with the thread library are compiled into explicit-switch threads [20]. The EM-4 recognizes two storage resources, namely, template segments and operand segments. The compiled codes of functions are stored in template segments. Invoking a function involves allocating an operand segment as an activation frame. The caller allocates the activation frame, deposits the argument value(s) into the frame, and sends its continuation as a packet to invoke the caller's thread. The first instruction of a thread operates on input tokens, which are loaded into two operand registers. The registers can hold values for one thread at a time. The current implementation allows no register sharing across threads, thus no implicit-switching support (which is currently being investigated). The caller saves any live registers to the current activation frame before a context-switch. The continuation packet sent from the caller is used to return results as in a conventional call. The result from the called function resumes the caller's thread by this continuation.

The level of thread activation/suspension can be nested and arbitrary. Activation frames (threads) form a tree rather than a stack, reflecting a dynamic calling structure. This tree of activation frames allow threads to spawn one to many threads on processors including itself. The level of thread activation/suspension is limited only by the amount of system memory. Other compiler support for multithreading include a global address space for remote reads/writes. Remote reads/writes are implemented through packets. Each remote read/write invokes a packet handler to perform the actual operation. A remote memory access packet uses a global address which consists of the processor number and the local memory address of the selected processor. Details on support for multithreading are discussed in [19,20]. Table 1 summarizes some of the measured execution time for primitive operations supported by the EM-4. The time for remote operations is averaged over 80 processors. The time for remote call is based on a call to a null body function. The table also illustrates the relationship between the number of arguments and the time to perform primitive operations. These measurements were taken when the network was lightly loaded.

Number of arguments	0	1	2	3
Local call	1.68	2.16	2.64	3.12
Fork	2.10	2.51	2.98	3.46
Thread invocation	6.24	6.64	7.13	7.60
Remote call	4.75	5.16	5.64	6.12
Remote read	1 - 2 micro second			

Table 1: Time in micro seconds for primitive operations.

3 Data Distribution

3.1 Two problems

Gaussian Elimination with Partial Pivoting (GEPP) is one of the most stable methods to solve a system of linear equations. Its implementation on sequential machines consists of three nested loops as shown in Figure 2. Line 1 indicates the outer-most i-loop which computes a column in each iteration. Line 2 searches for a pivot row. If the current column is not the pivot row, the current row and the newly found pivot row are swapped. Line 5 computes a pivot row, with the element a[i][i] being 1. Lines 6 and 7 show the elimination step. All entries belonging to column *i* are eliminated while entries belonging to other columns are modified.

A sequential implementation of GEPP can be as simple and straightforward as the one shown above. A few minor optimizations can be done to reduce the number of address computations in lines 5 and 7. A *Parallel* implementation of GEPP, however, can be problematic due to the pivot row computation and its 1-to-n broadcasting. Line 5 which computes a pivot row introduces loop-carried dependences, making the i-loop sequential. What makes the GEPP more sequential is the broadcasting of a pivot row to all processors, regardless of data distribution. Note that lines 6 and 7

```
for (i=0; i<n; i++) {
    for (j=i; j<n; j++) if (a[j][i] != 0) break;          /* Search a pivot row */
    if (i != j) swap_row(i,j);                             /* Swap two rows */
    inverse = 1.0/a[i][i];
    for (j=i; j<n; j++) a[i][j] = a[i][j] * inverse;      /* Prepare the pivot row */
    for (j=0; j<n; j++)                                    /* Eliminate other entries in column i */
        if (j != i) for (k=i; k<n; k++) a[j][k] = a[j][k] - a[i][k] * a[j][i];
}
```

Figure 2: A sequential version of Gaussian Elimination with Partial Pivoting

need a pivot row to eliminate and modify the entries in non-pivot rows. This is exactly the reason why we chose this GE with Partial Pivoting for our experimentations. We used several data distribution and workload distribution strategies to analyze the effectiveness of multithreading. Our implementation actually swaps two rows, instead of keeping permutation indices. One of the main reasons the two rows are actually swapped is to avoid any possible load imbalance. If the rows were not swapped, the initial data distribution would no longer be valid, which in turn would make it difficult to match workload distribution to data distribution.

The second problem we selected for experiment is Matrix Multiplication. A generic MM is much different from Gaussian Elimination from a multithreading perspective. While GEPP is sequential in nature, MM is inherently parallel with a very high degree of parallelism. While GEPP needs good parallelization for performance improvement, MM needs no apparent parallelization. This is exactly why we chose the two different problems. By using two drastically different problems, we will be able to identify the robustness or weakness of multithreading. In what follows, we will describe how data is distributed for the two problems.

3.2 Data distribution for Gaussian Elimination

The data distribution described in this report assumes that *no processors have the same element*. All processors have a different portion of a matrix. Let p be a number of processors, n be a matrix dimension, and $N = n^2$ be a total number of elements in a matrix. Each processor will get on the average N/p elements. The data distribution is thus concerned with how N elements are to be distributed to the p processors. Figure 3 illustrates the distribution of an 8x8 matrix to six processors. Figure 3(a) shows the distribution of 8 rows to 6 processors, where P_0 and P_1 each get two rows while others get only one row. Figure 3(b) shows a 2-way partial-row distribution where each row is divided into two equal-sized blocks. P_0 to P_3 each get three blocks (or 1.5 rows) while P_4 and P_5 each get two blocks (or 1 row). Figure 3(c) shows 4-way partial-row distribution where each row is now divided into four equal-sized blocks. P_0 and P_1 each get

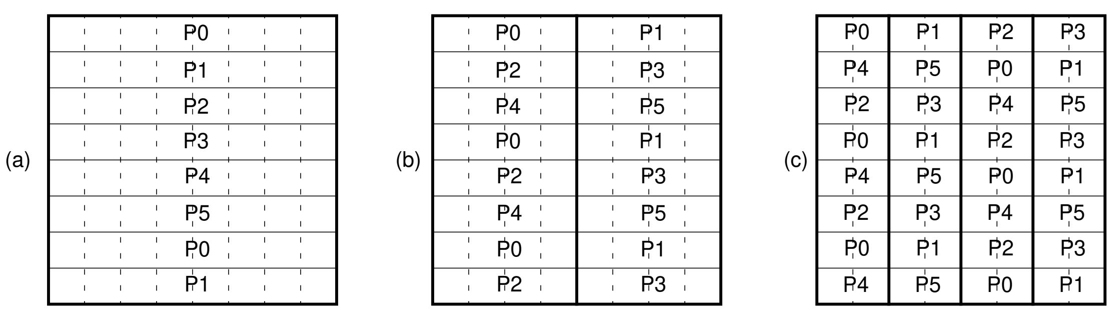

Figure 3: Distribution of an 8x8 matrix to 6 processors for GEPP. (a)row-wise cyclic, (b) 2-way partial-row, (c) 4-way partial-row.

six blocks (or 1.5 rows in total) while P_2 to P_5 each get five blocks (or 1.25 rows). Since the computational front of Gaussian Elimination with Partial Pivot moves along the diagonal of the matrix, a row-wise cyclic distribution may not be as efficient as other methods due to the fact that some processors have one more row than others. The *k*-way partial-row distribution method assures that (1) all processors will get a similar number of elements and (2) more processors will be affected as the computational front changes.

3.3 Data distribution for matrix multiplication

Two types of data distribution methods were used for matrix multiply: row-wise cyclic and blocked. Given two source matrices A and B and a destination matrix C, A and C are distributed *row*-major while B is distributed *column*-major. *No processors have the same element, as done in Gaussian Elimination*. Figure 4 illustrates two distribution methods for an 8x8 matrix to six processors. Figure 4(a) shows a row-major cyclic distribution of 8 rows to 6 processors, where a row is assigned to each processor cyclically. Figure 4(b) shows a blocked distribution, where a processor gets consecutive rows.

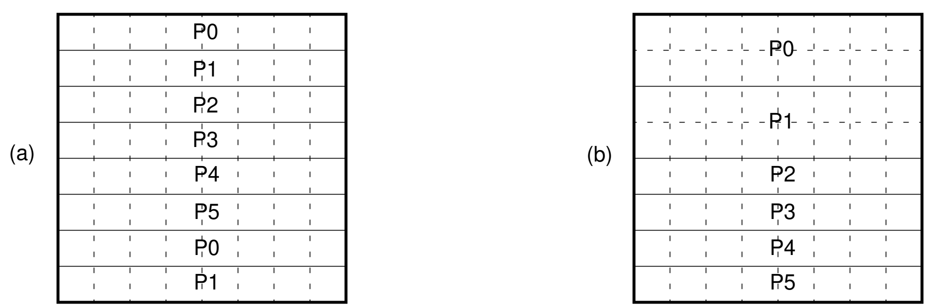

Figure 4: Distribution of an 8x8 matrix to 6 processors. (a) row-wise cyclic, (b) blocked.

4 Workload distribution

4.1 Gaussian Elimination

The workload distribution refers to how computation is distributed to processors. -- We define workload distribution or *thread granularity* in terms of the amount of computation. Three different types of threads are defined in this report: fine-grain, medium-grain, and coarse-grain threads:

- A coarse-grain thread computes *n*/*p* rows and has *no* remote read operations.
- A medium-grain thread computes a whole row.
- A fine-grain thread with degree *d* computes 1/*d* row.

Figure 5 defines three types of threads using the data distribution shown in Figure 3(a). Figure 5(a) shows *six* coarse-grain threads, where the number of threads is the same as the number of processors. Thread 0, or `thd0`, computes row0 and row 6, both of which are allocated to P_0. `Thd1` also computes two rows, row1 and row7, both are again allocated to the same processor, P_1. `Thd2` to `Thd5` each compute only one row. The amount of work each processor computes for coarse-grain threading can differ by a row. Coarse-grain threading allows each processor to invoke only one thread, hence *no* multithreading. Once invoked, the thread runs to completion. *No remote read operations are involved in coarse-grain threading.* Data reading is done before and after the thread invocation.

Figure 5(b) shows eight medium-grain threads assigned to six processors. Each thread computes an entire row. The total number of threads is the same as the number of rows, resulting in the same amount of computation per thread.

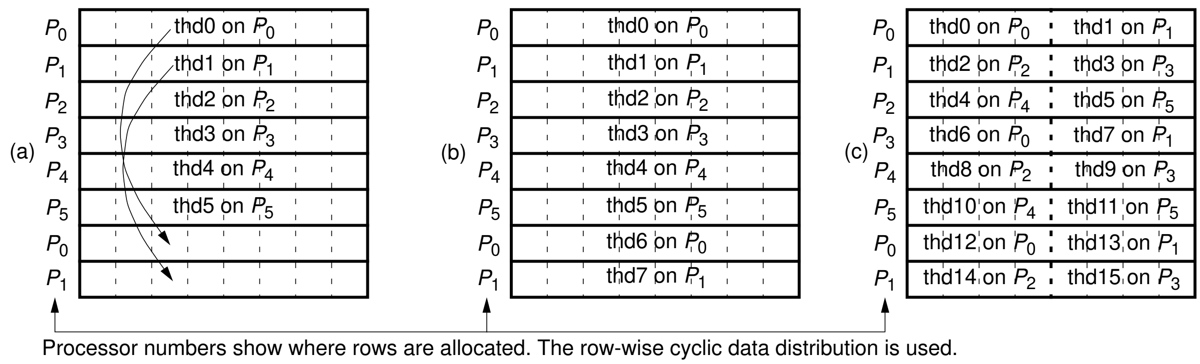

Figure 5: Workload distribution for an 8x8 matrix on 6 processors. (a) coarse-grain, (b) medium-grain, (c) fine-grain threads with $d = 2$.

Threads are assigned to processors in a *round-robin* fashion. P_0 gets two threads, `thd0` and `thd6`. P_1 also gets two threads, `thd1` and `thd7`. All other processors receive one thread each. Medium-grain threading allows each processor to invoke as many threads as assigned. Remote read/write operations can occur before/after thread invocation or in the course of thread execution. Thread execution can interleave and the degree of interleaving is determined by the two parameters p and n.

Figure 5(c) shows fine-grain threading with a degree of 2, i.e., two threads per row. The amount of computation involved in each thread is the same. Thread assignment is done in a round-robin fashion. P_0 gets three threads, `thd0`, `thd6`, and `thd12` while P_5 gets two threads, `thd5` and `thd11`. While it offers a larger degree of interleaving than medium-grain threading, fine-grain threading often requires a larger number of remote read/writes due to a smaller thread size. Note that the figure assumes a 1-row-wise data distribution of Figure 3, where P_0 and P_1 each have two rows allocated while others have only one row allocated. For example, P_0 has row0 and thd0 which will cause no remote read operations. However, due to the round-robin thread distribution, `thd1` which computes the second half of row0 is now assigned to P_1 while row0 is allocated to P_0. This disparity of data distribution and workload distribution incurs four remote reads in the course of executing `thd1` by P_1.

Fine-grain threading with d=4 divides a row into four equal-sized blocks, resulting in four threads per row. Processor assignment is performed in the same fashion as d=2. Table 2 below lists the parameters which characterize the workload distribution. While fine-grain threading with a high degree allows many threads to interleave, there is also a significant disadvantage due to address computation for reading/writing of remote data. In fact, this address computation is one of the serious problems for fine-grain multithreading as we shall see shortly.

	fine-grain with d=8	*fine*-grain with d=4	*fine*-grain with d=2	*medium*-grain	*coarse*-grain
Thread size	1/8 j-loop instance	1/4 j-loop instance	1/2 j-loop instance	1 j-loop instance	n/p j-loop instances
# of thds/processor/i-iteration	$8n/p$	$4n/p$	$2n/p$	n/p	1
Total # of threads	$8n$	$4n$	$2n$	n	p
# remote reads + writes/thread	n/8 reads, n/8 writes	n/4 reads, n/4 writes	n/2 reads, n/2 writes	n†	0*
# instructions for address computation/thread	≈ 50 to 100 instructions/thread			2	0

Table 2: Workload distribution. p = number of processors, n = matrix dimension. *Numbers do not include pivot search and swap*. Loop control instructions for each thread are not included either. †Medium-grain threads may or may not incur remote reads depending on thread assignment. *Coarse-grain threads are always assigned to processors where the needed rows are allocated, and therefore incur no remote reads.

4.2 Matrix Multiplication

Workload distribution for matrix multiplication is the same as those described in Gaussian Elimination (Figure 5) and will not be explained further. We instead list below the number of remote reads and some other pertinent parameters which characterize the workload distribution policies.

	fine-grain with d=8	*fine*-grain with d=4	*fine*-grain with d=2	*medium*-grain	*coarse*-grain
Thread size	1/8 i-loop instance	1/4 i-loop instance	1/2 i-loop instance	1 i-loop instance	n/p i-loop instances
# of thds/processor/i-iteration	$8n/p$	$4n/p$	$2n/p$	n/p	1
Total # of threads	$8n$	$4n$	$2n$	n	p
# remote reads + writes/thread	$(n^2 + n^2 + n)/8$	$(n^2 + n^2 + n)/4$	$(n^2 + n^2 + n)/2$	$n^2 + n^2 + n$	n/p (n2 + n2 + n)

Table 3: Workload distribution. p = number of processors, n = matrix dimension.

5 Execution Results and Discussion

Some of the experimental results are listed in this section. Three types of performance indicators are discussed, including absolute performance of multithreading, impact of multithreading on workload distribution, and the impact of multithreading on data distribution.

5.1 Experimental results

All the distribution methods described in the previous sections are implemented on the EM-4 distributed-memory machine [19,20]. The sequential version is similar to the one shown in Figure 2 with some optimization. It can run on any Unix environment without any modifications. The parallel version is implemented in EM-C, a super set of C, which can run only on the EM-4. Tables 4 and 5 list part of the execution results.

Dimension n	Sequential	row-wise cyclic distribution					k-way partial-row cyclic distribution		
		fine d=8	fine d=4	fine d=2	medium	coarse	fine d=8	fine d=4	fine d=2
80	5.671	1.1	0.6	0.4	0.3	0.3	0.5	0.7	1.2
160	45.954	4.4	2.7	2.0	1.6	1.6	2.0	2.7	4.5
240	155.780	10.0	6.7	5.3	4.4	4.6	5.5	7.3	10.5
320	369.895	18.6	13.4	11.6	9.2	10.1	12.0	15.3	20.1
400	731.578*	31.0	23.6	22.5	17.1	19.1	22.2	27.7	35.5
480	973.057*	44.1	35.3	36.8	27.1	29.0	36.4	45.6	57.0
560	1550.115*	62.7	50.4	57.4	36.5	38.1	56.6	70.6	86.2
640	2328.561*	87.4	73.1	90.4	52.5	57.2	82.2	102.6	124.0

Table 4: Execution time for GEPP on 80 processors. *Single processor execution time based on instruction count.

Dimension n	Sequential	Row-wise cyclic data distribution				Blocked-cyclic data distribution			
		fine d=4	fine d=2	medium	coarse	fine d=4	fine d=2	medium	coarse
80	2.362	0.053	0.062	0.077	0.088	0.051	0.057	0.083	0.083
160	18.787	0.475	0.436	0.347	0.455	0.570	0.456	0.482	0.505
240	63.282	2.133	1.175	0.955	1.320	2.112	1.621	1.520	1.867
320	149.857	4.984	3.623	2.046	2.979	6.251	4.235	3.519	4.351
400	290.200*	13.344	8.220	4.013	5.669	12.555	8.906	7.103	8.216
480	506.564*	23.028	16.461	7.273	9.616	22.317	15.702	12.722	14.245
560	811.098*	40.986	29.537	13.105	15.127	34.776	27.452	21.054	22.461
640	1189.063*	52.209	44.937	22.888	22.521	57.831	44.522	33.314	34.154

Table 5: Execution time for MM on 80 processors. * Single processor execution time based on instruction count.

5.2 Absolute performance

Gaussian Elimination is sequential due to the pivot row search and its broadcasting. The multithreading implementation demonstrates that such a sequential algorithm can also yield high speedup. Figure 6 shows the *absolute* performance of various methods for n=80 to 400. Speedup is defined as the ratio of the execution time on one processor over the execution time on p processors. The graphs show that most implementations can achieve over 30 to 40-fold speedup on 80 processors. Medium-grain multithreading yielded the best speedup reaching over 40-fold. Coarse-grain with *no*-threading is slightly below 40, close to the performance of medium-grain threading. The absolute performance of fine-grain threading was lower than what has been shown in the figure.

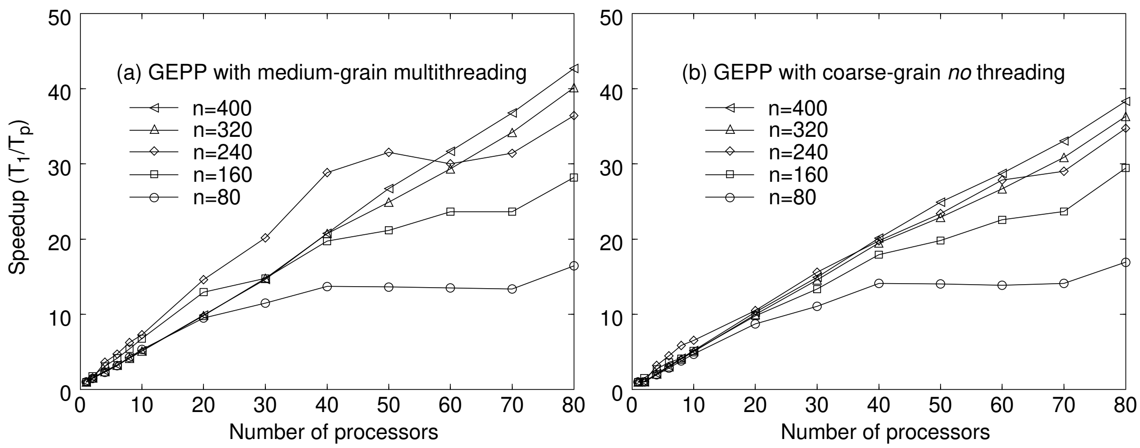

Figure 6: Absolute speedup for Gaussian Elimination. (a) medium-grain, (b)coarse-grain

The *absolute* performance of matrix multiplication is shown Figure 7, where the problem size runs to n = 400. The graphs again show that medium-grain multithreading yields the best performance, reaching near linear speedup as the problem size increases. Coarse-grain with *no*-threading has reached up to 50, which is considerably lower than the performance of medium-grain multithreading. For large problem size of up to 1000, we believe that the we can achieve the commonly reported linear speedup for MM. As we shall discuss in the following sections, we have made no attempt in this study to achieve high speedup as the main purpose of this study is to investigate the behavior of multithreading.

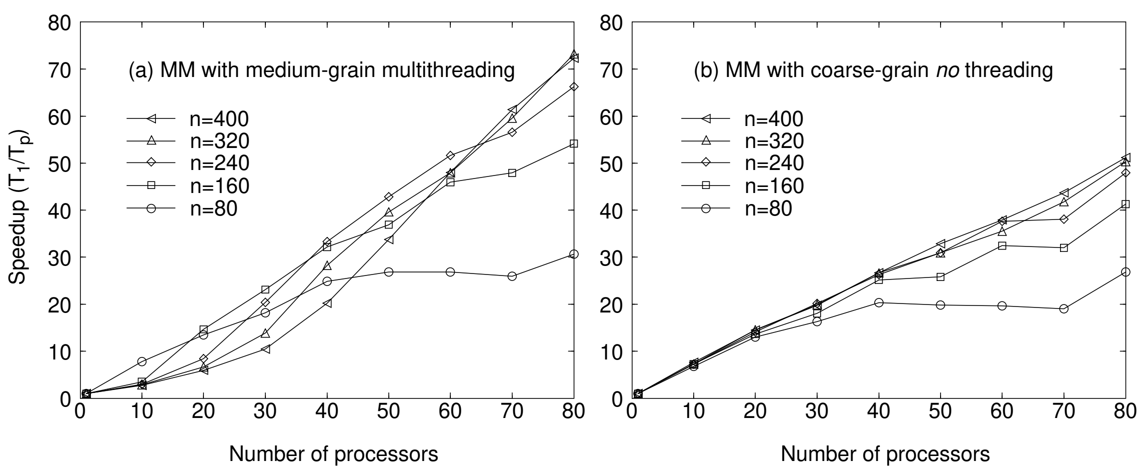

Figure 7: Absolute speedup for Matrix Multiplication. (a) medium-grain, (b)coarse-grain

5.3 Effects of multithreading on workload distribution

Figure 8 shows the performance of multithreading for Gaussian Elimination. Performance is measured against coarse-grain *no*-threading. Figure 8(a) shows the performance of multithreading based on *row-wise cyclic* data distribution while Figure 8(b) shows results based on a 2- to 8-way *partial-row cyclic* data distribution. Figure 8(a) indicates that medium-grain multithreading outperforms all other threading regardless of the workload distribution strategies. Fine-grain threading with a degree of 2 appears to perform well for a small problem size of up to n=400. However, as the problem size increases, the performance of fine-grain threading with degree 2 diminishes drastically while that of fine-grain threading with degree 8 increases. In general, fine-grain multithreading shows poor performance compared to medium-grain threading mainly because of the large number of remote reads. Note from Table 2 that medium-grain threading may or may not incur remote reads while fine-grain threading on the average incurs n remote reads/writes per thread, regardless of its granularity.

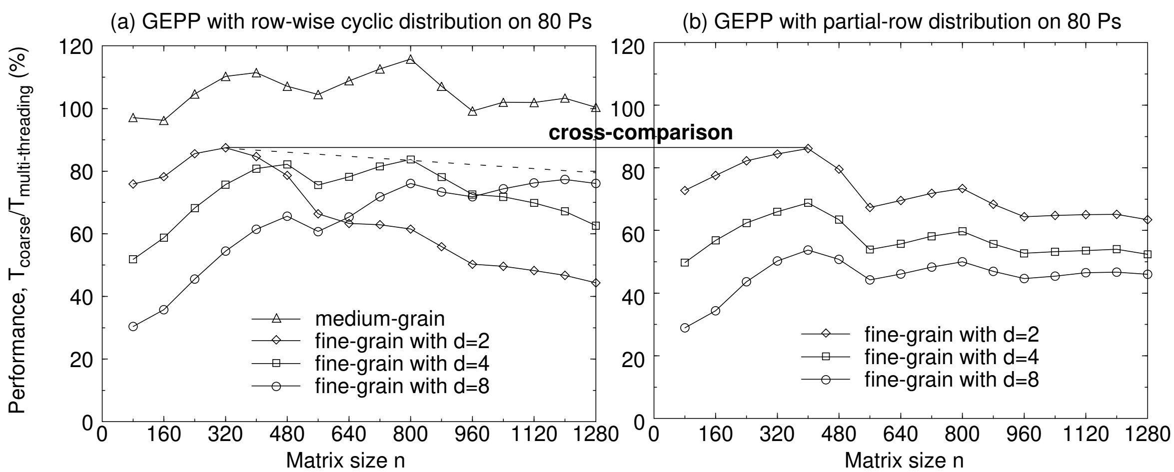

Figure 8: Comparison of workload distribution for GEPP based on (a) row-wise cyclic data distribution, (b) k-way partial-row cyclic data distribution. Multithreading is compared against coarse-grain *no* threading.

It is interesting to note that the performance of fine-grain threading with d=4 becomes better than the one with d =2 after n=640. The same behavior is observed for d=8 which performs better than d=2, as well as 4 for a problem size of over 960. Why does the performance of fine-grain threading fluctuate? The answer lies in the fact that performance of fine-grain threading can be classified into three regions. The first region, n=80 to 400, where performance is directly proportional to thread granularity, i.e., the larger the thread size is, the better the performance is. The main reasons are the twice longer computation time between remote reads for the fixed number of remote reads, and half the number of context switches compared to finer grain threading. The second region, n=400 to 960, gives mixed performance. The one with d=4 outperformed all other fine-grain threading due to the balanced combination of thread size and the number of threads, along with the problem size. The third region, n=960 to 1280, gives performance *inversely* proportional to thread size. The main reason is the finer-grain threading now has enough threads to mask the latency caused by the *increased* number of remote reads. The longer computation which was shown effective in the first region is no longer effective since the number of remote reads outweighs the computation time. The computation time of an element in the elimination step of GEPP consists only of two instructions: floating point multiplication and floating point subtraction. Twice the computation time does not mask a single remote read latency.

Specifically, in the first region the best performance occurs at *n*=320 where fine-grain with *d*=2 performs approximately 87% of coarse-grain no-threading. At *n*=320, the thread size is **160** (*n*/*d*=320/2) for *d*=2, 80 (*n*/*d*=320/4) for *d*=4, and 40 (*n*/*d*=320/8) for *d*=8 (see also Table 2 for thread size computation). At *n*=800 of the second region, fine-grain with *d*=4 performs the best, yielding approximately 83% of coarse-grain no-threading. The thread size at *n*=800 is 100 for *d*=2, **200** for *d*=4 and 400 for *d*=8. For the third region, fine-grain with *d*=8 performs the best, yielding approximately 76% of coarse-grain no-threading. The thread sizes at *n*=1200 are 600 for *d*=2, 300 for *d*=4, and **150** for *d*=8.

Common to the three regions is thread size and number of threads. When the thread size is approximately 200, fine-grain threading gives the peak performance. As the thread size increases to 300 for *d*=4 or even 600 for *d*=2 at *n*=1200, the performance actually decreases because there are not enough threads for context switch which will mask the remote read latency. Should the finest-grain threading perform the best since it has a large number of threads? Experimental results indicate otherwise. A large number of threads does not necessarily give good performance because the context switch time will then become a deciding factor.

A behavior similar to the one discussed above is observed in matrix multiplication. Figure 9 shows the performance of multithreading for MM, where performance is again measured against coarse-grain *no*-threading.

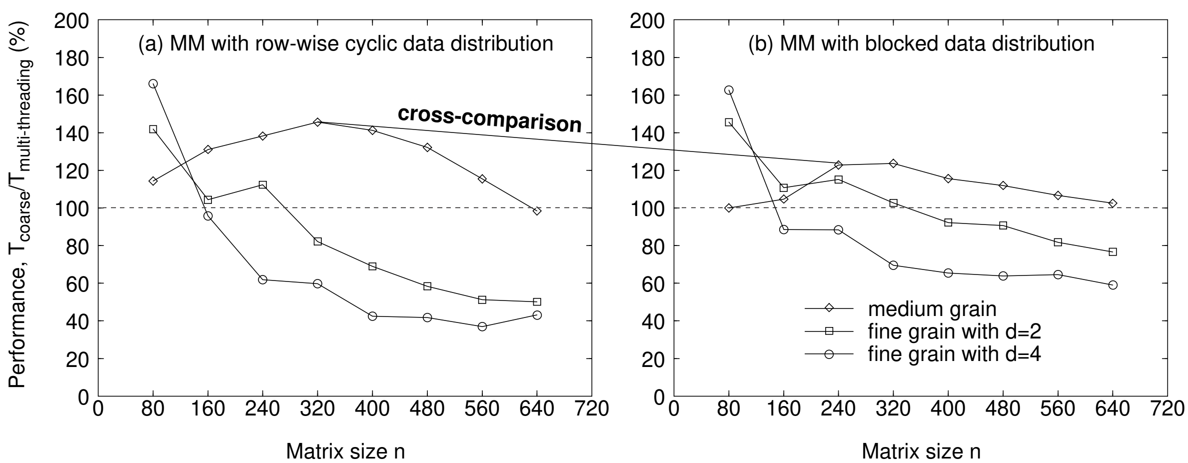

Figure 9: Comparison of data distribution for MM. (a) row-wise cyclic, (b) blocked.

We find in Figure 9 that medium-grain threading shows the best performance while fine-grain threading shows poor performance for the given problem instances. Increasing the number of threads does not necessarily increase the performance due to high cost for context switch. The plots also indicate that multithreading exhibits performance change. The main reason contributing to this performance change is due to the fact that workload distribution is not matched to data distribution, which is discussed below.

5.4 Effects of multithreading on data distribution

Two different data distribution methods are used for each problem. GEPP used row-wise cyclic and partial row-wise cyclic while MM used row-wise cyclic and blocked distribution. Figures 8 and 9 also showed the performance of multithreading across data distribution methods. Note that the solid line which crosses the two figures (labeled *cross-comparison*) clearly indicates that regardless of the data distribution methods, medium-grain threading performs better than other methods. Figure 10 summarizes the comparison of data distribution methods for the two problems. All the plots are based on the execution results on 80 processors.

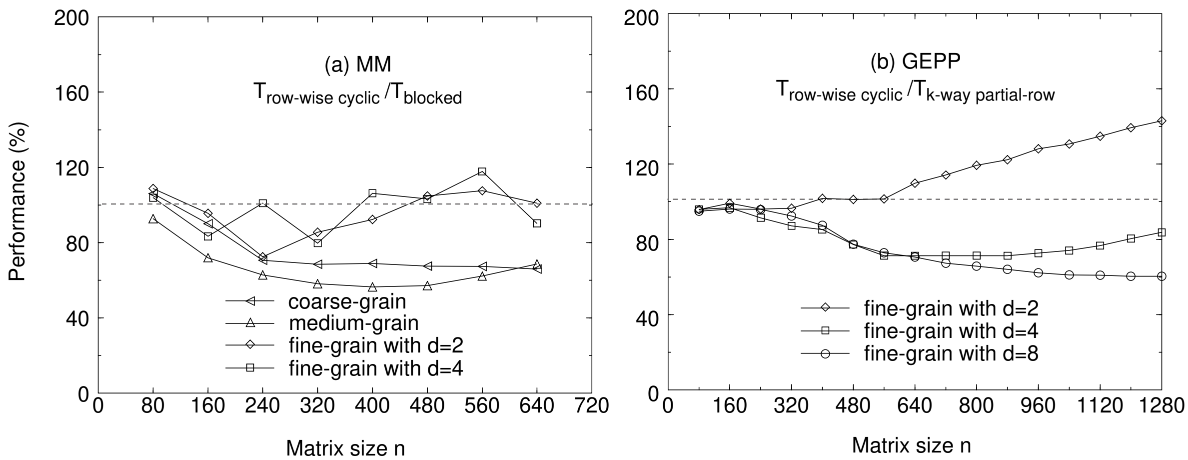

Figure 10: Comparison of data distribution for the two target problems.

Figure 10(a) compares row-cyclic and blocked data distribution with four types of workload distribution methods. Execution time using row-wise cyclic distribution is compared to the one with a *k*-way partial-row data distribution. The figure contrasts multithreading and no-threading. First, coarse-grain no threading gives a 10-40% performance difference when a different data distribution method is used. Second, medium-grain multithreading is less sensitive to a different data distribution, resulting in a 10-30% performance difference. Third, fine-grain multithreading is insensitive to data distribution as it shows essentially the same performance for two different data distributions. It is not surprising that coarse grain no threading gives such sensitivity to data distribution.

Recall the workload distribution of Figure 5, where three types of threading are defined. A coarse-grain thread computes *n/p non-consecutive* rows. Recall also the two data distribution methods defined in Figure 4. We find that a row-wise cyclic distribution completely matches the workload distribution, i.e., a row of A and a column of B are allocated to a processor where a coarse-grain thread which uses the row and the column is assigned. However, blocked distribution which allocates *n/p* consecutive rows/columns of A and B to the same processor does not match the workload distribution. Figure 11 illustrates a case study of coarse grain threading with blocked data distribution:

We find from Figure 11(c) that most threads are experiencing a very large number of remote reads. To be more precise, consider thd5 assigned to processor 5 which computes C5, row 5 of C. Note that both A5 and B5 are allocated

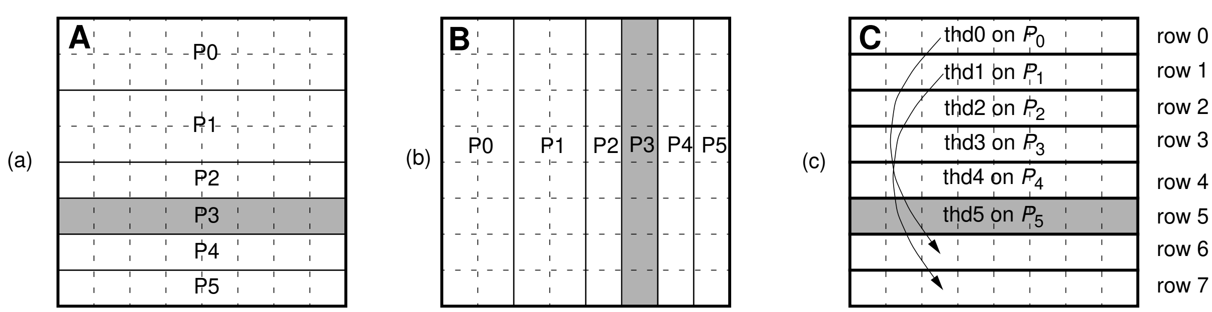

Figure 11: Coarse-grain threading for MM: (a) row-wise blocked data distribution of matrix A, (b) column-wise blocked data distribution of matrix B, (c) row-wise cyclic workload distribution to compute C.

to processor 3, shaded in Figure 11 (a) and (b) while C5 is to be computed by thd5 assigned to processor 5. This mismatch of data and workload distribution requires n remote reads for A and $n(n-1)$ remote reads for B, assuming that there is a *buffer* of size n which temporarily holds A5. If, however, there is no buffer to temporarily hold A5, the number of remote reads increases to $n^2 + n(n-1)$. This report does not assume any buffer to temporarily hold values of rows or columns. All the remote memory reads are done on-the-fly and as-needed-basis; Once read, they are consumed immediately. Is it then not appropriate to use a small buffer of size n to hold a row or a column? Storing the row that was remote-read in a buffer will eliminate the need for remote-reading of the same row in the next round, which in turn will reduce remote read latency. We rejected this type of buffer use because it requires each thread to have a buffer. If there are k active threads, the memory requirement will increase to k-fold. A small number of active threads will be effective since the row that was remote-read can be used until all the corresponding column are exhausted. A large number of threads, however, will essentially defeat the purpose of using multithreading and distributed-memory machines due to extra storage management. Even if a buffer is used to hold a row, the advantage may not be significant because it will force the computation into a *synchronous* mode to avoid buffer corruption.

While coarse-grain no threading experiences a 10-40% performance degradation due to the mismatch between data and workload distribution, medium-grain multithreading shows better performance. Figure 10(a) indicates medium-grain multithreading stays above coarse-grain threading in the presence of a mismatch between data and workload distribution. It is surprising to find that fine-grain multithreading performs well, experiencing almost no difficulties despite the mismatch. The two top curves of Figure 10(a) show that the performance of fine-grain threading remains essentially the same in the presence of different data distribution strategies. We also find a similar behavior in the case of Gaussian Elimination with Partial Pivoting, where fine-grain threading gives mixed performance for different data distribution. It would be too early to conclude that data distribution is not central to obtaining high performance on distributed-memory multiprocessors. However, we believe, based on our results, that multithreading can *lessen* the performance degradation caused by the mismatch between data distribution and workload distribution. We are currently investigating the performance of multithreading on different types of problems to further out insight.

6 Conclusions

Data distribution is known to be a key to obtaining high performance on distributed-memory multiprocessors. While it is possible to adopt a distribution strategy tailored to a particular algorithm and an architecture, we believe it is often not only difficult to do so but more importantly goes against the principles of general purpose parallel computing. This report has presented our study on the effects of multithreading on data and workload distribution. Specifically, we have demonstrated that multithreading can help make performance less sensitive to data distribution. To verify the effectiveness of multithreading, we have developed various types of data and workload distribution methods. Two different types of problems have been selected for experiments. The first one, Gaussian Elimination with Partial Pivoting (GEPP), was selected because of its sequential nature and irregular resource usage due to pivot row computing and its broadcasting. The second problem, Matrix Multiplication (MM), was selected because of its inherently parallel nature and deterministic resource usage.

Workload distribution has been realized in terms of thread granularity. Five types of threads have been defined, including (1) fine-grain threading to compute n/d elements of a row, (2) medium-grain threading to compute a row, and (3) coarse-grain threading to compute n/p rows, where d is degree of fine-grain threading, n is the matrix dimension, and p is the number of processors. Three types of data distribution strategies have been used in the experiments, including *k*-way partial-row cyclic distribution, row-wise cyclic distribution, and blocked distribution. The target machine for our experiments is the 80-processor EM-4 distributed-memory multiprocessor which has a minimal amount of built-in hardware and compiler support for multithreading. All the data and workload distribution strategies have been implemented on the target machine with a matrix dimension of up to 1280.

Experimental results have demonstrated that multithreading can indeed help make performance less sensitive to data distribution. Even when workload distribution does not match data distribution, multithreading has given performance comparable to or even better than when workload distribution nicely matches data distribution. Results have also indicated that increasing the number of threads does not necessarily increase the performance. In fact, a very large number of threads adversely affected the performance. We have found that approximately 30 to 40 fine-grain threads/processor per iteration of *i*-loop gave the best performance for GEPP. Our findings on multithreading can be summarized with the following observations. For the highly sequential problem GEPP, multithreading gave an absolute speedup of over 43 while no-threading gave a 37-fold speedup on 80 processors. Second, medium-grain multithreading has yielded the best performance among various types of workload distribution methods, giving over 115% performance of coarse-grain no-threading. Third, multithreading has given essentially the same performance of coarse-grain no-threading when workload distribution does not match data distribution.

Similar results have been obtained from our highly parallel matrix multiplication. Multithreading has yielded an almost linear speedup while coarse-grain no-threading reached a 50-fold speedup on 80 processors as the matrix dimension reached 400. Medium-grain threading again yielded the best performance among four workload distribution methods, giving up to 140% of the performance of coarse-grain no-threading. Multithreading has given 80-120% performance of no threading when there is a complete mismatch between workload distribution and data distribution. We are currently working on a completely different set of problems, in particular non-numeric problems such as search, sort, and expert systems, to further identify the performance of multithreading.

Acknowledgments

The authors would like to thank the EM-4 and EM-X group members, Mitsuhisa Sato, Yuetsu Kodama, Hirofumi Sakane, Hayato Yamana, Shuichi Sakai, and Yoshinori Yamaguchi, of the Electrotechnical Laboratory for various discussion on multithreading. Andrew Sohn is supported in part by the NASA University Joint Venture in Research Program JOVE NAG8 1114-2 and the Foreign Researcher Program of the Agency of Industrial Science and Technology of the Ministry of International Trade and Industry, Japan.

References

[1] A. Agarwal, R. Bianchini, D. Chaiken, K. L. Johnson, D. Kranz, J. Kubiatowicz, B-H. Lim, K. Mackenzie, and D. Yeung, "The MIT Alewife Machine: Architecture and Performances," in *Proceedings of ACM International Symposium on Computer Architecture,* Santa Margherita Ligure, Italy, June 1995, pp.2-13.

[2] T. Agerwala, J. L. Martin, J. H. Mirza, D. C. Sadler, D. M. Dias, and M. Snir, "SP2 system architecture", in *IBM Systems Journal* Vol. 34, No. 2, 1995

[3] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith, "The Tera computer system," In *Proceedings of ACM International Conference on Supercomputing*, Amsterdam, Netherlands, June 1990, ACM, pp.1-6

[4] R. D. Blumofe, M. Frigo, C. F. Joerg, C. E. Leiserson, and K. H. Randall, "Dag-Consistent Distributed Shared Memory," in *Proceedings of IEEE International Parallel Processing Symposium, Honolulu*, Hawaii, April

1996, pp.132-141.

[5] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: An Efficient Multithreaded Runtime System," in *Proceedings of ACM Symposium on Principles and Practice of Parallel Programming*, Santa Barbara, California, July 1995, pp.207-216.

[6] D. Chiou, B. S. Ang, R. Greiner, Arvind, J. C. Hoe, M. J. Beckerle, J. E. Hicks, and A. Boughton. "StarT-NG: Delivering Seamless Parallel Computing." in *Proceedings of EURO-PAR '95 -- International Conference on Parallel Processing*, Lecture Notes in Computer Science, Stockholm, Sweden, Springer-Verlag, August 29-31, 1995.

[7] D. Culler, S. Goldstein, K. Schauser, and T. von Eicken, "TAM - A Compiler Controlled Threaded Abstract Machine," *Journal of Parallel and Distributed Computing 18*, pp.347-370, 1993.

[8] G. Gao, L. Bic and J-L. Gaudiot (Eds.) Advanced Topic in Dataflow Computing and Multithreading, IEEE Computer society press, 1995.

[9] D. Garza-Salazar and Wim Böhm, "Reducing Communication by Honoring Multiple Alignments", in *Proceedings of ACM International Conference on Supercomputing*, Barcelona, Spain, July 1995, pp. 87-96

[10] K. Hayashi, T. Doi, T. Horie, Y. Koyanagi, O. Shiraki, N. Imamura, T. Shimizu, H. Ishihata, and T. Shindo, "AP1000+: Architectural Support of Put/Get Interface for Parallelizing Compiler," in *Proceedings of ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, San Jose, California, October 1994, pp.196-207.

[11] W. C. Hsieh, P. Wang, and W.E. Wiehl, "Computation Migration: Enhancing Locality for Distributed-Memory Parallel Systems," in *Proceedings of ACM Symposium on Principles and Practice of Parallel Programming*, San Diego, California, May 1993, pp.239-248.

[12] H. H. J. Hum, O. Maquelin, K. B. Theobald, X. Tian, X. Tang, G. R. Gao, P., N. Elmasri, L. J. Hendren, A. Jimenez, S. Krishnan, A. Marquez, S. Merali, S. S. Nemawarkar, P. Panangaden, X. Xue, and Y. Zhu, "A Design Study of the EARTH Multiprocessor," in *Proceedings of ACM/IEEE International Conference on Parallel Architectures and Compilation Techniques*, Limassol, Cyprus, June 1995, pp. 59-68.

[13] R. Iannucci, G. Gao, R. Halstead, and B. Smith (Eds.), Multithreaded Computer Architecture, Kluwer Publishers, Norwell, MA 1994.

[14] Y. Kodama, Y. Koumura, M. Sato, H. Sakane, S. Sakai, and Y. Yamaguchi, "EMC-Y: Parallel Processing Element Optimizing Communication and Computation," in *Proceedings of ACM International Conference on Supercomputing*, Tokyo, Japan, July 1993, pp.167-174.

[15] Y. Kodama, H. Sakane, M. Sato, H. Yamana, S. Sakai, and Y. Yamaguchi, "The EM-X Parallel Computer: Architecture and Basic Performance," in *Proceedings of ACM International Symposium on Computer Architecture*, Santa Margherita Ligure, Italy, June 1995, pp.14-23.

[16] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, E. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy, "The Stanford FLASH Multiprocessor," in *Proceedings of ACM International Symposium on Computer Architecture*, Chicago, Illinois, April 1994, pp.302-313.

[17] R. Nikhil, G. Papadopolous, and Arvind, "*T: A Multithreaded Massively Parallel Architecture," in *Proceedings of ACM International Symposium on Computer Architecture*, Gold Coast, Australia, May 1992, pp.156-167.

[18] R. Saavedra-Barrera, D. Culler, and T. von Eicken, "Analysis of Multithreaded Architectures for Parallel Computing," in *Proceedings of ACM Symposium on Parallel Algorithms and Architectures*, pages 169-178, July 1990.

[19] S. Sakai, Y. Yamaguchi, K. Hiraki, and T. Yuba, "An Architecture of a Data-flow Single Chip Processor," in *Proceedings of ACM International Symposium on Computer Architecture*, Jerusalem, Israel, May 1989, pp.46-53.

[20] M. Sato, Y. Kodama, S. Sakai, Y. Yamaguchi, and Y. Koumura, "Thread-based Programming for the EM-4

Hybrid Data-flow Machine," in *Proceedings of ACM International Symposium on Computer Architecture*, Gold Coast, Australia, May 1992, pp.146-155.

[21] A. Sohn, C. Kim, and M. Sato, "Multithreading with the EM-4 Distributed-Memory Multiprocessor," in *Proceedings of ACM/IEEE International Conference on Parallel Architectures and Compilation Techniques*, Limassol, Cyprus, June 1995, pp.27-36.

[22] A. Sohn, M. Sato, N. Yoo, and J-L Gaudiot, "Effects of Multithreading on Data and Workload Distribution for Distributed-Memory Multiprocessors," in *Proceedings of IEEE International Parallel Processing Symposium, Honolulu*, Hawaii, April 1996, pp. 116-122.

[23] R. Thekkath and S. Eggers, "The Effectiveness of Multiple Hardware Contexts," in *Proceedings of ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, San Jose, California, October 1994, pp.328-337.