Fine-Grain Multithreading with the EM-X Multiprocessor

Andrew Sohn[1], Yuetsu Kodama[2], Jui Ku[1], Mitsuhisa Sato[3], Hirofumi Sakane[2]
Hayato Yamana[2], Shuichi Sakai[2], Yoshinori Yamaguchi[2]

Abstract - Multithreading aims to tolerate latency by overlapping communication with computation. This report explicates the multithreading capabilities of the EM-X distributed-memory multiprocessor through empirical studies. The EM-X provides hardware supports for fine-grain multithreading, including a by-passing mechanism for direct remote reads and writes, hardware FIFO thread scheduling, and dedicated instructions for generating fixed-sized communication packets. Bitonic sorting and Fast Fourier Transform are selected for experiments. Parameters that characterize the performance of multithreading are investigated, including the number of threads, the number of thread switches, the run length, and the number of remote reads. Experimental results indicate that the best communication performance occurs when the number of threads is two to four. FFT yielded over 95% overlapping due to a large amount of computation and communication parallelism across threads. Even in the absence of thread computation parallelism, multithreading helps overlap over 35% of the communication time for bitonic sorting.

1 Introduction

In a distributed-memory machine, data needs to 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 tens of micro seconds for a single remote read operation [2,20]. The gap between processor cycle and remote memory access time becomes wider, as the processor technology improves and rigorously exploits instruction level parallelism. IBM SP-2 incurs approximately 40 μsec to read data allocated to remote processors [2,24]. Considering that the microprocessors are running at over 66.5 MHz (15 nano sec cycle time) for the SP-2 590 model, the loss due to a remote read operation is enormous; A single remote read operation would cost 40 μsec/15 nsec, or 2667 cycles.

Various approaches have been developed to reduce/hide/tolerate communication time, as well as to study communication behavior for general purpose parallel computing [8]. Data partitioning in HPF is a typical method to reduce communication overhead [10]. While data distribution can be carefully designed to minimize the number of remote reads for the given problem, this approach is effective for specific applications where data partitioning can be 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.

Multithreading aims at tolerating memory latency using context switch. Through a split-phase read transaction, a processor switches to another thread instead of waiting for the requested data to arrive, thereby masking the detrimental effect of latency [14]. The Heterogeneous Element Processor (HEP) designed by Burton Smith provides up to 128 threads [21]. A thread switch occurs in every instruction with 100 nsec switching cost. Threads are usually ended by remote read instructions since those may incur long latencies if the requested data is located in a remote processor [14]. The Monsoon data-flow machine developed at MIT switches context every instruction, where a thread consists of a single instruction [15].

The EM-4 multiprocessor, the predecessor of EM-X, provides hardware support for multithreading [17,18,19]. Thread switch takes place whenever a remote read is encountered. Threads can also be suspended with explicit thread scheduling. 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. The Tera multithreaded architecture (MTA) provides hardware support for multithreading [3]. The maximum of 128 threads are provided per processor. Context switch takes place whenever a remote load or synchronizing load is encountered.

An analytic model for multithreading is studied in [16]. The study indicated that the performance of multithreading can be classified into three regions: linear, transition, and saturation. The performance of multithreading is proportional to the number of threads in the linear region while it depends only on the remote reference rate and switch cost in the saturation region. The Threaded Abstract Machine studied by Culler et al. exploits parallelism across multiple threads [7]. Fine-grain threads share registers to exploit fine-grain parallelism using implicit switching.

Simulation 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 [25]. Some experimental results on EM-4, 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 [23]. The Cilk Project builds a software-based distributed shared memory programming environment using a multithreaded runtime system [5]. Threads specified in the high-level language Cilk are automatically scheduled by the runtime system and execute in a machine-independent multithreaded fashion.

This paper investigates the performance of multithreading with the EM-X multiprocessor. Critical parameters in multithreading are investigated, including the number of threads, the run length, the number of remote reads, and the number of switches. The interplay between the parameters is explained with experimental results. Bitonic sorting and Fast Fourier Transform are selected and their multithreaded algorithms are developed. Data and workload distribution strategies are developed to explicate their performance. The ultimate goal of multithreading is to tolerate communication time. In this respect, the experiments are carried out to identify how multithreading helps overlap communication with computation.

---

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

2 Computer Architecture Section, Electrotechnical Laboratory, Tsukuba-shi, Ibaraki 305, Japan; kodama@etl.go.jp

3 Real World Computing Tsukuba Research Center, Tsukuba, Ibaraki 305, Japan; msato@trc.rwcp.or.jp

In Proceedings of the Ninth ACM Symposium on Parallel Algorithms and Architectures, Newport, Rhode Island, June 22-25, 1997.

2 Multithreading Principles and Its Realization

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 [9,11]. Threads are usually delimited by remote reads 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 illustrates the basic principle.

Processor 0, P_0, has three threads, T0, T1,and T2, ready to execute in the 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 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.

The parameters which characterize the performance of multithreading include: (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 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 a few to tens of instructions while coarse-grain threading refers to thousands of 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-X 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 some machines such as SP-2 and AP1000+ service remote read requests concurrently with program execution, the EM-4 which is the predecessor of EM-X, treats a remote read as another 1-instruction thread which consumes processor cycles. This consumption adversely affects the performance. Details are presented in [18].

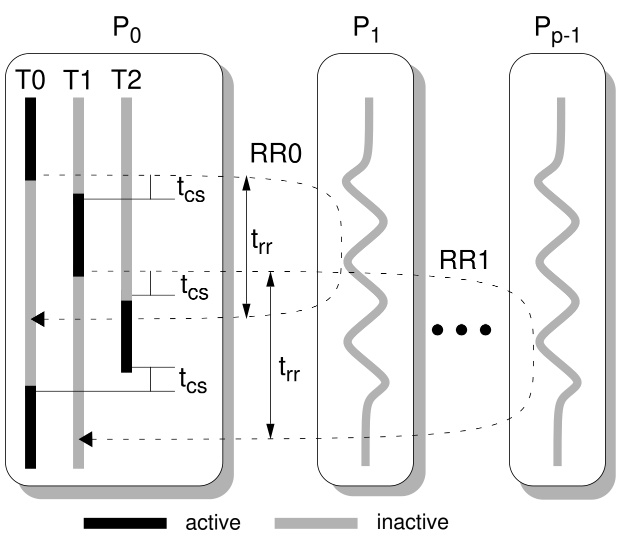

Figure 1: Multithreading on P processors. t_{cs} = context switch time, t_{rr} = remote read time, RR = remote read.

2.2 The EM-X multithreaded multiprocessor

The EM-X is a multithreaded distributed memory multiprocessor, built and operational at the Electrotechnical Laboratory since December 1995 [12,13]. Two types of computational principles are employed in designing the multiprocessor. The upper level uses the data-flow principles of execution for executing multiple threads simultaneously. The low level employs the conventional RISC-style execution to exploit program locality. The current prototype of EM-X has 80 EMC-Y processors, connected through a circular Omega network. Figure 2 shows an overview of the prototype EM-X multiprocessor. The network is the same as Omega network, except that each processor is attached to a switch box. Communication in the EM-X is done with 2-word fixed-sized packets.

A processing element is a single chip pipelined RISC-style processor, called EMC-Y, designed for fine-grain parallel computing. Each processor runs at 20 MHz with 4 MB of one-level static memory. The EMC-Y pipeline is designed to combine register-based RISC execution with packet-based dataflow execution for synchronization and message handling support. The processor consists of Switching Unit (SU), Input Buffer Unit (IBU), Matching Unit (MU), Execution Unit (EXU), Output Buffer Unit (OBU) and Memory Control Unit (MCU).

The Switch Unit sends/receives packets to/from the network. It consists of three types of components: two input ports, two output ports and a three-by-three cross-bar switch. Each port can transfer a packet, which consists of a word of address part and a word of data part, at every second cycle. A packet can be transferred in k+1 cycles to the processor k hops beyond by a virtual-cut-through routing. The message non-overtaking rule is enforced by this unit.

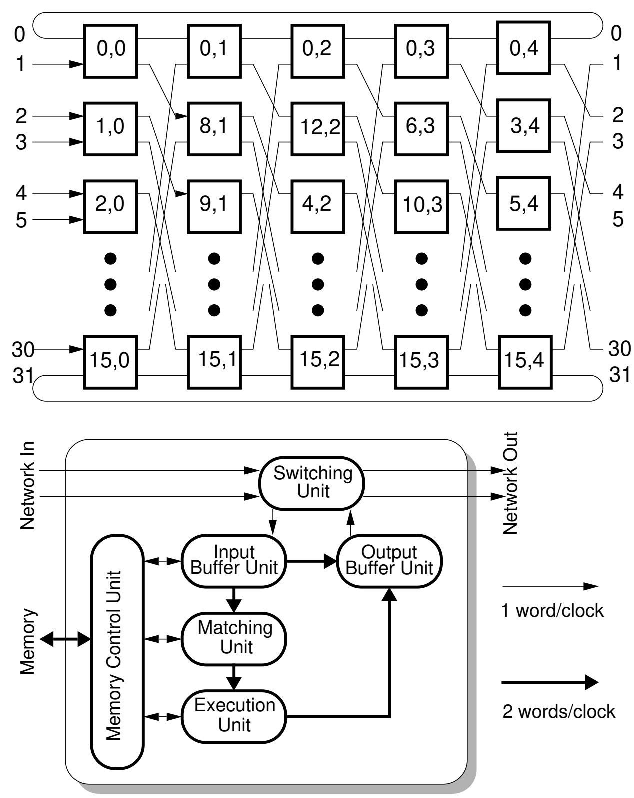

Figure 2: The 80-processor EM-X multiprocessor

The Input Buffer Unit receives packets from the switch unit. It has two levels of priority packet buffers for flexible thread scheduling. Each buffer is an on-chip FIFO, which can hold up to 8 packets. If the buffer becomes full, the packets are stored to on-memory buffer, and if not, they are automatically restored back to on-chip FIFO. The IBU operates independent of the EXU and the memory unit. Packets coming in from the network are immediately processed without interrupting the main processor. The path between IBU and MCU, called *by-passing direct memory access*, is one of the key features of EM-X. This by-passing DMA together with the path which connects IBU to OBU is the key to servicing remote read/write requests without consuming the cycles of Execution Unit.

The Matching Unit (MU) fetches the first packet in the IBU FIFO. If the packet requires matching, a sequence of actions will take place to prepare for thread invocation by direct matching. Actions include (1) obtaining the base address of the activation frame for the thread to be invoked, (2) loading mate data from matching memory, (3) fetching the top address of the template segment, (4) fetching the first instruction of the enabled thread, and (5) signaling the execution unit to execute the first instruction. Packets are sent out through the OBU which separates the EXU from the network. The MCU controls the access to the local memory off the EMC-Y chip.

The Execution Unit is a register-based RISC pipeline which executes a thread of sequential instructions. It has 32 registers, including five special purpose registers. All integer instructions take one clock cycle, with the exception of an instruction which exchanges the content of a register with the content of memory. Single precision floating point instructions are also executed in one clock, except floating point division. Packet generation is also performed by this unit, which takes one clock. Four types of send instructions are implemented, including remote read request for one data and for a block of data.

The Output Buffer Unit receives packets generated by the EXU or IBU. Again, the buffer can hold up to 8 packets. As we have briefly described above, the key feature of the OBU is to process packets generated by IBU. Remote read requests received by other processors are processed by the IBU which uses the by-pass DMA to read data from the memory. When the data fetched by the IBU is given to OBU, it will be immediately sent out to the destination address specified in the read request packet. This internal working of IBU and OBU is the key feature of EM-X for fast remote read/writes without consuming the main processor cycles.

2.3 Architectural support for fine-grain multithreading

The EM-X distributed-memory multiprocessor supports multithreading both in hardware and software. Hardware supports include thread invocation through packets, FIFO hardware scheduling of threads, and by-passing one-sided remote read/write. Software supports for multithreading include explicit context switch, global-address space, and register saving. Thread invocation or function spawning is done through 2-word-sized packets. When a thread needs to invoke a function (thread), a packet containing the starting address of the thread is generated and sent to the destination processor. The thread which just issued the packet continues the computation without any interruption unless it encounters a remote read or explicit thread switching.

As the thread invocation packet arrives at the destination processor, it will be buffered in the packet queue along with other packets arrived. Packets stored in the packet queue are read in the order in which they were received, hence *First-In-First-Out* (FIFO) thread scheduling. A thread of instructions is in turn invoked by using the address portion of the packet just dequeued. The thread will run to completion unless it encounters any remote memory operations or explicit thread switching. If the thread encounters a remote memory operation, it will be 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 the 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 using FIFO scheduling.

Whenever a thread encounters a remote read, a packet consisting of two 32-bit words is generated. The first 32-bit word contains the destination address whereas the second 32-bit contains the return address which is often called *continuation*. The read packet will be appropriately routed to the destination processor, where it will be stored in the input buffer unit for processing. The remote processor does not intervene to process the packet. The remote read packet will be processed through the by-passing mechanism which was explained earlier. When the read packet returns to the originating processor, it will be inserted in the hardware FIFO queue for processing, i.e., *thread resumption*. Remote writes do not suspend the issuing threads. For each remote write, a packet is generated which consists of two 32-bit words. The first word is the destination memory address and the second the data to be written. The write instruction is treated the same as other normal instructions. After sending out the write packet, the thread continues.

Software supports for multithreading include explicit context switch, global address space, and register saving. The current compiler supports C with thread library. Programs written in C with the thread library are compiled into explicit-switch threads. Two storage resources are used in EM-X: template segments and operand segments. The compiled 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 version does not share registers across threads. 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 and 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. The EM-X compiler supports a global address space. Remote reads/writes are implemented through packets. A remote memory access packet uses a global address which consists of the processor number and the local memory address of the selected processor. A typical remote read takes approximately 1 µs.

3 Designing Multithreaded Algorithms

3.1 Multithreaded bitonic sorting

Bitonic sorting, introduced by Batcher [4] consists of two steps: *local sort* and *merge*. Given P processors and n elements, each processor holds n/P elements. In the local sort step, each processor takes in n/P elements and sorts them in an ascending or descending order depending on the second bit of the processor number. The merge step consists of $O(\log^2 P)$ steps. In each merge step, elements are sorted across processors in a pair. As iterations progress, the distance between the pair of processors widens. The last iteration will sort elements on two processors with the distance of $P/2$.

Figure 3 illustrates bitonic sorting of n=32 elements on P=8 processors. Consider processors 0 and 1 at i=0, j=0. P0 has **L**=(5,13,24,32) and P1 has **L**=(6,14,23,31), resulted from the local sorting step. P0 and P1 will sort 8 elements in an ascending order as indicated by shaded circles. Hollow circles indicate that processors sort elements in a descending order. The line between P0 and P1 indicates that the processors communicate. P0 sends **L** to its mate processor P1 while P1 sends **L** to its mate P0. When P0 receives four elements from P1, it merges them with **L**, so does P1. Since P0 takes a lower position than P1, it takes the low half (5,6,13,14) while P1 takes the high half (23,24,31,32). This type of sending, receiving, and merging operations continues until the 32 elements are sorted across the eight processors.

A multithreaded version of bitonic sorting divides the inner *j* loop into *h* threads [22]. Each thread is responsible for merging *n*/*hP* elements. The main idea of the multithreaded algorithm is to first issue remote reads by *h* threads, called *thread communication parallelism*, followed by the computation whenever any *n*/*hP* elements are read, called *thread computation parallelism*. Read requests for *n*/*P* elements are issued before any merge. Whenever *n*/*hP* elements are read, i.e., whenever each thread finishes reading *n*/*hP* elements from the mate processor, it will merge them with its own list **L**. This reading (communication) and merging (computation) will take place simultaneously, to overlap computation with communication.

Figure 4 illustrates how two processors Px and Py sort 8 elements in an ascending order. For the illustration purpose we use *two* threads in each processor. Four elements are divided into two parts, each of which is assigned to a thread. Processors X and Y initially hold (2,5,6,7) and (1,3,4,8), respectively. Thread 0 of Px is responsible for reading and merging the first half (1,3) of Py while thread 1 does for the second half (4,8). Sorting of the eight elements on the two processors proceeds as follows:

1. At t_a, Thd0 sends out the read request RR0 to Py, and suspends itself.
2. Between t_a and t_b, the switch to Thd1 takes place, spending several clocks.
3. At t_b, Thd1 sends out the read request RR2 to Py, and in turn is suspended.
4. Between t_b and t_c, there are no threads running. Both threads are dormant.
5. At t_c, RR0 returns with value 1 which will be saved in a buffer for merge. The value *resumes* Thd0.
6. At t_d, RR2 returns with the value 4 but no further activities will take place since Thd0 is currently running.
7. At t_e, Thd0 sends out the read request, RR1, to Py, and then suspends itself. Switching to Thd1 takes place.
8. At t_f, Thd1 sends out the read request RR3 to Py, and in turn suspends itself.
9. Between t_f and t_g, there are no running threads. Both threads are in a suspended status, and therefore no computation takes place. Even though Thd1 has received the value 4, it cannot perform the merge operation since Thd0 is not complete. Merging 4 with the list will result in a wrong order. Thd1 can proceed only after Thd0 completes. This is exactly where sorting lacks computation parallelism across threads. As we shall see shortly, FFT has large computation parallelism across all threads.
10. At t_g, RR1 returns with value 3. Thd1 is still in the *suspended* status. Thd0 has now read all the necessary elements, and im-

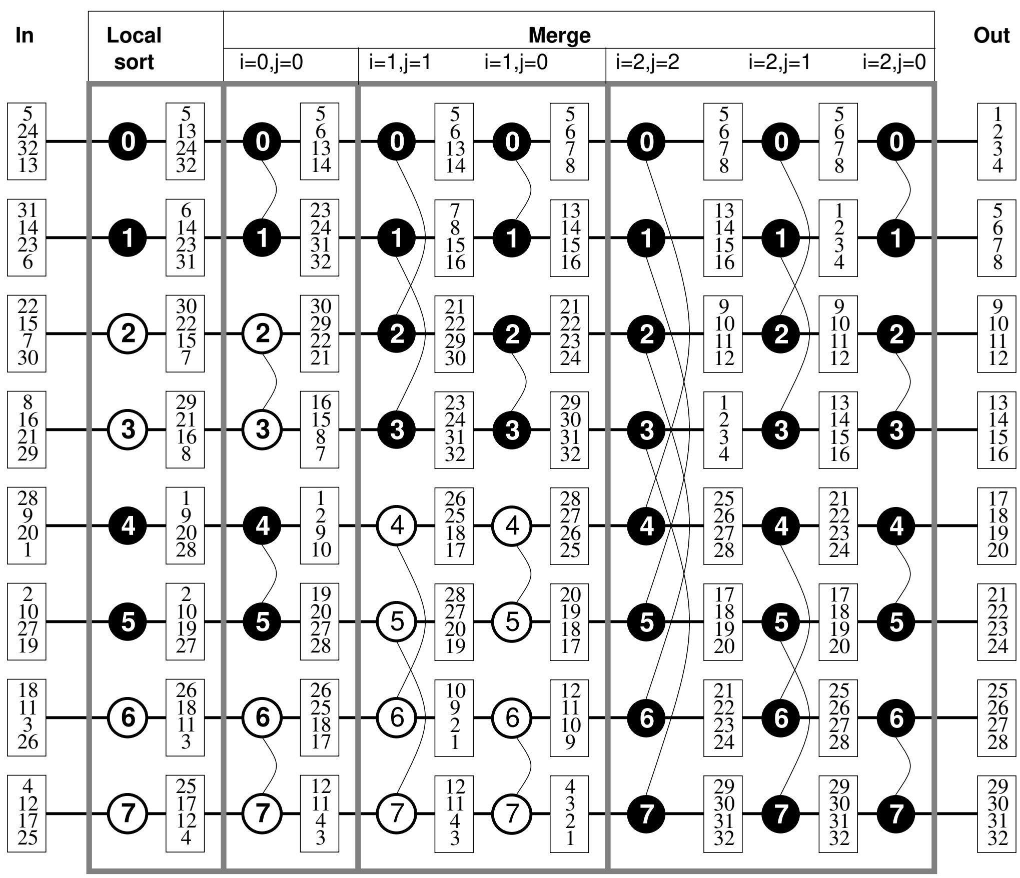

Figure 3: Bitonic sorting of 32 elements on 8 processors. Shaded circles indicate those processors performing ascending order merge while the hollow circles indicate processors performing descending order merge. Curves connecting two processors indicate that each processor reads four elements from the mate processor.

mediately merges the two elements with its own list.

11. At t_h, RR3 returns with value 8 but no actions will take place since Thd0 is currently running.
12. At t_i, Thd0 completes the merge, resulting in the output of (1,2,3). Switching to Thd1 now takes place. Since Thd1 also has two elements read from Px, it will immediately proceed to merging, which will give (1,2,3,4).

The above example assumes that each thread merges only after it reads *n/hP consecutive* elements from the mate processor. Bitonic sorting presents little computation parallelism across threads. Although communication can be done in parallel, computation must proceed in an orderly fashion so that the output buffer will contain elements sorted in a proper order. It should also be noted that the amount of computation for each processor is not the same. Thread 0 performed merge operations with 1 and 3. However, Thread1 performed merge operations with only one value, 4. When Thread1 reached t_i, the processor has four elements properly sorted. Thread 1 is therefore not required to read the fourth element 8 from the mate processor. This irregularity in computation occurs because not all the elements residing in the mate processor need to be read.

3.2 Multithreaded FFT

The second problem used in this study is Fast Fourier Transform (FFT) [6]. Consider a 16-point FFT on four processors. Using blocked data and workload distribution methods, the 16 elements are divided into four groups, each of which is assigned to a processor. P0 has elements 0 to 3, P1 4 to 7, etc. An FFT with n elements requires log n iterations. The 16-point FFT requires 4 iterations. In iteration 0, each processor obtains a copy of four elements by finding its mate processor. P0 remote reads four elements 8...11 while P1 does 12...15 from P3. P2 and P3 also obtain necessary data allocated to P0 and P1, respectively. Iteration 1 is essentially the same as iteration 0, except the logical communication distance reduces to half the first iteration. Iterations 2 and 3 do not requite communication since the required data are locally stored. In general, an FFT with blocked data distribution of n elements on P processors requires communication for the first log P iterations. The (log n) – (log P) iterations are local computations. In this report, only the first log P iterations are used.

Converting a single-threaded FFT to a multithreaded version is straightforward. Like bitonic sorting, the data assigned to each processor is grouped into h threads to control the thread granularity. Figure 5 explains the internal working of the multithreaded FFT with P=4, n=16, and h=2. Those four elements assigned to a processor are split into two groups. Each processor has two threads, each of which handles two elements.

Unlike Bitonic sorting, however, FFT possesses no data dependence between elements within an iteration. This observation leads to computation whenever *any* data is remote-read from the mate processor. In the above example, the threads compute and communicate independent of other threads. When Thd0 issues the remote read RR0, it is suspended. Processor 0 now switches to Thd1, which subsequently issues the remote read RR2. As RR0 returns value 8, Thd0 now proceeds to computation while RR2 is outstanding. As Thd0 completes the computation with the value 8, it sends out RR1, followed by its suspension. Thd1 *immediately* proceeds to computation with the value 10, which has returned sometime ago. Since the value 10 is the only one returned, the FIFO thread scheduling allows Thd1 to immediately proceed to computation with the value 10. Unlike bi-

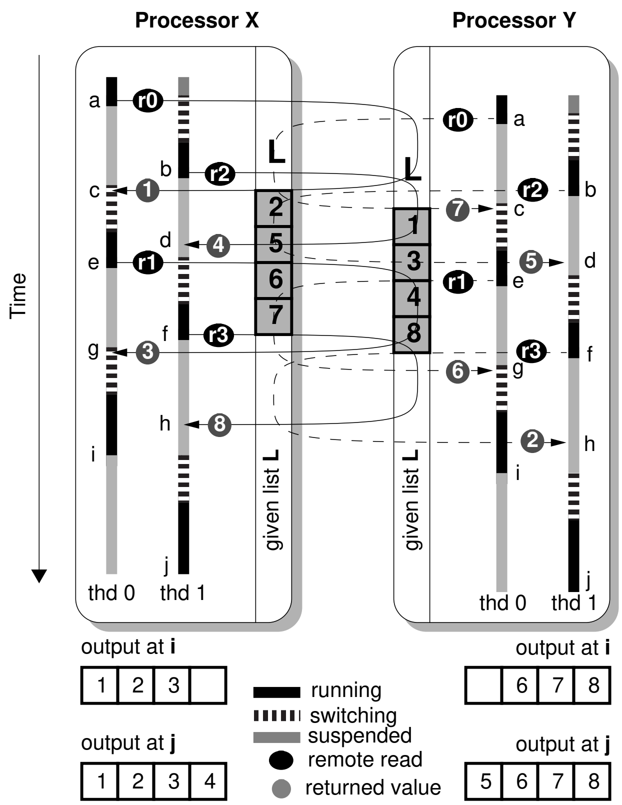

Figure 4: A multithreaded version of bitonic sorting. Processors x and y sort 8 elements in an ascending order. Characters *a..j* indicate the time sequence. Each processor has two threads, each of which handles two elements. Communication for Px is in solid lines and for Py in dotted lines.

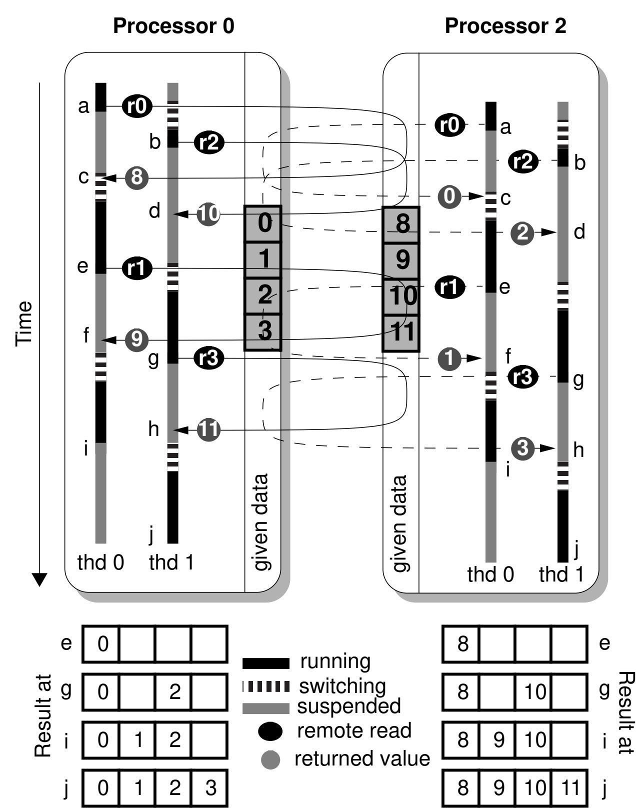

Figure 5: A multithreaded FFT, showing iteration 0. The figure is not drawn to scale. Each processor has two threads, each of which computes two points. No thread synchronization is required for FFT.

tonic sorting, no threads are synchronized for an orderly computation in FFT. No time is, therefore, lost for thread scheduling. This computation parallelism across threads will be evidenced by experimental results which will be shown later.

4 Overlapping Analysis

The multithreaded version of fine-grain bitonic sorting and FFT has been implemented on the EM-X. They are written in C with a thread library. To measure the effectiveness of overlapping capability we forced loops to execute synchronously by inserting a barrier at the end of each iteration. The terms elements and integers are used interchangeably in this paper. The unit for sorting is integers while that for FFT is points. An integer is 32 bits. A point consists of real and imaginary parts, each of which is 32 bits.

Communication times are plotted in Figure 6. The x-axis shows the number of threads while the y-axis shows the absolute communication time. The most important observation of the figure is that the communication time becomes minimal when the number of threads is *three* to *four*. The reason is clear. In bitonic sorting, each thread reads m elements from the mate processor before merging. The loop below shows an actual code taken from the program.

```
for (k=0;k<m;k++)     /* m = n/hP = # of elements/thread */
      buffer[k] = mem_read(mem_address++);
```

In each iteration, an element is read from the mate processor, assuming mem_address is properly initialized. After each read request, the thread is suspended and another thread is reactivated until each thread reads m elements. The loop body has 12 instructions, i.e., an iteration takes 12 clocks to execute, resulting in the run length of 12. The average remote memory latency, when the network is normally loaded, is approximately 1 to 2 μsec, or 20-40 clocks. Thus, each remote read needs two to four threads to mask off the 20-40 clock latency. This is precisely why the communication times become minimum when the number of threads is two to four. The number of threads higher than four does not give a notable advantage in masking off the latency.

The effect of multithreading is higher for FFT, as evidenced by the deep valleys. The run length for FFT is much higher than sorting. As we have explained earlier, bitonic sorting requires thread synchronization to ensure proper merge. However, FFT is free from thread synchronization. Therefore, the run length for FFT is very large. The following code shows how multithreading is actually implemented.

```
for(i=0;i<m;i++) {      /* m = n/hP = the # of points/thread */
   compute real_address and img_address;
   mate_real = remote_read(real_address++);
   mate_img = remote_read(img_address++);
   a lot of instructions with two reals and two imaginaries;
}
```

Unlike sorting, FFT proceeds to computation for the elements read from the mate processor. There is a very large number of instructions immediately after the second remote read. This large amount of computations can effectively mask off the latency. This is precisely why two or three threads simply outperform all other threads in FFT.

When the two problems are cross-compared, we note that sorting has much higher communication time than FFT. There are several reasons for the high communication. Among the reasons is the number of switches. Sorting requires thread synchronization whereas FFT does not. This thread synchronization presents a severe bottleneck as it limits the amount of computation parallelism across

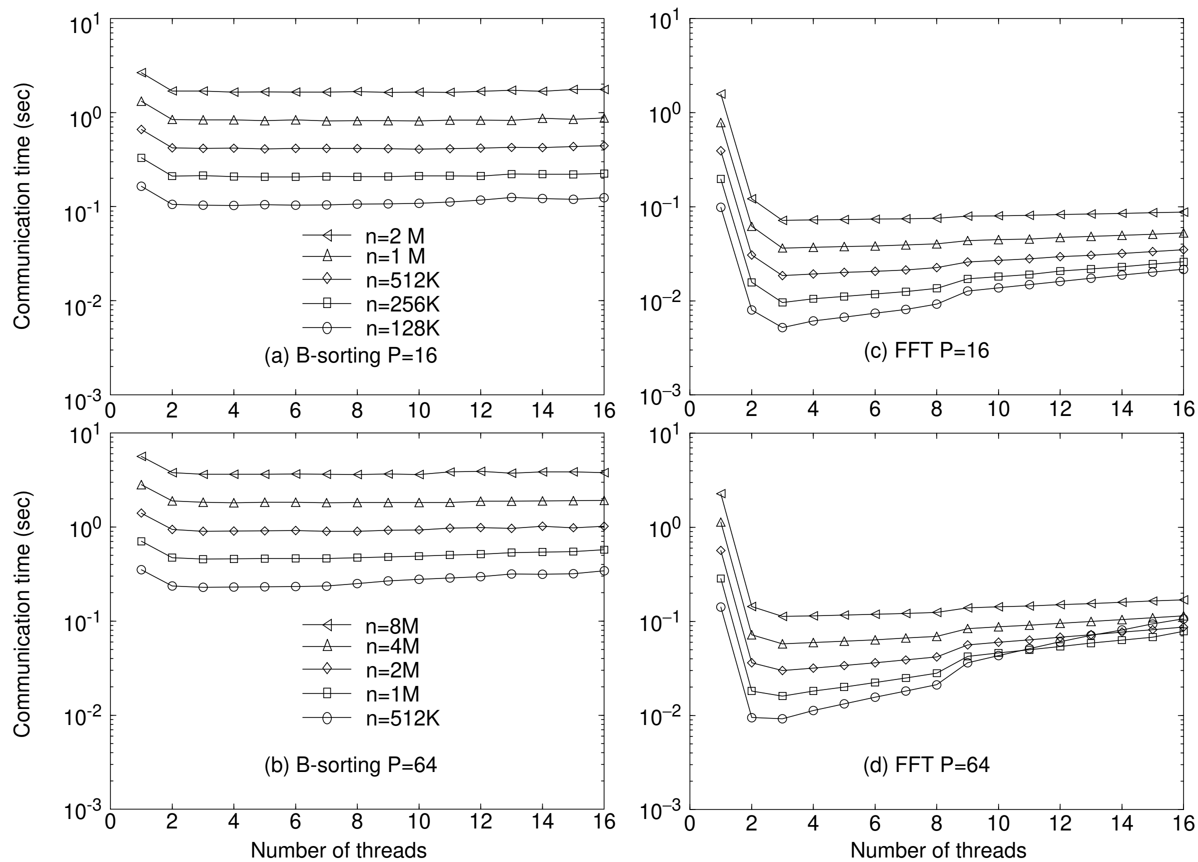

Figure 6: Communication time in seconds.

threads. Second, the sorting presents irregular computation and communication behavior due to the fact that not all the elements of the mate processor are needed to complete the merge operation. FFT, on the other hand, requires all the elements to be read for computation.

When the two problems are compared across different numbers of processors, the communication pattern is relatively consistent for both sorting and FFT. For bitonic sorting, increasing the number of processors to 64 rarely changes the communication pattern. As we can see, there is little difference in Figure 6(a) and (b) for sorting, or (c) and (d) for FFT. This consistency in communication pattern indicates that varying the number of processors is not the main factor for contributing to communication patterns. It should be noted that the data size for each processor is the same regardless of the total number of processors.

The effects of data size on communication pattern are inconsistent for both problems. For bitonic sorting with P=64, we find that varying data size rarely affects the communication performance, except for one thread. However, it becomes apparent for FFT. Note from Figure 6(d) with P=64 that the small data size of 512K gives a steeper curve than that of 8M, except one thread. In other words, the curve for 512K has a valley deeper than the one for 8M. The reason is that the data size of 8K for each processor is just too small compared to 128K. This relatively small data size is not significant enough to provide computations which can help mask off the communication latency.

To put the communication times into the multithreading perspective, we identify the efficiency of overlapping. Let $T_{\mathrm{comm,h}}$ be the communication time for h threads. We define the efficiency of overlapping as $E = (T_{\mathrm{comm,1}} - T_{\mathrm{comm,h}}) / T_{\mathrm{comm,1}}$. The communication time with one thread is used as the basis for overlapping analysis. When only one thread is used, there is no possibility that computation will overlap with communication since there is no other thread to switch to. Figure 7 shows the EM-X overlapping capabilities for the two problems.

Bitonic sorting has given roughly 35% overlapping of communication with computation. However, FFT has given over 95% of overlapping for two to four threads. This rather significant difference is attributed by two factors: First, bitonic sorting is sequential, presenting little parallelism among threads within an iteration while FFT is highly parallel. As we have explained in Section 3.1, communication for sorting can take place in any order but computation must be done in an ascending order of threads to ensure proper merge. Thread j cannot proceed to computation before Thread i, where $j > i$. Synchronization between threads is required to properly sort numbers. Therefore, bitonic sorting provides parallelism in remote reading only, not in computation. Threads in FFT, on the other hand, can proceed in any order, i.e., computation and communication can proceed in any order. Since there is no dependence between elements within an iteration, thread synchronization is not necessary, resulting in high parallelism among threads. This parallelism is clearly revealed in Figure 7(c)-(d).

The second reason FFT shows high overlapping efficiency is due to the fact that the amount of computation is much higher than that of communication. The total amount of computations for sorting is very small, consisting of several comparison and merging instructions. The computations for each element are not more than 10 instructions excluding loop control instructions. On the other hand, the computations involved in each element of FFT are large, which include some trigonometric function computations and a loop to find complex roots. There is a rather large difference between the two programs in terms of the computation associated with each element.

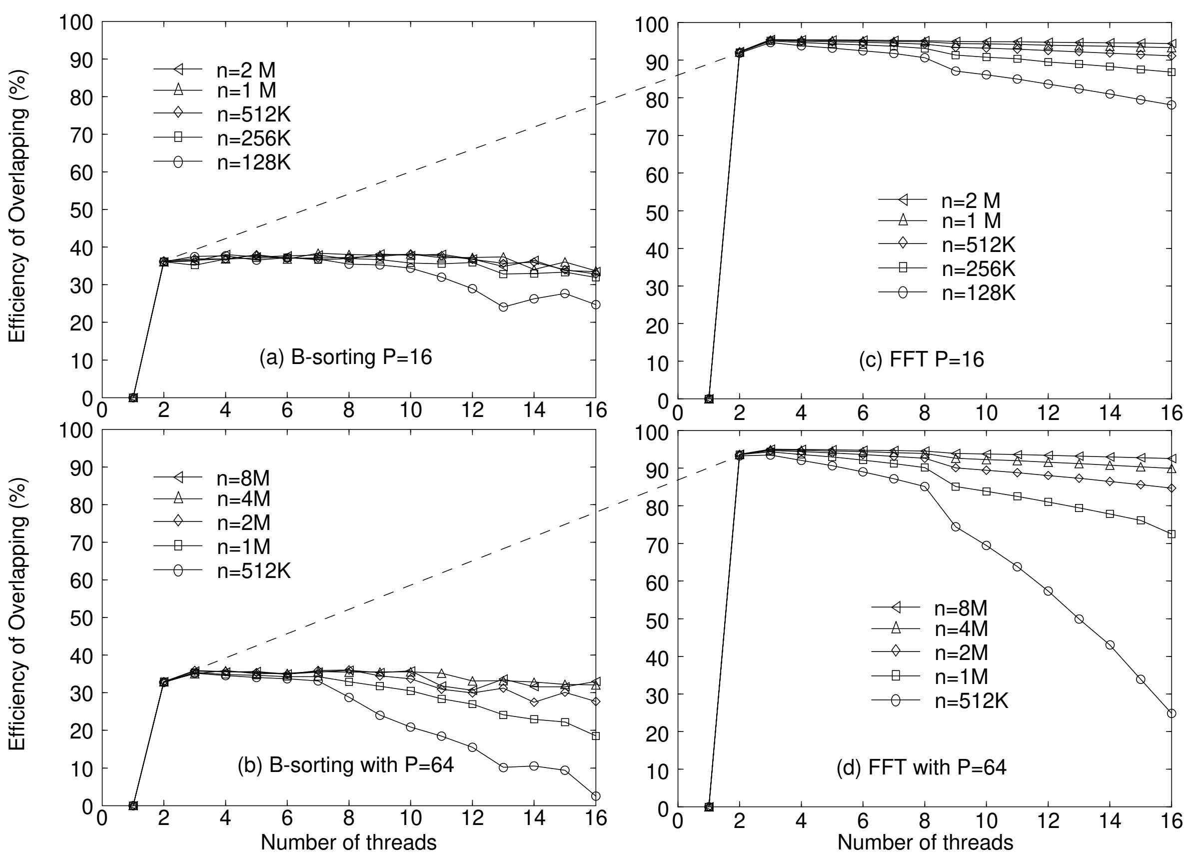

Figure 7: Efficiency of overlapping.

5 Analysis of Switches

Context switches are one of the key parameters which determine the performance of multithreading. In this section, we shall look further in to the behavior of multithreading in terms of switches. Figure 8 shows the individual execution time of the two problems. The plots have four timing components: computation, overhead, communication, and switching, listed from the bottom. There is no apparent anomaly for the distribution of times, except for one thread. The reason that the relative execution time for one thread is different from others is because one thread involves no overlapping, which makes the relative communication time 'look' larger. This relatively large communication time in turn makes the computation time look smaller.

Computation times for bitonic sorting are less than communication times. Figure 8(a)-(c) shows that computation times change as the number of threads changes. In fact, the total amount of computation must not change. The little change is attributed by the fact that the timing measurement is done through a global clock. When the problem size is large, no fluctuation occurs since the time to measure the global clock is negligible compared to the overall computation time. The reason bitonic sorting gives a little higher change in computation than FFT is attributed by another factor. Sorting is implemented in such a way that a processor may or may not have to read all the elements from the mate processor. As long as each processor produces n/P elements, it is done with the computation and will go into synchronization.

Overhead refers to the time taken to generate packets. It is essentially fixed not only for different numbers of processors but also for different problems since the total number of elements allocated to each processor is the same. We measured the overhead by using a null loop body, i.e., the loop body has no computation but instructions to generate packets. We find this was effective to measure the overhead cost for generating packets.

Switches are classified into three types: remote read switch, iteration synchronization switch, and thread synchronization switch. Figure 9 shows the three types of switches. The x-axis indicates the number of threads and the y-axis shows the absolute number of switches. The plots are drawn to the *same* scale. The figure reveals the internal working of multithreading. The remote read switching cost is in general the dominant factor contributing to the main switching cost. This is obvious because every remote read causes a thread switch. The remote read switching cost is fixed regardless of the number of threads because the number of elements to be read is indeed fixed. In fact, this switching can be readily derived from the given n, h, and P.

It is clear that thread synchronization switching cost is not the main factor for the two problems regardless of the numbers of processors. The behavior of thread synchronization switching is different for the two problems. The thread switching cost for bitonic sorting is rather high and is close to the iteration synchronization switching cost. On the other hand, FFT shows that there is a wide gap between thread and iteration synchronization switching costs. This gap shows that sorting spends a lot more time synchronizing threads within a processor. This was expected because threads in sorting are executed in sequence while FFT threads can execute in any order. The effects of the presence and absence of computation parallelism across threads are clearly manifested in the plots.

Iteration synchronization switching can be as high as remote read switching for the small problem size of 512K, as shown in Figure 9(a) and (c). As the number of threads reaches 16, the synchroniza-

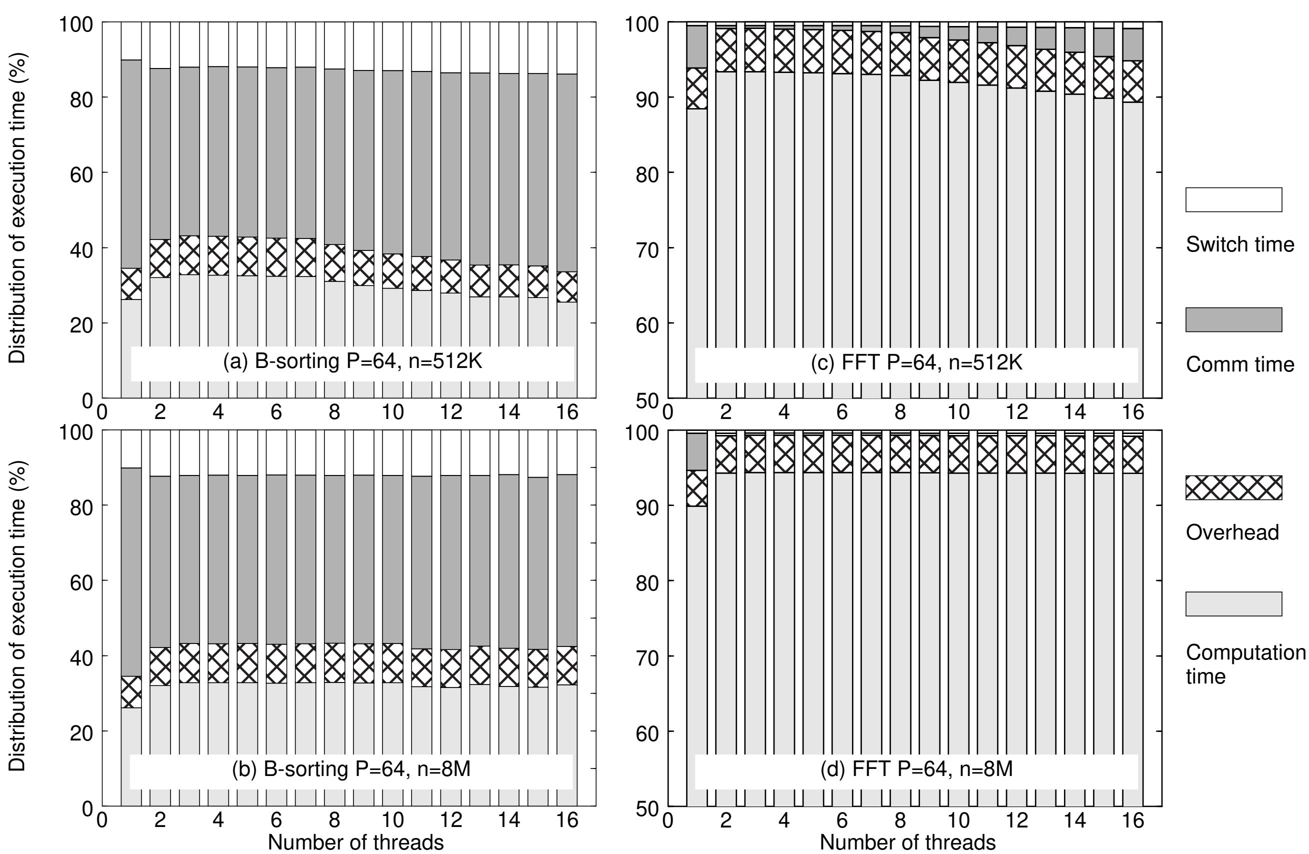

Figure 8: Distribution of execution time on 64 processors: Listed from the bottom are computation time, overhead, communication time, and switching time.

tion switch cost is in fact higher than the remote read switching cost. The reason is because the amount of computation is relatively small. After such small computations, 16 threads check if other threads are done for the *current* iteration. In fact, the iteration switching cost increases logarithmically as the number of threads increases linearly. There is approximately an order of magnitude difference in the number of iteration synchronization switches. For large problems shown in Figure 9(b) and (d), the amount of computation is now 16 times higher, which effectively eliminates the impact of iteration synchronization switching cost.

When the two problems are compared across different numbers of processors, switching pattern changes. Remote read switch and iteration synchronization switch do not meet. Each processor now finds more computations which separate the two curves. In fact, the switching cost no longer increases rapidly for P=64. The fluctuation for sorting with P=64 again shows that sorting possesses an irregular computation and communication pattern compared to FFT.

6 Conclusions

Reducing communication time is key to obtaining high performance on distributed-memory multiprocessors. Multithreading aims to reduce communication time by overlapping communication with computation. This paper has presented the internal working of multithreading through empirical studies. Specifically, we have used the 80-processor EM-X multithreaded distributed-memory machine to demonstrate how multithreading can help overlap communication with computation.

Bitonic sorting and Fast Fourier Transform have been selected to test the multithreading capabilities of EM-X. The criteria for the problem selection have been the computation-to-communication ratio and the amount of thread parallelism. Bitonic sorting has been selected for its nearly 1-to-1 computation-to-communication ratio and the small amount of thread computation parallelism. FFT has been selected because of its high computation-to-communication ratio and the large amount of thread computation parallelism. Both problems have been implemented on EM-X with blocked data and workload distribution strategies. The data size of up to 8M integers for sorting and 8M points for FFT have been used.

Experimental results have presented two key observations. First, the maximum overlapping has occurred when the number of threads is *two* to *four* for both problems. Sorting has the run length of 12 clocks per thread, and therefore four threads have been found adequate to mask off the latency of 20 to 40 clocks, or 1 to 2 μsec. Larger numbers of threads have adversely affected the amount of overlapping due to an excessive number of switches. In particular, iteration synchronization switch has been found the main cause for excessive synchronization costs among switches and a loop. The run-length of FFT is very large with hundreds of clocks due to trigonometric function computations. This rather high run-length has been found sufficient to effectively tolerate the latency of 20 to 40 clocks.

Second, the ratio of computation to communication plays a critical role in tolerating latency. Bitonic sorting results have shown that the maximum overlap has reached approximately 35%. The reason for the low overlapping was because bitonic sorting has small absolute computation time and lacks thread computation parallelism, requiring thread synchronization. FFT, on the other hand, has shown over 95% of communication overlapping due to its high computation-to-communication ratio and the large amount of both thread computation and communication parallelism. FFT threads can compute and communicate in any order within an iteration, requiring no thread synchronization.

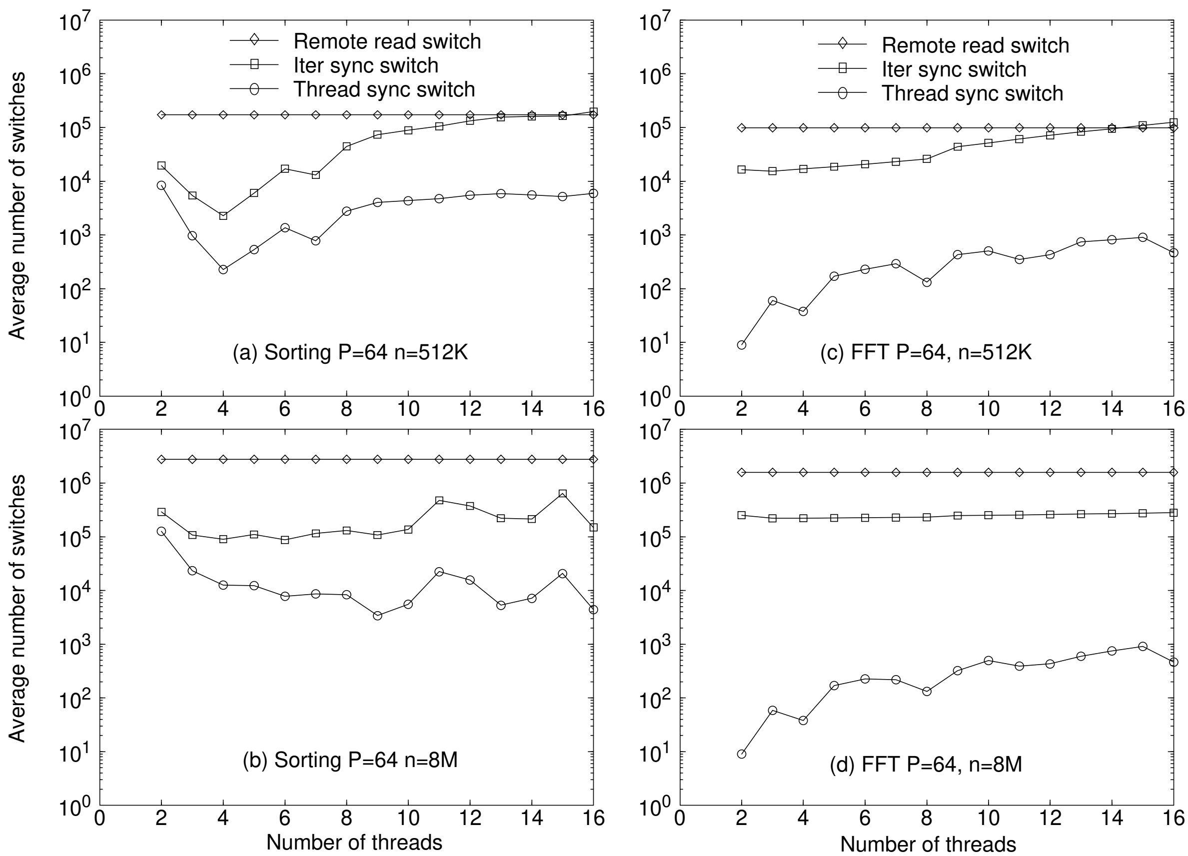

Figure 9: Average number of switches for each processor.

The study has indicated that fine-grain multithreading can hold a key to obtaining high performance on distributed-memory machines. The fact that multithreading can tolerate over 35% of the total communication time for sorting in the *absence* of computation parallelism clearly demonstrates such a premise. Problems which possess irregular computation behavior and moderate parallelism can be a logical target for obtaining high performance through multithreading. We believe it is a realistic goal to achieve high overlapping for such irregular problems if the thread scheduling and synchronization mechanisms are fine tuned to thread computation and communication parallelism. It is our next goal to fine-tune mechanisms for hardware thread scheduling and synchronization.

Acknowledgments

A. Sohn and J. Ku are supported in part by the NASA University Joint Venture in Research Program NAG8 1114-2. A. Sohn is supported in part by 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 *Proc. ACM Int’l 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, “SP-2 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. K. Batcher, “Sorting Networks and Their Applications,” in Proc. the AFIPS Spring Joint Computer Conference 32, Reston, VA, 1968, pp.307-314.
5. R. D. Blumofe, M. Frigo, C. F. Joerg, C. E. Leiserson, and K. H. Randall, “Dag-Consistent Distributed Shared Memory,” in *Proc. of IEEE Int’l Parallel Processing Symposium, Honolulu*, Hawaii, April 1996, pp.132-141.
6. J. M. Cooley and J. W. Tukey, “An algorithm for the machine calculation of complex Fourier series,” *Math. Comp.*, 19: 297-301, 1965.
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. D. Culler, R.M. Karp, D.A. Patterson, A. Sahay, E.E. Schauser, E. Santos, R. Subramonian, and T. von Eicken, “LogP: Towards a Realistic Model of Parallel Computation,” in *Proc. of the 4th ACM Symposium on Principles and Practice of Parallel Programming*, San Diego, CA, May 1993.
9. G. Gao, L. Bic and J-L. Gaudiot (Eds.) Advanced Topic in Dataflow Computing and Multithreading, IEEE Computer society press, 1995
10. High Performance Fortran Forum, High Performance Fortran Language Specification version 2.0, Center for Research on Parallel Computation, Rice University, Houston, TX, November 1996.
11. R. Iannucci, G. Gao, R. Halstead, and B. Smith (Eds.), Multithreaded Computer Architecture, Kluwer Publishers, Norwell, MA 1994
12. 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.
13. 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.
14. R. Nikhil, G. Papadopolous, and Arvind, “*T: A Multithreaded Massively Parallel Architecture,” in *Proceedings of ACM Int’l Symposium on Computer Architecture*, Gold Coast, Australia, May 1992, pp.156-167.
15. G. Papadopolous, An Implementation of General Purpose Dataflow Multiprocessor, MIT Press, Cambridge, MA, 1991.
16. R. Saavedra-Barrera, D. Culler, and T. von Eicken, “Analysis of Multithreaded Architectures for Parallel Computing,” in *Proc. of ACM Symposium on Parallel Algorithms and Architectures*, pages 169-178, July 1990.
17. S. Sakai, Y. Yamaguchi, K. Hiraki, and T. Yuba, “An Architecture of a Data-flow Single Chip Processor,” in *Proc. of ACM International Symposium on Computer Architecture*, Jerusalem, Israel, May 1989, pp.46-53.
18. M. Sato, Y. Kodama, S. Sakai, Y. Yamaguchi, and Y. Koumura, “Thread-based Programming for the EM-4 Hybrid Dataflow Machine,” in *Proceedings of ACM International Symposium on Computer Architecture*, Gold Coast, Australia, May 1992, pp.146-155.
19. M. Sato, Y. Kodama, S. Sakai, and Y. Yamaguchi, “Experience with Executing Shared-Memory Programs using Fine-grain Communication and Multithreading in EM-4,” in *Proceedings of the 8th IEEE International Parallel Processing Symposium*, Cancun, Mexico, April 1994, pp.630-636.
20. S. Scott, “Synchronization and Communication in the T3E Multiprocessor,” in *Proc. of ACM Conference on Architectural Support for Programming Languages and Operating Systems*, Boston, MA, October 1996.
21. B. J. Smith, “A Pipelined, Shared Resource MIMD Computer,” in *Proceedings of International Conference on Parallel Processing*, 1978, pp.6-8.
22. A. Sohn, “Communication Efficient Low Overhead Bitonic Sorting,” *Technical Report*, NJIT CIS Dept. 1996.
23. A. Sohn, M. Sato, N. Yoo, and J-L Gaudiot, Data and Workload Distribution in a Multithreaded Architecture, *Journal of Parallel and Distributed Computing 40*, February 1997, pp.256-264.
24. C. B. Stunkel, D. G. Shea, B. Abali, M. Atkins, C. A. Bender, D. G. Grice, P. H. Hochschild, D. J. Joseph, B. J. Nathanson, R. A. Swetz, R. F. Stucke, M. Tsao, and P. R. Varker, “The SP-2 Communication Subsystem,” Technical Report, IBM T. J. Watson Research Center, August 1994.
25. 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.
26. M.R. Thistle and B.J. Smith, “A Processor Architecture for Horizon,” in *Proceedings of Supercomputing 88*, Florida, October 1988, pp.35-40.