Identifying the Capability of Overlapping Computation with Communication

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

Y. Kodama, M. Sato[1], H. Sakane, H. Yamana, S. Sakai, Y. Yamaguchi
Computer Architecture Section
Electrotechnical Laboratory, 1-1-4 Umezono, Tsukuba-shi, Ibaraki 305, Japan, kodama@etl.go.jp
2 Real World Computing Tsukuba Research Center, Tsukuba, Ibaraki 305, Japan

Submitted to Parallel Architectures and Compilation Techniques, March 1996

Abstract

Overlapping computation with communication is central to obtaining high performance on distributed-memory multiprocessors. Identifying the capability of overlapping, machine architects and programmers will be able to provide tools which can effectively utilize the underlying architecture. This report explicates the overlapping capability of two distributed-memory multiprocessors: the laboratory prototype EM-X multithreaded multiprocessor and a commercially available IBM SP2 with wide nodes. The well-known bitonic sorting algorithm is selected for experiments. Various message sizes are used to determine when, where, how much and why overlapping takes place. Experimental results indicate that both multiprocessors would yield up to 30% to 40% overlap of communication time when the message size is approximately 1K integers. EM-X is found message-size insensitive yielding high overlap for various message sizes while SP2 was effective for a window of message size of 512 to 2K integers, depending on the amount of computation.

1 Introduction

Distributed-memory multiprocessors have been regarded as a viable architecture of scalable and economical design in building large parallel machines to meet the ever-increasing demand for high performance computing. In these paradigms of machine architecture, it is deemed relatively simple to increase a machine's capability simply by adding more processors to the system incrementally as required. Not only are there numerous research prototype machines having this distributed-memory/message-passing paradigm such as AP1000+, EM-4 and EM-X [16,19,20], FLASH, *T:NG [9], EARTH [15] and Alewife [1], but also there are some commercial machines built or being built around this scheme: T3D, SP-2 [2,23,24] and Tera [4].

The scalability of performance for general purpose computation, however, is still in question. The major problem which hinders the performance of distributed-memory machines is remote memory latency. Distributed-memory machines distribute data in a way that there is no overlapping or copying of *major* data. Typical distributed-memory machines incur much latency, ranging approximately from a few micro seconds 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 is advanced using pipelining or superscalar architecture and as the machine size becomes bigger.

Numerous approaches have been reported to study the behaviors of communication overhead. Data partitioning and alignment [13] is a typical method to reduce communication overhead. Analyzing the behaviors 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 [14] moves computation or data based on runtime-determined statistics. The heuristic used in the approach migrates 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. Multithreading aims at tolerating remote memory latency through split-phase read mechanism and context switch [10,12,16,20,22]. 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.

Models developed to capture the behaviors of communication include the BSP [25], the Postal model [6], LogP [11], LogGP [3], etc. The LogP model defines four parameters, L for latency, o for overhead, g for gap between messages, and P for processors. Each of the four parameters is designed to capture a certain aspect of communication behavior for short messages [11,3]. The LogGP model extends the LogP model by adding G to capture the communication behavior of long messages, where G captures the gap between bytes of the same message [3]. While it is very difficult to model the communication behavior due to unpredictable network contention, these models have captured communication behaviors for some subset of typical benchmark problems.

Distributed-memory machines such as T3D, AP1000+, and SP2 provide some hardware support to reduce communication overhead. Specifically, they provide a mechanism to overlap computation with communication in a way to take some of the burden from the main processor. It is precisely the purpose of this report to investigate the overlapping capability. Specifically, we identify when, where, how much and why overlapping takes place. Two distributed-memory multiprocessors are used in our experiments, including a commercially available IBM SP2 and the laboratory prototype EM-X multithreaded multiprocessors. We begin our discussion by giving a brief description on the two machines. Section 3 explains a bitonic sorting algorithm that is used for our experiments. Some of the experimental results are listed in section 4. We analyze the results in section 5 along with the LogP and LogGP models. The last section concludes our study to identify the capability of overlapping computation with communication.

2 Two Distributed-Memory Multiprocessors

2.1 The EM-X multithreaded distributed-memory multiprocessor

The EM-X is a multithreaded distributed memory multiprocessor, built and operational at the Electrotechnical Laboratory since December 1995 [16]. Two types of computational principles are employed in designing the multiprocessor. An upper level uses the data-flow principles of execution for executing multiple threads active simultaneously. The low level employs the conventional RISC style execution to exploit program locality. The current prototype of EM-X has 80 processors, called EMC-Y, connected through circular Omega network. Figure 1 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 and provides a maximum diameter of O(log p) for low latency and high throughput. Communication in the EM-X is done with 2-word fixed-sized packets.

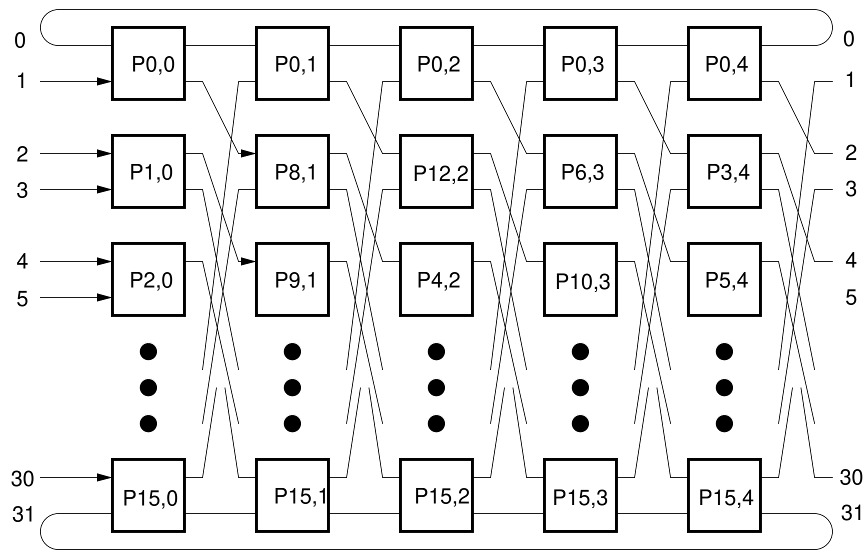

Figure 1: The 80-processor EM-X distributed-memory multiprocessor

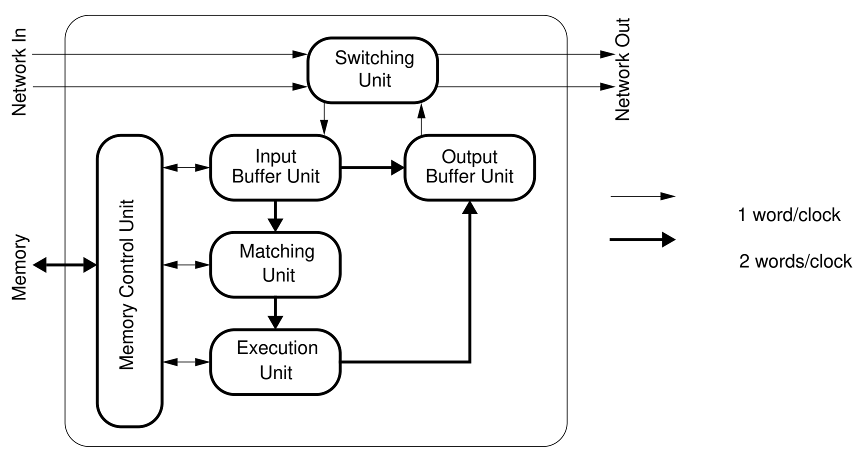

Figure 2: Architecture of the EMC-Y processor

A processing element shown in Figure 2 is a single chip pipelined RISC-style processor, called EMC-Y, designed for fine-grain parallel computing. Each processor has 4M bytes of one-level static memory running at 20 MHz. 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 h+1 cycles to the processor h hops beyond by a virtual-cut-through routing. The message non-overtaking rule is enforced by this switch unit.

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 the buffer is not full, 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 (DMA)*, 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 FIFO of IBU. 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 current 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 start execution of 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 a content of a register with a content of a memory. Single precision floating point instructions are also executed in one clock cycle, except floating point division. Packet generation is also performed by this unit, which takes one clock cycle. Four types of send instructions are implemented, including remote read request for one data and remote read request 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.

In summary, the EMC-Y generates and dispatches packets directly to and from the network and provides hardware support to handle queuing and scheduling of packets. Packets coming in from the network are buffered in the packet queue. As a packet is read from the packet queue, a thread of computation specified by the address portion of the packet is invoked with one-word data. The thread runs to completion unless it encounters remote memory reads or other thread invocation. In the event that the current thread is to be suspended, any live state in the thread will be saved in an activation frame associated with the thread, resulting in thread switch. The completion or suspension of a thread causes the next packet to be automatically dequeued from the packet queue according to the order in which they were received. Thread scheduling is, therefore, *explicit* First-In-First-Out (FIFO).

2.2 IBM SP2 Multiprocessor

IBM SP2 was designated as High Performance Computing and Communication Program Testbed 1 (HPCCPT-1) by NASA Ames Research Center. Since its first installation at the Center in the late summer of 1994, numerous studies and experiments have been performed for various large scale applications, including large scale computational aero-science applications such as computational fluid dynamics. There are numerous literatures and results on SP2 available and reiteration of the machine's features and characteristics will be redundant. We will describe the machine architectures and interconnection network which are related to our study on the communication aspect of SP2.

SP2 is developed at the IBM Scalable PowerParallel Systems [2,23,24]. Processors are POWER2 architecture which is a six instruction issue superscalar machine running at 66.7 MHz. The six instructions include two branch related instructions, two fixed point instructions, and two floating-point instructions. There are two types of nodes: thin nodes and wide nodes. The main difference between thin nodes and wide nodes are essentially memory and disk capacity. Thin nodes have POWER2 architecture with up to 64-512 MB of MM, 1-4 GB disk, and 4 micro channels. Wide nodes have the same processor with up to 64-2 GB of MM, 1-8 GB disk, and 8 micro channel slots. The processors used in this experiments are all based on wide nodes. The nodes are connected through High Performance Switch, called Vulcan chip [24]. Each chip connects up to 8 processors. Eight Vulcan switching chips comprise a switching board. Figure 3 shows one switch board where eight Vulcan chips connect 32 processors. Figure 4 shows an SP2 communication adaptor. The adaptor consisting of a 4K Byte FIFO and an Intel i860 processor is designed to take the burden of communication related activities off the main processor.

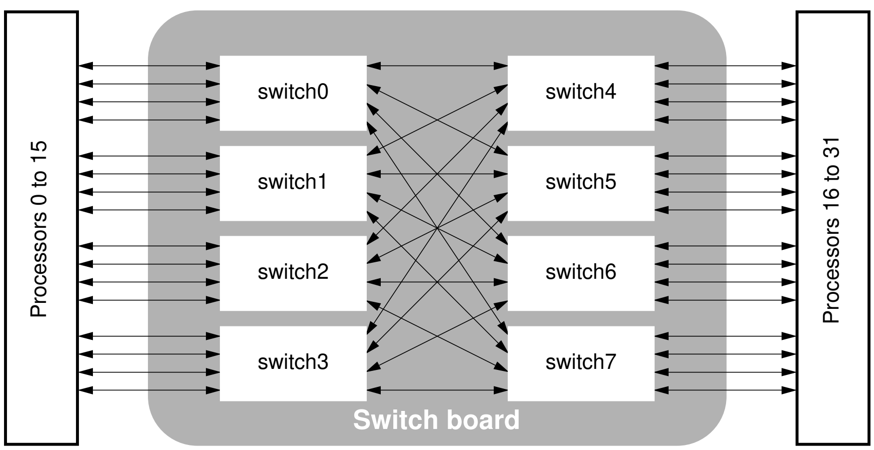

Figure 3: An SP2 with 32 processors connected through a switch board of 8 Vulcan switches.

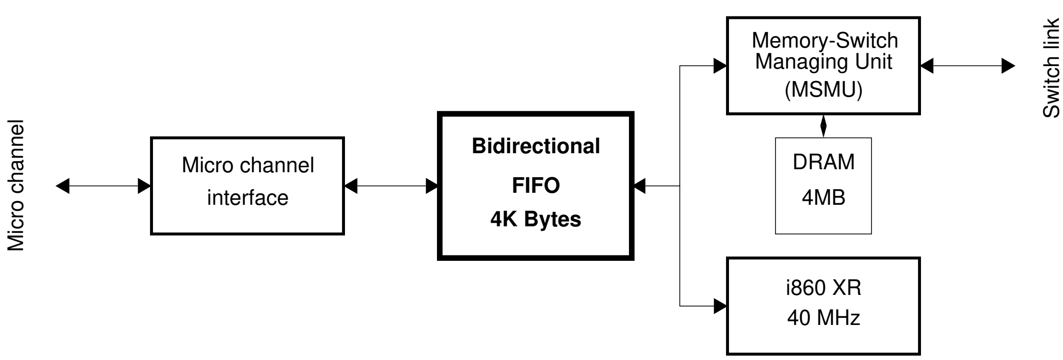

Figure 4: Communication adaptor for overlapping communication and computation.

By introducing a communication co-processor, the main processor can handle computation while the communication processor handles sending/receiving messages to/from other processors. The Bidirectional FIFO highlighted in the figure is central to overlapping communication and computation. The buffer can hold up to 4K bytes of data for two directions or 2K bytes for each direction. The i860 initiates sending/receiving messages. As we shall discuss later, this 4Kbytes of FIFO plays a central role to taking the burden off the processor, thereby providing the capability of overlapping of computation with communication. Other components in the adaptor help realize this overlapping.

3 Problem Description

Benchmark problem, bitonic sorting, is described in this section. We briefly describe the original landmark bitonic sorting by Batcher [7]. A problem in the original sorting is identified which cause a communication bottleneck in distributed-memory machines. We present a communication efficient bitonic sorting that we developed for our experiments.

3.1 Bitonic sorting

Bitonic sorting is a well-known parallel sorting algorithm introduced by Batcher [7]. It consists of three phases: *local sort*, *bitonic generation*, and *merge*. Figure 5 lists a pseudo code for bitonic sorting for p processors and n integers, where each processor holds n/p integers.

```
Local radix sort on L1 of n/p elements;                 /* L1 is my list, L2 is mate's list */
for (i=0; i<log p – 1; i++) {                           /* p = number of processors */
      for (j=i; j<=0; j++) {                            /* */
            Send L1 (n/p elements) to a mate processor  /* mate processors can be found with bit manipulations */
            Receive L2 (n/p elements) from the mate processor
            Merge_sort two lists L1 and L2
            Take a half depending on the position of my pid /* pid = processor identification number */
      }
}
for (i=0; i<log p; i++) {
      Send L1 (n/p elements) to a mate processor
      Receive L2 (n/p elements) from the mate processor
      Merge_sort two lists L1 and L2
      Take a half depending on the position of my pid
}
```

Figure 5: Pseudo code for bitonic sorting

In the local sorting step, each processor takes in a list of n/p elements, L1, and locally sorts it in an ascending/descending order depending on the processor identification number, or PID. No communication takes place in the local sorting. We use radix sort with 8 bits for the local sort. The second step, bitonic generation, generates the input in a bitonic sequence. This step consists of two nested loops, each of which has $\log p - 1$ iterations. Each processor first finds its mate processor to send L1, based on the iteration number and its PID. Upon receiving the list, now called L2, each processor merges L1 with L2. Depending on the iteration and position with respect to the mate processor, each processor takes half the merged list, storing back in L1. At the end of this generation step, processors 0 to $p/2 - 1$ will have $n/2$ elements sorted in an ascending order across $p/2$ processors while processors $p/2$ to $p - 1$ will have $n/2$ elements sorted in a descending order across $p/2$ processors. The last step, merging, combines the bitonic sequence into in an ascending/descending order across all processors. Since the example assumes an ascending order at the end, all processors merge two lists in an ascending order. Again, each processor receives n/p elements, merges, and takes first/second half, depending on its position.

Figure 6 graphically illustrates bitonic sorting of 32 elements on 8 processors, i.e., n=32 and p=8. For example, consider processors 0 and 1 of bitonic generation with i=0, j=0. P0 has L1=(5,13,24,32) and P1 has L1=(6,14,23,31), from the local sorting step. P0 and P1 will sort 8 elements in an ascending order as indicated by shaded circles. P0 sends L1 to its mate processor P1 while P1 also sends L1 to its mate P0. When P0 receives four elements from P1, it merges them with L1, 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 a bitonic sequence is generated across the eight processors.

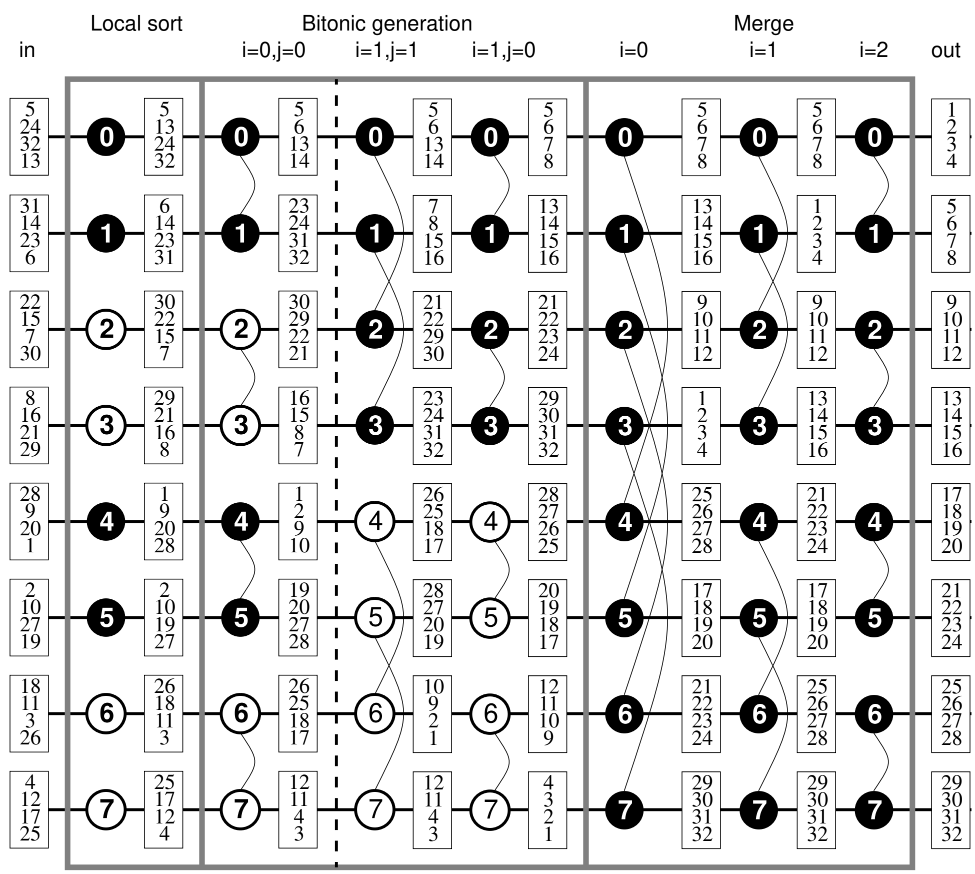

Figure 6: Bitonic sorting of 32 elements on 8 processors. Shaded circles indicate those processors performing ascending order merge while hollow circles indicate processors performing descending order merge.

3.2 Communication efficient bitonic sorting

Bitonic sorting described above presents a bottleneck in sending/receiving a list of n/p elements to/from its mate processor. After sending out its list, each processor waits for the n/p elements to arrive from the mate processor. No work can be done while processors are waiting. When $m=n/p$ is a small number of tens to hundreds of elements, the waiting time may be small, giving little performance difference. However, when *m* is large in the order of thousands or tens of thousands, the waiting time can be a severe bottleneck since no work can be performed unless the entire list is received.

We have developed a communication efficient bitonic sorting which eliminates this bottleneck by overlapping computation with communication [21]. The idea is straightforward. The list of n/p elements assigned to each processor is now divided into s smaller segments, each of which consists of m/s elements. Communication, therefore, is done in a segment of n/ps elements. Each processor sends a segment of n/ps elements at a time. Whenever a segment is received, the processor immediately proceeds to merge its own n/p elements with the segment just received, while the next segment is in transit. Computation and communication are therefore effectively overlapped to mask out part of the communication latency caused by sending the entire n/p elements. Figure 7 illustrates how this fine-grain bitonic sorting works.

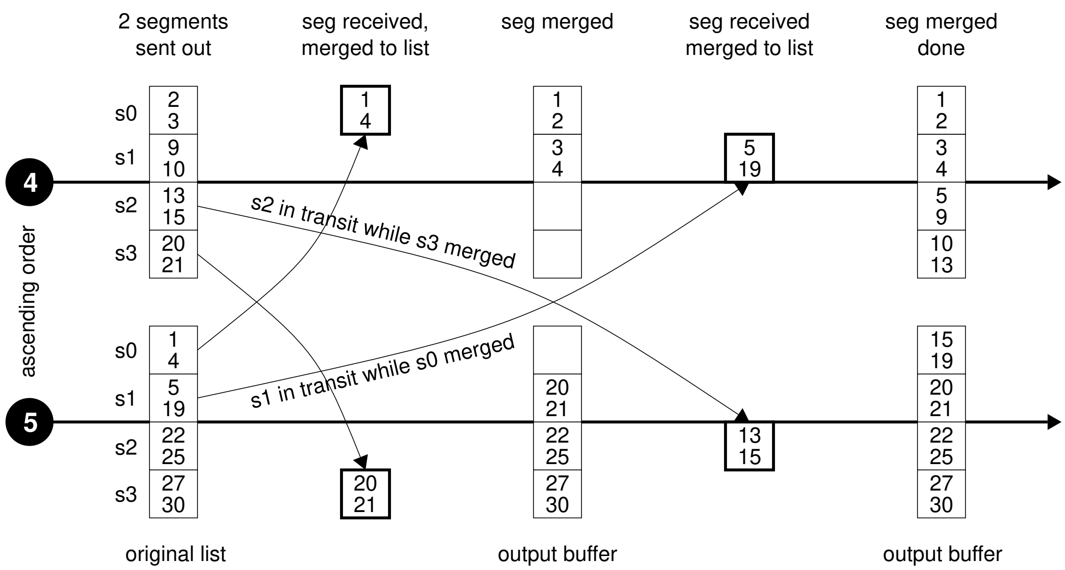

Figure 7: Overlapping of computation with communication in bitonic sorting. Processors 4 and 5 sort 16 elements in an ascending order. The example shows that only half of the list needs to be sent to its mate processor.

Figure 7 shows that processors 4 and 5 sort 16 elements in an ascending order. Each list is divided into four segments. Communication is done in the segment size of 2 elements. Initially, P4 sends out two segments, s3 and s2, while P5 does segments 0 and 1. For illustration purposes, the number of segments to send out is fixed to two but in actual experiments this number varies depending on n, p, and s. When P0 receives s0=(1,4) from P5, it immediately starts merging with its own list, resulting in (1,2,3,4). P5 also performs merging operations, resulting in (20,21,22,25,27,30). While this merging takes place, the second segment is in transit. In an ideal situation, the second segment has already been received when the merging on the previously received segment completes. Assuming that the segment has arrived when the merge completes, P4 and P5 immediately proceed to merge the second segment with the original list, resulting now in (1,2,3,4,5,9,10,13) and (15,19,20,21,22,25,27,30), respectively.

The above example shows that each processor needs to send half of the 16 elements, which reduces the communication to half. It is possible, however, that each processor may need to send only one segment or the entire list depending on the runtime list characteristics. Supposed that two processors, P_i and P_j, sort in an ascending order, and that P_i takes the low half while P_j takes the high half of the sorted list. If the largest integer of P_i is smaller than the smallest of P_j, it needs to send only one segment. However, if the largest of P_j is smaller than the smallest of P_i, it will need to send the entire list. This is a runtime characteristic which is difficult to capture. In our experiments, we varied the number of segments sent out at a time. The details on sorting itself is outside the scope of this report and we do not discuss any further. As we shall discuss in the following section, the above method is highly efficient on distributed-memory machines. We found that it is over 30% faster than the best known results using radix sort on SP2. The details on the sorting aspects are presented in [21].

4 Experimental Results

Fine-grain bitonic sorting has been implemented on the two distributed-memory machines. The SP2 version is written in C with Message-Passing Interface (MPI) [17]. Since its implementation does not require a special environment, it is portable and can be run on any machine which supports MPI. The EM-X version is written in C with thread library. It runs only on the EM4 and EM-X multithreaded machines. It should be noted, however, that the EM-X version does not utilize the multithreading capability of EM-X. Since it will be unfair to compare with non-multithreaded machine, we programmed such that there is only one thread active at a time. The capability of latency tolerating using multi-threading is reported in [19,20,22].

To measure the effectiveness of overlapping capability we forced loops to execute *synchronously*. For both version, we inserted a *barrier* at the end of *j*-loop of `generate` (after line 7) and *i*-loop of `merge` (after line 14). While the fine-grain bitonic sorting does not need to run synchronously, this is entirely for measuring the computation time and communication time of each iteration. Measuring communication time is straightforward. For the SP2 version, we timed MPI_Wait() as well as other message related functions such as non-blocking send/receive, MPI_Isend() and MPI_Irecv(). Even though MPI_Isend() and MPI_Irecv() are non-blocking functions and their contribution to the overall communication time is negligible, we still measured their function invocation time for the sake of completeness. Most of the communication time (over 95%) comes from MPI_Wait().

Several parameters are frequently used throughout this paper. Elements and integers are interchangeably used in this paper. An integer is 32 bit or 4 bytes. We list below several parameters used throughout this paper:

- p = number of processors
- n = the total number of integers (keys) to sort
- s = number of segments per processor
- $m = n/ps$ = number of integers per segment, or message size
- K = 1024, and M = 1048576

There are several time components we measured in the experiments. Execution times consist of five parts: local sort time, generate computation time, generate communication time, merge computation time, and merge communication time. Some of the experimental results are listed below in Tables 1 to 3 to show the scale of execution time. All execution times are in seconds.

segment size (integers)	SP2, p = 16, n = 2M						EM-X, p = 16, n = 2M					
	total	local sort	generate comp	generate comm	merge comp	merge comm	total	local sort	generate comp	generate comm	merge comp	merge comm
32	3.427	0.109	0.285	1.519	0.170	1.345	2.907	0.761	0.823	0.413	0.523	0.387
64	2.043	0.110	0.254	0.828	0.159	0.693	2.654	0.761	0.796	0.299	0.514	0.284
128	1.785	0.110	0.232	0.457	0.429	0.558	2.549	0.761	0.781	0.251	0.510	0.246
256	0.995	0.109	0.225	0.292	0.148	0.221	2.491	0.761	0.775	0.227	0.508	0.220
512	0.811	0.110	0.219	0.194	0.145	0.144	2.460	0.761	0.773	0.214	0.507	0.205
1K	0.742	0.110	0.217	0.158	0.145	0.113	2.453	0.761	0.770	0.213	0.506	0.203
2K	0.831	0.110	0.214	0.222	0.143	0.142	2.464	0.761	0.773	0.218	0.507	0.205
4K	0.805	0.110	0.215	0.204	0.143	0.134	2.453	0.761	0.774	0.219	0.506	0.193
8K	0.851	0.110	0.212	0.229	0.142	0.158	2.469	0.761	0.780	0.238	0.508	0.182
16K	0.823	0.110	0.212	0.214	0.143	0.144	2.488	0.761	0.787	0.238	0.506	0.196
32K	0.971	0.110	0.212	0.309	0.143	0.198	2.525	0.761	0.796	0.270	0.513	0.185
64K	0.981	0.110	0.212	0.313	0.142	0.204	2.529	0.761	0.830	0.235	0.508	0.195
128K	1.084	0.110	0.211	0.368	0.142	0.253	2.647	0.761	0.899	0.225	0.505	0.257

Table 1: Execution time (seconds) for p=16, n=2M (16 processors with 2 M integers)

segment size (integers)	SP2, p = 32, n = 4M						EM-X, p = 32, n = 4M					
	total	local sort	generate comp	generate comm	merge comp	merge comm	total	local sort	generate comp	generate comm	merge comp	merge comm
32	5.666	0.109	0.456	2.991	0.206	1.904	4.010	0.761	1.346	0.766	0.648	0.489
64	3.217	0.109	0.404	1.530	0.192	0.982	3.637	0.761	1.310	0.562	0.638	0.366
128	2.035	0.110	0.384	0.840	0.187	0.514	3.468	0.761	1.291	0.473	0.634	0.309
256	1.443	0.110	0.367	0.491	0.184	0.292	3.386	0.761	1.284	0.429	0.632	0.280
512	1.192	0.110	0.364	0.346	0.181	0.191	3.344	0.761	1.281	0.411	0.631	0.260
1K	1.253	0.110	0.359	0.406	0.179	0.199	3.322	0.761	1.281	0.402	0.630	0.248
2K	1.205	0.110	0.357	0.375	0.179	0.184	3.339	0.761	1.284	0.421	0.630	0.243
4K	1.290	0.110	0.355	0.437	0.178	0.209	3.358	0.761	1.286	0.435	0.631	0.245
8K	1.206	0.109	0.355	0.382	0.178	0.181	3.415	0.761	1.307	0.481	0.630	0.236
16K	1.379	0.110	0.357	0.492	0.183	0.237	3.516	0.761	1.317	0.569	0.629	0.240
32K	1.329	0.110	0.354	0.450	0.178	0.238	3.606	0.761	1.365	0.609	0.630	0.241
64K	1.577	0.110	0.354	0.611	0.177	0.323	3.630	0.761	1.396	0.568	0.629	0.276
128K	1.610	0.109	0.354	0.637	0.178	0.332	3.741	0.761	1.547	0.458	0.644	0.331

Table 2: Execution time (seconds) for p=32, n=4M (32 processors with 4M integers)

segment size (integers)	SP2, p = 64, n = 8M						EM-X, p = 64, n = 8M					
	total	local sort	generate comp	generate comm	merge comp	merge comm	total	local sort	generate comp	generate comm	merge comp	merge comm
32	8.923	0.109	0.687	5.309	0.250	2.569	5.372	0.761	1.994	1.234	0.772	0.611
64	4.983	0.109	0.607	2.700	0.231	1.336	4.853	0.761	1.949	0.929	0.763	0.451
128	3.012	0.109	0.577	1.420	0.225	0.681	4.610	0.761	1.926	0.784	0.759	0.380
256	2.135	0.110	0.554	0.861	0.219	0.391	4.506	0.761	1.918	0.741	0.757	0.329
512	2.020	0.110	0.546	0.807	0.218	0.340	4.450	0.761	1.915	0.704	0.756	0.314
1K	1.752	0.110	0.539	0.628	0.216	0.260	4.484	0.761	1.912	0.773	0.756	0.282
2K	1.671	0.109	0.536	0.584	0.216	0.227	4.492	0.761	1.925	0.766	0.754	0.286
4K	1.766	0.109	0.534	0.649	0.214	0.260	4.612	0.761	1.929	0.887	0.755	0.280
8K	1.644	0.110	0.534	0.558	0.216	0.227	4.718	0.761	1.957	0.950	0.756	0.294
16K	1.837	0.110	0.538	0.710	0.214	0.266	4.946	0.761	1.986	1.142	0.764	0.293
32K	1.760	0.109	0.532	0.662	0.214	0.242	4.930	0.761	2.033	1.085	0.760	0.291
64K	2.153	0.111	0.532	0.940	0.214	0.356	5.408	0.761	2.157	1.294	0.814	0.382
128K	1.990	0.109	0.532	0.801	0.214	0.334	5.403	0.761	2.214	1.258	0.774	0.396

Table 3: Execution time (seconds) for p=64, n=8 M (64 processors with 8M integers)

The absolute execution time listed in the above three tables can be misleading and should not be interpreted as they stand since SP2 runs at 66.7 MHz clock but EM-X at only 20 MHz. We listed them to show that for the given n and p, computation time remains relatively constant while communicant time changes as the segment size changes.

Table 4 lists some execution results of work reported in [5]. It is beyond the scope of this report to compare the efficiency of sorting algorithms themselves. We list the results to show that fine-grain bitonic sorting can be as efficient as radix sort which is known to be the fastest sorting method on parallel machines. Note also that our bitonic sorting is made synchronous at the end of every iteration to identify the computation time and communication time. The bitonic sorting can be even faster if barriers are removed.

n	n/p	Single-phase Radix sort [5]	Two-phase Radix sort [5]	Fine-grain Bitonic sort	Radix/Bitonic
64 K	4 K	0.474	0.107	0.033	3.24
128 K	8 K	n/r	n/r	0.058	
256 K	16 K	n/r	n/r	0.102	
512 K	32 K	n/r	n/r	0.196	
1 M	64 K	0.938	0.592	0.374	1.58
2 M	128 K	n/r	n/r	0.740	
4 M	256 K	n/r	n/r	1.596	
8 M	512 K	4.130	4.030	2.872	1.40
16 M	1 M	n/r	n/r	5.844	

Table 4: Comparison of parallel sorting on 16-processor SP2. Execution times are in seconds. n/r=not reported.

5 Analysis

Experimental results are analyzed in this section. We have made various attempts to make fair comparison as the two machines employ drastically different architectures and principles. First, we compare the absolute communication time. Second, we identify the individual components of time to contrast communication time and communication. We finally summarize the capability of overlapping computation with communication.

5.1 Absolute communication time

Two machines are built with different execution principles. Comparing such machines with absolute execution time can be misleading as they have different architectures. SP2 is based on POWER2 architectures with 6-issue superscalar machine (2 branch instructions, 2 integer instructions and 2 float instructions) running at 66.7 MHz. EM-X is a single-issue pipelined machine with drastically different architecture running at 20 MHz. SP2 clock is more than four times faster than EM-X. However, we feel that it will be still informative to find their performance based on absolute communication time.

Figure 8 depicts absolute communication times in terms of segment size. Computation time is *not* included in the plots as they remain essentially fixed for various segment sizes (Tables 1-3). The plots clearly indicate that the communication cost for the segment size of 2 to 128 integers is prohibitively expensive. For example, when n=512K, the maximum communication time t_{max} and minimum communication time t_{min} are 7.975 seconds at m=2 and 0.070 seconds at m=2048, respectively. The ratio of t_{max} to t_{min} is 7.975/0.070 = 114! This large communication ratio suggests that the communication overhead is the key factor contributing to this poor performance when the message size is small and the number of successive messages is excessively large.

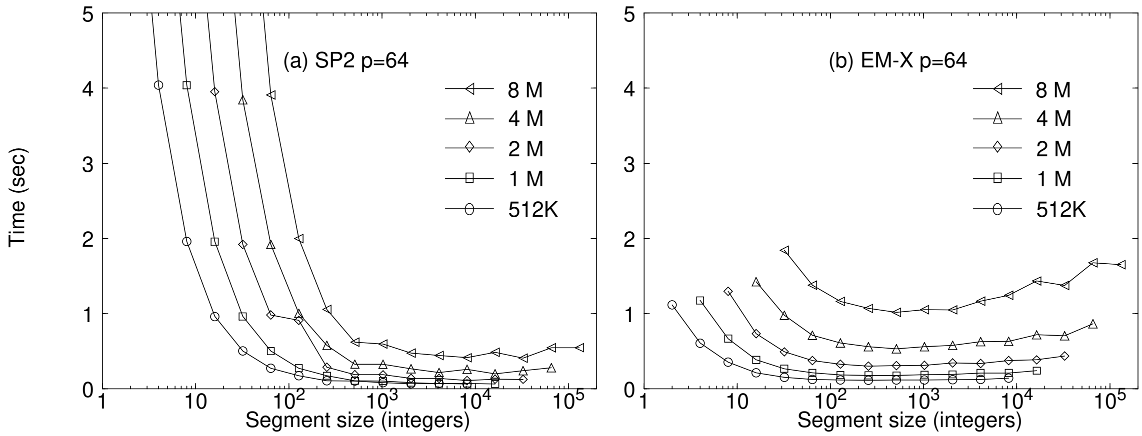

Figure 8: Communication time in seconds.

The LogP [11] and LogGP models [3] capture communication behaviors of distributed-memory machines for various size of messages. The LogGP model employs five terms, L, o, g, G, and P to explicate the behaviors of multiprocessors, where L is latency, o is overhead, g is gap between two consecutive messages, G is gap between bytes of the same message, and P is processors. Using the five terms, the LogGP model defines the communication time for a single message with k bytes as $t_{\text{comm}} = o + (k - 1)G + L + o$, where k is the number of bytes to send.

According to the LogGP model, we compute the communication time for m consecutive messages and the message size of s integers (4s bytes) as

$$t_{\text{comm}} = (o + (4*s - 1)G + L + o + g)*m \tag{1}$$

We have identified above that the maximum communication time for p=64 and n=512K occurs when the message size m is 2 integers and the number of messages (segments) is $s = n/p*m$ = 512K / 64*2 = 4096 segments. Using (1), we find the maximum communication time, $t_{\max}$, for m=4096 segments, and s=2 integers as

$$t_{\max} = (o + (4*2 - 1)G + L + o + g)*4096 = (2o + 7G + L + g)*4096 = 28672G + (2o + L + g)*4096 \tag{2}$$

The minimum communication time for p=64 and n=512K occurred when the message size m is 2K integers and the number of messages (segments) is $s = n/p*m$ = 512K / 64*2K = 4 segments. Using (1), we compute

$$t_{\min} = (o + (4*2048 - 1)G + L + o + g)*4 = (2o + 8191G + L + g)*4 = 32764G + (2o + L + g)*4 \tag{3}$$

Comparing (2) and (3), we find that the two G terms essentially cancel out since G is typically small on the order of 10 to 100 nano seconds. Dropping the two terms, we rewrite (2) and (3) as

$$t_{\max} = (2o + L + g)*4096 \tag{4}$$

$$t_{\min} = (2o + L + g)*4 \tag{5}$$

Now, we find that $t_{\max}$ and $t_{\min}$ are determined essentially by the number of segments (messages). Taking the ratio of (4) to (5), we have

$$t_{\max}/t_{\min} = (2o + L + g)*4096/(2o + L + g)*4 = 1024 = 1\text{K}.$$

We have seen above that the ratio of maximum communication time ($t_{\max}$ =7.975 sec) to minimum communication time ($t_{\min}$ =0.070 sec) is approximately 100. The ratio determined by LogP and LogGP models gave us 1000. It is not certain what contributed to this rather large difference. However, we believe that the difference between the experimental result of 100 and the analytical result of 1000 is due mainly to the network contention and some approximated values. Recall that this comparison is based on 64 processors which all together send out *up to* 512K integers (2M bytes) in each iteration of bitonic sorting. This large amount of communication can easily congest the network with unpredictable consequence.

The communication behaviors of EM-X, however, are substantially different from SP2. The variation of maximum communication time to minimum communication time is very small. For p=64 and n=512K, the maximum and minimum communication times are 1.5 seconds and 0.4 seconds, resulting in the ratio 1.5/0.4 = 3.8. For p=64 and n=8M, they are 2.5 sec and 1.5 sec, resulting in the ratio 2.5/1.5 = 1.7. The two ratios are very close compared to the SP2 results. This closeness is attributed by several factors: First, the 1 word-wide network is four times wider than the 1 byte-wide SP2. Second, message overhead is fixed to 1 clock cycle per word. Third, the by-passing mechanism allows to service remote read/write requests without consuming the main processor cycles.

We have plotted in Figure 9 the rate of change of communication time to further identify the communication behavior. It is clear from the figure that the rate of change of communication time for SP2 is rather fluctuating while the one for EM-X is essentially consistent as indicated by a *thick* line (all the three problems sizes are superimposed). While it would be difficult to distinguish whether it is the small problem size which gives the most consistent or the largest problem size which does, it is unnecessary since the best case will occur when all the lines superimpose. We shall identify below the components which contribute to this rate of change in communication.

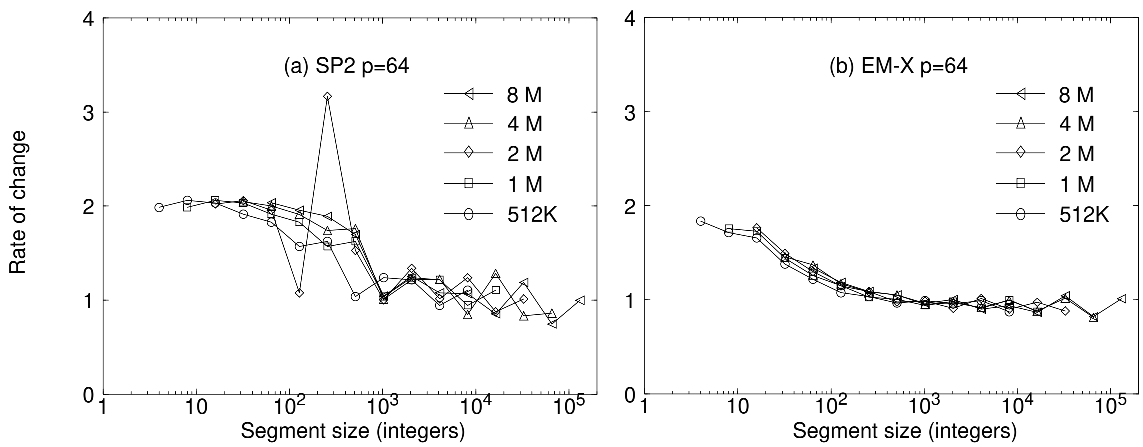

Figure 9: Ratio of change of communication time.

5.2 Comparison of computation time and communication time

Figure 10 shows the details of execution time. Four components are shown in the plots: generate computation time, generate communication time, merge computation time, and merge communication time. Local sorting time is not included in the plots since it does not contribute to communication. As we have seen in Tables 1 to 3, both computation times for generate and merge are essentially fixed. The only varying components are communication time. All the plots show that merge communication time is roughly less than half the generate communication time. Recall that generate has $(\log p - 1)^2$ iterations while merge has $\log p$ iterations. For p=64, the number of iterations for generate doubles the merge computation time. This is indicated by plots (e) and (f).

For small n and p, the impact of segment size on communication time is not significant as shown in Figure 10(a) and (b), except when the segment size is excessively small below 64 integers. As n and p increase, the impact of segment size on communication time, however, becomes significant as indicated in plots (e) and (f). The plots clearly demonstrate that the communication time prefers a segment size of approximately 1K integers. This observation, however, is different from the NAS communication benchmark [18]. The NAS CB indicated that when the message size is approximately 512 to 1K bytes, the overlapping capability becomes the maximum. The difference between 1K integers of our findings and 1K bytes of NAS findings is not negligible. We found, however, that the NAS results are based on the ratio of computation to communication being approximately 1.5 while our ratio of computation to communication is approximately the same which leads to yet another interesting observation.

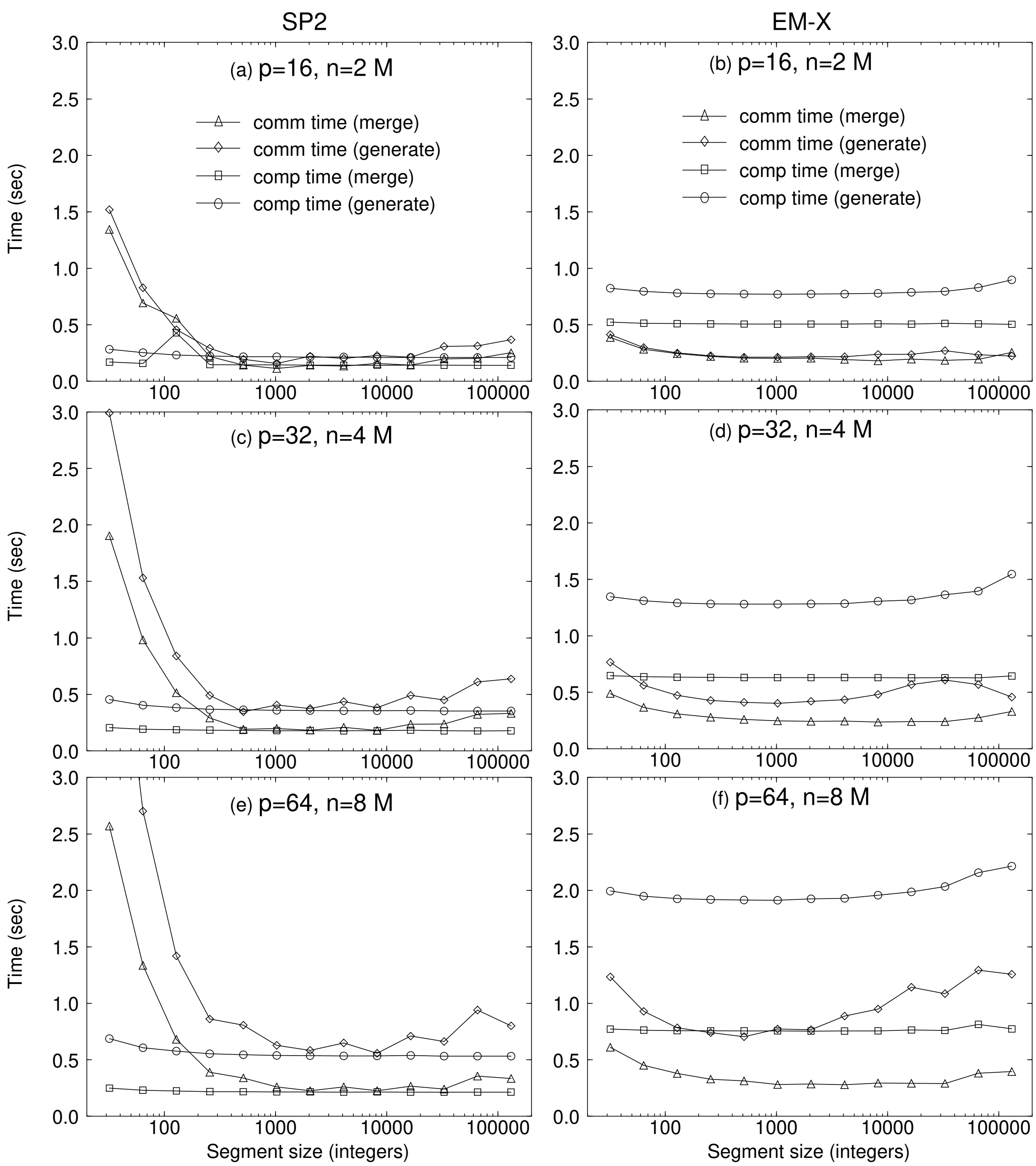

Figure 10: Details of execution time.

The plots in Figure 10 indicate that the maximum overlapping occurs when the computation time is approximately the same or larger than communication time for SP2. All the three plots (a), (c), (e) indicate that the minimum communication occurs when the computation time is approximately the same or larger than communication time. The main reasons, we believe, are that there are enough computation to mask off the communication time, and that the SP2 buffer size can hold up to 4K bytes for two directions (sending/receiving), or 2K bytes for each direction.

EM-X on the other hand shows that is communication is essentially insensitive to the size of segment and the amount of computation time. In fact, the computation time is always higher than the communication time. Furthermore, the computation time of EM-X is about four times higher than the SP2 computation time. The main reason is that the EM-X's clock is 20 MHz while the SP2's clock is 66.7 MHz. Clearly, the three times faster clock without even considering the superscalar capability gives three times faster computation time than EM-X.

5.3 Efficiency of overlapping

The main purpose of this report is to investigate the capability of overlapping computation with communication. As we have pointed out, we do not use the EM-X's multithreading capability. Since SP2 is not a multithreaded machine, it would be unfair to compare a multithreading with a non-multithreading machine. To maintain this fair comparison, we forced the loops to execute synchronously. At the end of each loop body (*j*-loop for generate and *i*-loop for merge), we inserted a barrier so that all other processors wait until the same loop iterations are complete. Doing this, we can precisely measure the effects of overlapping. We define the efficiency of overlapping as

$$E = (T_{\mathrm{comm},1} - T_{\mathrm{comm},s}) / T_{\mathrm{comm},1}$$

where $T_{\mathrm{comm},1}$ is communication time for 1 segment, and $T_{\mathrm{comm},s}$ is communication time for s segments. We use $T_{\mathrm{comm},1}$ as the basis for efficiency computation. If s=1, the entire data is sent and received, with no possibility of overlapping since computation starts only when the entire data is received. Figure 11 precisely identifies the capability of overlapping for SP2 and EM-X.

Figure 11(a) shows that for p=64, SP2 overlaps approximately 30% of communication time while EM-X does 40%. As the plots indicate, SP2 gives very little overlapping for small problem sizes of 512K to 2M. However, EM-X continuously demonstrates the effectiveness of overlapping capability for all problem sizes. Since SP2's overlapping capability is low, we plotted with larger problem sizes of up to 64M, as shown in Figure 11(c). It is not surprising that SP2 now exhibits over 40% overlap for the data size of up to 64M. This result is not surprising because the amount of computation is quadrupled. However, it should be noted from the figure that while SP2 yielded over 40% overlapping at the segment size of 1K, it also gives *negative* overlapping when the segment size is less than 128 integers.

On the other hand, EM-X shows little sensitivity for a small segment size of over 32 integers. If the problem sizes of 64M integers are used for EM-X, we believe EM-X will also exhibit similar capability SP2 demonstrated since EM-X exhibits no anomaly in terms of overlapping. Overlapping capability simply increased further as the problem size increased. This continuous increase in overlapping will hold since larger data size will increase the computation time. As we have seen earlier in Figure 10, computation time is always greater than communication time for EM-X, and therefore larger problem sizes will only make the computation time longer, thereby making communication time relatively smaller than computation time.

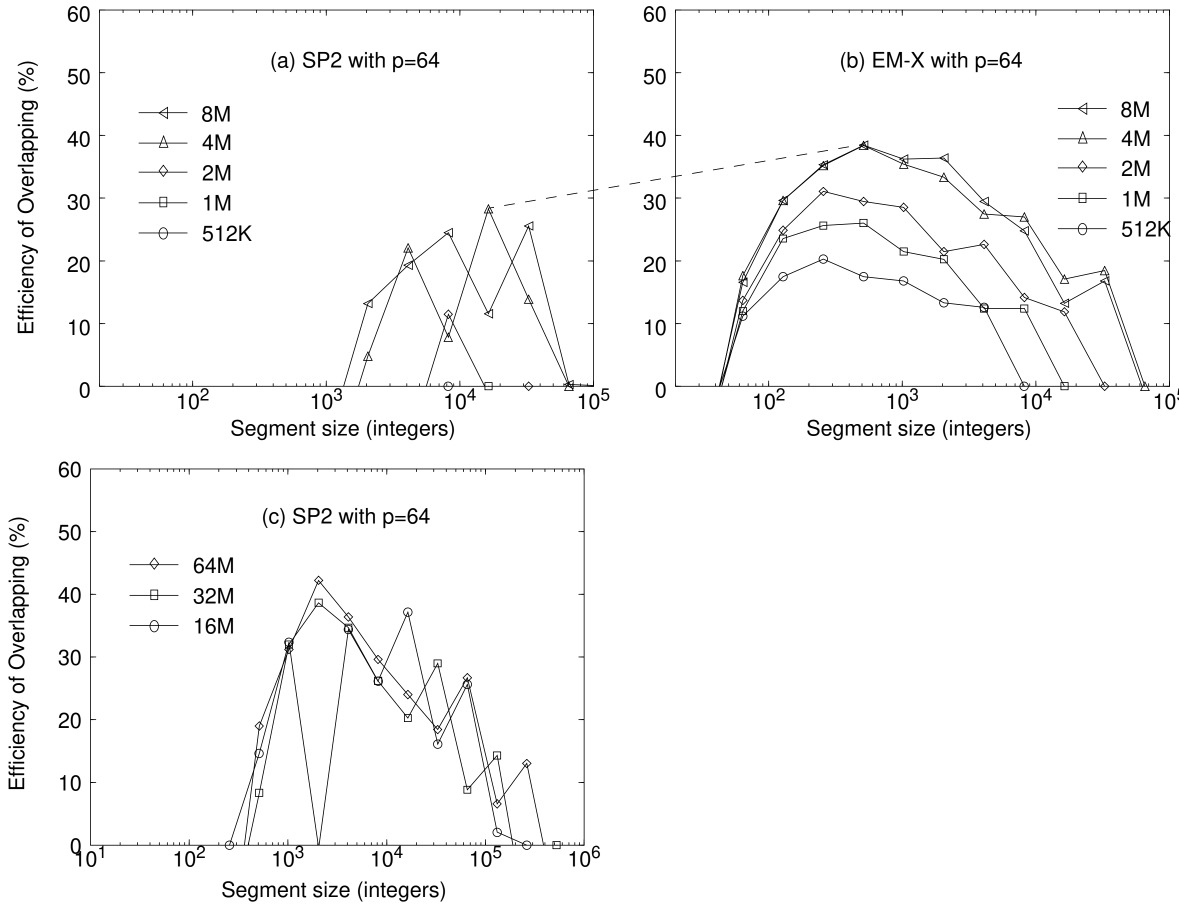

Figure 11: Efficiency of overlapping.

6 Conclusions

Reducing communication overhead is key to realizing high performance computing on distributed-memory multiprocessors. There are numerous approaches reported to reduce communication overhead, including efficient parallel algorithms, latency hiding techniques by multilevel cache, latency tolerating techniques by multithreading, and runtime computation and data migration. This report has examined the capability of overlapping computation with communication for two multiprocessors, IBM SP2 and EM-X distributed-memory multiprocessors. Each machine provides hardware support for overlapping computation with communication. IBM SP2 provides a *communication coprocessor* Intel i860 with a bidirectional FIFO *message buffer* of 4K bytes to hold incoming and outgoing messages. This small buffer together with i860 has been designed to take the burden off the main processor POWER2. The EM-X provides a *by-passing* mechanism which does *not* consume main processor cycles when servicing remote read/write requests.

To perform experiments, we have selected bitonic sorting and implemented on both machines. To measure both the computation time and communication time, a barrier is inserted at the end of every iteration which forced the sorting algorithm to execute synchronously. Various message sizes have been selected to observe the behaviors of overlapping capability. We used up to 64M integers for IBM SP2 and up to 8M integers for EM-X (an SP2 wide node has over 128M bytes/processor while EM-X has 4M bytes/processor).

According to the experimental results, we have made the following key observations: First, the minimum communication time occurred for both machines when the message size is approximately 512 to 2K integers (4K bytes). We have compared our experimental results with the analytical results using the LogP and LogGP models and found that they do not seem to agree. The main reason for the discrepancy stemmed from the fact that the models do not capture runtime dependent network contention which is indeed very difficult to predict. Our observation also disagrees with the NAS communication benchmarking where the maximum overlap can occur when the size is roughly 1K bytes. However, we find that the ratio of computation to communication for the NAS communication benchmarking results is approximately 1.5, which is much higher than what we had in our experiments. We have found that the communication time is dependent essentially on the number of messages sent/received.

Second, maximum overlapping occurred when the computation time was approximately the same or larger than the communication time. For up to 64 processors with 8M integers, SP2 overlapped 30% of the total communication time while EM-X did 40% of communication time. SP2 showed that the overlapping capability is highly sensitive to the small message size of 2 to 128 integers and over 32K integers. The main reason that SP2 is sensitive to the ratio of computation time to communication time is because of the message buffer used by the communication co-processor i860 and 8-bit wide network. The buffer called "bidi" (bidirectional) FIFO can hold up to 4K bytes. On the other hand, EM-X was found almost insensitive to message size, yielding high overlap for various different message sizes. The main reason is due to the facts that EM-X incurs very little overhead for sending messages due to its fixed-size two-word packets, 1 word-wide network, and by-passing remote memory servicing mechanism.

Since SP2 can execute larger data, we have executed with larger data size. When the data size was increased to 1M integers/processor, SP2 yielded over 40% overlapping of the communication time. This increase was due to the fact that computation is now much larger than communication time. Since EM-X did not show any anomaly of overlapping capability, we can safely predict that EM-X will also give corresponding overlapping capability of over 50% for larger data size. We have attempted to identify the overlapping capability of the two rather different distributed-memory multiprocessors. Our attempts have been verified with various experimental results. To further confirm our results, we are planning to perform similar procedures with different problems since identifying the communication overlapping capability is key to high performance computing.

Acknowledgments

Andrew Sohn is supported in part by the NASA University Joint Venture in Research Program and the Foreign Researcher Program of the Agency of Industrial Science and Technology of the Ministry of International Trade and Industry, Japan. The SP2 machines installed at the NASA Langley Research Center and the NASA Ames Research were used to perform part of the experiments reported in the paper.

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. the 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] A. Alexandrov, M. Ionescu, K. Schauser, and C. Scheiman, "LogGP: Incorporating Long Messages into the LogP Model - One step closer towards a realistic model for parallel computation," in *Proc. of the 7th ACM Symposium on Parallel Algorithms and Architectures*, Santa Barbara, CA, July 1995, pp. 95-105.

[4] R. Alverson et al. The Tera computer system. In *Proc. of the 1990 International Conference on Supercomputing*, Amsterdam, Netherlands, June 1990, ACM, pp.1-6

[5] D. Bader, D. Helman, and J. JáJá, "Practical Parallel Algorithms for Personalized Communication and Integer Sorting," Technical Report, University of Maryland, December 1995.

[6] A. Bar-Noy and S. Kipnis, "Designing Broadcasting Algorithms in the Postal Model for Message-Passing Systems," in *Proc. of the 4th ACM Symposium on Parallel Algorithms and Architectures*, June 1992.

[7] K. Batcher, "Sorting Networks and Their Applications," in Proc. the AFIPS Spring Joint Computer Conference 32, Reston, VA, 1968, pp.307-314.

[8] G. E. Blelloch, C. E. Leiserson, B. M. Maggs, C. G. Plaxton, S. J. Smith, and M. Zagha, "A Comparison of Sorting Algorithms for the Connection Machine CM-2," in *Proc. of the 7th ACM Symposium on Parallel Algorithms and Architectures*, July 1991, pp.3-16.

[9] 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.

[10] 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.

[11] 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.

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

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

[14] W. C. Hsieh, P. Wang, and W.E. Wiehl, "Computation Migration: Enhancing Locality for Distributed-Memory Parallel Systems," in Proc. of the 4th ACM Symposium on Principles and Practice of Parallel Programming, San Diego, CA, May 1993, pp.239-248.

[15] 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 the Third ACM/IEEE Conference on Parallel Architectures and Compilation Techniques,* Limassol, Cyprus, June 27-29, 1995, pp. 59-68.

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

[17] 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 International Symposium on Computer Architecture*, Santa Margherita Ligure, Italy, June 1995, pp.14-23

[18] Message Passing Interface Forum, MPI: Message-Passing Interface Standard, Version 1.1, Technical Report, University of Tennessee, Knoxville, TN, June 1995.

[19] W. C. Saphir, The NAS Communication Benchmarks Revision 0.9: Overlap Test, Personal Communication, NASA Ames Research Center, February 1996.

[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] 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 Par-*

allel Processing Symposium, Cancun, Mexico, April 1994, pp.630-636.

[22] A. Sohn, “Communication Efficient Bitonic Sorting on SP2 Multiprocessor,” *Technical Report*, NJIT CIS Dept., 96-10, March 1996.

[23] 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 the 10th IEEE International Parallel Processing Symposium*, Honolulu, Hawaii, April 1996.

[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 SP2 Communication Subsystem,” Technical Report, IBM T. J. Watson Research Center, August 1994.

[25] C. B. Stunkel, D. G. Shea, B. Abali, M. M. Denneau, P. H. Hochschild, D. J. Joseph, B. J. Nathanson, R. A. Swetz, R. F. Stucke, M. Tsao, and P. R. Varker, “Architecture and Implementation of Vulcan,” in *Proceedings of the 8th IEEE International Parallel Processing Symposium*, Cancun, Mexico, April 1994, pp.268-274.

[26] L. G. Valiant, “A Bridging Model for Parallel Computation,” *Communications of the ACM* 33 (8), August 1990, pp.103-111.