ABSTRACT

PORTING THE SISAL FUNCTIONAL LANGUAGE TO DISTRIBUTED-MEMORY MULTIPROCESSORS

by
Jui-Yuan Ku

Parallel computing is becoming increasingly ubiquitous in recent years. The sizes of application problems continuously increase for solving real-world problems. Distributed-memory multiprocessors have been regarded as a viable architecture of scalable and economical design for building large scale parallel machines. While these parallel machines can provide computational capabilities, programming such large-scale machines is often very difficult due to many practical issues including parallelization, data distribution, workload distribution, and remote memory latency.

This thesis proposes to solve the programmability and performance issues of distributed-memory machines using the Sisal functional language. The programs written in Sisal will be automatically parallelized, scheduled and run on distributed-memory multiprocessors with no programmer intervention. Specifically, the proposed approach consists of the following steps. Given a program written in Sisal, the front end Sisal compiler generates a directed acyclic graph(DAG) to expose parallelism in the program. The DAG is partitioned and scheduled based on loop parallelism. The scheduled DAG is then translated to C programs with machine specific parallel constructs. The parallel C programs are finally compiled by the target machine specific compilers to generate executables.

A distributed-memory parallel machine, the 80-processor ETL EM-X, has been chosen to perform experiments. The entire procedure has been implemented on the EM-X multiprocessor. Four problems are selected for experiments: bitonic sorting, search, dot-product and Fast Fourier Transform. Preliminary execution results indicate that automatic parallelization of the Sisal programs based on loop parallelism is effective. The speedup for these four problems is ranging from 17 to 60 on a 64-processor EM-X. Preliminary experimental results further indicate that programming distributed-memory multiprocessors using a functional language indeed frees the programmers from low-level programming details while allowing them to focus on algorithmic performance improvement.

PORTING THE SISAL FUNCTIONAL LANGUAGE TO DISTRIBUTED-MEMORY MULTIPROCESSORS

by
Jui-Yuan Ku

A Dissertation
Submitted to the Faculty of
New Jersey Institute of Technology
in Partial Fulfillment of the Requirements for the Degree of
Doctor of Philosophy

Department of Computer and Information Science

May 1999

APPROVAL PAGE

PORTING THE SISAL FUNCTIONAL LANGUAGE TO DISTRIBUTED-MEMORY MULTIPROCESSORS

Jui-Yuan Ku

Dr. Andrew Sohn, Dissertation Advisor — Date
Associate Professor of Computer and Information Science Department,
NJIT, Newark, NJ

Dr. David Nassimi, Committee Member — Date
Associate Professor of Computer and Information Science Department,
NJIT, Newark, NJ

Dr. James McHugh, Committee Member — Date
Professor of Computer and Information Science Department,
NJIT, Newark, NJ

Dr. Mengchu Zhou, Committee Member — Date
Associate Professor of Electrical and Computer Engineering,
NJIT, Newark, NJ

Dr Yunheung Paek, Committee Member — Date
Assistant Professor of Computer and Information Science Department,
NJIT, Newark, NJ

BIOGRAPHICAL SKETCH

Author: Jui-Yuan Ku

Degree: Doctor of Philosophy

Date: May 1999

Undergraduate and Graduate Education:

- Doctor of Philosophy in Computer and Information Science,
  New Jersey Institute of Technology, Newark, NJ, 1999
- Master of Science in Computer and Information Science,
  New Jersey Institute of Technology, Newark, NJ, 1994
- Bachelor of Science in Applied Mathematics,
  National Chung-Hsin University, Taichung, Taiwan R.O.C, 1987

Major: Computer and Information Science

Presentation and Publications:

A. Sohn, Y. Paek, J. Ku, Y. Kodama, Y. Yamaguchi,

"Communication Studies of Single-threaded and Multithreaded Distributed-Memory Multiprocessor," Submitted to the upcoming *IEEE Transactions on Computers Special Issue on Multithreading*, December 1998.

A. Sohn, Y. Kodama, J. Ku, M. Sato, H. Sakane, H. Yamana, S. Sakai, Y. Yamaguchi,

"Instruction-Level Multithreading in the EM-X Multithreaded Distributed-Memory Multiprocessor," Submitted to the upcoming *IEEE Transactions on Computers Special Issue on Multithreading*, December 1998.

A. Sohn, Y. Paek, J. Ku, Y. Kodama, Y. Yamaguchi,

"Communication Studies of Three Distributed-Memory Multiprocessors," *Proceeding of IEEE Symposium on High Performance Computer Architecture*, Orlando, Florida, pp.310-314, January 1999.

A. Sohn, Y. Kodama, J. Ku, M. Sato, Y. Yamaguchi,

"Tolerating Communication Latency through Dynamic Thread Invocation in a Multithreaded Architecture," *Springer-Velag Lecture Notes in Computer Science*, In press.

A. Sohn , J. Ku, Y. Kodama, H. Sakane, M. Sato, H. Yamana, Y. Yamaguchi,

"Analysis of Distribution Strategies on the EM-X Multithreaded Multiprocessor(A Preliminary Summary)," *Proceedings of the Workshop on Multi-Threaded Execution,Architecture and Compilation*, Las Vegas, Nevada, February 1-4, 1998.

A. Sohn, Y. Kodama, J. Ku, M. Sato, H. Sakane, H. Yamana, S. Sakai, Y. Yamaguchi,

"Fine-Grain Multithreading with the EM-X Multiprocessor," *The Ninth ACM Symposium on Parallel Algorithms and Architectures*, Newport, Rhode Island, pp.189-198, June 1997.

A. Sohn, J. Ku, Y. Kodama, M. Sato, H. Sakane, H. Yamana, S. Sakai, Y. Yamaguchi,

"Identifying the Capability of Overlapping Computation with Communication," *Proceeding of ACM/IEEE Conference on Parallel Architectures and Compilation Techniques*,Boston, MA, pp. 133-138,October 1996.

This dissertation is dedicated to
my beloved family

ACKNOWLEDGMENT

I would like to extend my sincere gratitude to Dr. Andrew Sohn, who not only served as my research supervisor providing valuable and countless resources, guidance, insight, and intuition, but also constantly gave me support, encouragement, and reassurance. Special thanks are given to Dr. David Nassimi, Dr. Yunheung Paek, Dr. James McHugh and Dr. Mengchu Zhou for serving as members of my committee. I thank the EM-X group members Yuetsu Kodama, Mitsuhisa Sato, Hirofumi Sakane, Hayato Yamana, Shuichi Sakai, and Yoshinori Yamaguchi of the Electrotechnical Laboratory of the Ministry of International Trade and Industry of Japan for providing access to the EM-X multiprocessor.

I would also like to thank my family for their love and moral support. Moreover, I wish to express my appreciation to my parents for their continuous support during these years. Without their support and encouragement, my accomplishment would not be possible.

Finally, I would like to express my deep appreciation to my loving girl friend, Chia-Chen Yu, for her love, understanding, and indefatigable efforts in helping me to finish this work.

TABLE OF CONTENTS

Chapter Page

TABLE OF CONTENTS
(Continued)

TABLE OF CONTENTS
(Continued)

LIST OF TABLES

LIST OF FIGURES

LIST OF FIGURES
(Continued)

LIST OF FIGURES
(Continued)

CHAPTER 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 increasing demand for high performance computing. In these paradigms of machine architecture, it is deemed relatively straightforward to increase the machine's capability simply by adding more processors to the system incrementally as required. There are various research prototypes as well as commercial machines having this distributed-memory/message-passing paradigm such as SGI/Cray T3E[10,42], ETL EM-X[11,12], Fujitsu AP1000+, IBM ASCI Blue, Intel ASCI[13], SP-2[14,15], RWC-1[20], Tera MTA[16,21].

Parallel computing is becoming increasingly ubiquitous in recent years. The complexity of the problems all researchers try to deal with increases proportionally. Researchers are always try to "byte" off more than their computers can chew. One of the resources to get exhausted is local memory. This situation is alleviated by having access to the aggregate memory made available by distributed computing environments.

Even though various kinds of distributed-memory multiprocessors can certify that all problems with any problem size can be solved, distributed-memory multiprocessors do not leave the burden of programming parallel machines for all programmers. Programming parallel machines involves numerous practical concerns. The problems under consideration need to be carefully studied to identify potential parallelism. Suitable algorithms for the problems need to be developed to manifest their potential parallelism. Programming must be done in a way to effectively realize the algorithms and harness their parallelism for the target machines. Each machine's special capabilities, if any, should be utilized to improve the performance. The programmer must have a good understanding of the characteristics of the problem as well as of the underlying machine architecture. Programmability that refers to the ease of programming parallel machines is one of the keys to success; it is particularly important for problems requiring complex synchronization and parallelization.

High Performance Fortran (HPF) aims at providing programmability for distributed-memory multiprocessors[22]. Data parallelism in large arrays is a typical target parallelism used in HPF[23]. A simple description of data distribution such as blocked or cyclic helps to keep programmers away from the low-level details of data distribution and communication issues. Programmers can instead concentrate on the algorithmic issues of the problem under consideration and after analyzing the behavior of the program, can partition data and allocate it to processors so that runtime data movement is minimized.

Just like other parallel programs written in imperative languages, those parallel programs written in HPF also require explicit parallel constructs such as message-passing or synchronization primitives. The programmers need to understand the low level implementation details of the machine architecture, data distribution, synchronization, and so on. Because these programs depend highly on machine architecture and machine provided parallel constructs, portability among different parallel systems is difficult.

Message Passing Interface (MPI) provides portability among different kinds of distributed-memory multiprocessors. MPI is a widely used standard for writing message passing programs; such interface establishes a practical, portable and flexible standard for message passing. Message passing is the standard means of communication among distributed memory processes.

MPI really releases one burden from those programmers who are using imperative languages as the language to develop their parallel algorithms. All concerns remain except the portability. We all know that it is not easy to write parallel programs in imperative languages since parallel programming in those languages is not straightforward. It will cost a lot of time and money to train the programmers to do this kind of parallel programming. Even an experienced programmer will spend some time to develop a suitable parallel algorithm to reach a reasonable performance for solving any benchmark problem. In other words, parallel programming in imperative languages is a time-consuming process, especially for an unexperienced programmer to reach an expected performance in distributed-memory multiprocessors. High performance is difficult to obtain in those distributed-memory multiprocessors due to latency caused by data distribution. It is possible to distribute data and workload for a particular problem to fit a particular distributed-memory architecture to improve the overall performance. People spend years learning how to achieve high performance, which can only be obtained from practical experience.

Above all, three problems occur when we program parallel machines: programmability, portability, and performance. Instead of seeking the solution from the same basic assumptions, we look in another direction.

Imperative languages have been designed based on the von Neumann architecture computers. Although the imperative style of programming has been found acceptable by most computers, its heavy reliance on the underlying architecture is an unnecessary restriction on the process of software development. It is natural to use imperative languages on sequential machines since they are all von Neumann architectures, but using these languages to do the programming in the parallel machines will cause problems.

Functional languages have been used to overcome the difficulties in parallel programming. They are different from programs written in imperative languages. The programs of functional programming languages are function definitions and function application specifications, and execution consists of evaluating the function applications. The execution of a function always produces the same result when given the same parameters. This is called referential transparency. It makes the semantics of functional languages simple.

Functional programs can be executed by first translating them into graphs. These graphs can then be executed through a graph reduction process, which can be done with a great deal of concurrency that was not specified by the programmer. The graph analysis can easily expose many opportunities for concurrent execution. Therefore, programs in functional languages can be divided into concurrent parts dynamically by the execution system, which extracts maximum parallelism and favors automatic parallelization. These factors make concurrency far simpler for the programmer. The programmer does not need to take care of explicit parallel constructs such as message-passing communication operations or synchronization primitives. Much of the responsibility for concurrency is placed on the execution system, making programs easy to read and to write.

The functional programming language does computation by applying functions given parameters. In a strict functional langauge, the parameters to a function are always evaluated before it is invoked while a non-strict functional langauge evaluate the parameters only when their values are actually required. Programmers need not be concerned with variables, because memory cells need not be abstracted into the language. This leads to the definite advantage of functional programming: high level programming. In other words, functional programming can be done without the knowledge of the low level implementation details of machine architecture, data distribution, synchronization, and so on. Hence, functional programming requires less labor than programming in an imperative language. Also, programmers can concentrate on the problem solving for algorithmic improvement or the quality of solutions: performance.

The non-strict functional language Id[24] has been ported to the Monsoon data-flow multiprocessor[25], and has demonstrated that functional language can be an alternative to parallel programming. Glasgow Parallel Haskell(GPH) is a non-strict parallel functional programming language. GPH is an extension of Haskell, adding just two new primitives to the language, namely, a form of parallel composition (par), and sequential composition (seq). With judicious use of par and seq it is possible to express how a program should be evaluated in parallel. The great thing about GPH is that it's very easy to take a sequential program written in Haskell, and add par and seq annotations to make it parallel. However, getting the best parallelism out of the code isn't always trivial from this process, so we do not use it. The strict functional language SISAL[26] and its optimizing compiler OSC[27] have demonstrated that functional languages can yield high performance on shared-memory machines[1]; thus, SISAL language was selected as the developing language in this dissertation.

The SISAL language project [1] began as a collaborative effort by Lawrence Livermore National Laboratory, Colorado State University, University of Manchester, and Digital Equipment Corp in 1983. They noticed that scientists must parallelize their codes and move to multiprocessor systems, so they developed the Optimizing SISAL Compiler (OSC) which automatically parallilizes programs written in sequential languages. OSC creates several architecture independent intermediate forms corresponding to a compiler stage, and then develops a code generator and run time systems for any specific machine. We use the 80 processor EM-X multithreaded multiprocessor as our target machine.

SISAL is a functional parallel programming language. It combines modern language features with readable syntax and mathematically sound semantic foundations to provide the easiest tool for parallel programming. Its optimizing compiler and runtime support system software provide portability, high performance, and determinate execution behavior. SISAL has been shown to port the entire language system fairly easily to new machines. SISAL is running today on Unix-based uniprocessors, the Cray X/MP [4], the Warp [5], the Connection Machine[48], the Mac II, a variety of data-flow machines [8,9] and conventional shared-memory multiprocessors [2,3,43]. Sisal programs in conventional shared-memory multiprocessors show the competitive performance compared with those programs written in FORTRAN.

It is clear that SISAL language needs to be extended to distributed-memory machines. Porting SISAL functional language to distributed-memory machine environments will advance the state-of-the-art in functional programming and large-scale parallel computing. The previous work related to this dissertation is Pande et. al's work which is briefly described below. SISAL was targeted to Intel Touchstone i860 systems[17,18,19], by mapping the functional parallelism in its intermediate IF1 representation. They modified the IF2PART to perform a compile time scheduling method to decide the number of active processors required and get the information about which task was assigned to which processor. Using this information from IF2PART, efficient codes were generated in IF2GEN(code generator) for the target machines by generating programs with a case statement which used the processor number to identify the code to be executed by the corresponding processor. The code generated from IF2GEN will feed into the RUNTIME system, which was modified to support message passing communication between different processors to send and receive the data values. Two communication primitives," pick" and "put" are defined and will operate asynchronously so that the sending and receiving processors do not have to rendezvous, which exposes more parallelism.

From Pande's work, we found that the required processors will be determined by IF2PART from a new compile time scheduling method. By using this method, there is a trade-off between the schedule length and the required number of processors. The functional parallelism enforced the information about which task was assigned to which processor and the required number of active processors is determined by the compile time scheduling method. The functional parallelism is not used in our approach since it was limited to the number of different functions of the whole program. Therefore, we propose the loop parallelism to port the SISAL to the distributed-memory multiprocessor.

In this dissertation, we exploit data parallelism so that we can spread the data among processors according to a certain distribution strategy. The main constraint of the number of processors will be the number of processors that the target system can provide. In other words, we can use as many processors as the system can support, so that the performance we can reach will be based mainly on all the parallelisms we can get. That is the reason why we are using outmost loop parallelism. Another benefit of using the SISAL language is that the SISAL language itself supports the parallel loop, so that programmers can easily predict the performance from the programs.

Since we are using loop parallelism, both data distribution and workload distribution must be considered in compiler time. In our experiment, two typical distribution strategies, blocked and cyclic, are used. Take the data distribution as an example. Given the data of size N and NPE processors, blocked distribution divides the data into NPE blocks, each of which holds consecutive M=N/NPE elements. Cyclic distribution divides the N elements into NPE blocks but in a round-robin fashion. The resulting data are therefore not consecutive in terms of index. It goes without saying these concepts, blocked and cyclic, are also applied to the workload distribution.

The proposed approach takes the existing Optimizing SISAL Compiler and modifies its last two parts of it to work for distributed-memory multiprocessors. Specifically, given the partitioned dataflow graph of a SISAL program, the proposed approach analyzes loops to exploit loop parallelism. A loop slicing technique is applied to divide the loop into pieces according to a certain distribution strategy. Once the loop is divided into pieces with assumptions, the machine specific C codes with Single Program Multiple Data (SPMD) execution model will be generated. At the time of generating C codes, parallel constructs will be inserted to make sure all active processors operate their corresponding program portions correctly.

In our approach, we only modified IF2GEN and RUNTIME. Users can set “blocked” data and workload distribution or “cyclic” data and workload distribution as an option of IF2GEN makefile. According to the assigned data distribution and workload distribution, data and workload are spread out among processors. IF2GEN will generate the code according to the outmost loop parallelism. Each loop slice will be run in its corresponding processor without using case statement since we are using the Single Program

Multiple Data (SPMD) execution model. The communication primitives are added whenever the reference data are checked and found to be not locally available according to the given data distribution. We are using by-passing one-sided remote read for ETL EM-X. All communication primitives are defined in the RUNTIME system so that the parallel code generated from IF2GEN can be fed into the RUNTIME system which will schedule and execute the code.

This dissertation is organized as follows: Chapter 2 addresses the background of distributed-memory multiprocessors ETL EM-X and Cray T3E. Chapter 3 briefly describes Sisal functional language, Optimizing Sisal Compiler(OSC), and the approach Pande used for distributed-memory multiprocessors: Intel Touchstone i860 systems. Chapter 4 discusses various kinds of message passing protocols. Chapter 5 analyzes the distribution strategies. Chapter 6 illustrates the approach by addressing the code generation and the RUNTIME system for distributed-memory multiprocessors. Chapter 7 gives the preliminary results and discussions. Chapter 8 states the conclusions. Chapter 9 outlines the future work.

CHAPTER 2

DISTRIBUTED-MEMORY MULTIPROCESSORS

It is easy to increase the distributed-memory multiprocessor's capability simply by adding more processors to the system as required. Therefore, it has a definite advantage in doing the high performance computing according to the increasing demand of large parallel machines. The drawback of this architecture is, it is more difficult to program than the shared-memory multiprocessor since data need to be distributed to all processors.

There are various research prototypes as well as commercial machines having this distributed-memory/message-passing paradigm such as SGI/Cray T3E[10,42],ETL EM-X[11,12], Fujitsu AP1000+,IBM ASCI Blue, Intel ASCI[13], SP-2[14,15], RWC-1[20], Tera MTA[16,21], etc.

The fine-grain multithreading principles are discussed in this chapter along with their realization of the EM-X multithreaded multiprocessor. The machine architecture is briefly presented to illustrate how the multithreading principles are realized in EM-X. The software and hardware support for multithreading are then listed. They include a by-passing mechanism for direct remote reads and writes, runtime thread invocation, hardware FIFO thread scheduling, and dedicated instructions for generating fixed-sized communication packets.

2.1 A Multithreaded Distributed Memory Multiprocessor - ETL EM-X

EM-X is a multithreaded distributed memory multiprocessor and the capabilities of mutithreading could help tolerate memory latency via context switch. However, we did not use the multithreading capabilities of this machine in this dissertation since we are focusing on the general parallel machines. But, we are going to use the mutithreading capabilities in the future.

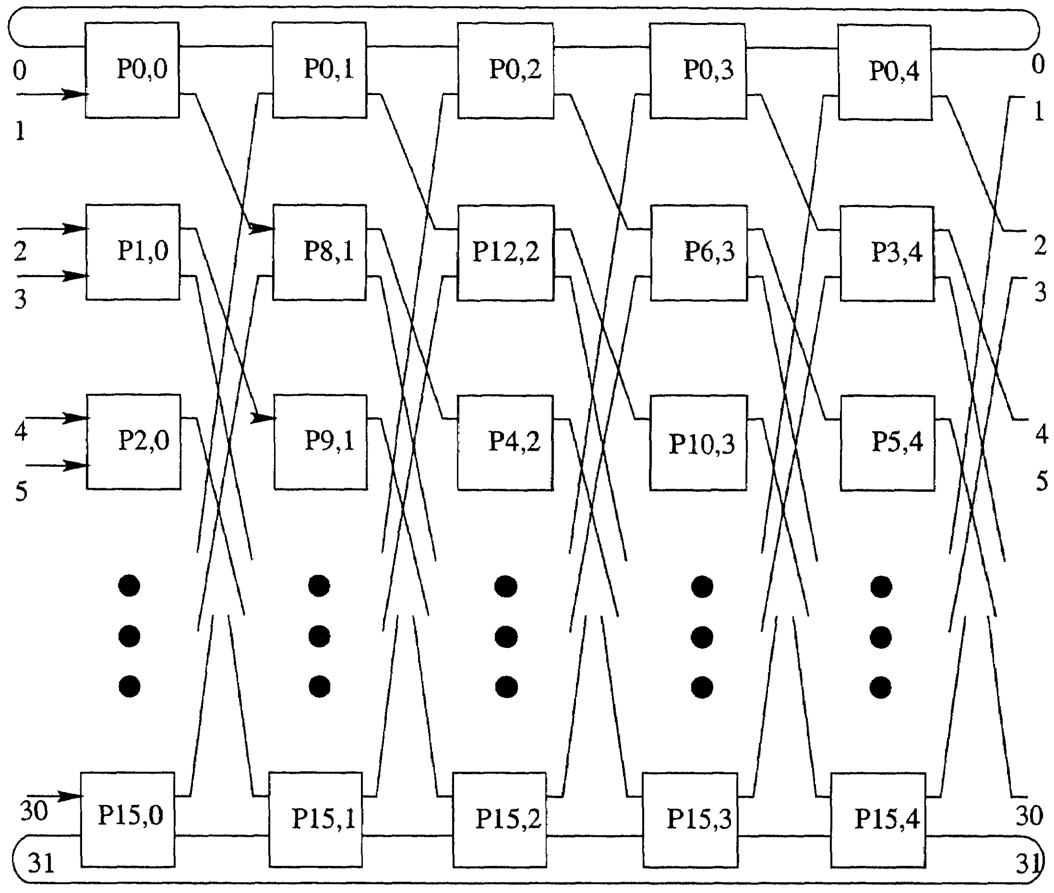


Figure 2-1 The 80-processor EM-X distributed-memory multiprocessor

2.1.1 The EM-X Multithreaded Multiprocessor

The EM-X multiprocessor is a large-scale multithreaded distributed-memory multiprocessor consisting of 80 processors, called EMC-Y. The machine was built at the Electrotechnical Laboratory and has been operational since December 1995 [11,12]. The main objective of building the machine was to investigate the performance of fine-grain instruction-level multithreading. Two types of computational principles have been employed in designing the multiprocessor. The first level uses the data-flow principles of execution to realize dynamic function spawning or runtime thread invocation. Any processor can dynamically generate function calls (or threads) on any other processor(s) including itself. This dynamic function spawning enables instruction-level multithreading and efficient fine-grain communication. The second level employs the conventional RISC-style execution to exploit program locality. The machine has a two-stage pipeline consisting of fetch and execute. Instructions that do not require remote reads/writes and dynamic function spawning are executed by this two-stage pipelining.

The current EM-X prototype has 80 EMC-Y processors connected through a circular Omega network. Figure 2-1 shows an overview of the prototype EM-X multiprocessor. The network is a variation of the Omega network which repeats a pair of shuffle and exchange operations. While Omega network is a multistage network, the EM-X is not. The main difference between Omega network and the EM-X network is that each processor is attached to a switch box. The EM-X network provides the maximum diameter of O(log NPE) for low latency and high throughput, where NPE is the number of processors. All the communication activities are one-sided, i.e., only one processor is involved in communication. The destination processor has no knowledge of any communication initiated by the source processor. Therefore, there is no notion of processor locking or sending and receiving that are used in the message-passing paradigm.

All the communication in the EM-X are done with 2-word fixed-sized packets. Communication can be classified into three categories: remote read, remote write, and thread invocation. A remote read packet consists of two 32-bit words. The first word contains the destination address, from where data will be read at the destination processor. The second word is the return address, also called continuation, which will be explained in detail shortly. A remote write packet also consists of two words. The first word is the destination address while the second is the data to be written on arrival at the destination processor. Thread invocation is done through packets which will be explained after we present some details of the processor architecture.

Figure 2-2 shows the organization of the EMC-Y processor. A processor element, EMC-Y, is a single chip pipelined RISC-style processor, 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 the register-based RISC execution principle with the packet-based dataflow execution principle for synchronization and message handling support. Each processor element consists of Switching Unit (SU), Input Buffer Unit (IBU), Matching Unit (MU), Execution Unit (EXU), Output Buffer Unit (OBU) and Memory Control Unit (MCU).

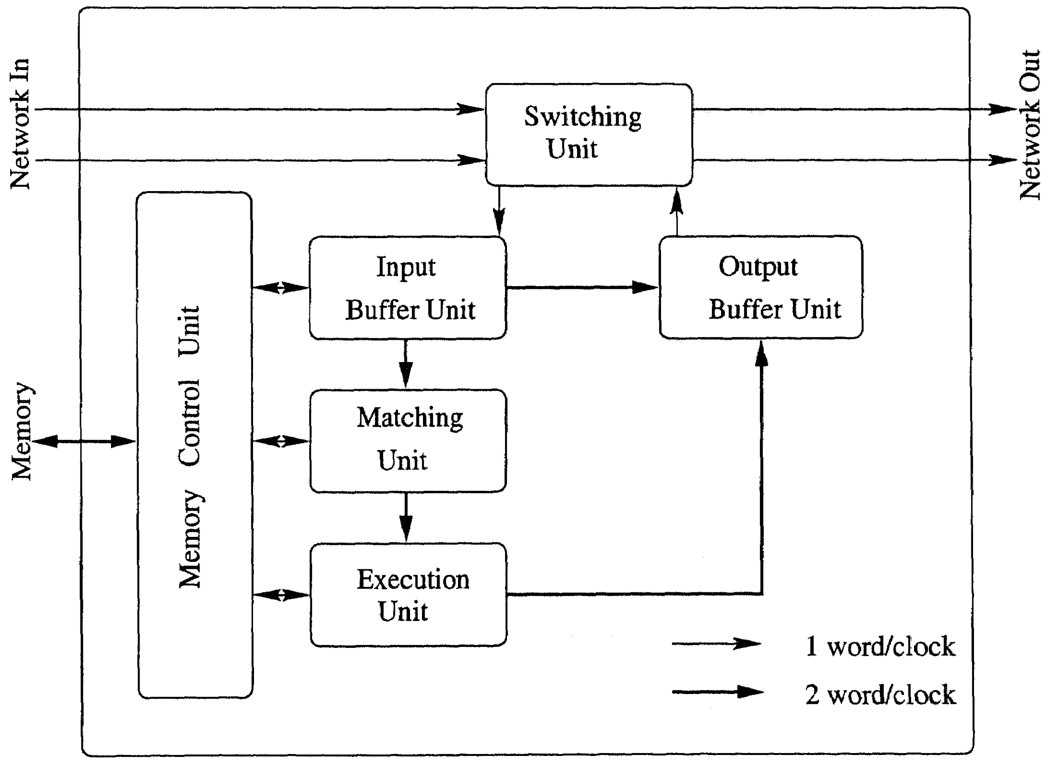


Figure 2-2 Architecture of the EMC-Y processor

The Switch Unit(SU) 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 containing an address part and a word containing a 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(IBU) 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 the on-memory buffer, and if the buffer is not full, they are automatically restored back to the 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 the Execution Unit.

The Matching Unit (MU) fetches the first packet in the FIFO of IBU. If the packet requires matching, a series 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 output buffer unit which separates the execution unit from the network. The Memory Control Unit(MCU) controls the access to the local memory off the EMC-Y chip.

The Execution Unit(EXU) 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 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 a data element and remote read request for a block of data.

The Output Buffer Unit(OBU) receives packets generated by the EXU or IBU. The buffer can hold up to 8 packets which is same as IBU. 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-passing 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 the EM-X for fast remote read/write without consuming the main processor cycles.

2.1.2 The Principle of Multithreading

A thread is a set of instructions that are executed in sequence. A multithreaded execution model exploits parallelism across threads to improve the performance of multiprocessors [28,29]. 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 staying in the thread which is waiting for the requested data to arrive, thereby masking the detrimental effect of latency. Figure 2-3 illustrates the basic principle of multithreading.

Processor 0, P0, has three threads, T0, T1,and T2, ready to execute in the ready queue. P0 indicates that T0 is currently being executed which is indicated by a thick dark line. P0 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.

Tcs = context switch time, Trr= remote read time, RR = remote read.

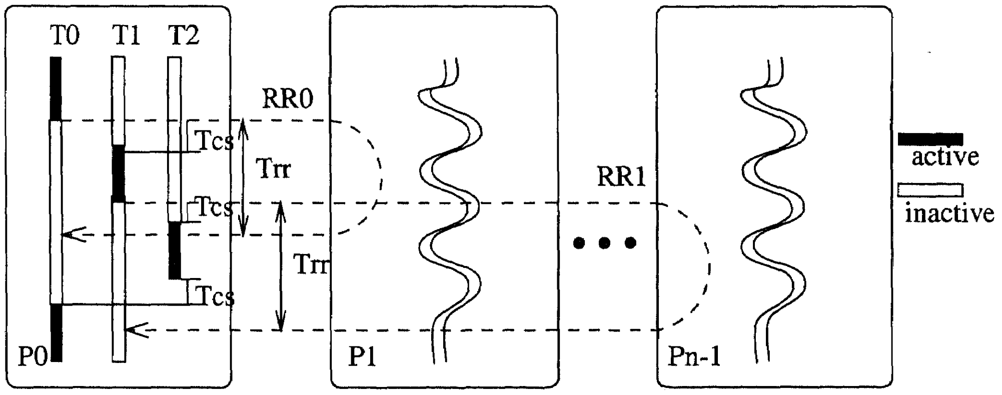


Figure 2-3 Multithreading on P processors.

Several parameters characterize the performance of multithreading. The parameters include (1) the number of threads per processor, (2) the number of remote reads per thread, (3) context switch mechanism, (4) remote memory latency, (5) remote memory servicing mechanism, and (6) the number of instructions in a thread. We will briefly explain the implications of these issues below.

The number of active threads indicates the amount of parallelism. To be more specific, the amount of parallelism can be classified into computation parallelism and communication parallelism. Computation parallelism refers to the `conventional' parallelism while communication parallelism refers to the way threads can communicate with other threads residing in other processors. Figure 2-3 shows that processor 0 has three active threads, indicating three thread computation parallelism, provided that they are independent of each other. Communication parallelism is not apparent from the figure as the way the three threads communicate is not specified in the figure. This will become clearer in the later sections. It is desirable to have a large number of threads since this will likely help tolerate latencies. However, the maximum number of threads that can be active (including the suspended status) at a point in time is bound by the amount of memory available to the program.

The number of remote reads per thread determines the frequency of thread switching and in turn run length. Each remote read corresponds to a thread switch. Hence, the number of switches is proportional to the number of remote reads. It is therefore desirable that the remote reads be distributed evenly over the life of a thread. This distribution leads to the issue of thread run length. Thread run length is determined by the number of uninterrupted instructions executed between two consecutive remote reads. The performance of multithreading is strongly affected by this parameter. If the run length is small, it will be difficult to tolerate the latency because there are not enough instructions to execute while the remote read is outstanding. Suppose that the dark area of T2 in the Figure 2-3 is very short, consisting of say 10 instructions. The machine will not be able to tolerate the remote memory latency since it is too short for RR0 to return(assuming a remote read takes tens to hundreds of cycles, if not thousands). The RR0 shown in Figure 2-3 will likely return after T2 runs to completion. In this event, the machine will wait until RR0 returns as there is no thread ready for execution.

Thread switch refers to how the control of a thread is transferred to another thread. Among the types of context switches are explicit switching and implicit switching. Implicit switching allows multiple threads to share registers while explicit switching does not. Implicit switching is literally implicit in the sense that there is essentially no visible switching from the register point of view. This implicit switching method, therefore, requires little or no switching overhead. However, the scheduling of registers and threads can be a challenging task. In the explicit switching, threads do not share registers. A single thread uses all the registers. Therefore, there is no issue as to how registers and threads are scheduled. However, the main problem of this explicit switching is the cost associated with register saving and restoring. For each thread switch, those registers currently being used by the thread need to be saved to preserve the status of the thread. When the thread that was suspended resumes upon the return of a remote read, the registers for that resuming thread will have to be restored. This register saving and restoring can be a bottleneck for efficient multithreading. Several approaches have been taken to solve the problems associated with the two switching mechanisms. One approach would be to have a few explicit sets of registers, where a thread is assigned to a register set. Another approach is to prefetch the corresponding registers in such a way that the thread can immediately resume when a thread switch occurs.

Communication latency is the main target of multithreading. It can vary depending on the technology used to build the machine and the interconnection network. A desirable combination would be that the network bandwidth be comparable with the processor clock speed. Large disparity between the machine clock speed and the network bandwidth can be problematic when multithreading. If the machine is fast but the network is slow, the pressure to tolerate the latency is high, in which case multithreading will probably not give noticeable results. On the other hand, if the machine is slow but the network is fast, the effect of multithreading may not be visible either since the communication latency is high in any case. There is a clear trade-off between the clock speed and the network bandwidth.

A mechanism to service remote memory operations is also an important factor determining the performance of multithreading. The simplest approach would be to have the main processor service remote read/write requests. In this case, the possibility of overlapping computation with communication will be very slim, giving little advantage of using multithreading. Some machines such as IBM SP-2 and Intel Paragon employ a communication co-processor to handle some communication related activities, i.e., copying data out of memory to the communication buffer or from the buffer to memory. The main advantage of using a communication co-processor is to take some burden off the main processor, thereby leaving the main processor for computation for most of the time. Multithreading can have a significant boost if equipped with such a remote servicing mechanism. The EM-X multithreaded processor employs a remote by-passing mechanism to perform direct remote read/write operations, or remote direct memory access (RDMA). The main processor is not aware of such remote read/write activities. This will be explained shortly.

The number of instructions in a thread is often referred to as thread granularity. While there is no clear agreement on such a classification, thread granularity can be classified into three categories: fine-grain, medium-grain, and coarse-grain. Fine-grain threading typically refers to a thread of a few to tens of instructions. It is used essentially for instruction-level multithreading. Medium-grain threading can be viewed as a loop-level or function-level threading, where a thread consists of hundreds of instructions. Coarse-grain threading may be viewed as more of a task-level threading, where each thread consists of thousands of instructions. However, coarse-grain threading should not be interpreted as operating system level multitasking, where different programs interleave to mask off page faults. How the threads are formed is another important question which needs be answered [30,31]. Threads can be automatically generated by compilers or explicitly specified by the programmers.

2.1.3 Architectural Support for 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 arrives. 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 of 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 with two 32-bit words is generated. The first 32-bit word contains the destination address, whereas the second 32-bit word contains the return address. The read packet will be 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. 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 complier 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, including template segments and operand segments. The compiled code of functions are stored in template segments. Invoking a function involves allocating an operand segment as an activation frame. The caller allocates the activation frame, deposits the argument value(s) into the frame, and sends its continuation as a packet to invoke the caller's thread. The first instruction of a thread operates on input tokens, which are loaded into two operand registers. The registers can hold values for one thread at a time. The current implementation allows no register sharing across threads, thus no implicit-switching support. The caller saves any live registers in 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) from a tree rather than a stack, reflect 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 us.

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 a 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 the packets were received. Thread scheduling is, therefore, explicit First-In-First-Out(FIFO).

2.2 A Distributed Memory Multiprocessor - SGI/Cray T3E

The SGI/Cray T3E is one of the latest developments in distributed-memory machines. Processing elements of T3E are DEC Alpha 21164 microprocessors. Each processor issues four instructions per clock with up to a 12-stage deep pipelining. The peak performance of the Alpha microprocessor reaches 600 MFLOPS.

The processors on the T3E are manufactured by Digital Equipment Corporation (DEC), and are known as EV-5 or Alpha. This reduced instruction set computing (RISC) microprocessor is cache-based, has pipeline functional units, issues multiple instructions per cycle, and supports IEEE standard 32-bit and 64-bit floating-point arithmetic. Figure 2-4 shows T3E PE block diagram.

SGI/Cray T3E processing elements (PEs) designed by Cray Research engineers and fabricated in low-cost CMOS include the DEC Alpha microprocessor(DEC 21164), performance-accelerating control logic, and local memory with amount ranging from 64MB to 2GB that it can address directly. Each PE runs at 450 MHz with 256 MB of local DRAM memory. The PEs are connected by a low latency, high-bandwidth interconnect network arranged in a bidirectional 3-D torus. Interprocessor communication rates are 480 M bytes per second. The network bisection bandwidth exceeds 122Gbytes per second. PEs access the content of each other's memory by passing messages using message passing libraries (or the data-parallel programming language HPF).

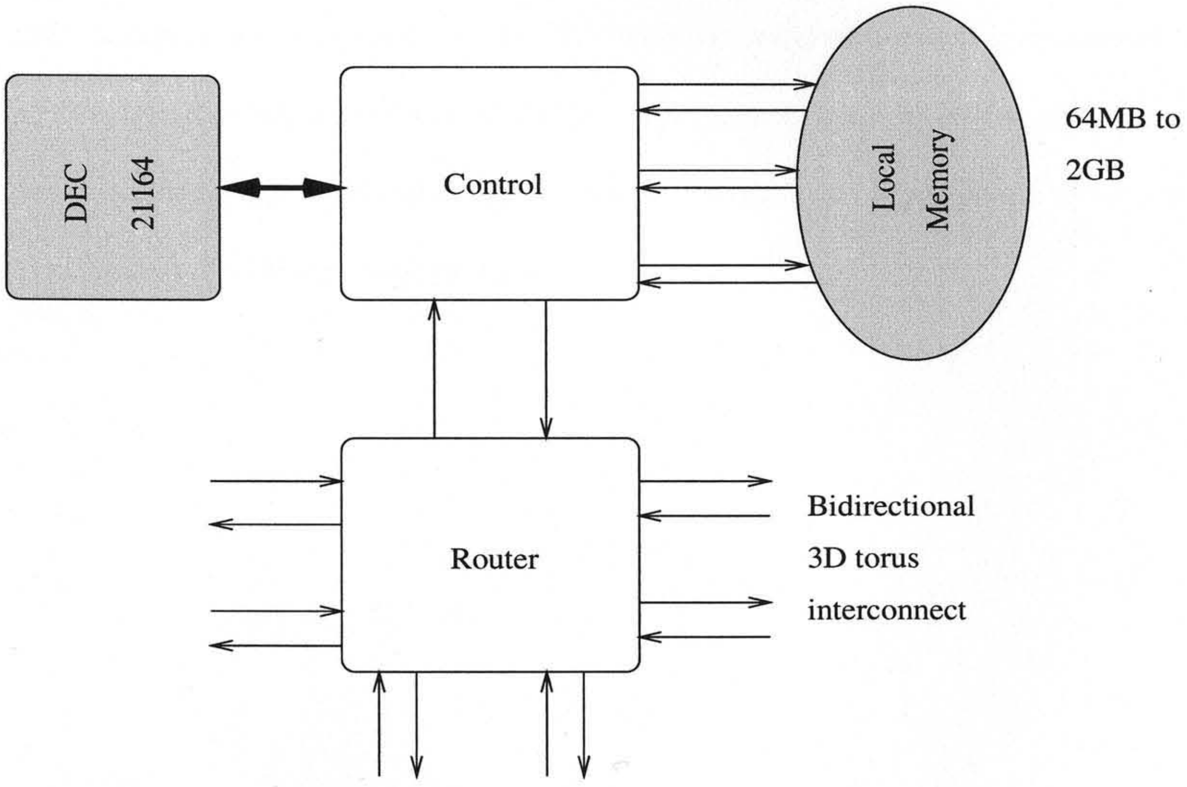


Figure 2-4 T3E PE block diagram

In addition to its scalable, high-bandwidth interconnection network, each processor module in the T3E has 512 off-chip memory-mapped registers, called *E-registers*. All remote access and synchronization operations are done between E-registers and memory.

These registers are quite effective in providing latency hiding, and barrier synchronization. Also, another important architectural feature for tolerating latency and enhancing scalability is *stream buffers* which are designed to detect and prefetch small-stride reference streams.

The latency of remote memory access can be hidden by using E-registers as a vector memory port, allowing a string of memory requests to flow to memory in a pipelined fashion, with results coming back while the PE works on other data. A range of synchronization mechanisms accommodates both Single Instruction Multiple Data (SIMD) and Multiple Instruction Multiple Data (MIMD) programming styles.

I/O channels are integrated into the 3-D torus network and increase in number with system size. As PEs are added to a SGI/Cray T3E system, interprocessor and I/O bandwidth scale accordingly. This means that scalable applications run as efficiently on larger configurations as they do on smaller ones.

CHAPTER 3

SISAL FUNCTIONAL LANGUAGE AND OPTIMIZING SISAL COMPILER

3.1 Sisal Functional Language

Sisal (Streams and Iteration in a Single Assignment Language) is a functional language for parallel numerical computation[26]. Functional languages promote the construction of correct parallel programs by isolating the programmer from the complexities of parallel processing[27]. Hence, this functional language is designed to simplify the process of writing parallel scientific programs by providing the compiler and runtime systems.

The Sisal language makes a distinction between the two basic kinds of loops, those in which the computations in the loop body are independent, and those in which they are dependent. The former contains the potential for parallel execution while the latter does not. The Sisal for-loop is used for repetitive computations that are independent, and is designed to expose the loop parallelism, so the language translation and support software can determine whether and how to exploit it. This allows programmers to concentrate on the algorithms used and the problems being solved.

A functional program consists of a set of mathematical expressions that map input parameters to output results. Based on the principles of mathematics that give correct Sisal programs with their definite meanings, Sisal exposes implicit parallelism through dataflow graph and guarantees determinate results regardless of the target architecture. This means that a Sisal program with the same input will produce the same output results, regardless of what computer it executes on, or how many processors (how much parallelism) are being used. Therefore, Sisal programs are said to be referentially transparent because they always produce the same result when the same parameters were given.

The Sisal language is a very good language for rapid prototyping. Sisal programs tend to be shorter, and easier to read and maintain. Sisal programmers tend to be able to write code faster, since they must describe only what the computation is, not the detail in which it must proceed.

Programming in the Sisal language with mathematical foundations allows compilers and runtime support systems to correctly analyze the program code and make good decisions about what is safe to execute in parallel. Many concerns which were required to be taken care of for parallel processing are realized automatically and are taken care of by the compiler and runtime system. Thus, the programmer does not and cannot manage these operations.

One or more machine independent dataflow graphs will be extracted from a Sisal program. Those graphs contain all the information regarding data dependencies among the operators which compose the program. These data dependencies form the only restraints associate with parallel execution. The Optimizing Sisal Compiler which will be described in detail later can make decisions about how to partition a Sisal program for parallel execution based on these graphs, which include the whole knowledge of the program's potential parallelism. Figure 3-1 shows the Sisal program for a dot-product problem. Figure 3-2 shows the graph for a dot-product problem:

```
define Main
type OneD = array[real];
function Main(N:integer; A, B: OneD returns real)
for I in 1,N
returns value of sum A[I]*B[I]
end for
end function
```

Figure 3-1 Sisal program for Dot-Product problem

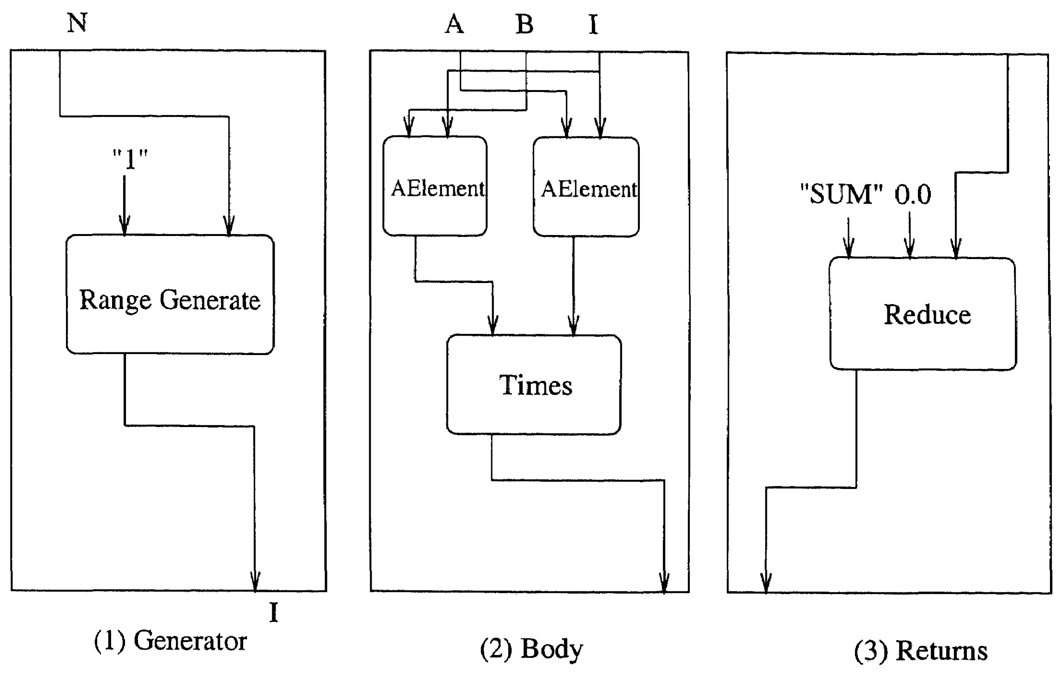


Figure 3-2 Directed Acylic Graph of Dot-Product

3.2 Overview of Optimizing Sisal Compiler(OSC)

If the Sisal compiler accepts a program, it has extracted complete information about what the program does and how its parts interact. In other languages such as Fortran or C, compilers typically just know how to do the parts, but do not know how they interact. The Sisal compiler is able to use its complete information to automatically determine how to execute the program, without the possibility of order-dependent or time-dependent errors. This enables the parallelism in the program to be exploited with certainty of correctness, determinacy, and the automatic management of parallelism. The automatic management removes the burden from the programmer.

The script below is a sample of shell script to get C code from a sisal program. Several intermediate files will be generated before the C code is generated in the same working directory. All of the intermediate files will be deleted by default except the final C code which will be fed into the runtime system to create the executable.

- osc -IF1 $1.sis
- if1ld -o $1.mono -FUR $1.if1
- if1opt $1.mono $1.opt -1 -l -e -U2
- if2mem $1.opt $1.mem
- if2up $1.mem $1.up -I
- if2part s.costs $1.up $1.part - - -SR -L0 -P1
- if2gen $1.part $1.c -U -G -O -Y3 -B -a

The whole process of the OSC(see Figure 3-3) will get the Sisal program. It will enter the first stage(Sisal Parser) with libraries to generate many IF1 graphs with the suffix.if1 in its filename, which will be the input file of the next stage. The second stage(IF1LD) will generate a monolithic IF1 graph with suffix .mono. The output file of any certain stage will always be the input of the next stage. The third stage(IF1OPT) will create IF2 graph with suffix .opt. The fourth stage(IF2MEM) will create a file with suffix .mem. The fifth stage (IF2UP) will get a file with suffix .up with memory optimization. The sixth stage (IF2PART) will do the parallelization and get a file with suffix .part.

The last two stages are IF2GEN and Runtime system which are machine dependent so that different C code generated from the IF2GEN stage will match its corresponding RUNTIME system. Wewill briefly describe some important stages of OSC.

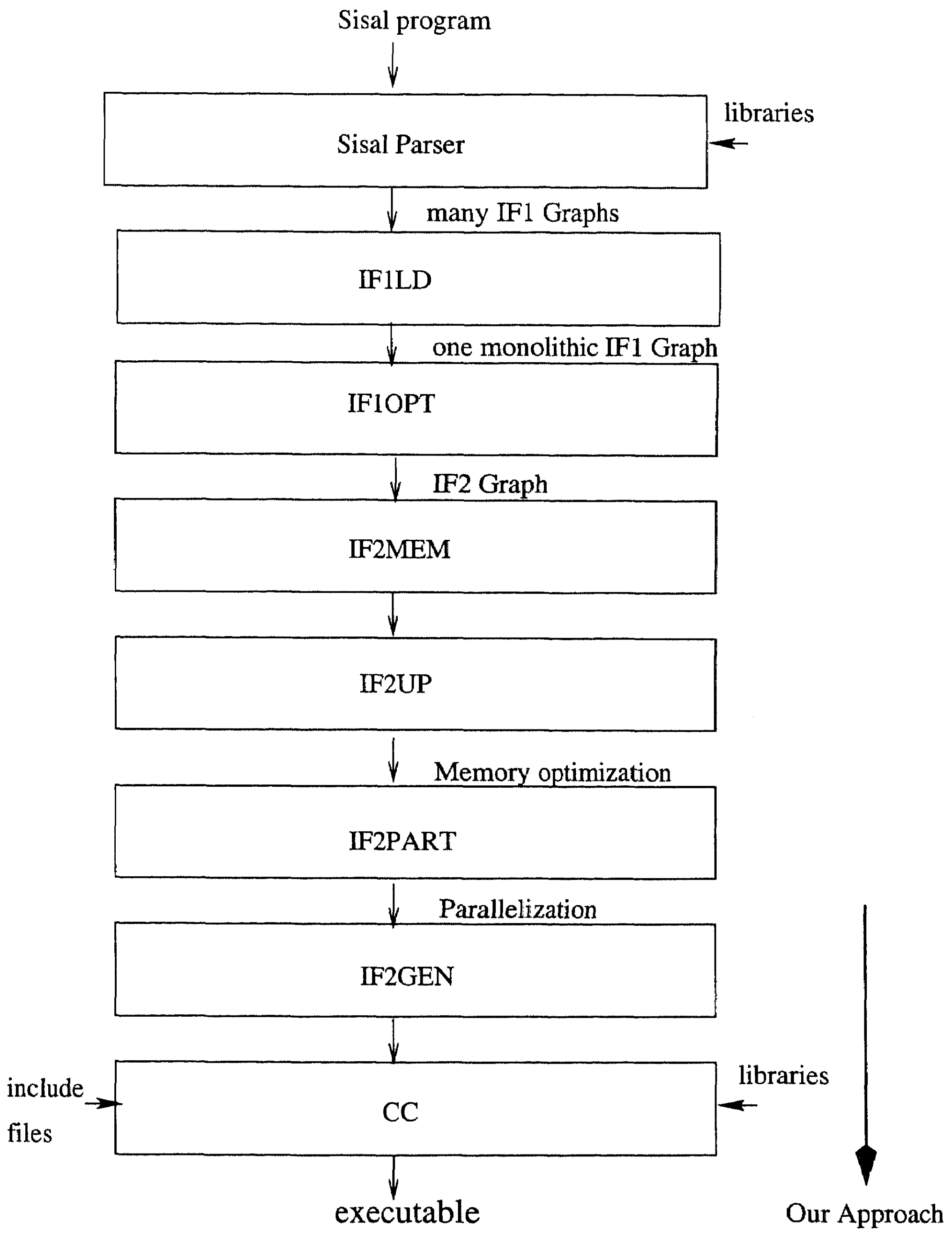


Figure 3-3 Overview of Sisal phases

3.2.1 Scalar Optimizations: IF1OPT

Applied optimizations includes:

- Function Inlining

OSC will attempt to inline as many non-recursive functions as possible.

- Record and Array Fission

Record and array fission are attempts to eliminate as many unnecessary records and arrays as possible to enforce more efficient programs.

- Loop Invariant Removal

Loop invariant removal eliminates unused computations in the loop body so that execution time can be reduced.

- Common Subexpression Elimination

Common subexpression elimination fuses identical computations within and between expressions, thus reducing the number of executed operations.

- Inverse Common Subexpression Elimination

Inverse common subexpression elimination pushes identical operation sequences out of conditionals.

- Dependent and Independent Loop and Conditional Fusion

Fusion merges two or more compound computation into a single compound computation, thus reducing control overhead and exposing opportunities for further optimization.

- Loop Unswitching

Loop unswitching is an attempt to eliminate as many conditionals as possible from innermost loops so that the complexity of time critical computations decreases.

- Loop Unrolling

If product-form loops can be determined to iterate N or fewer times, or to be within cost thresholds, loop unrolling will be performed.

- Loop Distribution

The purpose of loop distribution is to expose additional opportunities for vectorization and concurrentization. It can improve the effectiveness of loop fusion.

- Constant Folding and Operator Strength Reduction

Example: V:=exp(N,10); => T1:=N*N; T2:=T1*T1*N; V:=T2*T2

- Array Stripping and Dependence Exposure

Array stripping is a form of copy elimination that identifies and eliminates unnecessary aggregate constructions.

Array dependence exposure eliminates spurious dependencies; thus opening opportunities for further optimization.

- Work Reduction

Work reduction eliminates unnecessary work by pushing branch specific expressions into referencing conditionals.

- Dead Code Removal

Dead code will always be eliminated to avoid unnecessary computations.

3.2.2 Copy Optimizations: IF2MEM and IF2UP

Copy optimizations focus on solving data coping and excessive storage management due to the single assignment principle

- Build-in-Place Analysis(IF2MEM)

  Build-in-place analysis introduces runtime code to preallocate array storage whenever possible.

- Update-in-Place Analysis(IF2UP)

  Update-in-place analysis reorders operations to allow as many write operations as possible to run in-place.

- Array Prebuilding

  OSC will attempt to identify rectangular array constructions having invariant bounds and prebuild their storage frameworks. This helps to eliminate allocation-reclamation cycles in iterative computations.

3.2.3 Parallelization: IF2PART

This is for shared-memory machines

- Automatic Concurrentization

  In general, the compiler does a good job of automatically identifying product-form loops that warrant concurrent execution.

- Automatic Vectorization

1. Vectorizable Sequential Loops OSC will identify some sequential loops(such as Partial Sum) as candidates for vectorization.

2. Loop Distribution

On a parallel machine, OSC will attempt to extract parallelism from sequential loops and vectorization from non-vector parallel loops.

3. Loop Fusion and Vectorization

When requesting vectorization, OSC will not fuse vector and non-vector loops, or innermost independent vector loops. This complements loop distribution.

4. Associativity

As with concurrentization, OSC will consider reductions on real numbers safe for vectorization.

- Schedule parallel for loops

Parallel for loops can be exposed so that they can be scheduled.

3.3 Functional Parallelism: Pande's Approaches

Sisal language, was targeted to Intel Touchstone i860 systems[17,18,19], by mapping the functional parallelism in its intermediate IF1 representation. They modified the IF2PART to perform compile time scheduling of functional parallelism in IF1 nodes. This new compile time scheduling method was developed to investigate a trade-off between the schedule length and the required number of processors by using a new concept of threshold. Figure 3-4 shows the threshold scheduling algorithm used for allocating the task into a processor. A task graph could demonstrate the relationship among tasks where a node represents a task. The computation time of that task shown in the node as the weight of that node. The weight of the edge between node i and node j represents the communication cost required if node i and node j are not allocated to the same processor.

```
repeat
    find the edge(the smallest cost from all nonvisited imported edges) on that task;
  A : source node n of the actual dependence edge is not a task
        find a processor p(k) with schedule_time(n,p(k)) - est(n) <= THRESH
        => allocate the task into p(k), done.[2]
        if can't find such a processor then continue;
  B : source node n of the actual dependence edge is a task
        => the processor p(m) will be predecessor task m's processor
        if there is no collision when allocate the task n to p(m)
        then allocate the task n into processor p(m),done.[3]
        elseif schedule_time(n,p(m)) - est(n) <= THRESH
              then allocate the task n into p(m),done.[4]
              elseif merit(n,p(m)) >= merit(T,p(m)) T: a set of clashing tasks
                    then allocate the task n into p(m), tasks in T transfer, done.[5]
                    else continue
until (done) or (all imported edges are visited);
If (not done)
then find a processor p(k) to allocate the task n.[6]
else the task will allocate to a new processor.[1]
```

Figure 3-4 Threshold scheduling algorithm

We will briefly discuss the threshold scheduling algorithm in Figure 3-4 . A task has a predecessor task. Hence, the task is in B category. Case 3 shows a task that has no compete task in its predecessor task's processor; it is allocated to its predecessor task's processor. On the other hand, if there are some compete task(s) in the predecessor task's processor, this task will be either be allocated to the processor (if the difference between its scheduling time and earliest starting time are less than the threshold value(THRESH)) or be allocated to its predecessor task's processor by migrating all tasks in the task set T(all tasks that are compete with the newly allocated one) if the merit of this task is greater. This algorithm can control the number of active processors required simply by setting the THRESH. The algorithm will need fewer processors when THRESH is large; the algorithm will tend to use a new processor if the THRESH is quite small. Therefore, this algorithm enforced the information about which task was assigned to which processor and determined the required number of active processors by the compile time . Figure 3-5 shows an example which uses a certain THRESH(threshold) to apply into the threshold scheduling algorithm so that those 9 tasks are scheduled into 3 processors. Processor 1 has tasks 1,4,8,9, processor 2 has task 2 and processor 3 has tasks 3,5,6,7.

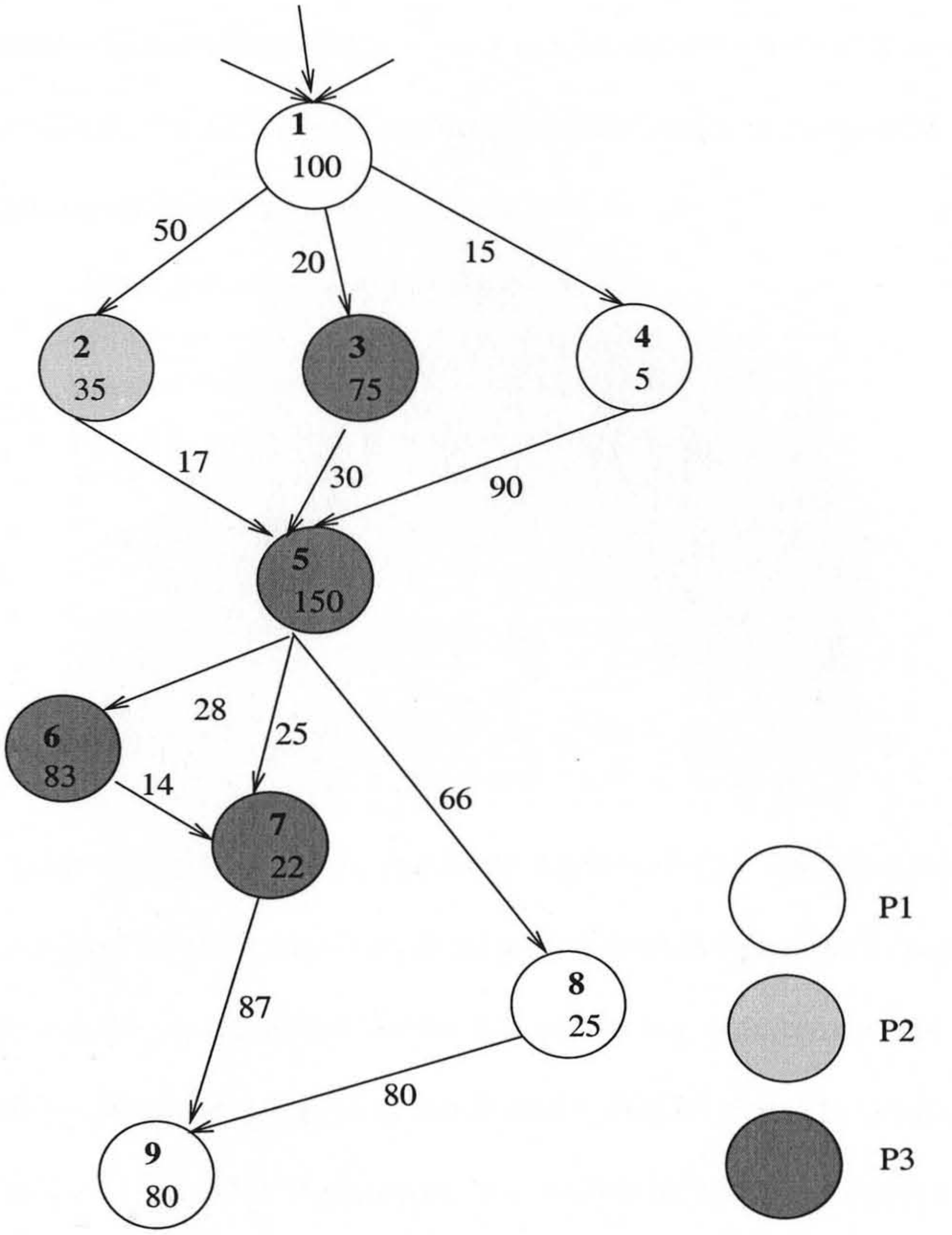


Figure 3-5 Example Task Graph

The time table of Figure 3-5(see Table 3-1) indicates the information related to the task graph shown in Figure 3-5, where est/ect denote the earliest starting/completion time, lst/lct denote the latest staring/completion time and ast/act denote the actual starting/completion time after all 9 tasks are scheduled to 3 processors. For example, node 2 (task 2) with weight 35 gets the earliest start time 100 and completion time 135 if it is assigned to the same processor where the node 1(task 1) is allocated. The communication time 50 can be saved since task 1 and task 2 are located in the same processor. On the other hand, if the communication time 50 is required then we will get the last start time 150 and completion time 185 of node 2(task 2), just like the ast and act since task 2 is assigned to processor 2 while task 1 is allocated to processor 1.

Table 3-1 Time table of Figure 3-5

	est	ect	lst	lct	ast	act
1	0	100	0	100	0	100
2	100	135	150	185	150	185
3	100	175	120	195	120	195
4	100	105	115	120	100	105
5	195	345	225	375	202	352
6	345	428	403	486	352	435
7	428	450	500	522	435	457
8	345	370	441	466	418	463
9	450	530	609	689	544	624

Efficient code was generated for the Intel Gamma, Delta, and Paragona family by generating the programs with a case statement which used the processor number to identify which code was to be executed by the corresponding processor. Each task(simple node in IF1 representation) is assigned to a processor. Before a certain task is executed in the processor to which that task is allocated, an embedded blocking receive call will be generated if the required data for the task is not locally available. In this way, the processor running that task gets the required data from the predecessor's processor to continue. On the other hand, after the task is completed, the data will be sent out to its destination processor which is the successor's processor if the data is not locally available there. This model will be simple if the Directed Acyclic Graph(DAG) was discovered in the IF2PART and all the information written into the result of the partitioning. Therefore, this part is modified in the IF2GEN(code generator).

The RUNTIME system was modified to support message passing communication between different processors sending and receiving the data values. Two basic communication primitives "pick" and "put" are defined and will operate asynchronously so that the sending and receiving processor do not have to rendezvous which exposes more parallelism. These two communication primitives which are expanded to appropriate system calls on the Intel machine are nothing but macros defined in "sisal.h" - runtime header.

CHAPTER 4

COMMUNICATION PARADIGMS

When programming the distributed-memory multiprocessors, the communication paradigm is one of the factors that we need to consider to ensure portability or to improve the performance. In this dissertation, we focus on message passing which is used widely on distributed-memory parallel machines. The two main issues in this dissertation are programmability and performance. Portability is not an important issue now, but will be considered in the future.

4.1 Two-Sided Communication

Message Passing Interface(MPI)[49] is a widely used standard means of communications among distributed memory processes. As in a distributed memory environment, all processors are independent with its own local memory. No physical memory is shared by any pairs of two processors. Hence, if any processor processes need to share information, they need to do the communication by passing messages. MPI was used in many message passing programs since it establishes a practical, portable and flexible standard for message passing.

The original MPI Standard document was published in 1994. The first product, Version 1.1 of the MPI specification, was released in June of 1995. MPI is widely used and well-known. Send/receive communications used via MPI require matching the operation by the sender and the receiver. In order to issue the matching operations, an application needs to distribute the transfer parameters. MPI provides for either blocking or nonblocking communications, the latter often lead to improve performance.

There are four communication modes, standard,buffered,synchronous and ready for both blocking and nonblocking communications. In standard mode, MPI decides whether to buffer the message after considering performance reasons and the size of available buffer space. In standard mode, the sending process does not know whether or not a matching receive has been posted. Sending in a buffer mode will be forced to copy the information to a temporary buffer if no matching receive has been posted. A synchronous mode send will complete only after the receive is posted(matching receive has been posted). A send operation with ready mode is the same as standard or synchronous mode, but the sender provides additional information(namely, the situation of the matching receive)to the system to save some overhead.

It will be difficult to program since this kind of two-sided communication needs to make efforts on synchronization while one-sided communication does not. Moreover, programs written by MPI which need to synchronize frequently will affect overall performance by the overhead of communication and synchronization.

4.2 One-Sided Communication

Remote Memory Access(RMA) extends the communication mechanisms by allowing a process to specify all communication parameters, both for the sending side and the receiving side. This mode of communication facilitates the coding of some applications with dynamically changing data access patterns where the data distribution is fixed. In such a case, each process can compute what data it needs to access or update at the other process. However, processors may not know which data in their own memory need to be accessed or updated by remote processes, and may not even know the identity of these processes.

Regular send/receive communications require a matching operation by sender and receiver. In order to issue the matching operations, an application needs to distribute the transfer parameters. This may require all processes to participate in a time consuming global computation, or to periodically poll for potential communication requests to receive and act upon. The use of RMA communication mechanisms avoids the need for global computations or explicit polling.

Message-passing communication achieves two effects: communication of data from sender to receiver and synchronization of sender with receiver. The RMA design separates these two functions. Two communication calls are always provided: put(remote write) or get(remote read). Various synchronization calls are provided to support different synchronization styles. Correct ordering of memory access has to be ensured by the programmer using synchronization calls; for efficiency, the implementation can delay communication operations until the synchronization calls occur.

The design of the RMA functions allows implementors to take advantage, in many cases, of fast communication mechanisms provided by various platforms. This is the reason why it was added into MPI-2 recently. In this dissertation, by-passing one-sided remote read for ETL EM-X is one-sided communications. The one-sided communication ability in MPI-2 will be the key of the future work since it is portable and relatively easy when compared to MPI-1.

4.2.1 By-passing One-Sided Remote Read/Write for ETL EM-X

All the communication activities for EM-X are one-sided, i.e., only one processor is involved in communication. The destination processor has no knowledge of any communication initiated by the source processor. Therefore, there is no notion of processor locking or sending and receiving that are used in the message passing paradigm.

All the communication in the EM-X is done with 2-word fixed-size packets. Remote read and remote write are two of three categories classified from the communication of EM-X. A remote read packet consists of two 32-bit words. The first word contains the destination address, from where data will be read at the destination processor. The second word is the return address, also called continuation. A remote write packet also consists of two words. The first word is the destination address while the second is the data to be written when arrived at the destination processor. The above characteristics in communication for EM-X were discussed in section 2.1.

Message passing in EM-X is asynchronous and buffered: the send(write) operation blocked the current thread until the entire message is sent, but does not waits for the message to be received. The receive(read) operation blocks the current until the message is received. In EM-X, message passing can copy a block of memory from the current processor to another processor via mem_copyout function.

The remote memory access facility enables a different programming style in threaded-based programming. The illusion of global memory shared among processors was provided. To access memory in a different processor element, the function mem_write(float_write) and mem_read(float_read) are used explicitly.

4.2.2 SHMEM for Cray T3E

Cray's logically shared, distributed memory access (SHMEM) routines operate on remote and local memory. On Cray MPP systems, these routines are supported in CrayLibs.

SHMEM routines minimize the overhead associated with data passing requests, maximize bandwidth, and minimize data latency (the period between which a PE request data, and when it can use the data).

SMA(Shared Memory Access) Library offers optimal interprocessor communications performance for SGI/Cray T3D and T3E applications. The latencies are 1-2 micro seconds only.

SHMEM stands for shared memory protocol. SHMEM is a set of functions that pass data in a variety of ways, provide synchronization, and perform reductions. SHMEM functions are implemented in Cray MPP systems and Cray PVP systems but not on any other company's computers.

What SHMEMs lacks in portability, it makes up in performance. SHMEM is the fastest of the Cray MPP programming styles. In other words, it is not as portable as the MPI but has better performance. That is the main reason we use it in this dissertation for Cray T3E machine.

The reason for the speed is its closeness to the hardware approach. This demands more from the programmers in synchronization, which is provided automatically with some of the other programming styles.

SHMEM does a direct memory to memory copy, which is the fastest way to move data on a Cray T3E system. It will almost always enhance the performance when we replace some statements of another programming style with SHMEM functions. It is easy for it to give us a major speedup with minimal effort when we simply replace the major data transfers with shmem_put or shmem_get.

The shmem_put and shmem_get functions avoid the extra overhead sometimes associated with message passing routines by moving data directly between the user-specified memory locations on local and remote processors.

The shmem_put and shmem_get functions perform almost at the same rates. For large transfers, shmem_put offers a different kind of performance advantage by letting the calling PE perform other work while the data is in the network. Because shmem_put is asynchronous, it may allow statements that follow it to execute while the data is in the process of being copied to the memory of the receiving processor. The shmem_get function forces the calling processor to wait until the data is in the local memory, meaning that no overlapping of data transfer and early work can be done. The characteristics for shmem_put and shmem_get are similar to remote write and remote read for ETL EM-X, respectively. The advantage of using remote read or shmem_get is that it only needs to wait until the data is arriving but without synchronization. The remote write or shmem_put needs to do the synchronization to ensure that the receiving processor does not try to use the coming data before its arrival. Therefore, we use remote read in this dissertation.

CHAPTER 5

ANALYSIS OF DISTRIBUTION STRATEGIES

5.1 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. The gap between processor cycle and remote memory access time become wider, as the processor technology improves and rigorously exploits instruction level parallelism.

Distributed-memory multiprocessors are known to have a scalable architecture for building large parallel machines. However, the main problem with distributed-memory machines is the latency due to the mismatch of data and work distribution. Various approaches have been developed to reduce/hide/tolerate communication time, as well as to study communication behavior for general purpose parallel computing. Data partitioning in HPF is a typical method to reduce communication overhead. 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 adaptive mesh-based computational, change their computational behavior at runtime. The initial data distribution is often found invalid and inefficient after some computations.

The mismatch of data and workload distribution is a key reason for long latency in distributed-memory multiprocessors. We have done research about how mutithreading can help tolerate such long latency caused by different distribution strategies. This research report explicates the impact of data and workload distribution on performance. Specifically, we vary the distribution strategies in a way that some combinations will have a good match of data and workload while some combinations will have a bad match of data and workload. Fast Fourier Transform was selected as a target problem. Two types of distribution strategies were used in the experiments: blocked and cyclic. Blocked distribution assigns consecutive data to a processor while cyclic distribution assigns data with stripe NPE, where NPE is the number of processors. These two strategies are applied to both data and workload distribution, resulting in four different combinations: blocked data and workload(BB), blocked data and cyclic workload(BC), cyclic data and blocked workload(CB), and cyclic data and workload(CC). All these strategies had been implemented on the 80-processor EM-X multithreaded distributed-memory multiprocessor, which is the target machine in this dissertation.

5.2 Data and Workload Distribution Strategies

Data distribution described in this dissertation assumes that no processors have the same element. All processors have a different portion of data. Distribution of data and workload is a critical factor for achieving performance on distributed-memory machines. One can use any data distribution strategy to distribute data across processors. Similarly, one can also distribute workload across processors. Distribution is typically done along with the major data,i.e., the data that may affect the overall performance.

Two typical distribution strategies are used in our experiments, which are blocked and cyclic. Given the data of size N and NPE processors, blocked distribution divides the data into NPE blocks, each of which holds consecutive $M= N/NPE$ elements. Cyclic distribution divides the N elements into NPE blocks, but in a round-robin fashion.

When only data is considered for distribution across processors, the data distribution can be done with a reasonable effort. The performance resulting from such distribution can be manageable in a sense that there is one major performance parameter to consider. However, when both data and workload are distributed across processors, the impact can be complicated since it will give a multiplicative effect on performance. Figure 5-1 shows the four possibilities of data and workload distribution based on blocked and cyclic methods.

Workload distribution

		Blocked	Cyclic
Data distribution	Blocked	BD-BW(BB)	BD-CW(BC)
	Cyclic	CD-BW(CB)	CD-CW(CC)

Figure 5-1 Combinations of data and workload distribution strategies

The upper left corner, labeled BB, shows blocked data and blocked workload distribution while the lower right corner, labeled CC, shows cyclic data and cyclic workload distribution. We will explain the implications of these different methods on performance shortly. For our experiments we use FFT as it exhibits regular computation and communication patterns.

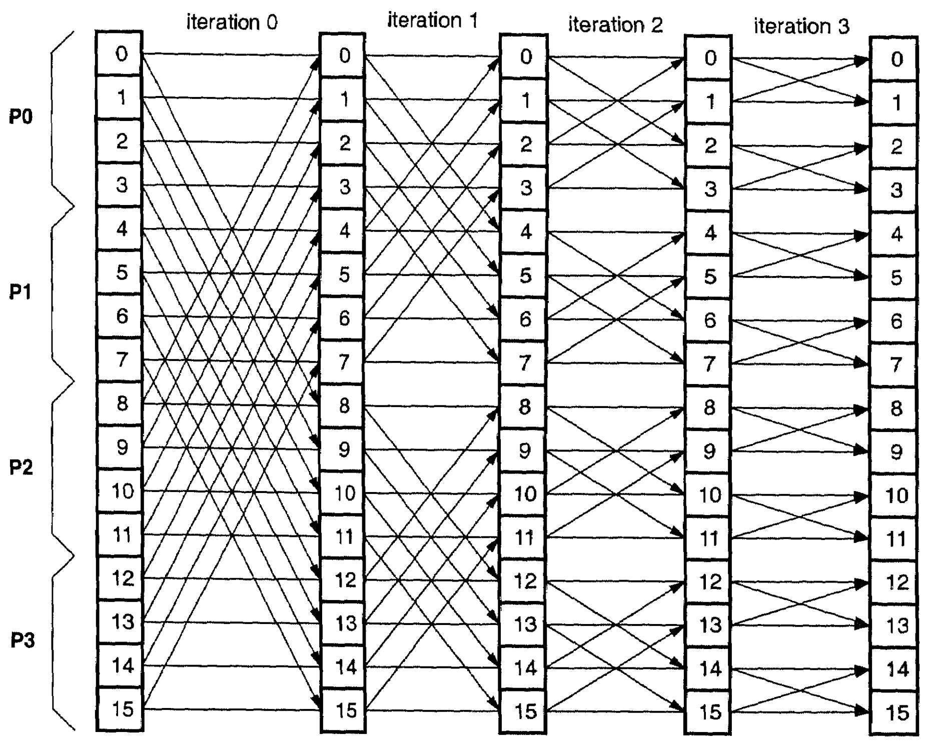


Figure 5-2 A 16-point FFT with blocked data and workload distribution on four processors

Figure 5-2 shows an FFT butterfly for the data size of 16 points with blocked data and workload distribution on four processors. An FFT with *N* points requires log *N* iterations. For the FFT with 16 points, it needs 4 iterations as shown in the figure. Using blocked data distribution methods, the 16 elements are divided into four groups, each of which is assigned to a processor. Processor 0, or P0, has elements 0 to 3, P1 has 4 to 7, and so on. Using blocked workload distribution, P0 computes four elements 0 to 3, P1 does four elements 4 to 7, etc. In the first iteration, or iteration 0, each processor obtains a copy of four points by finding its mate processor. P0 remote reads four points 8...11 from P2 while P1 does 12...15 from P3. P2 and P3 also obtain necessary data allocated to P0 and P1, respectively. The second iteration, or iteration 1, is essentially the same as iteration 0, except that the logical communication distance reduces to half the first iteration. So, P0 remote receives from P1 this time, the four elements which have been *newly* computed by P1 in iteration 0. Similarly, P1 obtains the newly computed four elements from P0. P2 and P3 perform similar operations. The last two iterations, iterations 2 and 3, do not require communication since the required data are locally available. In general, an FFT with blocked data distribution of *N* elements on *NPE* processors requires communication for the first log *NPE* iterations. The later log (*N/NPE*) iterations are local computations.

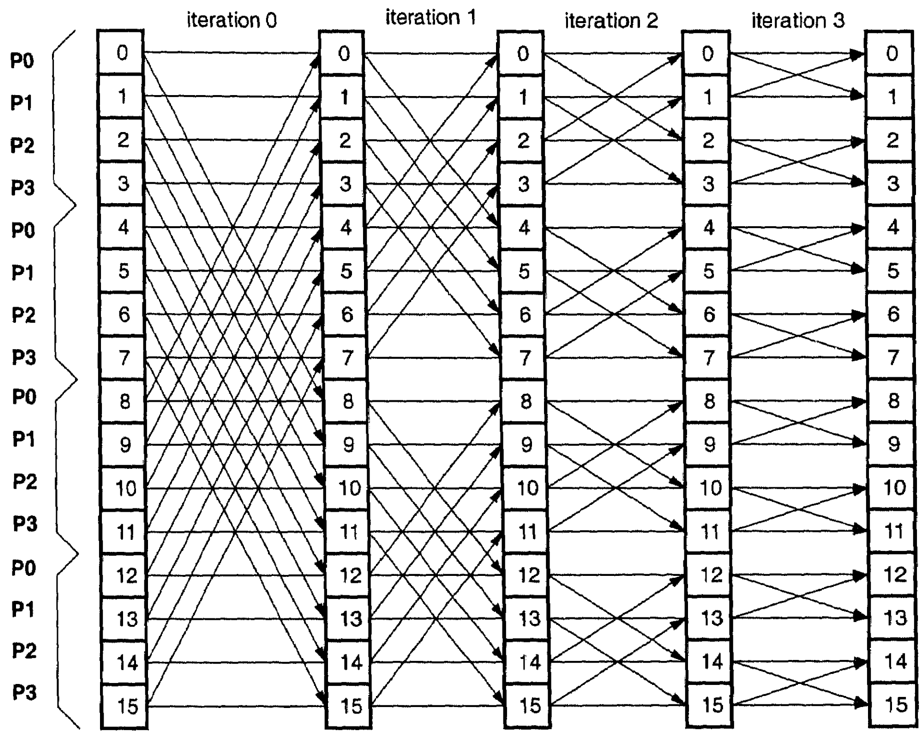


Figure 5-3 A 16-point FFT with cyclic data and workload distribution on four processors

Figure 5-3 shows an FFT butterfly for the data size of 16 points with cyclic data and workload distribution on four processors. Using cyclic data distribution methods, the 16 elements are also divided into four groups, each of which is assigned to a processor. P0 has elements 0,4,8,12; P1 has 1,5,9,13, and so on. Using cyclic workload distribution, P0 computes four elements 0,4,8,12; P1 does four elements 1,5,9,13, etc. Iteration 0 and 1 do not require communication since the required data are locally available. In iteration 2, each processor obtains a copy of four points by finding its mate processor. P0 remote reads four points 2,6,10,14 from P2 while P1 does 3,7,11,15 from P3. P2 and P3 also obtain necessary data allocated to P0 and P1, respectively. Iteration 3, is essentially the same as iteration 2, except that the logical communication distance reduces to half the iteration 2. So, P0 remote receives from P1 this time, the four elements which have been *newly* computed by P1 in iteration 2. Similarly, P1 obtains the newly computed four elements from P0. P2 and P3 perform similar operations. Hence, an FFT with cyclic data distribution of *N* elements on *NPE* processors requires communication for the last log *NPE* iterations, and the first log (*N/NPE*) iterations are local computations which are very similar to an FFT with blocked data distribution of *N* elements. It is therefore straightforward to perform computations based on blocked data and workload distribution or cyclic data and workload since they all have a perfect match between data and workload. Data is always there when needed for these two data and workload distribution: blocked data and workload distribution and cyclic data and workload.

Figure 5-4 shows what happens to the four different combinations. As we see in Figure 5-4, blocked data and workload distribution BB and cyclic data and workload distribution CC each show a perfect match - data is there when it is needed. However, the combination of blocked data and cyclic workload distribution BC and cyclic data and blocked workload distribution CB are problematic.

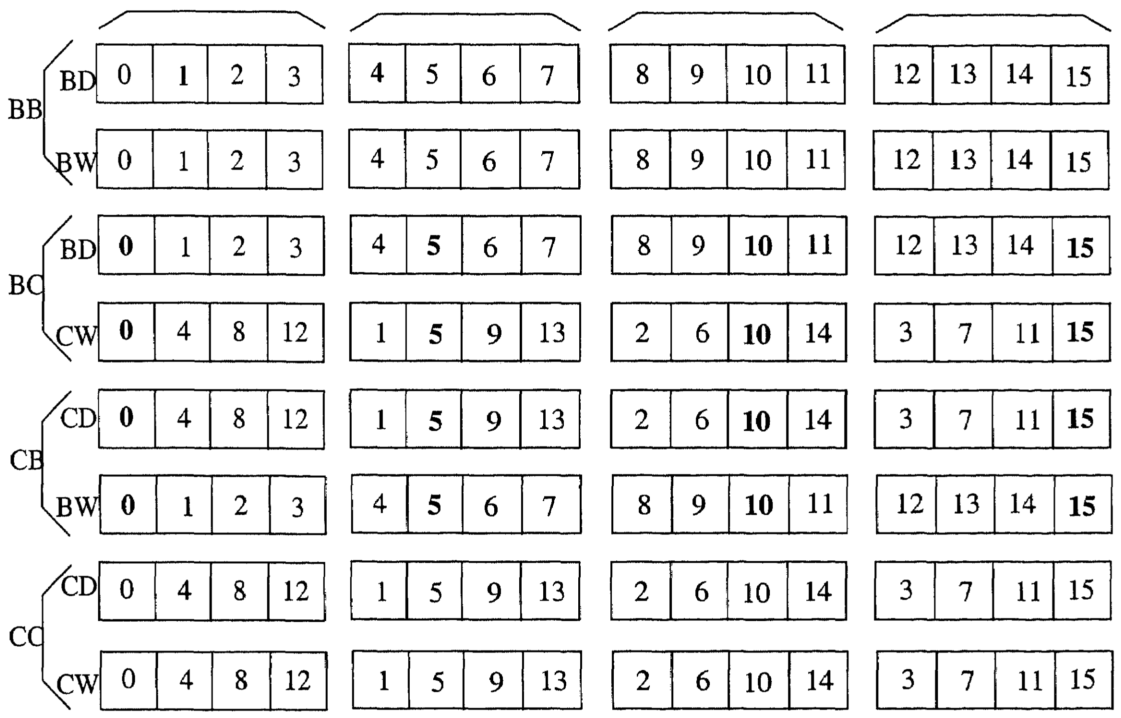


Figure 5-4 Four different combinations of data and workload distribution on four processors

5.3 Summary

Experimental results have shown several important observations. First, the best match, blocked data and workload distribution BB, has given the best communication result, as expected. This combination assures that data is there when it is needed for computation. While there are other minor remote reads for finding mate points, the maximum number of remote reads is limited to the data size of *N*log(NPE) for* the entire computation.

Second, the worst match of blocked data and cyclic workload BC (cyclic data and blocked workload CB) does incur three times the communication time of BB. The reason is BC (CB) needs up to *3N*log(NPE)* remote reads while BB needs only *N*log(NPE)* remote reads. It should be noted, however, that the communication time of BC (CB) is *lin-*

early proportional to the number of remote reads, not exponentially. The reason that the communication times did not exponentially increase is because of the machine's multithreading capability. We have found that the effects of multithreading continuously increased as the number of threads was increased to eight.

When the number of processors is increased to 64, the impact of different distribution strategies is less predictable because of the increased network load. However, as the problem size is proportionally increased together with the number of processors, the performance stabilizes because of the increased computations which can mask off the unpredictable communication time.

From the second observation, we realized that the communication time of BC(or CB) is a linear increase because of the machine's multithreading capability. It is needless to say that the communication time will be worse if the machine is without multithreading capability. Hence, we determined that we are not going to use BC(or CB) in this dissertation as these two combinations will cause a lot of communication overhead.

CHAPTER 6

THE APPROACHES

6.1 Functional Parallelism

Sisal language, was targeted to Intel Touchstone i860 systems[17,18,19], by mapping the functional parallelism in its intermediate IF1 representation. The IF2PART was modified to perform compile time scheduling of functional parallelism in IF1 nodes. This new compile time scheduling method was developed, by using a new concept of threshold to investigate a trade-off between the schedule length and the required number of processors. The compile time scheduling method determined the required number of active processors and the information about which task was assigned to which processor.

Efficient code was generated for the Intel Gamma, Delta, and Paragona family by generating programs with a case statement which used the processor number of the corresponding processor to identify which code was to be executed. Each task(simple node in IF1 representation) will be assigned to a processor. An embedded blocking receive call will be generated if the required data for the task is not locally available before this certain task was executed in the processor where that task was allocated. In this way the processor, to continue running that task can get the required data from the predecessor's processor. On the other hand, after the task is completed, the data will be sent out to its destination processor which is the successor's processor if the data is not locally available there. This model will be simple if the Directed Acyclic Graph(DAG) is discovered in IF2PART and all the information written into the result of the partitioning. Therefore, this part is modified in IF2GEN(code generator).

The RUNTIME system was modified to support message passing communication between different processors when sending and receiving the data values. Two basic communication primitives "pick" and "put" are defined and will operate asynchronously so that the sending and receiving processor do not have to rendezvous which exposes more parallelism. These two communication primitives which expand to appropriate system calls on the Intel machine are nothing but macros defined in "sisal.h" - runtime header.

6.2 Data Parallelism

Instead of functional parallelism, this dissertation uses data parallelism so that data will be distributed among processors. Several factors must be taken into consideration to develop a good model for porting the Sisal functional language to the distributed-memory multiprocessor. The basic idea is to create a simple model.

Scalability is the main reason we are using the distributed-memory multiprocessor, instead of shared-memory even though we all know programming on the shared-memory multiprocessor is a lot easier than programming on the distributed-memory multiprocessor.

Suppose we have a problem with a huge amount of data and there are no mega-beast shared-memory multiprocessors that can handle that data size. The solution would be to select a distributed-memory multiprocessor with each active processor processing a portion of the huge amount of data regardless of the fact that there are some issues that need to be taken care of for the distributed-memory multiprocessor.

6.2.1 Input/Output Data

Arrays are usually the largest data structures involved in the most time consuming computations in scientific high performance computing applications. Therefore, arrays will be the major data for all the problems we consider in our approach.

There are two different ways to get input major data, one is to generate the major data internally(create the major data by the source program); another way is to externally supplied major data which is to read all data from a data file. The former will ensure that the workload distribution matches the data distribution as long as the major data were created by a parallel loop with a certain distribution strategy(blocked or cyclic). If the data is externally supplied, the problem is that the size of the input data will be limited to the physical limitation of that processor where we are getting the data; it also contradicts the reason for using a distributed-memory multiprocessor. Therefore, we get all the major data from the source program so that the distribution strategy forms either BB or CC automatically.

Since the major data is generated internally, two input arguments required will be the number of processors(NPE) and the size of the major data(N). Although using a huge data file to make the major data externally supplied is not a good approach for parallel processing, we still provide this method for the sake of completeness. Hence, the input will always be a data file either containing the size of the major data or the whole major data. At the same time, the number of processors is always another argument for a parallel version.

Similarly, we will store all the output in a data file. But, all major data will be distributed among processors,i.e., all output arrays will be distributed to all active processors.

6.2.2 Distribution

In a distributed-memory machine, data needs to be distributed so there is no overlapping or copying of major data. All active processors operate and store a different portion of data. Distribution of data and workload is a critical factor for achieving performance on distributed-memory machines. One can use any data distribution strategy to distribute data across processors. Similarly, one can also distribute workload across processors. Distribution is typically done with respect to the major data,i.e., the data that may affect the overall performance.

Two typical distribution strategies, blocked and cyclic, are used in our experiments. Given the data of size N and NPE processors, blocked distribution divides the data into NPE blocks, each of which holds consecutive M=N/NPE elements. Cyclic distribution divided the N elements into NPE blocks but in a round-robin fashion. The resulting data are therefore stride in terms of index.

When data is the only one considered for distribution across processors, it can be done with a reasonable effort. The performance resulting from such distribution can be manageable in the sense that there is one major performance parameter to consider. However, when both data and workload are distributed across processors, the impact can be complicated since it will give a multiplicative effort on performance.

According to the previous research or study, the impact of different distribution strategies shown in Chapter 5 shows that the best match of blocked data and workload distribution(BB) has given the best communication result. The worst two matches are blocked data and cyclic workload(BC), and cyclic data and blocked workload(CB). They occur three times the communication times of BB. Therefore, we only provide BB and CC in this dissertation using BLOCKED or CYCLIC as the parameter of DIST in the makefile of IF2GEN to indicate BB or CC respectively. We use BB or CC because even with multi-threading capability, they are the only two good matches of four possible combinations when the blocked and cyclic distribution strategies are applied to both data and workload distribution.

Data and workload distribution make it easy for us to know where the data element is since either blocked or cyclic distribution has its own pattern. This factor makes it independent from the program flow.

6.2.3 Execution Model

The Single Program Multiple Data(SPMD) execution model was selected. SPMD is a generalization of Single Instruction Multiple Data(SIMD)-style parallelism, with much less strict synchronization requirements.

In this implementation, outmost loop parallelism is used for data parallelism. The Sisal language itself provides parallel loops so that it will make all developments easier. Suppose there are NPE processors assigned to a problem which only have N iterations in outmost parallel loop. The number of active processors(AP) will be the smallest number between NPE and N. Each active processor executes a slice of parallel loop which is about [N/AP] iterations. Each active processor does the exact same operations, but works on different parts of the major data. We can assign as many processors as the system can support, even some processors may be useless for a given problem which will occur when N is less than NPE so that AP will equal to N and makes NPE - N or NPE - AP processors idle. The information of N can not be determined at compile time most of the time. To avoid some processors idling, we could use the loop unrolling technique to make iterations of the outmost parallel loop large if when we write the program, we could predict that the iterations of the outmost parallel loop would be quite small when the parallel loop inside the outmost loops are quite large. We suppose all programs using our approach will not make any processor idle, since users could always check if their programs have parallel loops or not and could predict the size of the outmost parallel loop before they determined to use our approach. The IF2PART phrase, therefore, does not need to be modified since loop parallelism information is there already.

Three types of program are shown in Figure 6-1, 6-2 and 6-3. They are sequential version, BB parallel version, and CC parallel version, respectively.

```
tmp = 1;
while (tmp <= N) do
{

   /* parallel loop */

   tmp = tmp + 1;
}
```

Figure 6-1 Sequential version

```
epp = N/NPE;
remainder = N - (NPE*epp);
if (pid<remainder) { eppv=epp+1; tidx = (pid*eppv1); }
else { eppv= epp; tidx = (pid*eppv)+remainder; }

tmp3 =  tidx + 1;
while (tmp <= tidx + eppv) do
{

   /* parallel loop slice*/

   tmp = tmp + 1;
}
```

Figure 6-2 BB parallel version

```
tmp = 1;
while (tmp <= N) do
{

   /* parallel loop slice*/

   tmp = tmp + NPE;
}
```

Figure 6-3 CC parallel version

Sisal has parallel loops and sequential loops, two different kinds of loop representations. We want to parallel the outmost parallel loop. Therefore, one concern is the synchronization of the different combination of these two loop representations. A simple analysis shows the synchronization can be easily solved when we add a barrier step after any parallel loop.

6.2.4 Communication

We could use remote read or remote write to get the data from other processors if any communication is required. For the remote read case, we only need to follow the control flow graph by knowing which processor owns the data element required and where that element was stored. Therefore, according to the control flow graph and the distribution pattern we can check and determine whether a data element is locally available or not. Hence, we can issue a remote read by calculating which processor has that data element and go to that processor to fetch it from that place. There are two reasons that make the remote write case difficult and non-effective. The first reason is knowing when and where to issue a remote-write is not straightforward. The second reason is the remote-write operation needs to have a flag tag to confirm that the data element has been written to the receiver processor. This requires a synchronization step. This is a big drawback when the performance issue is considered. Hence, it will be easier for us to use a remote read operation for communication primitive in our approach instead of using a remote write operation.

6.3 Modifications in OSC for Our Approach

6.3.1 Pre-processing

Before generating the C code, the "inlining" optmization in OSC is activated so that all the functional calls are inlined at their respective call sites. Thus, after inlining,the data parallelism at each outmost loop is exposed in the whole program code, and a single "sisal.h" is created that contains all the inlined code.

In the IF-2 representation of the Sisal programs, multiple edges might carry the same values from one node to another. This results in sharing the same temporary(communication costs)among multiple edges. It does not matter, for a shared memory implementation, but in a distributed memory implementation, it might generate a lot of redundant communications. Therefore, before the code generation starts, the same algorithm as [17,18,19] performs the same pre-processing to select a unique edge to perform the real communication and ignore all other redundant communications.

6.3.2 Code Generation

The input file of the IF2GEN phrase(code generator) is the output file of the IF2PART phrase. In our approach, we directly get this file from the IF2PART phrase in the optimizing sisal compiler(OSC). From the information given by this file with suffix .part in IF2 representation of SISAL, we make necessary modifications of the C code generated.

We use data parallelism to parallelize the source sisal program. Therefore, workload distribution and data distribution both are the first two points that need to be considered. Both of them have two options, blocked and cyclic. A total of four possibilities can be created for our developments. They are blocked data and workload(BB), blocked data and cyclic workload(BC), cyclic data and blocked workload(CB), and cyclic data and workload(CC). According to our previous experiments and discussions in Chapter 5, we eliminate the worst two possibilities, BC and CB, in all four possibilities. Hence, only BB and CC can be used in our approach. The data distributed in our experiments will be the data mainly affecting the performance. They are the array data structure since arrays are usually the largest data structures involved in the most time consuming computations in scientific high performance computing applications. The case statement is not necessary for the task allocation since we are using the Single Program Multiple Data(SPMD) execution model so that we only need to take care of some special situations such as initialization or finalization. We try to make all the processors do the same program segments but with different data.

According to a certain data distribution, we need to find a way to compute where the data element is located. After all the data elements are distributed from a given strategy, we need to know which processor the data element was allocated to and how to access that data element. There are a division and a modulo with some additions to get all the information about that data element for both blocked and cyclic data distribution. When the data is not evenly distributed, we will need some more comparisons and computations for the blocked distribution. The cyclic distribution does not need to make any changes. The "on-the-fly" is the only way we were able to get any data element in our approach since it is the safe way for handling all situations. Furthermore, this "on-the-fly" pattern leads both blocked and cyclic data distribution to perform a similar performance. "On-the-fly" will use similar computation steps to get all the information on each data element and according to this information to determine whether that data element is locally available or not through a comparison. Through complete knowledge of each data element, we can get each data element either locally or from a remote access.

By inlining the Sisal function calls in the header file "sisal.h" for the portable purpose, a corresponding C _MAIN() function of the program will be created by code generator. Every processor has the same copy of the generated C code which uses a flag to indicate the processor is active or not. Each active processor runs a loop slice of the outmost loop with a specific workload distribution and a specific data distribution.

We did not use the function, "AElm", defined in "sisal.h". The purpose of that function is to retrieve an element from an array. The original design of the "AElm" function in "sisal.h" was to get the base register of that array and retrieve the element according to the index. I made some changes in it from the IF2GEN. First of all, I gave the base register of the array a name according to its characteristics. For example, I use float3 to indicate the array is the third array with the data type float. As a result, we can simply use float3[1] to retrieve the first element of that float3 array. In this way, we do not need to assign the base register of that array whenever "AElm" is used so that we can save some time for not repeating that assignment. Second of all, all main arrays are distributed in a certain data distribution and workload distribution(either blocked or cyclic). In addition, we need to do some computations to find out the dest, and idx according to the type of data distribution and the global index of the array we want to retrieve. Dest means the processor number where that element we want to retrieve will be allocated and idx presents the local index of that subarray and where it was located. After doing these computations, we can easily know whether that desired retrieval element is locally available or not. We also have the information about how to retrieve that data either locally or remotely. When the data is checked and found to be not locally available, we will need to do the remote access. Communication primitives are added in this kind of situation.

For the target machine ETL EM-X, we generate necessary global address space information. _MAIN will be the main function(thread) to be invoked by all the attending processors. We detect the parallel for loop and figure out which one is the outmost loop. Then, according to the specific workload distribution, we make some changes for the index manipulations which includes setting initial index and index computation of that loop or loop terminate condition. The indices for the start index and the end index will be computed for the blocked workload distribution whereas for the cyclic workload distribution these kinds of computations do not need to be done. In both blocked and cyclic workload distribution, we need to compute the size of the subarray which is a portion of the whole array after data are distributed. A flag is going to flip its initial value which was set by the initialize function before invoking _MAIN function when the processor is active and will run the partial of the parallel loop.

The output data array memory allocation was modified since the size differs from the original. Each active processor will only get a near even portion of the output data array instead of the total output data array that was generated per processor. Therefore, in the code generator, basic phases are unmodified. The main processor invokes the main function(thread) which invokes the initialize function from all the processors to set some initial variables (most of them are flags). The _MAIN function will be invoked by all attending processors as a main thread. After all attending processors finish the job, the work is complete with data array results distributed to all the active processors.

Two different kinds of data distributions are available. Either blocked data distribution or cyclic data distribution will only change the way of getting the major data. There are two different ways to get the input major data; one way is to create the major data by the source program whereas another way is the major data are externally supplied and usually stored as a data file. The former will ensure the workload distribution matches the data distribution as long as those major data were created by a parallel loop. If the data is externally supplied, the problem is that the size of the input data will be limited by the resources of the processor from which we are getting the data and it also contradicts the reason for wanting to use a parallel machine. Therefore, for major data we use the source program to generate BB or CC automatically according to the distribution strategy used.

The output has two different types, too. The first way is a processor print-out of all the output, which, again is limited to the resources of that processor which collects all the data. We do not use this way except when the output major data is a scalar type. But, the advantage of this way is that either blocked workload distribution or cyclic workload distribution makes no difference since the collecting steps are required to get the same output format. On the other hand, the output data are distributed to all the active processors and is the common way for parallel processing since they can handle a larger output size than the data size a single processor can handle. When we need to collect all portions of major data, it is preferable to use the blocked data distribution with blocked workload distribution for the ease of gathering the whole output data. The cyclic data distribution with cyclic workload distribution needs more effort to gather the whole output data. When the collection job is not required, different data and workload distribution will not make any difference at all.

6.3.3 Runtime System

The main job of the RUNTIME system is to properly support the communication primitives. Even though ETL EM-X is a multithreaded distributed-memory machine, we just use it as a single thread multiprocessor which means that we do not use the multithreaded capabilities. The memory of ETL EM-X is physically distributed, but appears to the user as a single shared memory (global address space). This approach is referred to as "virtual shared memory". The communication functions we used are remote memory operations provided by the thread library. The communication functions we are going to use for the Cray T3E are similar to get/put from the shared memory access library for interprocess communications.

Three communication primitives are defined as follows:

- pick(msgid, data_address, no_words, source_pid, flag_block_message)
- put(msgid, data_address, no_words, dest_pid, flag_block_message)
- etpick(msgid, data_address, no_words, source_pid, flag_block_message)

The put and pick primitives operate synchronously in message passing communication to guarantee that all processors can operate simultaneously after every parallel loop in this experiment. The etpick primitive operate as the remote memory access and similar to the remote_read/get function.

The different semaphore locks and cache locks from which memory is allocated to requesting processes are not necessary in the distribute memory implementation. Therefore, they are removed from the runtime system. Basically, no process management is required in distribute memory implementation. Most of the array and stream computes macros remain as they are to manage the arrays and streams.

CHAPTER 7

DISCUSSIONS

7.1 Problem Descriptions

Two simple problems: the search problem and the dot -product problem are too simple to discuss. Hence, we only describe two benchmark problems(Bitonic Sorting and Fast Fourier Transform) even though we use all four problems for this dissertation. To identify communication patterns of programs with different computational and communicational work loads, we chose one benchmark(Fast Fourier Transform) which is computation-bound and another benchmark(Bitonic Sorting) which is communication-bound.

7.1.1 Bitonic Sorting

Bitonic sorting introduced by Batcher [50] consists of two steps: *generate* and *merge*. The generate step transforms an input sequence of N unordered numbers into a bitonic sequence. For example, the first half(N/2 elements) was sorted in ascending order while another half was sorted in descending order. The merge step takes a bitonic sequence and outputs in sorted order. For example, sorted in an ascending order.

The generate steps need the outmost loop I = 1 to log N-1 with exactly J = I down to 1 iteration inside the Ith loop to generate a bitonic sequence. The merge step requires I = log N down to 1 iterations to sort the input bitonic sequence into a sorted order sequence. Researchers combine generate and merge steps into the outmost loop I = 1 to log N with exactly J = I down to 1 iteration inside the Ith loop algorithm. The total iterations required for bitonic sorting are logN(logN+1) /2). Thus, the total complexity of bitonic sorting is O(N*logN*logN) for there are O(N) comparisons required inside each iteration because each number and its mate number must form a certain order(ascending order or descending order). Figure 7-1 illustrates the generic bitonic sorting of 8 elements.

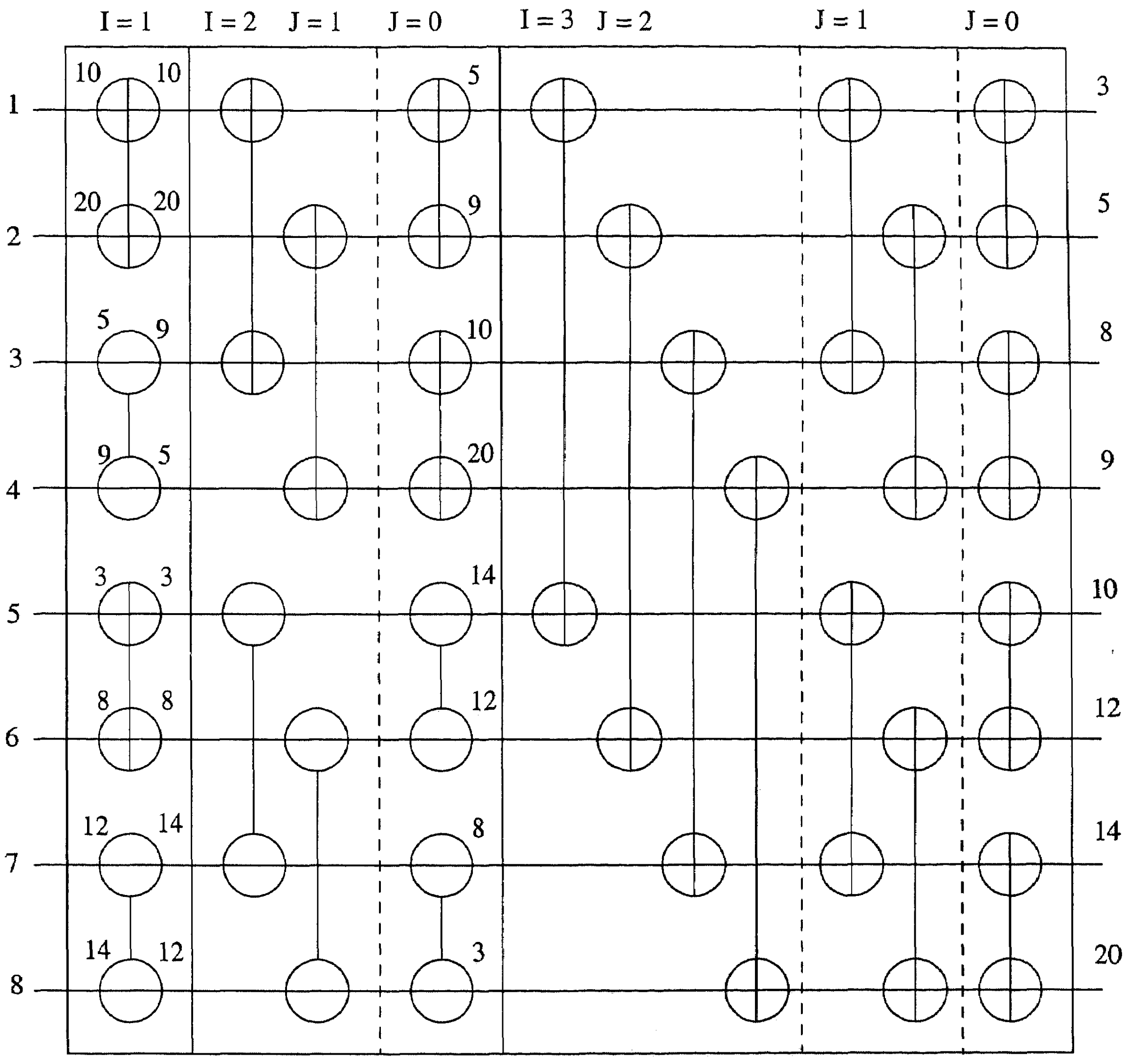
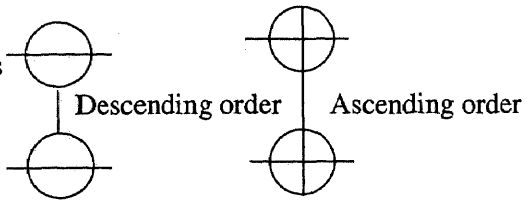


Figure 7-1 Bitonic Sorting of 8 elements



7.1.2 Fast Fourier Transform

The second problem used in this study is the Fast Fourier Transform (FFT) [51]. In the FFT problem, each element contains two parts, one is real and the other is imaginary. Because of the sin/cos function required for this FFT problem, FFT requires a lot of computation time. Figure 7-2 shows an FFT program.

```
for(m=0;m<logN;m++){
   for(z=0;z<N;z++) { real[z] = arr_r[z]; imag[z] = arr_i[z]; }
   x0 = 6.283/N;
   for(i=0;i<N;i++) {
   compute(i,m,logN,&j,&k);
   omega_(i,logN,m,&result);
   x = x0*result; cosx = (float) cos(x); sinx = (float) sin(x);
   arr_r[i] = real[j] + (real[k]*cosx  - imag[k]*sinx);
   arr_i[i] = imag[j] + (real[k]*sinx  + imag[k]*cosx);
   }
}
```

Figure 7-2 An FFT program

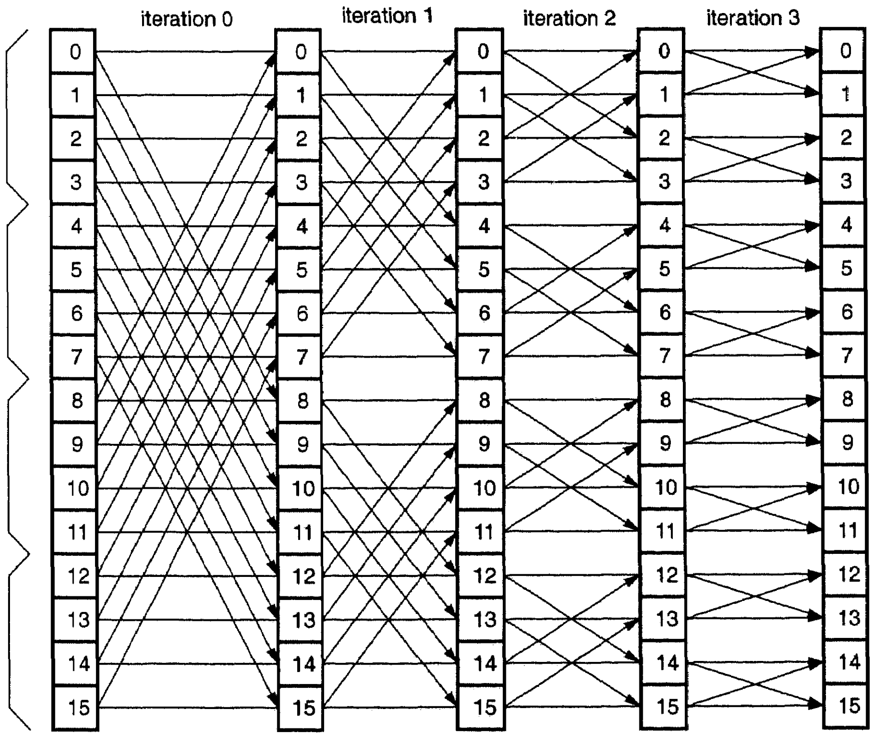


Figure 7-3 FFT with 16 elements

An FFT with *N* elements requires log N iterations. The butterfly shown in Figure 7-3 thus requires four iterations. In each iteration, each element must have its mate element's information so that the computation can go on.

7.2 Performance Discussion

7.2.1 Scalability

For problems of bitonic sorting and FFT, the code generated involves a function "DeallocArray"-memory free work, which does not work on the EM-X machine. This fact does not matter if the data size per processor is reasonably small, but it will cause problems when the data size per processor is getting large. Therefore, we slightly modified that source code without changing the algorithm but focused on solving this memory problem so that these two problems could run up to 8M. Since we also want to know the scalability in the pure automatic version, we only list the data size from 1K up to 128K for these two problems in this subsection. We will show the data size up to 8M (generated from the modified automatic version) for these two problems in the next subsection. For all tables in APPENDIX A-1 and A-2, the "*" mark means that the result comes from the modified automatic version.

From the statistics of the execution time for these four problems, the first three problems all show that the BB is superior to CC (see Table A-1, A-2, A-7, A-8, A-10 and A-11 in APPENDIX) while the FFT problem shows the opposite(see Table A-4 and A-5 in APPENDIX). In the next subsection, we will see that the BB version of the FFT problem is also superior to the CC version of the FFT problem when the data size is large(up to 8M). The ratios of execution time of CC(cyclic data distribution and workload distribution) to BB(blocked data distribution and workload distribution) are stable especially when the data size per processor is large(see Table A-3,A-6,A-9 and A-12 in APPENDIX). Only the FFT problem gets the ratio of execution time of CC to BB less than 1 with the other three problems greater than 1. Furthermore, the dot problem and search problem all show that the execution time of using 2 processors is equal to the execution time of the single processor in the BB version. The bitonic problem shows that the execution time of using 2 processors is more than the execution time of using any single processor in both the BB and CC versions. The FFT problem shows the best performance while the execution time of using 2 processors is almost half of the execution time of single processor in both the BB and CC versions.

The experimental results of the four problems - dot product, search, bitonic sorting and FFT on EM-X using BB or CC-all indicate that there is more speedup when we increase the number of processors while the data size is fixed or when we increase the data size while the number of processors remains the same(see Figure A-5, A-6, A-7 and A-8).

7.2.2 Analyze Results

In this subsection, data size is up to 8M for all four problems even though most of the data must be gained from a modified automatic version for both the FFT and bitonic sorting problems. Therefore, it will be fair to do some analyses for these four problems.

For the dot product and search problems, the overhead only occurs when the BB version is used since all the data are locally available except the final answer. Therefore, the overhead required for these two problems in BB will be the computation for the start index and the end index only.

The overhead for the other two problems(bitonic sorting and FFT) will mainly be the computations required for finding the dest processor and the local index of the remote processor for getting the data value. Figure 7-4 shows the percentage for the overhead time in these two problems using the number of processors 16 and 64. The overhead portion for bitonic sorting is between 35% to 45% while the FFT problem costs only between 5% to 10%. We also observe that the percentage for the overhead time on the FFT problem is very stable.

The communication time we computed will simply be the total time for all remote access operations. Figure 7-5 and 7-6 identify the percentage for the communication time individually of all four problems. Figure 7-5 is for 64 processors while Figure 7-6 for 16 processors. There is very little difference between the two sets of figures(7-5 and 7-6). They all present about 50% for both dot product and search problems for two versions(BB and CC), about 3% difference for the CC version of bitonic sorting and BB version of bitonic sorting and about 1% for the FFT problem on both the BB and CC versions. It is precisely our purpose to showing two sets of data on different numbers of processors.

We present Figure 7-7 to show the execution time of both the bitonic sorting problem and the FFT problem with data size up to 8M on 16 and 64 processors. This figure shows that the FFT problem will agree that the BB version is superior to the CC version for large data size when the problem with the small size shows the opposite.

From the experimental results of the dot product and search problem, we found that the speedup is almost half of the number of the processors for both the BB and CC versions. It is because the computation time is quite small that the communication time is almost the same as the computation time. Furthermore, these two problems both show similar results in the BB and CC versions(see Figure A-9,A-10,A-11,A-12). The distribution of execution time is totally insensitive to the number of processors.

The FFT problem, is different from the dot product and search problems. Unlike problems dot product and search, the FFT problem has a parallel loop in a sequential loop. There are a lot of computations required in the parallel loop of this problem so that communication time and other overheads will not make as many effect on the performance as the dot product problem or search problem do. Figures A-13 and A-14 illustrate the distribution of execution time for the FFT problem in the BB and the CC version respectively. These two figures show the similar distributions: the percentage of the computation time is up to 92%, the percentage of the time spent for the overhead is about 6% and the percentage of communication time is about 1%. Therefore, the speedup is up to 60 on the 64 processors for the FFT problem. The distribution of execution time of this problem is not sensitive to the number of processors, which increases about 0.5% for the computation time when the number of processors varies from 64 processors to 16 processors.

The bitonic sorting problem, on the other hand, is that the parallel loops inside two sequential loops. Not many computations are required inside this parallel loop so communication time and other synchronization overheads will make some effect on the performance the same as dot product problem did. The speedup reach is up to 17 on the 64 processors. The distributions of execution time for the bitonic problem in the BB and the CC version are shown in Figure A-15 and A-16 respectively. The distributions show that about 60% of the execution time is computation time for both the BB and CC versions, 38%(BB) and 35%(CC) of execution time is overhead time while the communication time for the BB version is 2% and the communication time for the CC version is 5%. These two figures also show that when the number of processors decreases from 64 processors to 16 processors, the distribution of execution time changes slightly. The percentage of computation time decreases about 1%, overhead time increases about 2% and communication time decreases about 1%. The other three problems do not show this kind of effect.

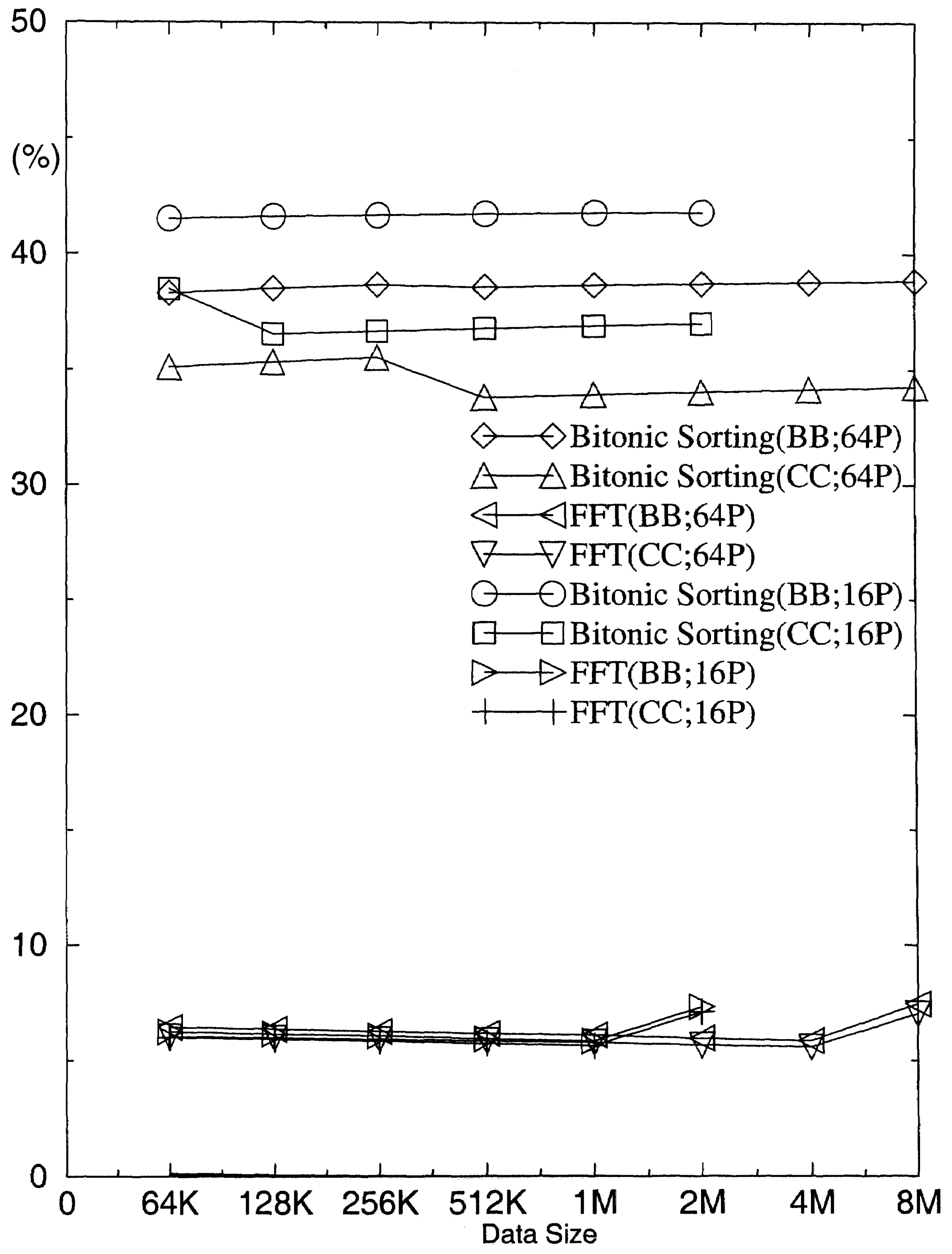


Figure 7-4 Percentage of Overhead Portion for Bitonic Sorting and FFT

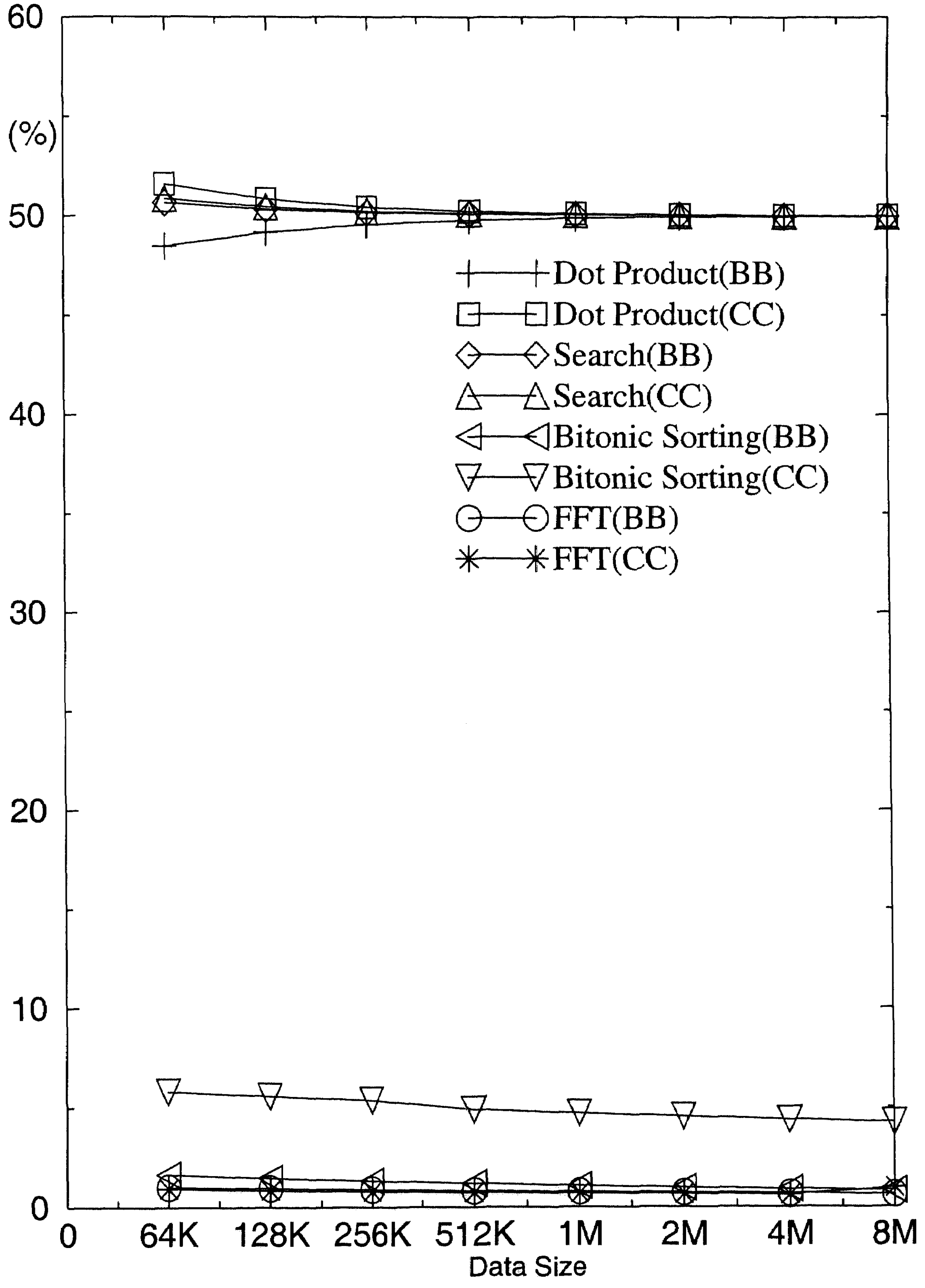


Figure 7-5 Percentage of Communication Portion for 64 Processors

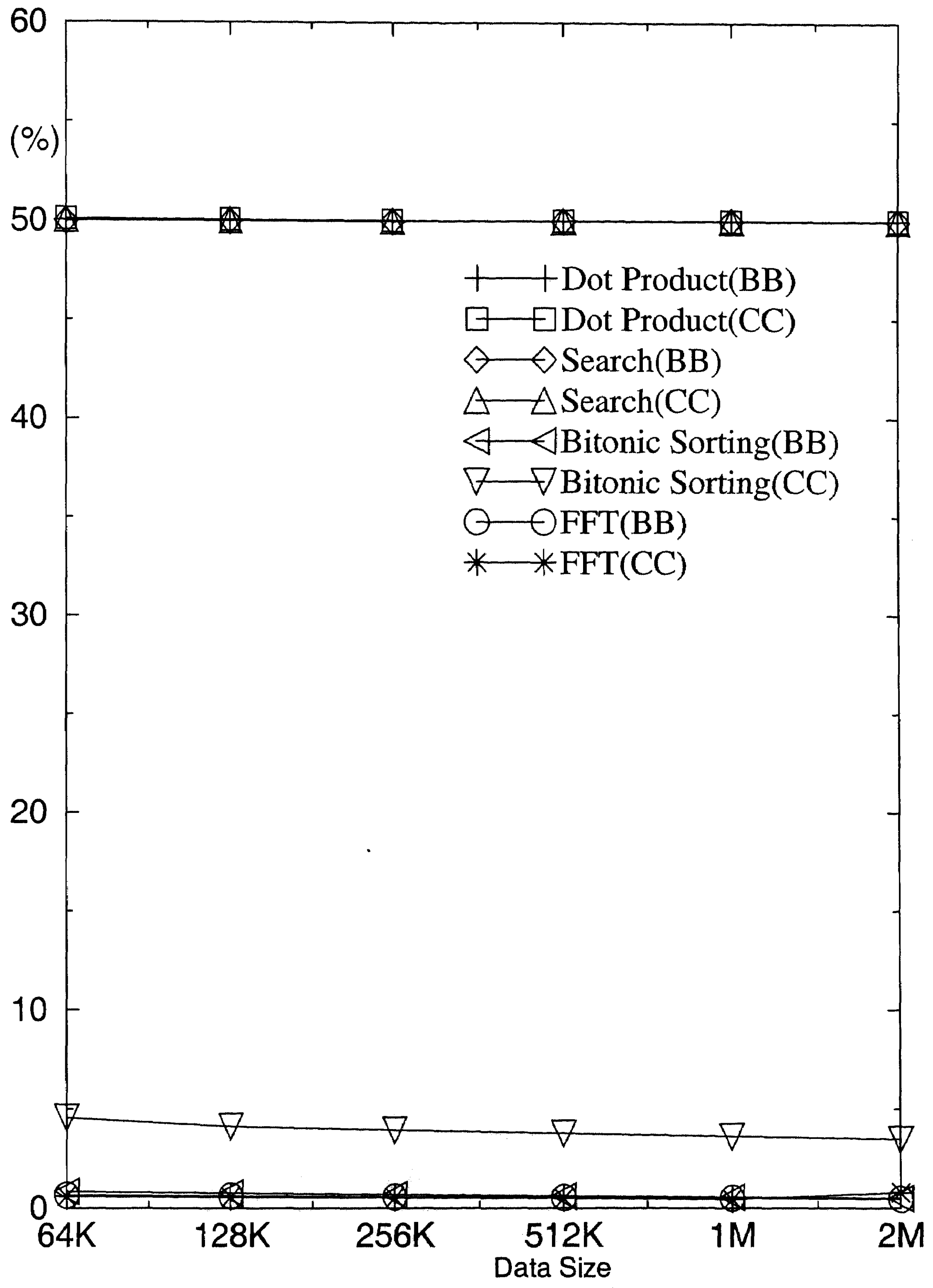


Figure 7-6 Percentage of Communication Portion for 16 Processors

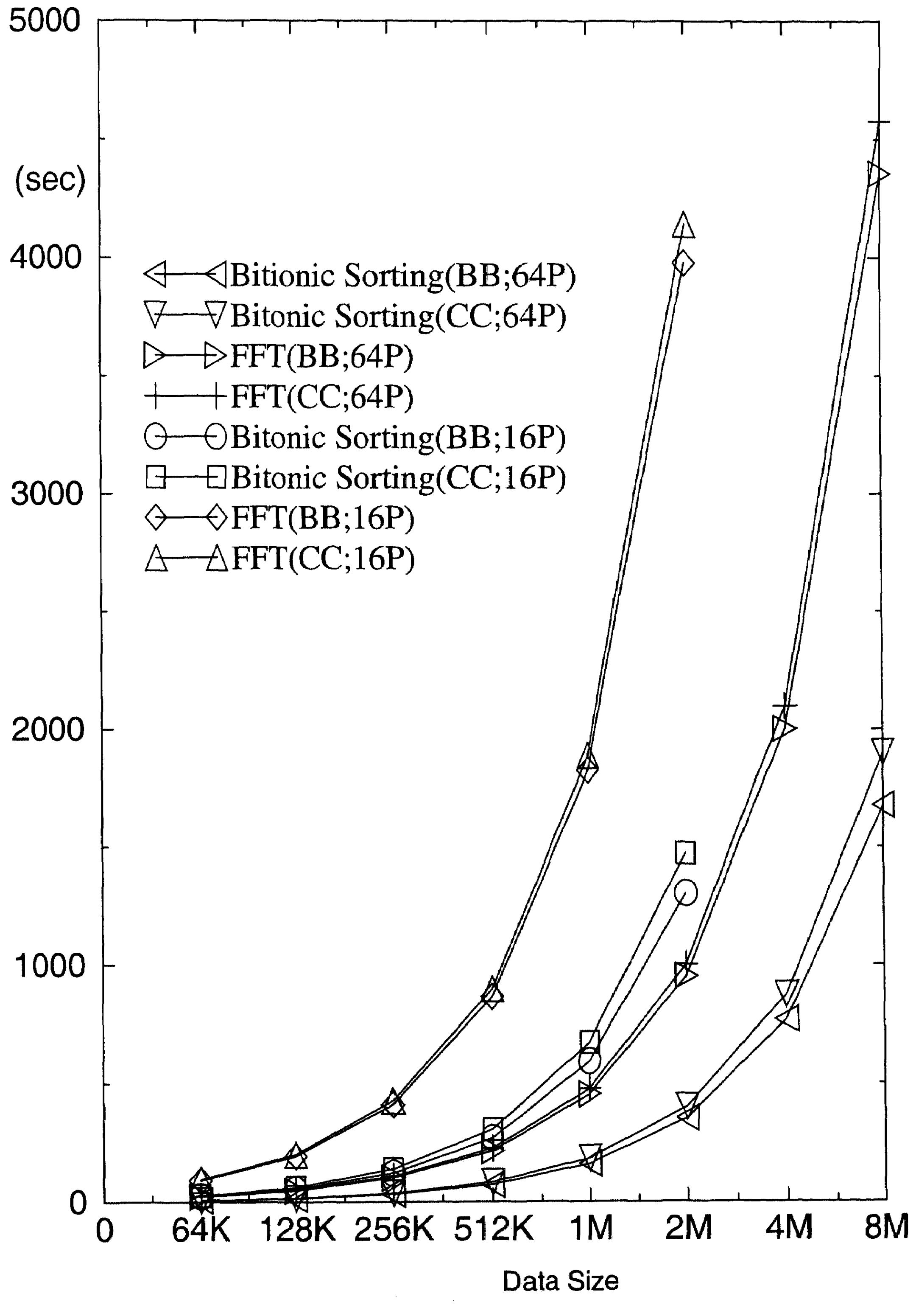


Figure 7-7 Execution Time of Bitonic Sorting and FFT

7.2.3 Compare with Manual Version

FFT especially shows that almost linear speedup can be gained through EM-X, which is because the distribution strategies we used minimize the communication time while the huge computation were evenly distributed to each processor. When we compared it with the hand-written code for the same algorithm, the ratio of execution time of our automatic version to the manual version on the FFT problem is about 0.914-0.934 and 0.911-0.924 for the BB version and CC version respectively(see Figure 7-8). Our FFT automatic version is comparable to the hand-written code when there are no other optimizing efforts for the hand-written code. One of the optimizing efforts is to create sin/cos tables so that we can we can use table lookups and some simple computations to get sin/cos values instead of calling a built-in sin/cos functions to get the value.

With dot product, on the other hand, even though the data size is large, the computation is relatively small according to the nature of the problem itself so that the speedup gained will be less than FFT. No optimizing efforts can be made to hand-written code when we compare it with the hand-written code for the generic algorithm. It is also comparable to the hand-written code. Figure 7-9 shows that the ratio of execution time of our automatic version to the manual version on the dot-product problem is about 1.784-1.851 and 1.943-2.036 for the BB version and CC versions respectively. From Figure 7-9, we can easily see that the BB version is better than the CC version.

The search problem is similar to the dot product only in that its computation time is lower than the dot-product since the problem itself does comparisons all the time; the only computation for this problem is to sum up the frequency. This search problem also can not have any optimizing for the hand-written code. Figure 7-10 shows this search problem has similar results as the dot product problem which is shown in Figure 7-9. Figure 7-9 also shows the BB version is better than the CC version. The noticeable ratio of this search problem, up to 4.409-4.999 and 4.626-5.262 for the BB version and CC version respectively, is believed to result from the tiny computation time.

Although bitonic sorting can get more speedup when the number of processors increases, it is not comparable to the hand-written code. This is because the algorithm we used is the generic bitonic sorting algorithm(see Figure 7-1). The main difference is that the generic algorithm treats an integer as a node while the other algorithm(parallel algorithm actually, see Figure 7-11)[39] treats a processor with a series of sorted integers as a node. Hence, the execution time for our experiments on this problem is relatively slower than another one used for hand-written code. The other one is an optimized parallel version which utilized the capabilities of all the processors involved in the beginning. We did not compare the automatic version to the manual version since it is not a fair comparison.

In Figure 7-11, shaded circles indicate those processors performing ascending order merge while hollow circles indicate processors performing descending order merge. Lines connecting two processors indicate communication to send/receive data. This revised parallel algorithm basically use the concept that two ordered list can be merged into one ordered list. Hence, this parallel algorithm treats a processor with a list of elements(say K elements) sorted as a node. Instead of arranging two relative nodes(two elements) into an order(possibly by a swap operation), this revised algorithm has each node(processor) performing a merge operation: merges a list from another mate processor and the processor's own list and determines which half of newly sorted list to keep and in what order.

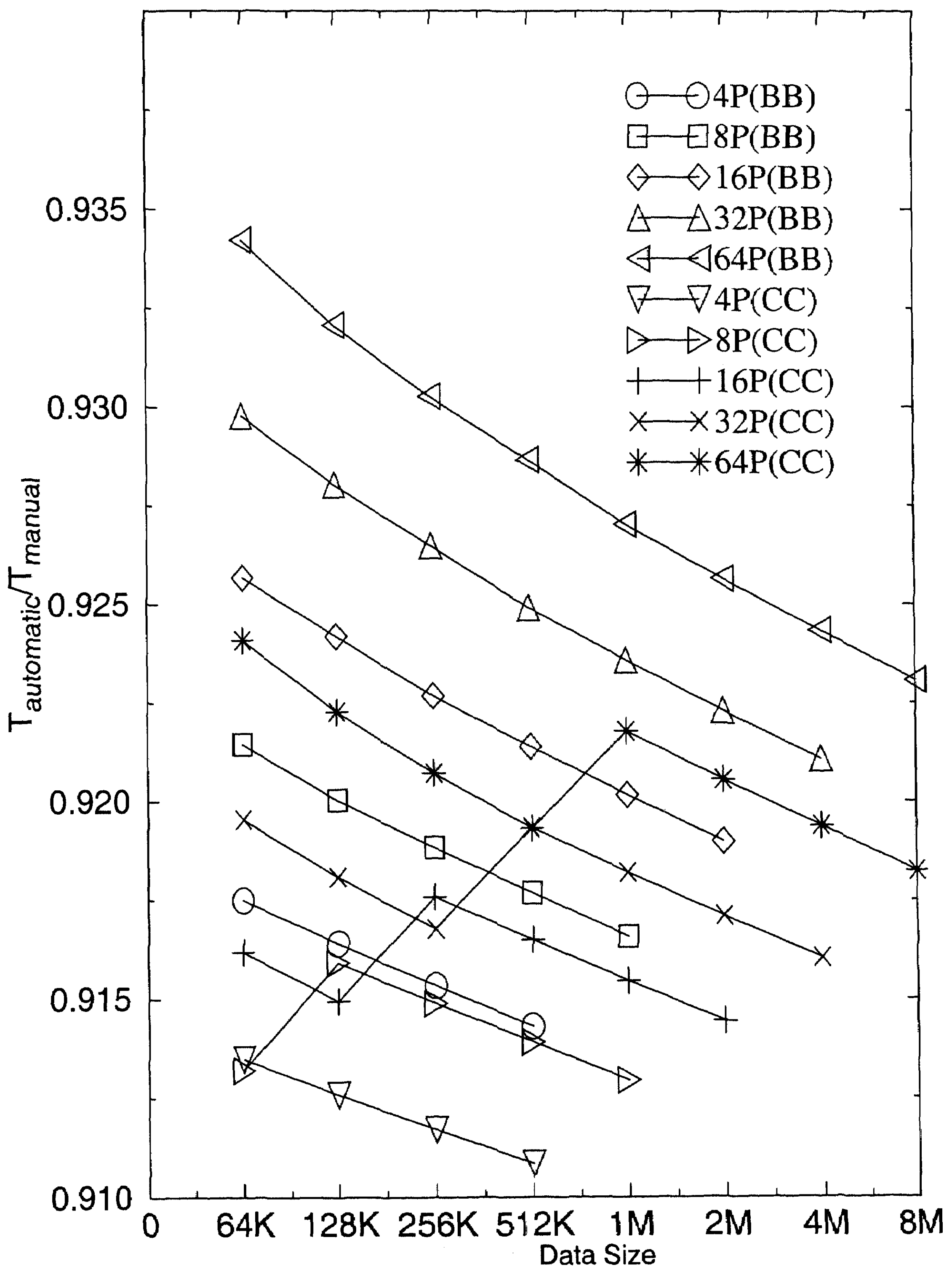


Figure 7-8 Ratio of Execution Time of Automatic to Manual on FFT

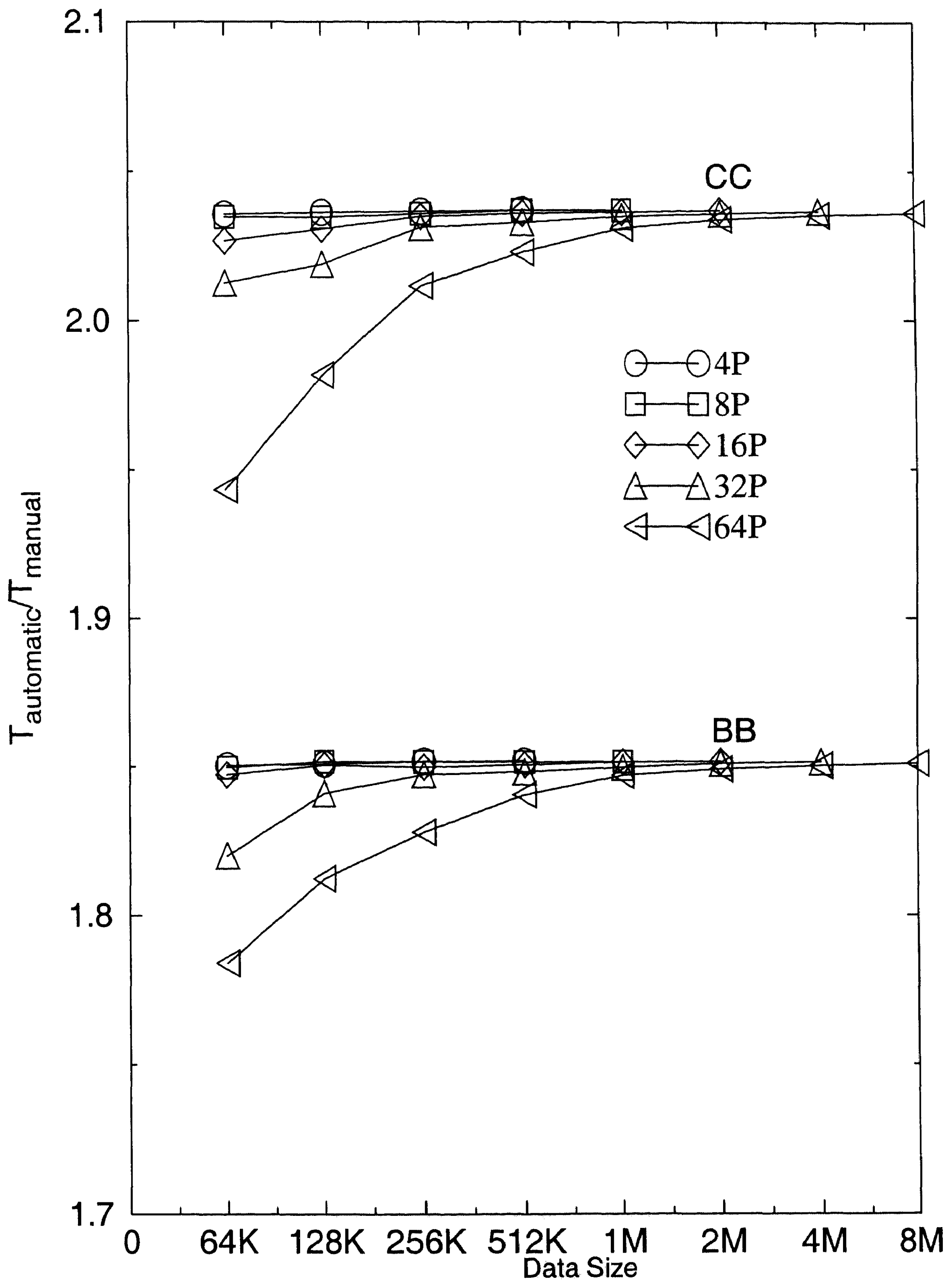


Figure 7-9 Ratio of Execution Time of Automatic to Manual on Dot Product

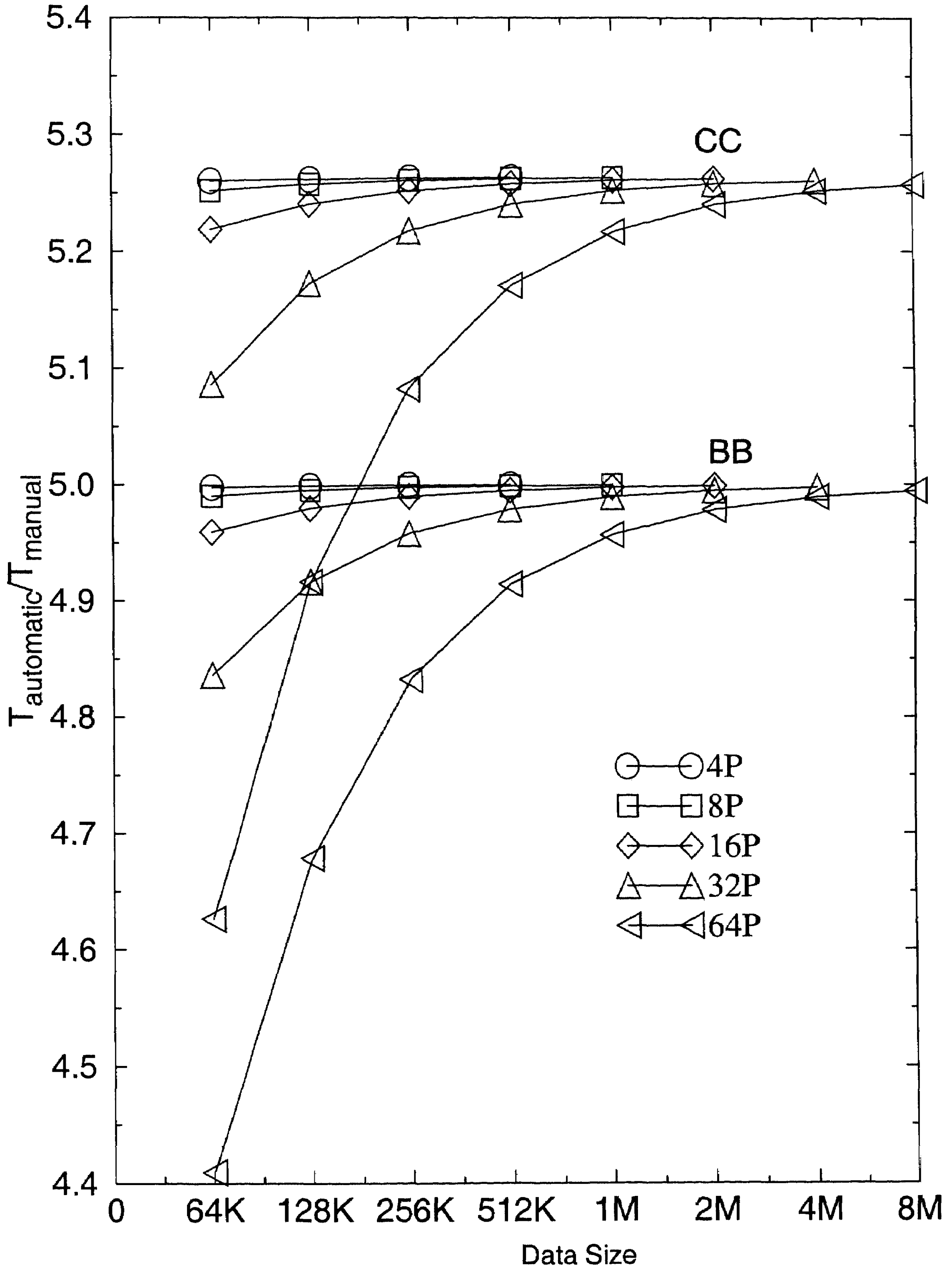


Figure 7-10 Ratio of Execution Time of Automatic to Manual on Search

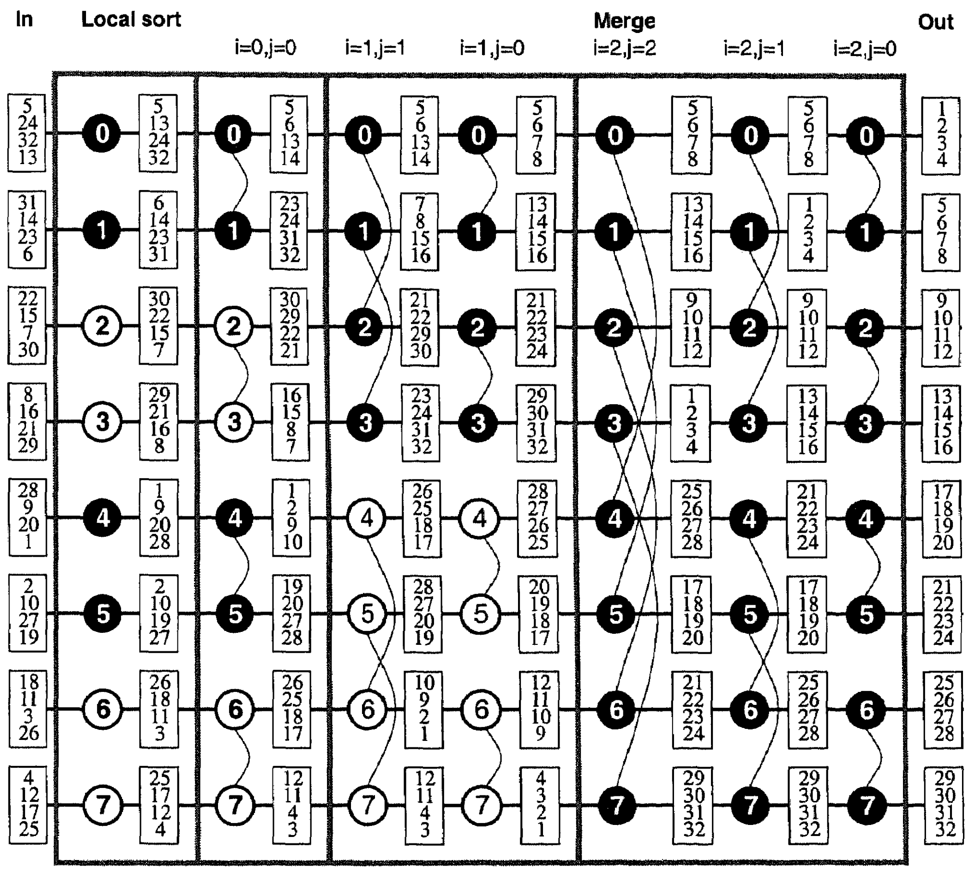


Figure 7-11 Bitonic Sorting of 32 elements on eight processors

CHAPTER 8

CONCLUSIONS

Programmability and performance are the two main key issues for the use of distributed-memory multiprocessors. Researchers expect programmability for easy programming of the parallel codes in the distributed-memory multiprocessors with high scalability. High performance is the utmost expectation for scientific computing no matter what kind of machine or system is used.

Distributed-memory multiprocessors deliver scalable performance but are difficult to program. Programmers need to learn all the abilities to develop a good parallel program. These abilities include identifying potential parallelism of the problems under consideration, developing suitable algorithms to manifest discovered parallelism, and having a good understanding of the characteristics of the problem as well as of the underlying architecture of the target machine. Moreover, programmers need to take care of data distribution, workload distribution, and explicit parallel constructs such as message-passing communication operations or synchronization primitives.

We present an approach which insulates the programmer from the underlying architecture of distributed-memory multiprocessors. Furthermore, it releases most of the burdens from the programmer. The approach uses the Sisal functional language as the developing language; the programmer describes only what the computation is, not the order it must proceed in.

The Optimizing Sisal Compiler will translate a Sisal source program(sequential) into one or more machine independent dataflow graphs which contain all the information regarding the input/output dependencies among the operators that make up the program.

These dependencies constitute the only inhibitions to parallel execution. Based on these graphs, which contain complete knowledge of the program's potential parallelism, the compiler can make judgements about how to partition a program for parallel execution. The IF2GEN and RUNTIME system were adapted for distributed-memory multiprocessors. To be precise for the IF2GEN, a correct parallel program for a particular distribution of data and workload will be generated from inserting the explicit parallel constructs into the code derived from all the knowledge obtained through the compiler. The RUNTIME system was changed to properly support the communication primitives to make sure the code generated from the IF2GEN could be executed on the target machines.

According to our approach, the determination of data dependencies, scheduling of operations, communication of data values, and synchronization of concurrent operations are realized automatically by the compiler. Programmers can be released of most of the burdens in the writing of parallel programs. Instead, programmers can concentrate on problem solving for algorithmic improvement or the quality of solutions.

We use loop level parallelism, commonly exploited by parallelizing compilers, to do the data parallelism for the four problems selected which contain two benchmark problems, fine-grain bitonic sorting and FFT. The other two problems are the dot-product and search problem. Only a distributed-memory multiprocessor is used in our experiments, which is the laboratory prototype EM-X with 80 processors. The multithreading capability of EM-X was turned off in this dissertation by using only one thread. Instead, the remote by-passing mechanism was used to reduce the overhead of the synchronization. From the information gained in previous experimental results, only the same data distribution and workload distribution will form a good match and perform a good performance. Two commonly used distribution strategies, blocked and cyclic, are selected. These options indicate "the blocked data distribution and workload distribution version"(BB) and "the cyclic data distribution and workload distribution version"(CC) respectively. All problems have been implemented in Sisal and executed on the EM-X machine.

The experimental results of the four problems - dot product, search, bitonic sorting and FFT- all indicate that there will be more speedup when we increase the number of processors while the data size is fixed or when we increase the data size while the number of processors remains the same(see Figure A-5, A-6, A-7 and A-8). This speedup occurs regardless of whether the BB or CC version is used. Furthermore, the BB version is superior to the CC version especially when data size is large according to the discussions in section 7.2. It also matches the previous experiments discussed in Chapter 5 which show that the communication overhead on the BB version is the lowest.

The experimental results on EM-X show that our approach successfully delivers the programmability via the Sisal functional language as well as it performs a reasonable performance. When we look into the Figures A-7, A-5, A-8 and A-6, they show our approach generates up to 17.9,17.36,32.0,29.1, 32, 30.4,59.5 and 61.3 speedup on the 64 processors for bitonic sorting(BB), bitonic sorting(CC), dot product (BB), dot product (CC), search(BB), search(CC), FFT(BB) and FFT(CC) respectively. Figure A-5 draws the raw data of the Table A-13 and A-14. Figure A-6 plots the raw data of the Table A-15 and A-16. Figure A-7 shows the figure from the data shown in Table A-17 and A-18 while Figure A-8 represents the data from Table A-19 and A-20.

According to the distribution of execution time in these four problems with BB and CC, we could find out that the distribution percentages are stable when the data size is large for both communication time and overhead time (for computing all information of each data element including which processor locates the data element and in what place).

The ratio of execution time of the code from our approach(automatic version) to the manual version is about 0.914-0.934, 0.911-0.924, 1.784-1.851,1.943-2.036, 4.409-4.999 and 4.626-5.262; all of them are related to Figures 7-8, 7-9 and 7-10 for the FFT(both the BB and CC versions), dot product(including the BB and CC versions) and search(including the BB and CC versions) respectively. The results show that FFT problem is competitive to hand-craft version while both dot product and search problems are a little inferior to their corresponding hand-craft version. But, it will not matter a lot because of their low execution time.

The algorithm we could use in our approach for bitonic sorting problem is the generic bitonic sorting algorithm(see Figure 7-1), which is not commonly used now. Instead of using this generic algorithm which treats an integer as a node, researchers now use the algorithm which treats a processor with a series of sorted integers as a node(see Figure 7-11). Hence, the execution time for our experiments on this problem is relatively slower than the other algorithm used for hand-craft code since that algorithm is an optimized parallel version and utilized the capabilities of all the processors involved in the beginning. It is of no use to compare this automatic version to the manual version for this bitonic sorting problem.

Our approach also has some drawbacks for the approach itself and the machine used. The SISAL functional language is originally ported to the share-memory multiprocessors. We use its runtime system, which will cause some problems because originally it did not consider distributed-memory environment.

First of all, it uses some memory-free work to fit to the share-memory environment which leads to a problem since not all distribution-memory multiprocessors provide memory-free function.

Second of all, for the data representation, it use a temp value to indicate an array element which could be a data value, a one-dimension array or a two-dimension array. For the example in Figure 8-1, tmp15 is an integer variable, tmp24 and tmp25 are array elements with float variable, tmp18 and tmp23 indicate a one-dimension array while tmp9 and tmp10 all represent a two-dimension array. This kind of data representation does not have trouble in share-memory multiprocessors. However, it will cause a lot of trouble when using distributed-memory multiprocessors, especially when it encounters two-dimension arrays tmp9 and tmp10 with a certain data distribution applied to the array tmp9 and tmp10.

```
while (tmp15 <= tmp16) {
tmp18 = ((Pointer*)(((ARRAYP)tmp9)->Base))[tmp15];
   while (tmp20 <= tmp7) {
   tmp22 =1;
   tmp21 = 0.0
      while (tmp22 <= tmp8) {
      tmp23 = ((Pointer*)(((ARRAYP)tmp10)->Base))[tmp22];
      float1 = (float*)(((ARRAYP)tmp23)->Base);
      tmp24 = float1[tmp20];
      float2 = (float*)(((ARRAYP)tmp18)->Base);
      tmp25 = float2[tmp22];
      tmp22++;

      }
   tmp20++;
   }

tmp15 ++;
}
```

Figure 8-1 A sample program

Finally, we use the SISAL sequential program as the input to generate its corresponding parallel program. In this method, it will cause the problem that only sequential algorithms are good for this approach such as FFT, search, and dot product. It will also cause all optimization methods for parallel versions useless for our approach such as the algorithm researchers use now for the bitonic sorting problem.

Above all, drawbacks are basically from the SISAL language and approach itself. The target machine, ETL-EMX, has two drawbacks for our approach, too. The first one is the ETL-EMX machine does not provide memory-free functions so it will cause trouble when memory-free functions are generated. The second drawback is the ETL-EMX multiprocessor does not provide block read operation(we use remote read for our approach) so it makes our approach favors fine-grain problem(those problems use one-dimension arrays) since we are using the outmost loop parallelism. We only need to encounter the array element with a simple data type.

To sum up, our approach is good for any fine-grain problem since it shows reasonable results. But, it also wipes out almost all parallel algorithm because of its nature. We will try to solve all middle-grain problems when we get a chance to use a machine such as the Cray T3E even though not a lot of scientific computing problems use multi-dimension arrays.

CHAPTER 9

FUTURE WORK

9.1 Multithreading

Multithreading aims at tolerating remote memory latency through a split-phase read mechanism and context switch[32,33,34,35,36,37,38,40]. 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[37]. The Heterogeneous Element Processor (HEP) designed by Burton Smith provides up to 128 threads. A thread switch occurs in every instruction with the 100 nsec switching cost. Threads are usually ended by remote read instructions since they may incur long latencies if the requested data is located in a remote processor. The Monsoon data-flow machine developed at MIT switches context every instruction, where a thread consists of a single instruction [25].

The EM-4 multiprocessor provides hardware support for multithreading [38,44]. Thread switch takes place whenever a remote memory read is encountered. Threads can also be suspended with explicit thread scheduling. The Alewife multiprocessor provides hardware support for multithreading [45]. Together with prefetching, block multithreading with four hardware contexts have 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 [16,21]. Up to a maximum of 128 threads is provided per processor. Context switch can take place every instruction as every memory access is a "remote" access because there is one flat memory hierarchy. The RWC-1 prototype minimizes the context switch overhead by prefetching [20].

An analytic model for multithreading is studied in [6]. 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 [46]. The study based on simulated multithreading further indicated that multiple hardware contexts have limited effects on unoptimized programs. 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 [41]. The EARTH multiprocessor emulates multithreading by using an execution unit and a synchronization unit in each processor [47]. The Execution unit initiates a remote read while the Synchronization unit completes the communication. The thread partitioning and scheduling algorithm based on a cost model is used to study the performance of the EARTH multiprocessor [31].

Some experimental results on EM-4, however, indicated that simple-minded data distribution and workload distribution can give performance comparable to that of the best performing algorithms with hand-crafted data distribution but no threading[41].

The multithreading capability of EM-X was turned off in this dissertation by using only one thread since the design of this dissertation is focused on our approach which can fit with various kinds of distributed-memory multiprocessors. We will turn on the multithreading capability of the EM-X for all problems experienced in this thesis so that we can see the effects of threading through the performance.

9.2 Portability

This dissertation focuses on programmability and performance. Therefore, we use the functional language, Sisal, to be the developing language to enhance the programmability for the programmers. To improve the performance, we also decided on the parallel programming paradigm used in each target machine with one-sided communication.

The utmost portability has not fully been fulfilled from our approach. The mature MPI-2 with one-sided communication will be the best parallel programming paradigm with our approach for performance as well as to deliver portability. In other words, the code generated by the IF2GEN will be the parallel program written by MPI-2 with one-sided communication ability. It can be ported to all distributed-memory machines which recognize MPI-2, keep the performance by one-sided communication ability as well as keep the programmability unchanged since the programmer only needs to know the Sisal functional language.

APPENDIX A.1 EXECUTION TIME

Table A-1 BB version for Dot Product

	64K	128K	256K	512K	1M	2M	4M	8M
1	0.164	0.328	0.655	1.311	2.621	5.243	10.486	20.972
2	0.164	0.328	0.655	1.311	2.621	5.243	10.486	20.972
4	0.082	0.164	0.327	0.655	1.311	2.621	5.243	10.486
8	0.041	0.082	0.164	0.328	0.655	1.311	2.622	5.243
16	0.021	0.041	0.082	0.164	0.328	0.656	1.311	2.622
32	0.010	0.021	0.041	0.082	0.164	0.328	0.656	1.311
64	0.006	0.011	0.021	0.041	0.082	0.164	0.328	0.656

Table A-2 CC version for Dot Product

	64K	128K	256K	512K	1M	2M	4M	8M
1	0.164	0.328	0.655	1.311	2.621	5.243	10.486	20.972
2	0.180	0.360	0.721	1.442	2.884	5.767	11.534	23.069
4	0.090	0.180	0.360	0.721	1.442	2.884	5.767	11.534
8	0.045	0.090	0.180	0.360	0.721	1.442	2.884	5.767
16	0.023	0.045	0.090	0.180	0.361	0.721	1.442	2.884
32	0.012	0.023	0.045	0.090	0.180	0.361	0.721	1.442
64	0.006	0.012	0.023	0.046	0.091	0.181	0.361	0.721

Table A-3 CC/BB for Dot Product

	64K	128K	256K	512K	1M	2M	4M	8M
2	1.1	1.1	1.1	1.1	1.1	1.1	1.1	1.1
4	1.1	1.1	1.1	1.1	1.1	1.1	1.1	1.1
8	1.1	1.1	1.1	1.1	1.1	1.1	1.1	1.1
16	1.1	1.1	1.1	1.1	1.1	1.1	1.1	1.1
32	1.1	1.1	1.1	1.1	1.1	1.1	1.1	1.1
64	1.09	1.09	1.1	1.1	1.1	1.1	1.1	1.1

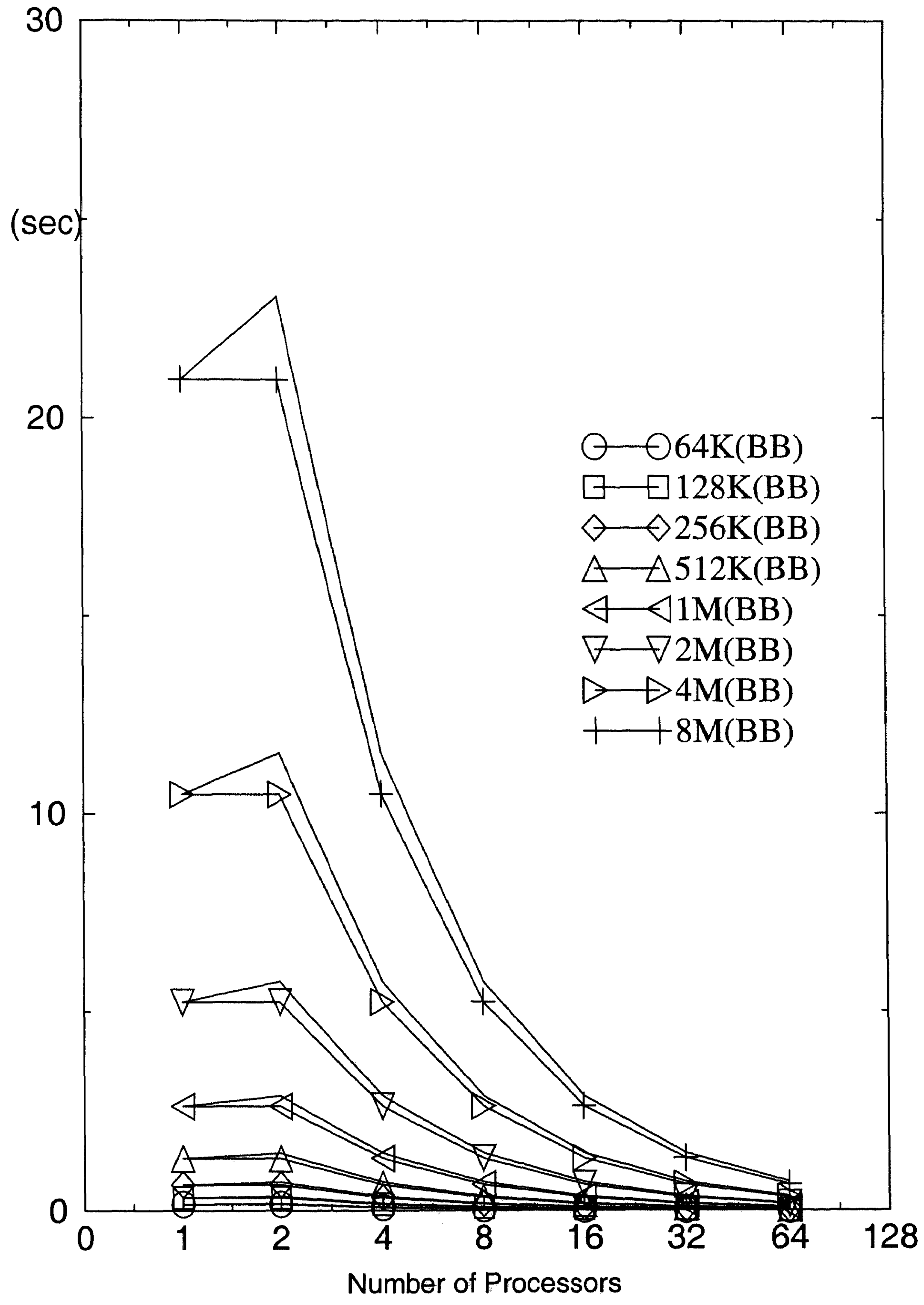


Figure A-1 The Execution Time for Dot Product with BB Version and CC Version(The one unmarked)

Table A-4 BB version for FFT

	1K	2K	4K	8K	16K	32K	64K	128K
1	11.332	25.068	54.968	119.65	258.80	556.74	1192.1	2541.9
2	6.099	13.424	29.314	63.589	137.15	294.3*	628.7*	1338*
4	3.073	6.755	14.741	31.96	68.903	147.82	315.7*	671.8*
8	1.552	3.404	7.419	16.073	34.634	74.275	158.61	337.4*
16	0.786	1.718	3.736	8.085	17.409	37.317	79.663	169.43
32	0.400	0.87	1.885	4.07	8.755	18.753	40.016	85.083
64	0.207	0.444	0.955	2.054	4.407	9.429	20.106	42.730

Table A-5 CC version forFFT

	1K	2K	4K	8K	16K	32K	64K	128K
1	11.332	25.068	54.968	119.65	258.80	556.74	1192.1	2541.9
2	5.826	12.876	28.219	61.398	132.76	286.7*	613.8*	1308*
4	2.930	6.469	14.168	30.813	66.608	143.23	307.8*	656.2*
8	1.478	3.256	7.123	15.479	33.446	71.896	153.85	329.3*
16	0.749	1.643	3.585	7.781	16.801	36.100	77.277	164.56
32	0.382	0.832	1.809	3.917	8.446	18.135	38.777	82.604
64	0.198	0.425	0.917	1.977	4.252	9.117	19.481	41.480

Table A-6 CC/BB for FFT

	1K	2K	4K	8K	16K	32K	64K	128K
2	0.96	0.96	0.96	0.97	0.97	0.97*	0.98*	0.98*
4	0.95	0.96	0.96	0.96	0.97	0.97	0.98*	0.98*
8	0.95	0.96	0.96	0.96	0.97	0.97	0.97	0.98*
16	0.95	0.96	0.96	0.96	0.97	0.97	0.97	0.97
32	0.95	0.96	0.96	0.96	0.96	0.97	0.97	0.97
64	0.96	0.96	0.96	0.96	0.96	0.97	0.97	0.97

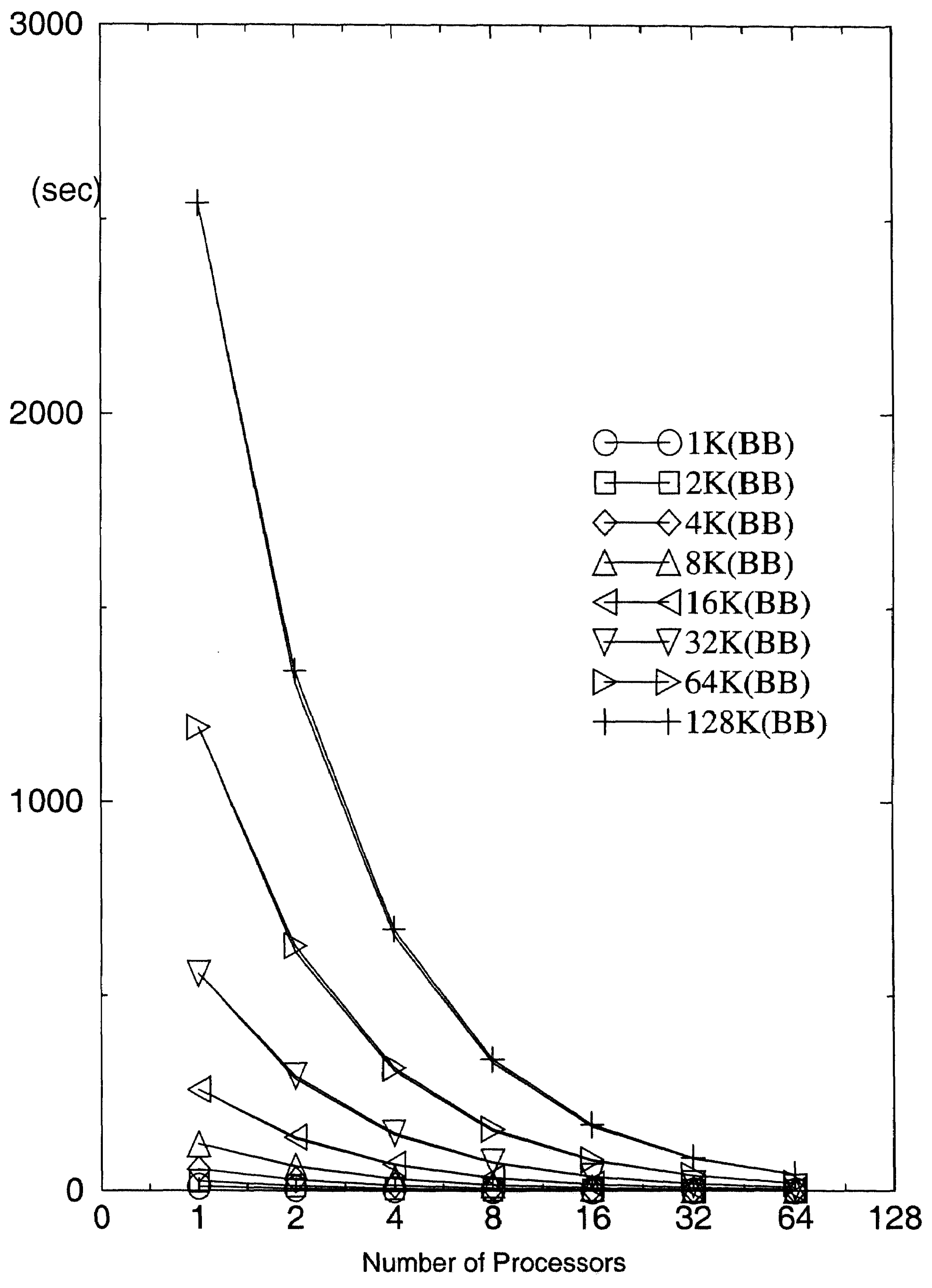


Figure A-2 The Execution Time for FFT with BB Version and CC Version(The one unmarked)

Table A-7 BB version for Bitonic Sorting

	1K	2K	4K	8K	16K	32K	64K	128K
1	0.535	1.280	3.023	7.050	16.264	37.166	84.226	189.47
2	0.718	1.714	4.038	9.404	21.677	49.09*	111.3*	250.3*
4	0.383	0.913	2.147	4.994	11.500	26.02*	58.97*	132.6*
8	0.207	0.489	1.148	2.665	6.129	13.977	31.37*	70.56*
16	0.112	0.264	0.613	1.421	3.261	7.427	16.790	37.42*
32	0.061	0.142	0.331	0.757	1.735	3.944	8.904	19.979
64	0.034	0.077	0.177	0.409	0.926	2.099	4.730	10.600

Table A-8 CC version for Bitonic Sorting

	1K	2K	4K	8K	16K	32K	64K	128K
1	0.535	1.280	3.023	7.050	16.264	37.166	84.226	189.47
2	0.731	1.744	4.107	9.560	22.027	55.13*	124.9*	281.0*
4	0.3922	0.9335	2.1945	5.0997	11.736	29.18*	66.08*	148.6*
8	0.2138	0.5039	1.1814	2.7402	6.2961	14.347	35.17*	79.06*
16	0.1152	0.2724	0.6323	1.4639	3.3577	7.6406	17.262	41.81*
32	0.0625	0.1462	0.3408	0.7808	1.7882	4.0621	9.1664	20.556
64	0.0346	0.0792	0.1825	0.4212	0.9539	2.1632	4.8735	10.916

Table A-9 CC/BB for Bitonic Sorting

	1K	2K	4K	8K	16K	32K	64K	128K
2	1.02	1.02	1.02	1.02	1.02	1.12*	1.12*	1.12*
4	1.02	1.02	1.02	1.02	1.02	1.12*	1.12*	1.12*
8	1.03	1.03	1.03	1.03	1.03	1.03	1.12*	1.12*
16	1.03	1.03	1.03	1.03	1.03	1.03	1.03	1.12*
32	1.03	1.03	1.03	1.03	1.03	1.03	1.03	1.03
64	1.03	1.03	1.03	1.03	1.03	1.03	1.03	1.03

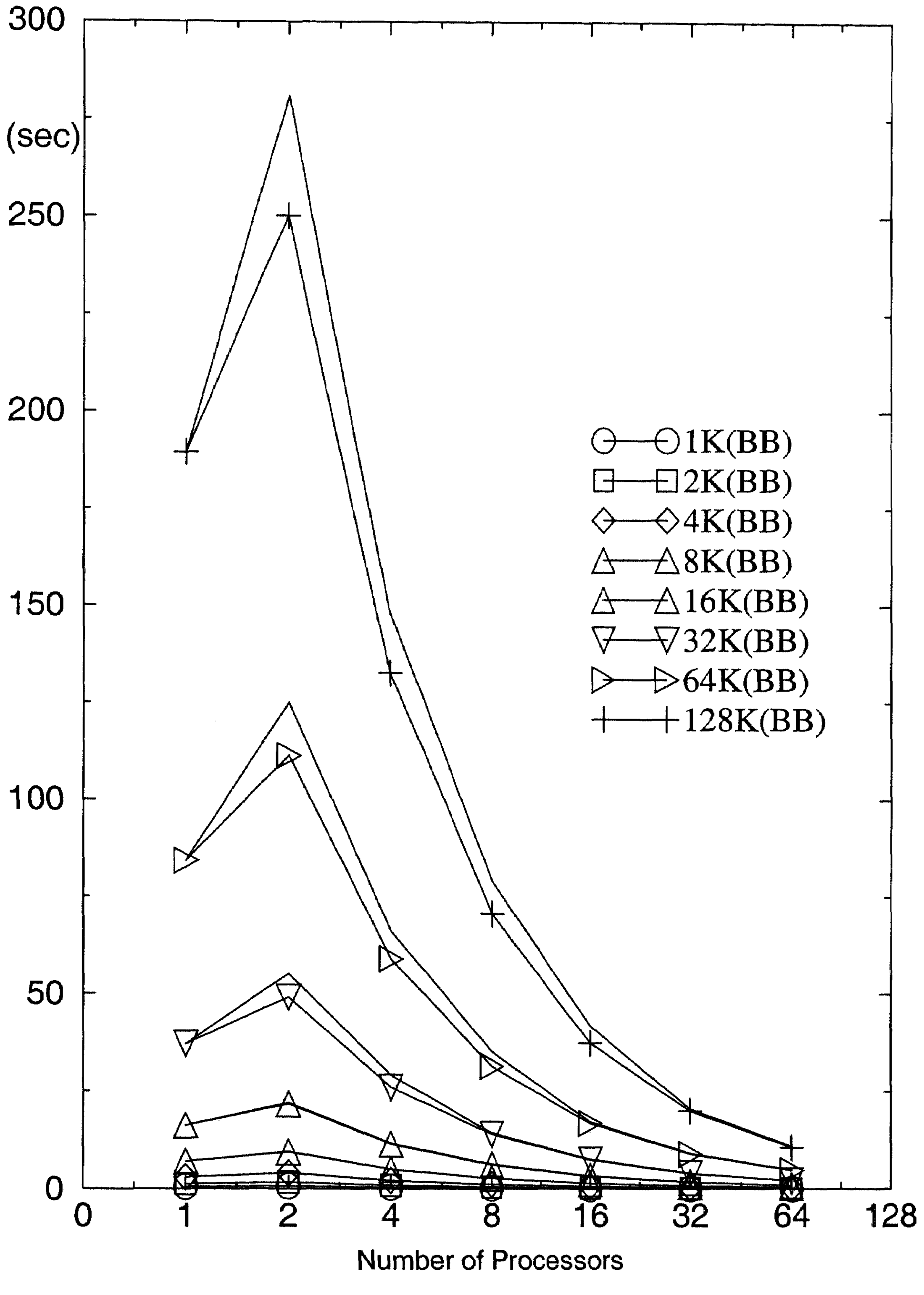


Figure A-3 The Execution Time for Bitonic Sorting with BB Version and CC Version(The one unmarked)

Table A-10 BB version for Search

	64K	128K	256K	512K	1M	2M	4M	8M
1	0.311	0.623	1.245	2.490	4.981	9.961	19.923	39.846
2	0.311	0.623	1.245	2.490	4.981	9.961	19.923	39.846
4	0.156	0.311	0.623	1.245	2.490	4.981	9.962	19.923
8	0.078	0.156	0.311	0.623	1.245	2.490	4.981	9.962
16	0.039	0.078	0.156	0.311	0.623	1.245	2.491	4.981
32	0.020	0.039	0.078	0.156	0.311	0.623	1.245	2.491
64	0.010	0.020	0.039	0.078	0.156	0.311	0.623	1.246

Table A-11 CC version for Search

	64K	128K	256K	512K	1M	2M	4M	8M
1	0.311	0.623	1.245	2.490	4.981	9.961	19.923	39.846
2	0.328	0.655	1.311	2.621	5.243	10.486	20.971	41.943
4	0.164	0.328	0.655	1.311	2.621	5.243	10.486	20.972
8	0.082	0.164	0.328	0.655	1.311	2.621	5.243	10.486
16	0.041	0.082	0.164	0.328	0.655	1.311	2.622	5.243
32	0.021	0.041	0.082	0.164	0.328	0.655	1.311	2.622
64	0.011	0.021	0.041	0.082	0.164	0.328	0.656	1.311

Table A-12 CC/BB for Search

	64K	128K	256K	512K	1M	2M	4M	8M
2	1.053	1.053	1.053	1.053	1.053	1.053	1.053	1.053
4	1..053	1.053	1.053	1.053	1.053	1.053	1.053	1.053
8	1.052	1.053	1.053	1.053	1.053	1.053	1.053	1.053
16	1.052	1.052	1.053	1.053	1.053	1.053	1.053	1.053
32	1.052	1.052	1.052	1.052	1.053	1.053	1.053	1.053
64	1.049	1.051	1.052	1.052	1.052	1.053	1.053	1.053

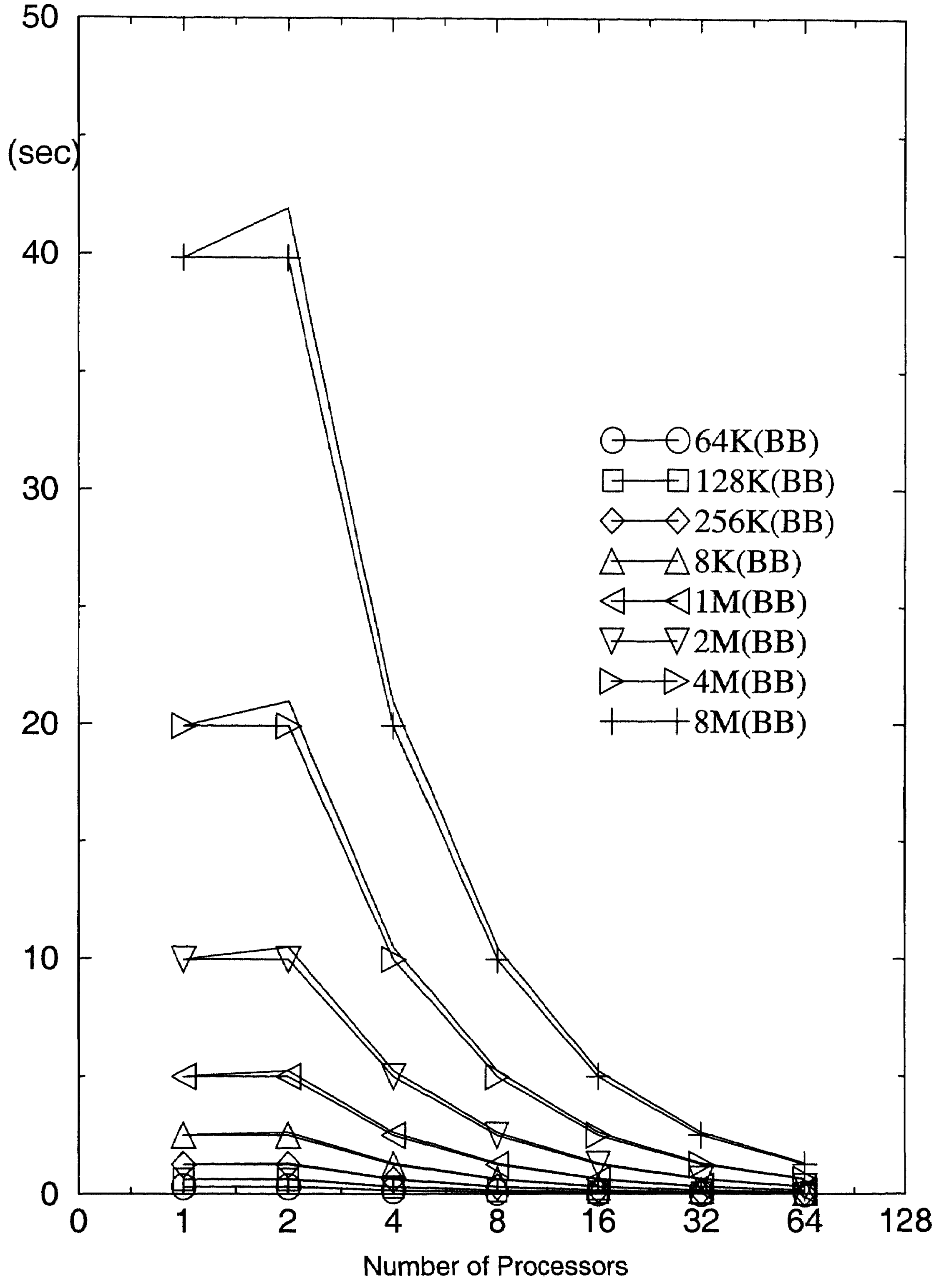


Figure A-4 The Execution Time for Search with BB Version and CC Version(The one unmarked)

APPENDIX A.2 SPEEDUP

Table A-13 BB version for Dot-Product(Speedup)

	64K	128K	256K	512K	1M	2M	4M	8M
2	1	1	1	1	1	1	1	1
4	2	2	2	2	2	2	2	2
8	3.99	4	4	4	4	4	4	4
16	7.95	7.97	7.99	7.99	8	8	8	8
32	15.61	15.8	15.9	15.95	15.98	15.99	15.99	16.00
64	29.16	30.51	31.24	31.61	31.81	31.9	31.95	31.98

Table A-14 CC version for Dot-Product(Speedup)

	64K	128K	256K	512K	1M	2M	4M	8M
2	0.91	0.91	0.91	0.91	0.91	0.91	0.91	0.91
4	1.82	1.82	1.82	1.82	1.82	1.82	1.82	1.82
8	3.63	3.63	3.63	3.64	3.64	3.64	3.64	3.64
16	7.23	7.25	7.26	7.27	7.27	7.27	7.27	7.27
32	14.23	14.39	14.47	14.51	14.53	14.54	14.54	14.54
64	26.75	27.87	28.48	28.78	28.93	29.01	29.05	29.07

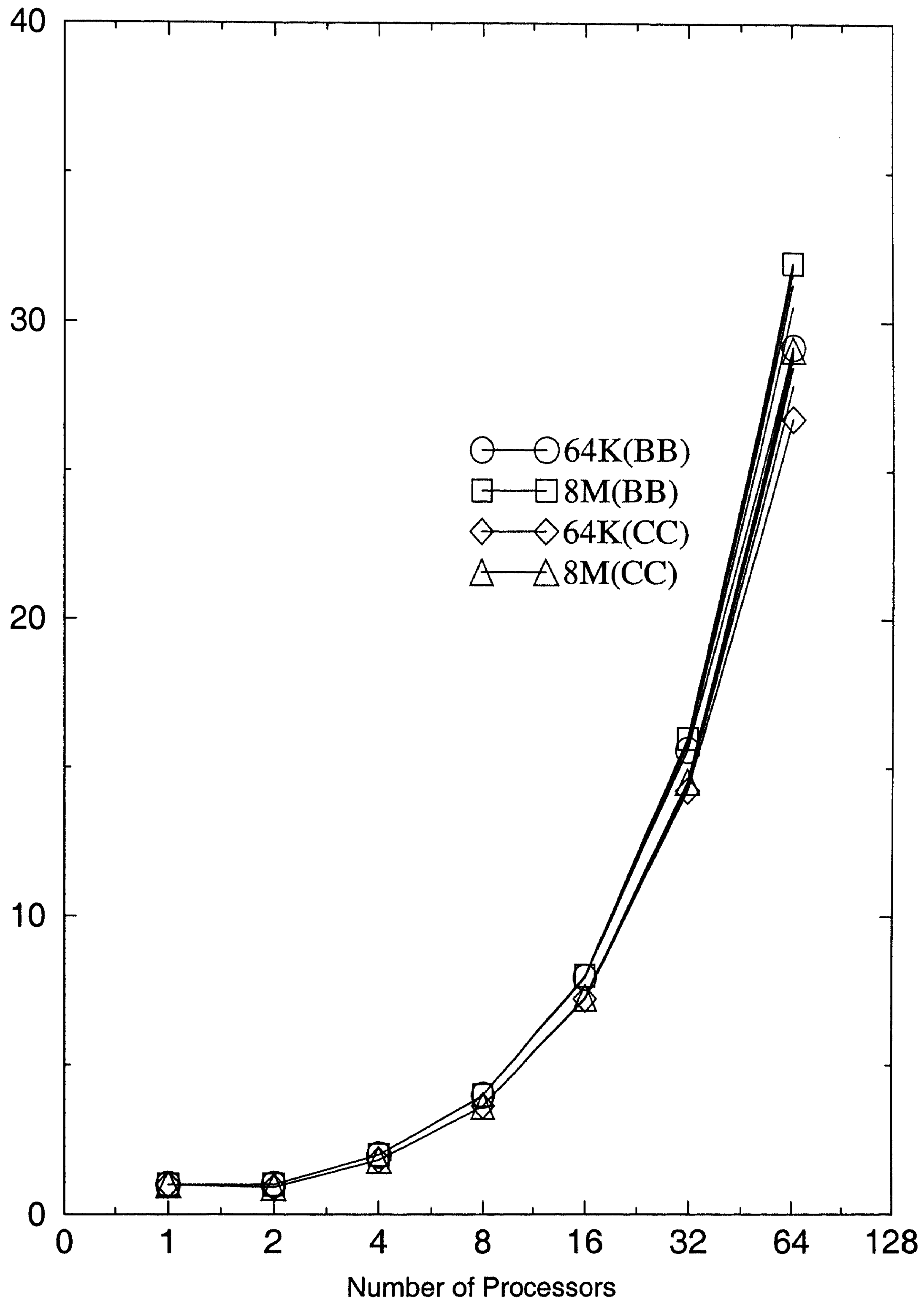


Figure A-5 The Speedup for Dot Product with BB and CC Version

Table A-15 BB version for FFT(Speedup)

	1K	2K	4K	8K	16K	32K	64K	128K
2	1.86	1.87	1.88	1.88	1.89	1.89*	1.9*	1.9*
4	3.69	3.71	3.73	3.74	3.76	3.77	3.78*	3.78*
8	7.3	7.36	7.41	7.44	7.47	7.5	7.52	7.53*
16	14.41	14.59	14.71	14.8	14.86	14.92	14.96	15
32	28.26	28.81	29.15	29.39	29.56	29.69	29.79	29.88
64	54.57	56.44	57.75	58.24	58.72	59.04	59.29	59.49

Table A-16 CC version for FFT(Speedup)

	1K	2K	4K	8K	16K	32K	64K	128K
2	1.95	1.95	1.95	1.95	1.95	1.94*	1.94*	1.94*
4	3.87	3.87	3.88	3.88	3.89	3.89	3.87*	3.87*
8	7.66	7.7	7.72	7.73	7.74	7.74	7.75	7.72*
16	15.13	15.25	15.33	15.37	15.4	15.42	15.44	15.45
32	29.62	30.11	30.38	30.54	30.64	30.7	30.74	30.77
64	57	58.91	59.94	60.51	60.86	61.06	61.19	61.28

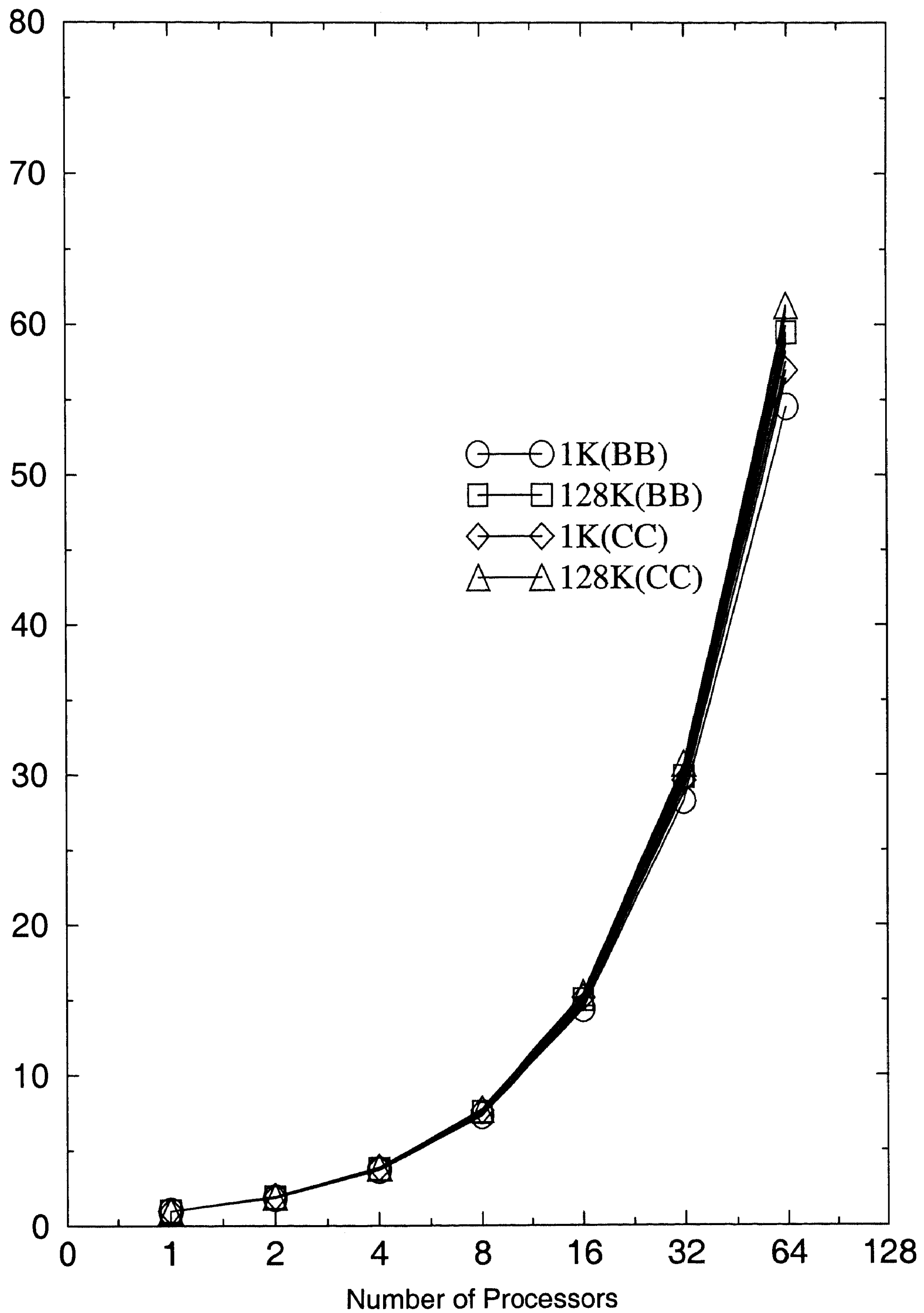


Figure A-6 The Speedup for FFT with BB and CC Version

Table A-17 BB version for Bitonic Sorting(Speedup)

	1K	2K	4K	8K	16K	32K	64K	128K
2	0.75	0.75	0.75	0.75	0.75	0.76*	0.76*	0.76*
4	1.39	1.4	1.41	1.41	1.41	1.43*	1.43*	1.43*
8	2.58	2.62	2.63	2.65	2.65	2.66	2.68*	2.69*
16	4.79	4.85	4.93	4.96	4.99	5	5.02	5.06*
32	8.8	9.03	9.15	9.31	9.37	9.42	9.46	9.48
64	15.83	16.63	17.06	17.25	17.57	17.71	17.81	17.88

Table A-18 CC version for Bitonic Sorting(Speedup)

	1K	2K	4K	8K	16K	32K	64K	128K
2	0.73	0.73	0.74	0.74	0.74	0.67*	0.67*	0.67*
4	1.36	1.37	1.38	1.38	1.39	1.27*	1.27*	1.28*
8	2.5	2.54	2.56	2.57	2.58	2.59	2.39*	2.4*
16	4.64	4.7	4.78	4.82	4.84	4.86	4.88	4.53*
32	8.54	8.76	8.87	9.03	9.1	9.15	9.19	9.22
64	15.42	16.17	16.56	16.74	17.05	17.18	17.28	17.36

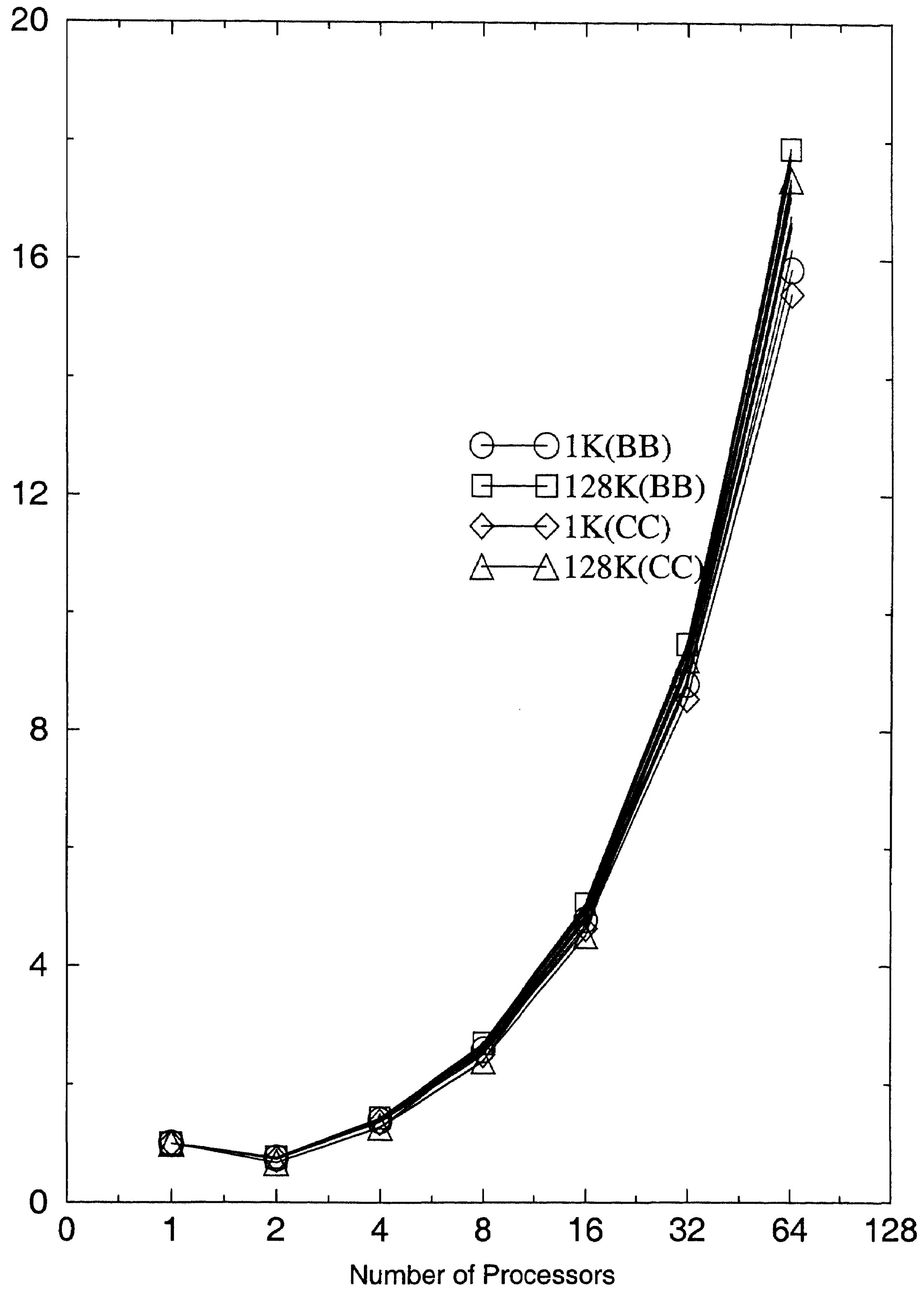


Figure A-7 The Speedup for Bitonic Sorting with BB and CC Version

Table A-19 BB version for Search(Speedup)

	64K	128K	256K	512K	1M	2M	4M	8M
2	1	1	1	1	1	1	1	1
4	2	2	2	2	2	2	2	2
8	4	4	4	4	4	4	4	4
16	7.97	7.99	7.99	8	8	8	8	8
32	15.79	15.9	15.95	15.97	15.99	15.99	16	16
64	30.43	31.2	31.59	31.79	31.9	31.95	31.97	31.99

Table A-20 CC version for Search(Speedup)

	64K	128K	256K	512K	1M	2M	4M	8M
2	0.95	0.95	0.95	0.95	0.95	0.95	0.95	0.95
4	1.9	1.9	1.9	1.9	1.9	1.9	1.9	1.9
8	3.8	3.8	3.8	3.8	3.8	3.8	3.8	3.8
16	7.58	7.59	7.59	7.6	7.6	7.6	7.6	7.6
32	15.02	15.11	15.15	15.18	15.19	15.19	15.2	15.2
64	29	29.69	30.04	30.22	30.31	30.35	30.38	30.39

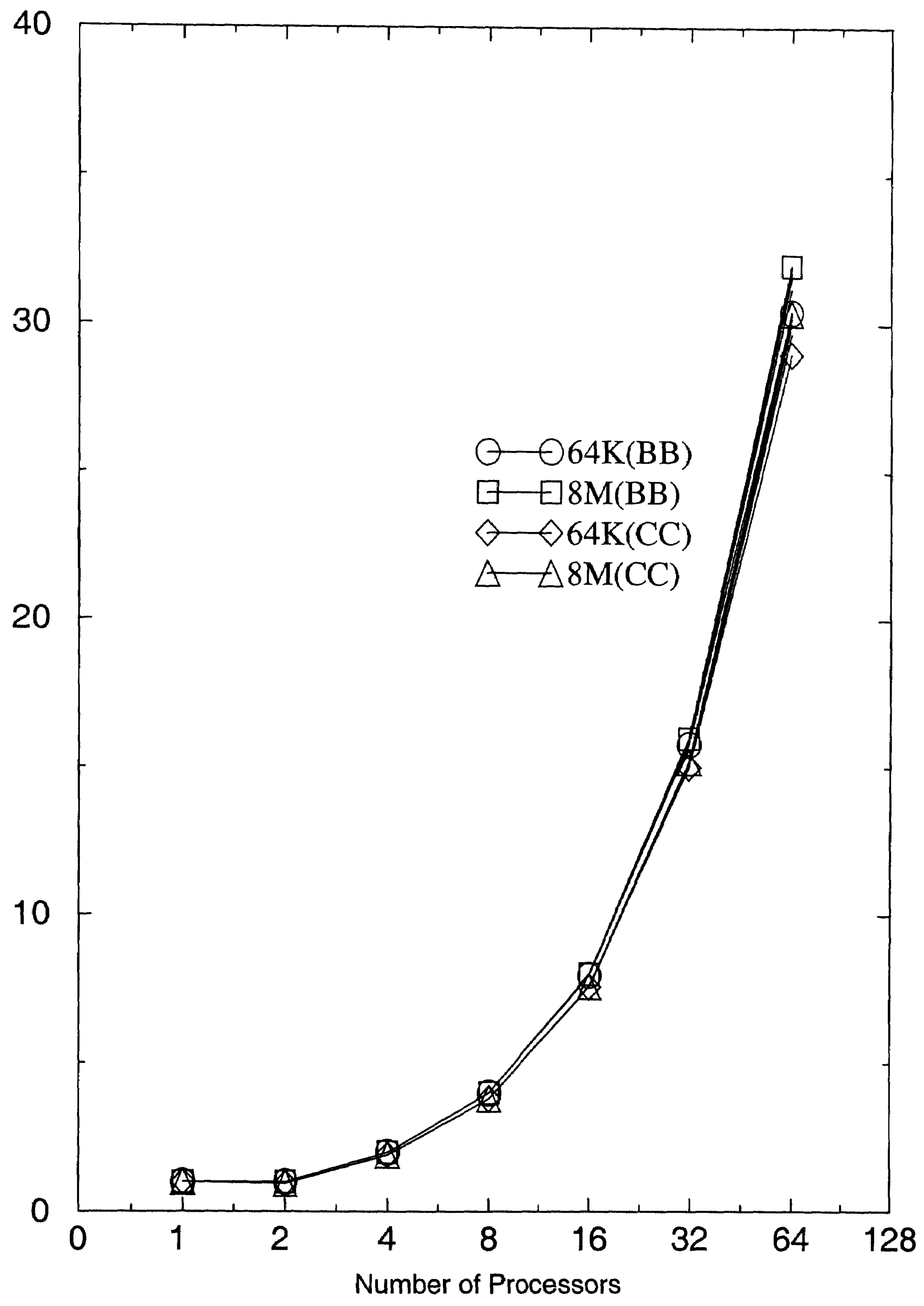


Figure A-8 The Speedup for Search with BB and CC Version

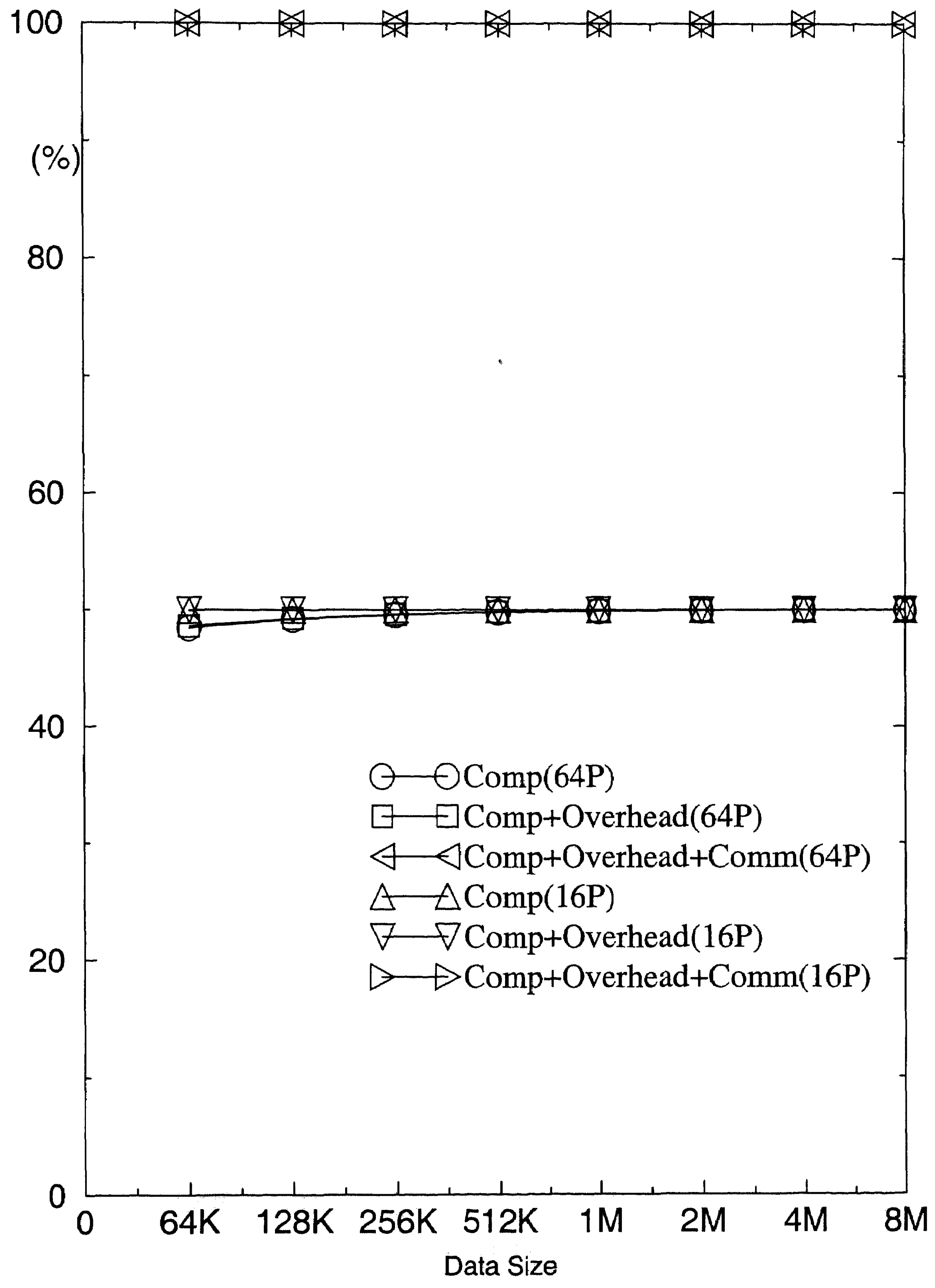


Figure A-9 Distribution of Execution Time on Dot-Product(BB)

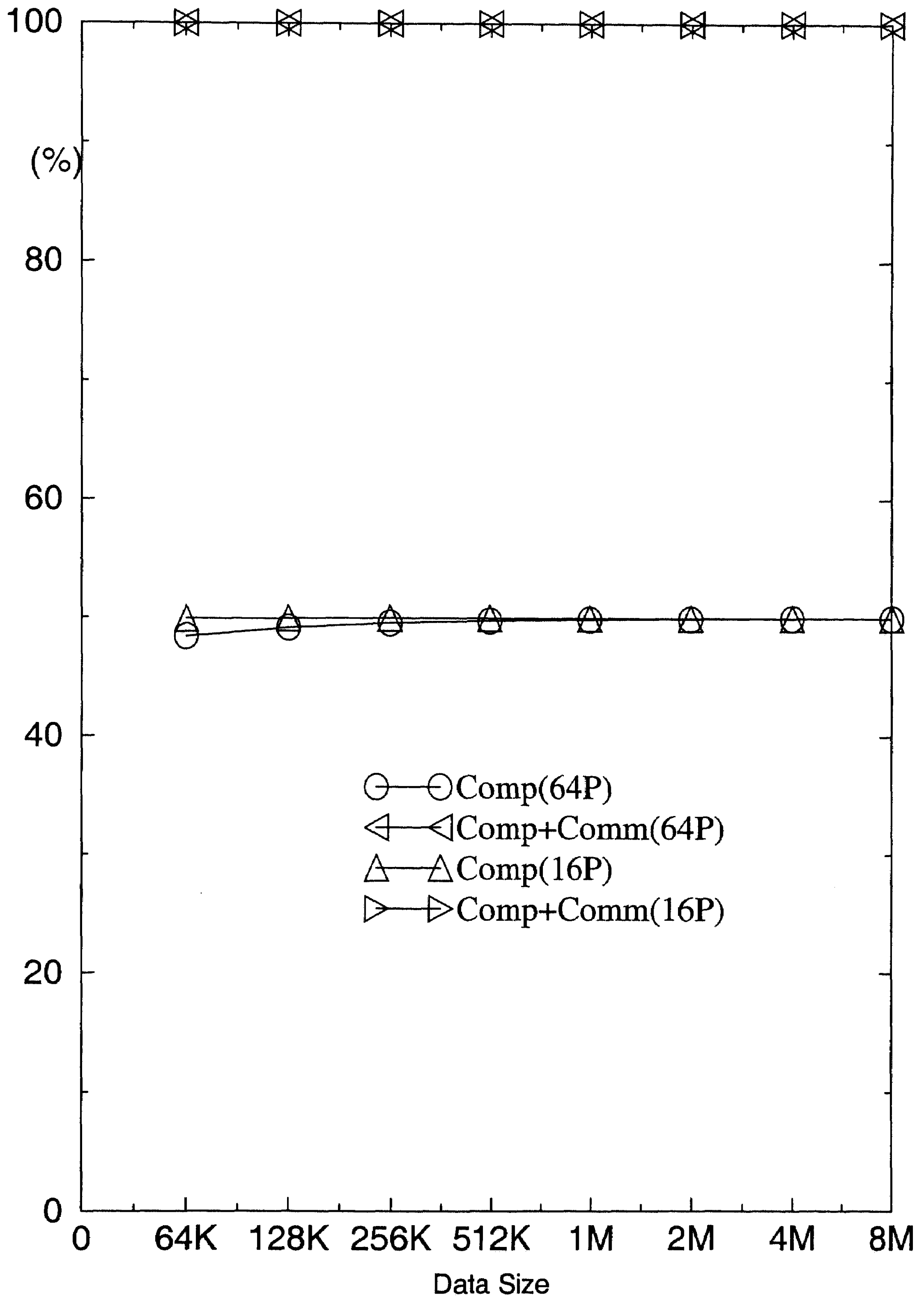


Figure A-10 Distribution of Execution Time on Dot-Product(CC)

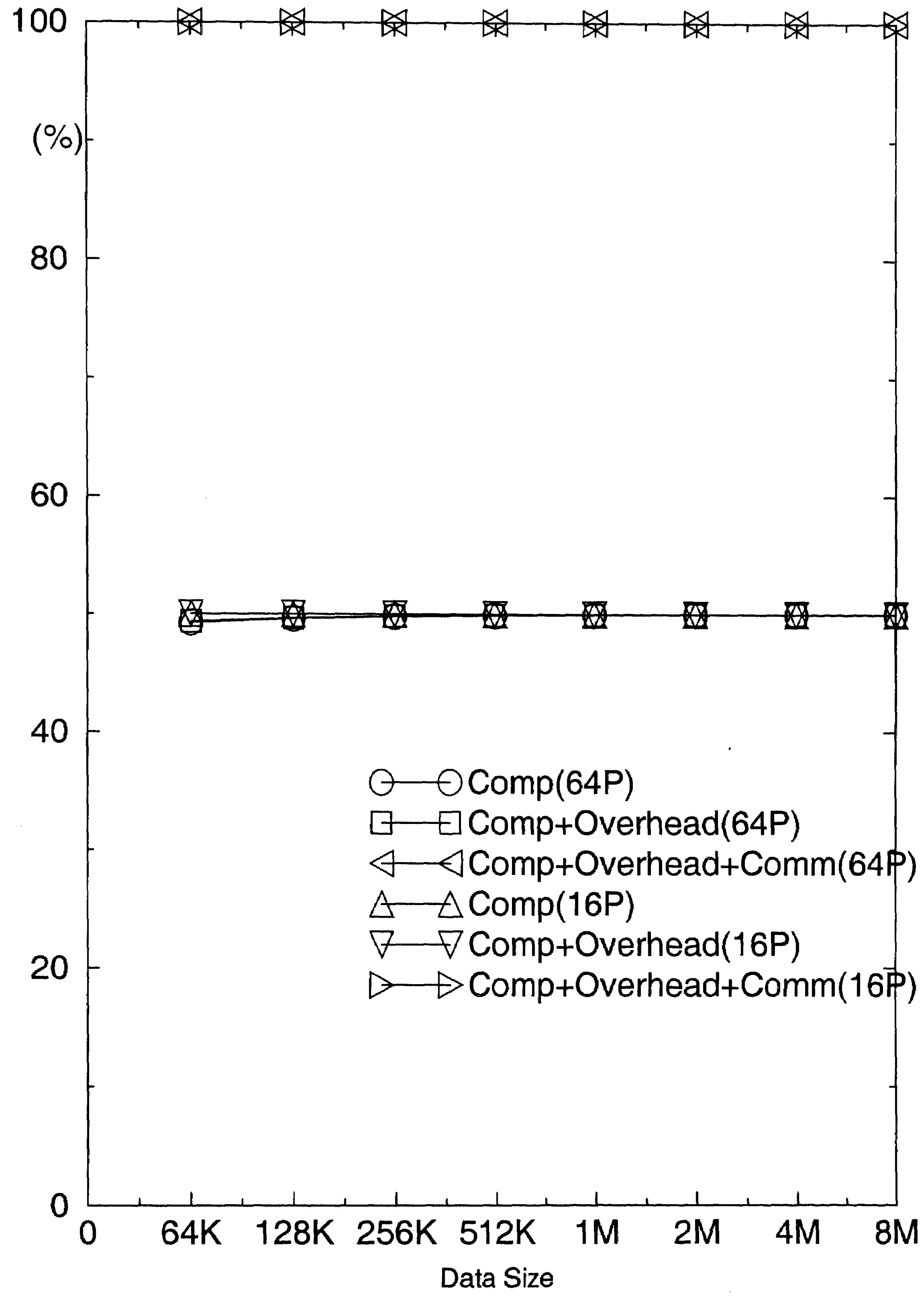


Figure A-11 Distribution of Execution Time on Search(BB)

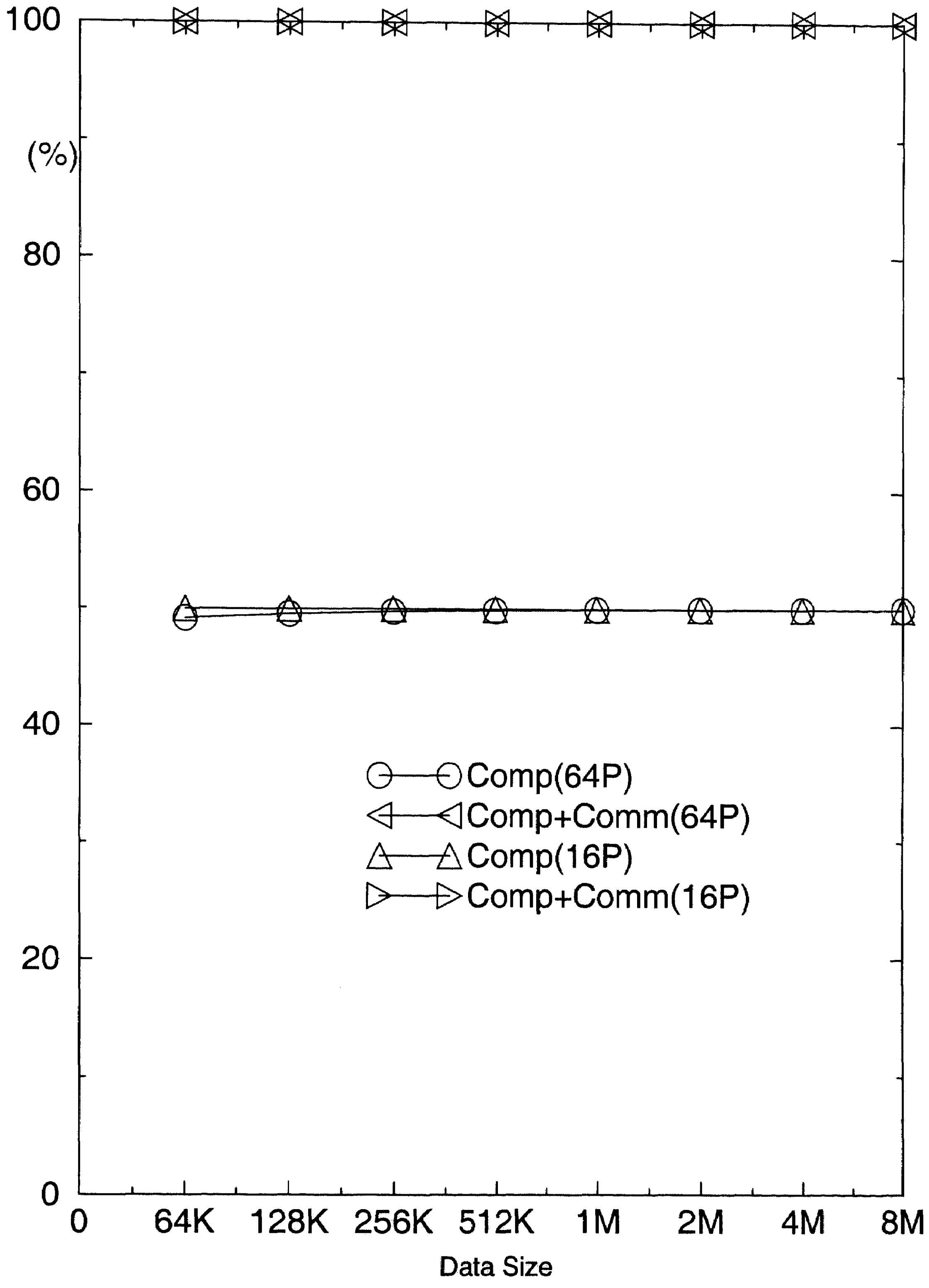


Figure A-12 Distribution of Execution Time on Search(CC)

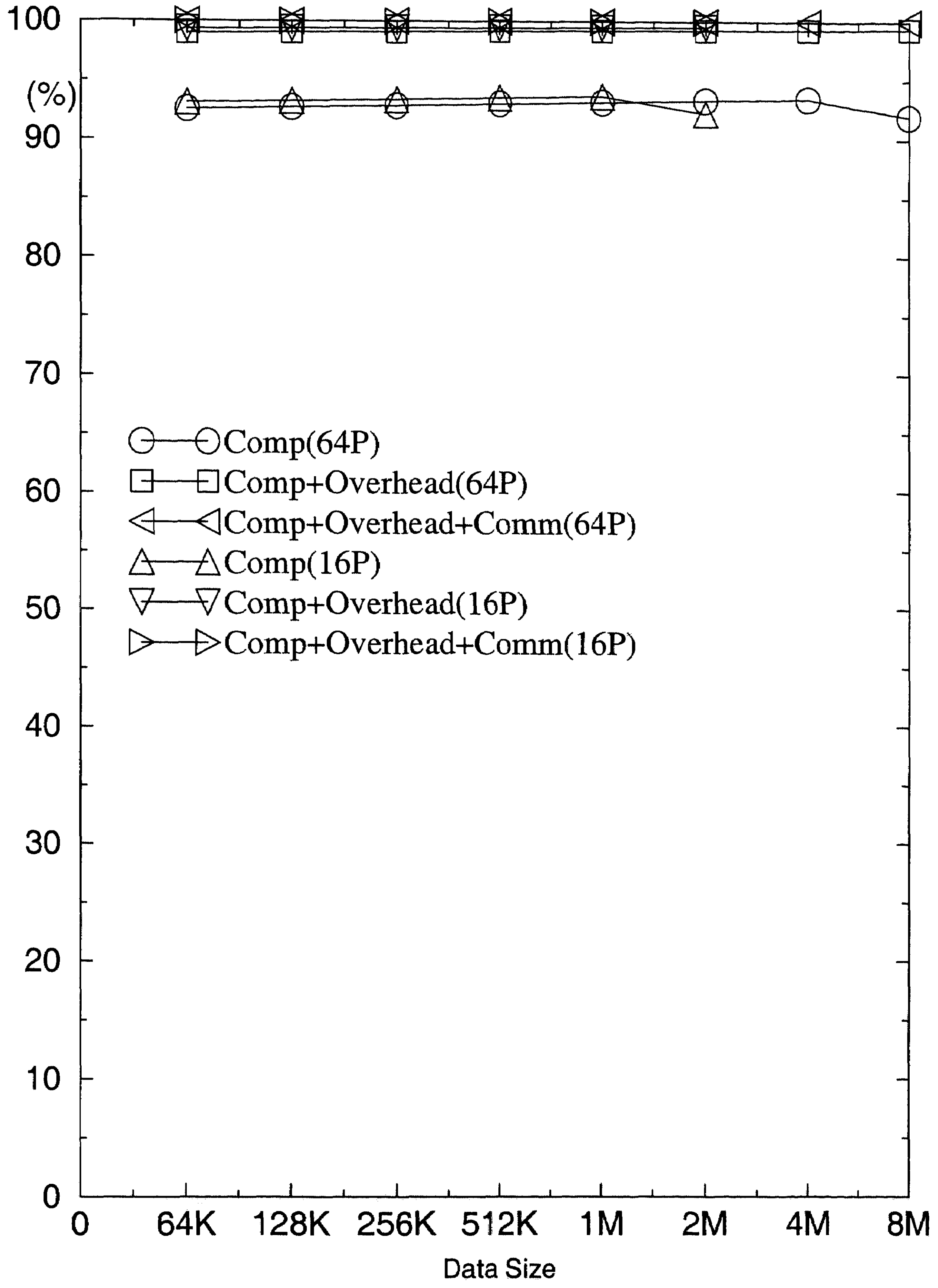


Figure A-13 Distribution of Execution Time on FFT(BB)

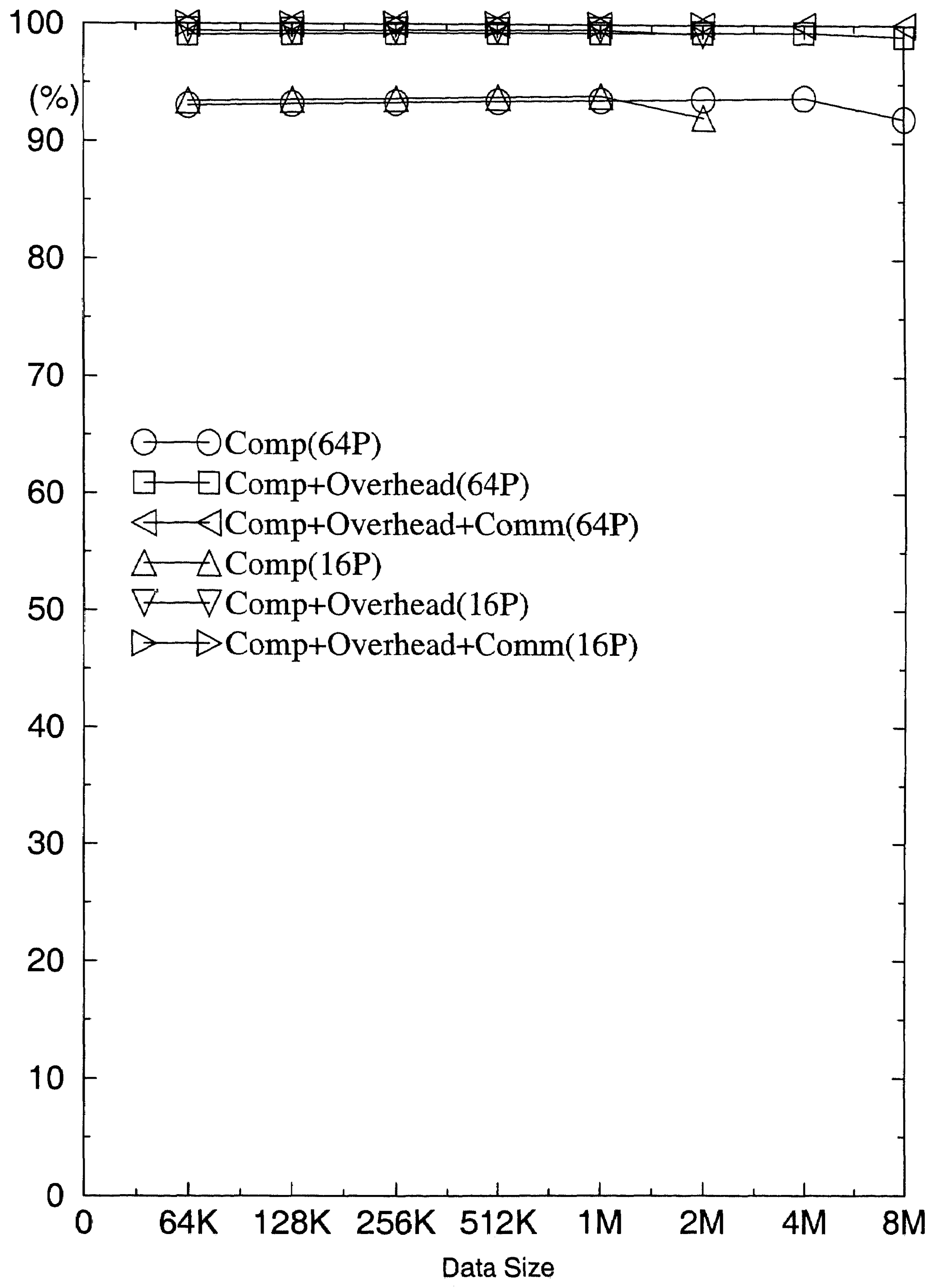


Figure A-14 Distribution of Execution Time on FFT(CC)

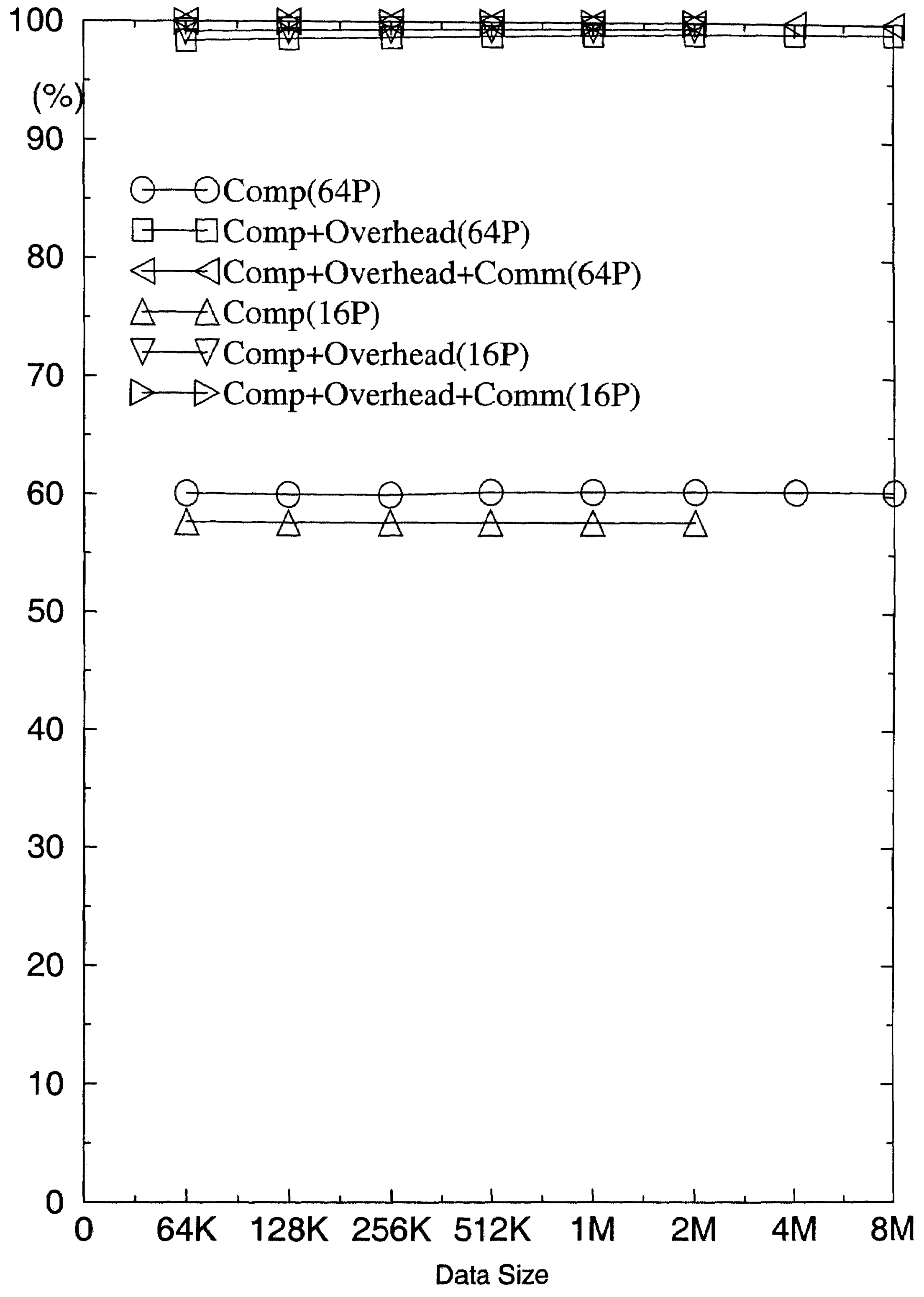


Figure A-15 Distribution of Execution Time on Bitonic Sorting(BB)

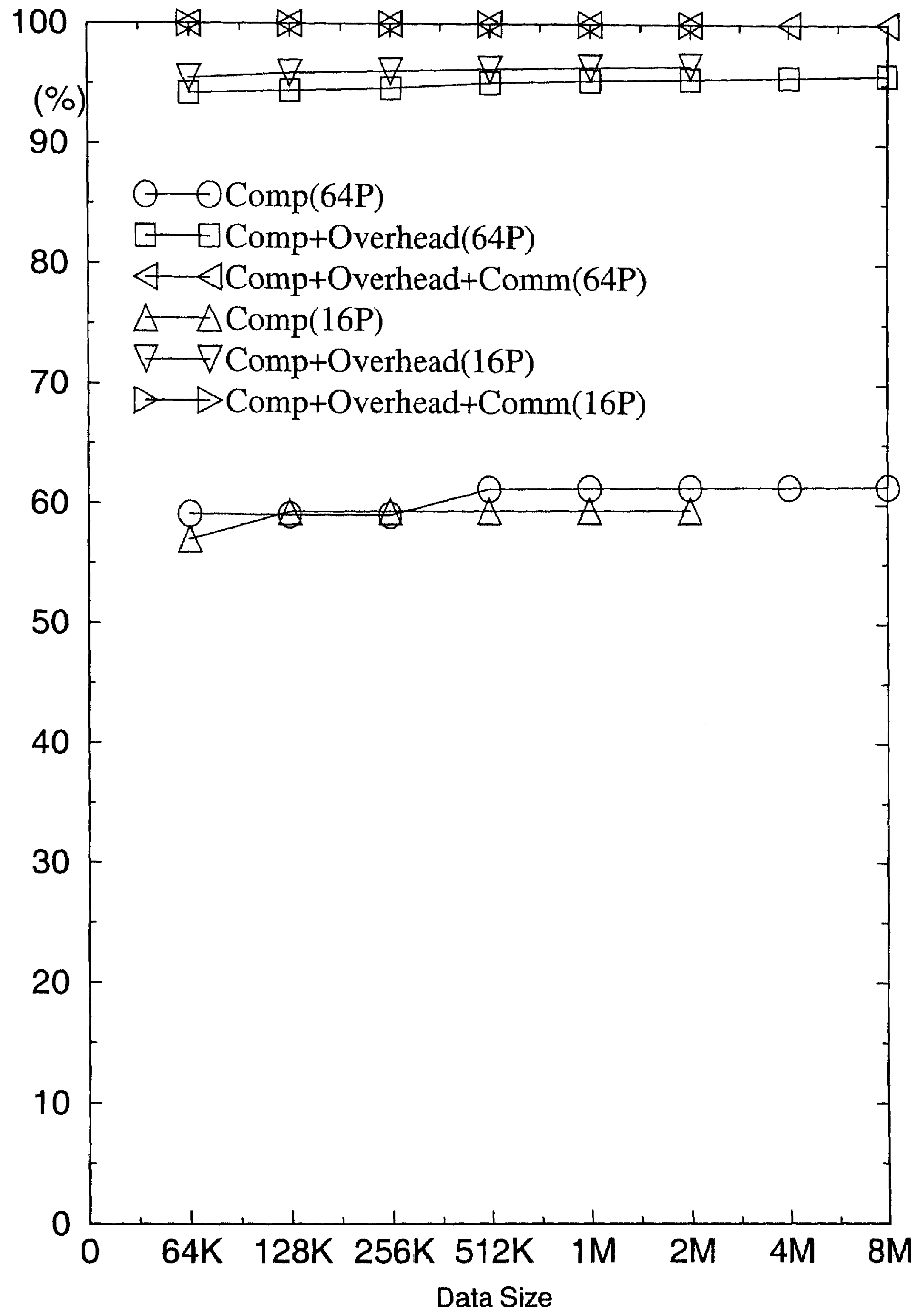


Figure A-16 Distribution of Execution Time on Bitonic Sorting(CC)

REFERENCES

1. J. T. Feo, D. C. Cann, and R. R. Oldehoeft," A Report on Sisal Language Project", *Journal of Parallel and Distributed Computing*, Vol. 10, No. 4, October 1990, pp. 349-366

2. C-C. Lee , S. K. Skedzielewski, and J. T. Feo On the implementation of applicative languages on shared-memory, MIMD multiprocessors. *Proc. Parallel Programming: Environments, Applications, Language, and Systems Conference. IEEE Computer Society*, New Haven, CT, July 1988, pp. 188-197

3. R. R. Oldehoeft, and D. C. Cann Applicative parallelism on a shared-memory multiprocessor. IEEE Software 5, 1(Jan,1988), 62-70

4. C-C. Lee Experience of implementing applicative parallelism on Cray X/MP. *Proc. CONPAR '88. British Computer Society*, Manchester, England, Sept, 1988, pp. 19-25

5. T. Gross, and A. Sussman Mapping a single-assignment language onto the Warp systolic array, In Kahn, G. (Ed.) *Proc. Functional Programming Languages and Computer Architecture. Spring-Verlag*, Portland. OR, 1987, pp. 347-363

6. R. Saavedra-Barrera, D. Culler, and T. von Eicken, Analysis of multithreaded architectures for parallel computing, in *Proceedings of ACM Symposium on Parallel Algorithms and Architectures*, July 1990, pp. 169-178.

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. Abramson, and G. K. Egan An overview of the RMIT/CSIRO parallel system architecture project. Austral. Comput. J. 20,3(Aug. 1988)

9. J. R. Gurd, C. C. Kirkham, and I. Watson The Manchester prototype dataflow computer. Comm. Appl. Math. Comput. 28, 1(Jan. 1985)

10. 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.

11. 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

12. 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

13. T. G. Mattson, D. Scott, and S. Wheat, " A TeraFLOP SuperComputer in 1996 The ASCI TFLOP System," in *Proc. of the 10th IEEE International Parallel Processing Symposium*, Honolulu, HI, April 1996, pp. 84-93

14. 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

15. 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.

16. 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

17. S. S. Pande, D. P. Agrawal , and J. Mauney,"A Fully Automatic Compiler for Distributed Memory Machines," *Proceedings of the 26th Hawaii International Conference on System Sciences*, January 1993.

18. ,S. S. Pande, D. P. Agrawal , and J. Mauney,"Sisal on Distributed Memory Systems," Lawrence Livermore National Laboratory, San Diego, CA, October 1993.

19. S. S. Pande, D. P. Agrawal , and J. Mauney, "A Threshold Scheduling Strategy for Sisal on Distributed Memory Machines," *Journal of Parallel and Distributed Computing*, Vol. 21, No. 2, May 1994, pp. 223-236.

20. H. Matsuoka, K. Okamoto, H. Hirono, M. Sato, T. Yokota, S. Sakai, Pipeline design and enhancement for fast network message handling in RWC-1 multiprocessor, in *Proceedings of the Workshop on Multithreaded Execution, Architecture and Compilation*, Las Vegas, Nevada, February 1998.

21. J. Feo and P. Briggs, Tera Programming Workshop, *IEEE/ACM International Conference on Parallel Architectures and Compilation Techniques*, Paris, France, October 1998.

22. High Performance Fortran Forum, High Performance Fortran Language Specification version 2.0, Center for Research on Parallel Computation, Rice University, Houston, TX, November 1996.

23. K. Kennedy and U. Kremer, Automatic Data Layout for High Performance Fortran, In *Proceedings of Supercomputing'95*, San Diego, CA, December 1995.

24. R. Nikhil, "Id (Version 90.1) Reference Manual," MIT CSG Memo 284-2, July 1991.

25. G. Papadopolous, *An Implementation of General Purpose Dataflow Multiprocessor*, MIT Press, Cambridge, MA, 1991.

26. J. R. McGraw, S. K. Skedzielewski, S. J. Allan, R. R. Oldehoeft, J. Glauert, C. Kirkham, W. Noyce, and R. Thomas, "SISAL: Streams and Iteration in a Single Assignment Language: Reference Manual version 1.2," *Manual* M-146, Rev. 1, Lawrence Livermore Laboratory, Livermore, CA, 1985.

27. D. Cann, The Optimizing SISAL Compiler: Version 12.0, Lawrence Livermore National Laboratory, 1992.

28. G. Gao, L. Bic and J-L. Gaudiot (Eds.) *Advanced topic in dataflow computing and multithreading*, IEEE Computer Society Press, Los Alamitos, CA, 1995

29. R. Iannucci, G. Gao, R. Halstead, and B. Smith (Eds.), *Multithreaded computer architecture*, Kluwer Publishers, Norwell, MA, 1994.

30. L. Roh, W. A. Najjar, B. Shankar, and A. P. W. Bohm, An evaluation of optimized threaded code generation. in *Proceedings of the Conference on Parallel Architectures and Compilation Techniques*, Montreal, Canada, August 1994, pp. 37-46.

31. X. Tang, J. Wang, K. B. Theobald, G. R. Gao, Thread partitioning and scheduling based on cost model, *Proceedings of ACM Symposium on Parallel Algorithms and Architectures*, Newport, Rhode Island, July 1997.

32. 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.

33. G. Gao, L. Bic and J-L. Gaudiot (Eds.) *Advanced Topic in Dataflow Computing and Multithreading*, IEEE Computer Society Press,, Los Alamitos, CA, 1995.

34. 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.

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

36. 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.

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

38. 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.

39. A. Sohn, "Communication Efficient Bitonic Sorting on Distributed-Memory Multiprocessors," *Technical Report*, NJIT CIS Dept., March 1996.

40. A. Sohn, M. Sato, N. Yoo, and J-L. Gaudiot, Data and Workload Distribution in a Multithreaded Architecture, *Journal of Parallel and Distributed Computing*, December 1996.

41. A. Sohn, M. Sato, N. Yoo, and J-L. Gaudiot, Data and workload distribution in a multithreaded architecture, *Journal of Parallel and Distributed Computing*, February 1997, pp.256-264.

42. S. Scott and G. Thorson, "The Cray T3E Network: Adaptive Routing in a High Performance 3D-Torus," HOT Interconnects IV, Stanford University, August 1996.

43. D. C. Cann,"Retire FORTRAN? A debate rekindled", CASM,35(8):81-89, August 1992

44. S. Sakai, Y. Yamaguchi, K. Hiraki, and T. Yuba, An architecture of a data-flow single chip processor, in *Proceedings of ACM International Symposium on Computer Architecture*, Jerusalem, Israel, May 1989, pp.46-53.

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

46. 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, CA, October 1994, pp.328-337.

47. H. Hum, O. Maquelin, K. Theobald, X. Tian, G. Gao, and L. J. Hendren. A study of the EARTH-MANNA multithreaded system. *International Journal of Parallel Programming 24*, August 1996, pp.319-347.

48. J.B. Dennis "Mapping programs for data parallel execution on the Connection Machine", Technical Report Research research Institute for Advanced Computer Science, November 1989.

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

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

51. J. M. Cooley and J. W. Tukey, "An algorithm for the machine calculation of complex Fourier series," *Math. Comp.*, 19: 297-301, 1965.