CEWES Major Shared Resource Center

MSRC

CEWES MSRC/PET TR/99-03

Benchmarking of HPC Systems

by

Dongsoo Kang

Henry W. Park

Jinwoo Suh

Viktor K. Prasanna

Sharad N. Gavali

Nichols

Work funded by the DoD High Performance Computing Modernization Program CEWES Major Shared Resource Center through

Programming Environment and Training (PET)

Supported by Contract Number: DAHC 94-96-C0002
Nichols Research Corporation

Views, opinions, and/or findings contained in this report are those of the author(s) and should not be construed as an official Department of Defense Position, policy, or decision unless so designated by other official documentation.

Final Report

Benchmarking of HPC Systems *

Dongsoo Kang, Henry W. Park, Jinwoo Suh, Viktor K. Prasanna
Department of Electrical Engineering–Systems
University of Southern California
Los Angeles, CA 90089-2562
URL: http://ceng.usc.edu/˜prasanna.html

and

Sharad N. Gavali
High Performance Processing, NAS System Division
NASA Ames Research Center, M/S 258-5
Moffett Field, CA 94035-1000

November 20, 1998

Abstract

High Performance Computing (HPC) platforms have gained widespread acceptance for meeting the computational requirements of large-scale applications. To evaluate the performance of these platforms, reasearchers have proposed various benchmarks. Some benchmarks attempt to measure the peak performance of these platforms. They employ various optimizations and performance tuning to deliver close-to-peak performance. These benchmarks showcase the full capability of the

*Work funded wholly by the DoD High Performance Computing Modernization Program CEWES Major Shared Resource Center through Programming Environment and Training (PET) supported by Contract Number DAHC 94-96-C0002, and Subcontract Number NRC-CR-98-0002.

Disclaimer: Views, opinions, and/or findings contained in this report are those of the authors and should not be construed as an official Department of Defense position, policy, or decision unless so designated by other official documentation.

products. However, for most users, these performance measures seem to be meaningless. For end-users, the actual performance depends on a number of factors including the architecture and the compiler used. Other benchmarks attempt to measure the performance of these platforms with a set of representative algorithms for a particular scientific domain. Although useful, these benchmarks do not give the end-users a simple method for evaluating their algorithms and implementations.

We take a different view of benchmarking. Our benchmarks address the actual performance available to end-users. The benchmarks allow the end-users to understand the machine characteristics, the communication environment, and the compiler features of the underlying HPC platform at a user level. Using the results of our benchmarks, we attain our goal to provide end-users with a simple and accurate model of HPC platforms, including that of the software environment. Using such a model, end-users will be able to analyze and predict the performance of a given algorithm. This will allow algorithm designers to understand tradeoffs and make critical decisions to optimize their code on a given HPC platform.

Our benchmarks provide the data and parameters necessary for the formulation of a model of HPC platforms. In predicting the performance of algorithms on HPC platforms, we assert that the key factor is accurate cost analysis of data access. Data may be communicated between memory and processor, between processors, or between secondary storage and processor. The possible locations of the data can be thought of as a data hierarchy. From our benchmarks, we formulated the Integrated Memory Hierarchy (IMH) model. The IMH model is a simple and accurate model that is able to predict the performance of data communication along the storage hierarchy. To demonstrate the accuracy and usefulness of the IMH model, we have used it to predict the performance of the end-to-end program supplied to us by CEWES. We identified the major bottlenecks of the implementation and, using the IMH model, modified the parallelization of the code to improve scalability. Our implementation has been ported to the IBM SP, SGI/Cray T3E, and Origin 2000.

Contents

1 Introduction **3**

2 HPC Platforms **7**

2.1 Architectural Classification 7

2.2 Example Platforms 9

2.2.1 IBM SP 9

2.2.2 SGI/Cray T3E 10

2.2.3 SGI/Cray Origin 2000 11

3 Overview of Our Approach **13**

4 Low-Level Benchmarks **15**

4.1 Previous Approaches 15

4.1.1 Low-Level Benchmarks 15

4.1.2 High-Level Benchmarks 15

4.2 Our Benchmarks 16

4.2.1 Processor-Memory 17

4.2.2 Processor-Processor 18

4.2.3 Memory-Disk 21

4.3 Implementation Results on HPC Platforms 24

4.3.1 Processor-Memory 25

4.3.2 Processor-Processor 26

4.3.3 Memory-Disk 28

5 Our Preliminary Model of HPC Platforms **31**

5.1 Previous Models 31

5.1.1 Parallel Memory Hierarchy (PMH) Model 31

5.1.2 Two-Level Memory Model 32

5.2 Integrated Memory Hierarchy Model 33

5.2.1 Processor-Memory 33

5.2.2 Processor-Processor 36

5.2.3 Memory-Disk 39

5.2.4 Integrated Memory Hierarchy Model 40

5.3 Significance and Use of Our Model 42

6 An Illustrative Example: Matrix Multiplication **44**

6.1 Previous Algorithm 44
6.2 Our Algorithm 44

7 Parallelizing a Benchmark Application 47
7.1 Overview of the Code 47
7.2 Previous Implementation 49
7.2.1 Workload Distribution for Computation 49
7.2.2 Interprocessor Communication 51
7.3 Our Implementation 51
7.3.1 Load Balancing 52
7.3.2 Parallel Interprocessor Communication 52
7.4 Communication Performance Prediction Using Our Model 57

8 Acknowlegement 58

A Appendix I: Benchmark Codes 62
A.1 Out-of-Cache Memory Communication Code 62
A.2 Permutation Communication Code 67
A.3 Pingpong Communication Code 70
A.4 Scatter Communication Code 73
A.5 Broadcast Communication Code 76
A.6 Disk Operation Code 79

B Appendix II: Detailed Results 81

C Appendix III: Modified Subroutines 108
C.1 OWN_PL Subroutine 108
C.2 UPDADD Subroutine 110
C.3 UPDATE Subroutine 114

1 Introduction

High Performance Computing (HPC) platforms composed of Commercial-Off-The-Shelf (COTS) components have gained wide-spread acceptance for meeting the computational requirements of large-scale applications. Many HPC platforms, such as the IBM SP, SGI/Cray T3E, and Origin 2000, are available to the user community. Although many applications are being written and ported onto these HPC platforms, there has been a lack of simple and useful benchmarks and models to aid in the design of algorithms from an end-user's perspective. A useful model of HPC platforms should allow users to predict the performance of a particular algorithm. This would allow algorithm designers to understand tradeoffs and make critical decisions concerning them. The model should also be useful to algorithm designers in tuning algorithm performance.

Various benchmarks have been proposed previously [2, 3]. However, they tend to fall into one of two categories. In one, the benchmarks are too low level to be useful to the end-user. They attempt to measure the peak performance of a given HPC platform. Often, the manufacturer's benchmarks fall into this category. Through extensive optimizations and performance tuning, they present performance measures that are close to peak performance. These benchmark results are impressive in showcasing the full capability of their products. However, for most users, these performance measures are often meaningless. They do not give a realistic expected performance measurement for the end-user. The actual performance depends on a number of factors, including the architecture and the compiler used. In the second category, the benchmarks are very high level. The NAS Parallel Benchmarks [3] fall into this category. The NAS Parallel Benchmarks are a set of representative algorithms for a particular scientific domain. These benchmarks measure and compare the performance of various HPC platforms. These benchmarks are useful in comparing the performance of a particular algorithm on various platforms. However, the results are very difficult for the end-user to apply directly to their own algorithms and codes. What the end-users need are benchmarks that fall between these two extremes.

We take a different view of benchmarking. Our benchmarks address the actual performance available to end-users. At a user level, the benchmarks allow the end-users to understand the machine characteristics, the communication environment, and the compiler features of the underlying HPC platform. Using the results of our benchmarks, we attain our goal to provide end-users with a simple and accurate model of HPC platforms, including that of the software environment. The model seamlessly incorporates the various hardware features and compiler optimizations. Using the model, end-users can analyze and predict the performance of a given algorithm. This allows the algorithm designer to understand the tradeoffs, make critical decisions to optimize their code on a given HPC platform, and keep the cost of parallelization low. Using the model, the designer can identify the bottlenecks in the code. This allows the designer to tune the performance of the code after an ini-

tial algorithm design decision has been made. By iterating this procedure, the designer can create efficient and scalable algorithms.

Our benchmarks provide the necessary data to design such a model of HPC platforms. In predicting the performance of a particular algorithm design on a given HPC platform, we assert that the key factor is accurate cost analysis of data access. The cost for communication of data is heavily affected by the data location. The data may be physically located in the local memory, in a remote processor, or on secondary storage such as a disk. The various possible data locations can be thought of as a data hierarchy. Thus, data may be communicated between processor and memory, between processors or between secondary storage and the processor. The cost to access data increases dramatically as the data moves down along the hierarchy. Our benchmarks measure the cost of accessing the data along the hierarchy.

Accessing data in the memory has always been an important consideration in designing computer architectures. The use of very fast cache memory is a well known technique that takes advantage of both spatial and temporal data locality. Since the 1980's, the speed of processors has been increasing dramatically. The speed improvement has been estimated between 50 percent to 100 percent per year. However, the speed of memory devices has not enjoyed such phenomenal growth rates, estimated at approximately 7 percent each year [14]. The large disparity in the growth rates of these two key components in computing platforms has led to a widening gap between the performance of the computing elements and the performance of the memory. This widening gap magnifies the importance of data placement during computation. If the algorithm is not designed to supply the processing units with data in a timely fashion, the large GFLOPs touted by the manufacturers become meaningless. Our benchmarks and model of HPC platforms aid the application and algorithm designer in placing the data in the processors to reduce the total aggregate cost of accessing the data from the memory during an application's execution.

In parallelizing large applications on HPC platforms using multiple processors, the cost of data communication between processors is a critical factor that must be accurately predicted. Coarse grain parallelization techniques that map the given data set onto multiple processors on HPC platforms are often deployed to obtain scalable performance for large-scale applications. During the execution of such applications, processors must exchange partial results in order to continue computation. This makes the interconnection network for communication among the processors a critical component. Fortunately, vast improvements in the performance of the interconnection networks have occured in the last few architecture generations. Many improvements come through improved hardware that offer increased speeds. The main bottleneck in interprocessor communication today is the operating system overhead. The actual hardware to move the data among the processors is quite fast. Improvements in processor technology have resulted in significantly reducing the operating system overhead. It is expected that the speed improvement in interconnection networks will out pace

that of memory subsystems. For example, the interconnection network bandwidth on the IBM SP2 improved from 28 Mbytes/sec to 100 Mbytes/sec in the IBM SPSC system. The 23 Mbytes/sec bandwidth on the SGI/Cray T3D improved to 167 Mbytes/sec on the new SGI/Cray T3E system. These speeds closely rival those of memory subsystems. Given that most applications spend a majority of their execution time in intra-processor computation and communication, this is very significant. It is conceivable that one can design an algorithm such that by increasing the amount of inter-processor communication, one can significantly reduce the communication between the memory and processor. Our benchmarks' results and our model of HPC platforms allow the algorithm designer to accurately predict the tradeoffs in increasing processor-processor communication in order to reduce the amount of memory-processor communication.

There are many applications that address very large data sets. These applications occur in diverse areas such as large-scale scientific computations, database applications, multimedia systems, information retrieval and data mining, visualization, among others. Although most HPC platforms have large memories, they are often not large enough to hold the large data sets in memory. Such programs are called Out-of-Core programs. Although HPC platforms such as T3E and SP are already able to provide GFLOPs of computational power, there have been significantly fewer improvements in disk I/O performance. For large scale Out-of-Core applications, the bottleneck is disk access. Our benchmarks and model allow algorithm designers to predict the performance of various access schemes for disk I/O. In conjunction with the predicted cost of inter-processor communication and memory access cost, algorithms that allow pre-fetching of data along the hierarchy can be designed. By tuning the algorithm, one may hide much of the actual cost of moving the data up the hierarchy by overlapping of computation and communication.

Using the results of our benchmarks, we formulated the Integrated Memory Hierarchy Model (IMH Model). The IMH Model is a simple and accurate model that is able to predict the performance of processor-memory, processor-processor, and secondary storage-processor communication. To demonstrate its accuracy and usefulness, we used the IMH model to predict the performance of a kernel operation, matrix multiplication, and an end-to-end benchmark program supplied to us by CEWES. Starting with a simple algorithms, we predicted the performance of these generic algorithms. Using the IMH Model to identify the bottlenecks, we then designed an efficient matrix multiplication algorithm. We accomplished this through several algorithmic techniques. The data access pattern was modified through data reorganization and data placement techniques. This reduced the cost of accessing data along the data hierarchy. Efficient schedules and data prefetching allowed overlapping of computation and communication. Efficient mapping and load balancing of processing elements yielded high utilization of all available processing node. The IMH Model allowed us to predict the performance of various improvements before actual coding. This greatly augmented the process of designing efficient and scalable algorithms. For the end-to-end program supplied to us

by CEWES, we identified the implementation bottleneck sections. Using the IMH model, we modified the code to improve scalability through load balancing and improved parallel interprocessor communication.

The rest of the report is organized as follows. Section 2 introduces the architectural characteristics of HPC platforms in general and the IBM SP, the SGI/Cray T3E, and the Origin 2000 in particular. Section 3 gives an overview of our approach to benchmarking HPC systems. Section 4 introduces our benchmarks. We describe previous benchmarks for comparison. Section 5 defines the Integrated Memory Hierarchy (IMH) Model. Section 6 presents the results of implementing our matrix multiplication algorithm using the IMH Model. Section 7 describes the improvements made to FT.f (an end-to-end application code supplied to us by CEWES) using our IMH Model and methodology. Section 8 concludes the report. We have included the codes used to measure the various benchmarks in Appendix I and the detailed results of our benchmarks in Appendix II. Appendix III includes the modified portions of the FT.f code.

2 HPC Platforms

HPC platforms are typically composed of Commercial-Off-The-Shelf (COTS) components. COTS allows flexibility in the design of the architecture and allows architecture designers to rapidly incorporate the latest trends and novel design techniques. To obtain high overall performance, many manufactures have targeted the interconnection network, which is a bottleneck. In the past few years, many HPC platforms have vastly improved the performance of interconnection networks.

There has been a large disparity in the growth of computational components as compared with memory components. We feel that the memory structure is the next major bottleneck to obtaining high performance on these HPC platforms. Indeed, most HPC platform architecture designers have emphasized improving main memory peformance. Both hardware and software support have been considered. Some efforts take advantage of the large improvements in semiconductor design and integration, integrating the memory into the chip [10, 18, 22, 25]. Cache control support circuitry and optimized compilers allow the user to control the use of cache memory explicitly [17]. These efforts to improve the performance of communication between processor and memory acknowledge that most current computational time is spent in processor-memory communication.

2.1 Architectural Classification

From an architectural and end-user's view, the HPC platforms can be broadly classified into shared memory and message passing machines. See Figure 1 for a comparison.

Currently, the dominant programming style for scientific computing and signal processing is message passing. In this approach, the address space is local to each node, and accesses to remote memory locations are through *explicit* message passing. The message passing style offers three key advantages:

1. Explicit user level control of communication
2. Predictable performance due to controlled interprocessor communication
3. Systems that are relatively easy to build and that are compact

On the other hand, the market for message passing systems is relatively small. Various vendors of large commercial market systems offer high performance *shared* memory systems as an alternative. Such systems include database servers and various other forms of servers, transaction processing systems, data warehouses, web-based applications, etc. These offer the end user, a single system view of the available multiple nodes. Such systems offer three key advantages:

1. Reduced overall cost due to proliferation of such systems in the commercial market
2. A single system view that facilitates porting of legacy code

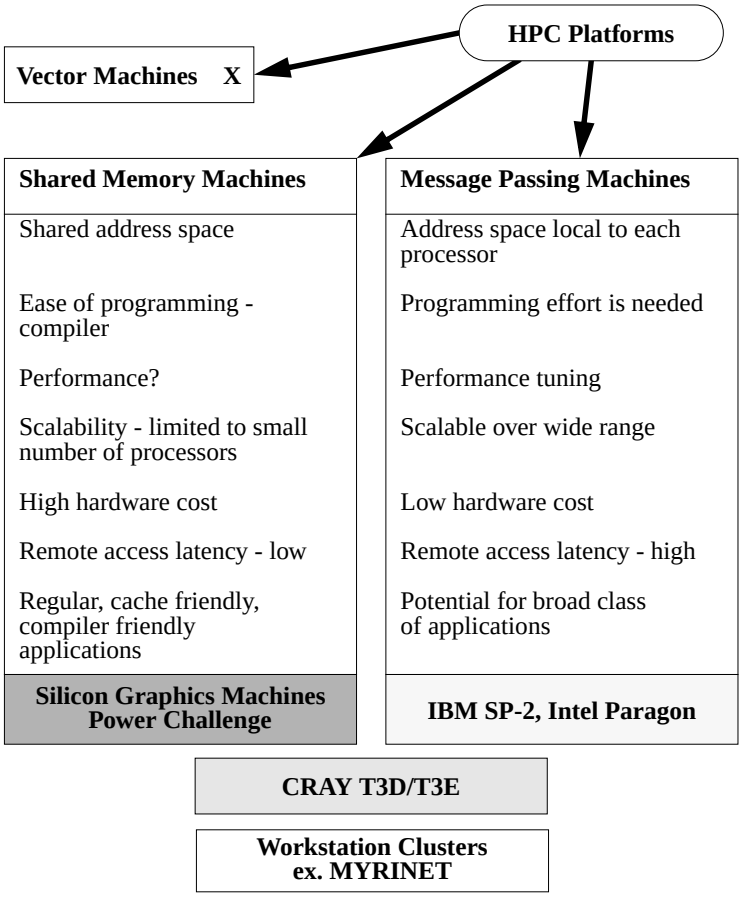


Figure 1: Comparison of shared memory and message passing machines.

3. Reduced remote memory access latencies leading to improved efficiency

The memory modules of shared memory systems can be either physically centralized (providing uniform access latencies to all the processors) or distributed over a number of processor nodes. In distributed shared memory systems, the access latency of remote memory modules is greater than that of the local memory modules. The key hardware component of a distributed shared memory system is an integrated memory system that provides access to local and remote memory. Such a memory system results in faster access to remote memory compared with message passing systems.

We now describe three example platforms: IBM SP, SGI/Cray T3E, and SGI Origin 2000.

2.2 Example Platforms

2.2.1 IBM SP

The most recent IBM SP platform uses a Power 2 Super Chip (P2SC) microprocessor. The chip consists of an Instruction Cache Unit (ICU), a Data Cache Unit (DCU), a Dual Floating Point Units (FPUs), a Dual Fixed Point Units (FXUs), and a Storage Control Unit. Since there are six processing units, the P2SC can issue up to six instructions in each clock cycle.

There are 54 physical registers. The number of registers defined by the architecture is only 32. The extra registers (54-32=22 registers) are used for register renaming which reduces the communication between the processor-cache.

Each FPU has a floating-point execution unit. The FPU can perform multiply-add instructions. Thus, each FPU can initiate two operations (multiply and add) each cycle. Since the clock speed of the SP installed at CEWES is 135 MHz, the peak MFLOPs of the SP at CEWES is 135 MHz $\times$ 2 (operations) $\times 2$ (units) = 540 MFLOPS.

There are two types of FXUs. Both FXUs can issue add and logic instructions concurrently. However, only one of the FXUs contain a multiply/division logic. Therefore, only one of the two FXU units is able to issue multiply or division instructions.

The size of the ICU is 32KB and the DCU is 64 to 256 KB. The DCU uses four-way set associative dual ported caches. It can be configured either as a 256KB with na 8-word memory bus or 128KB with a 4-word memory bus. There is no secondary cache.

The peak memory bandwidth is 2.1 GB/sec. However, the manufacturer-supplied sustained memory bandwidth is 910 MB/sec. The data access times for different hierarchies are as follows:

- Register: 0 clocks
- Cache: 1 clock
- Cache miss: 18 clocks
- TLB miss: 36-56 clocks
- Page Fault: $>$ 100,000 clocks

The nodes of SP are interconnected by a bidirectional Multistage Interconnection Network (MIN). Since the communication between any pair of processors needs the same number of hops, the distances between any pair of processors is the same. Further, the network allows any-to-any interconnection. The network is a packet-switched network. A peak bi-directional bandwidth of up to 150MB/sec is supported on the newer machines. The message latency without software overhead is 500 nsec to 875 nsec. However, the latency that is observed at the user level is much higher (tens of μsec).

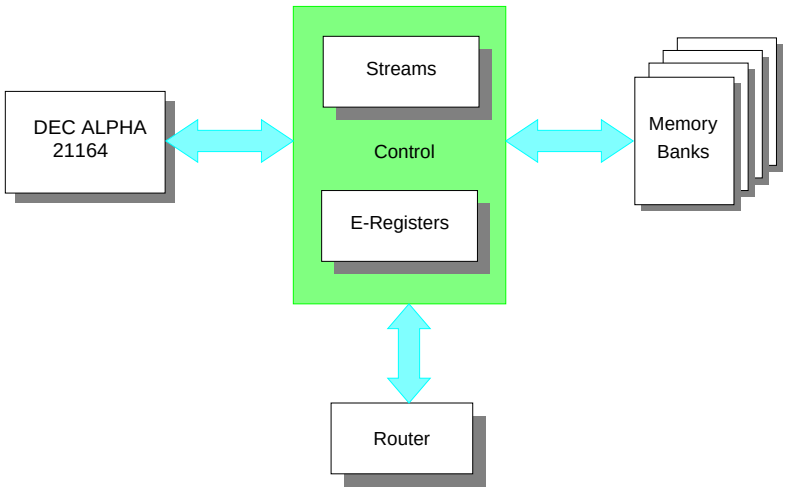


Figure 2: Block diagram of SGI/Cray T3E node

2.2.2 SGI/Cray T3E

The SGI/Cray T3E series is the second generation of scalable parallel processing systems from Cray Research built around COTS processors. The SGI/Cray T3E series is based on the DEC Alpha 21164 microprocessors with clock speeds ranging from 300 MHz to 600 MHz. The system logic runs at 75 MHz. The interconnection network is a 3-D torus that provides scalability of the system. Up to 2048 nodes can be configured in a single system. Each node of a SGI/Cray T3E system contains a processor, support circuitry, local memory, and a network router. A block diagram of a SGI/Cray T3E node is shown is Figure 2.

The DEC Alpha 21164 microprocessor can issue up to four instructions per clock period. The four concurrent instruction pipelines consist of:

- **FA**: floating-point add pipeline
- **FM**: floating-point multiply pipeline
- **E0**: first integer pipeline, also executes loads and stores
- **E1**: second integer pipeline, also executes loads

The two floating point pipelines allow a peak performance of 600 MFlops/sec to 1200 MFlops/sec, depending on the speed of the microprocessor.

Each processor contains an 8 KB direct-mapped instruction cache (Icache), an 8 KB direct-mapped data cache (Dcache), and a 96 KB, 3 way set associative, write-back, write-allocate secondary cache

(Scache). The Dcache cache line is 32 bytes while the Scache cache line is 64 bytes. The local memory is controlled by a set of four memory controller chips, directly controlling eight physical banks of DRAM. Each bank is organized into 64-bit words. The 64 bytes of Scache cacheline is spread across the eight banks with each bank containing 8 bytes. There is a 32-bit path on each of the channels between the memory controller and memory banks. This allows a theoretical maximum of 4 channels $\times$ 32 bits per channel $\times$ 75 MHz $=$ 1.2 GBytes/sec possible bandwidth. The sustainable maximum is 80% of peak or 960 MBytes/sec. The memory bandwidth is enhanced by six *stream buffers*. Each stream buffer can store up to two 64-byte Scache lines. These buffers automatically detect consecutive references to memory locations and prefetch data.

The SGI/Cray T3E processing elements (PE) are connected by a high-speed, low-latency 3-D torus interconnection network. It is capable of peak interprocessor transfer rates of 500 MBytes/sec in each direction and up to to 122 GBytes/sec of payload bisection bandwidth. The network operates asynchronously and independently of the PEs. Each node contains a network router, which is a crossbar switch connecting a PE port, an I/O port, and six network channels (one for each dimension and direction). Thus, the network router can operate bidirectionally in each dimension and can handle data transfers of up to 3.6 GBytes/sec. Routing can be both deterministic ordered routing or adaptive routing to avoid hot-spots and network contentions.

The I/O subsystem consists of a number of input/output nodes connnected by the high-speed *GigaRing I/O channel*. The GigaRing channel is a dual-ring design with data in the two rings traveling in opposite directions. This delivers high I/O data bandwidth, enhances reliability, and allows communication to occur along the shortest path. The I/O channels are integrated into the 3-D torus network, giving a single system image of I/O services. The I/O subsystem is able to scale with demand. A maximum of 16 GigaRing channels are available on air-cooled systems while up to 128 GigaRing channels are available for liquid-cooled systems. Each channel has a full duplex data bandwidth of up to 500 MBytes/sec to each T3E system interface.

2.2.3 SGI/Cray Origin 2000

The SGI/Cray Origin 2000 is based on the R10000 microprocessor from MIPS Technologies. Currently, 180 MHz and 195 MHz systems are available. The R10000 processor is a 4-way superscalar architecture. The microprocessor contains two primary floating point units (adder and multiplier). Both addition and muliplication require two clock cycles but can be pipelined for a 1-cycle repeat rate. In addition, there are two secondary floating point units for divide and square root. These secondary units are not pipelined. Therefore, the peak MFLOPs is twice that of the clock speed.

The Origin 2000 is a follow-up to the previous Challenge-class systems, that attempt to address the scalability isssue. Instead of a bus architecture as in the Challenge series, the Origin 2000 uses

crossbar switches to connect the nodes. This allows multiple data paths and increases the system's scalability.

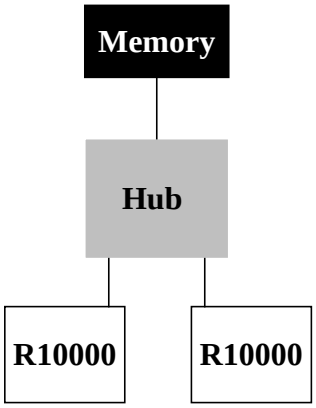


Figure 3: Block diagram of SGI/Cray Origin 2000 node

Each node in a Origin 2000 system consists of one or two processors, local memory, and a hub. Figure 3 shows a block diagram of a single Origin 2000 node. The system is scaled by combining these nodes into a multiple node system using routers to connect the nodes. An example Origin 2000 system with 16 nodes is shown in Figure 4. The routers are connected in a binary n-cube, or a hypercube network. Each router has six ports for interconnection networking. This allows up to 128 nodes to be configured in a single system using the hypercube network. Memory is organized using the SGI/Cray cache coherent non-uniform memory access (ccNUMA) architecture. All the memory in the Origin 2000 is organized into a single global address space. Thus, memory is shared among all processing elements. Data is accessed through the hub and the routers. The access time to memory is not uniform. It varies, depending on how far away the data is from the node accessing the data.

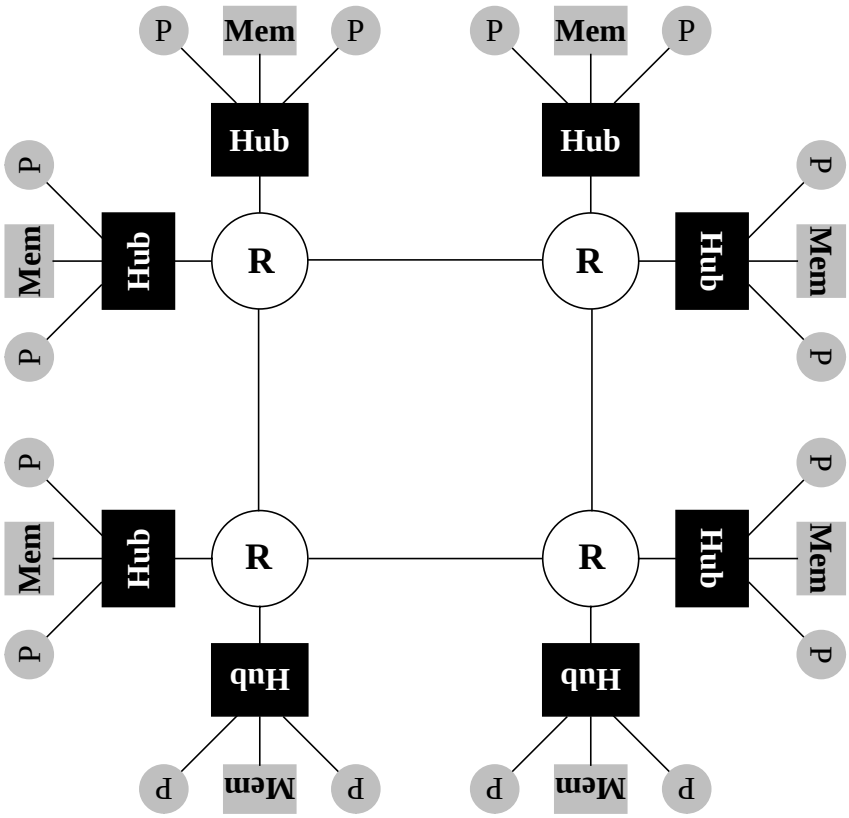


Figure 4: Block diagram of SGI/Cray Origin 2000 System

3 Overview of Our Approach

Current state-of-the-art HPC platforms are largely dominated by Message Passing Systems such as the IBM SP and Shared Memory Systems such as the SGI/Cray Origin 2000. Some hybrid systems such as the SGI/Cray T3E attempt to provide features of both types of architectures by including message passing capabilities and a globally shared address space. In using these HPC platforms, there are layers of interface to the actual hardware. These include the operating system, compilers, library codes for computation and communication such as ScaLAPACK [30], PBLAS [29], MPI [27], etc., and other support utilities. It is not possible for the end-user to modify the structure of the underlying hardware and software architecture such as the cache policy or the routing algorithm of the interconnection network. Our objective is to benchmark and model the above HPC platforms from a end-user's perspective. Our efforts will help users understand machine characteristics, the communication environment, and the compiler features.

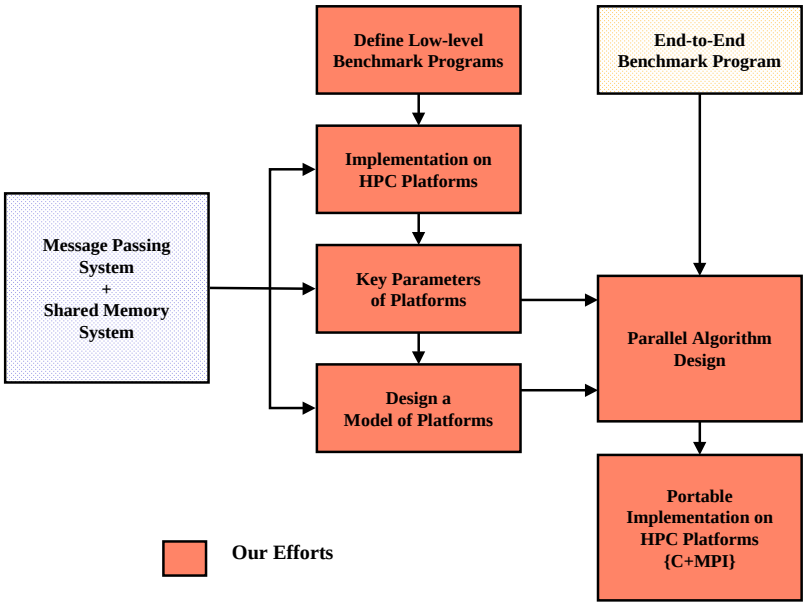


Figure 5: Overview of our approach

Figure 5 shows a high level view of our approach. We first define a set of benchmarks to measure the key parameters of HPC platforms. The architectural features, communication environment, and compiler features are encompassed into these parameters. The parameters should be simple yet accurate in defining and modeling the given HPC platform. We then implement the benchmarks on several state-of-the-art HPC platforms that include both message passing and shared memory architectures. Using the results of the benchmarks, and the key parameters obtained, we define a model of HPC platforms. The model will aid end-users in predicting and analyzing the performance of their algorithms. The algorithm designer will be able to analyze tradeoffs and make decisions for optimizing the algorithm on a given HPC platform, while minimizing parallelization costs. To verify the correctness and usefulness of our model, we analyse and predict the performance of several parallel algorithms, including an end-to-end program supplied from CEWES. Using our model, we enhance the performance of the given algorithm by predicting, identifying, and improving the bottleneck sections of the given code. Finally, we implement our improved algorithm using C and MPI. By measuring the actual performance of the implemented code, we verify the success of our model and methodology.

4 Low-Level Benchmarks

In this section, we first introduce previous benchmarks. Then, we present our benchmarks and their implementation results. We performed experiments on the IBM SP, SGI/Cray T3E, and SGI Origin 2000 systems.

4.1 Previous Approaches

Previous approaches to benchmarking can be classified as low-level and high-level benchmarks. They are briefly described.

4.1.1 Low-Level Benchmarks

Low level benchmark results are usually supplied by hardware manufacturers. Since they fully understand the architecture of the platform and behavior of the compiler, it is possible for them to obtain maximum optimization and fine performance tuning. Also, it is possible to execute programs under controlled environments. In these environments, one can avoid any potential actions that can aggravate the performance. Thus, the performance metrics obtained in these environments are close to the peak performance.

Supplied by the manufacturers, these metrics are not that useful to end-users. For most users, it is unlikely that their applications will run in the ideal environment in which low-level benchmark results were obtained. Also, usually the users do not have an in-depth understanding of all the details of the platforms and compilers that can be used to optimize application code. Moreover, in many real-life applications, the instruction mix makes it impossible to obtain high utilization of the available hardware resources. Therefore, the sustained performance of a platform is much lower than the performance metrics manufacturers provide. Thus, a machine with better performance with respect to these low-level metrics does not necessarily show better performance in real-life applications.

4.1.2 High-Level Benchmarks

4.1.2.1 Synthetic Benchmarks Another approach to the measurement of the computing performance is synthetic benchmarks. These include Whetstone and Dhrystone benchmarks [7, 35]. In these benchmarks, many computation modules are included based on the frequency of each module in sampled applications to represent real applications. However, since these benchmarks are synthetic, in many aspects, there are differences between them and the real applications, such as instruction mix, instruction sequence, and data access patterns. Thus, the performance obtained using these benchmarks does not always represent performance in real applications.

Another drawback of this approach is that compilers can easily optimize these benchmarks. For

example, some compilers remove unnecessary program portions if the calculated result of the portion is not used as input for another operation. Since outputs from many portions of the synthetic benchmarks are not used, these portions are not compiled and, thus, these portions do not contribute to the overall execution time. Therefore, these benchmarks can obtain much higher performance results than the actual performance.

4.1.2.2 Kernel Benchmarks and Compact Application Benchmarks To avoid the problems of synthetic benchmarks, some kernel benchmarks have been proposed. Many benchmarks fall in this category. Some examples are the Linpack benchmark [9], the Livermore benchmark [21], and part of the MITRE benchmark [13]. However, these kernel benchmarks often overstate the performance of the real applications [26].

To obtain results even closer to real applications, compact application benchmarks have also been proposed. In these, small real applications are used. These include SPEC [32] and NAS benchmarks [3]. These benchmarks may provide some useful information to assess platforms for similar application areas.

However, from the a user's perspective, these benchmark results can only be used to determine the relative speed of machines. Unless the same program is used, it is very difficult to use these results to predict user's code execution time. For design and analysis of algorithms, users need metrics that can be used for performance prediction.

4.2 Our Benchmarks

To address the lack of useful metrics to predict and analyze performance of algorithms, we define low-level benchmarks that can be used to predict performance and execution times of an algorithm and the program code.

In achieving high efficiency and scalable performance on HPC platforms, there are two key challenges. Most applications consist of interleaved sections of computation and data I/O or data communication. The performance of the computation section largely depends on the underlying hardware platform and the algorithm design. Efficient algorithms reduce the complexity of the computation. This reduces the total amount of execution time needed to complete the desired operation. The performance of data communication depends on the speed and architecture of the memory subsystem and the data access pattern. Efficient placement of data in the memory hierarchy can significantly reduce data access time. Given these characteristics, a logical set of benchmarks for measuring the performance of HPC platforms includes the computation performance of the functional units and the communication performance of the memory hierarchy.

We first identify three main costs in HPC computing environments: processor-memory, processor-processor, and processor-disk data movement costs. The processor-processor and processor-disk

costs involve only communication cost. However, the processor-memory cost consists of computation cost and communication cost between processor and memory. Our benchmark measures these three main costs.

4.2.1 Processor-Memory

In measuring the performance of the processor-memory hierarchy, we note two distinct performance categories. These are out-of-cache performance and in-cache performance. For out-of-cache operations, the dominating cost is moving data from memory to cache. The cost of the actual operation performed, after the data is placed in cache memory, is relatively small and negligible compared with the data transfer cost. For in-cache operations, data access time is very small. In this situation, the cost of the actual operation performed can be significant. For correct measurement in both situations, we defined and measured two distinct benchmark categories.

To measure the performance of out-of-cache operations, we first identified two key parameters that affect performance of the memory to the cache bandwidth. In the simplest scenario, data is brought into cache in stride 1. In this case, the number of cache lines brought into the cache is proportional only to the data size. However, often data is accessed in a stride other than 1. When the same number of elements are accessed in a stride other than 1, the total number of cache lines brought into the cache increases in proportion to the stride. Therefore, the size of the data and the stride in which the data is accessed are the two key parameters that affect the performance of the memory to the cache data transfer bandwidth.

Figure 6 shows the pseudo-code of our benchmark program. For each data set with N elements, the performance is measured for accessing the data from $stride = 1$ to $stride = S$. During the measurement of each stride, an $N \times S$ sized array is allocated. An $N \times S$ sized array is necessary to access N elements in stride S. First, the cache is flushed with dummy data. This assures that when the actual data is accessed, each new cache line brought into the cache creates a cache miss. Data is then accessed using various strides. In between the measurement for each stride, the cache is once again flushed to assure accurate measurement. This procedure repeats for various data sizes.

The type of operation issued affects the performance of in-cache operations. The cache memory-to-processor bandwidth is very high and therefore does not dominate the cost of the actual operation performed. Stride is also irrelevant for in-cache operations, since all the data is already in cache. Other important factors are the size of the data and the operation performed.

Figure 7 shows the pseudo-code for our benchmark program to measure the performance of in-cache operations. Data is first brought into the cache from memory. Then, for i iterations, the N elements are accessed and the candidate operation is performed. By repeating the loop for i iterations, the overhead cost to measure the time is amortized over the i iterations. By setting i to be very

```
for each data set of N elements
   for each stride S
      Allocate N*S memory space for data;
      Initialize data for given stride S;
      Flush cache with dummy data;
      Start timer;
      for each N elements
         Perform operation to be measured;
      End timer
      Print out the measured time for operation;
```

Figure 6: Pseudo-code for out-of-cache processor-memory communication

large, the overhead cost is made insignificant.

4.2.2 Processor-Processor

Most previous benchmarks for processor-processor communication measured the pingpong communication between two processors. However, in large-scale algorithms, many other types of communication pattern occur frequently. Thus, the performance of the pingpong operation may not be sufficient for one's understanding the overall performance of communication operations. Therefore, it is necessary to measure the communication performance of other basic communication primitives. MPBench [23] includes a set of communication primitives. Our processor-processor communication benchmark suites are based on the MPBench. However, we modified the benchmark as follows.

In our benchmark, we added permutation communication. Permutation communication is frequently used to achieve maximum utilization of the available bandwidth. We found that the communication parameters such as startup cost and bandwidth can be estimated more accurately from the permutation communication results, compared with the pingpong communication results.

Also, we extended the MPBench to include more processors. We perform pingpong communication on 8 and 16 processors because, in many cases, the pingpong operation is performed on pairs of processors in HPC platforms. A detailed explanation is given in each benchmark section.

Thus, our benchmarks measure communication performance in user aspects. The benchmarks also include every possible software overhead that occurs in user application. Therefore, our benchmark results are useful to the end-users.

Our benchmark suite consists of four communication operations:

- Permutation

```
for each data set of N elements
   Allocate memory
   Initialize data
   Flush cache with data
   Start timer
   for i number of iterations
      for each N elements
         Perform operation to be measured
   End timer
   Print out the measured time for operation
```

Figure 7: Pseudo-code for in-cache processor-memory communication

- Pingpong
- Scatter
- Broadcast

4.2.2.1 Permutation Communication In the permutation communication operation, a set of processors is involved in communication. Each processor in the set sends data to a destination processor and receives data from a source processor. The permutation communication occurs in many parallel communications. For example, an all-to-all communication algorithm consists of a number of steps of permutation communication. Also, it can be considered as a general communication pattern on HPC communications because many other communications can be implemented using the permutation communication. Thus, the permutation communication is one of the most important communication patterns on HPC platforms. Therefore, the permutation communication is included in our benchmark suites.

In Figure 8, the code for performing a permutation is shown. Each processor first issues a non-blocking receive command which lets processors proceed without waiting for an incoming message. Then, each processor issues a send command (see Figure 8) which lets each processor start sending data. When the data arrives at a destination processor, the processor can receive data due to the non-blocking receive command issued before. Thus, the processors communicate altogether. In our experiments, we measured the total communication time as a function of the message size and the difference in the processors' ID.

```
for (dist = 1; dist < Number_of_processors/2; dist *= 2)
    for (msg_size=1; msg_size <16 Mbytes; msg_size *=2)
        Start timer;
        Post Non-blocking receive from processor
              ((my_id + dist) MOD Number_of_processor);
        Blocking send to processor
              ((my_id - dist) MOD Number_of_processor);
        End timer;
```

Figure 8: Pseudo-code for permutation communication

```
for (src = 0; src < Number_of_processors; src++)
for (dst = src+1; dst < Number_of_processors; dst++)
for (msg_size=1; msg_size <16 Mbytes; msg_size *=2)
    Start timer;
    Processor (src) sends data to Processor (dst);
    Processor (dst) receives data from Processor (src);
    Processor (dst) sends the received data to Processor (src);
    Processor (src) receives data from the Processor (dst);
    End timer;
```

Figure 9: Pseudo-code for pingpong communication

4.2.2.2 Pingpong Communication In pingpong operation, the total time for a message to travel to another processor and return to the original processor is measured. Since this time has been used to assess many other platforms, this measurement gives a good metric to compare the targeted platforms with previous HPC platforms.

In Figure 9, the code for measuring pingpong communication time is shown. In each iteration, a couple of processors are paired and pingpong communication is performed. The first processor (source) sends data to the second processor (destination). The destination processor waits for the message. When it arrives, the destination processor returns the message to the source processor. To avoid confusion, we report the total pingpong time rather than the half pingpong time. In our experiments, we measured the pingpong communication time as a function of the message size communicated between each pair.

```
for (src = 0; src < Number_of_processors; src++)
for (msg_size=1; msg_size <16 Mbytes; msg_size *=2)
    Start timer;
    if (my_id == src)
        send message to other processors;
    else
        receive message from the source processor;
    End timer;
```

Figure 10: Pseudo-code for scatter communication

4.2.2.3 Scatter Communication In scatter operation, there are one source processor and N destination processors. The source processor has N data blocks, D_0, D_1, ... D_{N-1}. A data block, D_i is sent to i-th destination processor during the scatter communication. This is also a frequently used communication primitive. For example, when a root processor has data and needs to distribute the data to other processors to improve parallelism, this operation is used. Also, it is the reverse of the gather communication in which N processors send data to a root processor.

In Figure 10, the code for scattering communication is shown. In our experiments, we measured the total communication time as a function of the message size. Also, we measured the communication time for each source processor.

4.2.2.4 Broadcast Communication In broadcast operation, there are one source processor and N destination processors as in scatter operation. However, unlike the scatter operation, the same message is sent to all destination processors. This operation is used when the same data is necessary in all processors.

In Figure 11, the code for broadcast is shown. In our experiments, we measured the total communication time as a function of the message size. Also, we measured the communication time for each source processor: the broadcast communication is performed N times, and in the i-th broadcast communication, P_i $(0 \leq i \leq N-1)$, is chosen as the root processor.

4.2.3 Memory-Disk

Memory-disk operation is critical for out-of-core algorithms in which the data is larger than the available memory. The extra data that cannot be stored in the memory must be stored in a disk. Thus, data swapping is required which incurs memory-disk communication.

```
for (src = 0; src < Number_of_processors; src++)
for (msg_size=1; msg_size <16 Mbytes; msg_size *=2)
    Start timer;
    if (my_id == src)
        Send message to other processors;
    else
        Receive message from the source processor;
    End timer;
```

Figure 11: Pseudo-code for broadcast communication

There are two types of memory-disk communications: write-to-disk and read-from-disk communications. Write-to-disk operation involves three steps:

1. Check whether the pages that include the data are in the main memory.
2. If the page is not in the main memory, move the page from the disk to the memory.
3. Move data to the disk. The data is moved from user space to a library buffer, next to a disk buffer, and then to a physical hard disk.

If the data size is small, the user program can continue its operation without waiting for the completion of all three steps because the data is stored in a buffer. However, if the data size is larger than the minimum of these buffer sizes, then the user program must wait until all operations are completed. In our benchmarks, to measure the performance over a wide range of data sizes, the message size is increased to the maximum size allowed by the run-time environment.

In read operation, step 3 in the above write sequence is not necessary. Since the read operation takes a fewer number of steps, it takes less time than the write operation.

Since there is a large difference in the cost between read and write operations, we measured the two operation costs separately. To measure the memory-disk cost, we measured the communication time between the memory and the disk as a function of data size. The codes are shown in Figure 12 and Figure 13.

In the read operation, the buffer is cleaned before every read operation to force the read operation. Otherwise, the compiler may optimize the code so that the next iteration starts reading from the next portion of data since the first portion is already in the main memory. Also, in the write operation, for the same reason, the buffer is filled with dummy data. Then,, the compiler cannot avoid writing data that was written in the previous iteration. Finally, a file is opened and the time for read or write

```
for (msg_size = step_size; msg_size < 16 MB;
    msg_size +=step_size )

    Clean buffer;
    Open a file;
    Start timer;
    Read data from a file to the buffer;
    Stop timer;
```

Figure 12: Pseudo-code for disk read

```
for (msg_size = step_size; msg_size < 16 MB;
    msg_size +=step_size )

    Save random data into buffer;
    Open a file;
    Start timer;
    Write data in the buffer to a file;
    Stop timer;
```

Figure 13: Pseudo-code for disk write

operation is measured.

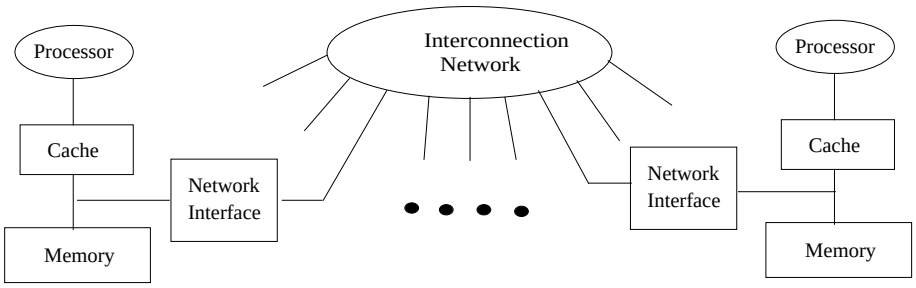


Figure 14: A typical HPC platform

4.3 Implementation Results on HPC Platforms

The benchmarks described in the previous section were implemented on several HPC platforms. A typical HPC platform is shown in Figure 14. Each processing node contains a processor, cache, memory, and a network interface.

The platforms on which the benchmarks were implemented were the IBM SP, the SGI/Cray T3E, and the SGI Origin 2000 installed at the Department of Defense (DoD) High Performance Computing Major Shared Resource Center, U.S. Army Corps of Engineers Waterways Experiment Station (CEWES MSRC). At the time of execution our benchmarks, the environments of these platforms were as follows:

- IBM SP
  - OS: IBM AIX v.4.1.5.x
  - Compiler: IBM C compiler v.3.1.4.0
  - MPI: IBM Parallel Operating Environment for AIX software v.2.1.0.24
- SGI/Cray T3E
  - OS: SGI/Cray UNICOS/mk v.2.0.3.14
  - Compiler: SGI/Cray C Compiler v.6.0.2.0
  - MPI: SGI/Cray MPI (Message Passing Toolkit - MPI) v.1.2.0.1
- SGI Origin 2000
  - OS: IRIX v.6.4
  - Compiler: SGI C Compiler v.7.2
  - MPI: SGI MPI (Message Passing Toolkit - MPI) v.3.0

4.3.1 Processor-Memory

The experimental results for processor-memory communication on the IBM SP are shown in Figures 45 - 56 in Appendix B. Figures 45 through 47 show the performance of the READ operation. Figures 48 through 50 show the performance of the WRITE operation. Figures 51 through 53 show the performance of the MULTIPLY operation. Figures 54 through 56 show the performance of the DIVIDE operation. Within each operation type, a seperate experiment was conducted for various data types (Integer, Single, Double). For example, Figure 45 shows Integer Read Operation, Figure 46 shows Single Read Operation, Figure 47 shows Double Read Operation, and so forth. Within each figure, part (a) shows the execution time while part (b) compares the execution time as the size of the total number of elements is increased by 512 in each iteration.

The results for the out-of-cache operations show the importance of two key parameters: the number of elements (N) and stride (S). The actual computational operation performed is not significant for most operations. The results for the in-cache operations show the significance of the type of operation performed.

For out-of-cache operations, two key factors determine the cost of memory-to-cache communication performance. The results in Figures 45 to Figures 56, part (a), show that for the basis stride of 1, the cost of communication increases linearly as N, the number of elements, increases. The figures show the costs of accessing N elements of various data types and performing READ, WRITE, MULTIPLY and DIVIDE operations on them. As it can be seen, the cost of the actual operation performed is not the dominating factor.

These figures also show the results of changing the stride in which the data is accessed. As stride S increases, the total data accessed increases linearly. As was explained previously in Section 4.2.1, in order to access N elements with stride S, the dataset must have $N \times S$ elements. Since data is brought into the cache in cache line blocks, the total size of the data brought into the cache affects the total cost of memory to cache communication performance. However, once the stride, S, is larger than the size of the cache line, the actual amount of data brought into the cache does not change because access to each element causes a cache miss to occur.

The difference in the cost of accessing N elements with $N \times i$ ($1 \leq i \leq 10$) elements with various strides is shown in Figure 45(b) to Figure 56(b). There is little variation to access an additional 512 elements, even when the total number of elements accessed is very large. The graph, in effect, shows the cost of accessing additional cache lines as stride S is increased. The slope tapers off after stride S becomes greater than the cache line size. This is expected since there is no difference in the number of cache lines brought into cache for $S \geq$ *cache line size*.

4.3.2 Processor-Processor

The processor-processor communication has been measured using the four communication primitives as described in Section 4.2.2 : permutation, pingpong, scatter, and broadcast. The implementation was performed using C and MPI. The experimental results are presented here.

4.3.2.1 Permutation Operation The experimental results on the SP, the T3E, and the O2K are shown in Figure 57 to 62 (See Appendix B). The results on the T3E and the O2K show that the communication time depends on the distance between processors. However, the SP results show that communication time is independent of the distance between processors. The reason is that the SP uses a Multi-Stage Interconnection Network (MIN) which provides the same low-level communication distance between any pair of processors. The T3E uses 3-D torus, and the O2K uses hierarchical hypercube; hence, the low-level distances between processors are not the same.

The more important reason for the difference in execution time is a link contention arbitration. As the pingpong operation results show (in Section 4.3.2.2), the low-level distance does not cause much difference in the communication time. Thus, we conclude that the ability of the operating system to arbitrate the link contention plays an important role in the observed difference in the execution times.

The communication time as a function of the message size shows that time is almost a linear function when the data size is large. As the message size doubles, the communication time also doubles. However, when the message size is small, the communication time increases by only a small amount.

From these results, the startup time and bandwidth/processor are obtained. The parameters are summarized in Table 1.

Table 1: Startup Time and Bandwidth

Platform	Startup Time (μsec)	Bandwidth (MB/sec/processor)
SP	54	50
T3E	29	100
O2K	85	15

The experimental results indicate that communication time on the T3E shows the best performance. This is due to the high performance network implemented on the T3E.

4.3.2.2 Pingpong Operation The experimental results on the SP, the T3E, and the O2K are shown in Figures 63 to 68 (See Appendix B). The results of the pingpong communication are different from

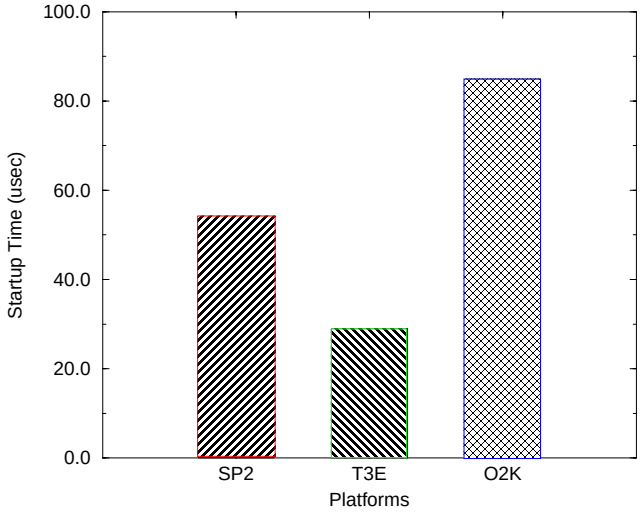


Figure 15: Startup Time

those of the permutation communication; Both the SP and the T3E results do not depend on the processor-IDs, while the results on the O2K shows a little variation.

The reason for the same communication time on the SP and T3E is the same as explained for the permutation results: the interconnection network is a MIN in which the distance between any pair of processors is the same.

On the T3E, the interconnection network is a 3-D torus. Thus, the distances between pairs of processors are not the same. However, our results show that actual communication time does not depend on physical distance. This shows that the cost difference caused by different physical distance is insignificant compared with the overhead incurred by other factors such as the software.

On the O2K, the interconnection network is a hierarchical hypercube. Thus, the distance between various processors is not the same. Also, the communication time between a pair of processors that are connected via one hub (does not need the interconnection fabric) takes less time than the communication time between a pair of processors that are connected via more than one hub and the interconnection fabric.

Also, we found that the pingpong operation shows "pipelined communication." When a source processor starts communication, the destination processor starts receiving a message. Then, the destination processor starts sending the data back to the source processor even before it finishes receiving the entire message from the source.

In pingpong experiments, the T3E shows the best performance because of better interconnection network hardware as described in Section 4.3.2.1.

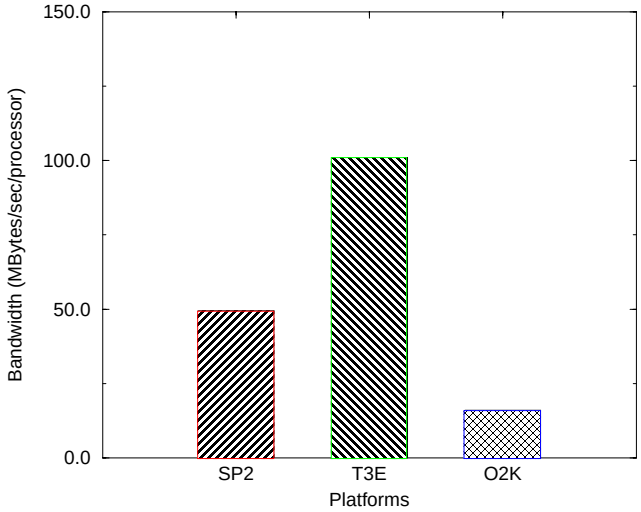


Figure 16: Bandwidth

4.3.2.3 Scatter Communication The experimental results on the SP, T3E, and the O2K are shown in Figure 69 to 74 (See Appendix B). The scatter communication results are similar to the pingpong communication results: The communication time does not depend on the distance between processors.

The results show that the T3E shows the best performance because of superior hardware interconnection network.

4.3.2.4 Broadcast Communication The experimental results on the SP, T3E, and the O2K are shown in Figure 75 to 80 (See Appendix B). The broadcast communication results are similar to permutation communication results: the communication time on the T3E depends on the root processor when the number of processors is 16. Similar results were obtained when the number of processors was larger than 16. We conclude that the difference is from the link contention among the processors as in permutation operation.

The results indicate that the T3E shows the best performance because of a superior hardware interconnection network.

4.3.3 Memory-Disk

The experimental results on the SP, the T3E, and the O2K are shown in Figure 81 and 83 (See Appendix B). The results show that the startup cost is very high compared with processor-processor or processor-memory communications. The startup cost for memory-disk is in the msec range while

startup cost for processor-processor and processor-memory is in the tens of μsec range.

When the data size is large, the communication time is proportional to the message size. Since the algorithms on HPC platforms manage large data sizes, the size of data that is transferred between the memory and disk is usually large. Thus, the initial startup cost can be easily hidden by the data transfer time.

The write operation takes more time than the read operation for the same data size. It is because the write operation needs to read pages from disk to memory before the writing operation if the pages are not in memory. That is, the write time is the sum of read time and "pure" write time.

The results are summarized in Table 2. The results show that the SP and the T3E have similar performance; but the O2K has the worst performance.

Table 2: Memory-Disk Communication Time

Operation	Platform	Startup Time (msec)	Bandwidth (MB/sec)
Write	SP	1.0	155
	T3E	3.5	149
	O2K	2.0	70
Read	SP	1.0	255
	T3E	1.0	266
	O2K	1.5	90

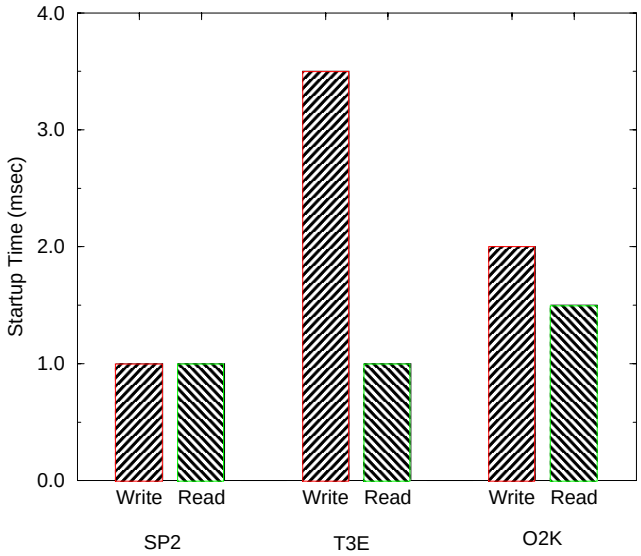


Figure 17: Startup Time

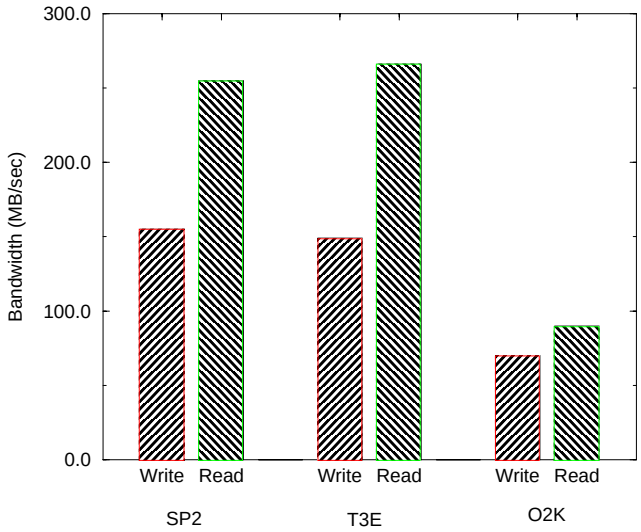


Figure 18: Bandwidth

5 Our Preliminary Model of HPC Platforms

In this section, we first describe previous models. Then, we present our model.

5.1 Previous Models

The PMH model [6] and the Two-Level Memory Model [33] have been proposed for HPC platforms. These are briefly described here.

5.1.1 Parallel Memory Hierarchy (PMH) Model

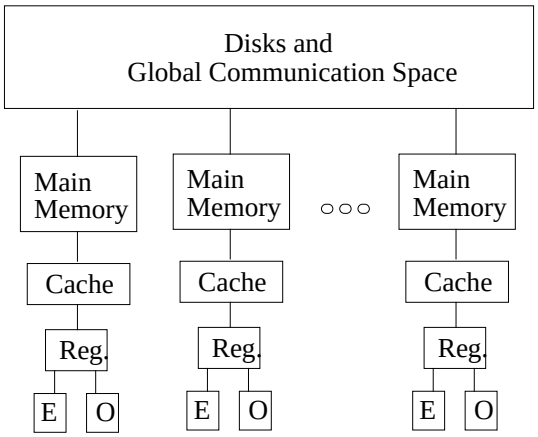


Figure 19: PMH model of the IBM SP1. Boxes labeled E (for EVEN) and O (ODD) are functional units that model the two-stage floating-point pipeline.

In this model [6], the interprocessor communication cost and the memory hierarchy are considered. A parallel computer is modeled as a tree of modules. Each non-leaf node represents a memory module such as disk, main memory, cache, and register. A leaf node represents a computing element such as a functional unit in a CPU. The PMH model of IBM SP1 is shown in Figure 19. Each child connects to its parent by a unique channel. Modules hold data. Data in a module are partitioned into blocks. A block is the unit of transfer on the channel connecting a module to its parent. The model has four parameters for each module m: s_m is the number of bytes per block of m, n_m is the number of blocks in m, c_m is the number of children of m, and t_m is the number of cycles to transfer a block between m and its parent.

This model considers the interprocessor communication and the secondary memory access. However, the model can not represent the hard disk system if the hard disk is distributed among processor nodes. Examples of the architectures using this model are shown in Figure 20. In Figure 20, (a)

shows a shared disk system. The processors are interconnected by a high-bandwidth network. In Figure 20, (b) shows the distributed disk system where the disks are interconnected by a low-bandwidth network. Thus, it inherently assumes that the interprocessor communication is performed through the disk. Hence, the model can not represent a distributed disk system in which the processors are interconnected by a high-bandwidth network.

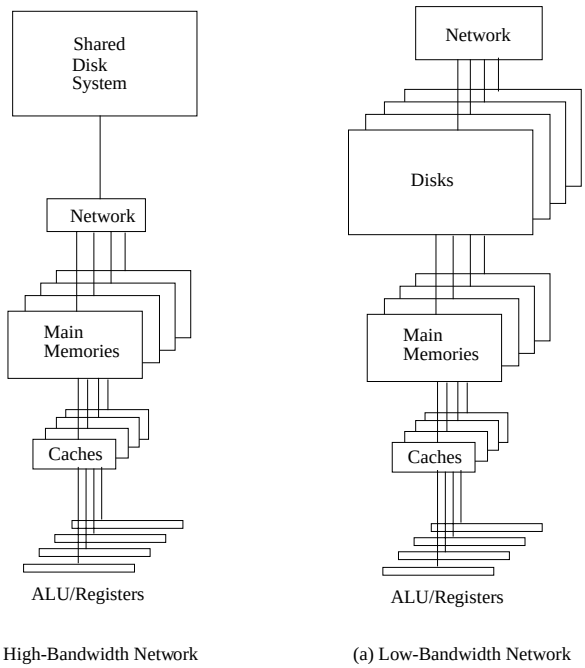


Figure 20: PMH model of parallel systems

Another drawback of this model is high complexity. Since every memory module including cache and registers is modeled using four parameters, the resulting model is too complicated. The more parameters a model has, the more difficult it is to design and optimize algorithms.

5.1.2 Two-Level Memory Model

This model [33] has been proposed for the development of parallel input/output algorithms. The underlying architecture is shown in Figure 21. In this model, the number of input/output operations is used to estimate the communication cost.

The data transfer time to or from the disk is ignored because that the seek time in a disk access operation is much larger than the data transfer time for small data sizes. However, the data transfer time is an important factor when the data size is large. The disadvantage of this model is that the communication time between processors is completely ignored because communication time between

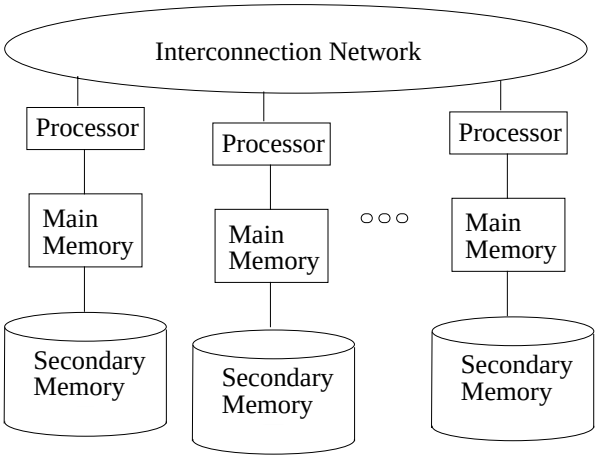


Figure 21: HPC architecture used in two-level memory model

processors is insignificant compared with the disk access time. However, it is an important factor if the number of communication operations or the amount of data transferred is large.

5.2 Integrated Memory Hierarchy Model

Our HPC platform model considers three main costs: processor-processor, processor-memory, and memory-disk costs. The processor-memory and memory-disk cost involves only communication cost. However, the processor-processor cost consists of computation and communication costs between processor and memory.

5.2.1 Processor-Memory

The experimental results of the processor-memory communication on the IBM SP are shown in Figures 45 - 56.

The results of the Integer READ operation experiment (Figure 45) is repeated here in Figure 23 and Figure 24 to illustrate our approach to modeling the processor-memory communication cost. The results of this out-of-cache operation shows the importance of two key parameters: the number of elements (N) and the stride (S). In Figure 22, the execution time of reading arrays with various number of elements is shown. As can be seen in the figure, the total execution time increases linearly as the number of elements is increased linearly.

In Figure 23 and 24, the stride is varied for each of the arrays (with various number of elements) shown in Figure 22. As the stride is increased, the total execution time increases linearly for each array, and peaks out towards the end. The peak point occurs when the stride is large enough that each access incurs a cache miss. This phenomenon can be clearly seen in Figure 24. In this graph,

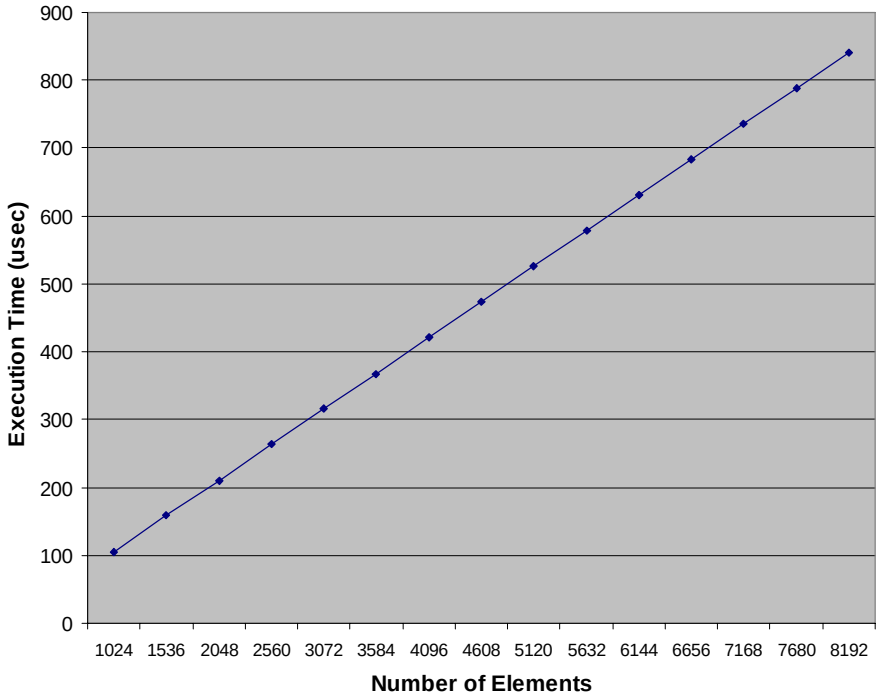


Figure 22: Read Integer using various number of elements

the difference in execution time for each consecutive array size from Figure 23 is shown. This graph, in essence, shows the cost of executing 512 additional elements starting from some base size.

The cost of communication increases linearly as the number of elements N increases for the basis stride = 1. This cost, T, can be modeled using the following linear equation

$$T = T_n \times N_n \tag{1}$$

where T_n is the time for transfering a byte of data between the processor and memory.

As stride S increases, the total size of the data increases linearly until S is equal to the cache line size. This phenomenon is explained in the implementation results section (Section 4.3.1). Also, this can be modeled using a linear equation

$$T = T_c \times N_c \tag{2}$$

where T_c is the time to bring a cache line to the cache, and N_c is the number of cache lines transferred to cache.

Finally, we identified a change in the slope of the stride-time graph. For example, in Figure 45(a),

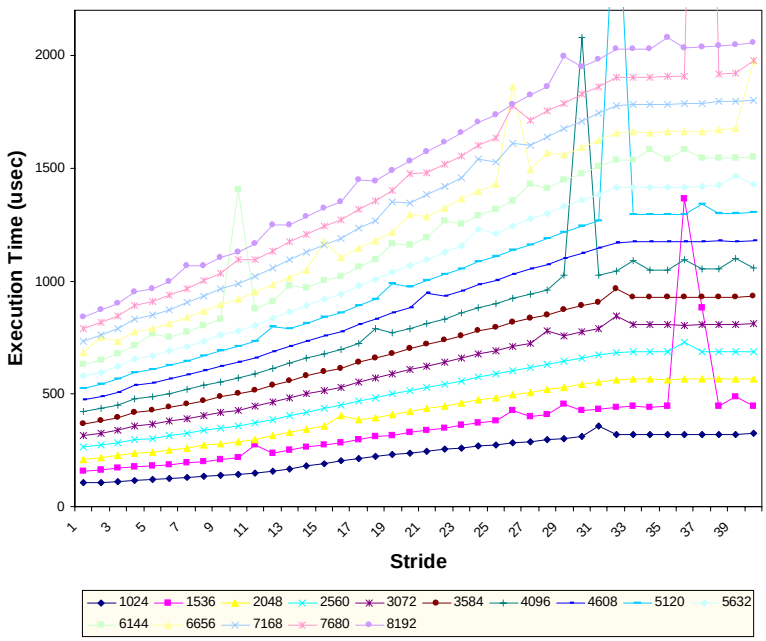


Figure 23: Processor-Memory communication: Read Integer

there is a change in slope at $S = 12$. The slope for $S \geq 12$ is steeper than that for $S \leq 12$. We cannot explain this phenomenon because we have no in-depth understanding of platform implementation. However, this causes inaccuracy in the calculations in the processor-memory time, and we add another linear equation to compensate the difference. The additional equation is

$$T = T_e \times S \tag{3}$$

where T_e is a constant to compensate for the difference in the slopes.

Therefore, the overall processor-memory cost is:

$$T = T_e \times N + T_c \times N_c + T_e \times S \tag{4}$$

where T_e is the time for transfer between the processor and memory per byte, T_c the time to bring a cache line to the cache, N_c the number of cache lines transferred to the cache, and T_e a constant to compensate for the difference in the slopes.

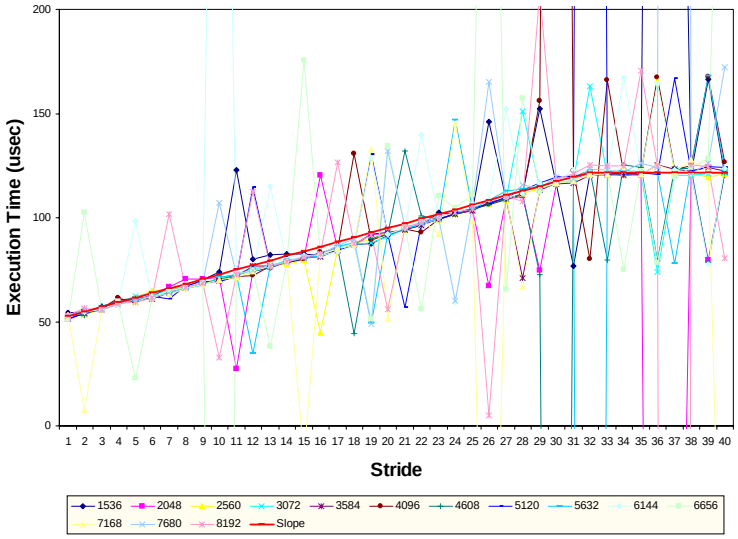


Figure 24: 512 Difference Graph: Read Integer

Table 3: Parameter Values for Processor-Memory Cost

Platform	T_n	T_c	T_e
SP	120 nsec	135 nsec	100 nsec

5.2.2 Processor-Processor

Our model of processor-processor communication uses results of permutation communication, since it can be considered as a general communication pattern. For example, an all-to-all communication can be implemented using many steps of permutation communication.

Figure 25 shows the permutation communication time as a function of message size. In Figure 26, the lower-left corner is enlarged.

Figure 25 shows that the communication cost is proportional to message size. However, when the message size is small (See Figure 26), there is a relatively large communication startup cost. Based on these observations, the communication time between two processors can be modeled using a linear function of the message size as follows:

$$\text{Communication time between a pair of processors} = T_s + m\tau_d \tag{5}$$

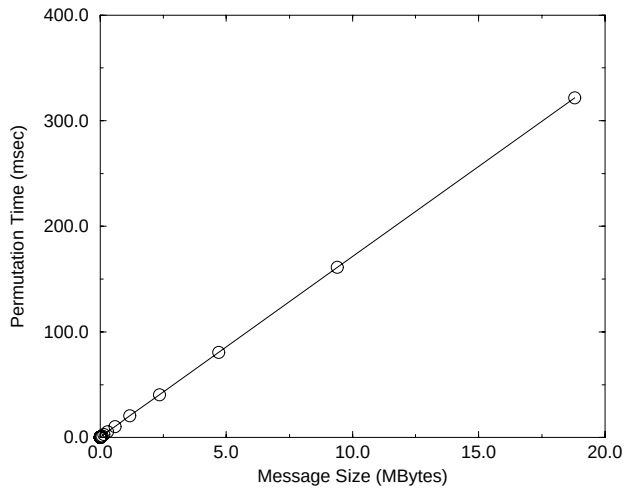


Figure 25: Permutation communication results on the SP

where T_s = startup time, and $\tau_d = 1/$bandwidth = data transfer time per byte per processor.

The T_s and τ_d are obtained using our permutation communication benchmark experimental results. The parameters for the SP and the T3E are shown in Table 4.

Table 4: Startup Time and Bandwidth

Platform	Startup Time (μsec)	Bandwidth (MB/sec/processor)
SP	54	50
T3E	29	100
O2K	85	15

Using equation (5), the time to perform more complex communication patterns can also be modeled. In these cases, we found that the startup time does not a show large variation for different communication patterns. However, the communication time depends on the total data size and the number of processors. Thus, when a communication pattern consists of j steps, the total message size is $\sum_{i=1}^{j} m_i$, and the total data transfer time is $\sum_{i=1}^{j} m_i \times \tau_d$. Therefore, the total communication time for a communication pattern is

$$\begin{aligned} \mathrm{T}_{comm} &= \text{Startup time} + \text{Total data transfer time} \\ &= T_s + \sum_{i=1}^{j} m_i \tau_d \end{aligned}$$

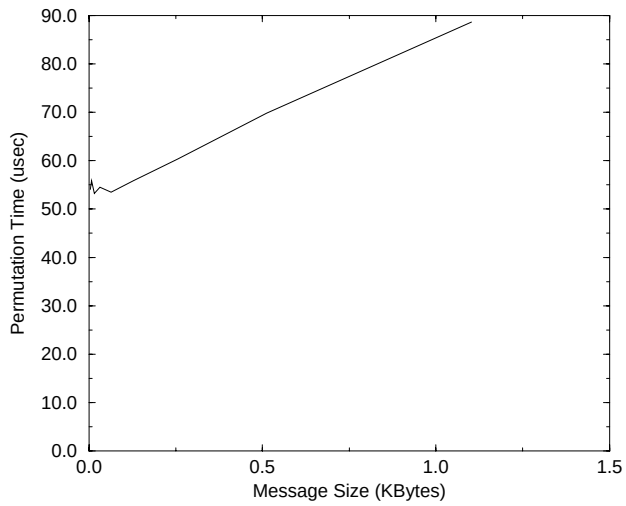


Figure 26: Permutation communication results on the SP

where T_s = startup time, τ_d = data transfer time per byte, and j = the number of communication steps in the communication pattern.

With this equation, the communication time for various communication patterns can be estimated as follows:

- Permutation time = $T_s + m\tau_d$
- Pingpong time = $T_s + m\tau_d$
- Scatter time = $T_s + (P-1)m\tau_d/2$
- Broadcast time = $T_s + (\lg P)m\tau_d/2$

where m is the size of the message that is transferred to each destination processor, and P is the total number of processors involved in the communication.

To validate our model, we compared the estimated communication time and the actual communication time for each of our benchmarks. The following data block sizes were used: each communication is:

- For permutation and pingpong: 16 MB,
- For scatter among 8 processors: 16 MB on the root processor. 2 MB sent to each destination processor,

- For scatter among 16 processors: 16 MB on the root processor and 1 MB for the destination processors,
- For broadcast among 8 processors: 2 MB, and
- For broadcast among 16 processors: 1 MB.

In estimating the pingpong communication time, we used the fact that the architectures support "pipelined communication" as explained Section 4.3.2.

The results show that the model can accurately predict the communication time on the SP. On the T3E, the maximum error was about 30%. The error range can be further reduced by adjusting the parameters.

Table 5: Predicted Time and Actual Time (msec)

Communication operation	Actual	Predicted	Error
Permutation	320	320	0%
Pingpong	350	320	9.4%
Scatter on 8 proc.	160	140	14%
Scatter on 16 proc.	170	150	13%
Broadcast on 8 proc.	63	60	5%
Broadcast on 16 proc.	38	40	5%

5.2.3 Memory-Disk

The read and write operation times are shown in Figure 27, Figure 28, and Figure 29 for the SP, the T3E, and O2K, respectively. The overall graph shows that the read and write times are proportional to the message size.

The spikes at message size = 1.5 and 3.5 are due to random operating system behavior. We performed our experiments over many iterations and found that the spikes are random, i.e., there was no regular pattern nor consistency in the appearance of these spikes. From this, we conclude that the spikes are not related to any parameter nor characteristic of the underlying hardware platform. We reason that this is probably due to the operating system behavior and interactions with other jobs on the system.

The disk operations can be modeled using a linear equation as a function of the data size. However, the write operation takes more time than the read operation because the write operation needs to perform a read before the data is written to the disk, if the page containing the data is not in the memory. Thus, the read and the write operations are modeled using different parameters.

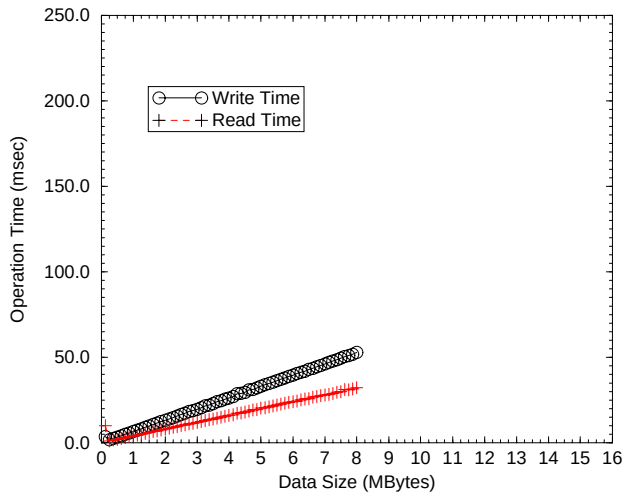


Figure 27: Disk operation results on the SP

Also, there is a large startup cost in the msec ranges. Even though recent technological advances have significantly improved the performance of disk, the startup cost is still large compared with data transfer time per byte. Hence, our model incorporates the startup cost.

Therefore, the disk operation time can be modeled using a linear equation for the read and write operations. However, since the parameters for read and write operations are different, we use a separate linear equation for each operation.

$$\begin{aligned} \text{Disk operation time} &= \text{disk read time + disk write time} \\ &= (T_r + m_r\tau_r) + (T_w + m_w\tau_w) \end{aligned}$$

where T_r is startup time for the read operation, m_r is the read message size, τ_r is inverse of the read bandwidth, T_w is startup time for the write operation, m_w is the write message size, and τ_w is the inverse of the write bandwidth.

The parameters obtained using our benchmark suites are summarized in Table 6.

5.2.4 Integrated Memory Hierarchy Model

The complete model is obtained by integrating the models for each communication. An overview of our model is shown in Figure 30. The complete model can be written as follows:

$$\text{Execution time} = \text{processor-memory execution time}$$

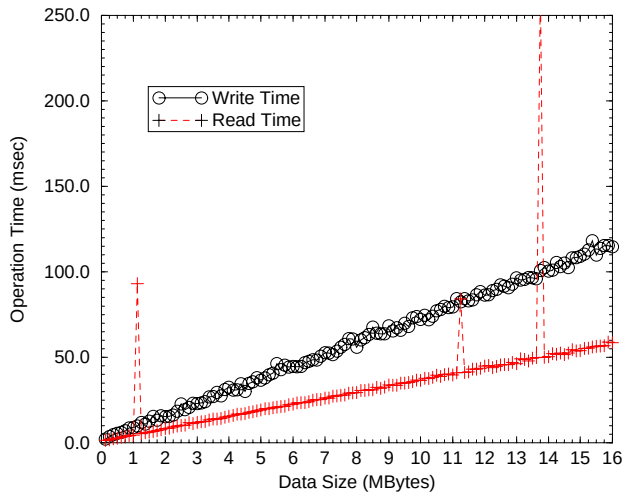


Figure 28: Disk operation results on the T3E

Table 6: Memory-Disk Communication Time

Operation	Platform	Startup Time (msec)	Bandwidth (MB/sec)
Write	SP	1.0	155
	T3E	3.5	149
	O2K	2.0	70
Read	SP	1.0	255
	T3E	1.0	266
	O2K	1.5	90

$$
\begin{aligned}
& + \text{processor-processor communication time} \\
& + \text{memory-disk communication time} \\
= \; & T_n \times N_n + T_c \times N_c + T_e \times S \\
& + T_s + m\tau_d \\
& + T_r + m_r\tau_r + T_w + m_w\tau_w
\end{aligned}
$$

where T_n = the data transfer time between the processor and memory per byte,
N_n = the number of data elements that are transferred to cache,
T_c = the time to bring a cache line to the cache,
N_c = the number of cache lines that are transferred to the cache,
T_e = a constant to compensate for the difference in the slopes,

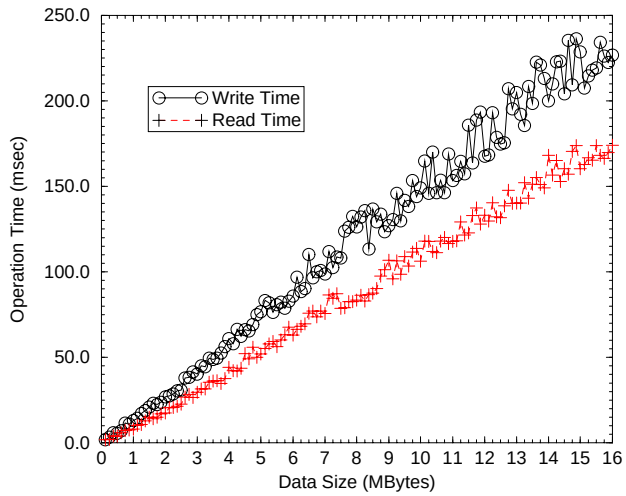


Figure 29: Disk operation results on the O2K

S = stride in which data is accessed,
T_s = startup time between processors,
m = size of the message transferred between processors,
$\tau_d = 1/$bandwidth,
T_r = startup time for the read operation,
m_r = the read message size,
τ_r = inverse of the read bandwidth,
T_w = startup time for the write operation,
m_w = the write message size, and
τ_w = the inverse of the write bandwidth.

5.3 Significance and Use of Our Model

Our preliminary model of HPC platforms is:

- As an integrated model, it consists of three main costs in peforming computation on HPC platforms: processor-memory, processor-processor, and memory-disk. The computation cost is included in the processor-memory cost. Thus, in our model, the costs for computation and communication among various HPC components are considered.

- As a simple model, an HPC platform consists of a number of components, each having a large number of parameters. To model HPC platforms with very high accuracy, a large number of

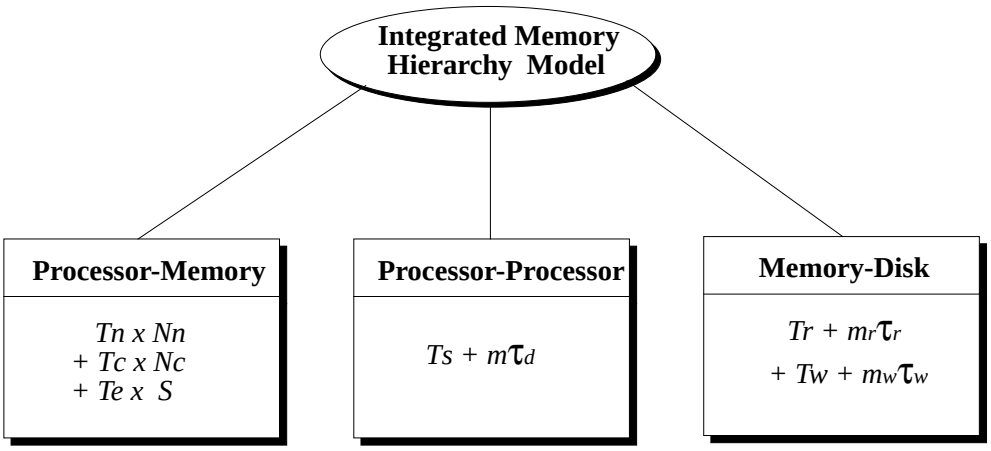


Figure 30: An overview of the preliminary model of HPC platforms

parameters need to be included in the model. Such a model becomes too complex to be of value to end-users. Therefore, we first identified the three main costs. These costs are modeled using simple equations providing users a simple view of HPC platforms. Also, we avoided discontinuous functions to avoid complex calculations. These efforts simplified the model for users.

- There is a trade-off between accuracy and simplicity. In our model, we sacrificed some amount of accuracy for simplicity. However, we obtained a model accurate enough for the design and analysis of algorithms on HPC systems. An example is shown in the next section.

- Our model is useful for design and analysis of algorithms on HPC platforms. Design and analysis of algorithms require understanding of HPC platforms on which the algorithm is used. Our model provides a simple and fairly accurate view of the HPC platforms. With the model, users can predict performance of their code. The users can optimize the code before the actual run. Also, after a test run, users can easily analyze the execution time using our model. Thus, the users can save time and effort in designing and analyzing algorithms on HPC platforms.

6 An Illustrative Example: Matrix Multiplication

In this section, we show the validity of the proposed HPC model using an example application. We use the matrix multiplication application. We estimate the execution time and compare it with the actual execution time.

6.1 Previous Algorithm

Cannon's algorithm [4] is one of the most widely used algorithms for matrix multiplication on multiprocessor platforms.

In this section, Cannon's algorithm [4] for computing $A = B \times C$ is explained with an example. We assume the number of processors is 3 $\times$ 3 = 9.

Initially, matrices A, B, and C are partitioned into $P \times P$ blocks (See Figure 31 (a)), where P^2 is the number of processors. These data blocks are distributed to each of the processors in a skewed fashion as in Figure 31 (b). Note that the distributions of B and C are different.

In each processor, the computation is performed using the data available in each processor.

Then, the data blocks of B are shifted left and the blocks of C are shifted upward as shown in Figure 31 (c). The next portion of the computation is performed. These shift-and-computations are repeated $P - 1$ times to complete the matrix multiplication. The last step is shown in Figure 31 (d) for P=3.

6.2 Our Algorithm

Even though the Cannon's algorithm minimizes the communication overhead among processors, it does not consider cost between processor and memory. In our algorithm, we analyzed the cost of communication between the processor and memory as well as the cost among processors using our model. Our model is very useful because it represents all processor-memory and processor-processor costs.

Using our model, we analyzed the algorithm and estimated the total execution time. Our analysis showed that when the the number of processors is small, the main cost of matrix transpose is the cost of data transfer between processor and memory, especially the transfer cost for matrix C.

To simplify the analysis, we assume all matrices are size $N \times N$. In performing matrix multiplication $A = B \times C$, a significant portion of the time is spent in accessing array C because matrix elements are usually stored in row major order, as in the C language compilers. Because array C is accessed in column major order, but stored in row major order, almost every access to an element causes a cache miss to occur. The number of cache misses is $O(N^3)$.

Intuitively, if the data in array C can be rearranged such that the number of cache misses is reduced, a large improvement could be obtained. In this case, transposing array C would allow con-

Matrix A

A_{00}	A_{01}	A_{02}
A_{10}	A_{11}	A_{12}
A_{20}	A_{21}	A_{22}

Matrix B

B_{00}	B_{01}	B_{02}
B_{10}	B_{11}	B_{12}
B_{20}	B_{21}	B_{22}

Matrix C

C_{00}	C_{01}	C_{02}
C_{10}	C_{11}	C_{12}
C_{20}	C_{21}	C_{22}

(a) Initial partitioning of matrices

P_0 A_{00} B_{00} C_{00}	P_1 A_{01} B_{01} C_{11}	P_2 A_{02} B_{02} C_{22}
P_3 A_{10} B_{11} C_{10}	P_4 A_{11} B_{12} C_{21}	P_5 A_{12} B_{10} C_{02}
P_6 A_{20} B_{22} C_{20}	P_7 A_{21} B_{20} C_{01}	P_8 A_{22} B_{21} C_{12}

(b) Initial distribution of the matrices

P_0 A_{00} B_{01} C_{10}	P_1 A_{01} B_{02} C_{21}	P_2 A_{02} B_{00} C_{02}
P_3 A_{10} B_{12} C_{20}	P_4 A_{11} B_{10} C_{01}	P_5 A_{12} B_{11} C_{12}
P_6 A_{20} B_{20} C_{00}	P_7 A_{21} B_{21} C_{11}	P_8 A_{22} B_{22} C_{22}

(c) After the first step

P_0 A_{00} B_{02} C_{20}	P_1 A_{01} B_{00} C_{01}	P_2 A_{02} B_{01} C_{12}
P_3 A_{10} B_{10} C_{00}	P_4 A_{11} B_{11} C_{11}	P_5 A_{12} B_{12} C_{22}
P_6 A_{20} B_{21} C_{10}	P_7 A_{21} B_{22} C_{21}	P_8 A_{22} B_{20} C_{02}

(d) After the second step

Figure 31: Illustration of Cannon's matrix multiplication algorithm (P=3)

secutive access of the data with optimal data access patterns. However, the cost to transpose array C could be significant. In order to evaluate this approach, we first analysed the performance of transposing an array. Even a naive transpose algorithm in which the array is accessed in column major order and stored in row major order incurs only $O(N^2)$ cache misses. Using our model, we estimated the cost of transposing array C. Without actual coding, we determined using the IMH model that for most matrix sizes of interest, the cost of transposing the array is insignificant compared to the $O(N^3)$ potential cache misses that would occur otherwise. After the transpose operation, both arrays are accessed in row major order.

We performed the experiments with 512×512 size arrays on each node of a SGI/CRAY T3E. The straightforward implementation of Cannon's matrix multiplication took 54.6 secs. Our IMH model had predicted 68.5 secs. Most of the time was spent in data access. Our memory hierarchy

optimized matrix multiplication took 8.6 secs, including the overhead for first transposing array C. The transpose operation overhead was 109 msecs. This is a significant improvement in performance.

7 Parallelizing a Benchmark Application

In this section, we illustrate the design and performance tuning of a parallel algorithm for a benchmark application in fluid dynamics. CEWES provided this code. We first describe the structure of this benchmark in terms of the data cube size, the basic operations, and the usage of allocated processors. Then we describe a previous approach to parallelize this benchmark. The previous algorithm is not scalable but suitable only for a fixed number of processors. When the number of processors is increased beyond six, the previous algorithm does not balance the workload. In addition, the algorithm uses a straightforward approach to exchange the boundary data among the processors. This simple approach results in significant overhead in communication cost.

In our approach, we designed a scalable and parallel algorithm to perform this benchmark application. We perform load balancing to distribute the workload onto the total processors assigned to the application program. In addition, we designed and implemented a communication algorithm which allows parallel communication without node contention. Using the communication algorithm, the remote data is delivered in parallel.

7.1 Overview of the Code

The data and the computations can be represented as a 3-dimensional grid of size 121 (Width) $\times$ 4 (Depth) $\times$ 81 (Height). Each data element can be viewed as a point in this grid. The value of each grid point is computed and updated using its current value and the values of its seven neighbors as shown in Figure 32. A flowchart of the code is shown in Figure 33. The top-level subroutines shown in the flowchart are described here.

- OWN_PL: This subroutine determines which processor computes and updates the value of a grid point. The 3-dimensional index of a grid point is mapped onto a 1-dimensional index.

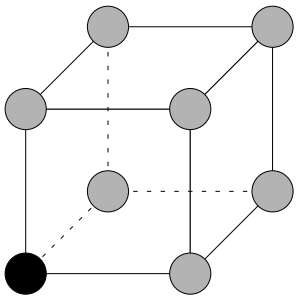


Figure 32: A grid point and its seven neighbors

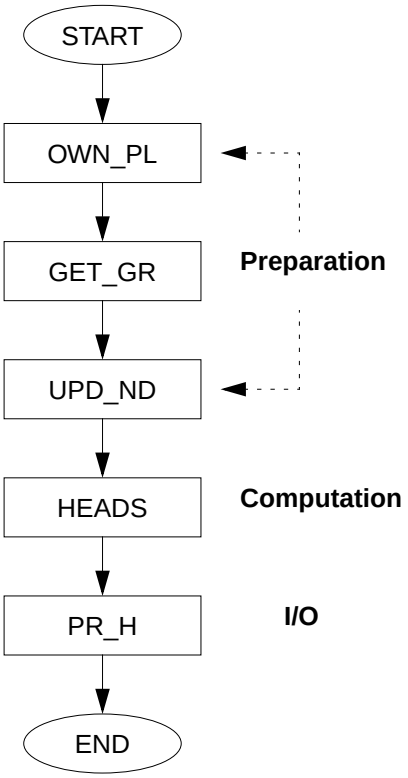


Figure 33: Flowchart of FT.F

This linear sequence of grid points is divided and assigned to all processors. Each processor determines the ownership of a grid point in the range of the sequence assigned to it. The ownership ranges from 0 to 5 that is a processor's identification. Therefore, only 6 processors will be involved in computing and updating the values of the grid points in the subroutine UPD_ND. At the end, processors determine and store the local index of a grid point to its owner through broadcasting ownership information.

- GET_GR: In this subroutine, all processors determine the real coordinates of grid points and generate their initial values. Then they broadcast the coordinates to others.
- UPD_ND: This subroutine updates the initial values of boundary points.
- HEADS: In this subroutine, the value of a grid point is computed and updated by its owner processor. This operation is performed in several loops. After each iteration, interprocessor communication occurs to update the boundary data. This subroutine consumes most of the execution time. Consequently, we focused our effort on efficiently parallelizing this subroutine and the low-level subroutines that it calls.
- PR_H: This subroutine prints the final results.

7.2 Previous Implementation

The previous parallel implementation does not scale as the number of processors increases. The implementation does not use all the available processors to compute the output data in the subroutine HEADS as mentioned in Section 7.1. In addition, its communication algorithm to exchange boundary data incurs a large overhead. As the number of processors increases, the communication cost begins to dominate. We address these problems below.

7.2.1 Workload Distribution for Computation

In the preparation step of the previous implementation, all processors are used to perform the required operations in parallel. However, in the computation step, only 6 processors are used to compute and update the values of grid points. As described in Section 7.1, the subroutine OWN_PL determines the ownership of a grid point. The Fortran code written for this operation is shown in Figure 34. Figure 35 demonstrates the scheme subroutine OWN_PL generated. The scheme is used in the subroutine HEADS to distribute the workload onto the processors. This scheme causes an unbalanced workload distribution. The measured execution time of the previous implementation on the IBM SP-2 and the SGI/Cray T3E is shown in Figure 36.

```
...
ndomi = 41
ndomj = 4
ndomk = 41

kplane = imax*jmax
n1 = myid*nodes + 1
n2 = min0(n1 + nodes - 1,, numpg)

do n = n1, n2
  k = (n - 1)/kplane
  kk = k/ndomk
  j = (n - 1 - k*kplane)/imax
  jj = j/ndomj
  i = mod(n - 1, imax)
  ii = i/ndomi
  ndproc(n - n1 + 1) = kk*3 + jj*6 + ii
end do
...
```

Figure 34: Fortran program used to determine the ownership of a grid point

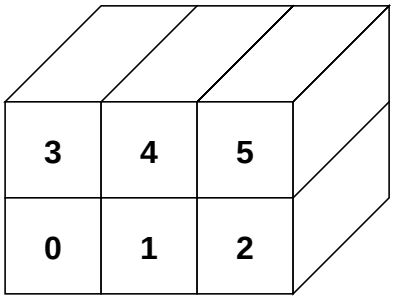


Figure 35: Workload distribution in the previous implementation

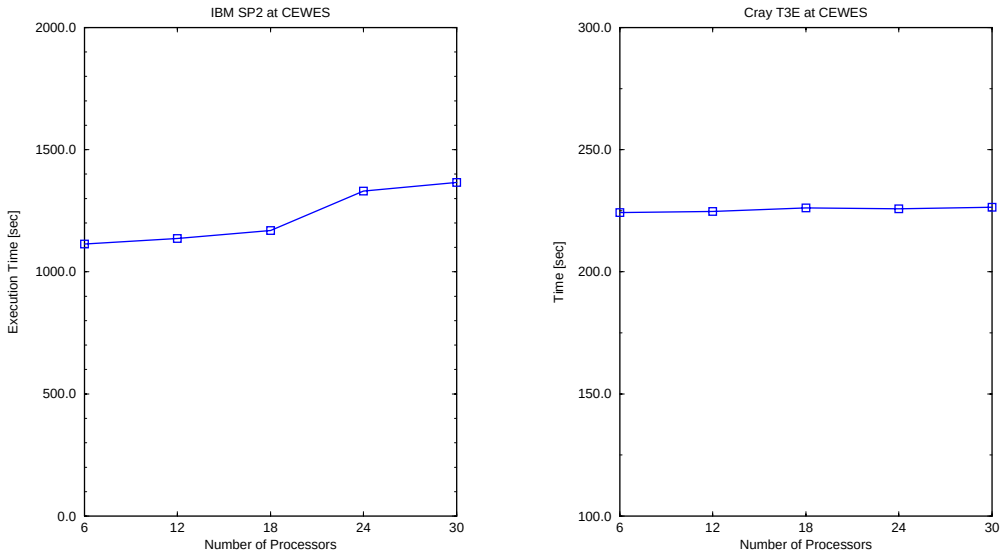


Figure 36: Execution time of the previous implementation

7.2.2 Interprocessor Communication

In each iteration of the subroutine HEADS, the value of each grid point is computed and updated based on its current value and values of its seven neighbors as shown in Figure 32. In addition, the workload is distributed onto the processors using the workload distribution scheme shown in Figure 35. Therefore, each processor needs to communicate with its three neighbors to obtain the updated values for its boundary grid points. Figure 37 illustrates the induced interprocessor communication pattern.

In the previous implementation, the interprocessor communication is performed in a serial manner (i.e., only a pair of processors exchange their data at a time, while all other processors remain idle). This approach is very inefficient. It causes significant communication overhead which degrades the overall performance of the parallel algorithm. As the number of processors increases, the time for performing the above serial interprocessor communication begins to dominate the total execution time.

7.3 Our Implementation

We have developed algorithmic techniques to improve the benchmark's performance. Based on these techniques, we developed a parallel algorithm.

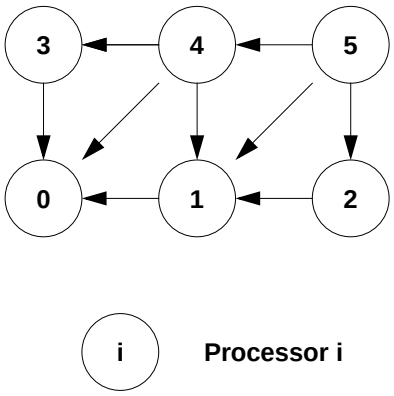


Figure 37: Communication pattern between processors

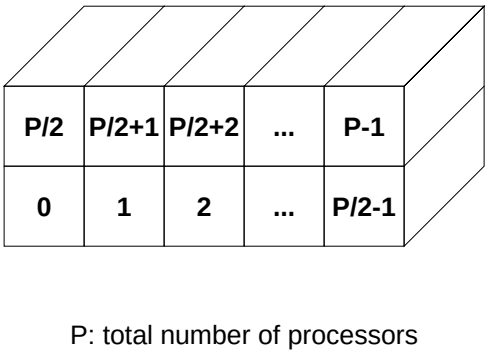


Figure 38: Workload distribution in our implementation

7.3.1 Load Balancing

In our algorithm, workload is distributed to all available processors by using the distribution scheme shown in Figure 38. The Fortran code rewritten for this operation is shown in Figure 39. To illustrate the effectiveness of our load balancing, we compare the execution time of our implementation using a balanced workload distribution scheme with the previous approach in Figure 40.

7.3.2 Parallel Interprocessor Communication

Using the workload distribution as shown in Figure 38, each processor needs to communicate with at most three neighbor processors to obtain the updated boundary data. In the previous implementation, interprocessor communication is performed in serial fashion. This approach allows only one

```
...
kplane = imax*jmax
n1 = myid*nodes + 1
n2 = min0(n1 + nodes - 1,, numpg)
noproc_half = noproc/2

do n = n1, n2
  k = (n - 1)/kplane
  kk = k/((kmax + 1)/2)
  i = mod(n - 1, imax)

  if(mod(imax, noproc_half) .eq. 0) then
    ii = i/(imax/noproc_half)
  else
    ii = i/(imax/noproc_half + 1)
  end if
  ndproc(n - n1 + 1) = noproc_half*kk + ii
end do
...
```

Figure 39: Rewritten Fortran program to determine the ownership of a grid point

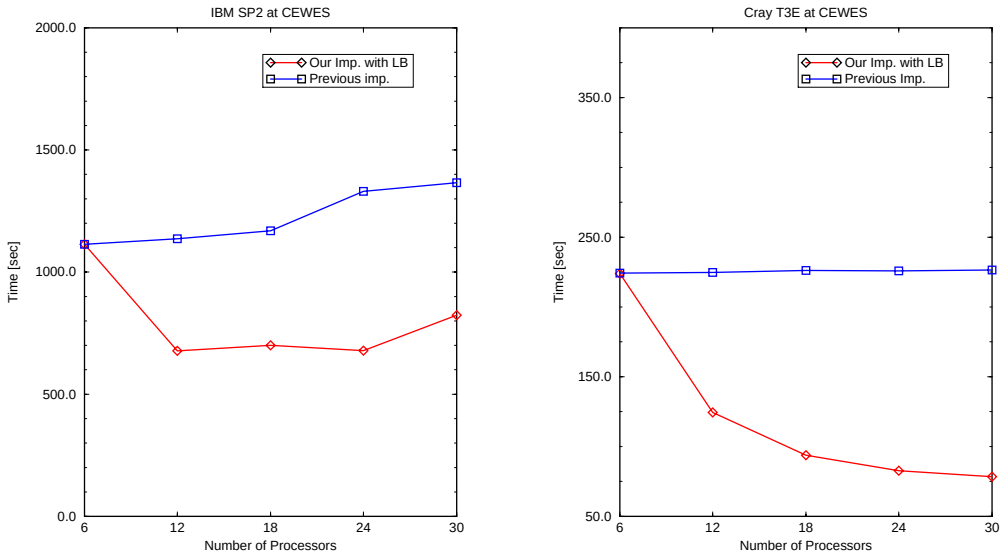


Figure 40: Execution time of our implementation with load balancing

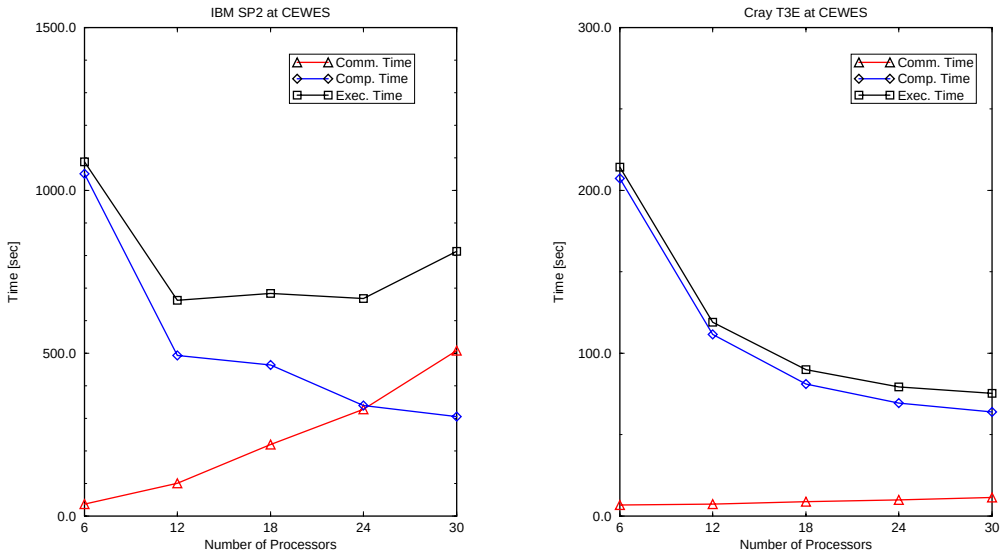


Figure 41: Computation time and communication time of our implementation with load balancing

pair of processors exchanging their data at a time. In this approach, the total communication time increases proportional to the number of processors. To show the communication overhead of a serial algorithm, we measured the communication cost after load balancing. As shown in Figure 41, the interprocessor communication cost increases as the number of processors increases.

In our implementation, interprocessor communication is performed in parallel as shown in Figure 42. Our parallel communication consists of three steps:

1. Each processor sends data to its left neighbor and receives data from its right neighbor.

2. A processor belongs to one top group or bottom group. Each processor in the top group sends data to its counterpart in the bottom group.

3. A processor belongs to one top group or bottom group. Each processor in the top group sends data to its diagonal counterpart in the bottom group.

We compare the communication times in Figure 43. The execution times are compared in Figure 44.

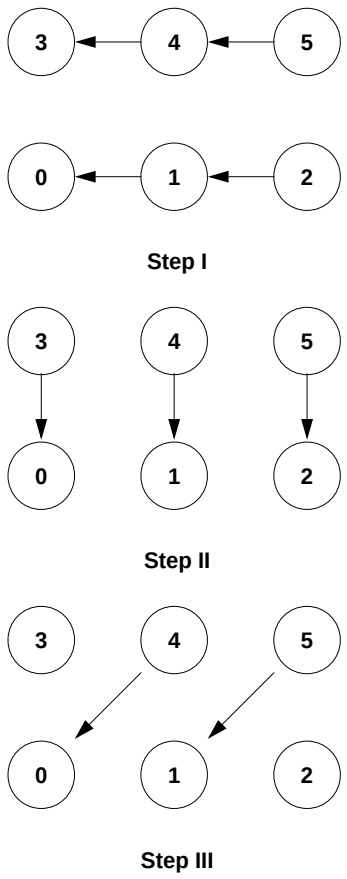


Figure 42: Parallel communication pattern in our implementation

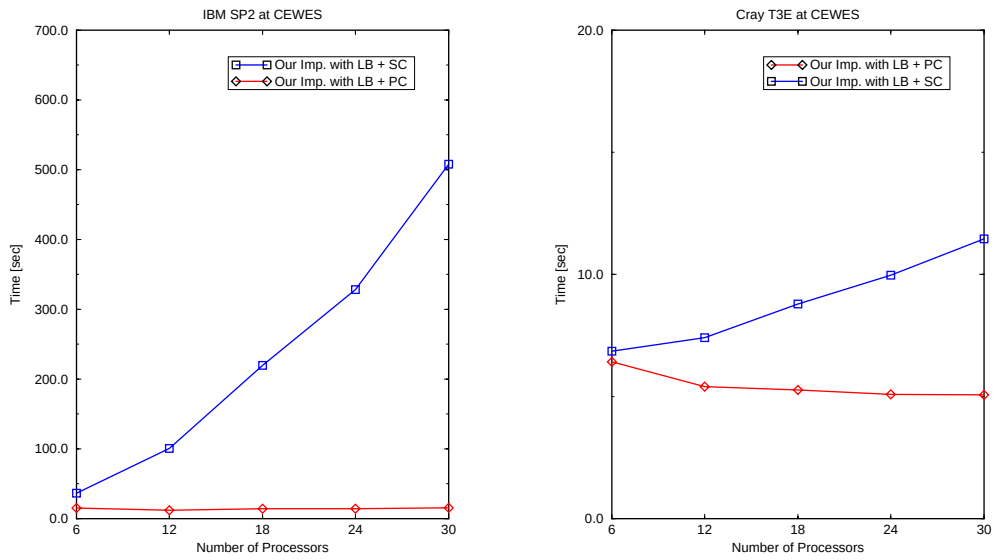


Figure 43: Comparison of communication time

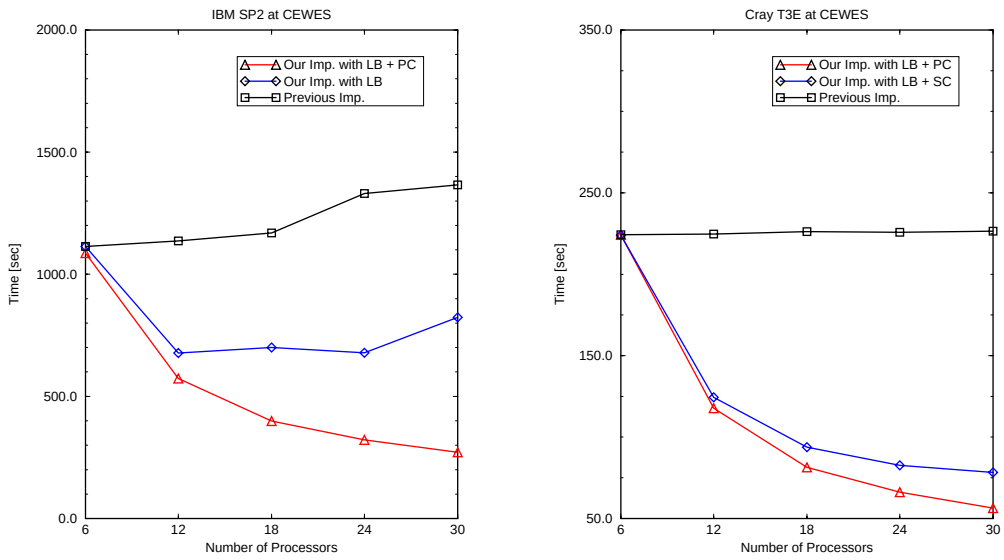


Figure 44: Execution time of our implementation with load balancing and parallel communication

7.4 Communication Performance Prediction Using Our Model

We compared the estimated communication time and the measured communication time for the parallel interprocessor communication. The start-up time (T_s) and the transfer rate (τ_d) obtained using our benchmark experimental results are used to estimate the communication time. Based on the experimental results, the communication time between two processors can be modeled as a function of the message size, m, as follows:

$$CT_{pp} = T_s + m\tau_d \tag{6}$$

where CT_{pp} = point-to-point communication time, T_s = start-up time, and τ_d = transfer rate for a byte.

As described in the previous section, we implemented interprocessor communication in parallel using three steps as shown in Figure 42. Thus, the total communication time for the communication pattern is

$$CT = CT_1 + CT_2 + CT_3 = 3 \times T_s + \sum_{i=1}^{3} m_i \tau_d \tag{7}$$

where CT_i = the communication of step i, m_i = the message size of step i.

The data block size of each step is as follows:

- Step 1: 468 $\times$ 2 Elements
- Step 2: 456 $\times$ 2 Elements
- Step 3: 6 $\times$ 2 Elements

The data type of the element is real. Its size is 4 bytes on the IBM SP and 8 bytes on the Cray T3E. The above data blocks are tranferred to a destination processor within a pair of processors and six processors were used to perform the communication pattern. The comparison is shown in Table 7.

Table 7: Estimated and measured communication times (usec)

IBM SP			Cray T3E		
Estimated	Measured	Error	Estimated	Measured	Error
364.80	323.54	13%	235.80	323.03	27%

8 Acknowlegement

This work was supported in part by the grant of HPC time from the DoD HPC Center, (SP, T3E, and Origin 2000 systems at MSRC CEWES).

References

[1] Analog Devices, Inc. *ADSP-2106x SHARC User's Manual*, First Edition, March 1995.

[2] Ed Anderson, Jeff Brooks and Tom Hewitt, Benchmarking Group, Cray Research, "The Benchmarker's Guide to Single-processor Optimization for CRAY T3E Systems," URL: http://www.cray.com/products/systems/crayt3e/benchmark.ps

[3] D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, S. Fineberg, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan, and S. Weeratunga, "The NAS Parallel Benchmarks," RNS Technical Report RNS-94-007, March 1994

[4] L. E. Cannon, "A cellular computer to implement the Kalman filter algorithm," Ph. D. Dissertation, Montana State University, Bozeman, MT, 1969.

[5] I. Carter, W. Hsieh, L. Stoller, M. Swanson, L. Zhang, E. Brunvand, A. Davis, C.-C. Kuo, R. Kuramkote, M. Parker, L. Schaelicke, and T. Tateyama, "Impulse: An Adaptable Memory System", Submitted to the Eighth Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS-8), October 1998.

[6] L. Carter, J. Ferrante and S. F. Hummel, "Hierarchical Tiling for Improved Superscalar Performance," Proceedings of IPPS '95, 1995.

[7] H. J. Curnow and B. A. Wichman, "A synthetic benchmarks," The Computer Journal, Vol. 19, No. 1, p. 80, 1976.

[8] "Cray T3E Series", URL: http://www.cray.com/products/systems/crayt3e/

[9] Jack J. Dongarra, "Performance of Various Computers Using Standard Linear Equations Software, (Linpack Benchmark Report)," University of Tennessee Computer Science Technical Report, CS-89-85, 1998

[10] D. Elliot, M. Stumm, and M. Snelgrove, "Computational RAM: The Case for SIMD Computing in Memory", Workshop on Mixing Logic and DRAM: Chips that Compute and Remember, 24th International Symposium on Computer Architecture, June 1997.

[11] "Embedded HPSCS,"
URL: http://www.sanders.com/hpc/HPSCS/HPSCS.html.

[12] "Embeddable Systems Homepage,"
URL: http://www.ito.darpa.mil/ResearchAreas/Embeddable.html.

[13] R. A. Games, “Benchmarking Methodology for Real-Time Embedded Scalable High Performance Computing,” MITRE Technical Report MTR 96B0000010, March 1996.

[14] J.L. Hennessy and D.A. Patterson, “Computer Architecture: A Quantitative Approach”, Morgan Kaufman, Second Edition, 1996.

[15] IBM Corporation World-Wide Web Page, “RS/6000 Scalable POWERparallel Systems(SP),” http://www.rs6000.ibm.com/hardware/largescale/index.html.

[16] S. Kaxiras, R. Sugumar, and J. Schwarzmeier, “Distributed Vector Architecture: Beyond a Single Vector-IRAM”, Workshop on Mixing Logic and DRAM: Chips that Compute and Remember, 24th International Symposium on Computer Architecture, June 1997.

[17] K. Keeton, R. Arpaci-Dusseau, and D.A. Patterson, “IRAM and SmartSIMM: Overcoming the I/O Bus Bottleneck”, Workshop on Mixing Logic and DRAM: Chips that Compute and Remember, 24th International Symposium on Computer Architecture, June 1997.

[18] P.M. Kogge, J.B. Brockman, T. Sterling, and G. Gao, “Processing In Memory: Chips to Petaflops”, Workshop on Mixing Logic and DRAM: Chips that Compute and Remember, 24th International Symposium on Computer Architecture, June 1997.

[19] S.A. McKee and W.A. Wulf, “Access Ordering and Memory-Conscious Cache Utilization”, First Symposium on High Performance Computer Architecture, January 1995.

[20] L. McLeod and C. McKenney, “Heterogeneous Multicomputing for Cost-Effective Embedded Systems,”
URL: http://www.mc.com/back/backgrl.html

[21] F. M. McMahon, “The Livermore FORTRAN kernels: A computer test of numerical performance range,” Tech. Rep. UCRL-55745, Lawrence Livermore National Laboratory, University of California, Livermore, CA, 1986.

[22] H. Miyajima, K. Inoue, K. Kai, and K. Murakami, “On-chip Memorypath Architectures for Parallel Processing RAM (PPRAM)”, Workshop on Mixing Logic and DRAM: Chips that Compute and Remember, 24th International Symposium on Computer Architecture, June 1997.

[23] P. J. Mucci and K. London, “The MPBench Report,” http://www.cs.utk.edu/ mucci/DOD/mpbench.ps, 1998.

[24] M. Oskin, F.T. Chong, and T. Sherwood, “Active Pages: A Computation Model for Intelligent Memory”, 25th International Symposium on Computer Architecture, June 1998.

[25] D. Patternson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas, and K. Yelick, “A Case for Intelligent RAM: IRAM”, IEEE Micro, April 1997.

[26] D. Patternson and J. L. Hennessy, “Computer Organization & Design: The Hardware/Software Interface,” Morgan kaufmann, 1994.

[27] “The Message Passing Interface Standard,” URL:http://www.mcs.anl.gov/mpi/

[28] C. L. Seitz, “The Two-Level-Multicomputer Project,” The first Myricom Muticomputer User’s Group Meeting, 1996.

[29] “Parallel Basic Linear Algebra Subprograms,” URL:http://www.netlib.org/scalapack/html/pblas_qref.html

[30] “The SCALAPACK Project,” URL:http://www.netlib.org/scalapack/index.html

[31] “SGI Origin2000: The Perfect System for Evolving Compute, Memory, and I/O Requirements”, URL: http://www.sgi.com/origin/2000/

[32] SPEC, URL:http://www.specbench.org/

[33] J. S. Vitter and E. A. M. Shriver, “Algorithms for Parallel Memory, I: Two-Level Memories,” Algorithmica, Vol. 12, pp. 110-147, 1994.

[34] C.-L. Wang, P. B. Bhat, and V. K. Prasanna, “High-Performance Computing for Vision,” Proceedings of the IEEE, Vol. 84, No.7, July, pp. 931-946, 1996.

[35] R. P. Weicker, “Dhrystone: A synthetic systems programming benchmark,” Comm. ACM, Vol. 27, No. 10, p.1013-1030, 1984.

[36] P. Zhong and M. Martonosi, “Using Reconfigurable Hardware to Customize Memory Hierarchies”, SPIE Conference on Reconfigurable Technology for Rapid Product Development and Computing, November 1996.

A Appendix I: Benchmark Codes

In this appendix, our benchmark code are shown.

A.1 Out-of-Cache Memory Communication Code

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>

#ifdef SP2
#define CacheSZ (1024 * 64)
#define CacheLN 128
#endif

#ifdef T3E
#define CacheSZ (1024 * 96)
#define CacheLN 64
#endif

#define NITER   20
#define SITER   0
#define EITER   15

/* The Shell Sort */
void bsort(double *item)
{
  register int i, j, gap, k, count;
  int a[5];
  double x;

  count = NITER;

  a[0]=9; a[1]=5; a[2]=3; a[3]=2; a[4]=1;

  for (k=0; k<5; k++) {
    gap = a[k];
    for (i=gap; i<count; ++i) {
      x = item[i];
      for (j=i-gap; x<item[j] && j>=0; j=j-gap)
item[j+gap] = item[j];
      item[j+gap] = x;
    }
  }
} /* bsort() */



int main(int argc, char* argv[])
{
#ifdef DINT
  register int max;
```

```
  int* pData;
  char Mssg[20] = "Integer Op";
#define ATYPE int
#endif

#ifdef DSING
  register float max;
  float* pData;
  char Mssg[20] = "Float Op";
#define ATYPE float
#endif

#ifdef DDBL
  register double max;
  double* pData;
  char Mssg[20] = "Double Op";
#define ATYPE double
#endif

#define ELEM_CACHE  CacheSZ/sizeof(max)
#define ELEM_LINE   CacheLN/sizeof(max)


  long int STNUM;
  long int ENDNUM;
  long int INTNUM;
  long int NUM;

  int STSTRIDE;
  int ENDSTRIDE;
  int INTSTRIDE;
  int STRIDE;

  int i;
  long int SIZE;
  int rank, count;
  int OPTYPE;
  int RANDHALF;
  int FLUSH[ELEM_CACHE];
  double start, finish, wtime0, wtime1;
  double alltime[NITER];
  double dTotal;

  STNUM = atol(argv[1]);
  ENDNUM = atol(argv[2]);
  INTNUM = atol(argv[3]);
  STSTRIDE = atoi(argv[4]);
  ENDSTRIDE = atoi(argv[5]);
  INTSTRIDE = atoi(argv[6]);
  OPTYPE = atoi(argv[7]);


  RANDHALF = (int) RAND_MAX / 2;
```

```
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  wtime0 = MPI_Wtime();
  wtime1 = MPI_Wtime();
  wtime1 -= wtime0;

  fprintf(stdout,"%s ", Mssg);
  switch (OPTYPE) {

  case 1: /* read */
    fprintf(stdout,"Read Test:\n");
    break;
  case 2: /* write */
    fprintf(stdout,"Write Test:\n");
    break;
  case 3: /* multiply with register */
    fprintf(stdout,"Multiply Test:\n");
    break;
  case 4: /* divide with register */
    fprintf(stdout,"Divide Test:\n");
    break;

  }

  fprintf(stdout,"Cache Size is %d, Cache Line size is %d\n", CacheSZ, CacheLN);
  fprintf(stdout,"Elements/Cache = %d, Elements/CacheLine = %d\n",ELEM_CACHE, ELEM_LINE);
  fprintf(stdout,"NUM = %d, %d, %d\n", STNUM,ENDNUM,INTNUM);
  fprintf(stdout,"Stride = %d, %d, %d\n", STSTRIDE, ENDSTRIDE, INTSTRIDE);
  fprintf(stdout,"MPI_Wtime=%f usec\n",wtime1*1000000);

  for (NUM = STNUM; NUM <= ENDNUM; NUM += INTNUM) {
    fprintf(stdout,"$\n");

    for (STRIDE = STSTRIDE; STRIDE <= ENDSTRIDE; STRIDE += INTSTRIDE) {

      SIZE = NUM*STRIDE;
      pData = malloc(SIZE*sizeof(max));

      /* Put random data in data array */
      for (count = 0; count < NUM; count++)
        *(pData + count*STRIDE) = (ATYPE) ( rand() > RANDHALF ? 3 : 2);

      for (i=0; i<NITER; i++) {

/* Flush the Cache with dummy data */
for (count = 0; count < ELEM_CACHE; count+=ELEM_LINE)
  *(FLUSH + count) = rand();

max = 0;

switch (OPTYPE) {
```

```
case 1: /* read */
  start = MPI_Wtime();
  for(count = 0; count < NUM; count++)
    max += *(pData + count*STRIDE);
  finish = MPI_Wtime();
  break;
case 2: /* write */
  start = MPI_Wtime();
  for(count = 0; count < NUM; count++)
    *(pData + count*STRIDE) = max++;
  finish = MPI_Wtime();
  break;
case 3: /* multiply with register */
  start = MPI_Wtime();
  for(count = 0; count < NUM; count++)
    max *= *(pData + count*STRIDE);
  finish = MPI_Wtime();
  break;
case 4: /* Divide with register */
  max = (ATYPE) 100000;
  start = MPI_Wtime();
  for(count = 0; count < NUM; count++)
    max /= *(pData + count*STRIDE);
  finish = MPI_Wtime();
  break;

} /* Operation Type Case */

alltime[i] = (finish - start -wtime1)*1000000 ;

      } /* Each iteration */

      bsort(alltime);
      dTotal = 0;

/*       for (i=0; i< NITER; i++) */
/*  printf("%f\n", alltime[i]); */

      for (i=SITER; i< EITER; i++)
dTotal += alltime[i];

      dTotal /= (EITER-SITER);

      fprintf(stdout,"%f usec, max = %f \n",
      dTotal, (double) max);
      fflush(stdout);
      free(pData);
/*       printf("*************************\n"); */
    } /* Various strides */
  } /* Number of elements */

  MPI_Finalize();
```

```
}
```

A.2 Permutation Communication Code

```
/*****************************************************************/
/*   permutation.c: measure permutation communication time         */
/*                      By Jinwoo Suh                              */
/*****************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include <sys/time.h>
#include <mpi.h>

#define nloops      20
#define NumP        1024
#define INT         sizeof(int)
#define MAX_DATA_SIZE         16777216

double max_data();
void print_min_data();

main(int argc, char *argv[])
{
        MPI_Group     MPI_GROUP_WORLD;
        MPI_Comm      COMM;
        MPI_Group     Group;
        MPI_Request   req;
        MPI_Status    status;

        int           TotalP, myrank, id, rc, ranks[NumP];
        int           tag, dist, dist2;
        int           run, loop, src;
        int           i, j, k, l, m, n, cnt[10], dst;
        int           size, sz, idx;
        void          *in, *out;
        double        ST[128][nloops], ET[128][nloops], max, sum, min;
        double        TT[nloops], PT[nloops];

        /*---------------------------------------------------------------*/
        /*    MPI Initialize                                             */
        /*---------------------------------------------------------------*/
        rc  = MPI_Init (&argc, &argv);
        rc |= MPI_Comm_size (MPI_COMM_WORLD, &TotalP);
        rc |= MPI_Comm_rank (MPI_COMM_WORLD, &myrank);
        if (rc != 0)
                fprintf (stderr, "error init MPI and otaining task ID info\n");
        MPI_Comm_group (MPI_COMM_WORLD, &MPI_GROUP_WORLD);

        for (i=0; i<TotalP; ++i) ranks[i]=i;
        MPI_Group_incl (MPI_GROUP_WORLD,TotalP,ranks,&Group);
        MPI_Comm_create (MPI_COMM_WORLD,Group,&COMM);
        MPI_Comm_rank (COMM, &id);
```

```
in = (void *)malloc(MAX_DATA_SIZE);
out = (void *)malloc(MAX_DATA_SIZE);

if (id == 0) {
        fprintf (stdout,"\n==============================\n");
        fflush  (stdout);
}

for(dist=1; dist<=TotalP/2; dist++){
        if (id == 0) {
                fprintf (stdout,"dist=%d\n", dist);
                fflush  (stdout);
        }

        for(size=4; size<=MAX_DATA_SIZE; size *=2 ){

                MPI_Barrier (COMM);

                src = (TotalP+id-dist)%TotalP;
                dst = (id+dist)%TotalP;

                for (loop=0; loop<nloops; ++loop) {
                        MPI_Barrier (COMM);
                        ST[0][loop] = MPI_Wtime ();
                        MPI_Isend (out, size, MPI_BYTE, dst, 100,
                                COMM,&req);
                        MPI_Recv (in, size, MPI_BYTE, src, 100,
                                COMM,&status);
                        ET[0][loop] = MPI_Wtime ();
                } /* loop */

                /*---- calculate time ---------------------------*/

                MPI_Barrier (COMM);

                if (id==0) {
                        for(i=1; i<TotalP; i++){
                                MPI_Recv(&ET[i][0],nloops,MPI_DOUBLE,i,
                                        100,COMM,&status);
                                MPI_Recv(&ST[i][0],nloops,MPI_DOUBLE,i,
                                        100,COMM,&status);
                        }
                        for(j=0; j<nloops; j++) {
                                for(i=0; i<TotalP; i++)
                                        TT[i]=(ET[(i+dist)%TotalP][j]
                                                -ST[i][j])*1000;
                                PT[j]= max_data(TT,TotalP);
                        }
                        print_min_data(PT,nloops);
                }
                else {
                        MPI_Send(ET,nloops,MPI_DOUBLE,0,100,COMM);
                        MPI_Send(ST,nloops,MPI_DOUBLE,0,100,COMM);
```

```
                        }

                }
        }
        MPI_Group_free (&Group);
        MPI_Comm_free  (&COMM);
        MPI_Finalize ();
}
double max_data(double *TT,int cnt)
{
        int i,j;
        double max,min,sum, t1,t2;

        t1 = MPI_Wtime ();
        t2 = MPI_Wtime ();
        t2 -= t1;
        max = TT[0];
        for(i=1; i<cnt; i++){
                if( TT[i] > max ) max = TT[i];
        }
        return(max-t2);
}
void print_min_data(double *TT, int cnt)
{
        int i,j;
        double max,min,sum, t1,t2;

        min = TT[0];
        for(i=1; i<cnt; i++){
                if( TT[i] < min ) min = TT[i];
        }
        fprintf (stdout, "%lf\n", min);
        fflush(stdout);
}
```

A.3 Pingpong Communication Code

```
/*****************************************************************/
/*   pingpong.c: measure pingpong communication time              */
/*                      By Jinwoo Suh                            */
/*****************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include <sys/time.h>
#include <mpi.h>

#define nloops       20
#define NumP         1024
#define MAX_DATA_SIZE   16777216
#define INT          sizeof(int)

double print_min_data();

main(int argc, char *argv[])
{
MPI_Group     MPI_GROUP_WORLD;
MPI_Comm      COMM;
MPI_Group     Group;
MPI_Request   req;
MPI_Status    status;

int           TotalP, myrank, id, rc, ranks[NumP];
int           tag, dist, dist2;
int           run, loop, src, dst;
int           i, j, k, l, m, n;
int           size, sz, idx;
void       *in, *out;
double        ST[nloops], ET[nloops],max,sum,min;
double        TT[nloops];

/*---------------------------------------------------------------*/
/*    MPI Initialize                                             */
/*---------------------------------------------------------------*/
rc  = MPI_Init (&argc, &argv);
rc |= MPI_Comm_size (MPI_COMM_WORLD, &TotalP);
rc |= MPI_Comm_rank (MPI_COMM_WORLD, &myrank);
if (rc != 0)
fprintf (stderr, "error init MPI and otaining task ID info\n");
MPI_Comm_group (MPI_COMM_WORLD, &MPI_GROUP_WORLD);

for (i=0; i<TotalP; ++i) ranks[i]=i;
MPI_Group_incl (MPI_GROUP_WORLD,TotalP,ranks,&Group);
MPI_Comm_create (MPI_COMM_WORLD,Group,&COMM);
MPI_Comm_rank (COMM, &id);

in = (void *)malloc(MAX_DATA_SIZE);
```

```
out = (void *)malloc(MAX_DATA_SIZE);

for (src=0; src<TotalP-1; src++) {
for (dst=src+1; dst<TotalP; dst++) {
MPI_Barrier (COMM);
if (id == 0) {
fprintf (stdout,"src,dst=%d,%d\n", src,dst);
fflush  (stdout);
}

for(size=4; size<=MAX_DATA_SIZE; size *=2 ){
    for (loop=0; loop<nloops; ++loop) {
MPI_Barrier (COMM);
if(id==src) {
ST[loop] = MPI_Wtime ();
MPI_Send (out, size, MPI_BYTE, dst, 100,
COMM);
MPI_Recv (in, size, MPI_BYTE, dst, 100,
COMM,&status);
ET[loop] = MPI_Wtime ();
}
else if(id==dst) {
MPI_Recv (in, size, MPI_BYTE, src, 100,
COMM,&status);
MPI_Send (out, size, MPI_BYTE, src, 100,
COMM);
}
    } /* loop */

    /*---- calculate time ---------------------------*/
    if( id==src) {
for(j=0; j<nloops; j++)
TT[j]=(ET[j]-ST[j]) *1000;
print_min_data(TT,nloops);
            }
        }
}
}

MPI_Barrier (COMM);
MPI_Group_free (&Group);
MPI_Comm_free  (&COMM);
MPI_Finalize ();
}
double print_min_data(double *TT,int cnt)
{
int i,j;
double max,min,sum, t1,t2;

t1 = MPI_Wtime ();
t2 = MPI_Wtime ();
t2 -= t1;
```

```
min = TT[0];
for(i=1; i<cnt; i++)
if( TT[i] < min ) min = TT[i];
fprintf(stdout, "%lf\n", min-t2);
fflush(stdout);
}
```

A.4 Scatter Communication Code

```
/*****************************************************************/
/*   scatter.c: measure scatter communication time                */
/*                      By Jinwoo Suh                             */
/*****************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include <sys/time.h>
#include <mpi.h>

#define nloops          20
#define NumP            1024
#define MAX_DATA_SIZE   16777216
#define INT             sizeof(int)

double print_min_data();

main(int argc, char *argv[])
{
        MPI_Group     MPI_GROUP_WORLD;
        MPI_Comm      COMM;
        MPI_Group     Group;
        MPI_Request   req;
        MPI_Status    status;

        int           TotalP, myrank, id, rc, ranks[NumP];
        int           tag, dist, dist2;
        int           run, loop, src, dst;
        int           i, j, k, l, m, n;
        int           size, sz, idx;
        void          *in, *out;
        double        ST[nloops], ET[nloops],max,sum,min;
        double        TT[NumP][nloops],TTT[nloops];

        /*-----------------------------------------------------------*/
        /*    MPI Initialize                                         */
        /*-----------------------------------------------------------*/
        rc  = MPI_Init (&argc, &argv);
        rc |= MPI_Comm_size (MPI_COMM_WORLD, &TotalP);
        rc |= MPI_Comm_rank (MPI_COMM_WORLD, &myrank);
        if (rc != 0)
                fprintf (stderr, "error init MPI and otaining task ID info\n");
        MPI_Comm_group (MPI_COMM_WORLD, &MPI_GROUP_WORLD);

        for (i=0; i<TotalP; ++i) ranks[i]=i;
        MPI_Group_incl (MPI_GROUP_WORLD,TotalP,ranks,&Group);
        MPI_Comm_create (MPI_COMM_WORLD,Group,&COMM);
        MPI_Comm_rank (COMM, &id);

        in = (void *)malloc(MAX_DATA_SIZE);
```

```
        out = (void *)malloc(MAX_DATA_SIZE);

        for (src=0; src<TotalP; src++) {
                        MPI_Barrier (COMM);
                        if (id == 0) {
                                fprintf (stdout,"src=%d\n", src);
                                fflush  (stdout);
                        }

                        for(size=4; size<=MAX_DATA_SIZE/TotalP; size *=2 ){
                            for (loop=0; loop<nloops; ++loop) {
                                MPI_Barrier (COMM);
                                ST[loop] = MPI_Wtime ();
                                MPI_Scatter (out, size, MPI_BYTE, in, size,
                                        MPI_BYTE, src, COMM);
                                ET[loop] = MPI_Wtime ();
                            } /* loop */

                            /*---- calculate time ---------------------------*/
                            if( id==src) {
                                for(i=0; i<TotalP; i++)
                                    if(i != src)
                                        MPI_Recv(&(TT[i][0]),nloops*sizeof(float),
                                                MPI_BYTE,i,100,COMM,&status);
                                    else
                                        for(j=0; j<nloops; j++)
                                                TT[i][j] = ET[j];
                                for(i=0; i<nloops; i++)
                                        for(j=0; j<TotalP; j++)
                                                TT[j][i] = TT[j][i] - ST[i];
                                for(i=0; i<nloops; i++) {
                                        max = TT[0][i];
                                        for(j=1; j<TotalP; j++)
                                                if(TT[j][i]>max)
                                                        max=TT[j][i];
                                        TTT[i]=max*1000; /* msec */
                                }
                                print_min_data(TTT,nloops);
                            }
                            else {
                                MPI_Send (ET, nloops*sizeof(float), MPI_BYTE,
                                        src, 100, COMM);
                            }
                        }
        }

        MPI_Barrier (COMM);
        MPI_Group_free (&Group);
        MPI_Comm_free  (&COMM);
        MPI_Finalize ();
}
double print_min_data(double *TT,int cnt)
{
```

```
        int i,j;
        double max,min,sum, t1,t2;

        t1 = MPI_Wtime ();
        t2 = MPI_Wtime ();
        t2 -= t1;

        min = TT[0];
        for(i=1; i<cnt; i++)
                if( TT[i] < min ) min = TT[i];
        fprintf(stdout, "%lf\n", min-t2);
        fflush(stdout);
}
```

A.5 Broadcast Communication Code

```
/*****************************************************************/
/*  broadcast.c: measure broadcast communication time             */
/*                      By Jinwoo Suh                             */
/*****************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include <sys/time.h>
#include <mpi.h>

#define nloops          20
#define NumP            1024
#define MAX_DATA_SIZE   16777216
#define INT             sizeof(int)

double print_min_data();

main(int argc, char *argv[])
{
        MPI_Group     MPI_GROUP_WORLD;
        MPI_Comm      COMM;
        MPI_Group     Group;
        MPI_Request   req;
        MPI_Status    status;

        int           TotalP, myrank, id, rc, ranks[NumP];
        int           tag, dist, dist2;
        int           run, loop, src, dst;
        int           i, j, k, l, m, n;
        int           size, sz, idx;
        void          *in, *out;
        double        ST[nloops], ET[nloops],max,sum,min;
        double        TT[NumP][nloops],TTT[nloops];

        /*-----------------------------------------------------------*/
        /*    MPI Initialize                                         */
        /*-----------------------------------------------------------*/
        rc  = MPI_Init (&argc, &argv);
        rc |= MPI_Comm_size (MPI_COMM_WORLD, &TotalP);
        rc |= MPI_Comm_rank (MPI_COMM_WORLD, &myrank);
        if (rc != 0)
                fprintf (stderr, "error init MPI and otaining task ID info\n");
        MPI_Comm_group (MPI_COMM_WORLD, &MPI_GROUP_WORLD);

        for (i=0; i<TotalP; ++i) ranks[i]=i;
        MPI_Group_incl (MPI_GROUP_WORLD,TotalP,ranks,&Group);
        MPI_Comm_create (MPI_COMM_WORLD,Group,&COMM);
        MPI_Comm_rank (COMM, &id);

        in = (void *)malloc(MAX_DATA_SIZE);
```

```
        out = (void *)malloc(MAX_DATA_SIZE);

        for (src=0; src<TotalP; src++) {
                        MPI_Barrier (COMM);
                        if (id == 0) {
                                fprintf (stdout,"src=%d\n", src);
                                fflush  (stdout);
                        }

                        for(size=4; size<=MAX_DATA_SIZE/TotalP; size *=2 ){
                            for (loop=0; loop<nloops; ++loop) {
                                MPI_Barrier (COMM);
                                ST[loop] = MPI_Wtime ();
                                MPI_Bcast (out, size, MPI_BYTE, src, COMM);
                                ET[loop] = MPI_Wtime ();
                            } /* loop */

                            /*---- calculate time ---------------------------*/
                            if( id==src) {
                                for(i=0; i<TotalP; i++)
                                    if(i != src)
                                        MPI_Recv(&(TT[i][0]),nloops*sizeof(float),
                                                MPI_BYTE,i,100,COMM,&status);
                                    else
                                        for(j=0; j<nloops; j++)
                                                TT[i][j] = ET[j];
                                for(i=0; i<nloops; i++)
                                        for(j=0; j<TotalP; j++)
                                                TT[j][i] = TT[j][i] - ST[i];
                                for(i=0; i<nloops; i++) {
                                        max = TT[0][i];
                                        for(j=1; j<TotalP; j++)
                                                if(TT[j][i]>max)
                                                        max=TT[j][i];
                                        TTT[i]=max*1000; /* msec */
                                }
                                print_min_data(TTT,nloops);
                            }
                            else {
                                MPI_Send (ET, nloops*sizeof(float), MPI_BYTE,
                                        src, 100, COMM);
                            }
                        }
        }

        MPI_Barrier (COMM);
        MPI_Group_free (&Group);
        MPI_Comm_free  (&COMM);
        MPI_Finalize ();
}
double print_min_data(double *TT,int cnt)
{
        int i,j;
```

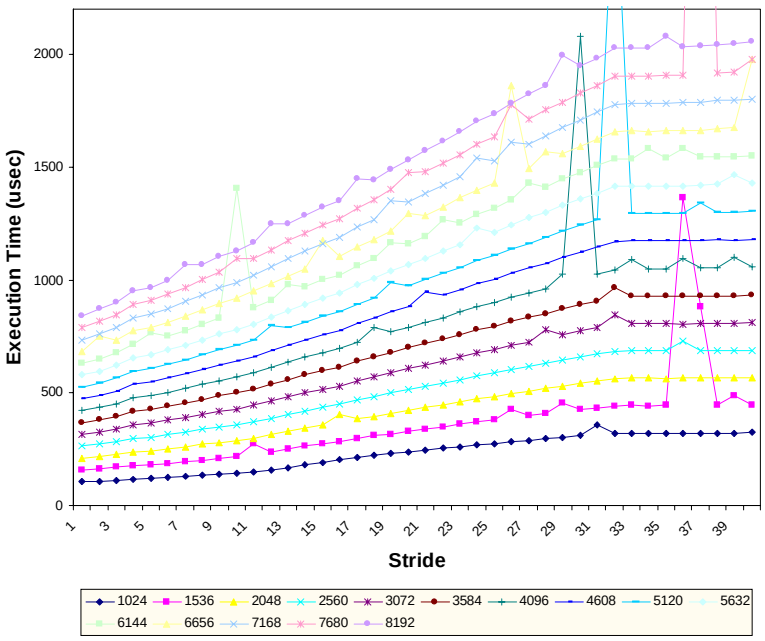
```
        double max,min,sum, t1,t2;

        t1 = MPI_Wtime ();
        t2 = MPI_Wtime ();
        t2 -= t1;

        min = TT[0];
        for(i=1; i<cnt; i++)
                if( TT[i] < min ) min = TT[i];
        fprintf(stdout, "%lf\n", min-t2);
        fflush(stdout);
}
```

A.6 Disk Operation Code

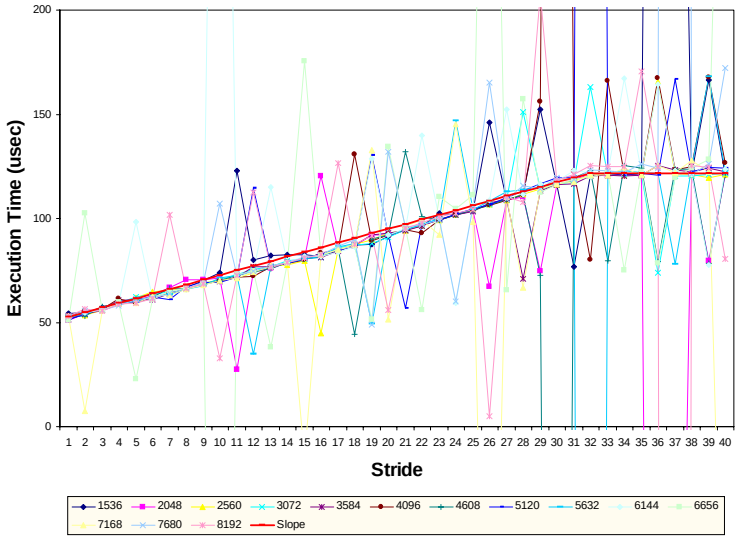
```
/*****************************************************************/
/*   disk.c: measure disk read and write time                    */
/*                      By Jinwoo Suh                            */
/*****************************************************************/
#include <stdio.h>
#include <fcntl.h>
#include <time.h>
#include <mpi.h>

#define N 8388608 /* Data size range*/
#define M 131072 /* Data size range*/
#define NumP 1024

char    s[5],d[N];

main(int argc, char *argv[])
{
        MPI_Group     MPI_GROUP_WORLD;
        MPI_Comm      COMM;

        int     TotalP, myrank, id, rc, ranks[NumP];
        long    data_size,i,j,sum = 0;
        int     fd, fe, rep;
        double ST[1000], MT[1000], NT[1000], ET[1000];

        /*-----------------------------------------------------------*/
        /*    MPI Initialize                                         */
        /*-----------------------------------------------------------*/
        rc  = MPI_Init (&argc, &argv);
        rc |= MPI_Comm_size (MPI_COMM_WORLD, &TotalP);
        rc |= MPI_Comm_rank (MPI_COMM_WORLD, &myrank);
        if (rc != 0)
                fprintf (stderr, "error init MPI and otaining task ID info\n");
        MPI_Comm_group (MPI_COMM_WORLD, &MPI_GROUP_WORLD);

        printf("size of char = %d\n",(int)sizeof(char)); fflush(stdout);

        for(rep=0,data_size=N; data_size>=M; rep++,data_size-=M) {
                for(i=0; i<data_size; i++)
                        d[i]=rand()/10000;
                fd = creat("tmp",0777);

                ST[rep] = MPI_Wtime ();
                write(fd, d, data_size);
                MT[rep] = MPI_Wtime ();

                close(fd);

                for(i=0; i<data_size; i++)
                        d[i]=rand()/10000;
```

```
                fd = open("tmp",O_RDWR,0777);

                NT[rep] = MPI_Wtime ();
                read(fd, d, data_size);
                ET[rep] = MPI_Wtime ();
         for(i=sum=0; i<data_size; i++)
sum += d[i];
                if (sum<0)
printf(" ");
        }

        for(i=0; i<rep; i++) {
                printf("%f msec\n",(float)(MT[i] - ST[i])*1000);
                fflush(stdout);
        }

        printf("READ\n"); fflush(stdout);

        for(i=0; i<rep; i++) {
                printf("%f msec\n",(float)(ET[i] - NT[i])*1000);
                fflush(stdout);
        }
}
```

B Appendix II: Detailed Results

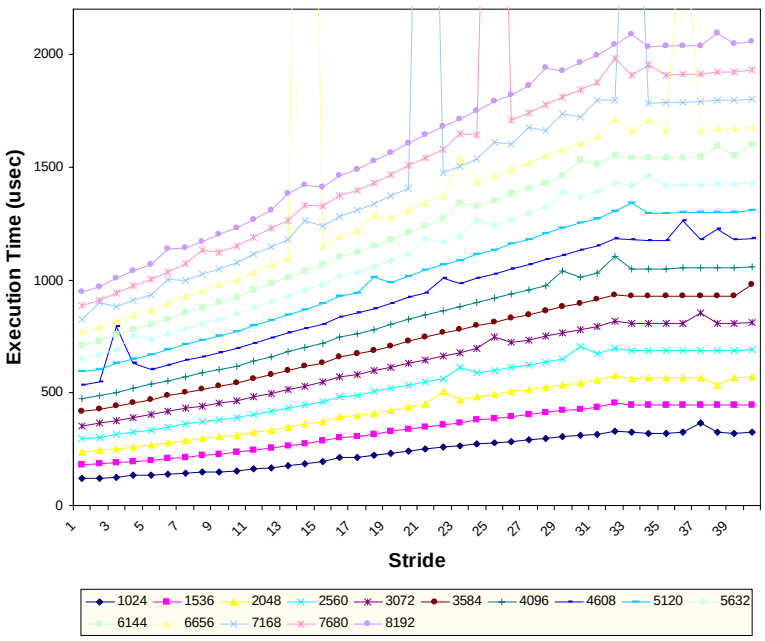


(a) Read Operation

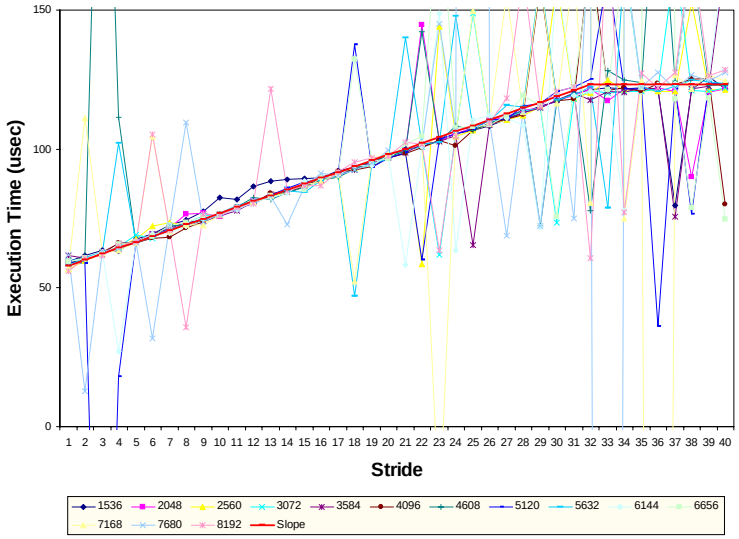


(b) Stride Comparison

Figure 45: Processor-Memory communication: Read Integer

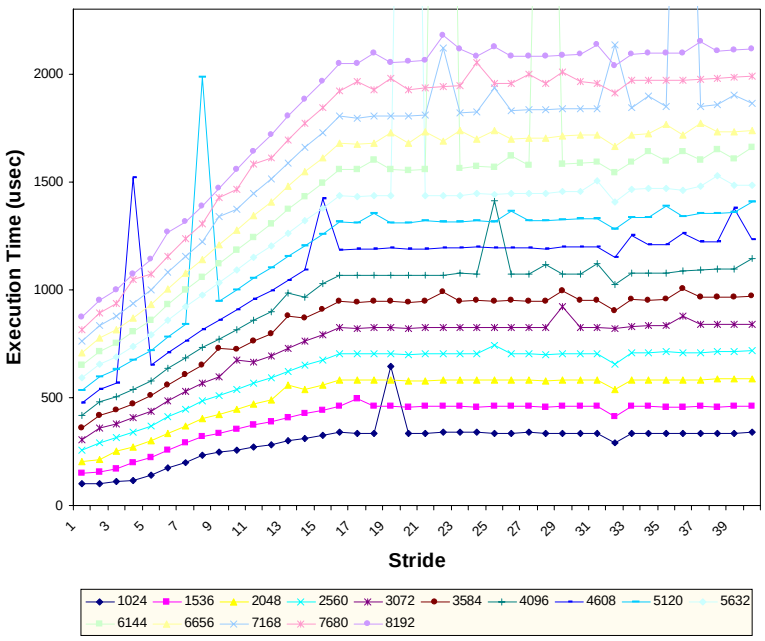


(a) Read Operation

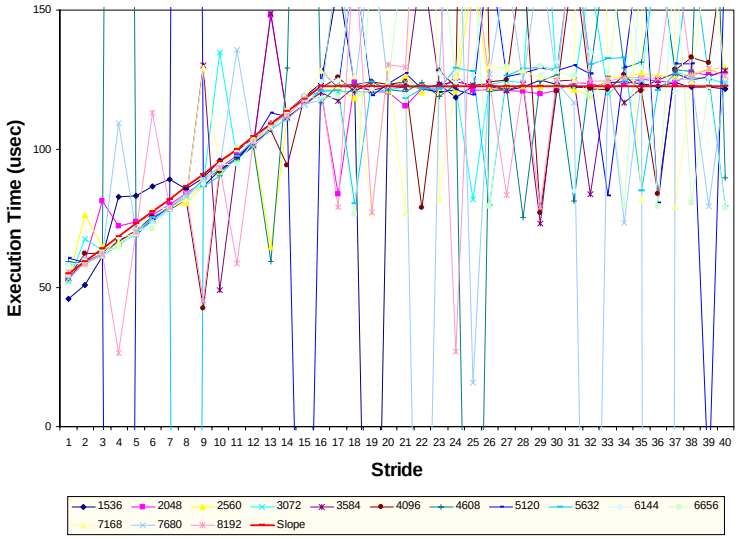


(b) Stride Comparison

Figure 46: Processor-Memory communication: Read Single

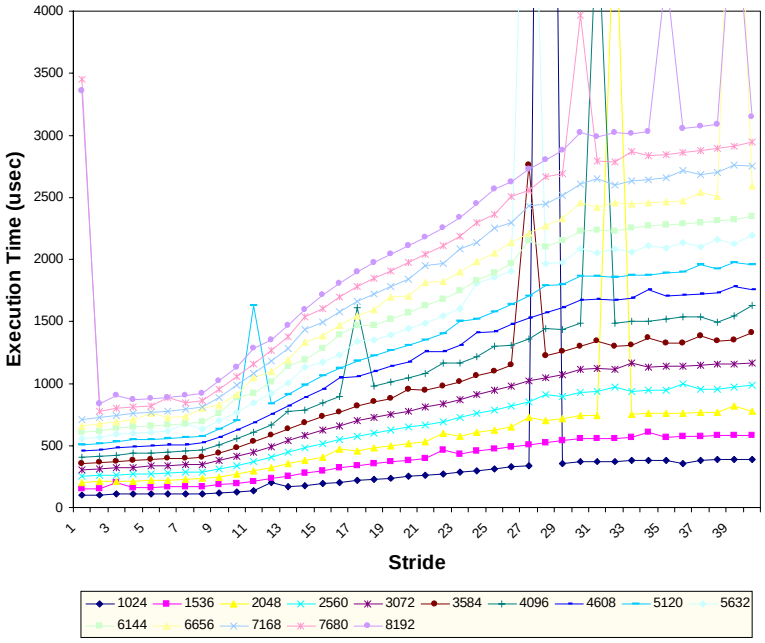


(a) Read Operation

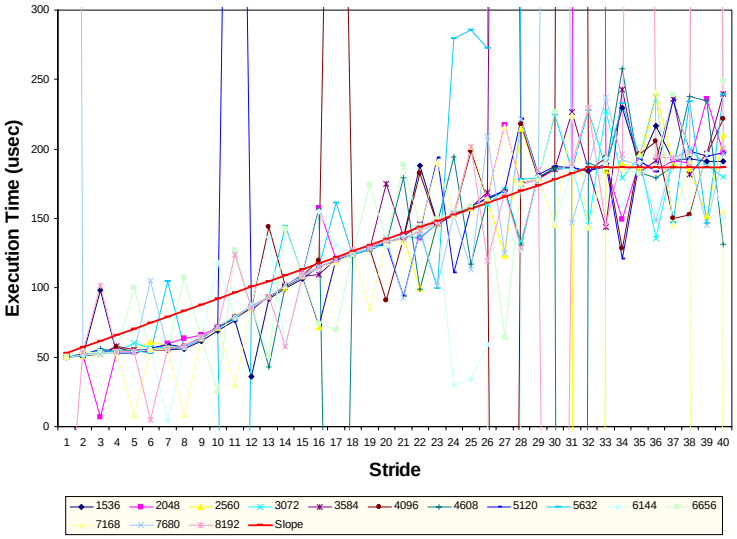


(b) Stride Comparison

Figure 47: Processor-Memory communication: Read Double

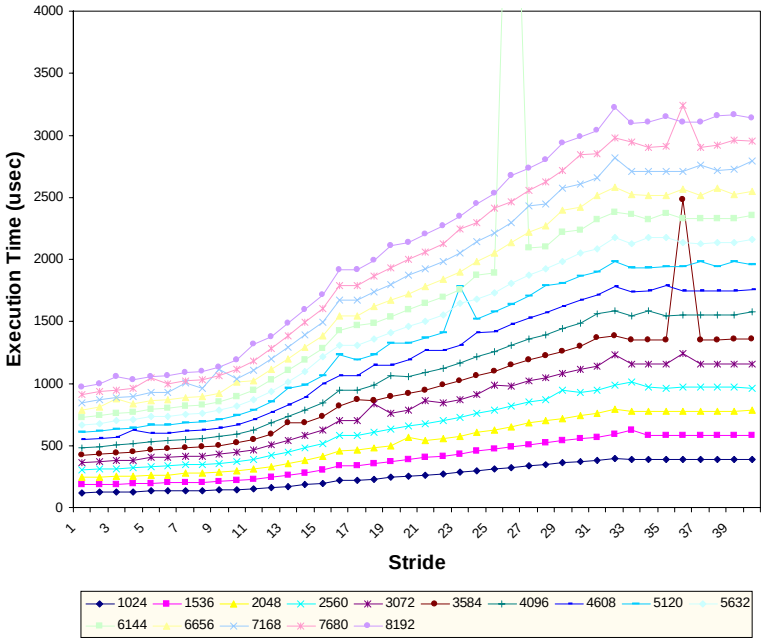


(a) Write Operation

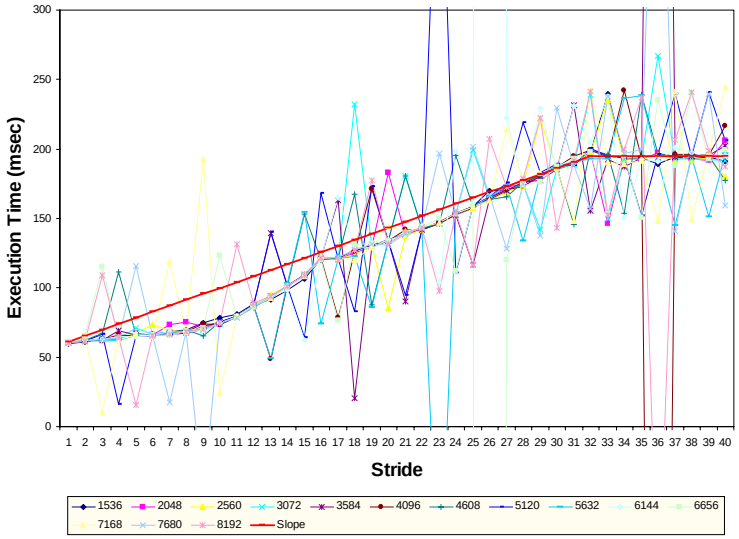


(b) Stride Comparison

Figure 48: Processor-Memory communication: Write Integer

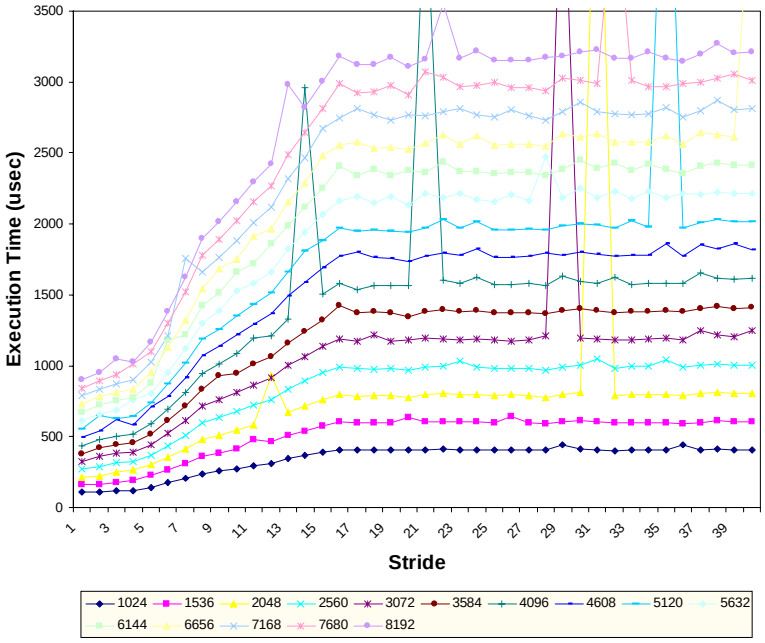


(a) Write Operation

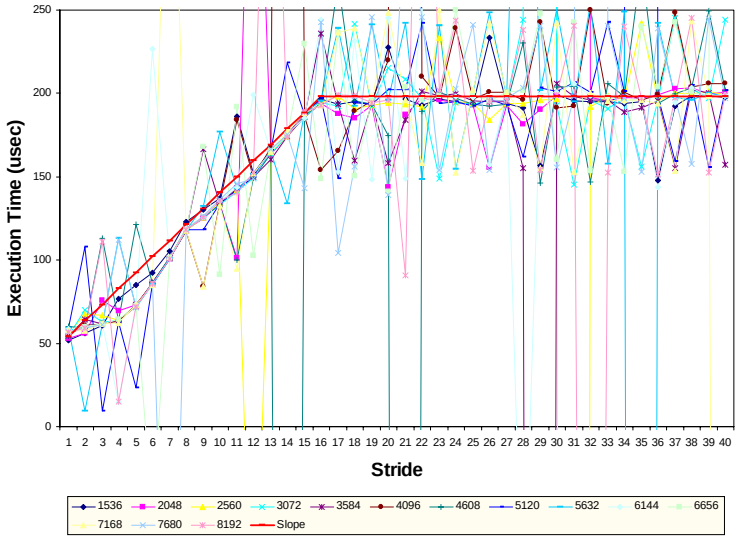


(b) Stride Comparison

Figure 49: Processor-Memory communication: Write Single

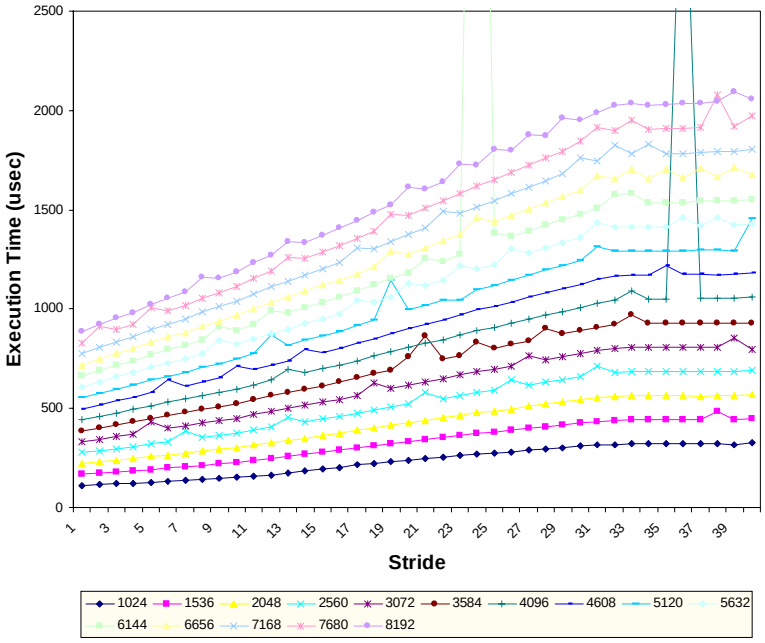


(a) Write Operation

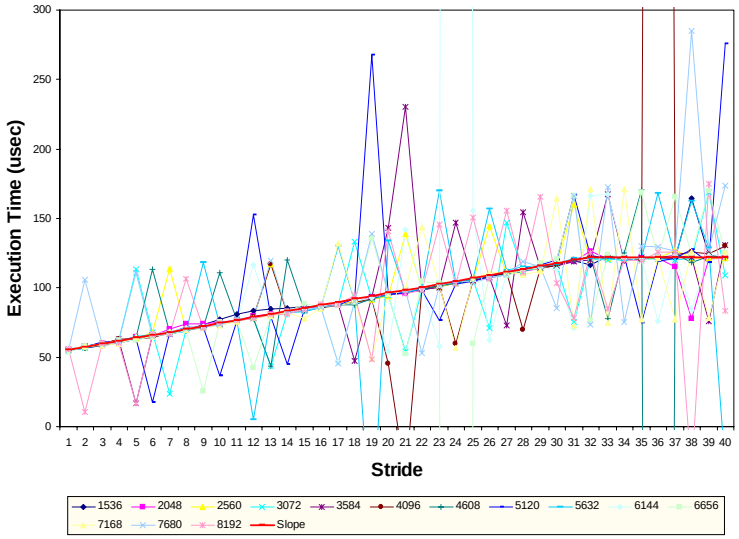


(b) Stride Comparison

Figure 50: Processor-Memory communication: Write Double

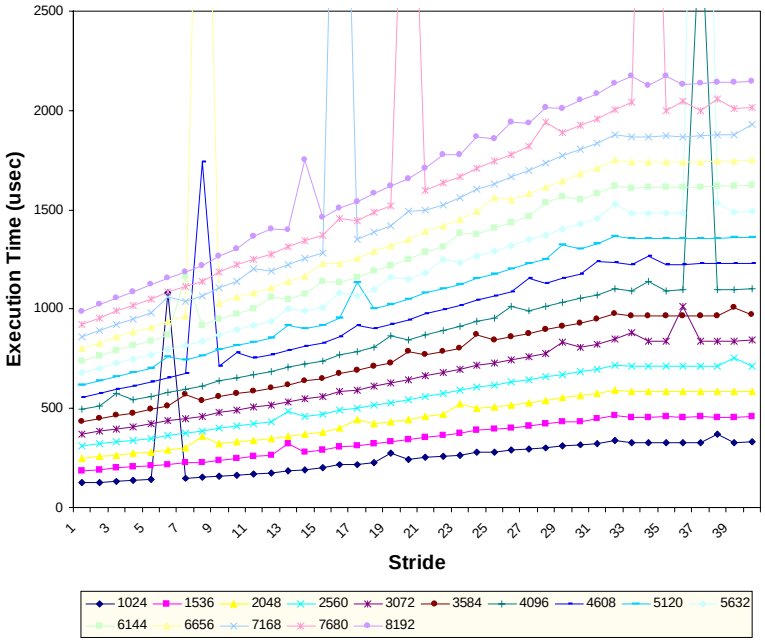


(a) Multiply Operation

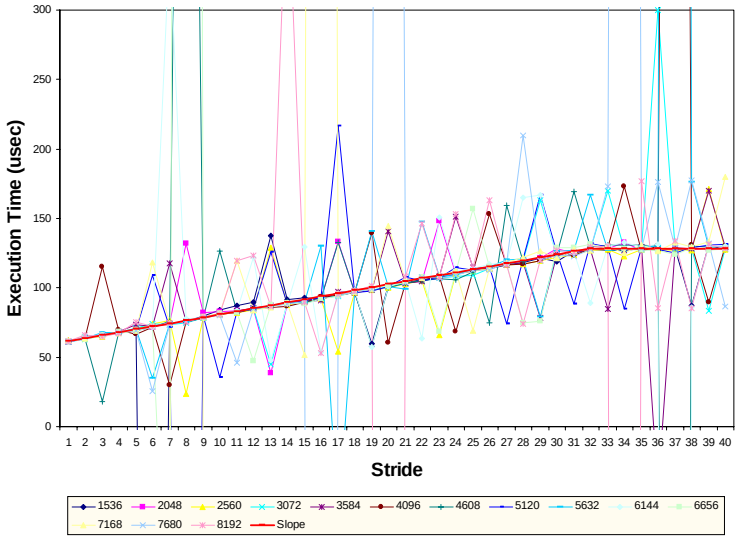


(b) Stride Comparison

Figure 51: Processor-Memory communication: Multiply Integer

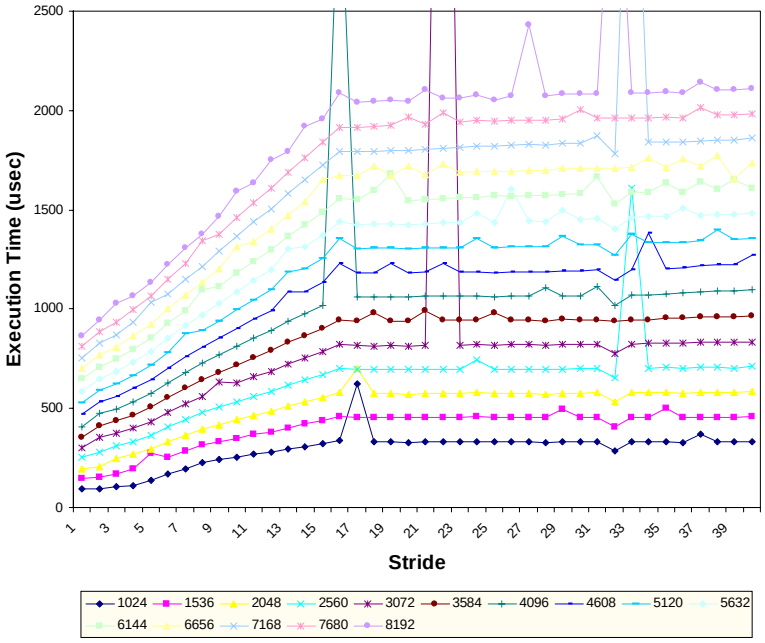


(a) Multiply Operation

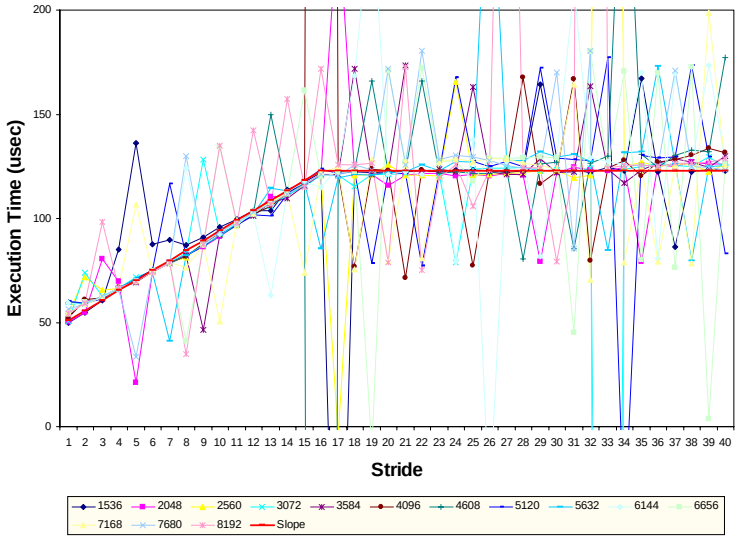


(b) Stride Comparison

Figure 52: Processor-Memory communication: Multiply Single

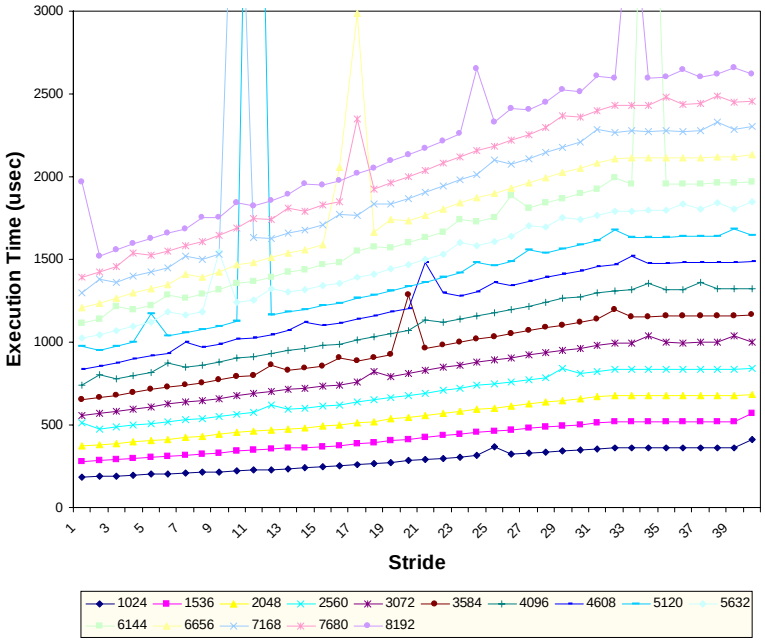


(a) Multiply Operation

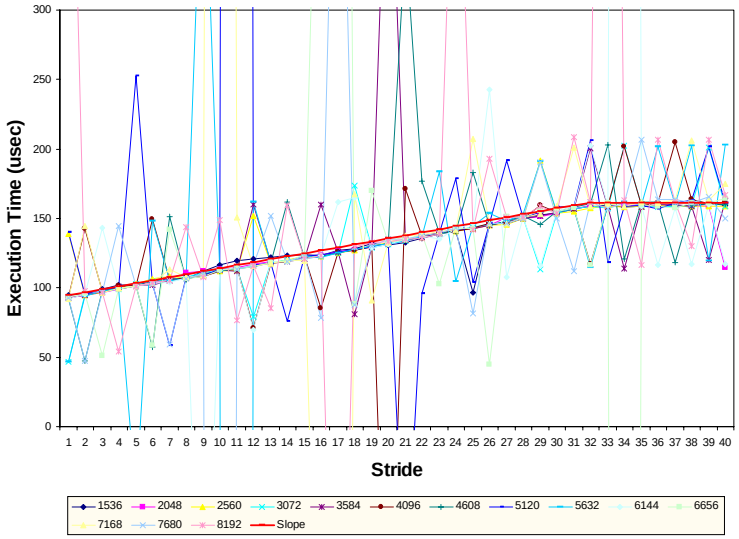


(b) Stride Comparison

Figure 53: Processor-Memory communication: Multiply Double

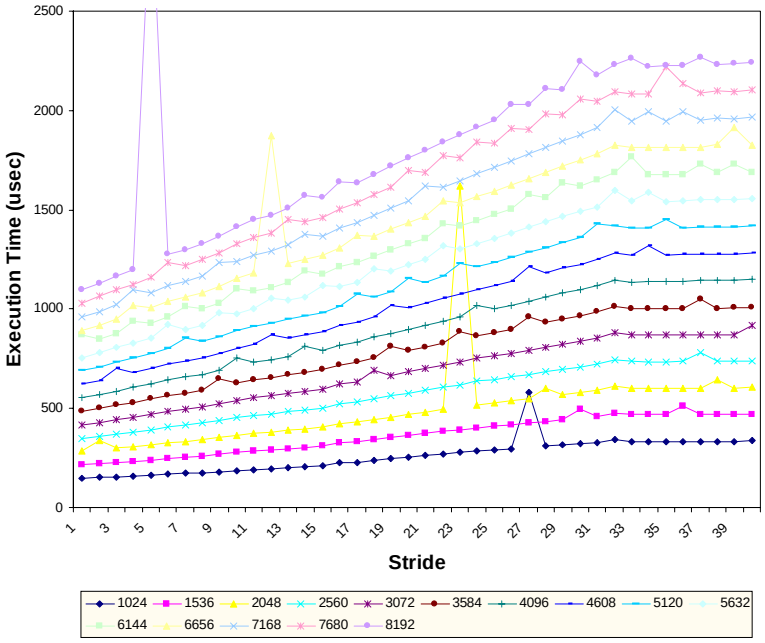


(a) Divide Operation

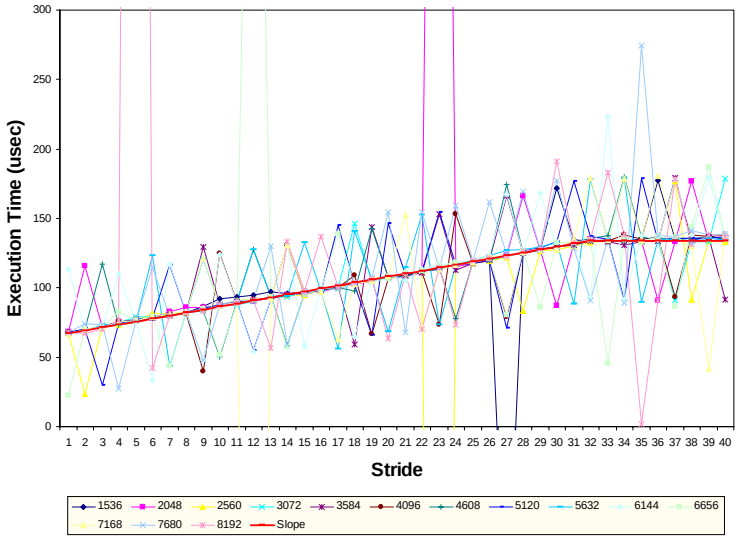


(b) Stride Comparison

Figure 54: Processor-Memory communication: Divide Integer

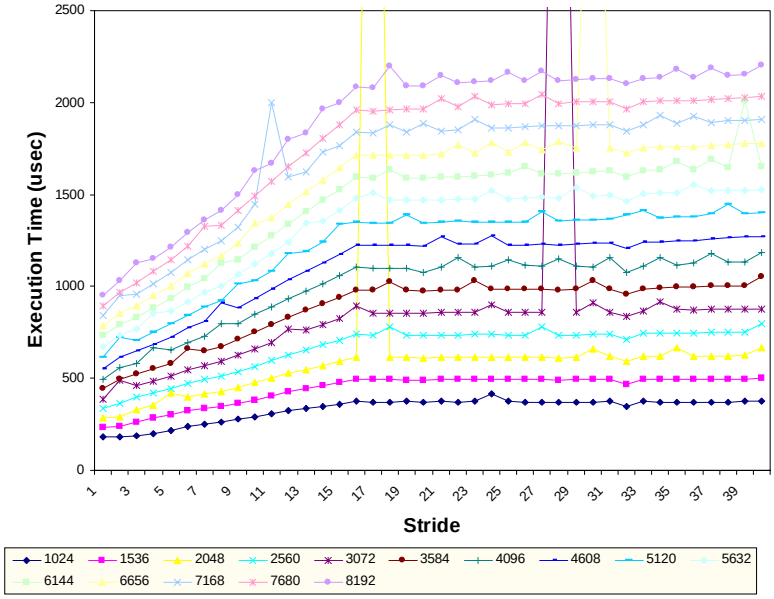


(a) Divide Operation

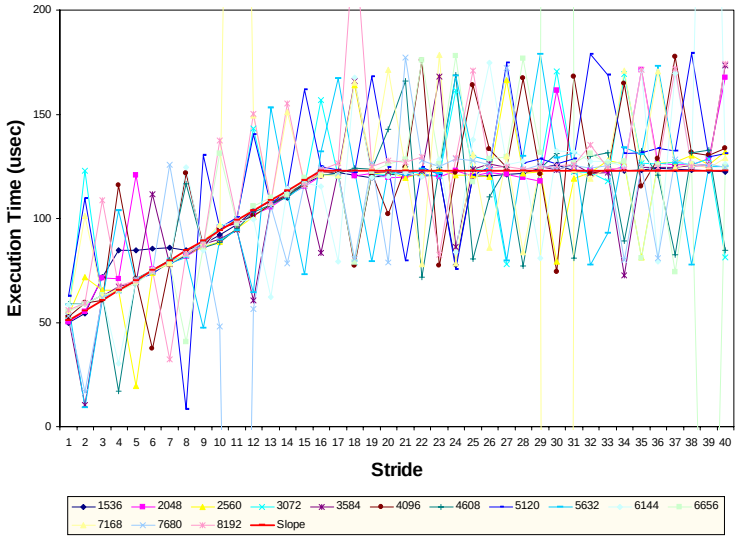


(b) Stride Comparison

Figure 55: Processor-Memory communication: Divide Single



(a) Divide Operation



(b) Stride Comparison

Figure 56: Processor-Memory communication: Divide Double

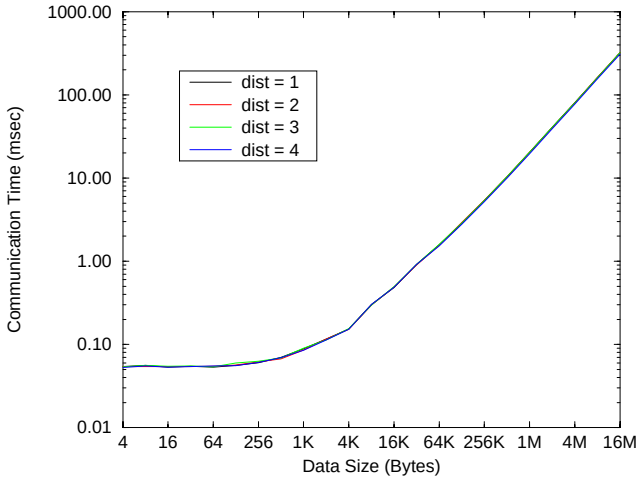


Figure 57: Permutation communication results using 8 processors on SP

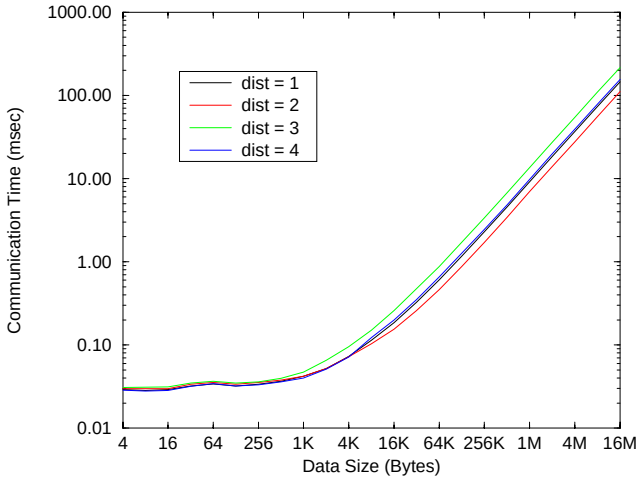


Figure 58: Permutation communication results using 8 processors on T3E

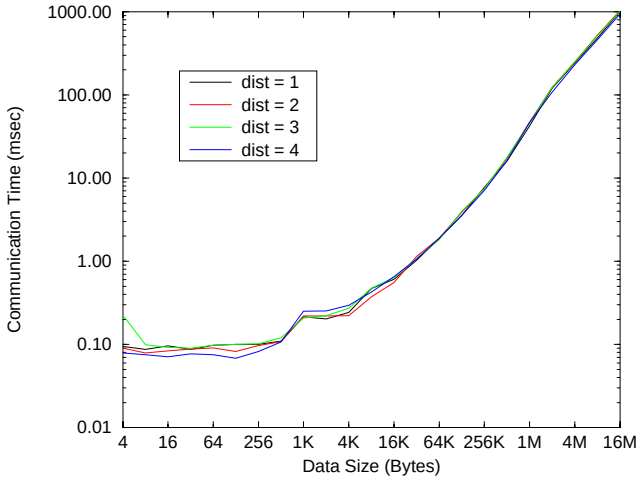


Figure 59: Permutation communication results using 8 processors on O2K

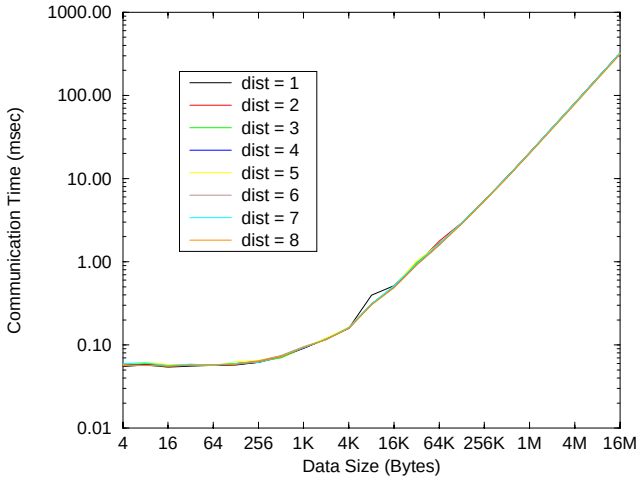


Figure 60: Permutation communication results using 16 processors on SP

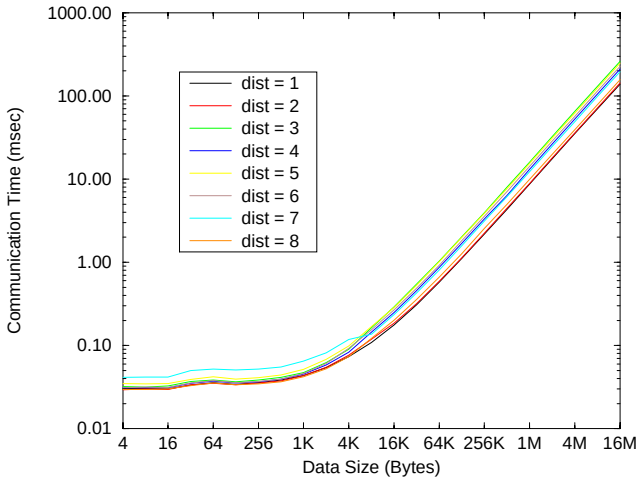


Figure 61: Permutation communication results using 16 processors on T3E

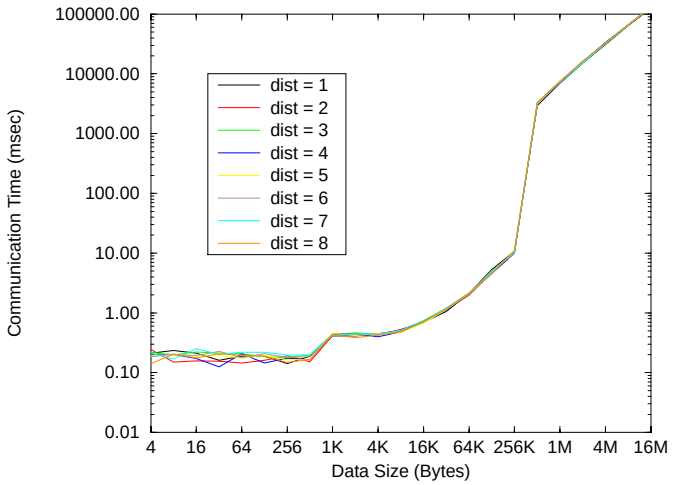


Figure 62: Permutation communication results using 16 processors on O2K

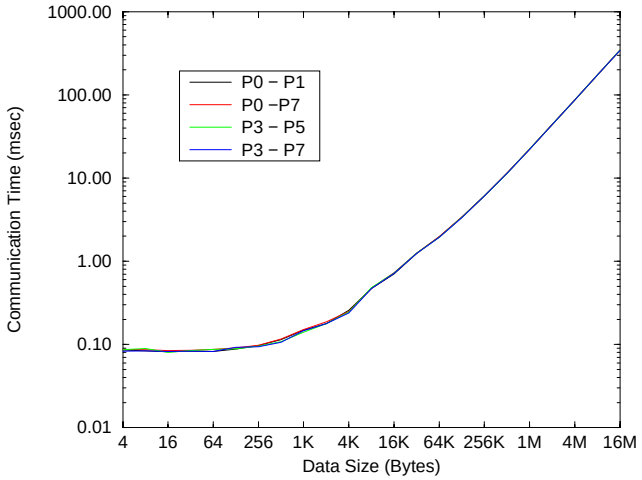


Figure 63: Pingpong communication results using 8 processors on SP

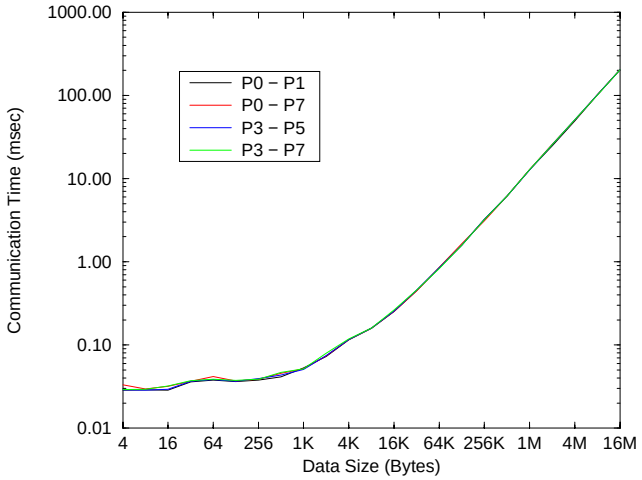


Figure 64: Pingpong communication results using 8 processors on T3E

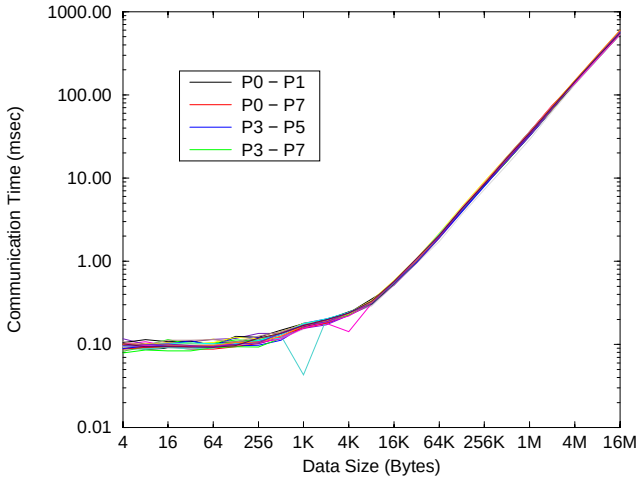


Figure 65: Pingpong communication results using 8 processors on O2K

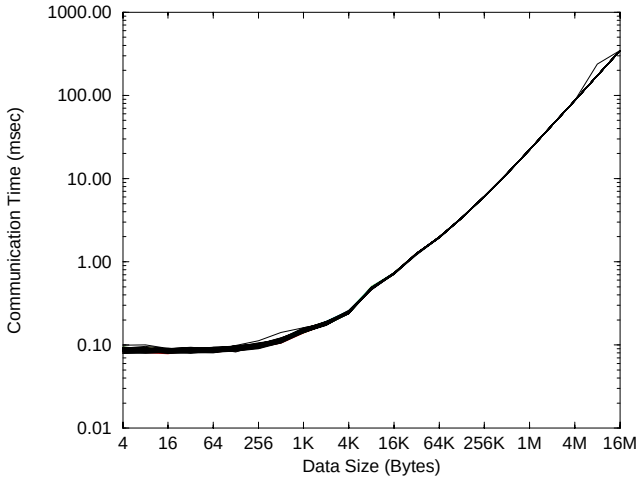


Figure 66: Pingpong communication results using 16 processors on SP

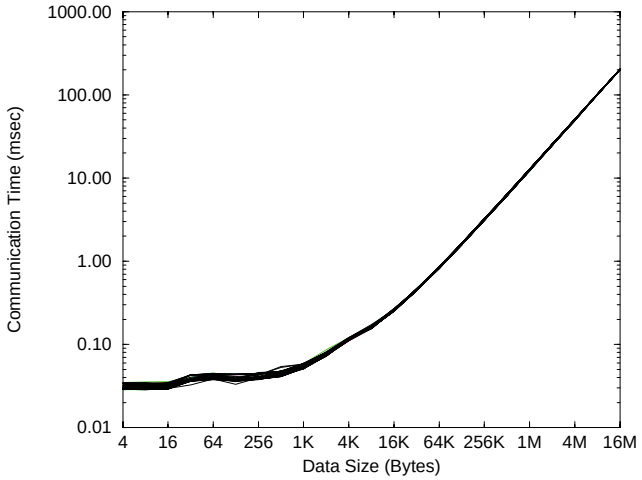


Figure 67: Pingpong communication results using 16 processors on T3E

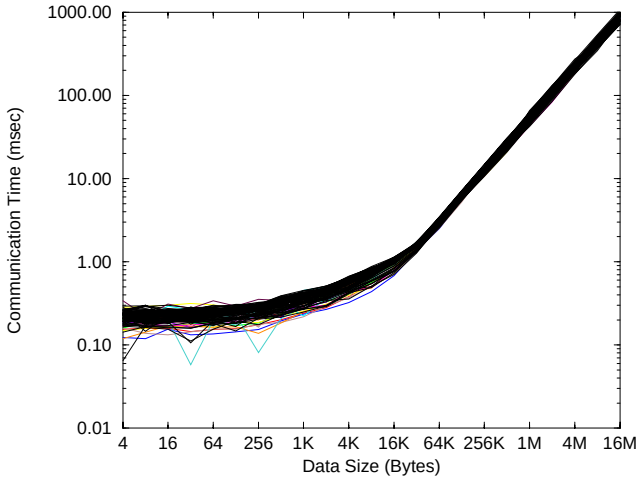


Figure 68: Pingpong communication results using 16 processors on O2K

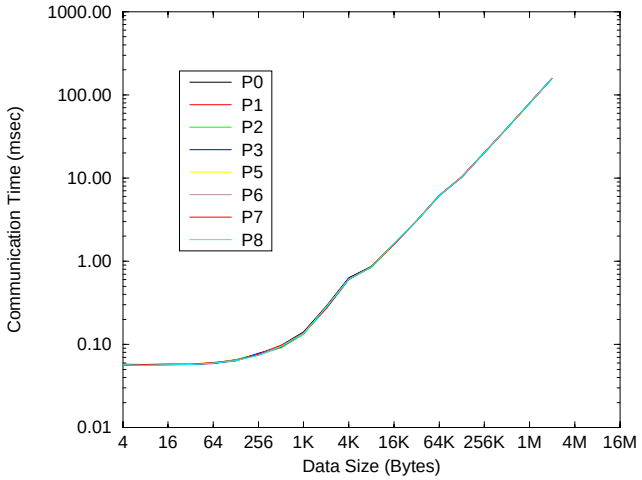


Figure 69: Scatter communication results using 8 processors on SP

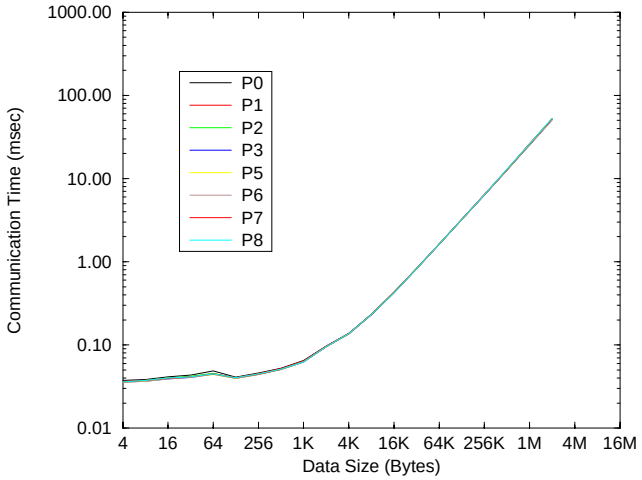


Figure 70: Scatter communication results using 8 processors on T3E

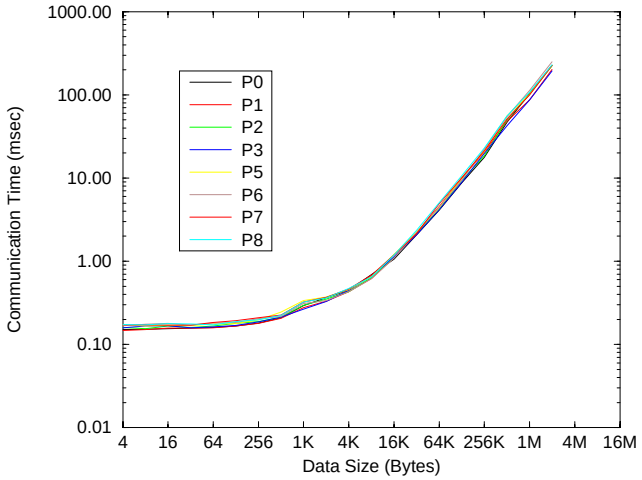


Figure 71: Scatter communication results using 8 processors on O2K

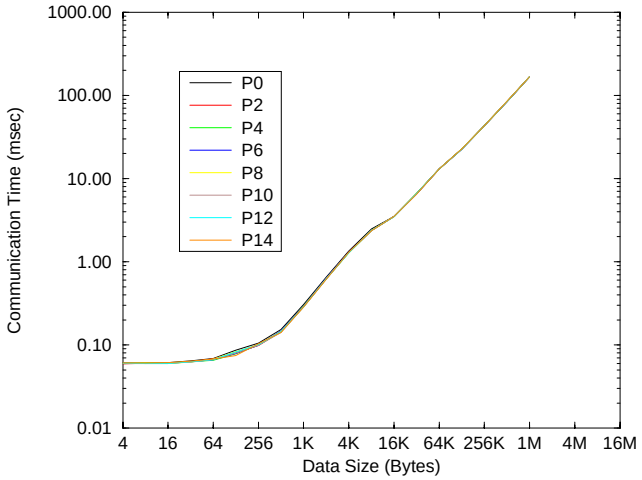


Figure 72: Scatter communication results using 16 processors on SP

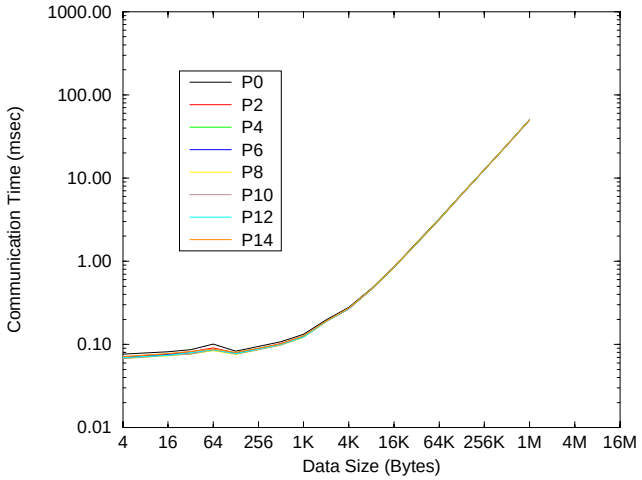


Figure 73: Scatter communication results using 16 processors on T3E

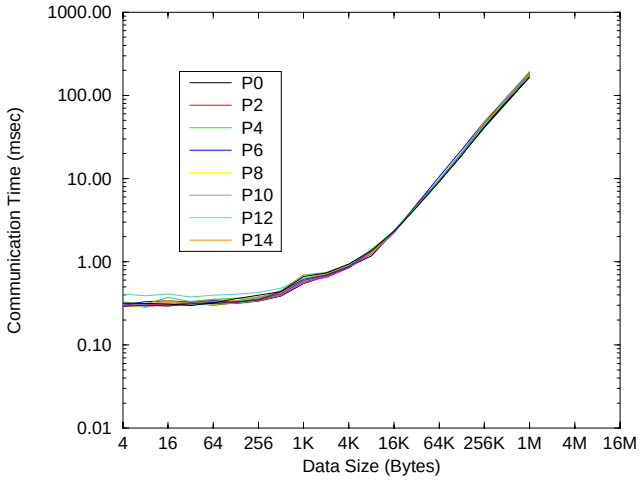


Figure 74: Scatter communication results using 16 processors on O2K

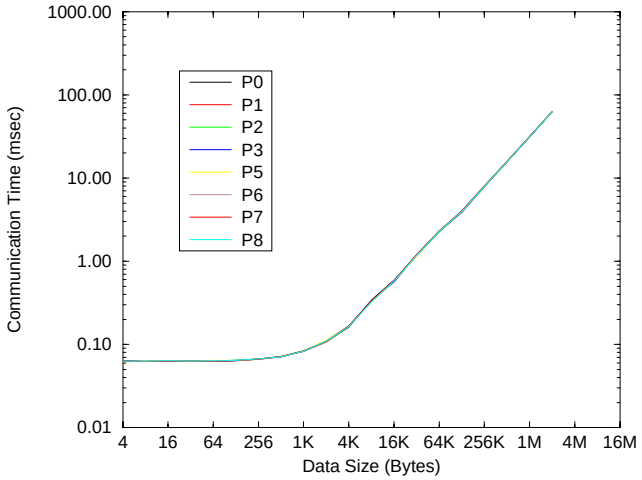


Figure 75: Broadcast communication results using 8 processors on SP

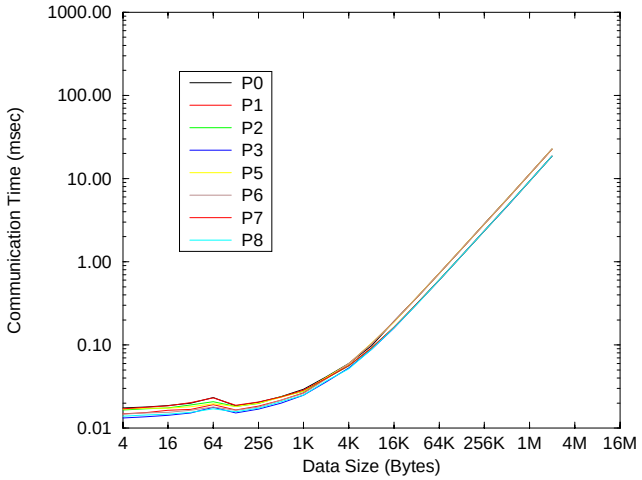


Figure 76: Broadcast communication results using 8 processors on T3E

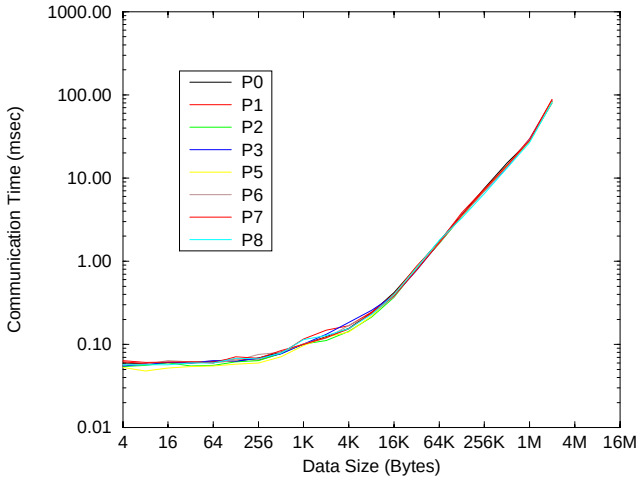


Figure 77: Broadcast communication results using 8 processors on O2K

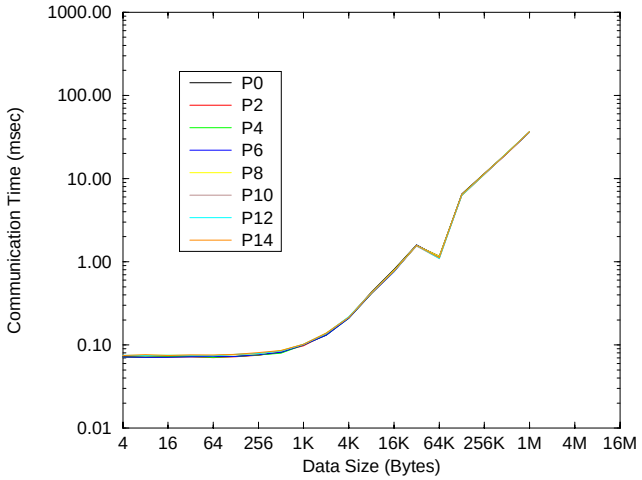


Figure 78: Broadcast communication results using 16 processors on SP

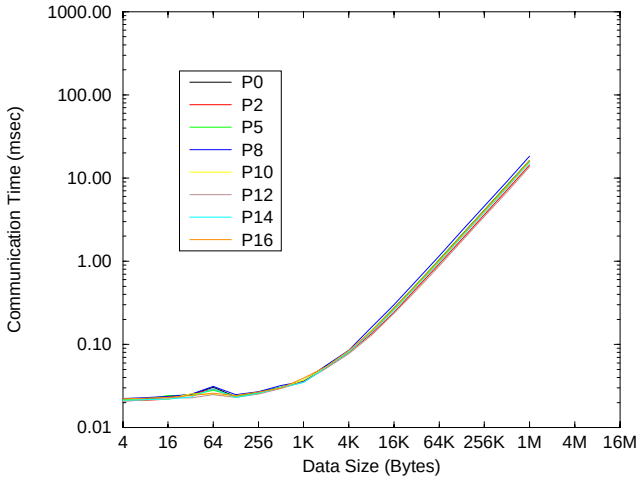


Figure 79: Broadcast communication results using 16 processors on T3E

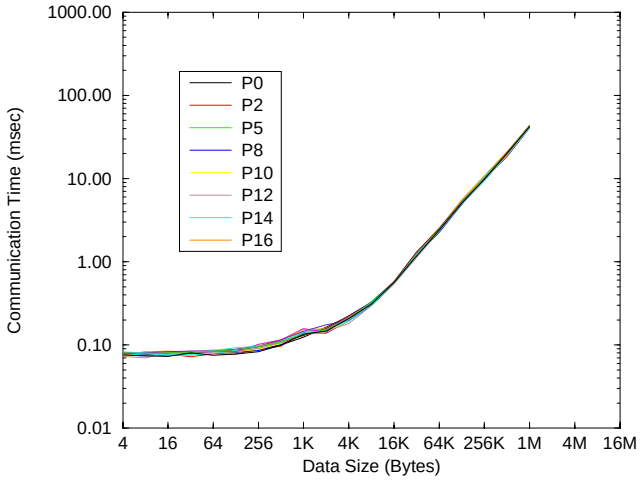


Figure 80: Broadcast communication results using 16 processors on O2K

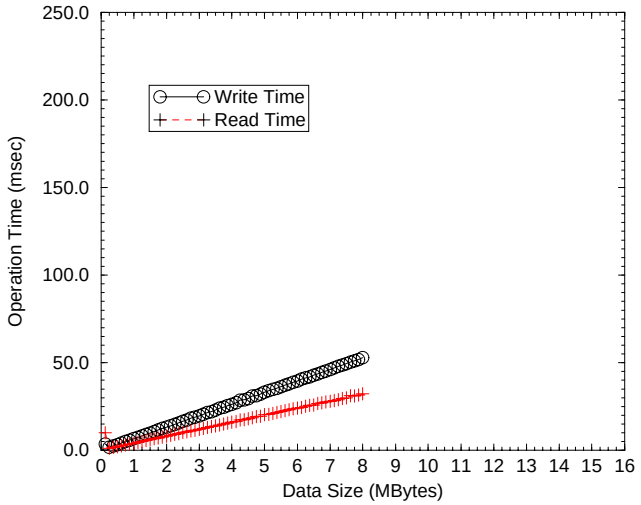


Figure 81: Disk operation results on SP

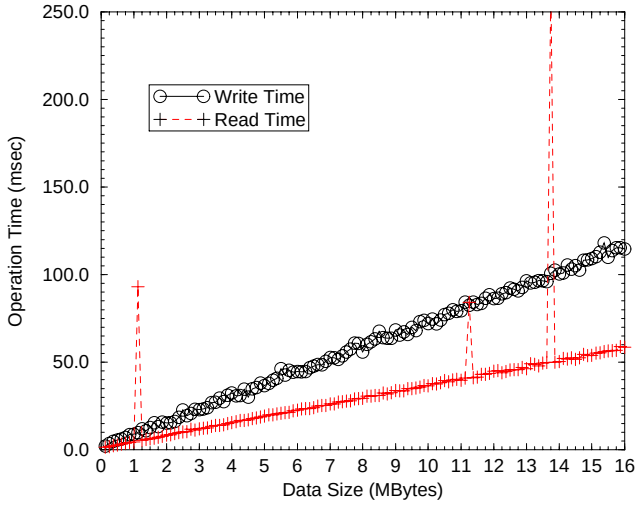


Figure 82: Disk operation results on T3E

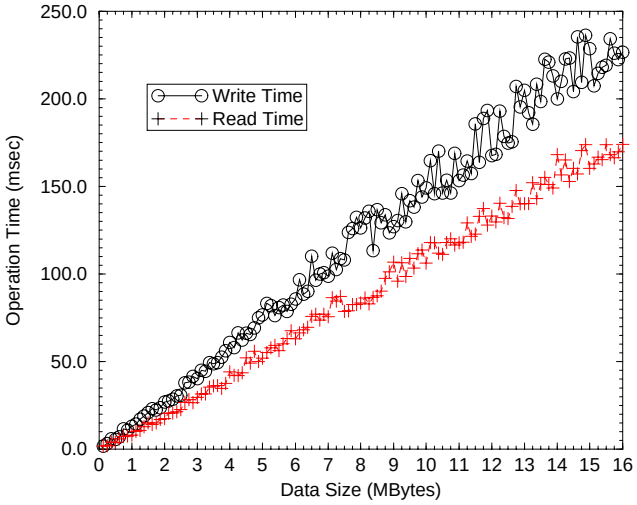


Figure 83: Disk operation results on O2K

C Appendix III: Modified Subroutines

In this appendix, modified subroutines are shown.

C.1 OWN_PL Subroutine

```
      subroutine own_pl
c
c
c         This subroutine defines ownership of nodes to processors and
c         placement of that node in the processor.
c
c
      include 'ft.inc'
c
      common / parall / noproc, myid, ndproc(ndlmx), ndlocl(ndlmx),
     &       ndglob(ndlmx), jbuf(ndlmx * 2)
      common / grid / imax, jmax, kmax, numnpg, numnpo, numnpl,
     &       numell, x(ndlmx), y(ndlmx), z(ndlmx), fbc(ndlmx),
     &       bc(ndlmx), hinit(ndlmx), hyd(ndlmx), ix(8, nelmx)
c
      include 'mpif.h'
c
      dimension num(npmx)
c
c         The global processor and local node arrays are distributed across
c         processors.
c
      nodes = numnpg / noproc
      if (mod (numnpg, noproc) .ne. 0) nodes = nodes + 1
c
c         Define mapping.
c
      kplane = imax * jmax
      n1 = myid * nodes + 1
      n2 = min0 (n1 + nodes - 1, numnpg)
c
      noproc_half = noproc/2
      do n = n1, n2
        k = (n - 1) / imax/jmax
        kk = k/( (kmax + 1)/2)
        i = mod (n - 1, imax)
        if(mod(imax, noproc_half) .eq. 0) then
          ii = i / (imax/noproc_half)
        else
          ii = i/(imax/noproc_half + 1)
        end if
        ndproc(n - n1 + 1) = noproc_half*kk + ii
      end do
c
c         Determine local node numbers.
c
```

```
        call MPI_BARRIER (MPI_COMM_WORLD, ierror)
c
      do i = 1, noproc
        num(i) = 0
      end do
c
      do i = 1, noproc
c
        n1 = (i - 1) * nodes + 1
        n2 = min0 (i * nodes, numnpg)
        numm = n2 - n1 + 1
c
        if (myid .eq. i - 1) then
          do j = 1, numm
            jbuf(j) = ndproc(j)
          end do
        end if
        call MPI_BCAST (jbuf, numm, MPI_INTEGER, i - 1, MPI_COMM_WORLD,
     &     ierror)
c
        do n = n1, n2
c
          ip = jbuf(n - n1 + 1)
          num(ip + 1) = num(ip + 1) + 1
          nloc = num(ip + 1)
          if (i .eq. myid + 1) then
            ndlocl(n - n1 + 1) = nloc
          end if
c
c         Define global node corresponding to nloc.
c
          if (ip .eq. myid) then
            ndglob(nloc) = n
          end if
c
        end do
c
      end do
c
      numnpo = num(myid + 1)
c
      return
      end
```

C.2 UPDADD Subroutine

```
      subroutine updadd (var)
c
c
c         This subroutine updates and adds var for the ghost nodes.
c
c
      include 'ft.inc'
c
      common / parall / noproc, myid, ndproc(ndlmx), ndlocl(ndlmx),
     &       ndglob(ndlmx), jbuf(ndlmx * 2)
      common / ghost / kbuf(nbufmx), ibuf(nbufmx, npmx)
      common / buffer / prbuff(iprbuf), buff(ndlmx * 2 + 1)
c
      include 'mpif.h'
c
      dimension istat(MPI_STATUS_SIZE)
      dimension var(ndlmx), icount(npmx)
      dimension sbuff(ndlmx*2 + 1), rbuff(ndlmx*2 + 1)
c
c         Place all ghost data in every processor.
c
      nphalf = noproc/2
c
c     Step I - send to the RIGHT
c
      nsend = 0
      nrecv = 0
c
c     right-most processors, receive only
c
      if(mod(myid, nphalf) .eq. nphalf - 1) then
         ngh = ibuf(1, myid)
         j = 2
         do while(j .le. ngh + 1)
            ip = ibuf(j, myid)
            if(ip .eq. myid) then
               nrecv = nrecv + 2
            end if
            j = j + 1
         end do
c
         call MPI_RECV(rbuff, nrecv, MPI_REAL, myid - 1, 100,
     &          MPI_COMM_WORLD, istat, ierror)
         do k = 1, nrecv/2
            nlocr = rbuff(k*2 - 1)
            var(nlocr) = var(nlocr) + rbuff(k*2)
         end do
c
c     left-most processors, send only
c
      else if(mod(myid, nphalf) .eq. 0) then
```

```
         ngh = ibuf(1, myid + 1)
         j = 2
         do while(j .le. ngh + 1)
            ip = ibuf(j, myid + 1)
            if(ip .eq. myid + 1) then
               nlocr = ibuf(j + ndghmx, myid + 1)
               nlocs = ibuf(j + ndghmx*2, myid + 1)
               nsend = nsend + 2
               sbuff(nsend - 1) = nlocr
               sbuff(nsend) = var(nlocs)
            end if
            j = j + 1
         end do
c
         call MPI_SEND(sbuff, nsend, MPI_REAL, myid + 1, 100,
     &          MPI_COMM_WORLD, ierror)
c
c     middle processors, send & receive
c
      else
         ngh = ibuf(1, myid)
         j = 2
         do while(j .le. ngh + 1)
            ip = ibuf(j, myid)
            if(ip .eq. myid) then
               nrecv = nrecv + 2
            end if
            j = j + 1
         end do

         ngh = ibuf(1, myid + 1)
         j = 2
         do while(j .le. ngh + 1)
            ip = ibuf(j, myid + 1)
            if(ip .eq. myid + 1) then
               nlocr = ibuf(j + ndghmx, myid + 1)
               nlocs = ibuf(j + ndghmx*2, myid + 1)
               nsend = nsend + 2
               sbuff(nsend - 1) = nlocr
               sbuff(nsend) = var(nlocs)
            end if
            j = j + 1
         end do
c
         call MPI_SENDRECV(sbuff, nsend, MPI_REAL, myid + 1, 100,
     &        rbuff, nrecv, MPI_REAL, myid - 1, 100,
     &        MPI_COMM_WORLD, istat, ierror)
         do k = 1, nrecv/2
            nlocr = rbuff(k*2 - 1)
            var(nlocr) = var(nlocr) + rbuff(k*2)
         end do
      end if
c
```

```
c     Step II - send to the top
c
      nsend = 0
      nrecv = 0
c
c     top processors, receive only
c
      if(myid .ge. nphalf) then
         ngh = ibuf(1, myid - nphalf + 1)
         j = 2
         do while(j .le. ngh + 1)
            ip = ibuf(j, myid - nphalf + 1)
            if(ip .eq. myid) then
               nrecv = nrecv + 2
            end if
            j = j + 1
         end do
c
         call MPI_RECV(rbuff, nrecv, MPI_REAL, myid - nphalf, 100,
     &          MPI_COMM_WORLD, istat, ierror)
         do k = 1, nrecv/2
            nlocr = rbuff(k*2 - 1)
            var(nlocr) = var(nlocr) + rbuff(k*2)
         end do
c
c     bottom processors, send only
c
      else
         ngh = ibuf(1, myid + 1)
         j = 2
         do while(j .le. ngh + 1)
            ip = ibuf(j, myid + 1)
            if(ip .eq. myid + nphalf) then
               nlocr = ibuf(j + ndghmx, myid + 1)
               nlocs = ibuf(j + ndghmx*2, myid + 1)
               nsend = nsend + 2
               sbuff(nsend - 1) = nlocr
               sbuff(nsend) = var(nlocs)
            end if
            j = j + 1
         end do
c
         call MPI_SEND(sbuff, nsend, MPI_REAL, myid + nphalf, 100,
     &          MPI_COMM_WORLD, ierror)
      end if
c
c     Step III - send to the upper diagonal
c
      nsend = 0
      nrecv = 0
c
c     upper diagonal, receive only
c
```

```
      if(myid .gt. nphalf) then
         ngh = ibuf(1, myid - nphalf)
         j = 2
         do while(j .le. ngh + 1)
            ip = ibuf(j, myid - nphalf)
            if(ip .eq. myid) then
               nrecv = nrecv + 2
            end if
            j = j + 1
         end do
c
         call MPI_RECV(rbuff, nrecv, MPI_REAL, myid - nphalf - 1, 100,
     &          MPI_COMM_WORLD, istat, ierror)
         do k = 1, nrecv/2
            nlocr = rbuff(k*2 - 1)
            var(nlocr) = var(nlocr) + rbuff(k*2)
         end do
c
c     lower diagonal, send only
c
      else if(myid .lt. nphalf - 1) then
         ngh = ibuf(1, myid + 1)
         j = 2
         do while(j .le. ngh + 1)
            ip = ibuf(j, myid + 1)
            if(ip .eq. myid + nphalf + 1) then
               nlocr = ibuf(j + ndghmx, myid + 1)
               nlocs = ibuf(j + ndghmx*2, myid + 1)
               nsend = nsend + 2
               sbuff(nsend - 1) = nlocr
               sbuff(nsend) = var(nlocs)
            end if
            j = j + 1
         end do
c
         call MPI_SEND(sbuff, nsend, MPI_REAL, myid + nphalf + 1, 100,
     &          MPI_COMM_WORLD, ierror)
      end if
c
      return
      end
```

C.3 UPDATE Subroutine

```
      subroutine update (var)
c
c
c         This subroutine updates var for the ghost nodes.
c
c
      include 'ft.inc'
c
      common / parall / noproc, myid, ndproc(ndlmx), ndlocl(ndlmx),
     &       ndglob(ndlmx), jbuf(ndlmx * 2)
      common / ghost / kbuf(nbufmx), ibuf(nbufmx, npmx)
      common / buffer / prbuff(iprbuf), buff(ndlmx * 2 + 1)
c
      include 'mpif.h'
c
      dimension istat(MPI_STATUS_SIZE)
      dimension var(ndlmx), icount(npmx)
      dimension sbuff(ndlmx*2 + 1), rbuff(ndlmx*2 + 1)
c
      nphalf = noproc/2
c
c     Step I - send to the LEFT
c
      nsend = 0
      nrecv = 0
c
c     left-most processors, receive only
c
      if(mod(myid, nphalf) .eq. 0) then
         ngh = ibuf(1, myid + 1)
         j = 2
         do while(j .le. ngh + 1)
            ip = ibuf(j, myid + 1)
            if(ip .eq. myid + 1) then
               nrecv = nrecv + 2
            end if
            j = j + 1
         end do
c
         call MPI_RECV(rbuff, nrecv, MPI_REAL, myid + 1, 100,
     &          MPI_COMM_WORLD, istat, ierror)
         do k = 1, nrecv/2
            nlocr = rbuff(k*2 - 1)
            var(nlocr) = rbuff(k*2)
         end do
c
c     right-most processors, send only
c
      else if(mod(myid, nphalf) .eq. nphalf - 1) then
         ngh = ibuf(1, myid)
         j = 2
```

```
         do while(j .le. ngh + 1)
            ip = ibuf(j, myid)
            if(ip .eq. myid) then
               nlocs = ibuf(j + ndghmx, myid)
               nlocr = ibuf(j + ndghmx*2, myid)
               nsend = nsend + 2
               sbuff(nsend - 1) = nlocr
               sbuff(nsend) = var(nlocs)
            end if
            j = j + 1
         end do
c
         call MPI_SEND(sbuff, nsend, MPI_REAL, myid - 1, 100,
     &          MPI_COMM_WORLD, ierror)
c
c     middle processors, send & receive
c
      else
         ngh = ibuf(1, myid + 1)
         j = 2
         do while(j .le. ngh + 1)
            ip = ibuf(j, myid + 1)
            if(ip .eq. myid + 1) then
               nrecv = nrecv + 2
            end if
            j = j + 1
         end do

         ngh = ibuf(1, myid)
         j = 2
         do while(j .le. ngh + 1)
            ip = ibuf(j, myid)
            if(ip .eq. myid) then
               nlocs = ibuf(j + ndghmx, myid)
               nlocr = ibuf(j + ndghmx*2, myid)
               nsend = nsend + 2
               sbuff(nsend - 1) = nlocr
               sbuff(nsend) = var(nlocs)
            end if
            j = j + 1
         end do
c
         call MPI_SENDRECV(sbuff, nsend, MPI_REAL, myid - 1, 100,
     &        rbuff, nrecv, MPI_REAL, myid + 1, 100,
     &        MPI_COMM_WORLD, istat, ierror)
         do k = 1, nrecv/2
            nlocr = rbuff(k*2 - 1)
            var(nlocr) = rbuff(k*2)
         end do
      end if
c
c     Step II - send to the bottom
c
```

```
      nsend = 0
      nrecv = 0
c
c     bottom processors, receive only
c
      if(myid .lt. nphalf) then
         ngh = ibuf(1, myid + 1)
         j = 2
         do while(j .le. ngh + 1)
            ip = ibuf(j, myid + 1)
            if(ip .eq. myid + nphalf) then
               nrecv = nrecv + 2
            end if
            j = j + 1
         end do
c
         call MPI_RECV(rbuff, nrecv, MPI_REAL, myid + nphalf, 100,
     &          MPI_COMM_WORLD, istat, ierror)
         do k = 1, nrecv/2
            nlocr = rbuff(k*2 - 1)
            var(nlocr) = rbuff(k*2)
         end do
c
c     top processors, send only
c
      else
         ngh = ibuf(1, myid - nphalf + 1)
         j = 2
         do while(j .le. ngh + 1)
            ip = ibuf(j, myid - nphalf + 1)
            if(ip .eq. myid) then
               nlocs = ibuf(j + ndghmx, myid - nphalf + 1)
               nlocr = ibuf(j + ndghmx*2, myid - nphalf + 1)
               nsend = nsend + 2
               sbuff(nsend - 1) = nlocr
               sbuff(nsend) = var(nlocs)
            end if
            j = j + 1
         end do
c
         call MPI_SEND(sbuff, nsend, MPI_REAL, myid - nphalf, 100,
     &          MPI_COMM_WORLD, ierror)
      end if
c
c     Step III - send to the diagonal
c
      nsend = 0
      nrecv = 0
c
c     bottom diagonal, receive only
c
      if(myid .lt. nphalf - 1) then
         ngh = ibuf(1, myid + 1)
```

```
         j = 2
         do while(j .le. ngh + 1)
            ip = ibuf(j, myid + 1)
            if(ip .eq. myid + nphalf + 1) then
               nrecv = nrecv + 2
            end if
            j = j + 1
         end do
c
         call MPI_RECV(rbuff, nrecv, MPI_REAL, myid + nphalf + 1, 100,
     &          MPI_COMM_WORLD, istat, ierror)
         do k = 1, nrecv/2
            nlocr = rbuff(k*2 - 1)
            var(nlocr) = rbuff(k*2)
         end do
c
c     top diagonal, send only
c
      else if(myid .gt. nphalf) then
         ngh = ibuf(1, myid - nphalf)
         j = 2
         do while(j .le. ngh + 1)
            ip = ibuf(j, myid - nphalf)
            if(ip .eq. myid) then
               nlocs = ibuf(j + ndghmx, myid - nphalf)
               nlocr = ibuf(j + ndghmx*2, myid - nphalf)
               nsend = nsend + 2
               sbuff(nsend - 1) = nlocr
               sbuff(nsend) = var(nlocs)
            end if
            j = j + 1
         end do
c
         call MPI_SEND(sbuff, nsend, MPI_REAL, myid - nphalf - 1, 100,
     &          MPI_COMM_WORLD, ierror)
      end if
c
      return
      end
```