INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

ProQuest Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

Message Passing Interface Implementation for Concordia Parallel Programming Environment

Zhong Guan

A Major Report

in

The Department

of

Computer Science

Presented in Partial Fulfillment of the Requirements
for the Degree of Master of Computer Science at
Concordia University
Montreal, Quebec, Canada

September 2000

© Zhong Guan, 2000

National Library of Canada
Acquisitions and Bibliographic Services
395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque nationale du Canada
Acquisitions et services bibliographiques
395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

Our file Notre référence

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-59322-3

Canada

Abstract

Message Passing Interface Implementation for Concordia Parallel Programming Environment

Zhong Guan

In this report, we present the design and implementation of a Message Passing Interface (MPI) [1] for the Concordia Parallel Programming Environment (CPPE), an environment for parallel computing simulation. The purpose of this project is to provide Message Passing Interface (MPI) support for CPPE, so that user can get access to MPI programming with CPPE. Also the CPPE environment, which allows users to study impacts of system and software factors on program performance and locate performance bottlenecks in parallel programming [2], in turn, benefits the developing of the MPI applications with all of its advanced features.

Parallel computing offers the potential to push the performance of computer systems into new dimensions. One of the main obstacles for a broad application of the parallel technology is the lack of parallel programming standards. The Message Passing Interface (MPI) is just such a portable message passing standard that facilitates the development of parallel applications and libraries. The goal of MPI is to achieve efficient communication, portability and rich functionality.

This report outlines the development and implementation of MPI for CPPE. As a development tool, this implementation enables MPI parallel software developing easy and inexpensive. As a learning tool, it provides a larger group of computer users with the opportunity to gain experience with MPI programming on their personal computers.

We also discuss the MPI standard in detail, and furthermore illustrate the MPI parallel programming with some example programs. We focus on design and implementation issues for MPI, as well as the simulation procedure under CPPE virtual parallel machine. Special consideration is given to confirm the correctness of this implementation by comparing the running of a series of MPI applications on CPPE with that of MPICH implementation on UNIX.

Acknowledgements

I would like to thank Dr. Lixin Tao for his thesis supervision during my study at Concordia University for my master degree.

My thanks also go to the CPPE team: Dr. Lixin Tao, the team leader; Hassan Hosseini and Ai Kong (CPCC), Hoang Uyen Trang Nguyen and Thien Bui (CPSS), which provided the good basis for the contributions of this thesis.

I am grateful to the professors and administrative staffs in the Department of Computer Science, especially Dr. Greg Butler, who gave the lectures in Software Engineering and Software Design Methodology, Dr. H.F. Li, who taught an excellent course in Computer Architecture and offered me the guidance in graduate study, Mr. Stan Sweircz, who gave me so much help in the use of computer system and system programming and Ms. Halina Monkiewicz, whose friendliness and administrative support have made my student's life much easier and pleasant.

I also have a very supportive family that backs me up. My wife supported me with her unconditional love, sharing and inspiration throughout my study.

I am also grateful to my parents for their encouragement and help.

Contents

List of Tables

List of Figures

Chapter 1

1. Introduction

In this chapter, we first discuss the motivations and objectives of this research. Then we present the contributions of this thesis. At the end we give a thesis outline.

1.1 Motivations

1.1.1 The Need of Parallel Computation

Since the invention of computers about 50 years ago, the world of computing has changed rapidly. Roughly in each decade, a new generation of computer systems sets higher standards with regards to performance, size, price, and usability. Computers continue to conquer new spheres in industry, research, and management and affect virtually every aspect of daily life.

In the 1990's, this race goes into a new round, the Era of Parallel Computers. Parallel computer systems adopt the idea of cooperation by employing multiple processors. Huge computational problems are divided, separately solved, and integrated into a final solution.

The boost of parallel computers is no coincidence. Many scientific and engineering problems in research and industry demand tremendous computational power for large-scale computations that cannot be provided by uniprocessors. Parallel processing has shown its potential to meet the demands for high computing power. Complex processes such as atmospheric activities, crash test simulations, fluid dynamics, oil reservoir, seismic study and computer visualization are modeled theoretically by mathematical methods, resulting in complexities of large-scale computations. Only parallel computers provide the sufficient computational performance to solve such problems in an acceptable period of time.

1.1.2 The Need of Simulating Environment for Parallel Computation

Computing power has always been an issue in research and industry areas. Despite the tremendous growth of processor speed over years, there is always a wide range of important computational problems in science and engineering that require much greater computer speed. Parallel programming has shown its potential to meet the demands for high computing power. However, parallel programming is difficult and error-prone, not only because parallel processes are difficult to trace and debug, but also because many hardware and software factors, such as algorithm design, system architecture, routing technique, and networking speed, influence the performance of parallel programs.

In order to evaluate and improve the performance of parallel applications effectively, all deciding factors must be taken into account. Also, programmers/designers should be able to observe the effects of these factors on their applications so as to detect system bottlenecks and thus optimize performance of the application. Fortunately, CPPE is a powerful tool developed for parallel research. CPPE offers programmers a development environment superior to those on real multiprocessors. It provides flexible and efficient software tools for developing parallel applications and optimizing their performance. It supports testing and debugging, as well as algorithmic and architectural performance evaluation and tuning which allows users to evaluate impacts of system and software factors on performance of parallel applications. It allows you to produce efficient parallel programs by locating and eliminating performance bottlenecks in the programs.

CPPE (Concordia Parallel Programming Environment) is one of the simulated parallel systems, which is studied and developed by our group, aiming to provide users with flexible and efficient software tools for developing parallel programs, evaluating and optimizing performance of parallel applications.

1.1.3 The Need of Another Standard as MPI

By the mid-1950's, computers had come into a state of development that suggested their successful utilization not only for research but also for applications in industry and administration. A number of manufacturers designed and marketed computer systems. Many computer programs were written for these systems. During this period, assembly language was used for software development, which, from today's point of view, has several drawbacks:

- designed for a particular hardware
- source code difficult to understand
- laborious and time-consuming debugging

As a result, computer programs could be used only for a particular computer system and proved to be very expensive and error-prone. Efficient software development required new concepts.

In the late 1950's, the solution appeared in the form of high-level languages such as Fortran and Algol 60 [3]. Source code became more intuitive and easier to read and to debug. A compiler translated the source code to machine-dependent assembly (or executable) code. Suddenly, it was possible to write reliable software of greater complexity in a shorter time.

Additionally, high-level languages defined a hardware-independent programming interface; programs could run on different computer systems. The standardization of high-level languages allowed for portable software developing.

Computers became even easier to use when UNIX was adopted as the de-facto operating system standard. Today, no major workstation vendor can afford not to offer UNIX for its computer systems. Once familiar with the UNIX user interface, users are able to operate machines from different manufacturers without additional training.

During the last two decades, parallel computing has evolved into a major field of research in computer science. Manufacturers such as Cray, IBM, INTEL, nCube, Maspar, Sequent, and Silicon Graphics have developed and marketed computer systems equipped with multiple processors. Two major groups of parallel systems can be distinguished:

- Shared memory machines - Multiple processors have access to a common main memory (shared memory communication).
- Distributed memory machines - Each processor has associated it's own local main memory. The processors exchange messages via separate communication channels (message passing).

The UNIX standard proved to be so strong that even vendors of parallel machines felt compelled to offer UNIX-like operating systems for their computers. Parallel machines can be programmed using popular high-level languages such as C, Fortran, and Pascal. However, parallel computers require specific extensions to the UNIX operating system for communication and synchronization. Hence, vendors added functions to the UNIX programming interface, which are used for implementing parallel application software.

The number of problems in research and industry requiring huge computational power is steadily increasing; nevertheless, the parallel computers are very expensive since the number of sold systems is still small. A less expensive alternative is the use of distributed systems, which consist of a number of workstations connected by a communication network (workstation cluster). Distributed systems are suitable for parallel applications although they do not reach the performance of parallel systems.

The current state of development for parallel and distributed systems is comparable to the crisis in assembly language programming in the late 1950's. An application program written for a particular parallel computer system is difficult to port to another manufacturer's hardware because the parallel extensions to the UNIX programming environment differ. As a solution to this dilemma, a standard is desirable. This new standard will accomplish for parallel applications what high-level languages permitted for sequential application software - application programs running on different parallel machines without any changes.

Numerous attempts have been made to propose a standard. Intel's NX/2 [4], PICL [5], Express [6], p4 [7], and PVM [8] are only a few examples. Common drawbacks of these programming systems are:

- Designed with regard to the hardware of a particular computer manufacturer.
- Designed as a research project, not suitable for commercial use.

Meanwhile, the availability of a widely accepted standard for parallel computers is the key to growing acceptance of parallel computers in research and industry. In order to be suitable for all types of parallel computers, a parallel program standard must be based on the most general communication paradigm, the message-passing paradigm. In the message-passing paradigm, sharing resources and synchronization among processes are achieved by sending messages.

About 60 people from 40 organizations participated in the standardization of the Message-Passing Interface Standard (MPI). Most major manufacturers of parallel computers and researchers from universities, government laboratories, and industry were involved in the development of MPI.

The MPI standardization process began in April 1992, with the Workshop on Standards for Message Passing in a Distributed Memory Environment, which was held in Williamsburg, Virginia [7]. At this workshop the essential features for a message-passing interface standard were discussed, and the MPI working group was formed to continue the standardization process.

In November 1992, this working group held a meeting in Minnesota where the standardization process was reformed by adopting the organization and procedures of the High Performance Fortran Forum. For each major component area, a subcommittee was formed and an email discussion service provided. Both together embodied the MPI Forum, which invited all members of the high performance computing community to participate in the standardization process.

A Revised Version of the preliminary draft proposal, called MPI1, was available in February 1993 [10]. The intention behind MPI1 was to provide a base for a broad discussion. The draft MPI standard was presented at the Supercomputing '93 conference in November 1993. It is available as a technical report from the University of Tennessee report [11], as a postscript file by ftp (from info.mcs.anl.gov in /pub/mpi/mpi-report.Z), as a hypertext on the World Wide Web at http://www.mcs.anl.gov/mpi, and as an article in the Journal of Supercomputing Applications [12].

Meanwhile, MPI has reached a state, where several proprietary, native implementations are in progress and portable public domain implementations are available [13].

A discussion at the final MPI meeting in February 1994, resulted in the decision of MPI standard. MPI 2 extends the MPI message passing standard in the later years.

1.1.4 The Need of MPI for Parallel Computation in CPPE

Due to the utilization of the latest technology, immense development costs, and lack of competition, parallel computers have been very expensive and only available to a limited number of institutions. Distributed systems of interconnected workstations are an alternative. They are widely available and offer a far better price-performance ratio.

Currently, many users have access to parallel computing technology and are willing to use it for their particular needs. Both demand and supply of parallel software are increasing considerably. Software developers and users face the nonexistence of a widely accepted standard as a severe obstacle. Software for a particular parallel computer is expensive to rewrite for another system or is simply not portable. Numerous attempts have been made to establish a standard. Due to a broad foundation of support from vendors and researchers, a standard called Message Passing Interface (MPI) is adopted as the base for future parallel application software. The standard is stable, and implementations are wide used for parallel and distributed systems.

CPPE offers a development environment superior to those available on real multiprocessors. It provides flexible and efficient software tools for developing parallel applications and optimizing their performance. It supports testing and debugging, as well as algorithmic and architectural performance evaluation and tuning, which allows users to evaluate impacts of system and software factors on performance of parallel applications. It allows you to produce efficient parallel programs by locating and eliminating performance bottlenecks in the programs by optimal mapping functions for wormhole-routed networks, and providing accurate information about the timing and behavior of parallel applications and the underlying simulated architecture.

The development of CPPE and its MPI porting is to provide a superior environment for parallel computing software development.

As part of the CPPE project, the research objective of this thesis is to implement the MPI standard in order to make CPPE a more portable, flexible and efficient system

for parallel program development. The focus of this project is on MPI implementation for CPPE.

1.2 Overview of CPPE

In this section we are going to discuss the architecture and high level design of CPPE as a start point of this MPI project.

1.2.1 Architecture of CPPE

CPPE-MPI implementation, which is part of the research project of Concordia Parallel Programming Environment (CPPE), provides CPPE with MPI programming support.

CPPE consists of two major moduli [14] (Figure 1):

1. Concordia Parallel C Compiler (CPCC): The CPCC accepts parallel programs written in the CPC (Concordia Parallel C) language and generates virtual machine code (vCode), which will be the input to the CPSS.
2. Concordia Parallel System Simulator (CPSS): The CPSS reads in the intermediate code produced by the CPCC, simulates the execution of the application, and yields the program output.

The development and implementation of MPI need to extend both CPCC and CPSS components, with focus on the CPSS components.

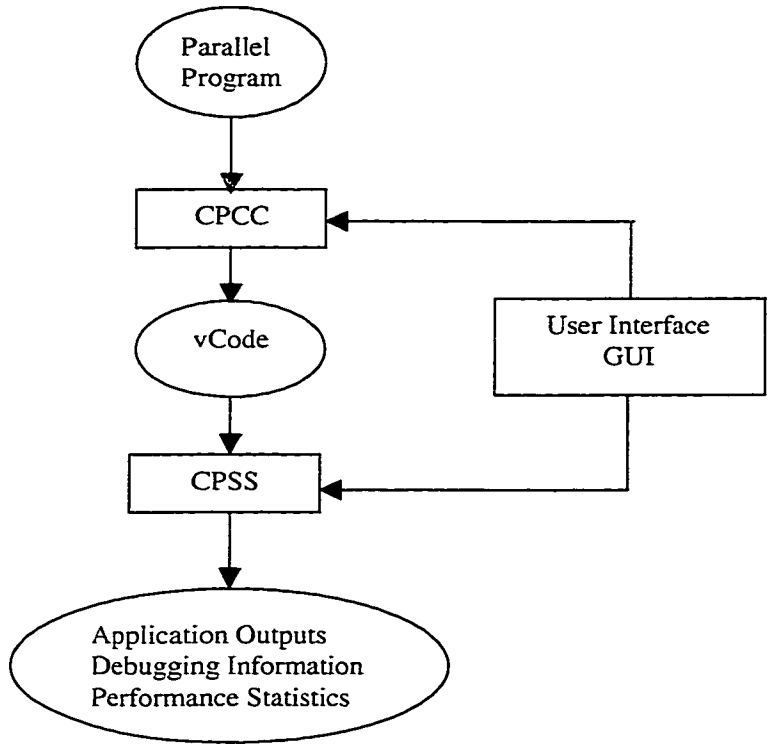


Figure 1: General structure of the CPPE

1.2.2 Architecture of CPSS

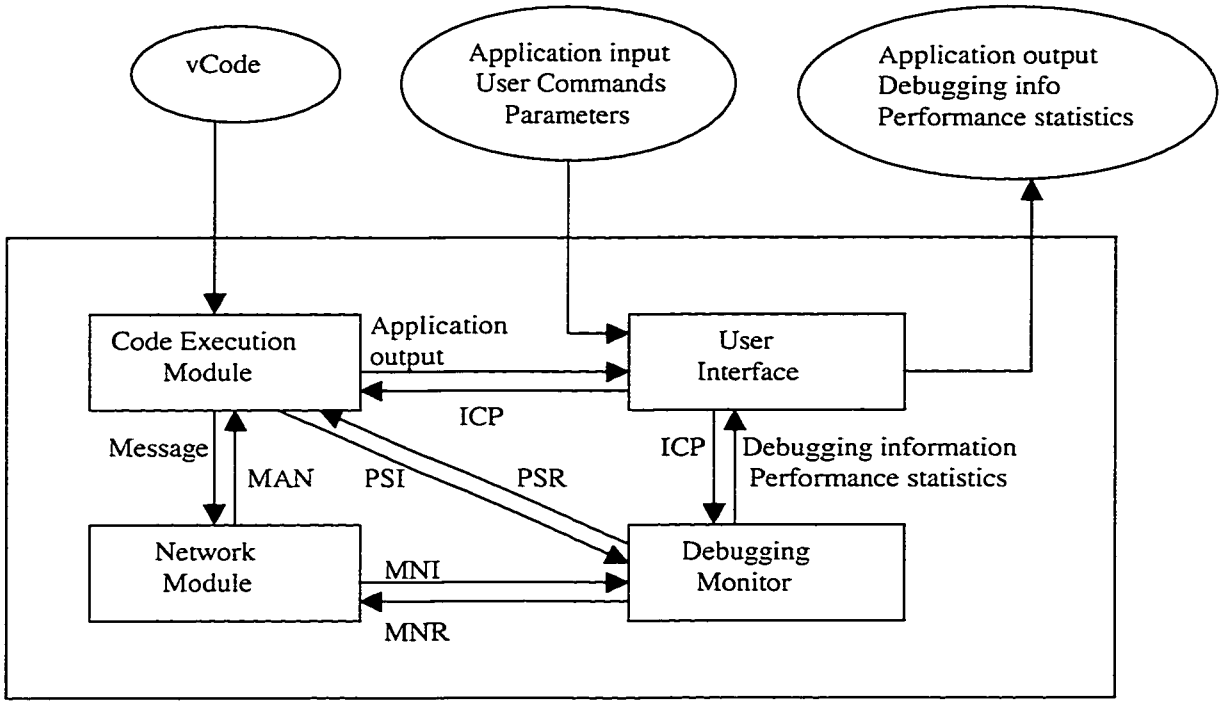
CPPE (Concordia Parallel Programming Environment) consists of two components: the CPCC (Concordia Parallel C Compiler) and the CPSS (Concordia Parallel System Simulator) [14].

The core of the CPCC (Concordia Parallel C Compiler) is a compiler. After reading a parallel program written in CPC (Concordia Parallel C) language, the CPCC builds a complete abstract syntax tree to perform syntax and semantics analysis, and produces object code for a generic virtual machine. Such object code is called vCode in CPPE. The defination of the vCode instruction set is based on an analysis of common operations of parallel computer systems. To produce vCode, the compilation process makes use of the virtual architecture instead of calling for the physical architecture. The advantage of this design is that the CPC parallel program does not need to be re-compiled every time the underlying target architecture is changed.

The vCode produced by the CPCC will be input to the CPSS. Other inputs to the CPSS are parameters and commands from the user. For example, the user can specify the physical topology, on which the program will run, and the virtual-to-physical topology mapping. The CPSS then executes the vCode, using the parameters and commands entered by the user. The outputs from the CPSS are the application outputs, performance statistics, and debugging information.

The CPSS consists of two major components: the code execution module and the network module. The code execution module models the processing elements of the parallel computer system: it executes the parallel code specified by the parallel program. The network module manages the inter-processor communication via message passing. There are two other utility modules interacting with the code execution module and network module in CPSS: the debugging monitor and the user interface.

The interactions between the components in CPSS are illustrated in Figure 3.



MAN: Message Arrival Notification
ICP: Input/Commands/Parameters
MNR: Message/Network Request
PSR: Process/Processor Status Request
MNI: Message/Network Information
PSI: Process/Processor Status Information

Figure 2: CPSS structure and operations

The Code Execution Module

The Code Execution Module (CEM) plays a role in processing elements of a parallel computer system: it executes the parallel code specified by the parallel program. There are three key issues that influence the design of the debugging monitor: simulation at the functional level, sequential execution model and the way in which timing system is implemented. CPSS uses the functional simulation technique, with sequential execution model emulating the parallel execution and interpreting the parallel object code instructions at the functional level. This technique offers the most accurate results among the existing simulation techniques. In addition, this technique provides a good basis for performance debugging:

- Functional simulation: CEM interprets the intermediate parallel code at the functional level in such a way as if they were executed on the target machine. Each instruction of the target machine is usually expressed as a host macro or procedure, whose size varies with the complexity of the instruction and the desired level of simulation accuracy. This technique permits the simulator to have complete control over program execution. It establishs the connection between user program statements and intermediate instructions. Thus the user can set breakpoints, examine trace variables, or step-through the program fragment of a particular process. Monitor code can be added to the simulating code without affecting the execution outcomes, because the CEM is able to distinguish between application code and monitor code and the execution time for monitoring code is not accumulated.

- By using functional simulation technique, we can parameterize system measurements (e.g, system clock cycle, execution time of object code instructions, network packet size, link buffer size, network delay, message and packet startup overheads). Performance statistics are based on these parameterized measurements.

- The sequential simulation is deterministic in nature. Therefore executing a parallel program repetitively with the same system parameters will always produce the same results and performances. This provides a stable environment to study the performance of parallel programs at different levels of detail and from different perspectives.

- Timing system: CPSS does not use the machine clock for performance timing. There is a global clock for the simulated parallel computer system, which is updated periodically by the CEM. Each process has a local clock that keeps track of the present time of this process. In the CPSS, parallelism is simulated by time slicing, with each application process being given a quantum to run and scheduled in a round-robin fashion. During each quantum, the process scheduler traverses the list of processes, scheduling one at a time for execution. The local clock of the scheduled process is updated after each instruction is executed. The cost to execute an instruction is dependent on the complexity of the instruction and thus estimatable. The process runs until its time quantum expires or it is put to sleep by some event. The process scheduler then schedules the next process for execution. When every parallel application process has finished its quantum, the global clock is advanced to the next quantum. By using this timing system, CPSS can provide accurate and repeatable performance statistics for performance debugging.

The Network Module

The Network Module is responsible for inter-process communication via message passing. It is under control of the network manager. The network manager allocates network resources to the messages to be sent, routes them and delivers them to the destination processors, and detects and resolves deadlocks, if any.

The following design issues of the network module is crucial for accurately simulating the communication behavior, yet providing feasibility for performance evaluation for parallel applications:

- By using the functional simulation technique and the same global clock mentioned in the CEM design, the network module can effectively simulate the network behavior and communication cost such as message startup overhead, routing overhead and congestion delay. New messages, which are being initialized for routing, are queued at a new message list. The waiting time at this list simulates message startup overheads. When the startup overhead time of a new message expires, the message is removed from the list and appended to a list of active messages. In each quantum, all active packets that are not blocked are advanced by one link. The network module simulates the movement of packets by advancing their ID numbers.

- Most of the network and communication parameters are well defined with appropriate data structures. User can configure most of the network parameters such as packet size, flit size, routing scheme, link bandwidth, communication delay, network topologies and virtual-to-physical mapping without recompile of the simulator software and application programs, which adds to the flexibility of the performance debugging environment.

1.3 Overview of MPI

1.3.1 What Is MPI?

MPI is a message-passing library, a collection of routines for facilitating communication (exchange of data and synchronization of tasks) among processors in a distributed memory parallel program. The acronym stands for Message-Passing Interface. MPI is the first standard and portable message-passing library with good performance.

The MPI "standard" was introduced by the MPI Forum in May 1994 and updated in June 1995. The document that defines it is entitled "MPI: A Message-Passing Standard", published by the University of Tennessee. MPI 2 extends the MPI message passing standard without changing it in the following areas:

- Dynamic process management
- One-sided operations
- Parallel I/O
- C++ and FORTRAN 90 bindings
- External interfaces
- Extended collective communications
- Real-time extensions
- Other areas

1.3.2 What does MPI offer?

MPI offers portability, standardization, performance, functionality, and several high quality implementations.

- **Standardization** MPI is standardized on many levels. Since the functional behavior of MPI calls is standardized, there is no need to worry about which implementation of MPI is currently on the machine; the MPI calls should behave the same regardless of the implementation. Performance, however, varies slightly with different implementations.

- **Portability** With environments of high performance computers changing rapidly and communication technology developing fast, portability is given a thought to by almost everyone. Who wants to develop a program that can be run on only one machine, or only poorly on others? All massively parallel processing (MPP) systems provide some sort of message passing library specific to their hardware. These provide great performance, but an application code written for one platform cannot be ported easily to another. With MPI, one can write portable programs that still take advantage of the specifications of the hardware and software provided by vendors. Happily, this is mostly taken care of by simply using MPI calls because the implementers have tuned these calls to the underlying hardware and software environment.

- **Performance** A number of environments, including PVM, Express, and P4, have attempted to provide a standardized parallel-computing environment. However, none of these attempts has shown the same high performance as MPI.

- **Richness** MPI has more than one quality implementation. These implementations provide asynchronous communication, efficient message buffer management, efficient groups, and rich functionality. MPI includes a large set of collective communication operations, virtual topologies, and different communication modes. MPI supports libraries and heterogeneous networks as well.

1.3.3 MPI Programs

This section give you an introduction to a simple MPI program; the intent here is just to give you a visual image that you can relate to and refer back to if you have questions concerning things like these:

- What order should these calls be made in?
- What does the parameter list look like?

MPI routines

As you'll see, the basic programming of a MPI follows these general steps:

- Initialize for communications
- Communicate to share data between processes
- Exit in a "clean" fashion from the message-passing system

MPI has about 128 functions. However, a beginning programmer usually can make do with only six of these functions. These six functions, illustrated and discussed in the sample program, are:

- Initialize for communications

  MPI_INIT initializes for the MPI environment

  MPI_COMM_SIZE returns the number of processes

MPI_COMM_RANK returns this process's number (rank)

- Communicate to share data between processes

  MPI_SEND sends a message

  MPI_RECV receives a message

- Exit from the message-passing system

  MPI_FINALIZE

A MPI sample program

As you look at the code below, note the six basic calls to MPI routines.

```
#include <stdio.h>
#include "mpi.h"
int main(int argc, char **argv)
{
  int rank, size, tag, rc, i;
  MPI_Status status;
  char message[20];

  rc = MPI_Init(&argc, &argv);
  rc = MPI_Comm_size(MPI_COMM_WORLD, &size);
  rc = MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  tag = 100;

  if(rank == 0) {
    strcpy(message, "Hello, world");
    for (i=1; i<size; i++)
      rc = MPI_Send(message, 13, MPI_CHAR, i, tag, MPI_COMM_WORLD);
  }
  else
    rc = MPI_Recv(message, 13, MPI_CHAR, 0, tag, MPI_COMM_WORLD, &status);

  printf( "node %d : %.13s\n", rank,message);
  rc = MPI_Finalize();
}
```

To summarize the program:

This is a SPMD code, so copies of this program are running on multiple nodes. Each process initializes itself with MPI (MPI_INIT), determines the number of processes

(MPI_COMM_SIZE), and learns its rank (MPI_COMM_RANK). Then one process (with rank 0) sends messages in a loop (MPI_SEND), setting the destination argument to the loop index to ensure that each of the other processes is sent one message. The remaining processes receive one message (MPI_RECV). All processes then print the message, and exit from MPI (MPI_FINALIZE).

It is also worth noting what doesn't happen in this program. There is no routine that causes additional copies of the program to run. For MPI-1 all processes are started on the command line, in an implementation-specific manner.

1.3.4 MPI Messages

MPI messages consist of two basic parts: the actual data that you want to send/receive, and an envelope of information that helps to route the data. There are usually three calling parameters in MPI message-passing calls that describe the data, and another three parameters that specify the routing:

Message = data (3 parameters) + envelope (3 parameters)

Let's look at the data and envelope in more detail. We'll describe each parameter, and discuss whether these must be coordinated between the sender and receiver.

Data

When we use the term buffer to describe parameters in MPI calls, we mean a space in the computer's memory where the MPI messages are to be sent from or stored. So, in this context, a buffer is simply memory that the compiler has assigned to a variable (usually an array) in the program. To specify the buffer, one has to give three parameters in the MPI calls:

- **Startbuf:** the address where the data start. For example, it could be the start of an array in the program.
- **Count:** the number of elements (items) of data in the message. Note that it is an element, not byte. This makes it a portable code, since you don't have to worry

about different representations of data types on different computers.

The software implementation of MPI determines the number of bytes automatically. The count specified by the received call should be greater than or equal to the count specified by the sent call. If the length of the sent data is longer than the storage available in the receiving buffer, an error will occur.

- **Datatype:** the type of data to be transmitted. For example, it could be floating point. The types of data already defined for you are called "basic datatypes", and you can also define additional datatypes, which will be covered in later sections.

Envelope

As mentioned earlier, a message consists of the actual data and the message envelope. The envelope provides information on how to match send to receive. Three parameters used to specify the message envelope are:

- **Destination or source:** Arguments that are set to a rank in a communicator. Ranks range from 0 to (size-1), where size is the number of processes in the communicator.

  Destination is specified by the sender and is used to route the message to the appropriate process. The source is specified by the receive. Only messages coming from that source can be accepted by the receive call.

- **Tag:** An arbitrary number to help distinguish among messages. The tags specified by the sender and receiver must match.

- **Communicator:** The communicator specified by the sender must equal that specified by the receiver. We'll describe communicators in more depth later in the sections. For now, we'll just say that a communicator defines a communication "universe", and that processes may belong to more than one communicator. The first and most that you are going to work with is the predefined communicator MPI_COMM_WORLD, which includes all processes in the application.

1.3.5 Communicators

A communicator is an object that represents a group of processes and their communication medium or context. These processes exchange messages with each other to transfer data. In this context, communicators encapsulate their processes such that communication is restricted to processes only within the group.

The default communicators provided by MPI are MPI_COMM_WORLD and MPI_COMM_SELF. MPI_COMM_WORLD consists of all processes that are running when an application begins execution. Each process is the single member of its own MPI_COMM_SELF.

Many MPI applications depend upon MPI_Comm_size and MPI_Comm_rank to know the number of processes and the process rank within a given communicator.

- To determine the number of processes in a communicator named comm, use MPI_Comm_size(MPI_Comm comm, int *size);
- To determine the rank of each process in comm, use MPI_Comm_rank (MPI_Comm comm, int *rank); where rank is an integer between zero and (size - 1).

1.3.6 Summary

Although MPI provides an extensive, sometimes complex, set of calls, one can begin with just the six basic calls:

- MPI_INIT
- MPI_COMM_RANK
- MPI_COMM_SIZE
- MPI_SEND
- MPI_RECV
- MPI_FINALIZE

However, for programming convenience and optimization of your code, you should consider using other calls.

MPI Messages

MPI messages consist of two parts:

- data (startbuf, count, datatype)
- envelope (destination/source, tag, communicator)

The data defines the information to be sent or received. The envelope is used in routing messages to the receiver, and in matching sending calls to receiving calls.

Communicators

Communicators guarantee unique message spaces. In conjunction with process groups, they can be used to limit communication to a subset of processes.

1.4 Contributions of This Thesis

This thesis is part of the project that aims at the design and implemenation of the Concordia Parallel C programinng (CPC), compiler (CPCC), programming environment (CPPE) and systems simulator (CPSS) by a team of researchers led by Dr. Lixin Tao. My task aims particularly at the implementation of a message passing interface for CPC parallel process communication based on the MPI model. The research work consists in the implementation of the MPI libraries to the CPPE.

1.5 Thesis Outline

In chapter 2, we present a review of Message Passing Interface Implementations. The review makes an analytical comparison of different implementations including free distributed and commercial implementations, as a starting-point to build our CPPE-MPI implementation. Chapter 3 gives a detailed description of the Message Passing Interface that will provide a foundation for our MPI implementation for CPPE. In chapter 4, major components of the CPPE-MPI are described at a high level, and the design and

implementation of the code execution modules are described in details. The discussion focuses on major MPI routines, regarding point to point communication, collective communication, derived datatype, communicator and topology. In chapter 5, we discuss porting MPI application between CPPE and UNIX implementations and coding principles with MPI. At last, we give two applications in MPI – Jacobi's method and Bitonic Sort method to demenstrate our design and implementation. Chapter 6 provides a summary of the thesis and give suggestions for future work. Appendix A lists all the MPI programming examples created to run on the CPPE-MPI implementation. Appendix B provides a description of all routines that MPI supports as the convenience for CPPE-MPI user.

Chapter 2

2. MPI Implementation Survey

An MPI program consists of a set of processes and a logical communication medium connecting those processes. These processes may be the same program (SPMD - Single Program Multiple Data) or different programs (MPMD – Multiple Programs Multiple Data). The MPI memory model is logically distributed: an MPI process cannot directly access memory in another MPI process, and inter-process communication requires calling MPI routines in both processes. MPI defines a library of subroutines through which MPI processes communicate — this library is the core of MPI and implicitly defines the programming model.

The most important routines in the MPI library are the so-called "point-to-point" communication routines, which allow processes to exchange data cooperatively - one process sends data to another process, which receives the data. This cooperative form of communication is called "message passing."

The MPI standard is a specification, rather than a piece of software. What is specified is the application interface, not the implementation of that interface. In order to allow implementers to implement MPI efficiently, the MPI standard does not specify protocols, or require that implementations be able to inter-operate. Thus MPI can make sense in a wide range of environments, the standard does not specify how processes are created or destroyed, and does not even specify precisely what a process is. The most important considerations in the design of MPI were:

- **Portability.** An MPI application should require only recompilation to use a different MPI implementation. Furthermore, it should be possible to implement MPI on any MIMD (Multiple Instruction, Multiple Data) parallel computer. MPI should support (though not require) execution in heterogeneous environments.

- **Efficiency.** It should be possible to implement MPI efficiently. In particular, good MPI implementations should perform as well as proprietary "native" message passing libraries.
- **Robustness.** MPI should provide all the important functionality in "common practice," and some MPI should provide significant support for the development of parallel libraries.

2.1 Relation of MPI to PVM and HPF

In the context of software standards for parallel computing, there are two other terms - Parallel Virtual Machine (PVM) and High Performance Fortran (HPF). We will take a look at how MPI fits in the larger context.

PVM is a package of software that provides message passing functionality as well as infrastructure for building a virtual parallel computer out of a network of workstations 0. It is often thought of as a competitor of MPI, but it is actually a different beast. PVM is a research project of the University of Tennessee at Knoxville and Oak Ridge National Laboratory. While quite popular for writing message-passing programs, PVM is a vehicle for performing research in parallel computing rather than a parallel-computing standard. Compared with MPI, its weaknesses are also its strengths: it is not bound by an absolute requirement for backward compatibility; its design is not constrained to be efficient on any imaginable MIMD parallel architecture; there is no rigorous specification of PVM behavior. In some sense, the tradeoff is between efficiency and portability in MPI, and flexibility and adaptability in PVM.

Successful features of PVM are finding their way into MPI, though MPI is unlikely to provide any support for fault tolerance or a virtual distributed operating system in the near future. Moreover, since PVM is defined by a full implementation rather than by a specification, possibilities for interoperability in PVM are higher than in MPI.

HPF is an industry standard for the data parallel model of parallel computation. HPF was standardized a year earlier than MPI, and the successful HPF standardization

process was copied by the MPI Forum. Despite the conceptual appeal and simplicity of HPF, MPI is much more widely used than HPF for several reasons. These include:

- HPF is much more difficult to implement, and to implement efficiently. MPI, on the other hand, has benefited greatly from the large number of good implementations, including an implementation that was available at about the same time the standard was released.
- MPI is a more general model, and can be used to implement almost any parallel computation, while HPF is applicable only to certain types of problems.
- Obtaining high performance in a HPF program can be more difficult than would be expected from the superficial simplicity of the HPF model. It is an open question whether this is a fundamental obstacle or can be addressed by more mature compilers.

2.2 Freely Available Implementations

MPICH

Without question, the most important MPI implementation is MPICH, a freely available portable implementation of MPI developed at Argonne National Laboratory and Mississippi State University [21]. MPICH has played an important role in the development of MPI.

MPICH is the parent of a large number of commercial implementations of MPI. These include vendor-supported implementations from Digital, Sun, HP, SGI/Cray, NEC and Fujitsu. In some cases (e.g., SGI and HP) the implementation has evolved far from its roots; in others (e.g., Digital and Sun) the implementation is young and still close to MPICH. Only two of the major vendor-supported implementations are not directly derived from MPICH: the Cray T3D/E implementation (which derives from the CHIMP implementation) and the IBM SP implementation (for which MPICH still provided substantial inspiration). The HP implementation also has a second parent in LAM. MPICH is also the basis for most experimental and research versions of MPI.

The first version of MPICH was written during the MPI standardization process. The experiences of the MPICH authors provided important feedback to the MPI Forum. MPICH was released at approximately the same time as the original MPI 1.0 standard.

The portability of MPICH stems from its two-layer design. The bulk of MPICH code is device independent and is implemented on top of an Abstract Device Interface (ADI). The ADI interface hides most hardware-specific details, allowing MPICH to be easily ported to new architectures. The ADI design allows for efficient layering, and the device-independent top layer takes care of the majority of MPI syntax and semantics.

LAM - Local Area Multi-computer

The LAM implementation of MPI is a freely available and portable implementation developed at the Ohio Supercomputer Center [17]. LAM and MPICH are the two most important free options for running MPI on a network of workstations. LAM existed before MPI and was adopted to implement the MPI interface. LAM runs on many platforms, including RS6000, Irix 5, Irix 6, Linux86, HPUX, OSF/1 and Solaris.

LAM provides an infrastructure to turn a network of workstations (possibly heterogeneous) into a virtual parallel computer. A user-level daemon running on each node provides process management, including signal handling and I/O management.

LAM also provides extensive monitoring capabilities to support tuning and debugging. The XMPI graphic debug tool that comes with LAM has been adopted by many other MPI implemenations.

CHIMP - Common High-level Interface to Message Passing

The CHIMP project is based at the Edinburgh Parallel Computing Center [18]. Like LAM, CHIMP started off as an independent portable message-passing infrastructure and was later adapted to implement MPI. CHIMP is best known as the basis for the vendor-supplied optimized versions of MPI for the Cray T3D and T3E. Chimp is portable, running on many platforms including Solaris, Irix, AIX, OSF/1, and Meiko. To the best of this reviewer's knowledge, CHIMP is not in active development and is not widely used, at least in the U.S.

NT

Students at Mississippi State University have developed an MPICH ADI implementation for Microsoft Windows NT clusters. It is a demonstration implementation, rather than a high performance implementation.
See http://www.erc.msstate.edu/mpi/mpiNT.html for more information.

Win32

A researcher in Portugal has implemented the MPICH ADI device for Microsoft Windows.
See http://alentejo.dei.uc.pt/ fafe/w32mpi/ for more information.

Active Messages

A student at the University of California at Berkeley has implemented the MPICH ADI (ADI-2) on top of Generic Active Messages (GAM) and Active Messages II (AM2).
See http://now.cs.berkeley.edu/Fastcomm/MPI/ for more information.

Fast Messages

Students at the University of Illinois at Urbana Champaign have implemented the MPICH ADI on top of Fast Messages, which runs on PCs running NT or Linux with Myrinet or Winsock 32.
See http://www-csag.cs.uiuc.edu/projects/comm/mpi-fm.html for more in-formation.

Multithreaded (MT) Device

A researcher in Germany has implemented the MPICH ADI so that MPI "processes" are in fact threads on a multiprocessor machine. Communication between these processes can be done with a single copy.

2.3 Vendor Implementations

IBM

IBM has been a consistently strong supporter of MPI. IBM's implementation of MPI for its SP systems was one of the first vendor-supported MPI implementations. MPI has replaced IBM's proprietary library MPL as the preferred message-passing library on SP systems. The first optimized version of MPI available for SP systems, MPI-F, was a research prototype based on MPICH. The currently available implementation of MPI (hereafter referred to as IBM MPI) is rewritten from scratch [19].

IBM MPI runs on IBM SP systems and AIX workstation clusters in one of two modes. In User Space (US) mode, an MPI application has direct access to the SP high performance switch (if one exists). This provides the best performance, with the restriction that only one process may access the switch on each node. In IP mode, MPI processes communicate using IP — over the high performance switch if there exists, or over any other network if not. Latency (minimum message transfer time) in US mode is an order of magnitude lower than in IP mode.

HP

HP provides an implementation of MPI that runs on all current HP hardware, including the S-class and X-class Exemplar systems [20]. HP MPI was derived from MPICH, but also was significantly influenced by LAM. HP MPI uses whatever communication medium it has access to: TCP/IP between hosts, shared memory within a host, and a hardware data mover for long messages on Exemplar systems. HP MPI is interoperable among all supported HP systems. HP MPI is well tuned on the high-end systems, with both very low latency and high bandwidth on Exemplar servers. It has also been optimized to use shared memory to implement collective operations where possible, rather than layering on top of point-to-point routines. HP MPI is compliant with MPI 1.2.

Sun

The Sun implementation of MPI is quite recent. Version 2 is in beta release as of this writing and should be generally available soon. Version 1 was a repackaged MPICH. Sun MPI is derived from MPICH. For version 2, it has been integrated with a new Sun HPC environment and optimized for SCI, though it can run over any network using TCP. The Sun HPC environment is layered software that includes parallel job management. Users can launch (tmrun), examine (tmps) or kill (tmkill) parallel jobs. There is considerable flexibility in specifying where jobs are started, how standard input and output should be handled (approximately the same as the functionality in IBM MPI, plus a bit more), etc.

Digital

Digital is another newcomer to the MPI world, having recently released a version for clusters of Alpha SMP servers connected by Digital's proprietary Memory Channel interconnect [21]. Digital MPI is quite close to the original MPICH, with special optimizations for communication over local shared memory and over the memory channel.

Digital's implementation of the MPICH ADI uses a lower level communication layer, UMP (Universal Message Passing), that provides low-level communication functionality over the Memory Channel and over shared memory. For long messages, UMP uses a background thread to allow overlap of communication and computation.

SGI

Now that Silicon Graphics, Inc (SGI) has bought Cray Research Inc. (CRI), SGI has three separate MPI implementations for its three types of machines —parallel vector (e.g. J90/C90/T90), Irix (including Origin 2000), and T3E. These implementations all have different roots and are therefore treated as separate implementations here. SGI is in the process of merging at least two of the implementations. In each case, MPI is part of a package called MPT (Message Passing Toolkit) that also includes SGI/Cray's shmem library and PVM.

NEC SX-4

NEC MPI is another new implementation. NEC has experimented with several very different implementations. The one described here is just becoming available on the SX-4 as of this writing and should be standard on the SX-4 in the near future 0.

NEC MPI is a recent descendant of MPICH, starting from the ch lfshmem device, which was originally implemented for the SX-4. NEC MPI has been highly optimized for both a single-node SX-4, where MPI uses shared memory for communication, and a multi-node SX-4, where communication between nodes is done through the Inter-node Crossbar Switch (IXS).

NEC MPI is integrated with the VAMPIR and VAMPIR trace tools from Pallas, which allows users to visualization message trace information to optimize pro-grams 0.

Other features of the NEC implementation are at this time limited to what is available in MPICH. Because of its recent release, I have not had an opportunity to assess its usability.

Mercury Race

Hughes Aircraft Co. has implemented MPI for Mercury RACE systems 0. RACE MPI is derived from MPICH. There are a few interesting features of this implementation that are worth noting. On SHARC systems, where a "byte" (defined by ANSI C to be the size of a char) is 32 bits, not 8 bits, this implementation exposed a portability problem for MPI codes. The MPICH library has been modified to conserve as much space as possible. Only needed routines are linked, argument checking and strings are omitted.
Several collective operations have been optimized to use shared memory.

Hitachi

Hitachi provides an implementation of MPI based on MPICH for its SR2201 series computers. This implementation uses the SR2201's remote DMA facility.

NEC Cenju-3

NEC provides an MPICH device for its Cenju-3 computers. See: http://www.ccrl-nece.technopark.gmd.de/mpich/mpich-cenju3.html.

Alpha Data

Alpha Data provides an implementation of MPI for its AD66 systems. This implementation was developed jointly with the Edinburgh Parallel Computing Center and is presumably related to CHIMP.
See http://www.alphadata.co.uk/softhome.htm.

Fujitsu

MPI for the Fujitsu VPP machines has been recently developped by Pallas (http://www.pallas.de) for Fujitsu. This implementation is reported to be in final testing, but no further information is available. MPI for the Fujitsu AP1000 is available from Australian National University 0.
See http://cap.anu.edu.au/cap/projects/mpi/mpi.html for more information.

2.4 Summary

The discussions in the last several sections have illuminated the promise of MPI implementations. The primary message is that there are many MPI implementations out there, that they are well supported by vendors. The overwhelming majority of the MPI standard has been correctly implemented by all implementations.

MPI implementations discussed above are built over real hardware system, most of those are kind of expensive. Individual beginners are unlikely to have access to them. More important, those hardware based systems are hard to change to suit for research on performance study under different system configuration.

As we introduced in previous sections, CPPE offers a development environment superior to those that is available on real multiprocessors. It provides flexible and efficient software tools for developing parallel applications and optimizing their performance. It supports testing and debugging, as well as algorithmic and architectural performance evaluation and tuning so as to allow users to evaluate impacts of system and software factors on performance of parallel applications. With the help of optimal mapping functions for wormhole-routed networks, it is capable of locating and eliminating performance bottlenecks in the programs. It also provides accurate information about the timing and behavior of parallel applications and the underlying simulated architecture. These enable efficient parallel MPI programming.

We believe a MPI standard that is built on CPPE parallel virtual machine, that was implemented in C, and run virtually on any platform, will provide a great tool for

both beginners and senior MPI developers for developing high quality parallel programs. That also is the major purpose of this project.

Chapter 3

3. Message Passing Interface Standard

In this chapter, we give a detailed introduction of Message Passing Interface standard, to make it the base of our design and implementation in the next chapter.

3.1 Basic Concepts

Through Message Passing Interface (MPI), an application views its parallel environment as a static group of processes. This initial collection of processes is called the world group. A unique number, called a rank, is assigned to each member process from the sequence 0 through N-1, where N is the total number of processes in the world group. A member can query its own rank and size of the world group. Processes may all be running the same program (SPMD) or different programs (MIMD). The world group processes may be subdivided and create additional subgroups with a potentially different rank in each group.

A process sends a message to a destination rank in the desired group. Messages are further filtered by an arbitrary, user specified synchronization integer called tag.

An important feature of MPI is its ability to guarantee independent software developers that their choice of tag in a particular library will not conflict with that of any other independent developer or end user of the library. A further synchronization integer called a context is allocated by MPI and is automatically attached to every message. Thus, the four main synchronization variables in MPI are the source, the destination ranks, the tag and the context.

A communicator is an opaque MPI data structure that contains information on one group and that contains one context. A MPI_COMM_WORLD communicator is an argument to all MPI communication routines after a process is created and initialized.

Many applications require no other communicators beyond the world communicator - MPI_COMM_WORLD. If new subgroups or new contexts are needed, additional communicators must be created.

MPI constants, templates and prototypes are in the MPI header file, mpi.h.

- #include <mpi.h>

3.2 Initialization

MPI_Init	Initialize MPI state.
MPI_Finalize	Clean up MPI state.
MPI_Abort	Abnormally terminate.
MPI_Comm_size	Get group process count.
MPI_Comm_rank	Get my rank within process group.
MPI_Initialized	Has MPI been initialized?

Table 1: Initialization related MPI routines

The first MPI routine called by a program must be MPI_Init(). The command line arguments are passed to MPI_Init().

- MPI_Init(int *argc, char **argv[]);

A process ceases MPI operations with MPI_Finalize().

- MPI_Finalize(void);

In response to an error condition, a process can terminate itself and all members of a communicator with MPI_Abort(). The implementation may report the error code argument to the user in a manner consistent with the underlying operation system.

- MPI_Abort (MPI_Comm comm, int errcode);

Basic Parallel Information

Two numbers that are very useful to most parallel applications are the total numbers of parallel processes and self-process identification. This information is learned from the MPI_COMM_WORLD communicator using the routines MPI_Comm_size() and MPI_Comm_rank().

- MPI_Comm_size (MPI_Comm comm, int *size);
- MPI_Comm_rank (MPI_Comm comm, int *rank);

3.3 Blocking Point-to-Point Communication

MPI_Send	Send a message in standard mode.
MPI_Recv	Receive a message.
MPI_Get_count	Count the elements received.
MPI_Probe	Wait for message arrival.
MPI_Bsend	Send a message in buffered mode.
MPI_Ssend	Send a message in synchronous mode.
MPI_Rsend	Send a message in ready mode.
MPI_Buffer_attach	Attach a buffer for buffered sends.
MPI_Buffer_detach	Detach the current buffer.
MPI_Sendrecv	Send in standard mode then receive.
MPI_Sendrecv_replace	Send and receive from/to one area.
MPI_Get_elements	Count the basic elements received.

Table 2: Blocking point to point communication related MPI routines

This section focuses on point-to-point message passing routines. A point-to-point message is sent by one process and received by another process.

Send Modes

The issues of flow control and buffering present different choices in designing message passing primitives. MPI does not impose a single choice but instead offers four transmission modes that cover the synchronization, data transfer and performance needs of most applications. The mode is selected by the sender through four different send routines, all of them with identical argument lists. There is only one receive routine. The four send modes are:

- **Standard** The send completes when the system buffers the message (it is not obligated to do so) or when the message is received.
- **Buffered** The send completes when the message is buffered in application supplied space, or when the message is received.
- **Synchronous** The send completes when the message is received.
- **Ready** The send must not be started unless a matching receive has been started. The send completes immediately.

Standard Send

Standard mode satisfies the needs of most applications. A standard mode message is sent with MPI_Send().

- MPI_Send (void *buf, int count, MPI_Datatype dtype, int dest, int tag, MPI_Comm comm);

An MPI message is not merely a raw byte array. It is a count of typed elements. The element type may be a simple raw byte or a complex data structure. The source rank is the caller's. The destination rank and message tag is explicitly given. The context is a property of the communicator.

Standard Receive

A message in any mode is received with MPI_Recv().

- MPI_Recv (void *buf, int count, MPI_Datatype dtype, int source, int tag, MPI_Comm comm, MPI_Status *status);

Again the four synchronization variables are indicated with source and destination swapping places. The source rank and the tag can be ignored with the special values MPI_ANY_SOURCE and MPI_ANY_TAG. If both of these wildcards are used, the next message for the given communicator is received.

Status Object

An argument not present in MPI_Send() is the status object pointer. The status object is filled with useful information when MPI_Recv() returns. If the source and/or tag wildcards are used, the actual received source rank and/or message tag is accessible directly from the status object. status.MPI_SOURCE is the sender's rank, and status.MPI_TAG is the tag given by the sender.

Message Lengths

It is erroneous for an MPI program to receive a message longer than the specified receive buffer. It is completely acceptable to receive a message shorter than the specified receive buffer. If a short message arrives, the application queries the actual length of the message with MPI_Get_count().

- MPI_Get_count (MPI_Status *status, MPI_Datatype dtype, int *count);

The status object and MPI datatype are those provided to MPI_Recv(). The count returned is the number of elements received of the given datatype.

Probe

Sometimes it is impractical to pre-allocate a receive buffer. MPI_Probe() synchronizes a message and returns information about it without actually receiving it. Only synchronization variables and the status object are provided as arguments. MPI_Probe() does not return until a message is synchronized.

- MPI_Probe (in source, int tag, MPI_Comm comm, MPI_Status *status);

3.4 Nonblocking Point-to-Point Communication

MPI_Isend	Begin to send a standard message.
MPI_Irecv	Begin to receive a message.
MPI_Wait	Complete a pending request.
MPI_Test	Check or complete a pending request.
MPI_Iprobe	Check message arrival.
MPI_Ibsend	Begin to send a buffered message.
MPI_Issend	Begin to send a synchronous message.
MPI_Irsend	Begin to send a ready message.
MPI_Request_free	Free a pending request.
MPI_Waitany	Complete any one request.
MPI_Testany	Check or complete any one request.
MPI_Waitall	Complete all requests.
MPI_Testall	Check or complete all requests.
MPI_Waitsome	Complete one or more requests.
MPI_Testsome	Check or complete one or more requests.
MPI_Cancel	Cancel a pending request.
MPI_Test_cancelled	Check if a pending request was cancelled.

Table 3: Nonblocking point to point communication related MPI routines

The term "non-blocking" in MPI means that the routine returns immediately and may only have started the message transfer operation, not necessarily completed it. The four blocking send routines and one blocking receive routine all have non-blocking counterparts. The non-blocking routines have an extra output argument -a request object. The request is later passed to one of a suite of completion routines.
MPI_Isend() begins a standard non-blocking message send.

- MPI_Isend (void *buf, int count, MPI_Datatype dtype, int dest, int tag, MPI_Comm comm, MPI_Request *req);

Likewise, MPI_Irecv() begins a non-blocking message receive.

- MPI_Irecv (void *buf, int count, MPI_Datatype dtype, int source, int tag, MPI_Comm comm, MPI_Request *req);

Request Completion

Both routines accept arguments with the same meaning as their blocking counterparts. When the application wishes to complete a non-blocking send or receive, a completion routine is called with the corresponding request. The MPI_Test() routine is non-blocking and the MPI_Wait() routine is blocking. Other completion routines operate on multiple requests.

- MPI_Test (MPI_Request *req, int *flag, MPI_Status *status);
- MPI_Wait (MPI_Request *req, MPI_Status *status);

MPI_Test() returns a flag in an output argument that indicates if the request completed. If true, the status object argument is filled with information. If the request was a receive operation, the status object is filled as in MPI_Recv(). Since MPI_Wait() blocks until completion, the status object argument is always filled.

Probe

MPI_Iprobe() is the nonblocking counterpart of MPI_Probe(), but it does not return a request object since it does not begin any message transfer that would need to complete. It sets the flag argument, which indicates the presence of a matching message (for a subsequent receives).

- MPI_Iprobe (int source, int tag, MPI_Comm comm, int *flag, MPI_Status *status);

Programmers should not consider the non-blocking routines simply as fast versions of the blocking calls and therefore the preferred choice in all applications. Some implementations cannot take advantage of the opportunity to optimize performance offered by the non-blocking routines. In order to preserve the semantics of the message passing interface, some implementations may be even slower with non-blocking transfers. Programmers should have a clear and substantial computation overlap before considering non-blocking routines.

3.5 Message Datatypes

MPI_Type_vector	Create a strided homogeneous vector.
MPI_Type_struct	Create a heterogeneous structure.
MPI_Address	Get absolute address of memory location.
MPI_Type_commit	Use datatype in message transfers.
MPI_Pack	Pack element into contiguous buffer.
MPI_Unpack	Unpack element from contiguous buffer.
MPI_Pack_size	Get packing buffer size requirement.
MPI_Type_continuous	Create contiguous homogeneous array.
MPI_Type_hvector	Create vector with byte displacement.
MPI_Type_indexed	Create a homogeneous structure.
MPI_Type_hindexed	Create an index with byte displacements.
MPI_Type_extent	Get range of space occupied by a datatype.
MPI_Type_size	Get amount of space occupied by a datatype.
MPI_Type_lb	Get displacement of datatype's lower bound.
MPI_Type_ub	Get displacement of datatype's upper bound.
MPI_Type_free	Free a datatype.

Table 4: Message datatype related MPI routines

Heterogeneous computing requires that message data be typed or described somehow so that its machine representation can be converted as necessary between computer architectures. MPI can thoroughly describe message datatypes, from the simple primitive machine types to complex structures, arrays and indices.

All the message-passing routin es accept a datatype argument, whose C typedef is MPI_Datatype. For example, recall MPI_Send(). Message data is specified as a number of elements of a given type.

Several MPI_Datatype values, covering the basic data units on most computer architectures, are predefined:

MPI_CHAR	signed char
MPI_SHORT	signed short
MPI_INT	signed int
MPI_LONG	signed long
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	a raw byte

Table 5: MPI data type

The number of bytes occupied by these basic datatypes follows the corresponding C definition. Thus, MPI_INT could occupy four bytes on one machine and eight bytes on another machine. A message count of one MPI_INT specified by both sender and receiver would, in one direction, require padding and always be correct. In the reverse direction, the integer may not be representable in the lesser number of bytes and the communication will fail.

Derived Datatypes

Derived datatypes are built by combining basic datatypes, or previously built derived datatypes. A derived datatype describes a memory layout, which consists of multiple arrays of elements. A generalization of this capability is that the four varieties of constructor routines offer more or less control over array length, array element datatype and array displacement.

Contiguous	one array length, no displacement, one datatype
Vector	one array length, one displacement, one datatype
Indexed	multiple array lengths, multiple displacements, one datatype
Structure	multiple everything

Table 6: MPI routines that derive new datatype

Strided Vector Datatype

Consider a two dimensional matrix with R rows and C columns stored in row major order. The application wishes to communicate one entire column. A vector-derived datatype fits the requirement.

- MPI_Type_Vector (int count, int blocklength, int stride, MPI_Datatype oldtype, MPI_Datatype *newtype);

Assuming the matrix elements are of MPI_INT, the arguments for the stated requirement would be:

```
int R, C;
MPI_Datatype newtype;
MPI_Type_vector(R, 1, C, MPI_INT, &newtype);
MPI_Type_commit(&newtype);
```

The count of blocks (arrays) is the number of elements in a column (R).
Each block contains just one element and the elements are strided (displaced) from each other by the number of elements in a row (C).

Structure Datatype

An arbitrary record whose template is a C structure is a common message form. The most flexible MPI derived datatype, the structure, is required to describe the memory layout.

- MPI_Type_struct (int count, int blocklengths[], MPI_Aint displacements[], MPI_Datatype dtypes[], MPI_Datatype *newtype);

In the following code fragment, a C struct of diverse fields is described with MPI_Type_struct() in the safest, most portable manner.

```
// nontrivial structure
struct cell {
double energy;
char flags;
float coord[3];
};

// We want to be able to send arrays of this datatype.
struct cell cloud[2];

// New datatype for cell struct
 MPI_Datatype celltype;
```

Note that this datatype is not sufficient to send multiple columns from the matrix, since it does not presume the final displacement between the last element of the first column and the first element of the second column. One solution is to use MPI_Type_struct() and MPI_UB.

```
int blocklengths[4] = {1, 1, 3, 1};
MPI_Aint base;
MPI_Aint displacements[4];
MPI_Datatype types[4] = {MPI_DOUBLE, MPI_CHAR, MPI_FLOAT, MPI_UB};
```

```
MPI_Address(&cloud[0].energy, &displacement[0]);
MPI_Address(&cloud[0].flags, &displacement[1]);
MPI_Address(&cloud[0].coord, &displacement[2]);
MPI_Address(&cloud[1].energy, &displacement[3]);
base = displacement[0];
for (i = 0; i < 4; ++i) displacement[i] = base;
MPI_Type_struct(4, blocklengths, displacements, types, &celltype);
MPI_Type_commit(&celltype);
```

The displacements in a structure datatype are byte offsets from the first storage location of the C structure. Without guessing the compiler's policy for packing and alignment in a C structure, the MPI_Address() routine, together with some pointer arithmetic, is the best way to get the precise values.

MPI_Address() simply returns the absolute address of a location in memory. The displacement of the first element within the structure is zero.

When transferring arrays of a given datatype (by specifying a count greater than 1 in MPI_Send(), for example), MPI assumes that the array elements are stored contiguously. If necessary, a gap can be specified at the end of the derived datatype memory layout by adding an artificial element of type MPI_UB to the datatype description and giving it a displacement, which extends to the first byte of the second element in an array.

MPI_Type_Commit() separates the datatypes to be used to transfer messages from the intermediate ones scaffolded on the way to some very complicated datatype. A derived datatype must be committed before being used in communication.

Packed Datatype

The description of a derived datatype is fixed after creation at runtime. If any slight detail change accue, (say, in the blocklength of a particular field in a structure.) a new datatype is required. In addition to the tedium of creating many derived datatypes, a receiver may not know in advance which of a nearly identical suite of datatypes will arrive in the next message. MPI's solution is packing and unpacking routines that incrementally assemble

and disassemble a contiguous message buffer. The packed message has the special MPI datatype, MPI_PACKED, and is transferred with a count equal to its length in bytes.

- MPI_Pack_size (int incount, MPI_Datatype dtype, MPI_Comm comm, int *size);

MPI_Pack_size() returns the packed message buffer size requirement for a given datatype. This may be greater than one would expect from the type description due to hidden, implementation dependent packing overhead.

- MPI_Pack (void *inbuf, int incount, MPI_Datatype dtype, void *outbuf, int outsize, int *position, MPI_Comm comm);

Contiguous blocks of homogeneous elements are packed one at a time with MPI_Pack(). After each call, the current location in the packed message buffer is updated. The "in" data are the elements to be packed and the "out" data is the packed message buffer. The outsize is always the maximum size of the packed message buffer, to guard against overflow.

- MPI_Unpack (void *inbuf, int insize, int *position, void *outbuf, int outcount, MPI_Datatype datatype, MPI_Comm comm);

MPI_Unpack() is the natural reverse of MPI_Pack() where the "in" data is the packed message buffer and the "out" data are the elements to be unpacked.

Consider a networking application that is transferring a variable length message consisting of a count, several (count) Internet addresses as four byte character arrays and an equal number of port numbers as shorts.

```
#define MAXN 100
unsigned char addrs[MAXN][4];
short ports[MAXN];
```

In the following code fragment, a message is packed and sent based on a given count.

```
unsigned int membersize, maxsize;
int position;
int nhosts;
int dest, tag;
char *buffer;

// Do this once.
MPI_Pack_size(1, MPI_INT, MPI_COMM_WORLD, &membersize);
maxsize = membersize;
MPI_Pack_size(MAXN * 4, MPI_UNSIGNED_CHAR, MPI_COMM_WORLD,
&membersize);
maxsize += membersize;
MPI_Pack_size(MAXN, MPI_SHORT, MPI_COMM_WORLD, &membersize);
maxsize += membersize;
buffer = malloc(maxsize);

// Do this for every new message.
nhosts = /* some number less than MAXN */ 50;
position = 0;
MPI_Pack(nhosts, 1, MPI_INT, buffer, maxsize, &position,
MPI_COMM_WORLD);
MPI_Pack(addrs, nhosts * 4, MPI_UNSIGNED_CHAR, buffer, maxsize,
&position, MPI_COMM_WORLD);
MPI_Pack(ports, nhosts, MPI_SHORT, buffer, maxsize, &position,
MPI_COMM_WORLD);
MPI_Send(buffer, position, MPI_PACKED, dest, tag, MPI_COMM_WORLD);
```

A buffer is allocated once to contain the maximum size of a packed message. In the following code fragment, a message is received and unpacked, based on a count packed into the beginning of the message.

```
int src;
int msgsize;
MPI_Status status;
MPI_Recv(buffer, maxsize, MPI_PACKED, src, tag, MPI_COMM_WORLD,
&status);
position = 0;
MPI_Get_count(&status, MPI_PACKED, &msgsize);
MPI_Unpack(buffer, msgsize, &position, &nhosts, 1, MPI_INT,
MPI_COMM_WORLD);
MPI_Unpack(buffer, msgsize, &position, addrs, nhosts * 4,
MPI_UNSIGNED_CHAR, MPI_COMM_WORLD);
MPI_Unpack(buffer, msgsize, &position, ports, nhosts, MPI_SHORT,
MPI_COMM_WORLD);
```

3.6 Collective Message-Passing

MPI_Bcast	Send one message to all group members.
MPI_Gather	Receive and concatenate from all members.
MPI_Scatter	Separate and distribute data to all members.
MPI_Reduce	Combine messages from all members.
MPI_Barrier	Wait until all group members reach this point.
MPI_Gatherv	Vary counts and buffer displacements.
MPI_Scatterv	Vary counts and buffer displacements.
MPI_Allgather	Gather and then broadcast.
MPI_Allgatherv	Variably gather and then broadcast.
MPI_Alltoall	Gather and then scatter.
MPI_Alltoallv	Variably gather and then scatter.
MPI_Op_create	Create reduction operation.
MPI_Allreduce	Reduce and then broadcast.
MPI_Reduce_scatter	Reduce and then scatter.
MPI_Scan	Perform a prefix reduction.

Table 7: Collective message passing related MPI routines

Collective operations consist of many point-to-point messages, which happen more or less concurrently (depending on the operation and the internal algorithm) and involve all processes in a given communicator. Every process must call the same MPI collective routine. Most of the collective operations are variations and/or combinations of four primitives: broadcast, gather, scatter and reduce.

Broadcast

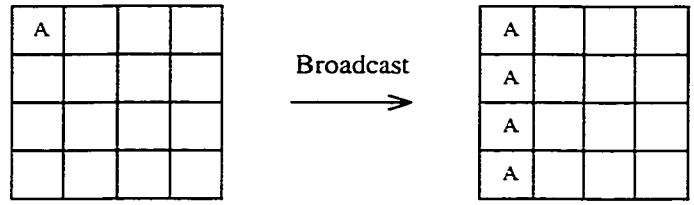


Figure 3: Broadcast message passing communication

- MPI_Bcast (void *buf, int count, MPI_Datatype dtype, int root, MPI_Comm comm);

In the broadcast operation, all processes specify the same root process, whose buffer contents will be sent. Processes other than the root, specify receive buffers. After the operation, all buffers contain the message from the root process.

Scatter

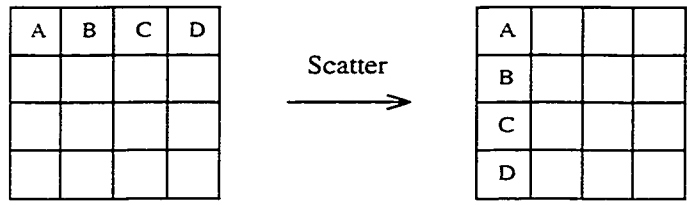


Figure 4: Scatter message passing communication

- MPI_Scatter (void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm);

MPI_Scatter() is also a one-to-many collective operation. All processes specify the same receive count. The send arguments are only significant to the root process, whose buffer

actually contains sendcount * N elements of the given datatype, where N is the number of processes in the given communicator. The send buffer will be divided equally and dispersed to all processes (including itself). After the operation, the root has sent sendcount elements to each process in ascending rank order. Rank 0 receives the first sendcount elements from the send buffer. Rank 1 receives the second send-count elements from the send buffer, and so on.

Gather

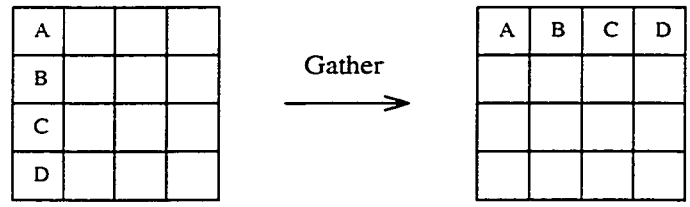


Figure 5: Gather message passing communication

- MPI_Gather (void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm);

MPI_Gather() is a many-to-one collective operation and is a complete reverse of the description of MPI_Scatter().

Reduce

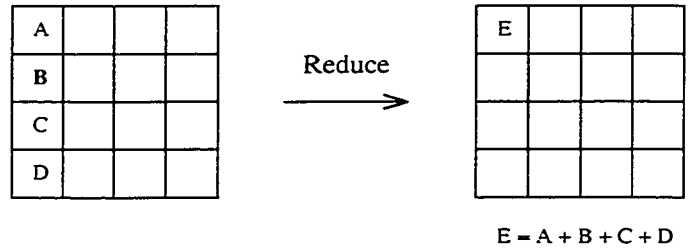


Figure 6: Reduce message passing communication

- MPI_Reduce (void *sendbuf, void *recvbuf, int count, MPI_Datatype dtype, MPI_Op op, int root, MPI_Comm comm);

MPI_Reduce() is also a many-to-one collective operation. All processes specify the same count and reduction operation. After the reduction, all processes have sent count elements from their send buffer to the root process.

Elements from corresponding send buffer locations are combined pair wise to yield a single corresponding element in the root process's receive buffer. The full reduction expression over all processes is always associative and may or may not be commutative. Application specific reduction operations can be defined at runtime. MPI provides several predefined operations, all of which are commutative. They can be used only with sensible MPI predefined datatypes.

MPI_MAX	maximum
MPI_MIN	minimum
MPI_SUM	sum
MPI_PROD	product
MPI_LAND	logical and
MPI_BAND	bitwise and
MPI_LOR	logical or
MPI_BOR	bitwise or
MPI_LXOR	logical exclusive or
MPI_BXOR	bitwise exclusive or

Table 8: Reduce operators

The following code fragment illustrates the primitive collective operations together in the context of a statically partitioned regular data domain (e.g., 1-D array). The global domain information is initially obtained by the root process (e.g., rank 0) and is broadcast

to all other processes. The initial dataset is also obtained by the root and is scattered to all processes. After the computation phase, a global maximum is returned to the root process followed by the new dataset itself.

```
// parallel programming with a single control process
int root;
int rank, size;
int i;
int full_domain_length;
int sub_domain_length;
double *full_domain, *sub_domain;
double local_max, global_max;
root = 0;
MPI_Comm_size(MPI_COMM_WORLD, &size);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);

// Root obtains full domain and broadcasts its length.
if (rank == root) {
get_full_domain(&full_domain, &full_domain_length);
}
MPI_Bcast(&full_domain_length, 1 MPI_INT, root, MPI_COMM_WORLD);

// Distribute the initial dataset.
sub_domain_length = full_domain_length / size;
sub_domain = (double *) malloc(sub_domain_length * sizeof(double));

MPI_Scatter(full_domain, sub_domain_length, MPI_DOUBLE, sub_domain,
sub_domain_length, MPI_DOUBLE, root, MPI_COMM_WORLD);

// Compute the new dataset.
compute(sub_domain, sub_domain_length, &local_max);

// Reduce the local maxima to one global maximum at the root.
MPI_Reduce(&local_max, &global_max, 1, MPI_DOUBLE, MPI_MAX, root,
MPI_COMM_WORLD);

// Collect the new dataset.
MPI_Gather(sub_domain, sub_domain_length, MPI_DOUBLE, full_domain,
sub_domain_length, MPI_DOUBLE, root, MPI_COMM_WORLD);
```

3.7 Creating Communicators

MPI_Comm_dup	Duplicate communicator with new context.
MPI_Comm_split	Split into categorized sub-groups.
MPI_Comm_free	Release a communicator.
MPI_Comm_remote_size	Count intercomm. remote group members.
MPI_Intercomm_merge	Create an intracomm. from an intercomm.
MPI_Comm_compare	Compare two communicators.
MPI_Comm_create	Create a communicator with a given group.
MPI_Comm_test_inter	Test for intracommunicator or intercommunicator.
MPI_Intercomm_create	Create an intercommunicator.
MPI_Group_size	Get number of processes in group.
MPI_Group_rank	Get rank of calling process.
MPI_Group_translate_ranks	Processes in group A have what ranks in B?
MPI_Group_compare	Compare membership of two groups.
MPI_Comm_group	Get group from communicator.
MPI_Group_union	Create group with all members of 2 others.
MPI_Group_intersection	Create with common members of 2 others.
MPI_Group_difference	Create with the complement of intersection.
MPI_Group_incl	Create with specific members of old group.
MPI_Group_excl	Create with the complement of incl.
MPI_Group_range_incl	Create with ranges of old group members.
MPI_Group_range_excl	Create with the complement of range_incl.
MPI_Group_free	Release a group object.

Table 9: Communication and group create related MPI routines

A communicator could be described simply as a process group. Its creation is synchronized and its membership is static. There is no period in user code where a communicator is created but not all its members have joined. These qualities make communicators a solid parallel programming foundation. Three communicators are

prefabricated before the user code is first called: MPI_COMM_WORLD, MPI_COMM_SELF and MPI_COMM_PARENT.

Communicators carry a hidden synchronization variable called the context. If two processes agree on source rank, destination rank and message tag, but use different communicators, they will not synchronize. The extra synchronization means that the global software industry does not have to divide, allocate or reserve tag values. When writing a library or a module of an application, it is a good idea to create new communicators, and hence a private synchronization spaces. The simplest MPI routine for this purpose is MPI_Comm_dup(), which duplicates everything in a communicator, particularly the group membership, and allocates a new context.

- MPI_Comm_dup (MPI_comm comm, MPI_comm *newcomm);

Applications may wish to split into many subgroups, sometimes for data parallel convenience (i.e. a row of a matrix), sometimes for functional grouping (i.e. multiple distinct programs in dataflow architecture). The group membership can be extracted from the communicator and manipulated by an entire suite of MPI routines. The new group can then be used to create a new communicator. MPI also provides a powerful routine,

MPI_Comm_split() starts with a communicator and results in one or more new communicators. It combines group splitting with communicator creation and is sufficient for many common application requirements.

- MPI_Comm_split (MPI_comm comm, int color, int key, MPI_Comm *newcomm);

The color and key arguments guide the group splitting. There will be one new communicator for each value of color. Processes providing the same value for color will be grouped in the same communicator. Their ranks in the new communicator are determined by sorting the key arguments. The lowest value of key will become rank 0. Ties are broken by rank in the old communicator. To preserve relative order from the old communicator, simply use the same key everywhere.

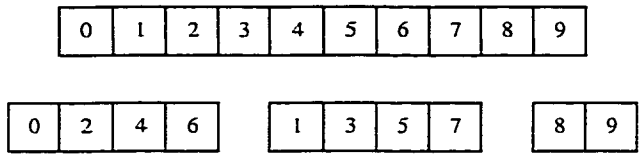


Figure 7: Communicator Split

A communicator is released by MPI_Comm_free(). Underlying system resources may be conserved by means of releasing unwanted communicators.

- MPI_Comm_free (MPI_Comm *comm);

Intercommunicators

An intercommunicator contains two groups: a local group in which the owning process is a member and a remote group of separate processes. The remote process group has the mirror image intercommunicator - the groups are flipped. Sp awning new processes creates an intercommunicator. MPI_Intercomm_merge() creates an intracommunicator (the common form with a single group) from an intercommunicator. This is often done to permit collective operations, which can only be done on intracommunicators.

- MPI_Intercomm_merge (MPI_Comm intercomm, int high, MPI_Comm *newintracomm);

The new intracommunicator group contains the union of the two groups of the intercommunicator. The operation is collective over both groups. Rank ordering within the two founding groups is maintained. Ordering between the two founding groups is controlled by the high parameter, a Boolean value. The intercommunicator group that sets this parameter true will occupy the higher ranks in the intracommunicator.

The number of members in the remote group of an intercommunicator is obtained by MPI_Comm_remote_size().

- MPI_Comm_remote_size (MPI_Comm comm, int *size);

3.8 Process Topologies

MPI_Cart_create	Create cartesian topology communicator.
MPI_Dims_create	Suggest balanced dimension ranges.
MPI_Cart_rank	Get rank from cartesian coordinates.
MPI_Cart_coords	Get cartesian coordinates from rank.
MPI_Cart_shift	Determine ranks for cartesian shift.
MPI_Cart_sub	Split into lower dimensional sub-grids.
MPI_Graph_create	Create arbitrary topology communicator.
MPI_Topo_test	Get type of communicator topology.
MPI_Graphdims_get	Get number of edges and nodes.
MPI_Graph_get	Get edges and nodes.
MPI_Cartdim_get	Get number of dimensions.
MPI_Cart_get	Get dimensions, periodicity and local coordinates.
MPI_Graph_neighbors_count	Get number of neighbors in a graph topology.
MPI_Graph_neighbors	Get neighbor ranks in a graph topology.
MPI_Cart_map	Suggest new ranks in an optimal cartesian mapping.
MPI_Graph_map	Suggest new ranks in an optimal graph mapping.

Table 10: Process topology related MPI routines

MPI is a process oriented programming model that is independent of underlying nodes in a parallel computer. Nevertheless, to enhance performance, the data movement patterns in a parallel application should match, as closely as possible, the communication topology of the hardware. Since it is difficult for compilers and message passing systems to guess at an application's data movement, MPI allows the application to supply a

topology to a communicator, in the hope that the MPI implementation will use that information to identify processes in an optimal manner.

For example, if the application is dominated by Cartesian communication and the parallel computer has a cartesian topology, it is preferable to align the distribution of data with the machine, and not blindly place any data coordinate at any node coordinate.

MPI provides two types of topologies, the ubiquitous cartesian grid, and an arbitrary graph. Topology information is attached to a communicator by creating a new communicator. MPI_Cart_create() does this for the cartesian topology.

- MPI_Cart_create (MPI_Comm oldcomm, int ndims, int *dims, int *periods, int reorder, MPI_Comm *newcomm);

The essential information for a cartesian topology is the number of dimensions, the length of each dimension and a periodicity flag (does the dimension wrap around?) for each dimension. The reorder argument is a flag that indicates if the application will allow a different ranking in the new topology communicator. Reordering may make coordinate calculation easier for the MPI implementation.

With a topology enhanced communicator, the application will use coordinates to decide source and destination ranks. Since MPI communication routines still use ranks, the coordinates must be translated into a rank and vice versa. MPI eases this translation with MPI_Cart_rank() and MPI_Cart_coords().

- MPI_Cart_rank (MPI_comm comm, int *coords, int *rank);
- MPI_Cart_coords (MPI_Comm comm, int rank, int maxdims, int *coords);

To further assist process identification in cartesian topology applications, MPI_Cart_shift() returns the ranks corresponding to common neighbourly shift communication. The direction (dimension) and relative distance are input arguments and two ranks are output arguments, one on each side of the calling process along the given direction. Depending on the periodicity of the cartesian topology associated with the

given communicator, one or both ranks may be returned as MPI_PROC_NULL, indicating a shift off the edge of the grid.

- MPI_Cart_shift (MPI_Comm comm, int direction, int distance, int *rank_source, *int rank_dest);

Consider a two-dimensional cartesian dataset. The following code skeleton establishes a corresponding process topology for any number of processes, and then creates a new communicator for collective operations on the first column of processes. Finally, it obtains the ranks that hold the previous and next rows, which would lead to data exchange.

```
int mycoords[2];
int dims[2];
int periods[2] = {1, 0};
int rank_prev, rank_next;
int size;
MPI_Comm comm_cart;
MPI_Comm comm_col1;

// Create communicator with 2D grid topology.
MPI_Comm_size(MPI_COMM_WORLD, &size);
MPI_Dims_create(size, 2, dims);
MPI_Cart_create(MPI_COMM_WORLD, 2, dims, periods, 1, &comm_cart);

 //Get local coordinates.
MPI_Comm_rank(comm_cart, &rank);
MPI_Cart_coords(comm_cart, rank, 2, mycoords);

// Build new communicator on first column.
if (mycoords[1] == 0) {
MPI_Comm_split(comm_cart, 0, mycoords[0], &comm_col1);
} else {
MPI_Comm_split(comm_cart, MPI_UNDEFINED, 0, &comm_col1);
}

// Get the ranks of the next and previous rows, same column.
MPI_Cart_shift(comm_cart, 0, 1, &rank_prev, &rank_next);
```

MPI_Dims_create() suggests the most balanced (``square'') dimension ranges for a given number of nodes and dimensions.

A good reason for building a communicator over a subset of the grid, in this case the first column in a mesh, is to enable the use of collective operations.

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

Figure 8: Cartesian Topology

0,0	0,1	0,2	0.3
1,0	1,1	1,2	1,3
2,0	2,1	2,2	2,3
3.0	3,1	3,2	3,3

Figure 9: Cartesian Coordinator

3.9 Process Creation

MPI_Spawn	Start copies of one program.
MPI_Spawn_multiple	Start multiple programs.
MPI_Port_open	Obtain a connection point for a server.
MPI_Port_close	Release a connection point.
MPI_Accept	Accept a connection from a client.
MPI_Connect	Make a connection to a server.
MPI_Name_publish	Publish a connection point under a service name.
MPI_Name_unpublish	Stop publishing a connection point.
MPI_Name_get	Get connection point from service name.
MPI_Info_create	Create a new info object.
MPI_Info_set	Store a key/value pair to an info object.
MPI_Info_get	Read the value associated with a stored key.
MPI_Info_get_valuelen	Get the length of a key value.
MPI_Info_get_nkeys	Get number of keys stored with an info object.
MPI_Info_get_nthkey	Get the key name in a sequence position.
MPI_Info_dup	Duplicate an info object.
MPI_Info_free	Destroy an info object.
MPI_Info_delete	Remove a key/value pair from an info object.

Table 11: Process creation related MPI routines

Due to the static nature of process groups in MPI (a virtue), process creation must be done carefully. Process creation is a collective operation over a given communicator. A group of processes are created by one call to MPI_Spawn(). The child processes start up, initialize and communicate in the traditional MPI way. They must begin by calling MPI_Init(). The child group has its own MPI_COMM_WORLD, which is distinct from the world communicator of the parent group.

- MPI_Spawn (char program[], char *argv[], int maxprocs, MPI_Info info, int root, MPI_Comm, parents, MPI_Comm *children, int errs[]);

How do the parents communicate with their children? The natural mechanism for communication between two groups is the intercommunicator. An intercommunicator, whose remote group contains the children, is returned to the parents in the second communicator argument of MPI_Spawn(). The children get the mirror communicator, whose remote group contains the parents, as the predefined communicator MPI_COMM_PARENT. In the application's original process world that has no parent, the remote group of MPI_COMM_PARENT is of size 0.

The maxprocs parameter is the number of copies of the single program that will be created. Each process will be passed command line arguments consisting of the program name followed by the arguments specified in the argv parameter. (The argv parameter should not contain the program name.) The program name, maxprocs and argv are only significant in the parent process whose rank is given by the root parameter. The result of each individual process spawn is returned through the errs parameter, an array of MPI error codes.

Portable Resource Specification

New processes require resources, beginning with a processor. The specification of resources is a natural area where the MPI abstraction succumbs to the underlying operating system and its entire domestic customs and conventions. It is thus difficult if not impossible for an MPI application to make a detailed resource specification and remain portable. The info parameter to MPI_Spawn is an opportunity for the programmer to choose control over portability. MPI implementations are not required to interpret this argument. Thus the only portable value for the info parameter is MPI_INFO_NULL.

Consult each MPI implementation's documentation for (nonportable) features within the info parameter and for the default behavior with MPI_INFO_NULL.

A common and fairly abstract resource requirement is simply to fill the available processors with processes. MPI makes an attempt, with no guarantees of accuracy, to supply that information through a predefined attribute called MPI_UNIVERSE_SIZE, which is cached on MPI_COMM_WORLD. In typical usage, the application would subtract the value associated with MPI_UNIVERSE_SIZE from the current number of

processes, often the size of MPI_COMM_WORLD. The difference is the recommended value for the maxprocs parameter of MPI_Spawn().

3.10 Miscellaneous MPI Features

MPI_Errhandler_create	Create custom error handler.
MPI_Errhandler_set	Set error handler for communicator.
MPI_Error_string	Get description of error code.
MPI_Error_class	Get class of error code.
MPI_Abort	Abnormally terminate application.
MPI_Attr_get	Get cached attribute value.
MPI_Wtime	Get wall clock time.
MPI_Errhandler_get	Get error handler from communicator.
MPI_Errhandler_free	Release custom error handler.
MPI_Get_processor_name	Get the caller's processor name.
MPI_Wtick	Get wall clock timer resolution.
MPI_Get_version	Get the MPI version numbers.
MPI_Keyval_create	Create a new attribute key.
MPI_Keyval_free	Release an attribute key.
MPI_Attr_put	Cache an attribute in a communicator.
MPI_Attr_delete	Remove cached attribute.

Table 12: Miscellaneous MPI routines

Error Handling

An error handler is a software routine, which is called when an error occurs during some MPI operation. One handler is associated with each communicator and is inherited by created communicators, which derive from it. When an error occurs in an MPI routine that uses a communicator, that communicator's error handler is called. An application's

initial communicator, MPI_COMM_WORLD, gets a default built-in handler, MPI_ERRORS_ARE_FATAL, which aborts all tasks in the communicator.

An application may supply an error handler by first creating an MPI error handler object from a user routine.

- MPI_Errhandler_create (void (*function)(), MPI_Errhandler *errhandler);

The first parameter is the handler's communicator and the second is the error code describing the problem.
void function (MPI_Comm *comm, int *code, ...);

The error handler object is then associated with a communicator by MPI_Errhandler_set().

- MPI_Errhandler_set (MPI_Comm comm, MPI_Errhandler errhandler);

A second built-in error handler is MPI_ERRORS_RETURN, which does nothing and allows the error code to be returned by the offending MPI routine where it can be tested and acted upon. In C the error code is the return value of the MPI function.

- MPI_Error_string (int code, char *errstring, int *resultlen);

Error codes are converted into descriptive strings by MPI_Error_string(). The user provides space for the string that is a minimum of MPI_MAX_ERROR_STRING characters in length. The actual length of the returned string is returned through the resultlen argument.

MPI defines a list of standard error codes (also called error classes) that can be examined and acted upon by portable applications. All additional error codes, specific to the implementation, can be mapped to one of the standard error codes. The idea is that

additional error codes are variations on one of the standard codes, or members of the same error class. Two standard error codes catch any additional error code that does not fit this intent: MPI_ERR_OTHER (doesn't fit but convert to string and learn something) and MPI_ERR_UNKNOWN (no clue). Again, the goal of this design is portable, intelligent applications.

The mapping of error code to standard error code (class) is done by MPI_Error_class().

- MPI_Error_class (int code, int class);

Attribute Caching

MPI provides a mechanism for storing arbitrary information with a communicator. A registered key is associated with each piece of information and is used, like a database record, for storage and retrieval. Several keys and associated values are predefined by MPI and stored in MPI_COMM_WORLD.

MPI_TAG_UB	maximum message tag value
MPI_HOST	process rank on user's local processor
MPI_IO	process rank that can fully accomplish I/O
MPI_WTIME_IS_GLOBAL	Are clocks synchronized?
MPI_UNIVERSE_SIZE	#processes to fill machine

Table 13: Attribute caching related MPI routines

All cached information is retrieved by calling MPI_Attr_get() and specifying the desired key.

- MPI_Attr_get (MPI_Comm comm, int keyval, void *attr_val, int *flag);

The flag parameter is set to true by MPI_Attr_get() if a value has been stored the specified key, as will be the case for all the predefined keys.

Timing

Performance measurement is assisted by MPI_Wtime(), which returns an elapsed wall clock time from some fixed point in the past.

- double MPI_Wtime (void);

3.11Summery

In this chapter, we gave a detailed introduction of Message Passing Interface standard that include point-to-point communication, collective communication, message datatype, comminicator, process toponology and others. In next chaper we are going to get into details about how to implement MPI libraries in the CPPE environment. This part will give you the idea that how MPI library routines is embeded in CPCC and CPSS.

Chapter 4

4. Design and Implementation

CPPE-MPI is a software simulation environment for parallel computation. It features the Message Passing Interface (MPI) programming standard, supported by extensive visual system configuration and debugging tools.

This chapter is organized into four major sections. We begin with point-to-point Communication that provides the most basic communication between two processes. The second section is Collective Communications that provide collective communication among a group of processes in the same context. Derived datatypes provide the library routines to let user define his/her datatype, this make noncontigous data communication a lot easier. The last section covers Groups and Communicators that provide service to create and manipulate communication context of processes. Also, Process Topologies that provide a set of utility functions to assist in the mapping of process group (a linearly ordered set) to richer topological structures such as multi-dimensional grids, is introduced at last.

4.1 Point to Point Communication

As point-to-point communication is the start point of all others, we will take a little time to get into details to see how a message is transferred around network in CPPE simulation environment.

In CPSS, network communication is emulated in the way of that messages were transferred between channels. When a message is sent out, it is sent out to a channel where waiting for its destination processor to retrieve it from the channel eventually.

The channel value structure in C is as below that include buffer address, message arrival time, message size, message tag and message destination.

```
struct ChannelValue
{
  basicValue value;     // value for basic types; address for buffer otherwise
  float time;           // message arrival time
  int size;             // message size
  int processID;        // message destination process
  int tag;              // user tag for message, can be thread ID
  int next;             // link the channel value node together
};
```

Another important thing is that CPSS have to take care of the synchronic problem between the message sent into channel and the processor to retrieve it. When retrieve process try to read a message that is not currently inside channel, this process is going to be hanged into a channel message waiting queue to wait its message to come. Later when the message arrived into the channel (the delay may caused by network delay or the sender sent it out too late), the message is going to wake up this waiting process to have itself retrieved.

The structure of communication network channel in C language

```
Struct Channel {
  Int head,                       // head of list of values for this channel var
  dataCount;                      // number of values available in channel
  ProcDesPtr waitProcQueue;  // list of waiting processes for read
  Float earliestReadTime;         // earliest time the channel can be read again
  Int channElemSize;              // value size
  Int link;                       // index to next available channel
};
```

In our MPI implementation design, we use two basic routine to transfer message from process to process; one is mpiSend() and the other is mpiRecv(). Our MPI communication library routines will use those two basic functions to achieve their communication goal for both point-to-point communication and collective communication that we will discuss in next section.

In mpiSend(), major functionality includes packing message into a channel message package in order to insert it into destination channel's message queue. After the packed message has been in its destination channel, the process sending this message have to wake up the process that is sleeping inside channel message waiting queue waiting for this message by setting its wake up time and its state to "Delayed".

```
mpiSend( )
{
  ...
  // pack message into a channel message package
  CHANN_VAL_Time(msgAdr) = (float) INFINITY;
  CHANN_VAL_TAG(msgAdr) = tag;
  CHANN_VAL_SIZE(msgAdr) = dataSize;
  CHANN_VAL_PROCESSID(msgAdr) = curProcess->phyProcessor;
  CHANN_VAL_INT(msgAdr) = getBlock(dataSize);
  Memcpy(STACK_ADR(CHANN_VAL_INT(msgAdr)), STACK_ADR(dataAdr),
dataSize*sizeof(basicValue));
  ...
  // insert message into channel queue
  CHANN_VAL_NEXT(node1) = msgAdr;
  ...
  // wake up the process if exist, which is waiting inside the channel waiting queue to retrieve it
  while (t != NULL)
  {
    if ( (t->messageTag == CHANN_VAL_TAG(msgAdr) || t->messageTag == ANY )
                          && t->messageSize == CHANN_VAL_SIZE(msgAdr)
                          && ( t->processID == CHANN_VAL_PROCESSID(msgAdr) ||
CHANN_VAL_PROCESSID(msgAdr) == curProcess->phyProcessor) )
    {
      t->state = Delayed;
      t->wakeTime = CHANN_VAL_Time(msgAdr);
      break;
    }
    t = t->qNext;
  }
}
```

First, mpiSend( ) pack the message by

```
CHANN_VAL_Time(msgAdr) = (float) INFINITY;
CHANN_VAL_TAG(msgAdr) = tag;
CHANN_VAL_SIZE(msgAdr) = dataSize;
CHANN_VAL_PROCESSID(msgAdr) = curProcess->phyProcessor;
CHANN_VAL_INT(msgAdr) = getBlock(dataSize);
```

Than, send this packed message into the channel queue that connected to destination process by

```
CHANN_VAL_NEXT(node1) = msgAdr;
```

At last, if the destination process is inside the channel waiting queue to wait to retrieve this message, wake up this process to retrieve it by

```
t->state = Delayed;
t->wakeTime = CHANN_VAL_Time(msgAdr);
```

mpiRecv() is the reverse operation of mpiSend(), first all messages in the channel queue will be scanned for matching message. If a matching message is found, it is going to be copied to local buffer, and at the same time it is going to be removed from the message queue for that channel. If no matching message is found, the process is going to insert itself into the message wait queue for this channel, waiting the arrival of the matching message to wake it up. Below is mpiRecv() main loop in C.

```
MpiRecv( )
{
 ...
 // search all messages inside channel queue for matching message
 node = channPtr->head;
 while (node >0 &&
            ((CHANN_VAL_TAG(node) != tag && tag != ANY)
             ||
             CHANN_VAL_SIZE(node) != size
             ||
             CHANN_VAL_PROCESSID(node) != source ))
 {
  prevNode = node;
  node = CHANN_VAL_NEXT(node);
 }
 ...
 // if no message found or too earier to retreive, insert process into waiting queue
 if (node == 0 || CHANN_VAL_Time(node) > curProcess->time) //no matching data OR arrival
time > current time
 {
  ...
 // insert itself into waiting queue to wait for the message to arrive
  ProcDesPtr temp = channPtr->waitProcQueue;
  while ( temp->qNext != NULL )
 {
   temp = temp->qNext;
 }
  temp->qNext = curProcess;
 ...
 }
 // if a matched message find inside the channel
 else
```

```
 {
 // copy message to local buffer and release channel message buffer
   memcpy(STACK_ADR(dataAdr), STACK_ADR(CHANN_VAL_INT(node)),
size*sizeof(basicValue));
   retBlock(CHANN_VAL_INT(node), size);
 // remove message form channel
   CHANN_VAL_NEXT(prevNode) = CHANN_VAL_NEXT(node);
   ...
 }
 ...
}
```

We can see, first, mpiSend() searches all messages inside channel queue for matched message by checking condition below.

```
 node >0 && ((CHANN_VAL_TAG(node) != tag && tag != ANY)
 || CHANN_VAL_SIZE(node) != size || CHANN_VAL_PROCESSID(node) !=
source
```

If a message is found at the right time, it will be retrieved by.

```
 memcpy(STACK_ADR(dataAdr), STACK_ADR(CHANN_VAL_INT(node)),
size*sizeof(basicValue));
```

Otherwise, this process will be inserted into the process wait queue of this channel to wait the right message to arrive to wake it up as.

```
 temp->qNext = curProcess
```

We implement all the MPI library routines as CPSS built-in function. When a MPI routine is called from application program, CEM (Code Execution Module) will invoke the corresponding routine to manipulate the parameters on the stack top.

The mostly used MPI point-to-point communication routines are MPI send (MPI_Send) and MPI receive (MPI_Recv). We want take a little time to talk about it.

- int MPI_Send (void* buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)

The first three parameters of the send call specify the data to be sent: the outgoing data is to be taken from buf; it consists of count entries; each of type MPI_Datatype, The fourth parameter, specifies the message destination. The fifth parameter specifies the message

tag. Finally, the last parameter is a communicator that specifies a communication domain for this communication. Among other things, a communicator serves to define a set of processes that can be contacted. Each such process is labeled by a process rank. Process ranks are integers and are discovered by inquiry to a communicator.

- int MPI Recv (void* buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI Status *status)

The receiving process specified that the incoming data was to be placed in buf and that it had a size of count entries of type MPI_Datatype. The variable status, set by MPI Recv(), gives information on the source and tag of the message and how many elements were actually received.

The above two are the interface of MPI send and receive call. Either MPI_Send() or MPI_Recv() is called, CEM will execute the following code showing in next.

```
case MPI_SEND:
 {
        int comm = STACK_INT((*T)--);
        int tag = STACK_INT((*T)--);
        int dest = STACK_INT((*T)--);
        int datatype = STACK_INT((*T)--);
        int count = STACK_INT((*T)--);
        int buf = STACK_INT(*T);

        mpiSend( buf, count, datatype, dest, tag, comm );
 }
 break;

 case MPI_RECV:
 {
        int status = STACK_INT((*T)--);
        int comm = STACK_INT((*T)--);
        int tag = STACK_INT((*T)--);
        int source = STACK_INT((*T)--);
        int datatype = STACK_INT((*T)--);
        int count = STACK_INT((*T)--);
        int buf = STACK_INT(*T);

        if ( mpiRecv( buf, count, datatype, source, tag, comm, status ) == 0 )
        {
                curProcess->T += 6;  // Restore the 3 parameters on the stack
```

```
                curProcess->PC--;  // next time this channel read will be re-executed
        }
}
break;
```

The MPI_SEND case is simple, just call mpiSend( ) to send out the message. But with MPI_RECV, we have to recover the stack and PC counter in order to repeat this operation next time, if mpiRecv() failed to retrieve a matching message from communication channel due to the delay of network or the delay of the process which send out the message. Recovery of Stack and Processor Counter is down in C as follows.

```
curProcess->T += 6;  // Restore the 3 parameters on the stack
curProcess->PC--;  // next time this channel read will be re-executed
```

Starting from point-to-point communicaton, we are going to discuss the more advance communication mode, collective communication in next secion.

4.2 Collective Communication

Collective communications transmit data among all processes in a group specified by an intracommunicator object. One function, the barrier, serves to synchronize processes without passing data. MPI provides the following collective communication functions.

Global communication functions include Broadcast, Gather, Scatter, and Reduce. Broadcast will transfer message from one member to all members of a group in the same communication context. Gather will collect data from all group members to one member. Scatter will send different data from one member to all other members in the group. A variation on Gather where all members of the group receive the result, that is allgather. Scatter/Gather will transfer data from all members to all members of the group that is called all-to-all. Global reduction will do calculation (such as sum, max, and min.) with the data from all processes. This includes Reductions where the result is returned to all group members and a variation where the result is returned to only one member.

Some of these functions, such as broadcast or gather, have a single origin or a single receive process. Such a process is called the root. Global communication functions

basically come in three patterns: Root sends data to all processes (itself included): broadcast and scatter. Root receives data from all processes (itself included): gather. Each process communicates with each process (itself included): allgather and all-to-all.

The syntax and semantics of the MPI collective functions were designed to be consistent with point-to-point communications. However, to keep the number of functions and their argument lists to a reasonable level of complexity, the MPI committee made collective functions more restrictive than the point-to-point functions in several ways. One restriction is that, in contrast to point-to-point communication, the amount of data sent must exactly match the amount of data specied by the receiver.

A major simplification is that collective functions come in blocking versions only. Though a standing joke at committee meetings concerned the "non-blocking barrier," such functions can be quite useful and may be included in a future version of MPI. Collective functions do not use a tag argument. Thus, within each intragroup communication domain, collective calls are matched strictly according to the order of execution.

A final simplification of collective functions concerns modes. Collective functions come in only one mode, and this mode may be regarded as analogous to the standard mode of point-to-point. Specifically, the semantics are as follows. A collective function (on a given process) can return as soon as its participation in the overall communication is complete. As usual, the completion indicates that the caller is now free to access and modify locations in the communication buffer(s). It does not indicate that other processes have completed, or even started, the operation. Thus, a collective communication may, or may not have the effect of synchronizing all calling processes. The barrier, of course, is an exception to this statement. This choice of semantics was made so as to achieve a variety of implementations. The consequence of the desire of MPI to: allow efficient implementations on a variety of architectures; and, be clear about exactly what is, and what is not, guaranteed by the standard.

Broadcast

- int MPI_Bcast(void* buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm)

MPI BCAST broadcasts a message from the process with rank root to all processes of the group. The argument root must have identical values on all processes, and comm must represent the same intragroup communication domain. On return, the content of root's communication buffer has been copied to all processes. General, derived datatypes are allowed for datatype. The type signature of count and datatype on any process must be equal to the type signature of count and datatype at the root. This implies that the amount of data sent must be equal to the amount received, pairwise between each process and the root. MPI BCAST and all other data-movement collective routines make this restriction. Distinct type maps between sender and receiver is still allowed.

Below is the C code to implement MPI_Bcast routines in C.

```
case MPI_BCAST:
 {
        int comm = stack_int((*T)--);
        int root = stack_int((*T)--);
        int datatype = stack_int((*T)--);
        int count = stack_int((*T)--);
        int buf = stack_int((*T));

       // if I am root, sent out message to all the process in this communicatior but itself
        if ( curProcess->phyProcessor == TT[comm].processor[root] )
        {
                for (int i = 0; i < TT[comm].size; i++)
                {
                        if ( i != root )
                        {
                                mpiSend(buf, count, datatype, i, MPI_BCAST, comm);
                        }
                }
        }

       // if I am not the source process, trying receive message from source process
        else
        {
               // if no matched message in channel yet, recover PC count an stack.
                if ( !mpiRecv(buf, count, datatype, root, MPI_BCAST, comm, 1) )
                {
                        curProcess->T += 4;
                        curProcess->PC--;
                }
        }
```

```
}
break;
```

All process in the communicator context, either send out message or receive it from sender depend on whether it is the root process.

The root process will send out message to all other processes inside the communicator domain by

```
mpiSend(buf, count, datatype, i, MPI_BCAST, comm);
```

Others will retrieve the message sent by root from their channel by

```
mpiRecv(buf, count, datatype, root, MPI_BCAST, comm, 1)
```

No matter what the reason is, if a process fails to sucessfully retrieve a message from the channel, it need to try againt next time slot. So we have to recover the computation stack and process count to perform the same action. This is down in C by follows.

```
curProcess->T += 4;
curProcess->PC--;
```

Gather

- int MPI_Gather(void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)

Each process (root process included) sends the contents of its send buffer to the root process. The root process receives the messages and stores them in rank order. The outcome is as if each of the n processes in the group (including the root process) had executed a call to MPI_Send(sendbuf, sendcount, sendtype, root, ...), and the root had executed n calls to MPI Recv(recvbuf+i*recvcount*recvtype, recvcount, recvtype, i ,...).

An alternative description is that the n messages sent by the processes in the group are concatenated in rank order, and the resulting message is received by the root as

if by a call to MPI_RECV(recvbuf, recvcount*n, recvtype, ...). The receive buffer is ignored by all non-root processes.

General, derived datatypes are allowed for both sendtype and recvtype. The type signature of sendcount and sendtype on process i must be equal to the type signature of recvcount and recvtype at the root. This implies that the amount of data sent must be equal to the amount of data received, pairwise between each process and the root. Distinct type maps between sender and receiver is still allowed.

All arguments to the function are significant on process root, while on other processes, only argument sendbuf, sendcount, sendtype, root, and comm are significant. The argument root must have identical values on all processes and comm must represent the same intragroup communication domain.

Below is the C code to implement MPI_Gather routines in C.

```
case MPI_GATHER:
 {
        int comm = stack_int((*T)--);
        int root = stack_int((*T)--);
        int recvtype = stack_int((*T)--);
        int recvcnt = stack_int((*T)--);
        int recvbuf = stack_int((*T)--);
        int sendtype = stack_int((*T)--);
        int sendcnt = stack_int((*T)--);
        int sendbuf = stack_int((*T));

        if ( curProcess->phyProcessor == TT[comm].processor[root] )
        {
                if ( zero == root )             // @@ copy root buffer locally
                {
                        memcpy(STACK_ADR(recvbuf+zero*recvcnt*recvtype),
STACK_ADR(sendbuf), sendcnt*sendtype*sizeof(basicValue));
                        zero++;
                }
                else if ( mpiRecv((recvbuf+zero*recvcnt*recvtype), recvcnt, recvtype, zero,
MPI_GATHER, comm, 1) ) // @@ receive from others
                {
                        zero++;    // @@ if receive succeed
                }

                if ( zero != TT[comm].size )          // @@  recover to redo
                {
                        curProcess->T += 7;
                        curProcess->PC--;
```

```
                }
                else
                {
                        mpiBarrier(comm);       // @@ received all buffers
                        zero = 0;
                }
        }
        else
        {
                mpiSend(sendbuf, sendcnt, sendtype, root, MPI_GATHER, comm);
                mpiBarrier(comm);
        }
}
break;
```

All process in the communicator context, either send out message or receive it from sender depend on whether it is the root process.

The root process will retrieve messages from all other processes inside the communicator domain at its channel by

```
mpiRecv((recvbuf+zero*recvcnt*recvtype), recvcnt, recvtype, zero, MPI_GATHER, comm, 1)
```

Others will send out the message to root by

```
mpiSend(sendbuf, sendcnt, sendtype, root, MPI_GATHER, comm);
```

No matter what is the reason, if a process fail to sucessfully retrieve a message from the channel, it need to try againt next time slot. So we have to recover the computation stack and process count to perform the same action. This is down in C by follows.

```
curProcess->T += 7;
curProcess->PC--;
```

Scatter

- int MPI_Scatter(void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)

The outcome is as if the root executed n send operations, MPI_Send(sendbuf+i* sendcount*sendtype, sendcount, sendtype, i,...), i = 0 to n − 1, and each process executed a receive, MPI_Recv(recvbuf, recvcount, recvtype, root,...).

An alternative description is that the root sends a message with MPI_Send(sendbuf, sendcount*n, sendtype, ...). This message is split into n equal segments; the ith segment is sent to theith process in the group, and each process receives this message as above.

The type signature associated with sendcount and sendtype at the root must be equal to the type signature associated with recvcount and recvtype at all processes. This implies that the amount of data sent must be equal to the amount of data received, pairwise between each process and the root. Distinct type maps between sender and receiver is still allowed.

All arguments to the function are significant on process root, while on other processes, only arguments recvbuf, recvcount, recvtype, root, comm are significant. The argument root must have identical values on all processes and comm must represent the same intragroup communication domain. All non-root processes ignore the send buffer. The specification of counts and types should not cause any location on the root to be read more than once.

Below is the C code to implement MPI_Scatter routines in C.

```
case MPI_SCATTER:
 {
        int comm = stack_int((*T)--);
        int root = stack_int((*T)--);
        int recvtype = stack_int((*T)--);
        int recvcnt = stack_int((*T)--);
        int recvbuf = stack_int((*T)--);
        int sendtype = stack_int((*T)--);
        int sendcnt = stack_int((*T)--);
        int sendbuf = stack_int((*T));

       // If current process is root, send out message
        if ( curProcess->phyProcessor == TT[comm].processor[root] )
        {
                for (int i = 1-1; i < TT[comm].size + 1-1; i++)
                {
                        if ( i == root )
                        {
```

```
                                memcpy(STACK_ADR(recvbuf),
                                    STACK_ADR(sendbuf+(i-1+1)*sendcnt*sendtype),
                                    Sendcnt*sendtype*sizeof(basicValue));
                        }
                        else
                        {
                                mpiSend(sendbuf+(i-1+1 )*sendcnt*sendtype, sendcnt,
                                    sendtype, i, MPI_SCATTER, comm);
                        }
                }
        }

        // If current processor is not root, receive message from root processor
        else
        {
                //If message is not in channel yet, recover PC counter and stack for next time.
                If ( !mpiRecv(recvbuf, recvcnt, recvtype, root, MPI_SCATTER, comm, 1) )
                {
                        curProcess->T += 7;
                        curProcess->PC--;
                }
        }
}
break;
```

All process in the communicator context, either send out message or receive it from sender depend on whether it is the root process.

The root process will send out message to all other processes inside the communicator domain by

```
mpiSend(sendbuf+(i-1+1)*sendcnt*sendtype, sendcnt, sendtype, i, MPI_SCATTER, comm);
```

Others will retrieve the message sent by root from their channel by

```
mpiRecv(recvbuf, recvcnt, recvtype, root, MPI_SCATTER, comm, 1)
```

No matter what the reason is, if a process fails to sucessfully retrieve a message from the channel, it need to try againt next time slot. So we have to recover the computation stack and process count to perform the same action. This is down in C by follows.

```
curProcess->T += 7;
curProcess->PC--;
```

Reduce

- int MPI_Reduce(void* sendbuf, void* recvbuf, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)

Global Reduction functions perform a global reduce operation (such as sum, max, logical AND, etc.) across all the members of a group. The reduction operation can be either one of a predefined list of operations, or a user-defined operation.

The global reduction functions come in several flavors: a reduce that returns the result of the reduction at one node, an all-reduce that returns this result at all nodes, and a scan (parallel prefix) operation. In addition, a reduce-scatter operation combines the functionality of a reduce and a scatter operation. In order to improve performance, the functions can be passed an array of values; one call will perform a sequence of element-wise reductions on the arrays of values.

MPI_Reduce() combines the elements provided in the input buffer of each process in the group, using the operation op, and returns the combined value in the output buffer of the process with rank root. The input buffer is defined by the following arguments: sendbuf, count and datatype; the output buffer is defined by the arguments of recvbuf, count and datatype; both have the same number of elements, with the same type. The arguments: count, op and root must have identical values at all processes, the datatype arguments should match, and comm should represent the same intragroup communication domain. Thus, all processes provide input buffers and output buffers of the same length, with elements of the same type. Each process can provide one element, or a sequence of elements, in which case the combine operation is executed element-wise on each entry of the sequence. For example, if the operation is MPI MAX and the send buffer contains two elements which are floating point numbers (count = 2 and datatype = MPI FLOAT), then recvbuf(0) = global max(sendbuf(0)) and recvbuf(1) = global max(sendbuf(1)).

Below is the C code to implement MPI_Reduce routines in C.

```
Case MPI_REDUCE:
 {
        int tempbuf;

        int comm = stack_int((*T)--);
        int root = stack_int((*T)--);
        int op = stack_int((*T)--);
        int datatype = stack_int((*T)--);
        int count = stack_int((*T)--);
        int recvbuf = stack_int((*T)--);
        int sendbuf = stack_int((*T));

        if (zero == 0)
        {
                tempbuf = getBlock(count*datatype);
        }

        if ( curProcess->phyProcessor == TT[comm].processor[root] )
        {
                if ( zero == root )             // @@ copy root buffer locally
                {
                        memcpy(STACK_ADR(tempbuf), STACK_ADR(sendbuf),
count*datatype*sizeof(basicValue));
                        mpiReduce(recvbuf, tempbuf, count, datatype, op);
                        zero++;
                }
                else if (mpiRecv(tempbuf, count, datatype, zero, MPI_REDUCE, comm, 1) )
                {
                        mpiReduce(recvbuf, tempbuf, count, datatype,op);
                        zero++;    // @@ if receive succeed
                }

                if ( zero != TT[comm].size )          // @@  recover to redo
                {
                        curProcess->T += 6;
                        curProcess->PC--;
                }
                else
                {
                        mpiBarrier(comm);        // @@ received all buffers
                        retBlock(tempbuf,count*datatype);
                        zero = 0;
                }
        }
        else
        {
                mpiSend(sendbuf, count, datatype, root, MPI_REDUCE, comm);
                mpiBarrier(comm);
        }
 }
 break;
```

All process in the communicator context, either send out message or receive it from sender depend on whether it is the root process.

The root process will retrieve messages from all other processes inside the communicator domain at its channel by

```
mpiRecv(tempbuf, count, datatype, zero, MPI_REDUCE, comm, 1);
```

Whenever root received a message, it will do reduce operation also by

```
mpiReduce(recvbuf, tempbuf, count, datatype,op);
```

Others will send out the message to root by

```
mpiSend(sendbuf, count, datatype, root, MPI_REDUCE, comm);
```

No matter what the reason is, if a process fails to sucessfully retrieve a message from the channel, it need to try againt at its next time slot. So we have to recover the computation stack and process count to perform the same action. This is down in C by follows.

```
curProcess->T += 6;
curProcess->PC--;
```

4.3 Derived Datatype

MPI provides mechanisms to specify more general, mixed, and noncontiguous communication. A general datatype in MPI is an opaque object that specifies two things: Sequence of basic datatypes and sequence of integer (byte) displacements.

The MPI communication mechanisms allow one to send or receive a sequence of identical elements that are contiguous in memory. It is often desirable to send data that is not homogeneous, such as a structure, or that is not contiguous in memory, such as an array section. This allows one to amortize the fixed overhead of sending and receiving a message over the transmittal of many elements, even in these more general circumstances. MPI provides two mechanisms to achieve this.

The user can define derived datatypes that specify more general data layouts. User-defined datatypes can be used in MPI communication functions, in place of the basic, predefined datatypes.

A sending process can explicitly pack noncontiguous data into a contiguous buffer, and next send it; a receiving process can explicitly unpack data received in a contiguous buffer and store in noncontiguous locations.

The construction and use of derived datatypes is allowed. The use of Pack and Unpack functions is also implemented. It is often possible to achieve the same data transfer using either mechanism. All MPI communication functions take a datatype argument. In the simplest case this will be a primitive type, such as an integer or floating-point number. An important and powerful generalization results by allowing user-defined (or "derived") types wherever the primitive types can occur. These are not types as far as the programming language is concerned. They are only types in that MPI is made aware of them through the use of type-constructor functions, and they describe the layout, in memory, of sets of primitive types. Through user-defined types, MPI supports the communication of complex data structures, such as array sections and structures containing combinations of primitive datatypes.

As we have discussed in previous, derived datatypes are built by combining basic datatypes, or previously built derived datatypes. A derived datatype describes a memory layout, which consists of multiple arrays of elements.

There are three kinds of derived datatype related to MPI routines. One is the kind of routines that will create a new datatype from the old datatypes, or in another words, derive a new datatype from old ones. (MPI_Type_struct() is one of this kind routine.) The second one is the kind of routines that query datatype information by being given a datatype name. (MPI_Type_size() is one of them.) The last one is the kind of MPI routines. They are point-to-point or collective communication routines, but they take datatype all the time as parameter and use its information to manipulate communication message. (MPI_Send() is one of them.)

Because the functionality related to the derived datatype routines almost has no overlap with the CPCC and CPSS, we decide to design and implement the datatype as a

C++ class object. The main attribute of this class is an array typeTable[] that is used to store all the information of newly derived datatype.

Newly derived datatype structure in C is as below:

```
Structure datatype
{
        int NEWTYPE,
        int COUNT,
        int LENGTH,
        int DISPLACEMENT,
        int OLDTYPE
}
```

TypeClass provides the following methods for MPI routines to call when they create new datatype or query the information of the old datatype. Those methods are:

```
int count(int type)
int length(int type)
int displacement(int type)
int oldtype(int type)
 int newtype(int caller, int count, int length, int displacement, int oldtype)
```

count() will return the count of datatype type, length() will return the length of datatype type, displacement() will return the displacement of datatype type, and oldtype() will return which old type this new datatype derived from.

NewType (caller, ...) is the one that will create new data type for MPI data type creating routines including MPI_Type_contiguous(), MPI_Type_vector(), MPI_Type_Hvector(), MPI_Indexed(), MPI_Hindexed(), and MPI_Type_struct(). By specifying caller with different value as TYPE_CONTIGUOUS, TYPE_VECTOR, TYPE_HVECTOR, TYPE_INDEXED, TYPE_HINDEXED or TYPE_STRUCT, and different other parameters it will works for different MPI routines.

4.4 Communicator and Topology

In some applications, it is desirable to divide up the processes to allow different groups of processes to perform independent work. For example, we might want an application to utilize 2/3 of its processes to predict the weather based on data already processed, while the other 1/3 of the processes initially process new data. This would allow the application to regularly complete a weather forecast. However, if no new data is available for processing, we might want the same application to use all of its processes to make a weather forecast.

Being able to do this efficiently and easily requires the application to be able to logically divide the processes into independent subsets. It is important that these subsets are logically the same as the initial set of processes. For example, the module to predict the weather might use process 0 as the master process to dole out work. If subsets of processes are not numbered in a consistent manner with the initial set of processes, then there may be no process 0 in one of the two subsets. This would cause the weather prediction model to fail.

Applications also need to have collective operations working on a subset of processes. If collective operations only work on the initial set of processes, it is impossible to create independent subsets that perform collective operations. Even if the application does not need independent subsets, having collective operations working on subsets is desirable. Since the time to complete most collective operations increases with the number of processes, limiting a collective operation to only the processes that need to be involved yields much better scaling behavior. For example, if a matrix computation needs to broadcast information along the diagonal of a matrix, only the processes containing diagonal elements should be involved.

Regarding to MPI topology mechanism, a topology is an extra, optional attribute that one can give toan intra-communicator; topologies cannot be added to inter-communicators. A topology can provide a convenient naming mechanism for the processes of a group (within a communicator), and additionally, may assist the runtime system in mapping the processes onto hardware.

A process group in MPI is a collection of n processes. Each process in the group is assigned a rank between 0 and n-1. In many parallel applications a linear ranking of processes does not adequately reflect the logical communication pattern of the processes (which is usually determined by the underlying problem geometry and the numerical algorithm used). Often the processes are arranged in topological patterns such as two- or three-dimensional grids.

More generally, the logical process arrangement is described by a graph. A clear distinction must be made between the virtual process topology and the topology of the underlying physical hardware. The virtual topology can be exploited by the system in the assignment of processes to physical processors if this helps to improve the communication performance on a given machine. How this mapping is done, however, is outside the scope of MPI. The description of the virtual topology, on the other hand, depends only on the application, and is machine independent.

We create an array TT[] as storage for newly created communicator and topology. This table is accessed through a class called commClass. This class provides two kinds of methods for MPI routine to call. One is related to create a new communicator or topology, the other is query information for already created communicators and topologies.

int CommClass::newComm(commEnum caller, int p1, int p2, int p3, int p4, int p5) is the method to create new communicator or topologies. It will serve different MPI routine's call by given different parameters. The caller could be one of following:

GROUP_UNION,
GROUP_INTERSECTION,
GROUP_DIFFERENCE,
GROUP_INCL,
GROUP_EXCL,
GROUP_RANGE_INCL,
GROUP_RANGE_EXCL,
COMM_DUP,
COMM_GREATE,
COMM_SPLIT,
INTERCOMM_CREATE,
INTERCOMM_MERGE,
CART_CREATE,
GRAPH_CREATE,
CART_SUB

When following MPI routines call newComm().

MPI_Group_union(),
MPI_Group_intersection(),
MPI_Group_difference(),
MPI_Group_incl(),
MPI_Group_excl(),
MPI_Group_range_incl(),
MPI_Group_range_excl(),
MPI_Comm_dup(),
MPI_Comm_create(),
MPI_Comm_split(),
MPI_Intercomm_create(),
MPI_Intercomm_merge(),
MPI_Cart_create(),
MPI_Graph_create(),
MPI_Cart_sub()

The following methods provide service for MPI routines that its functions rely on communicator or topology informatiom, but will not generate new communicator or topology, but query or use the topology information to act.

---

int groupSize(int group)
int groupRank(int group)
int groupTranslateRank(int group_a, int n, int rank_a, int group_b, int rank_b)
int commGroup(int comm)
int groupCompare(int group1, int group2, int result)
int groupFree(int group)
int commSize(int comm)
int commRank(int comm)
int commCompare(int comm1, int comm2)
int commFree(int comm)
int communicatorInter(int comm)
int commRemoteSize(int comm)
int commRemoteGroup(int comm)
int keyvalCreate(int copy_fn, int delete_fn, int extra_state)
int keyvalFree(int keyval)
int attrPut(int comm, int keyval, int attribute_val)
int attrGet(int comm, int keyval, int attribute_val, int flag)
int attrDelete(int comm, int keyval)
int dimsCreate(int nnodes, int ndims, int dims)
int topoTest()
int graphDimsGet(int comm, int nnodes, int nedges)
int graphGet(int comm, int maxindex, int maxedges, int index, int edges)
int cartDimGet(int comm)
int cartGet(int comm, int maxdims, int dims, int periods, int coords)
int cartRank(int comm, int coords)

```
int cartCoords(int comm, int rank, int maxdims, int coords)
int graphNeighborsCount(int comm, int rank, int nneighbours)
int graphNeighbors(int comm, int rank, int maxneighbors, int neighbours)
int cartShift(int comm, int direction, int disp, int rank_source, int rank_dest)
int cartMap(int comm_old, int ndims, int dims, int periods, int newrank)
int graphMap(int comm_old, int nnodes, int index, int edges, int newrank)
```

The following MPI routines will call above methods to achieve its MPI functionality. We choose not to discuss here in detail as most of them are simple information store and query functions.

```
MPI_Group_size()
MPI_Group_rank()
MPI_Group_translate_rank()
MPI_Comm_group()
MPI_Group_compare()
MPI_Group_free()
MPI_Comm_size()
MPI_Comm_rank()
MPI_Comm_compare()
MPI_Comm_fee()
MPI_Communicator_inter()
MPI_Comm_remote_size()
MPI_Comm_remote_group()
MPI_Keyval_create()
MPI_Keyval_free()
MPI_Attr_put()
MPI_Attr_get()
MPI_Attr_delete()
MPI_Dims_create()
MPI_Topo_test()
MPI_Graphdims_get()
MPI_Graph_get()
MPI_Cartdim_get()
MPI_Cart_get()
MPI_Cart_rank()
MPI_Cart_coords()
MPI_Graph_neighborsCount()
MPI_Graph_neighbors()
MPI_Cart_shift()
MPI_Cart_map()
MPI_Graph_map()
```

4.5 Summery

In this chapter, implementation of MPI libraries is introduced including Point-to-Point Communication, Collective Communications, Derived Datatypes, Groups, Communicators, and Process Topologies. In next chapter, we are going to begin with how to porting MPI application programs between CPC-MPI and other MPI implementations. And then, we will give a introduction of how to progrmming MPI application programes over CPC-MPI with examples.

Chapter 5

5. Example Applications of MPI programming with CPPE-MPI

In this chapter, we first talk about porting MPI program into and out of CPPE. Then discuss issues related to parallel programming with MPI. Finally, we present two MPI application programs: Jacobi's method and Bitonic sort.

5.1 Porting MPI between UNIX and CPPE

First, to port MPI Programs from UNIX to CPPE, or CPPE to UNIX in reverse, the only thing you need to do is to replace these two lines:

- int main (int argc, char* argv[]) // for UNIX
- int MPI_main (int argc, char* argv[]) // for CPPE

Second, to run MPI program under CPPE, the only thing you need to do is to follow CPSS User's Manual. It is totally the same as you running a regular CPC program. In the following two sections, two of MPI applications will be discussed.

5.2 Design and Coding of Parallel MPI Programs

There are essentially two approaches to designing parallel programming:

- The data-parallel approach. In this approach, we partition the data among the processes, and each process executes more or less the same set of commands on its data.
- The control-parallel approach. In this approach, we partition the tasks we wish to carry out among the processes, and each process executes commands that are essentially different from some or all other processes.

It should be noted that most parallel programs involve both approaches. However, data-parallel programming is more common and generally much easier to do. Most importantly perhaps, data-parallel programs tend to scale well: loosely, this means that they can be used to solve larger and larger problems with more and more processes.

How do we design a parallel program? Generally, we start by examining serial solutions to the problem. Then we try partitioning data and control in various ways among the processes. In the following sections, we are going to discuss two algorithms, Jacobi and Bitonic methods in MPI programming.

5.3 Jacobi's Method

Jacobi's method is used to solve a system of linear equations (AX = B). It is an iterative method; that is, after making any initial guess, X0, to a solution, the method generates a sequence of approximations Xk, k = 1,2,3, ..., Xf.

The iteration formula is:

$$x(k+i) = 1/a(i)(i)*(b(i) - (a(i)(0)*x(0)+a(i)(1)*x(1)+...+a(i)(i-1)*x(i-1)+a(i)(i+1)*x(i+1)+...+a(i)(n-1)*x(n-1)))$$

In general, this iteration may not converge. However, if the system is strictly diagonally dominant, Jacobi's method will converge.

In order to terminate the iteration, we can compute the size of the difference between successive estimates, and when this becomes less than some pre-defined tolerance, the iteration terminates. Since the iteration may not converge, we should also keep track of the number of iterations, and terminate it if there is no convergence after some maximum number of iterations.

Here is the serial Jacobi's method

```
/* Return 1 if iteration converged, 0 otherwise */
/* MATRIX_T is just a 2-dimensional array       */
int Jacobi(
      MATRIX_T  A   /* in  */,
      float     x[]         /* out */,
      float     b[]         /* in  */,
      int       n           /* in  */,
      float     tol         /* in  */,
```

```
      int     max_iter     /* in */) {
   int   i, j;
   int   iter_num;
   float  x_old[MAX_DIM];

   float Distance(float x[], float y[], int n);

   /* Initialize x */
   for (i = 0; i < n; i++)
      x[i] = b[i];

   iter_num = 0;
   do {
      iter_num++;

      for (i = 0; i < n; i++)
         x_old[i] = x[i];

      for (i = 0; i < n; i++){
         x[i] = b[i];
         for (j = 0; j < i; j++)
            x[i] = x[i] - A[i][j]*x_old[j];
         for (j = i+1; j < n; j++)
            x[i] = x[i] - A[i][j]*x_old[j];
         x[i] = x[i]/A[i][i];
      }
   } while ((iter_num < max_iter) &&
         (Distance(x,x_old,n) >= tol));

   if (Distance(x,x_old,n) < tol)
      return 1;
   else
      return 0;
} /* Jacobi */
```

5.4 Jacobi's Method with MPI

If there are p processes, and we make the assumption that n>=p, a natural approach to parallelizing Jacobi is to have each process calculate the entries in a sub-vector of the solution vector X.

To do so, we begin by considering how to distribute the data among the processes. In other words, how should A, n, b, tol, max_iter, X, and X_old be distributed? In our case, after process 0 reads these values, it should broadcast them to all the processes. The heart of the algorithm is

Process q: Calculate the entries in x that are assigned to q.

In order to do this we need to calculate process q :

```
for each subscript i assigned to q
{
x[i] = b[i];
for (j=0; j<i; j++)
x[i] = x[i] - A[i][j]*x_old[j];
for (j=i+1; j<n; j++)
x[i] = x[i] - A[i][j]*x_old[j];
x[i] = x[i]/A[i][i];
}
```

In order to carry out the basic calculations, each process needs a complete copy of x_old, as well as its own entries in x and b. This calculation also suggests a partition of A (each process is assigned the rows of A corresponding to its entries in x and b.).

The remaining issue is which entries of x should be assigned to each process. If there are p processes, in order to balance the computational load, we would like to assign approximately p/n entries to each process.

The only parts of the algorithm that we haven't yet looked at are Copy entries of x_local from each process to x_old and the calculation of Distance. Since the distance between two vectors x and y is just sqrt(X.Y). There is no advantage of using one partition scheme over the other in calculation of Distance.

In order to carry out the calculation, copy entries of x_local from each process to x_old.

We want to execute something like the following:

```
Copy x_local to temp;
for (root=0; root<p; root++)
{
        MPI_Bcast(temp, n_bar, MPI_FLOAT, root, MPI_COMM_WORLD);
        Copy temp into appropriate locations in x_old;
}
```

If we have used block partition of x, this has the same overall effect as a single call to the MPI collective communication function, MPI_Allgather;

```
MPI_Allgather (b_local, n_bar, MPI_FLOAT, x_temp1, n_bar,
MPI_FLOAT, MPI_COMM_WORLD);
```

That is, each process' array x_local is sent to every other process, and the received arrays are copied in process rank order into x_old. Now we are ready to write a parallel Jacobi routine.

```
#define Swap(x,y) {float* temp; temp = x; x = y; y = temp;}
/* Return 1 if iteration converged, 0 otherwise */
/* MATRIX_T is a 2-dimensional array          */
int Parallel_jacobi(
      MATRIX_T  A_local    /* in  */,
      float     x_local[]  /* out */,
      float     b_local[]  /* in  */,
      int       n          /* in  */,
      float     tol        /* in  */,
      int       max_iter   /* in  */,
      int       p          /* in  */,
      int       my_rank    /* in  */) {
   int    i_local, i_global, j;
   int    n_bar;
   int    iter_num;
   float  x_temp1[MAX_DIM];
   float  x_temp2[MAX_DIM];
   float* x_old;
   float* x_new;

   float Distance(float x[], float y[], int n);

   n_bar = n/p;

   /* Initialize x */
   MPI_Allgather(b_local, n_bar, MPI_FLOAT, x_temp1,
      n_bar, MPI_FLOAT, MPI_COMM_WORLD);
   x_new = x_temp1;
   x_old = x_temp2;

   iter_num = 0;
   do {
      iter_num++;

      /* Interchange x_old and x_new */
      Swap(x_old, x_new);
      for (i_local = 0; i_local < n_bar; i_local++){
         i_global = i_local + my_rank*n_bar;
         x_local[i_local] = b_local[i_local];
         for (j = 0; j < i_global; j++)
            x_local[i_local] = x_local[i_local] -
               A_local[i_local][j]*x_old[j];
         for (j = i_global+1; j < n; j++)
            x_local[i_local] = x_local[i_local] -
               A_local[i_local][j]*x_old[j];
         x_local[i_local] = x_local[i_local]/
               A_local[i_local][i_global];
```

```
      }

      MPI_Allgather(x_local, n_bar, MPI_FLOAT, x_new,
         n_bar, MPI_FLOAT, MPI_COMM_WORLD);
   } while ((iter_num < max_iter) &&
         (Distance(x_new,x_old,n) >= tol));

   if (Distance(x_new,x_old,n) < tol)
      return 1;
   else
      return 0;
} /* Jacobi */

float Distance(float x[], float y[], int n) {
   int i;
   float sum = 0.0;

   for (i = 0; i < n; i++) {
      sum = sum + (x[i] - y[i])*(x[i] - y[i]);
   }
   return sqrt(sum);
} /* Distance */
```

Note that we use storage for an extra temporary vector - x_new. We do this so that each pass through the main loop involves only one communication - the call to MPI_Allgather.

5.5 Bitonic Sort Method

The bitonic-sorting algorithm is based on the idea of sorting network. Recollect that a monotonic sequence is on that either increase or decreases, but not both. The word "bitonic" was coined to describe sequences that increase and then decrease.

First, it is obvious that any sequence containing two elements is bitonic - in fact, any sequence containing two elements is monotic. We also know that the iterated bitonic split converts a bitonic sequence into a sorted sequence. So we can converts any sequence to a sorted sequence by bitonic split.

If we assume that n is a power of two, it's easy to code a serial bitonic sort.

```
/* Successive subsequences will switch between
 * increasing and decreasing bitonic splits.
```

```
 */
#define INCR 0
#define DECR 1
#define Reverse(ordering) ((ordering) == INCR ? DECR : INCR)
#define Swap(a,b) {KEY_T temp; temp = a; a = b; b = temp;}

int MPI_main() {
   int   list_length;
   int   n;
   int   start_index;
   int   ordering;
   KEY_T  A[MAX];

   printf("Enter the list size (a power of 2)\n");
   scanf("%d", &n);

   Generate_list(n, A);

   Print_list("The unsorted list is", n, A);

   for (list_length = 2; list_length <= n;
       list_length = list_length*2)
      for (start_index = 0, ordering = INCR;
          start_index < n;
          start_index = start_index + list_length,
          ordering = Reverse(ordering))
         if (ordering == INCR)
            Bitonic_sort_incr(list_length,
               A + start_index);
         else
            Bitonic_sort_decr(list_length,
               A + start_index);

   Print_list("The sorted list is", n, A);

} /* main */

void Bitonic_sort_incr(
      int    length  /* in    */,
      KEY_T  B[]     /* in/out */) {
   int i;
   int half_way;

   /* This is the bitonic split */
   half_way = length/2;
   for (i = 0; i < half_way; i++)
      if (B[i] > B[half_way + i])
         Swap(B[i],B[half_way+i]);

   if (length > 2) {
      Bitonic_sort_incr(length/2, B);
      Bitonic_sort_incr(length/2, B + half_way);
   }
} /* Bitonic_sort_incr */
```

The nested for loops in the main program arrange first that successive two element subsequences are monotone (increasing or decreasing), then successive four element subsequences are monotone, etc. For a 16-element sequence, the effect of the algorithm can be visualized as illustrated in Figure 10. An up arrow indicates an increasing subsequence, a down arrow a decreasing subsequence.

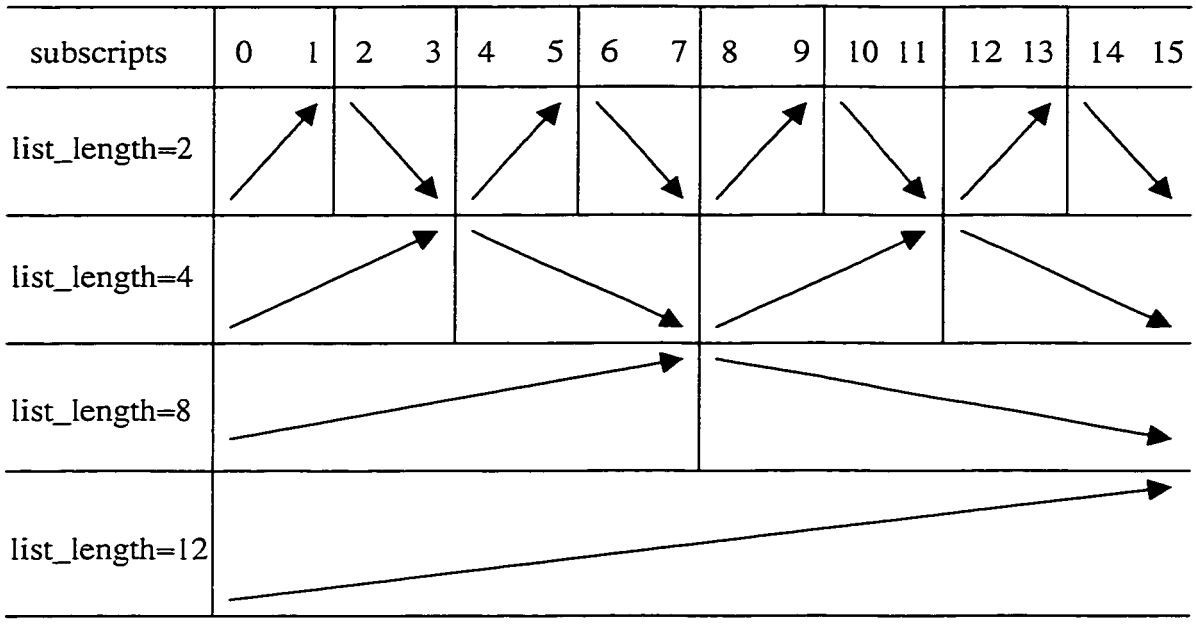


Figure 10: Orientation of Subsequences during stages of bitonic sort

5.6 Bitonic Sort Method with MPI

For the serial bitonic sort, we will assume that the number of key n is a power of two. We'll also assume that the number of process p is a power of two. At each stage of the algorithm, each process will have a sub-list of the global list containing n/p keys.

Let's first consider the case p= 4. We can begin by using a local sorting algorithm to convert the local keys on each process into an increasing or decreasing sequence. The order will be determined by the parity of the process rank: even numbered processes will be sorted into increasing order, odd-numbered processed will be sorted into decreasing order. Then, each pair of processes, (0,1) and (2,3) jointly owns a bitonic sequence, and

the global sequence is bitonic. That is, the 4 tuple of processes, (0,1,2,3), jointly owns a bitonic sequence.

Observe that we can now carry out a bitonic split on the entire distributed sequence by carrying out a bitonic split on the sequence shared by processes 0 and 2 and by carrying out a bitonic split on the sequence shared by processes 1 and 3.

To finish up and obtain a sorted sequence on the four processes, we can perform a bitonic split on the sequence shared by process 0 and 1 and a bitonic split on the sequence shared by processes 2 and 3.

Suppose processes A and B, where A < B, share a bitonic sequence with A's keys increasing and B's keys decreasing. Merge splits is an alternative to obtain a monotonic sequence. Instead of performing a bitonic split, we sort the keys on B in increasing order, merge the keys from the two processes, and assign the smaller keys to A and the larger keys to B. In other words, we can replace our bitonic splits with merge splits.

This is a cheaper operation since it always maintains a sorted list, if there are n/p keys per process, we only have to sort n/p times. If we used bitonic splits, we would sort them every time we sorted across a set of processes.

Let's make all this precise by writing down some of the code for carrying it out. The core of the algorithm, the bitonic split, is replaced by a merge split.

```
#include "mpi.h"
KEY_T temp_list[MAX]; /* buffer for keys received in Merge_split  */
/* Merges the contents of the two lists. */
/* Returns the smaller keys in list1     */
void Merge_list_low(
      int   list_size  /* in     */,
      KEY_T  list1[]    /* in/out */,
      KEY_T  list2[]    /* in     */) {
   int  i;
   int  index1 = 0;
   int  index2 = 0;

   for (i = 0; i < list_size; i++)
      if (list1[index1] <= list2[index2]) {
         scratch_list[i] = list1[index1];
         index1++;
      } else {
         scratch_list[i] = list2[index2];
```

```
        index2++;
      }
   for (i = 0; i < list_size; i++)
      list1[i] = scratch_list[i];

} /* Merge_list_low */

void Merge_split(
      int      list_size    /* in    */,
      KEY_T    local_list[] /* in/out */,
      int      which_keys   /* in    */,
      int      partner      /* in    */,
      MPI_Comm  comm        /* in    */ ) {

   MPI_Status status;

   /* key_mpi_t is an MPI (derived) type */
   MPI_Sendrecv(local_list, list_size, key_mpi_t,
            partner, 0, temp_list, list_size,
            key_mpi_t, partner, 0, comm, &status);
   if (which_keys == HIGH)
      Merge_list_high(list_size, local_list,
                  temp_list);
   else
      Merge_list_low(list_size, local_list,
                  temp_list);
} /* Merge_split */
```

Since we are effectively parallelizing the inner loop of the serial main program, the main program will now have a single for loop:

```
int MPI_main(int argc, char* argv[]) {
   int      list_size;       /* Local list size  */
   int      n;               /* Global list size */
   KEY_T    local_list[MAX];
   int      proc_set_size;
   int      my_rank;
   int      p;
   unsigned  and_bit;
   MPI_Comm  io_comm;

   MPI_Init(&argc, &argv);
   MPI_Comm_size(MPI_COMM_WORLD, &p);
   MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
   MPI_Comm_dup(MPI_COMM_WORLD, &io_comm);
   Cache_io_rank(MPI_COMM_WORLD, io_comm);

   Cscanf(io_comm,"Enter the global list size","%d",&n);
   list_size = n/p;

   Generate_local_list(list_size, local_list);
```

```
   Print_list("Before local sort", list_size, local_list, io_comm);

   Local_sort(list_size, local_list);

   /* and_bit is a bitmask that, when "anded" with  */
   /* my_rank, tells us whether we're working on an */
   /* increasing or decreasing list                 */
   for (proc_set_size = 2, and_bit = 2;
       proc_set_size <= p;
       proc_set_size = proc_set_size*2,
       and_bit = and_bit << 1)
      if ((my_rank & and_bit) == 0)
         Par_bitonic_sort_incr(list_size,
               local_list, proc_set_size, MPI_COMM_WORLD);
      else
         Par_bitonic_sort_decr(list_size,
               local_list, proc_set_size, MPI_COMM_WORLD);

   Print_list("After sort", list_size, local_list, io_comm);

   MPI_Finalize();
} /* main */
```

The processes that have been grouped for a bitonic sort are paired by exclusive or by the process rank with a bitmask consisting of a single bit successively right shifted.

```
void Par_bitonic_sort_incr(
      int      list_size      /* in     */,
      KEY_T*   local_list      /* in/out */,
      int      proc_set_size   /* in     */,
      MPI_Comm  comm           /* in     */ ) {

   unsigned  eor_bit;
   int       proc_set_dim;
   int       stage;
   int       partner;
   int       my_rank;

   MPI_Comm_rank(comm, &my_rank);

   proc_set_dim = log_base2(proc_set_size);
   eor_bit = 1 << (proc_set_dim - 1);
   for (stage = 0; stage < proc_set_dim; stage++) {
      partner = my_rank ^ eor_bit;
      if (my_rank < partner)
         Merge_split(list_size, local_list, LOW,
            partner, comm);
      else
         Merge_split(list_size, local_list, HIGH,
            partner, comm);
```

```
        eor_bit = eor_bit >> 1;
    }
} /* Par_bitonic_sort_incr */
```

5.7 Summery

In previus several chapters, we start with introducing CPPE environment including CPC, CPCC and CPSS, then MPI standard is presented in details. Later, we focus on implementation of MPI libraries in CPPE environment and MPI application program development with CPPE-MPI. Now it's time to make a conclusion summery for this search work as a part of CPPE project in next chapter.

Chapter 6

6. Conclusion and Future Work

MPI is designed for writing message-passing programs that are portable to all existing parallel architectures.

Current MPI implementations are mostly based on UNIX as operating system, and are available for most parallel computers and distributed systems.

There are two major ideas behind building MPI on CPPE.

One is that CPPE-MPI is based on CPPE, thus inherits all CPPE's advantageous features:

- Accuracy: The CPSS employs the functional simulation technique that offers the most accurate results among the existing simulation techniques. In addition, configurable parameters enable the user to accurately simulate a particular multi-computer system by simply setting the values of system parameters to those belonging to the architecture to be simulated.
- Flexibility: The CPSS can simulate a wide range of multi-computer topologies and sizes. It also supports a large set of configurable parameters that permit users to fine-tune their applications and simulate various multi-computer systems. Moreover, the same virtual-architecture program can be mapped to different physical architectures at run time. The flexibility offered by the CPSS is unique among existing simulators.
- Performance: The simulation is fast because the low levels of details are selectively left out to retain essential characteristics of the target processors and network. The entire simulation system, including the application program, is run by a single process, which gives no rise to host context switching at all.
- Repeatability: The CPSS provides repeatability, which is essential for implementing a stable and reliable debugging environment. The CPSS also supports multiple executions of a non-deterministic application. Multiple executions are equally useful for testing the robustness of a deterministic application.

- Correctness and performance debugging tools: The CPSS provides a rich set of correctness and performance-debugging tools to facilitate users' code development. Performance statistics at various levels of details are also available to support algorithmic and architectural performance evaluation and tuning.
- User-friendliness and portability: The parallel programming language used in the CPPE is based on the popular language C. Design concepts of the CPSS user interface and debugging tools are borrowed from sequential programming environments. Currently, the simulator can work on UNIX workstations and Windows or MacOS PCs.
- Expandability: The design and implementation of the simulator are modular and decoupled. Future changes and enhancements to the simulator would be quick and easy.

The second one is that CPPE-MPI provides MPI application development environment for a large community of users that do not have access to multiprocessor computer or multi-computer system. CPPE-MPI fills a gap left by the currently available MPI implementations. Its goals are to promote the acceptance of MPI and improve the developing procedure of MPI applications. Both goals have been achieved by CPPE-MPI implementation. Virtually, CPPE-MPI can run on any uni-processor system with any operating system (DOS, Windows, MacOS or UNIX, etc.).

The CPPE-MPI allows porting parallel programs from CPPE to any other platform MPI system with minimum modification. With a wide range of users enabled to use CPPE-MPI on their PC, the MPI applications are sure to be well developed and promoted. CPPE-MPI, as a development tool, helps decrease implementation cost for MPI applications by relying on less expensive system. As a learning tool, it is accessible to a growing audience of people who feel attracted or challenged by the virtues of parallel computing. CPPE-MPI will play an important role for years to come.

CPPE-MPI is available in a form that permits writing and executing MPI applications under CPPE. The current version of CPPE-MPI supports most library routines. Hopefully, full implementation of MPI including support of C++ language binding will be completed in the future.

Appendix A

A. MPI Programming Examples

```
/* greetings.c -- greetings program
 *
 * Send a message from all processes with rank != 0 to process 0.
 *    Process 0 prints the messages received.
 *
 * Input: none.
 * Output: contents of messages received by process 0.
 *
 */

//#define UNIX

#ifdef UNIX
#include "mpi.h"
#else
#include "mpi8.h"
#endif

#ifdef UNIX
#include <stdio.h>
#include <string.h>
#endif


#ifdef UNIX
void main (argc, argv)
#else
int MPI_main (argc, argv)
#endif
int argc;
char* argv[];
{
   int        my_rank;       /* rank of process      */
   int        p;             /* number of processes  */
   int        source;        /* rank of sender       */
   int        dest;          /* rank of receiver     */
   int        tag = 0;       /* tag for messages     */
   char       message[100]; /* storage for message  */
   MPI_Status  status;       /* return status for    */
                             /* receive              */

   /* Start up MPI */
   MPI_Init(&argc, &argv);

   /* Find out process rank  */
   MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

   /* Find out number of processes */
   MPI_Comm_size(MPI_COMM_WORLD, &p);

   if (my_rank != 0) {
      /* Create message */
      dest = 0;
      /* Use strlen+1 so that '\0' gets transmitted */
      MPI_Send(&my_rank, 1, MPI_INT, dest, tag, MPI_COMM_WORLD);
   } else { /* my_rank == 0 */
```

```
      for (source = 1; source < p; source++) {
         MPI_Recv(&my_rank, 1, MPI_INT, source, tag, MPI_COMM_WORLD, &status);
         printf("Greetings from process %d to %d!\n", my_rank, dest);
      }
   }

   /* Shut down MPI */
   MPI_Finalize();
} /* main */
```

```
/* serial.c -- serial trapezoidal rule
 *
 * Calculate definite integral using trapezoidal rule.
 * The function f(x) is hardwired.
 * Input: a, b, n.
 * Output: estimate of integral from a to b of f(x)
 *    using n trapezoids.
 *
 */

//#define UNIX

#ifdef UNIX
#include <stdio.h>
#include "mpi.h"
#else
#include "mpi1.h"
#endif

float f();

#ifdef UNIX
void main (argc, argv)
#else
int MPI_main (argc, argv)
#endif
int argc;
char* argv[];
{
   float  integral;   /* Store result in integral   */
   float  a, b;       /* Left and right endpoints   */
   int    n;          /* Number of trapezoids       */
   float  h;          /* Trapezoid base width       */
   float  x;
   int    i;

a = 0.0; b = 0.0; n = 0;

   printf("Enter a, b, and n\n\n\n");
/*
   scanf("%f %f %d", &a, &b, &n);
*/
a = 0.0; b = 1.0; n = 100;
   h = (b-a)/n;

   integral = (f(a) + f(b))/2.0;
   x = a;
   for (i = 1; i <= n-1; i++) {
      x = x + h;
      integral = integral + f(x);
   }
   integral = integral*h;

   printf("With n = %d trapezoids, our estimate\n",
      n);
   printf("of the integral from %f to %f = %f\n",
      a, b, integral);
} /* main */
```

```
float f(x)
float x;
{
   float return_val;
   /* Calculate f(x).  Store calculation in return_val. */

   return_val = x*x;
   return return_val;
} /* f */
```

```
/* trap.c -- Parallel Trapezoidal Rule, first version
 *
 * Input: None.
 * Output:  Estimate of the integral from a to b of f(x)
 *    using the trapezoidal rule and n trapezoids.
 *
 * Algorithm:
 *    1.  Each process calculates "its" interval of
 *        integration.
 *    2.  Each process estimates the integral of f(x)
 *        over its interval using the trapezoidal rule.
 *   3a. Each process != 0 sends its integral to 0.
 *   3b. Process 0 sums the calculations received from
 *        the individual processes and prints the result.
 *
 * Notes:
 *    1.  f(x), a, b, and n are all hardwired.
 *    2.  The number of processes (p) should evenly divide
 *        the number of trapezoids (n = 1024)
 *
 */

//#define UNIX

#ifdef UNIX
#include <stdio.h>
#include "mpi.h"
#else
#include "mpi8.h"
#endif

/* We'll be using MPI routines, definitions, etc. */
float f();
/* function we're integrating */
float Trap();
/* Calculate local integral  */

#ifdef UNIX
void main (argc, argv)
#else
int MPI_main (argc, argv)
#endif
int argc;
char* argv[];
{
   int        my_rank;   /* My process rank           */
   int        p;         /* The number of processes   */
   float      a = 0.0;   /* Left endpoint             */
   float      b = 1.0;   /* Right endpoint            */
   int        n = 1024;  /* Number of trapezoids      */
   float      h;         /* Trapezoid base length     */
   float      local_a;   /* Left endpoint my process  */
   float      local_b;   /* Right endpoint my process */
   int        local_n;   /* Number of trapezoids for  */
                         /* my calculation            */
   float      integral;  /* Integral over my interval */
   float      total;     /* Total integral            */
   int        source;    /* Process sending integral  */
```

```
    int        dest = 0; /* All messages go to 0      */
    int        tag = 0;
    MPI_Status  status;

    /* Let the system do what it needs to start up MPI */
    MPI_Init(&argc, &argv);

    /* Get my process rank */
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

    /* Find out how many processes are being used */
    MPI_Comm_size(MPI_COMM_WORLD, &p);

    h = (b-a)/n;    /* h is the same for all processes */
    local_n = n/p;  /* So is the number of trapezoids */

    /* Length of each process' interval of
     * integration = local_n*h.  So my interval
     * starts at: */
    local_a = a + my_rank*local_n*h;
    local_b = local_a + local_n*h;
    integral = Trap(local_a, local_b, local_n, h);

    /* Add up the integrals calculated by each process */
    if (my_rank == 0) {
       total = integral;
       for (source = 1; source < p; source++) {
          MPI_Recv(&integral, 1, MPI_FLOAT, source, tag,
             MPI_COMM_WORLD, &status);
          total = total + integral;
       }
    } else {
       MPI_Send(&integral, 1, MPI_FLOAT, dest,
          tag, MPI_COMM_WORLD);
    }

    /* Print the result */
    if (my_rank == 0) {
       printf("With n = %d trapezoids, our estimate\n",
          n);
       printf("of the integral from %f to %f = %f\n",
          a, b, total);
    }

    /* Shut down MPI */
    MPI_Finalize();
} /*  main  */


float Trap(local_a, local_b, local_n, h)
float  local_a;  /* in */
float  local_b;  /* in */
int    local_n;  /* in */
float  h;        /* in */
{

    float integral;  /* Store result in integral  */
    float x;
    int i;

    integral = (f(local_a) + f(local_b))/2.0;
    x = local_a;
    for (i = 1; i <= local_n-1; i++) {
       x = x + h;
       integral = integral + f(x);
    }
    integral = integral*h;
    return integral;
} /*  Trap  */
```

```
float f(x)
float x;
{
   float return_val;
   /* Calculate f(x). */
   /* Store calculation in return_val. */
   return_val = x*x;
   return return_val;
} /* f */


/* reduce.c -- Parallel Trapezoidal Rule.  Uses 3 calls to MPI_Bcast to
 *    distribute input.  Also uses MPI_Reduce to compute final sum.
 *
 * Input:
 *    a, b: limits of integration.
 *    n: number of trapezoids.
 * Output:  Estimate of the integral from a to b of f(x)
 *    using the trapezoidal rule and n trapezoids.
 *
 * Notes:
 *    1.  f(x) is hardwired.
 *    2.  the number of processes (p) should evenly divide
 *        the number of trapezoids (n).
 *
 */

//#define UNIX

#ifdef UNIX
#include <stdio.h>
#include "mpi.h"
#else
#include "mpi2.h"
#endif

float f();
void Get_data2();
float Trap();

#ifdef UNIX
void main (argc, argv)
#else
int MPI_main (argc, argv)
#endif
int argc;
char* argv[];
{
   int        my_rank;   /* My process rank         */
   int        p;         /* The number of processes  */
   float      a;         /* Left endpoint            */
   float      b;         /* Right endpoint           */
   int        n;         /* Number of trapezoids     */
   float      h;         /* Trapezoid base length    */
   float      local_a;   /* Left endpoint my process */
   float      local_b;   /* Right endpoint my process */
   int        local_n;   /* Number of trapezoids for  */
                         /* my calculation           */
   float      integral;  /* Integral over my interval */
   float      total;     /* Total integral           */
   int        source;    /* Process sending integral  */
   int        dest = 0;  /* All messages go to 0     */
   int        tag = 0;
   MPI_Status  status;

   /* Let the system do what it needs to start up MPI */
   MPI_Init(&argc, &argv);

   /* Get my process rank */
```

```
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

    /* Find out how many processes are being used */
    MPI_Comm_size(MPI_COMM_WORLD, &p);

    Get_data2(&a, &b, &n, my_rank);

    h = (b-a)/n;    /* h is the same for all processes */
    local_n = n/p;  /* So is the number of trapezoids */

    /* Length of each process' interval of
     * integration = local_n*h.  So my interval
     * starts at: */
    local_a = a + my_rank*local_n*h;
    local_b = local_a + local_n*h;
    integral = Trap(local_a, local_b, local_n, h);

    /* Add up the integrals calculated by each process */
    MPI_Reduce(&integral, &total, 1, MPI_FLOAT,
        MPI_SUM, 0, MPI_COMM_WORLD);

    /* Print the result */
    if (my_rank == 0) {
        printf("With n = %d trapezoids, our estimate\n",
            n);
        printf("of the integral from %f to %f = %f\n",
            a, b, total);
    }

    /* Shut down MPI */
    MPI_Finalize();
} /*  main  */


/*********************************************************************/
/* Function Get_data2
 * Reads in the user input a, b, and n.
 * Input parameters:
 *     1.  int my_rank:  rank of current process.
 *     2.  int p:  number of processes.
 * Output parameters:
 *     1.  float* a_ptr:  pointer to left endpoint a.
 *     2.  float* b_ptr:  pointer to right endpoint b.
 *     3.  int* n_ptr:  pointer to number of trapezoids.
 * Algorithm:
 *     1.  Process 0 prompts user for input and
 *         reads in the values.
 *     2.  Process 0 sends input values to other
 *         processes using three calls to MPI_Bcast.
 */

void Get_data2(a_ptr, b_ptr, n_ptr, my_rank)
float*  a_ptr;   /* out */
float*  b_ptr;   /* out */
int*    n_ptr;   /* out */
int     my_rank; /* in  */
{

    if (my_rank == 0) {
        printf("Enter a, b, and n\n\n");

*a_ptr = 0; *b_ptr = 0;             *n_ptr = 0;
/*
        scanf("%f %f %d", a_ptr, b_ptr, n_ptr);
*/
*a_ptr = 0.0; *b_ptr = 1.0;         *n_ptr = 1024;
    }
    MPI_Bcast(a_ptr, 1, MPI_FLOAT, 0, MPI_COMM_WORLD);
    MPI_Bcast(b_ptr, 1, MPI_FLOAT, 0, MPI_COMM_WORLD);
    MPI_Bcast(n_ptr, 1, MPI_INT, 0, MPI_COMM_WORLD);
```

```
} /* Get_data2 */


/********************************************************************/
float Trap(local_a, local_b, local_n, h)
float  local_a;  /* in */
float  local_b;  /* in */
int    local_n;  /* in */
float  h;        /* in */
{

   float integral;  /* Store result in integral  */
   float x;
   int i;

   integral = (f(local_a) + f(local_b))/2.0;
   x = local_a;
   for (i = 1; i <= local_n-1; i++) {
      x = x + h;
      integral = integral + f(x);
   }
   integral = integral*h;
   return integral;
} /*  Trap  */


/********************************************************************/
float f(x)
float x;
{
   float return_val;
   /* Calculate f(x). */
   /* Store calculation in return_val. */
   return_val = x*x;
   return return_val;
} /* f */
```

```
/* get_data.c -- Parallel Trapezoidal Rule, uses basic Get_data function for
 *    input.
 *
 * Input:
 *    a, b: limits of integration.
 *    n: number of trapezoids.
 * Output:  Estimate of the integral from a to b of f(x)
 *    using the trapezoidal rule and n trapezoids.
 *
 * Notes:
 *    1.  f(x) is hardwired.
 *    2.  Assumes number of processes (p) evenly divides
 *        number of trapezoids (n).
 *
 */

//#define UNIX

#ifdef UNIX
#include <stdio.h>
#include "mpi.h"
#else
#include "mpi8.h"
#endif

float f();
/* function we're integrating */
void Get_data();
float Trap();   /* Calculate local integral  */
```

```
#ifdef UNIX
void main (argc, argv)
#else
int MPI_main (argc, argv)
#endif
int argc;
char* argv[];
{
   int        my_rank;  /* My process rank          */
   int        p;        /* The number of processes  */
   float      a;        /* Left endpoint            */
   float      b;        /* Right endpoint           */
   int        n;        /* Number of trapezoids     */
   float      h;        /* Trapezoid base length    */
   float      local_a;  /* Left endpoint my process */
   float      local_b;  /* Right endpoint my process*/
   int        local_n;  /* Number of trapezoids for */
                        /* my calculation           */
   float      integral; /* Integral over my interval*/
   float      total;    /* Total integral           */
   int        source;   /* Process sending integral */
   int        dest = 0; /* All messages go to 0     */
   int        tag = 0;
   MPI_Status  status;

   /* Let the system do what it needs to start up MPI */
   MPI_Init(&argc, &argv);

   /* Get my process rank */
   MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

   /* Find out how many processes are being used */
   MPI_Comm_size(MPI_COMM_WORLD, &p);

   Get_data(&a, &b, &n, my_rank, p);

   h = (b-a)/n;    /* h is the same for all processes */
   local_n = n/p;  /* So is the number of trapezoids */

   /* Length of each process' interval of
    * integration = local_n*h.  So my interval
    * starts at: */
   local_a = a + my_rank*local_n*h;
   local_b = local_a + local_n*h;
   integral = Trap(local_a, local_b, local_n, h);

   /* Add up the integrals calculated by each process */
   if (my_rank == 0) {
      total = integral;
      for (source = 1; source < p; source++) {
         MPI_Recv(&integral, 1, MPI_FLOAT, source, tag,
            MPI_COMM_WORLD, &status);
         total = total + integral;
      }
   } else {
      MPI_Send(&integral, 1, MPI_FLOAT, dest,
         tag, MPI_COMM_WORLD);
   }

   /* Print the result */
   if (my_rank == 0) {
      printf("With n = %d trapezoids, our estimate\n",
         n);
      printf("of the integral from %f to %f = %f\n",
         a, b, total);
   }

   /* Shut down MPI */
   MPI_Finalize();
} /*  main  */
```

```
/*********************************************************************/
/* Function Get_data
 * Reads in the user input a, b, and n.
 * Input parameters:
 *    1. int my_rank:  rank of current process.
 *    2. int p:  number of processes.
 * Output parameters:
 *    1. float* a_ptr:  pointer to left endpoint a.
 *    2. float* b_ptr:  pointer to right endpoint b.
 *    3. int* n_ptr:  pointer to number of trapezoids.
 * Algorithm:
 *    1. Process 0 prompts user for input and
 *       reads in the values.
 *    2. Process 0 sends input values to other
 *       processes.
 */

void Get_data(a_ptr, b_ptr, n_ptr, my_rank, p)
float*  a_ptr;   /* out */
float*  b_ptr;   /* out */
int*    n_ptr;   /* out */
int     my_rank; /* in  */
int     p;       /* in  */
{

   int source = 0;    /* All local variables used by */
   int dest;          /* MPI_Send and MPI_Recv       */
   int tag;
   MPI_Status status;

   if (my_rank == 0){

            *a_ptr = 0.0; *b_ptr = 0.0; *n_ptr = 0;

      printf("Enter a, b, and n\n\n");
/*
      scanf("%f %f %d", a_ptr, b_ptr, n_ptr);
*/
            *a_ptr = 0.0; *b_ptr = 1.0; *n_ptr = 1024;

      for (dest = 1; dest < p; dest++){
         tag = 0;
         MPI_Send(a_ptr, 1, MPI_FLOAT, dest, tag,
            MPI_COMM_WORLD);
         tag = 1;
         MPI_Send(b_ptr, 1, MPI_FLOAT, dest, tag,
            MPI_COMM_WORLD);
         tag = 2;
         MPI_Send(n_ptr, 1, MPI_INT, dest, tag,
            MPI_COMM_WORLD);
      }
   } else {
      tag = 0;
      MPI_Recv(a_ptr, 1, MPI_FLOAT, source, tag,
        MPI_COMM_WORLD, &status);
      tag = 1;
      MPI_Recv(b_ptr, 1, MPI_FLOAT, source, tag,
         MPI_COMM_WORLD, &status);
      tag = 2;
      MPI_Recv(n_ptr, 1, MPI_INT, source, tag,
                        MPI_COMM_WORLD, &status);
   }
} /* Get_data */


/*********************************************************************/
float Trap(local_a, local_b, local_n, h)
float  local_a;  /* in */
```

```
float  local_b;  /* in */
int    local_n;  /* in */
float  h;        /* in */
{

    float integral;  /* Store result in integral  */
    float x;
    int i;

    integral = (f(local_a) + f(local_b))/2.0;
    x = local_a;
    for (i = 1; i <= local_n-1; i++) {
        x = x + h;
        integral = integral + f(x);
    }
    integral = integral*h;
    return integral;
} /*  Trap  */


/********************************************************************/
float f(x)
float x;
{
    float return_val;
    /* Calculate f(x). */
    /* Store calculation in return_val. */
    return_val = x*x;
    return return_val;
} /* f */
```

```
/* get_data1.c -- Parallel Trapezoidal Rule; uses a hand-coded
 *    tree-structured broadcast.
 *
 * Input:
 *    a, b: limits of integration.
 *    n: number of trapezoids.
 * Output:  Estimate of the integral from a to b of f(x)
 *    using the trapezoidal rule and n trapezoids.
 *
 * Notes:
 *    1.  f(x) is hardwired.
 *    2.  the number of processes (p) should evenly divide
 *        the number of trapezoids (n).
 *
 */

//#define UNIX

#ifdef UNIX
#include <stdio.h>
#include "mpi.h"
#else
#include "mpi8.h"
#endif

float f();
int Ceiling_log2();
int l_receive();
int l_send();
void Send();
void Receive();

void Get_data1();
float Trap();

#ifdef UNIX
void main (argc, argv)
#else
```

```
int MPI_main (argc, argv)
#endif
int argc;
char* argv[];
{
    int        my_rank;  /* My process rank          */
    int        p;        /* The number of processes  */
    float      a;        /* Left endpoint            */
    float      b;        /* Right endpoint           */
    int        n;        /* Number of trapezoids     */
    float      h;        /* Trapezoid base length    */
    float      local_a;  /* Left endpoint my process */
    float      local_b;  /* Right endpoint my process */
    int        local_n;  /* Number of trapezoids for */
                         /* my calculation           */
    float      integral; /* Integral over my interval */
    float      total;    /* Total integral           */
    int        source;   /* Process sending integral */
    int        dest = 0; /* All messages go to 0     */
    int        tag = 0;
    MPI_Status  status;

    /* Let the system do what it needs to start up MPI */
    MPI_Init(&argc, &argv);

    /* Get my process rank */
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

    /* Find out how many processes are being used */
    MPI_Comm_size(MPI_COMM_WORLD, &p);

    Get_data1(&a, &b, &n, my_rank, p);

    h = (b-a)/n;    /* h is the same for all processes */
    local_n = n/p;  /* So is the number of trapezoids */

    /* Length of each process' interval of
     * integration = local_n*h.  So my interval
     * starts at: */
    local_a = a + my_rank*local_n*h;
    local_b = local_a + local_n*h;
    integral = Trap(local_a, local_b, local_n, h);

    /* Add up the integrals calculated by each process */
    if (my_rank == 0) {
       total = integral;
       for (source = 1; source < p; source++) {
          MPI_Recv(&integral, 1, MPI_FLOAT, source, tag,
             MPI_COMM_WORLD, &status);
          total = total + integral;
       }
    } else {
       MPI_Send(&integral, 1, MPI_FLOAT, dest,
          tag, MPI_COMM_WORLD);
    }

    /* Print the result */
    if (my_rank == 0) {
       printf("With n = %d trapezoids, our estimate\n",
          n);
       printf("of the integral from %f to %f = %f\n",
          a, b, total);
    }

    /* Shut down MPI */
    MPI_Finalize();
} /* main */


/*********************************************************************/
```

```
/* Ceiling of log_2(x) is just the number of times
 * times x-1 can be divided by 2 until the quotient
 * is 0.  Dividing by 2 is the same as right shift.
 */

int Ceiling_log2(x)
int  x;  /* in */
{
   /* Use unsigned so that right shift will fill
    * leftmost bit with 0
    */
   unsigned temp = (unsigned) x - 1;
   int result = 0;

   while (temp != 0) {
      temp = temp >> 1;
      result = result + 1 ;
   }
   return result;
} /* Ceiling_log2 */


/*****************************************************************/
int I_receive(stage, my_rank, source_ptr)
int   stage;      /* in  */
int   my_rank;    /* in  */
int*  source_ptr; /* out */
{
   int   power_2_stage;

   /* 2^stage = 1 << stage */
   power_2_stage = 1 << stage;
   if ((power_2_stage <= my_rank) &&
         (my_rank < 2*power_2_stage)){
      *source_ptr = my_rank - power_2_stage;
      return 1;
   } else return 0;
} /* I_receive */

/*****************************************************************/
int I_send(stage, my_rank, p, dest_ptr)
int   stage;    /* in  */
int   my_rank;  /* in  */
int   p;        /* in  */
int*  dest_ptr; /* out */
{
   int power_2_stage;

   /* 2^stage = 1 << stage */
   power_2_stage = 1 << stage;
   if (my_rank < power_2_stage){
      *dest_ptr = my_rank + power_2_stage;
      if (*dest_ptr >= p) return 0;
      else return 1;
   } else return 0;
} /* I_send */

/*****************************************************************/
void Send(a, b, n, dest)
float  a;    /* in */
float  b;    /* in */
int    n;    /* in */
int    dest; /* in */
{
   MPI_Send(&a, 1, MPI_FLOAT, dest, 0, MPI_COMM_WORLD);
   MPI_Send(&b, 1, MPI_FLOAT, dest, 1, MPI_COMM_WORLD);
   MPI_Send(&n, 1, MPI_INT, dest, 2, MPI_COMM_WORLD);
} /* Send */
```

```
/*********************************************************************/
void Receive(a_ptr, b_ptr, n_ptr, source)
float*  a_ptr;  /* out */
float*  b_ptr;  /* out */
int*    n_ptr;  /* out */
int     source; /* in  */
{
   MPI_Status status;

   MPI_Recv(a_ptr, 1, MPI_FLOAT, source, 0,
      MPI_COMM_WORLD, &status);
   MPI_Recv(b_ptr, 1, MPI_FLOAT, source, 1,
      MPI_COMM_WORLD, &status);
   MPI_Recv(n_ptr, 1, MPI_INT, source, 2,
      MPI_COMM_WORLD, &status);
} /* Receive */


/*********************************************************************/
/* Function Get_data1
 * Reads in the user input a, b, and n.
 * Input parameters:
 *    1. int my_rank:  rank of current process.
 *    2. int p:  number of processes.
 * Output parameters:
 *    1. float* a_ptr:  pointer to left endpoint a.
 *    2. float* b_ptr:  pointer to right endpoint b.
 *    3. int* n_ptr:  pointer to number of trapezoids.
 * Algorithm:
 *    1. Process 0 prompts user for input and
 *       reads in the values.
 *    2. Process 0 sends input values to other
 *       processes using hand-coded tree-structured
 *       broadcast.
 */
void Get_data1(a_ptr, b_ptr, n_ptr, my_rank, p)
float*  a_ptr;   /* out */
float*  b_ptr;   /* out */
int*    n_ptr;   /* out */
int     my_rank; /* in  */
int     p;       /* in  */
{

   int source;
   int dest;
   int stage;

   if (my_rank == 0){

*a_ptr = 0.0; *b_ptr = 0.0; *n_ptr = 0;
      printf("Enter a, b, and n\n\n");
/*
      scanf("%f %f %d", a_ptr, b_ptr, n_ptr);
*/
*a_ptr = 0.0; *b_ptr = 1.0; *n_ptr = 1024;
   }
   for (stage = 0; stage < Ceiling_log2(p); stage++)
      if (I_receive(stage, my_rank, &source))
         Receive(a_ptr, b_ptr, n_ptr, source);
      else if (I_send(stage, my_rank, p, &dest))
         Send(*a_ptr, *b_ptr, *n_ptr, dest);
} /* Get_data1*/


/*********************************************************************/
float Trap(local_a, local_b, local_n, h)
float  local_a;  /* in */
float  local_b;  /* in */
int    local_n;  /* in */
float  h;        /* in */
```

```
{

   float integral;   /* Store result in integral  */
   float x;
   int i;

   integral = (f(local_a) + f(local_b))/2.0;
   x = local_a;
   for (i = 1; i <= local_n-1; i++) {
      x = x + h;
      integral = integral + f(x);
   }
   integral = integral*h;
   return integral;
} /*  Trap  */


/*********************************************************************/
float f(x)
float x;
{
   float return_val;
   /* Calculate f(x). */
   /* Store calculation in return_val. */
   return_val = x*x;
   return return_val;
} /* f */
```

```
/* get_data2.c -- Parallel Trapezoidal Rule.  Uses 3 calls to MPI_Bcast to
 *    distribute input data.
 *
 * Input:
 *    a, b: limits of integration.
 *    n: number of trapezoids.
 * Output:  Estimate of the integral from a to b of f(x)
 *    using the trapezoidal rule and n trapezoids.
 *
 * Notes:
 *    1.  f(x) is hardwired.
 *    2.  the number of processes (p) should evenly divide
 *        the number of trapezoids (n).
 *
 */

//#define UNIX

#ifdef UNIX
#include <stdio.h>
#include "mpi.h"
#else
#include "mpi8.h"
#endif

float f();
void Get_data2();
float Trap();

#ifdef UNIX
void main (argc, argv)
#else
int MPI_main (argc, argv)
#endif
int argc;
char* argv[];
{
   int        my_rank;   /* My process rank           */
   int        p;         /* The number of processes   */
   float      a;         /* Left endpoint             */
   float      b;         /* Right endpoint            */
```

```
    int        n;        /* Number of trapezoids      */
    float      h;        /* Trapezoid base length     */
    float      local_a;  /* Left endpoint my process  */
    float      local_b;  /* Right endpoint my process */
    int        local_n;  /* Number of trapezoids for  */
                         /* my calculation            */
    float      integral; /* Integral over my interval */
    float      total;    /* Total integral            */
    int        source;   /* Process sending integral  */
    int        dest = 0; /* All messages go to 0      */
    int        tag = 0;
    MPI_Status status;

    /* Let the system do what it needs to start up MPI */
    MPI_Init(&argc, &argv);

    /* Get my process rank */
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

    /* Find out how many processes are being used */
    MPI_Comm_size(MPI_COMM_WORLD, &p);

    Get_data2(&a, &b, &n, my_rank);

    h = (b-a)/n;    /* h is the same for all processes */
    local_n = n/p;  /* So is the number of trapezoids */

    /* Length of each process' interval of
     * integration = local_n*h.  So my interval
     * starts at: */
    local_a = a + my_rank*local_n*h;
    local_b = local_a + local_n*h;
    integral = Trap(local_a, local_b, local_n, h);

    /* Add up the integrals calculated by each process */
    if (my_rank == 0) {
       total = integral;
       for (source = 1; source < p; source++) {
          MPI_Recv(&integral, 1, MPI_FLOAT, source, tag,
             MPI_COMM_WORLD, &status);
          total = total + integral;
       }
    } else {
       MPI_Send(&integral, 1, MPI_FLOAT, dest,
          tag, MPI_COMM_WORLD);
    }

    /* Print the result */
    if (my_rank == 0) {
       printf("With n = %d trapezoids, our estimate\n",
          n);
       printf("of the integral from %f to %f = %f\n",
          a, b, total);
    }

    /* Shut down MPI */
    MPI_Finalize();
} /*  main  */


/*--------------------------------------------------------------------*/
/* Ceiling of log_2(x) is just the number of times
 * times x-1 can be divided by 2 until the quotient
 * is 0.  Dividing by 2 is the same as right shift.
 */
int Ceiling_log2(x)
int  x;  /* in */
{
    /* Use unsigned so that right shift will fill
     * leftmost bit with 0
```

```
    */
    unsigned temp = (unsigned) x - 1;
    int result = 0;

    while (temp != 0) {
        temp = temp >> 1;
        result = result + 1 ;
    }
    return result;
} /* Ceiling_log2 */


/*****************************************************************/
int I_receive(stage, my_rank, source_ptr)
int  stage;      /* in  */
int  my_rank;    /* in  */
int* source_ptr; /* out */
{
    int  power_2_stage;

    /* 2^stage = 1 << stage */
    power_2_stage = 1 << stage;
    if ((power_2_stage <= my_rank) &&
          (my_rank < 2*power_2_stage)){
        *source_ptr = my_rank - power_2_stage;
        return 1;
    } else return 0;
} /* I_receive */

/*****************************************************************/
int I_send(stage, my_rank, p, dest_ptr)
int  stage;    /* in  */
int  my_rank;  /* in  */
int  p;        /* in  */
int* dest_ptr; /* out */
{
    int power_2_stage;

    /* 2^stage = 1 << stage */
    power_2_stage = 1 << stage;
    if (my_rank < power_2_stage){
        *dest_ptr = my_rank + power_2_stage;
        if (*dest_ptr >= p) return 0;
        else return 1;
    } else return 0;
} /* I_send */

/*****************************************************************/
void Send(a, b, n, dest)
float a;    /* in */
float b;    /* in */
int   n;    /* in */
int   dest; /* in */
{

    MPI_Send(&a, 1, MPI_FLOAT, dest, 0, MPI_COMM_WORLD);
    MPI_Send(&b, 1, MPI_FLOAT, dest, 1, MPI_COMM_WORLD);
    MPI_Send(&n, 1, MPI_INT, dest, 2, MPI_COMM_WORLD);
} /* Send */


/*****************************************************************/
void Receive(a_ptr, b_ptr, n_ptr, source)
float* a_ptr; /* out */
float* b_ptr; /* out */
int*   n_ptr; /* out */
int    source; /* in  */
{

    MPI_Status status;
```

```
   MPI_Recv(a_ptr, 1, MPI_FLOAT, source, 0,
      MPI_COMM_WORLD, &status);
   MPI_Recv(b_ptr, 1, MPI_FLOAT, source, 1,
      MPI_COMM_WORLD, &status);
   MPI_Recv(n_ptr, 1, MPI_INT, source, 2,
      MPI_COMM_WORLD, &status);
} /* Receive */


/*********************************************************************/
/* Function Get_data2
 * Reads in the user input a, b, and n.
 * Input parameters:
 *    1. int my_rank:  rank of current process.
 *    2. int p:  number of processes.
 * Output parameters:
 *    1. float* a_ptr:  pointer to left endpoint a.
 *    2. float* b_ptr:  pointer to right endpoint b.
 *    3. int* n_ptr:  pointer to number of trapezoids.
 * Algorithm:
 *    1. Process 0 prompts user for input and
 *       reads in the values.
 *    2. Process 0 sends input values to other
 *       processes using three calls to MPI_Bcast.
 */

void Get_data2(a_ptr, b_ptr, n_ptr, my_rank)
float*  a_ptr;   /* out */
float*  b_ptr;   /* out */
int*    n_ptr;   /* out */
int     my_rank; /* in  */
{

   if (my_rank == 0) {

*a_ptr = 0.0; *b_ptr = 0.0; *n_ptr = 0;

      printf("Enter a, b, and n\n\n");
/*
      scanf("%f %f %d", a_ptr, b_ptr, n_ptr);
*/
*a_ptr = 0.0; *b_ptr = 1.0; *n_ptr = 1024;
   }
   MPI_Bcast(a_ptr, 1, MPI_FLOAT, 0, MPI_COMM_WORLD);
   MPI_Bcast(b_ptr, 1, MPI_FLOAT, 0, MPI_COMM_WORLD);
   MPI_Bcast(n_ptr, 1, MPI_INT, 0, MPI_COMM_WORLD);
} /* Get_data2 */


/*********************************************************************/
float Trap(local_a, local_b, local_n, h)
float  local_a;  /* in */
float  local_b;  /* in */
int    local_n;  /* in */
float  h;        /* in */
{

   float integral;  /* Store result in integral  */
   float x;
   int i;

   integral = (f(local_a) + f(local_b))/2.0;
   x = local_a;
   for (i = 1; i <= local_n-1; i++) {
      x = x + h;
      integral = integral + f(x);
   }
   integral = integral*h;
   return integral;
```

```
} /* Trap */


/*********************************************************************/
float f(x)
float x;
{
   float return_val;
   /* Calculate f(x). */
   /* Store calculation in return_val. */
   return_val = x*x;
   return return_val;
} /* f */
```

```
/* get_data3.c -- Parallel Trapezoidal Rule.  Builds a derived type
 *     for use with the distribution of the input data.
 *
 * Input:
 *    a, b: limits of integration.
 *    n: number of trapezoids.
 * Output:  Estimate of the integral from a to b of f(x)
 *    using the trapezoidal rule and n trapezoids.
 *
 * Notes:
 *    1.  f(x) is hardwired.
 *    2.  the number of processes (p) should evenly divide
 *        the number of trapezoids (n).
 *
 */

//#define UNIX

#ifdef UNIX
#include <stdio.h>
#include "mpi.h"
#else
#include "mpi8.h"
#endif

void Build_derived_type();
float f();
void Get_data3();
float Trap();

#ifdef UNIX
void main (argc, argv)
#else
int MPI_main (argc, argv)
#endif
int argc;
char* argv[];
{
   int        my_rank;   /* My process rank         */
   int        p;         /* The number of processes  */
   float      a;         /* Left endpoint            */
   float      b;         /* Right endpoint           */
   int        n;         /* Number of trapezoids     */
   float      h;         /* Trapezoid base length    */
   float      local_a;   /* Left endpoint my process */
   float      local_b;   /* Right endpoint my process */
   int        local_n;   /* Number of trapezoids for  */
                         /* my calculation           */
   float      integral;  /* Integral over my interval */
   float      total;     /* Total integral           */
   int        source;    /* Process sending integral  */
   int        dest = 0;  /* All messages go to 0     */
   int        tag = 0;
   MPI_Status  status;
```

```
    /* Let the system do what it needs to start up MPI */
    MPI_Init(&argc, &argv);

    /* Get my process rank */
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

    /* Find out how many processes are being used */
    MPI_Comm_size(MPI_COMM_WORLD, &p);

    Get_data3(&a, &b, &n, my_rank);

    h = (b-a)/(float)n;

    local_n = n/p;  /* So is the number of trapezoids */

    /* Length of each process' interval of
     * integration = local_n*h.  So my interval
     * starts at: */
    local_a = a + my_rank*local_n*h;
    local_b = local_a + local_n*h;
    integral = Trap(local_a, local_b, local_n, h);

    /* Add up the integrals calculated by each process */
    MPI_Reduce(&integral, &total, 1, MPI_FLOAT,
        MPI_SUM, 0, MPI_COMM_WORLD);

    /* Print the result */
    if (my_rank == 0) {
        printf("With n = %d trapezoids, our estimate\n",
            n);
        printf("of the integral from %f to %f = %f\n",
            a, b, total);
    }

    /* Shut down MPI */
    MPI_Finalize();
} /*  main  */


/*********************************************************************/
void Build_derived_type(a_ptr, b_ptr, n_ptr, mesg_mpi_t_ptr)
float*        a_ptr;          /* in  */
float*        b_ptr;          /* in  */
int*          n_ptr;          /* in  */
MPI_Datatype* mesg_mpi_t_ptr; /* out */
{
    /* pointer to new MPI type */
    int a, b, c, d;

    /* The number of elements in each "block" of the   */
    /*    new type.  For us, 1 each.                   */
    int block_lengths[3];

    /* Displacement of each element from start of new  */
    /*    type.  The "d_i's."                          */
    /* MPI_Aint ("address int") is an MPI defined C    */
    /*    type.  Usually an int.                       */
    MPI_Aint displacements[3];

    /* MPI types of the elements.  The "t_i's."        */
    MPI_Datatype typelist[3];

    /* Use for calculating displacements               */
    MPI_Aint start_address;
    MPI_Aint address;

    block_lengths[0] = block_lengths[1]
                = block_lengths[2] = 1;

    /* Build a derived datatype consisting of  */
```

```
    /* two floats and an int               */
    typelist[0] = MPI_FLOAT;
    typelist[1] = MPI_FLOAT;
    typelist[2] = MPI_INT;

    /* First element, a, is at displacement 0     */
    displacements[0] = 0;

    /* Calculate other displacements relative to a */
    MPI_Address(a_ptr, &start_address);

    /* Find address of b and displacement from a   */
    MPI_Address(b_ptr, &address);
    displacements[1] = address - start_address;

    /* Find address of n and displacement from a   */
    MPI_Address(n_ptr, &address);
    displacements[2] = address - start_address;

    /* Build the derived datatype */
    MPI_Type_struct(3, block_lengths, displacements,
        typelist, mesg_mpi_t_ptr);

    /* Commit it -- tell system we'll be using it for */
    /* communication.                              */
    MPI_Type_commit(mesg_mpi_t_ptr);
} /* Build_derived_type */


/*********************************************************************/
void Get_data3(a_ptr, b_ptr, n_ptr, my_rank)
float*  a_ptr;   /* out */
float*  b_ptr;   /* out */
int*    n_ptr;   /* out */
int     my_rank; /* in  */
{
    MPI_Datatype  mesg_mpi_t; /* MPI type corresponding */
                              /* to 3 floats and an int */

    if (my_rank == 0){
        printf("Enter a, b, and n\n");
*a_ptr = 0.0; *b_ptr=0.0;
/*
        scanf("%f %f %d", a_ptr, b_ptr, n_ptr);
*/
*a_ptr = 0.0; *b_ptr=1.0; *n_ptr=1024;
    }

    Build_derived_type(a_ptr, b_ptr, n_ptr, &mesg_mpi_t);

    MPI_Bcast(a_ptr, 1, mesg_mpi_t, 0, MPI_COMM_WORLD);
} /* Get_data3 */


/*********************************************************************/
float Trap(local_a, local_b, local_n, h)
float  local_a;   /* in */
float  local_b;   /* in */
int    local_n;   /* in */
float  h;         /* in */
{

    float integral;   /* Store result in integral  */
    float x;
    int i;

    integral = (f(local_a) + f(local_b))/2.0;
    x = local_a;
    for (i = 1; i <= local_n-1; i++) {
        x = x + h;
```

```
      integral = integral + f(x);
   }
   integral = integral*h;
   return integral;
} /* Trap */


/*********************************************************************/
float f(x)
float x;
{
   float return_val;
   /* Calculate f(x). */
   /* Store calculation in return_val. */
   return_val = x*x;
   return return_val;
} /* f */
```

```
/* get_data4.c -- Parallel Trapezoidal Rule.  Uses MPI_Pack/Unpack in
 *    distribution of input data.
 *
 * Input:
 *    a, b: limits of integration.
 *    n: number of trapezoids.
 * Output:  Estimate of the integral from a to b of f(x)
 *    using the trapezoidal rule and n trapezoids.
 *
 * Notes:
 *    1.  f(x) is hardwired.
 *    2.  the number of processes (p) should evenly divide
 *        the number of trapezoids (n).
 *
 */


//#define UNIX

#ifdef UNIX
#include <stdio.h>
#include "mpi.h"
#else
#include "mpi8.h"
#endif

float f();
void Get_data4();
float Trap();

#ifdef UNIX
void main (argc, argv)
#else
int MPI_main (argc, argv)
#endif
int argc;
char* argv[];
{
   int        my_rank;   /* My process rank         */
   int        p;         /* The number of processes  */
   float      a;         /* Left endpoint            */
   float      b;         /* Right endpoint           */
   int        n;         /* Number of trapezoids     */
   float      h;         /* Trapezoid base length    */
   float      local_a;   /* Left endpoint my process */
   float      local_b;   /* Right endpoint my process */
   int        local_n;   /* Number of trapezoids for */
                         /* my calculation           */
   float      integral;  /* Integral over my interval */
   float      total;     /* Total integral           */
   int        source;    /* Process sending integral */
```

```
    int        dest = 0; /* All messages go to 0      */
    int        tag = 0;
    MPI_Status  status;

    /* Let the system do what it needs to start up MPI */
    MPI_Init(&argc, &argv);

    /* Get my process rank */
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

    /* Find out how many processes are being used */
    MPI_Comm_size(MPI_COMM_WORLD, &p);

    Get_data4(&a, &b, &n, my_rank);

    h = (b-a)/n;    /* h is the same for all processes */
    local_n = n/p;  /* So is the number of trapezoids */

    /* Length of each process' interval of
     * integration = local_n*h.  So my interval
     * starts at: */
    local_a = a + my_rank*local_n*h;
    local_b = local_a + local_n*h;
    integral = Trap(local_a, local_b, local_n, h);

    /* Add up the integrals calculated by each process */
    MPI_Reduce(&integral, &total, 1, MPI_FLOAT,
        MPI_SUM, 0, MPI_COMM_WORLD);

    /* Print the result */
    if (my_rank == 0) {
        printf("With n = %d trapezoids, our estimate\n",
            n);
        printf("of the integral from %f to %f = %f\n",
            a, b, total);
    }

    /* Shut down MPI */
    MPI_Finalize();
} /*  main  */


/*********************************************************************/
void Get_data4(a_ptr, b_ptr, n_ptr, my_rank)
float*  a_ptr;   /* out */
float*  b_ptr;   /* out */
int*    n_ptr;   /* out */
int     my_rank; /* in  */
{

    char  buffer[100]; /* Store data in buffer       */
    int   position;    /* Keep track of where data is */
                       /*    in the buffer           */

    if (my_rank == 0){
*a_ptr = 0.0; *b_ptr=0.0;
        printf("Enter a, b, and n\n");
/*
        scanf("%f %f %d", a_ptr, b_ptr, n_ptr);
*/
*a_ptr = 0.0; *b_ptr=1.0; *n_ptr=1024;

        /* Now pack the data into buffer.  Position = 0 */
        /* says start at beginning of buffer.          */
        position = 0;

        /* Position is in/out */
        MPI_Pack(a_ptr, 1, MPI_FLOAT, buffer, 100,
            &position, MPI_COMM_WORLD);
        /* Position has been incremented: it now refer- */
```

```
      /* ences the first free location in buffer.  */

      MPI_Pack(b_ptr, 1, MPI_FLOAT, buffer, 100,
         &position, MPI_COMM_WORLD);
      /* Position has been incremented again. */

      MPI_Pack(n_ptr, 1, MPI_INT, buffer, 100,
         &position, MPI_COMM_WORLD);
      /* Position has been incremented again. */

      /* Now broadcast contents of buffer */
      MPI_Bcast(buffer, 100, MPI_PACKED, 0,
         MPI_COMM_WORLD);
   } else {
      MPI_Bcast(buffer, 100, MPI_PACKED, 0,
         MPI_COMM_WORLD);

      /* Now unpack the contents of buffer */
      position = 0;
      MPI_Unpack(buffer, 100, &position, a_ptr, 1,
         MPI_FLOAT, MPI_COMM_WORLD);
      /* Once again position has been incremented: */
      /* it now references the beginning of b.  */

      MPI_Unpack(buffer, 100, &position, b_ptr, 1,
         MPI_FLOAT, MPI_COMM_WORLD);
      MPI_Unpack(buffer, 100, &position, n_ptr, 1,
         MPI_INT, MPI_COMM_WORLD);
   }
} /* Get_data4 */


/*********************************************************************/
float Trap(local_a, local_b, local_n, h)
float  local_a;  /* in */
float  local_b;  /* in */
int    local_n;  /* in */
float  h;        /* in */
{

   float integral;  /* Store result in integral  */
   float x;
   int i;

   integral = (f(local_a) + f(local_b))/2.0;
   x = local_a;
   for (i = 1; i <= local_n-1; i++) {
      x = x + h;
      integral = integral + f(x);
   }
   integral = integral*h;
   return integral;
} /*  Trap  */


/*********************************************************************/
float f(x)
float x;
{
   float return_val;
   /* Calculate f(x). */
   /* Store calculation in return_val. */
   return_val = x*x;
   return return_val;
} /* f */
```

```
/* serial_dot.c -- compute a dot product on a single processor.
 *
 * Input:
```

```
 *    n: order of vectors
 *    x, y:  the vectors
 *
 * Output:
 *    the dot product of x and y.
 *
 * Note:  Arrays containing vectors are statically allocated.
 *
 */

//#define UNIX

#ifdef UNIX
#include <stdio.h>
#include "mpi.h"
#else
#include "mpi1.h"
#endif

#define MAX_ORDER 100

void Read_vector();
float Serial_dot();

#ifdef UNIX
void main (argc, argv)
#else
int MPI_main (argc, argv)
#endif
int argc;
char* argv[];
{
   float  x[MAX_ORDER];
   float  y[MAX_ORDER];
   int    n;
   float  dot;

   printf("Enter the order of the vectors\n\n");
/*
   scanf("%d\n\n", &n);
*/
n = 4;
   Read_vector("the first vector", x, n);
x[0]=3; x[1]=1; x[2]=4; x[3]=2;
   Read_vector("the second vector", y, n);
y[0]=2; y[1]=5; y[2]=1; y[3]=3;
   dot = Serial_dot(x, y, n);
   printf("The dot product is %f\n", dot);
} /* main */


/*********************************************************************/
void Read_vector(prompt, v, n)
char*  prompt;  /* in  */
float* v;       /* out */
int    n;       /* in  */
{
   int i;

   printf("Enter %s\n\n", prompt);
   for (i = 0; i < n; i++)
   {

            v[i] = 0;
/*
        scanf("%f", &v[i]);
*/
   }
} /* Read_vector */
```

```
/*********************************************************************/
float Serial_dot(x, y, n)
float* x;  /* in */
float* y;  /* in */
int    n;  /* in */
{

   int    i;
   float  sum = 0.0;

   for (i = 0; i < n; i++)
   {
      sum = sum + x[i]*y[i];
   }
   return sum;
} /* Serial_dot */
```

```
/* parallel_dot.c -- compute a dot product of a vector distributed among
 *    the processes.  Uses a block distribution of the vectors.
 *
 * Input:
 *    n: global order of vectors
 *    x, y:  the vectors
 *
 * Output:
 *    the dot product of x and y.
 *
 * Note:  Arrays containing vectors are statically allocated.  Assumes
 *    n, the global order of the vectors, is divisible by p, the number
 *    of processes.
 *
 */

//#define UNIX

#ifdef UNIX
#include <stdio.h>
#include "mpi.h"
#else
#include "mpi2.h"
#endif

#define MAX_LOCAL_ORDER 100

float  Serial_dot();
void Read_vector();
float Parallel_dot();

#ifdef UNIX
void main (argc, argv)
#else
int MPI_main (argc, argv)
#endif
int argc;
char* argv[];
{
   float  local_x[MAX_LOCAL_ORDER];
   float  local_y[MAX_LOCAL_ORDER];
   int    n;
   int    n_bar;  /* = n/p */
   float  dot;
   int    p;
   int    my_rank;

   MPI_Init(&argc, &argv);
   MPI_Comm_size(MPI_COMM_WORLD, &p);
   MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
```

```
    if (my_rank == 0) {
       printf("Enter the order of the vectors\n\n");
       scanf("%d", &n);
    }
    MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
    n_bar = n/p;

    Read_vector("the first vector", local_x, n_bar, p, my_rank);
    Read_vector("the second vector", local_y, n_bar, p, my_rank);

    dot = Parallel_dot(local_x, local_y, n_bar);

    if (my_rank == 0)
       printf("The dot product is %f\n", dot);

    MPI_Finalize();
} /* main */


/*********************************************************************/
void Read_vector(prompt, local_v, n_bar, p, my_rank)
char*  prompt;    /* in  */
float* local_v;   /* out */
int    n_bar;     /* in  */
int    p;         /* in  */
int    my_rank;   /* in  */
{
    int i, q;
    float temp[MAX_LOCAL_ORDER];
    MPI_Status status;

    if (my_rank == 0) {
       printf("Enter %s\n\n", prompt);
       for (i = 0; i < n_bar; i++)
       {
          local_v[i] = 0;
          scanf("%f", &local_v[i]);
       }
       for (q = 1; q < p; q++) {
          for (i = 0; i < n_bar; i++)
          {
             temp[i] = 0;
             scanf("%f", &temp[i]);
          }
          MPI_Send(temp, n_bar, MPI_FLOAT, q, 0, MPI_COMM_WORLD);
       }
    } else {
       MPI_Recv(local_v, n_bar, MPI_FLOAT, 0, 0, MPI_COMM_WORLD,
          &status);
    }
} /* Read_vector */


/*********************************************************************/
float Serial_dot(x, y, n)
float* x;  /* in */
float* y;  /* in */
int    n;  /* in */
{
    int    i;
    float  sum = 0.0;

    for (i = 0; i < n; i++)
       sum = sum + x[i]*y[i];

    return sum;
} /* Serial_dot */


/*********************************************************************/
```

```
float Parallel_dot(local_x, local_y, n_bar)
float* local_x; /* in */
float* local_y; /* in */
int n_bar;      /* in */
{
   float local_dot = 0.0;
   float dot = 0.0;

   local_dot = Serial_dot(local_x, local_y, n_bar);
   MPI_Reduce(&local_dot, &dot, 1, MPI_FLOAT,
      MPI_SUM, 0, MPI_COMM_WORLD);

   return dot;
} /* Parallel_dot */
```

```
/* count.c -- send a subvector from process 0 to process 1
 *
 * Input: none
 * Output: contents of vector received by process 1
 *
 * Note: Program should only be run with 2 processes.
 *
 */

//#define UNIX

#ifdef UNIX
#include <stdio.h>
#include "mpi.h"
#else
#include "mpi2.h"
#endif

#ifdef UNIX
void main (argc, argv)
#else
int MPI_main (argc, argv)
#endif
int argc;
char* argv[];
{
   float vector[100];
   MPI_Status status;
   int p;
   int my_rank;
   int i;

   MPI_Init(&argc, &argv);
   MPI_Comm_size(MPI_COMM_WORLD, &p);
   MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

   /* Initialize vector and send */
   if (my_rank == 0) {
      for (i = 0; i < 60; i++)
         vector[i] = 0.0;
      for (i = 60; i < 100; i++)
         vector[i] = 1.0;
      MPI_Send(vector+50, 50, MPI_FLOAT, 1, 0,
         MPI_COMM_WORLD);
   } else { /* my_rank == 1 */
      MPI_Recv(vector+50, 50, MPI_FLOAT, 0, 0,
         MPI_COMM_WORLD, &status);
      for (i = 50; i < 100; i++)
         printf("%3.1f ",vector[i]);
      printf("\n");
   }

   MPI_Finalize();
} /* main */
```

```
/* parallel_dot1.c -- Computes a parallel dot product.  Uses MPI_Allreduce.
 *
 * Input:
 *    n: order of vectors
 *    x, y:  the vectors
 *
 * Output:
 *    the dot product of x and y as computed by each process.
 *
 * Note:  Arrays containing vectors are statically allocated.  Assumes that
 *    n, the global order of the vectors, is evenly divisible by p, the
 *    number of processes.
 *
 */

//#define UNIX

#ifdef UNIX
#include <stdio.h>
#include "mpi.h"
#else
#include "mpi2.h"
#endif

float Serial_dot();
void Read_vector();
float Parallel_dot();
void Print_results();

#define MAX_LOCAL_ORDER 100

#ifdef UNIX
void main (argc, argv)
#else
int MPI_main (argc, argv)
#endif
int argc;
char* argv[];
{
   float  local_x[MAX_LOCAL_ORDER];
   float  local_y[MAX_LOCAL_ORDER];
   int    n;
   int    n_bar;  /* = n/p */
   float  dot;
   int    p;
   int    my_rank;

   MPI_Init(&argc, &argv);
   MPI_Comm_size(MPI_COMM_WORLD, &p);
   MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

   if (my_rank == 0) {
      printf("Enter the order of the vectors\n");
      scanf("%d", &n);
   }
   MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
   n_bar = n/p;

   Read_vector("the first vector", local_x, n_bar, p, my_rank);
   Read_vector("the second vector", local_y, n_bar, p, my_rank);

   dot = Parallel_dot(local_x, local_y, n_bar);

   Print_results(dot, my_rank, p);

   MPI_Finalize();
} /* main */
```

```
/*********************************************************************/
void Read_vector(prompt, local_v, n_bar, p, my_rank)
char*  prompt;    /* in  */
float* local_v;   /* out */
int    n_bar;     /* in  */
int    p;         /* in  */
int    my_rank;   /* in  */
{
   int i, q;
   float temp[MAX_LOCAL_ORDER];
   MPI_Status status;

   if (my_rank == 0) {
      printf("Enter %s\n", prompt);
      for (i = 0; i < n_bar; i++)
      {
         local_v[i] = 0;
         scanf("%f", &local_v[i]);
      }
      for (q = 1; q < p; q++) {
         for (i = 0; i < n_bar; i++)
         {
            temp[i] = 0;
            scanf("%f", &temp[i]);
         }
         MPI_Send(temp, n_bar, MPI_FLOAT, q, 0, MPI_COMM_WORLD);
      }
   } else {
      MPI_Recv(local_v, n_bar, MPI_FLOAT, 0, 0, MPI_COMM_WORLD,
         &status);
   }
} /* Read_vector */


/*********************************************************************/
float Serial_dot(x, y, n)
float* x; /* in */
float* y; /* in */
int    n; /* in */
{

   int   i;
   float sum = 0.0;

   for (i = 0; i < n; i++)
      sum = sum + x[i]*y[i];
   return sum;
} /* Serial_dot */


/*********************************************************************/
float Parallel_dot(local_x, local_y, n_bar)
float* local_x; /* in */
float* local_y; /* in */
int n_bar;      /* in */
{

   float local_dot;
   float dot = 0.0;

   local_dot = Serial_dot(local_x, local_y, n_bar);
   MPI_Allreduce(&local_dot, &dot, 1, MPI_FLOAT,
      MPI_SUM, MPI_COMM_WORLD);
   return dot;
} /* Parallel_dot */


/*********************************************************************/
void Print_results(dot, my_rank, p)
```

```
float  dot;     /* in */
int    my_rank; /* in */
int    p;       /* in */
{
   int       q;
   float     temp;
   MPI_Status  status;

   if (my_rank == 0) {
      printf("dot = \n");
      printf("Process 0 > %f\n", dot);
      for (q = 1; q < p; q++) {
         MPI_Recv(&temp, 1, MPI_FLOAT, q, 0, MPI_COMM_WORLD,
            &status);
         printf("Process %d > %f\n", q, temp);
      }
   } else {
      MPI_Send(&dot, 1, MPI_FLOAT, 0, 0, MPI_COMM_WORLD);
   }

} /* Print_results */
```

```
/* serial_mat_vect.c -- computes a matrix-vector product on a single processor.
 *
 * Input:
 *    m, n: order of matrix
 *    A, x:  the matrix and the vector to be multiplied
 *
 * Output:
 *    y: the product vector
 *
 * Note:  A, x, and y are statically allocated.
 *
 */

//#define UNIX

#ifdef UNIX
#include <stdio.h>
#include "mpi.h"
#else
#include "mpi1.h"
#endif

#define MAX_ORDER 100

typedef float MATRIX_T[MAX_ORDER][MAX_ORDER];
typedef float VECTOR_T[MAX_ORDER];

void Read_matrix();
void Read_vector();
void Serial_matrix_vector_prod();
void Print_vector();

#ifdef UNIX
void main (argc, argv)
#else
int MPI_main (argc, argv)
#endif
int argc;
char* argv[];
{
   float  A[MAX_ORDER][MAX_ORDER];
   float    x[MAX_ORDER];
   float    y[MAX_ORDER];
   int      m, n, i, j;

   printf("Enter the order of the matrix (m x n)\n\n");
/*
```

```
   scanf("%d %d", &m, &n);
*/
m=n=4;
   Read_matrix("the matrix", A, m, n);
   Read_vector("the vector", x, m);
   Serial_matrix_vector_prod(A, m, n, x, y);
   Print_vector(y, n);
} /* main */


/*********************************************************************/
void Read_matrix(prompt, A, m, n)
char*    prompt; /* in */
VECTOR_T* A;      /* out */
int      m;      /* in */
int      n;      /* in */
{
   int i, j;

   printf("Enter %s\n\n", prompt);
   for (i = 0; i < m; i++)
      for (j = 0; j < n; j++)
      {
         A[i][j] = 0.0;
/*
         scanf("%f", &A[i][j]);
*/
      }
A[0][0]=4; A[0][1]=2; A[0][2]=2; A[0][3]=1;
A[1][0]=5; A[1][1]=4; A[1][2]=7; A[1][3]=3;
A[2][0]=3; A[2][1]=5; A[2][2]=4; A[2][3]=4;
A[3][0]=7; A[3][1]=8; A[3][2]=3; A[3][3]=6;
} /* Read_matrix */


/*********************************************************************/
void Read_vector(prompt, v, n)
char*  prompt; /* in */
float* v;      /* out */
int    n;      /* in */
{
   int i;

   printf("Enter %s\n\n", prompt);
   for (i = 0; i < n; i++)
   {
      v[i] = 0;
/*
      scanf("%f", &v[i]);
*/
   }
v[0]=2 ; v[1]=1 ; v[2]=5 ; v[3]=3 ;
} /* Read_vector */


/*********************************************************************/
void Serial_matrix_vector_prod(A, m, n, x, y)
VECTOR_T* A;  /* in */
int      m;   /* in */
int      n;   /* in */
float*   x;   /* in */
float*   y;   /* out */
{

   int k, j;
   float z;

   for (k = 0; k < m; k++) {
      y[k] = 0.0;
      for (j = 0; j < n; j++)
```

```
            y[k] = y[k] + A[k][j]*x[j];
      }
   } /* Serial_matrix_vector_prod */


/*********************************************************************/
void Print_vector(y, n)
float* y;   /* in */
int    n;   /* in */
{
   int i;

   printf("Result is \n");
   for (i = 0; i < n; i++)
      printf("%4.1f ", y[i]);
   printf("\n");
} /* Print_vector */
```

```
/* parallel_mat_vect.c -- computes a parallel matrix-vector product.  Matrix
 *     is distributed by block rows.  Vectors are distributed by blocks.
 *
 * Input:
 *     m, n: order of matrix
 *     A, x: the matrix and the vector to be multiplied
 *
 * Output:
 *     y: the product vector
 *
 * Notes:
 *     1.  Local storage for A, x, and y is statically allocated.
 *     2.  Number of processes (p) should evenly divide both m and n.
 *
 */

//#define UNIX

#ifdef UNIX
#include <stdio.h>
#include "mpi.h"
#else
#include "mpi4.h"
#endif

#define MAX_ORDER 10

typedef float LOCAL_MATRIX_T[MAX_ORDER][MAX_ORDER];
typedef float LOCAL_VECTOR_T[MAX_ORDER];

void Read_matrix();
void Read_vector();
void Parallel_matrix_vector_prod();
void Print_matrix();
void Print_vector();

#ifdef UNIX
void main (argc, argv)
#else
int MPI_main (argc, argv)
#endif
int argc;
char* argv[];
{
   int           my_rank;
   int           p;
   LOCAL_MATRIX_T  local_A;
   float         global_x[MAX_ORDER];
   float         local_x[MAX_ORDER];
   float         local_y[MAX_ORDER];
   int           m, n, i,j;
```

```
    int             local_m, local_n;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &p);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

    if (my_rank == 0) {
       printf("Enter the order of the matrix (m x n)\n");
/*
       scanf("%d %d", &m, &n);
*/
m=n=4;
    }
    MPI_Bcast(&m, 1, MPI_INT, 0, MPI_COMM_WORLD);
    MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);

    local_m = m/p;
    local_n = n/p;

    Read_matrix("Enter the matrix", local_A, local_m, n, my_rank, p);
    Print_matrix("We read", local_A, local_m, n, my_rank, p);

    Read_vector("Enter the vector", local_x, local_n, my_rank, p);
    Print_vector("We read", local_x, local_n, my_rank, p);

    Parallel_matrix_vector_prod(local_A, m, n, local_x, global_x,
       local_y, local_m, local_n);
    Print_vector("The product is", local_y, local_m, my_rank, p);

    MPI_Finalize();

} /* main */


/*--------------------------------------------------------------------*/
void Read_matrix(prompt, local_A, local_m, n, my_rank, p)
char*         prompt;   /* in  */
LOCAL_VECTOR_T* local_A;  /* out */
int           local_m;  /* in  */
int           n;        /* in  */
int           my_rank;  /* in  */
int           p;        /* in  */
{

    int           i, j;
    LOCAL_MATRIX_T  temp;

    /* Fill dummy entries in temp with zeroes */
    for (i = 0; i < p*local_m; i++)
       for (j = n; j < MAX_ORDER; j++)
          temp[i][j] = 0.0;

    if (my_rank == 0) {
       printf("%s\n\n", prompt);
       for (i = 0; i < p*local_m; i++)
          for (j = 0; j < n; j++)
          {
             temp[i][j] = 0.0;
/*
             scanf("%f",&temp[i][j]);
*/
          }
temp[0][0]=4; temp[0][1]=2; temp[0][2]=2; temp[0][3]=1;
temp[1][0]=5; temp[1][1]=4; temp[1][2]=7; temp[1][3]=3;
temp[2][0]=3; temp[2][1]=5; temp[2][2]=4; temp[2][3]=4;
temp[3][0]=7; temp[3][1]=8; temp[3][2]=3; temp[3][3]=6;

    }
    MPI_Scatter((float*)temp, local_m*MAX_ORDER, MPI_FLOAT, (float*)local_A,
       local_m*MAX_ORDER, MPI_FLOAT, 0, MPI_COMM_WORLD);
```

```
} /* Read_matrix */


/*********************************************************************/
void Read_vector(prompt, local_x, local_n, my_rank, p)
char*  prompt;    /* in */
float* local_x;   /* out */
int    local_n;   /* in */
int    my_rank;   /* in */
int    p;         /* in */
{

   int  i;
   float temp[MAX_ORDER];

   if (my_rank == 0) {
      printf("%s\n\n", prompt);
      for (i = 0; i < p*local_n; i++)
      {
         temp[i] = 0;
/*
         scanf("%f", &temp[i]);
*/
      }
temp[0]=2 ; temp[1]=1 ; temp[2]=5 ; temp[3]=3 ;
   }
   MPI_Scatter(temp, local_n, MPI_FLOAT, local_x, local_n, MPI_FLOAT,
      0, MPI_COMM_WORLD);

} /* Read_vector */


/*********************************************************************/
/* All arrays are allocated in calling program */
/* Note that argument m is unused              */
void Parallel_matrix_vector_prod(local_A, m, n, local_x, global_x, local_y, local_m, local_n)
LOCAL_VECTOR_T* local_A;    /* in */
int             m;          /* in */
int             n;          /* in */
float*          local_x;    /* in */
float*          global_x;   /* in */
float*          local_y;    /* out */
int             local_m;    /* in */
int             local_n;    /* in */
{

   /* local_m = m/p, local_n = n/p */

   int i, j;

   MPI_Allgather(local_x, local_n, MPI_FLOAT,
                 global_x, local_n, MPI_FLOAT,
                 MPI_COMM_WORLD);

   for (i = 0; i < local_m; i++) {
      local_y[i] = 0.0;
      for (j = 0; j < n; j++)
         local_y[i] = local_y[i] +
                      local_A[i][j]*global_x[j];
   }
} /* Parallel_matrix_vector_prod */


/*********************************************************************/
void Print_matrix(title, local_A, local_m, n, my_rank, p)
char*           title;     /* in */
LOCAL_VECTOR_T* local_A;   /* in */
int             local_m;   /* in */
int             n;         /* in */
```

```
int          my_rank;   /* in */
int          p;         /* in */
{

   int  i, j;
   float temp[MAX_ORDER][MAX_ORDER];

   MPI_Gather((float*)local_A, local_m*MAX_ORDER, MPI_FLOAT, (float*)temp,
      local_m*MAX_ORDER, MPI_FLOAT, 0, MPI_COMM_WORLD);

   if (my_rank == 0) {
      printf("%s\n", title);
      for (i = 0; i < p*local_m; i++) {
         for (j = 0; j < n; j++)
            printf("%4.1f ", temp[i][j]);
         printf("\n");
      }
   }
} /* Print_matrix */


/*********************************************************************/
void Print_vector(title, local_y, local_m, my_rank, p)
char*  title;     /* in */
float* local_y;   /* in */
int    local_m;   /* in */
int    my_rank;   /* in */
int    p;         /* in */
{

   int  i;
   float temp[MAX_ORDER];

   MPI_Gather(local_y, local_m, MPI_FLOAT, temp, local_m, MPI_FLOAT,
      0, MPI_COMM_WORLD);

   if (my_rank == 0) {
      printf("%s\n", title);
      for (i = 0; i < p*local_m; i++)
         printf("%4.1f ", temp[i]);
      printf("\n");
   }
} /* Print_vector */
```

```
/* send_row.c -- send third row of a matrix from process 0 to process 1
 *
 * Input: none
 * Output: the row received by process 1
 *
 * Note:  Program should only be run with 2 processes
 *
 */

//#define UNIX

#ifdef UNIX
#include <stdio.h>
#include "mpi.h"
#else
#include "mpi2.h"
#endif

#ifdef UNIX
void main (argc, argv)
#else
int MPI_main (argc, argv)
#endif
int argc;
char* argv[];
```

```
{
   int p;
   int my_rank;
   float A[10][10];
   MPI_Status status;
   int i, j;

   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

   if (my_rank == 0) {
      for (i = 0; i < 10; i++)
         for (j = 0; j < 10; j++)
            A[i][j] = (float) i;
      MPI_Send(&(A[2][0]), 10, MPI_FLOAT, 1, 0,
         MPI_COMM_WORLD);
   } else { /* my_rank = 1 */
      MPI_Recv(&(A[2][0]), 10, MPI_FLOAT, 0, 0,
         MPI_COMM_WORLD, &status);
      for (j = 0; j < 10; j++)
         printf("%3.1f ", A[2][j]);
      printf("\n");
   }

   MPI_Finalize();
} /* main */
```

```
/* send_col.c -- send the third column of a matrix from process 0 to
 *    process 1
 *
 * Input:  None
 * Output:  The column received by process 1
 *
 * Note:  This program should only be run with 2 processes
 *
 */

//#define UNIX

#ifdef UNIX
#include <stdio.h>
#include "mpi.h"
#else
#include "mpi2.h"
#endif

#ifdef UNIX
void main (argc, argv)
#else
int MPI_main (argc, argv)
#endif
int argc;
char* argv[];
{
   int p;
   int my_rank;
   float A[10][10];
   MPI_Status status;
   MPI_Datatype column_mpi_t;
   int i, j;

   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

   MPI_Type_vector(10, 1, 10, MPI_FLOAT, &column_mpi_t);
   MPI_Type_commit(&column_mpi_t);

   if (my_rank == 0) {
      for (i = 0; i < 10; i++)
```

```
         for (j = 0; j < 10; j++)
            A[i][j] = (float) j;
      MPI_Send(&(A[0][2]), 1, column_mpi_t, 1, 0,
         MPI_COMM_WORLD);
   } else { /* my_rank = 1 */
      MPI_Recv(&(A[0][2]), 1, column_mpi_t, 0, 0,
         MPI_COMM_WORLD, &status);
      for (i = 0; i < 10; i++)
         printf("%3.1f ", A[i][2]);
      printf("\n");
   }

   MPI_Finalize();
} /* main */
```

```
/* send_col_to_row.c -- send column 1 of a matrix on process 0 to row 1
 *     on process 1.
 *
 * Input: none
 * Output: The row received by process 1.
 *
 * Note:  This program should only be run with 2 processes
 *
 */

//#define UNIX

#ifdef UNIX
#include <stdio.h>
#include "mpi.h"
#else
#include "mpi2.h"
#endif

#ifdef UNIX
void main (argc, argv)
#else
int MPI_main (argc, argv)
#endif
int argc;
char* argv[];
{
   int p;
   int my_rank;
   float A[10][10];
   MPI_Status status;
   MPI_Datatype column_mpi_t;
   int i, j;

   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

   MPI_Type_vector(10, 1, 10, MPI_FLOAT, &column_mpi_t);
   MPI_Type_commit(&column_mpi_t);

   if (my_rank == 0) {
      for (i = 0; i < 10; i++)
         for (j = 0; j < 10; j++)
            A[i][j] = (float) i;
      MPI_Send(&(A[0][0]), 1, column_mpi_t, 1, 0,
            MPI_COMM_WORLD);
   } else { /* my_rank = 1 */
      for (i = 0; i < 10; i++)
         for (j = 0; j < 10; j++)
            A[i][j] = 0.0;
      MPI_Recv(&(A[0][0]), 10, MPI_FLOAT, 0, 0,
            MPI_COMM_WORLD, &status);
      for (j = 0; j < 10; j++)
         printf("%3.1f ", A[0][j]);
```

```
      printf("\n");
   }

   MPI_Finalize();
} /* main */
```

```
/* send_triangle.c -- send the upper triangle of a matrix from process 0
 *    to process 1
 *
 * Input:  None
 * Output:  The matrix received by process 1
 *
 * Note:  This program should only be run with 2 processes.
 *
 */

//#define UNIX

#ifdef UNIX
#include <stdio.h>
#include "mpi.h"
#else
#include "mpi2.h"
#endif

#define n 10

#ifdef UNIX
void main (argc, argv)
#else
int MPI_main (argc, argv)
#endif
int argc;
char* argv[];
{
   int p;
   int my_rank;
   float       A[n][n];       /* Complete Matrix */
   float       T[n][n];       /* Upper Triangle  */
   int         displacements[n];
   int         block_lengths[n];
   MPI_Datatype  index_mpi_t;
   int         i, j;
   MPI_Status   status;

   MPI_Init(&argc, &argv);
   MPI_Comm_size(MPI_COMM_WORLD, &p);
   MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

   for (i = 0; i < n; i++) {
      block_lengths[i] = n-i;
      displacements[i] = (n+1)*i;
   }
   MPI_Type_indexed(n, block_lengths, displacements,
      MPI_FLOAT, &index_mpi_t);
   MPI_Type_commit(&index_mpi_t);

   if (my_rank == 0) {
      for (i = 0; i < n; i++)
         for (j = 0; j < n; j++)
            A[i][j] = (float) i + j;
      MPI_Send(&A[0][0], 1, index_mpi_t, 1, 0, MPI_COMM_WORLD);
   } else {/* my_rank == 1 */
      for (i = 0; i < n; i++)
         for (j = 0; j < n; j++)
            T[i][j] = 0.0;
      MPI_Recv(&T[0][0], 1, index_mpi_t, 0, 0, MPI_COMM_WORLD, &status);
      for (i = 0; i < n; i++) {
         for (j = 0; j < n; j++)
```

```
            printf("%4.1f ", T[i][j]);
         printf("\n\n");
         }
      }
   }

   MPI_Finalize();
} /* main */
```

```
/* sparse_row.c -- pack a row of a sparse matrix and send from process 0
 *    to process 1.  Process 1 allocates required storage after partially
 *    unpacking.
 *
 * Input: none
 * Output: the row received by process 1.
 *
 * Notes:
 *    1. This program should only be run with 2 processes.
 *    2. Only the row of the matrix is created on both processes.
 *
 */

//#define UNIX

#ifdef UNIX
#include <stdio.h>
#include <stdlib.h>
#include "mpi.h"
#else
#include "mpi2.h"
#endif

#define HUGE 22

#ifdef UNIX
void main (argc, argv)
#else
int MPI_main (argc, argv)
#endif
int argc;
char* argv[];
{
   int        p;
   int        my_rank;
   float*     entries;
   int*       column_subscripts;
   int        nonzeroes;
   int        position;
   int        row_number;
   char       buffer[HUGE]; /* HUGE is a predefined constant */
   MPI_Status  status;
   int        i;

   MPI_Init(&argc, &argv);
   MPI_Comm_size(MPI_COMM_WORLD, &p);
   MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

   if (my_rank == 0) {
      /* Get the number of nonzeros in the row. */
      /* Allocate storage for the row. */
      /* Initialize entries and column_subscripts */
      nonzeroes = 10;
      entries = (float*) malloc(nonzeroes*sizeof(float));
      column_subscripts = (int*) malloc(nonzeroes*sizeof(int));
      for (i = 0; i < nonzeroes; i++) {
         entries[i] = (float) 2*i;
         column_subscripts[i] = 3*i;
```

```
      }

      /* Now pack the data and send */
      position = 0;
      MPI_Pack(&nonzeroes, 1, MPI_INT, buffer, HUGE,
         &position, MPI_COMM_WORLD);
      MPI_Pack(&row_number, 1, MPI_INT, buffer, HUGE,
         &position, MPI_COMM_WORLD);
      MPI_Pack(entries, nonzeroes, MPI_FLOAT, buffer,
         HUGE, &position, MPI_COMM_WORLD);
      MPI_Pack(column_subscripts, nonzeroes, MPI_INT,
         buffer, HUGE, &position, MPI_COMM_WORLD);
      MPI_Send(buffer, position, MPI_PACKED, 1, 0,
         MPI_COMM_WORLD);


   } else { /* my_rank == 1 */
      MPI_Recv(buffer, HUGE, MPI_PACKED, 0, 0,
         MPI_COMM_WORLD, &status);
      position = 0;
      MPI_Unpack(buffer, HUGE, &position, &nonzeroes,
         1, MPI_INT, MPI_COMM_WORLD);
      MPI_Unpack(buffer, HUGE, &position, &row_number,
         1, MPI_INT, MPI_COMM_WORLD);
      /* Allocate storage for entries and column_subscripts */
      entries = (float *) malloc(nonzeroes*sizeof(float));
      column_subscripts = (int *) malloc(nonzeroes*sizeof(int));
      MPI_Unpack(buffer,HUGE, &position, entries,
         nonzeroes, MPI_FLOAT, MPI_COMM_WORLD);
      MPI_Unpack(buffer, HUGE, &position, column_subscripts,
         nonzeroes, MPI_INT, MPI_COMM_WORLD);
      for (i = 0; i < nonzeroes; i++)
         printf("%4.1f %2d\n", entries[i], column_subscripts[i]);
   }

   MPI_Finalize();
} /* main */
```

```
/* comm_test.c -- creates a communicator from the first q processes
 *    in a communicator containing p = q^2 processes.  Broadcasts
 *    an array to the members of the newly created communicator.
 *
 * Input: none
 * Output: Contents of array broadcast to each process in the newly
 *    created communicator
 *
 * Note:  MPI_COMM_WORLD should contain p = q^2 processes.
 *
 */

//#define UNIX

#ifdef UNIX
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include "mpi.h"
#else
#include "mpi4.h"
#endif

#ifdef UNIX
void main (argc, argv)
#else
int MPI_main (argc, argv)
#endif
int argc;
char* argv[];
{
```

```
    int       p;
    int       q; /* = sqrt(p) */
    int       my_rank;
    int       n_bar = 2;
    MPI_Group  group_world;
    MPI_Group  first_row_group;
    MPI_Comm   first_row_comm;
    int*       process_ranks;
    int       proc, i;
    float*     A_00;
    int       my_rank_in_first_row;


    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &p);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

    q = ceil(sqrt((double) p));

    /* Make a list of the processes in the new
     * communicator */
    process_ranks = (int*) malloc(q*sizeof(int));
    for (proc = 0; proc < q; proc++)
       process_ranks[proc] = proc;

    /* Get the group underlying MPI_COMM_WORLD */
    MPI_Comm_group(MPI_COMM_WORLD, &group_world);

    /* Create the new group */
    MPI_Group_incl(group_world, q, process_ranks,
       &first_row_group);

    /* Create the new communicator */
    MPI_Comm_create(MPI_COMM_WORLD, first_row_group,
       &first_row_comm);

    /* Now broadcast across the first row */
    if (my_rank < q) {
       MPI_Comm_rank(first_row_comm, &my_rank_in_first_row);

       /* Allocate space for A_00 */
       A_00 = (float*) malloc (n_bar*n_bar*sizeof(float));
       if (my_rank_in_first_row == 0) {
          /* Initialize A_00 */
          for (i = 0; i < n_bar*n_bar; i++)
             A_00[i] = (float) i;
       }
       MPI_Bcast(A_00, n_bar*n_bar, MPI_FLOAT, 0,
          first_row_comm);

       printf("Process %d > ", my_rank);
       for (i = 0; i < n_bar*n_bar; i++)
          printf("%4.1f ", A_00[i]);
       printf("\n\n");
    }
    MPI_Finalize();
} /* main */
```

```
/* comm_split.c -- build a collection of q communicators using MPI_Comm_split
 *
 * Input: none
 * Output:  Results of doing a broadcast across each of the q communicators.
 *
 * Note:  Assumes the number of processes, p = q^2
 *
 */

//#define UNIX
```

```
#ifdef UNIX
#include <stdio.h>
#include <math.h>
#include "mpi.h"
#else
#include "mpi4.h"
#endif

#ifdef UNIX
void main (argc, argv)
#else
int MPI_main (argc, argv)
#endif
int argc;
char* argv[];
{
   int      p;
   int      my_rank;
   MPI_Comm  my_row_comm;
   int      my_row;
   int      q;
   int      test;
   int      my_rank_in_row;

   MPI_Init(&argc, &argv);
   MPI_Comm_size(MPI_COMM_WORLD, &p);
   MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

   q = ceil(sqrt((double) p));

   /* my_rank is rank in MPI_COMM_WORLD.
    * q*q = p */
   my_row = my_rank/q;
   MPI_Comm_split(MPI_COMM_WORLD, my_row, my_rank,
      &my_row_comm);

   /* Test the new communicators */
   MPI_Comm_rank(my_row_comm, &my_rank_in_row);

   if (my_rank_in_row == 0)
      test = my_row;
   else
      test = 0;

   MPI_Bcast(&test, 1, MPI_INT, 0, my_row_comm);

   printf("Process %d > my_row = %d, my_rank_in_row = %d, test = %d\n",
      my_rank, my_row, my_rank_in_row, test);

   MPI_Finalize();
} /* main */
```

```
/* top_fcns.c -- test basic topology functions
 *
 * Input: none
 * Output: results of calls to various functions testing topology
 *    creation
 *
 * Algorithm:
 *    1. Build a 2-dimensional Cartesian communicator from
 *       MPI_Comm_world
 *    2. Print topology information for each process
 *    3. Use MPI_Cart_sub to build a communicator for each
 *       row of the Cartesian communicator
 *    4. Carry out a broadcast across each row communicator
 *    5. Print results of broadcast
```

```
 *    6. Use MPI_Cart_sub to build a communicator for each
 *       column of the Cartesian communicator
 *    7. Carry out a broadcast across each column communicator
 *    8. Print results of broadcast
 *
 * Note: Assumes the number of processes, p, is a perfect square
 *
 */

//#define UNIX

#ifdef UNIX
#include <stdio.h>
#include <math.h>
#include "mpi.h"
#else
#include "mpi8.h"
#endif

#ifdef UNIX
void main (argc, argv)
#else
int MPI_main (argc, argv)
#endif
int argc;
char* argv[];
{
   int      p;
   int      my_rank;
   int      q;
   MPI_Comm  grid_comm;
   int      dim_sizes[2];
   int      wrap_around[2];
   int      reorder = 1;
   int      coordinates[2];
   int      my_grid_rank;
   int      grid_rank;
   int      free_coords[2];
   MPI_Comm  row_comm;
   MPI_Comm  col_comm;
   int      row_test;
   int      col_test;


   MPI_Init(&argc, &argv);
   MPI_Comm_size(MPI_COMM_WORLD, &p);
   MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

   q = ceil(sqrt((double) p));

   dim_sizes[0] = dim_sizes[1] = q;
   wrap_around[0] = wrap_around[1] = 1;
   MPI_Cart_create(MPI_COMM_WORLD, 2, dim_sizes, wrap_around, reorder, &grid_comm);

   MPI_Comm_rank(grid_comm, &my_grid_rank);
   MPI_Cart_coords(grid_comm, my_grid_rank, 2, coordinates);

   MPI_Cart_rank(grid_comm, coordinates, &grid_rank);

   printf("Process %d > my_grid_rank = %d, coords = (%d,%d), grid_rank = %d\n",
      my_rank, my_grid_rank, coordinates[0], coordinates[1], grid_rank);

   free_coords[0] = 0;
   free_coords[1] = 1;
   MPI_Cart_sub(grid_comm, free_coords, &row_comm);

   if (coordinates[1] == 0)
      row_test = coordinates[0];
   else
      row_test = -1;
```

```
/*
   if (coordinates[0] == 0)
      row_test = coordinates[1];
   else
      row_test = -1;
*/
   MPI_Bcast(&row_test, 1, MPI_INT, 0, row_comm);

   printf("row_comm %d Process %d > coords = (%d,%d), row_test = %d\n",
       row_comm, my_rank, coordinates[0], coordinates[1], row_test);

   free_coords[0] = 1;
   free_coords[1] = 0;
   MPI_Cart_sub(grid_comm, free_coords, &col_comm);
   if (coordinates[0] == 0)
      col_test = coordinates[1];
   else
      col_test = -1;
   MPI_Bcast(&col_test, 1, MPI_INT, 0, col_comm);

   printf("Process %d > coords = (%d,%d), col_test = %d\n",
     my_rank, coordinates[0], coordinates[1], col_test);

   MPI_Finalize();
} /* main */
```

```
/* parallel_jacobi.c -- parallel implementation of Jacobi's method
 *    for solving the linear system Ax = b.  Uses block distribution
 *    of vectors and block-row distribution of A.
 *
 * Input:
 *    n:  order of system
 *    tol:  convergence tolerance
 *    max_iter:  maximum number of iterations
 *    A:  coefficient matrix
 *    b:  right-hand side of system
 *
 * Output:
 *    x:  the solution if the method converges
 *    max_iter:  if the method fails to converge
 *
 * Notes:
 *    1.  A should be strongly diagonally dominant in
 *        order to insure convergence.
 *    2.  A, x, and b are statically allocated.
 *
 */

//#define UNIX

#ifdef UNIX
#include <stdio.h>
#include <math.h>
#include "mpi.h"
#else
#include "mpi4.h"
#endif

#define MAX_DIM 4
#define Swap(x,y) { float* temp; temp = x; x = y; y = temp;}

float Distance ();
int Parallel_jacobi ();
void Read_matrix ();
void Read_vector ();
void Print_matrix ();
void Print_vector ();

#ifdef UNIX
```

```
void main (argc, argv)
#else
int MPI_main (argc, argv)
#endif
int argc;
char* argv[];
{
   int       p;
   int       my_rank;
   float     A_local[MAX_DIM*MAX_DIM];
   float     x_local[MAX_DIM];
   float     b_local[MAX_DIM];
   int       n;
   float     tol;
   int       max_iter;
   int       converged;

   MPI_Init(&argc, &argv);
   MPI_Comm_size(MPI_COMM_WORLD, &p);
   MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

   if (my_rank == 0)
   {
            tol=0.01;
      printf("Enter n, tolerance, and max number of iterations\n\n");
/*
      scanf("%d %f %d", &n, &tol, &max_iter);
*/
n=4; tol=0.01; max_iter=100;
   }

   MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
   MPI_Bcast(&tol, 1, MPI_FLOAT, 0, MPI_COMM_WORLD);
   MPI_Bcast(&max_iter, 1, MPI_INT, 0, MPI_COMM_WORLD);

   Read_matrix("Enter the matrix here\n", A_local, n, my_rank, p);
   Print_matrix("Matrix entered\n", A_local, n, my_rank, p);
   Read_vector("Enter right-hand side vector\n", b_local, n, my_rank, p);
   Print_vector("Vector entered at right-hand side", b_local, n, my_rank, p);

   converged =
     Parallel_jacobi(A_local, x_local, b_local, n, tol, max_iter, p, my_rank);

   if (converged)
      Print_vector("The solution is", x_local, n, my_rank, p);
   else
      if (my_rank == 0)
         printf("Failed to converge in %d iterations\n", max_iter);

   MPI_Finalize();
} /* main */


/*****************************************************/
/* Return 1 if iteration converged, 0 otherwise */
/* MATRIX_T is a 2-dimensional array          */

int Parallel_jacobi (A_local, x_local, b_local, n, tol, max_iter, p, my_rank)
float*      A_local;
float*  x_local;        /* out */
float*  b_local;
int     n;
float   tol;
int     max_iter;
int     p;
int     my_rank;
{
   int     i_local, i_global, j;
   int     n_bar;
   int     iter_num;
```

```
    float  x_temp1[MAX_DIM];
    float  x_temp2[MAX_DIM];
    float* x_old;
    float* x_new;
    float* x_tmp;
    float a, b, c, x, y;

    n_bar = n/p;

    /* Initialize x */
    MPI_Allgather
      (b_local, n_bar, MPI_FLOAT, x_temp1, n_bar, MPI_FLOAT, MPI_COMM_WORLD);
    x_new = x_temp1;
    x_old = x_temp2;

    iter_num = 0;
    do
    {
       iter_num++;

       /* Interchange x_old and x_new */
       Swap(x_old, x_new);
       for (i_local = 0; i_local < n_bar; i_local++)
       {
          i_global = i_local + my_rank*n_bar;
          x_local[i_local] = b_local[i_local];
          for (j = 0; j < i_global; j++)
             x_local[i_local]
               = x_local[i_local] - 1*A_local[MAX_DIM*i_local+j]*x_old[j];
          for (j = i_global+1; j < n; j++)
             x_local[i_local]
               = x_local[i_local] - 1*A_local[MAX_DIM*i_local+j]*x_old[j];

          x_local[i_local]
            = x_local[i_local]/A_local[MAX_DIM*i_local+i_global];
       }

       MPI_Allgather
         (x_local, n_bar, MPI_FLOAT, x_new, n_bar, MPI_FLOAT, MPI_COMM_WORLD);
    }
    while ((iter_num < max_iter) && (Distance(x_new,x_old,n) >= (tol*tol) ));

    if (Distance(x_new,x_old,n) < (tol*tol))
       return 1;
    else
       return 0;
} /* Jacobi */


/*********************************************************************/

float Distance(x, y, n)
float* x;
float* y;
int n;
{
    int i;
    float sum = 0.0;

    for (i = 0; i < n; i++)
    {
       sum = sum + (x[i] - y[i])*(x[i] - y[i]);
    }
    return (sum);
} /* Distance */


/*********************************************************************/

void Read_matrix(prompt, A_local, n, my_rank, p)
```

```
char*      prompt;
float* A_local; /* out */
int        n;
int        my_rank;
int        p;
{
   int                  i, j;
   float    temp[MAX_DIM*MAX_DIM];
   int                  n_bar;

   n_bar = n/p;

   /* Fill dummy entries in temp with zeroes */
   for (i = 0; i < n; i++)
      for (j = n; j < MAX_DIM; j++)
         temp[MAX_DIM*i+j] = 0.0;

   if (my_rank == 0)
   {
      printf("%s\n", prompt);
      for (i = 0; i < n; i++)
         for (j = 0; j < n; j++)
;
/*
            scanf("%f",&temp[MAX_DIM*i+j]);
*/
temp[0] = 9.0; temp[1] = 1; temp[2] = 3; temp[3] = 2;
temp[4] = 2; temp[5] = 6; temp[6] = 1; temp[7] = 2;
temp[8] = 3; temp[9] = 1; temp[10] = 7; temp[11] = 2;
temp[12] = 1; temp[13] = 1; temp[14] = 1; temp[15] = 5;

   }

   MPI_Scatter(&temp[0], n_bar*MAX_DIM, MPI_FLOAT,
            A_local, n_bar*MAX_DIM, MPI_FLOAT, 0, MPI_COMM_WORLD);
} /* Read_matrix */


/*********************************************************************/

void Read_vector (prompt, x_local, n, my_rank, p)
char*      prompt;
float*     x_local; /* out */
int        n;
int        my_rank;
int        p;
{
   int   i;
   float temp[MAX_DIM];
   int   n_bar;

   n_bar = n/p;

   if (my_rank==0)
   {
      printf("%s\n", prompt);
      for (i = 0; i < n; i++)
      {
         temp[i] = 0.0;             /* without this scanf not working */
/*
         scanf("%f", &temp[i]);
*/
      }
temp[0] = 28; temp[1] = 25; temp[2] = 34; temp[3] = 26;
   }

   MPI_Scatter
     (temp, n_bar, MPI_FLOAT, x_local, n_bar, MPI_FLOAT, 0, MPI_COMM_WORLD);
} /* Read_vector */
```

```
/*********************************************************************/

void Print_matrix (title, A_local, n, my_rank, p)
char*       title;
float*      A_local;
int         n;
int         my_rank;
int         p;
{
   int      i, j;
   float temp[MAX_DIM*MAX_DIM];
   int      n_bar;

   n_bar = n/p;

   MPI_Gather(A_local, n_bar*MAX_DIM, MPI_FLOAT,
         &temp[0], n_bar*MAX_DIM, MPI_FLOAT, 0, MPI_COMM_WORLD);

   if (my_rank == 0)
   {
      printf("%s\n", title);
      for (i = 0; i < n; i++)
      {
         for (j = 0; j < n; j++)
            printf("%4.3f ", temp[MAX_DIM*i+j]);

         printf("\n");
      }
      printf("\n\n");
   }
} /* Print_matrix */


/*********************************************************************/

void Print_vector (title, x_local, n, my_rank, p)
char*       title;
float*      x_local;
int         n;
int         my_rank;
int         p;
{
   int   i;
   float temp[MAX_DIM];
   int   n_bar;

   n_bar = n/p;

   MPI_Gather
     (x_local, n_bar, MPI_FLOAT, temp, n_bar, MPI_FLOAT, 0, MPI_COMM_WORLD);

   if (my_rank == 0)
   {
      printf("%s\n", title);
      for (i = 0; i < n; i++)
         printf("%4.3f ", temp[i]);
      printf("\n");
   }
   printf("\n\n");
} /* Print_vector */
```

```
/* parallel_bitonic.c -- parallel bitonic sort of randomly generated list
 *    of integers
 *
 * Input:
 *    n: the global length of the list -- must be a power of 2.
 *
 * Output:
```

```
 *    The sorted list.
 *
 * Notes:
 *    1. Assumes the number of processes p = 2^d and p divides n.
 *    2. The lists are statically allocated -- size specified in MAX.
 *    3. Keys are in the range 0 -- KEY_MAX-1.
 *    4. Implementation can be made much more efficient by using
 *       pointers and avoiding re-copying lists in merges.
 *
 */

/* Get rand and qsort */

//#define UNIX

#ifdef UNIX
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "mpi.h"
#else
#include "mpi4.h"
#endif

#define MAX 100
#define LOW 0
#define HIGH 1
#define KEY_MAX 10000
#define key_mpi_t MPI_INT

typedef int KEY_T;

void Generate_local_list ();
void Print_list ();
void Local_sort ();
int  Key_compare ();
int  log_base2 ();
void Par_bitonic_sort_incr ();
void Par_bitonic_sort_decr ();
void Merge_split ();
void Merge_list_low ();
void Merge_list_high ();


/*********************************************************************/
#ifdef UNIX
void main(argc, argv)
#else
int MPI_main(argc, argv)
#endif
int argc;
char* argv[];
{
   int      list_size;        /* Local list size  */
   int      n;                /* Global list size */
   KEY_T    local_list[MAX];
   int      proc_set_size;
   int      my_rank;
   int      p;
   unsigned  and_bit;
   MPI_Comm  io_comm;

   MPI_Init(&argc, &argv);
   MPI_Comm_size(MPI_COMM_WORLD, &p);
   MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
   MPI_Comm_dup(MPI_COMM_WORLD, &io_comm);

   if (my_rank == 0)
   {
      printf("Enter the global list size.\n");
```

```
/*
      scanf("%d", &n);
*/
n = 20;
   }

   MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);

   list_size = n/p;

   Generate_local_list(list_size, local_list);

   Print_list("\n\n\nBefore local sort", list_size, local_list, io_comm);

   Local_sort(list_size, local_list);
/*
   Print_list("\nAfter local sort", list_size, local_list, io_comm);
*/
   /* and_bit is a bitmask that, when "anded" with  */
   /* my_rank, tells us whether we're working on an */
   /* increasing or decreasing list                 */

   for (proc_set_size = 2, and_bit = 2;
            proc_set_size <= p;
       proc_set_size = proc_set_size*2, and_bit = and_bit << 1)
      if ((my_rank & and_bit) == 0)
                        Par_bitonic_sort_incr
                           (list_size, local_list, proc_set_size, MPI_COMM_WORLD);
      else
                        Par_bitonic_sort_decr
                           (list_size, local_list, proc_set_size, MPI_COMM_WORLD);

   Print_list("\nAfter sort", list_size, local_list, io_comm);

   MPI_Finalize();
} /* main */


/*********************************************************************/
void Generate_local_list (list_size, local_list)
int         list_size;
KEY_T*   local_list;  /* out */
{
   int i;
   int my_rank;

   MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
   srand(my_rank);

   for (i = 0; i < list_size; i++)
#ifdef UNIX
      local_list[i] = rand() % KEY_MAX;
#else
      local_list[i] = (int)(fabs(rand()) * 10000) % KEY_MAX;
#endif
} /* Generate_local_list */


/*********************************************************************/
void Print_list (title, list_size, local_list, io_comm)
char*       title;
int         list_size;
KEY_T*   local_list;
MPI_Comm io_comm;
{
   int      i, q;
   int      p;
   int      my_rank;
   int      root = 0;
   MPI_Status  status;
```

```
    KEY_T           temp_list[MAX];

    MPI_Comm_size(io_comm, &p);
    MPI_Comm_rank(io_comm, &my_rank);

    if (my_rank == root)
    {
       printf("%s\n", title);
       for (q = 0; q < root; q++)
       {
          MPI_Recv(temp_list, list_size, key_mpi_t, q, 0, io_comm, &status);
          printf("Process %d > ",q);
          for (i = 0; i < list_size; i++)
             printf("%d ", temp_list[i]);
          printf("\n");
       }
       printf("Process %d > ", root);
       for (i = 0; i < list_size; i++)
          printf("%d ", local_list[i]);
       printf("\n");
       for (q = root+1; q < p; q++)
       {
          MPI_Recv(temp_list, list_size, key_mpi_t, q, 0, io_comm, &status);
          printf("Process %d > ",q);
          for (i = 0; i < list_size; i++)
             printf("%d ", temp_list[i]);
          printf("\n");
       }
    }
    else
    {
       MPI_Send(local_list, list_size, key_mpi_t, root, 0, io_comm);
    }

} /* Print_list */


/*********************************************************************/

void Local_sort (list_size, local_keys)
int list_size;
KEY_T* local_keys;  /* in/out */
{
            int i, j, k, tmp, temp[100];

            for (i=0; i<list_size; i++)
            {
                        tmp = 0;
                        for (j=0; j<list_size; j++)
                        {
                                    if (tmp < local_keys[j])
                                    {
                                                tmp = local_keys[j];
                                                k = j;
                                    }
                        }
                        temp[list_size-i-1] = tmp;
                        local_keys[k] = 0;
            }

            MPI_Comm_rank(MPI_COMM_WORLD, &k);

            for (i=0; i<list_size; i++)
                        local_keys[i] = temp[i];

}


/*********************************************************************/
```

```
int Key_compare (p, q)
KEY_T* p;
KEY_T* q;
{

    if (*p < *q)
        return -1;
    else if (*p == *q)
        return 0;
    else /* *p > *q */
        return 1;

} /* Key_compare */


/*****************************************************************/

int log_base2 (x)
int x;
{
    int count = 0;

    while (x > 1)
    {
        x = x/2;
        count++;
    }
    return count;
} /* log_base2 */


/*****************************************************************/

void Par_bitonic_sort_incr (list_size, local_list, proc_set_size, comm)
int         list_size;
KEY_T*      local_list; /* in/out */
int         proc_set_size;
MPI_Comm comm;
{
    unsigned  eor_bit;
    int       proc_set_dim;
    int       stage;
    int       partner;
    int       my_rank;

    MPI_Comm_rank(comm, &my_rank);

    proc_set_dim = log_base2(proc_set_size);
    eor_bit = 1 << (proc_set_dim - 1);
    for (stage = 0; stage < proc_set_dim; stage++)
    {
        partner = my_rank ^ eor_bit;
        if (my_rank < partner)
            Merge_split(list_size, local_list, LOW, partner, comm);
        else
            Merge_split(list_size, local_list, HIGH, partner, comm);
        eor_bit = eor_bit >> 1;
    }
} /* Par_bitonic_sort_incr */


/*****************************************************************/

void Par_bitonic_sort_decr (list_size, local_list, proc_set_size, comm)
int       list_size;
KEY_T*    local_list; /* in/out */
int       proc_set_size;
MPI_Comm  comm;
{
    unsigned  eor_bit;
```

```
    int     proc_set_dim;
    int     stage;
    int     partner;
    int     my_rank;

    MPI_Comm_rank(comm, &my_rank);

    proc_set_dim = log_base2(proc_set_size);
    eor_bit = 1 << (proc_set_dim - 1);
    for (stage = 0; stage < proc_set_dim; stage++)
    {
       partner = my_rank ^ eor_bit;
       if (my_rank > partner)
          Merge_split(list_size, local_list, LOW, partner, comm);
       else
          Merge_split(list_size, local_list, HIGH, partner, comm);
       eor_bit = eor_bit >> 1;
    }
} /* Par_bitonic_sort_decr */


/*****************************************************************/

void Merge_split (list_size, local_list, which_keys, partner, comm)
int         list_size;
KEY_T*      local_list; /* in/out */
int         which_keys;
int         partner;
MPI_Comm comm;
{

    MPI_Status status;
    KEY_T temp_list[MAX];

    /* key_mpi_t is an MPI (derived) type */

    MPI_Sendrecv(local_list, list_size, key_mpi_t, partner, 0, temp_list,
            list_size, key_mpi_t, partner, 0, comm, &status);

    if (which_keys == HIGH)
       Merge_list_high(list_size, local_list, temp_list);
    else
       Merge_list_low(list_size, local_list, temp_list);
} /* Merge_split */


/*****************************************************************/
/* Merges the contents of the two lists. */
/* Returns the smaller keys in list1     */

void Merge_list_low (list_size, list1, list2)
int         list_size;
KEY_T*      list1; /* in/out */
KEY_T*      list2;
{
    int  i;
    int  index1 = 0;
    int  index2 = 0;
    KEY_T scratch_list[MAX];

    for (i = 0; i < list_size; i++)
       if (list1[index1] <= list2[index2])
       {
          scratch_list[i] = list1[index1];
          index1++;
       }
       else
       {
          scratch_list[i] = list2[index2];
          index2++;
```

```
      }

   for (i = 0; i < list_size; i++)
      list1[i] = scratch_list[i];

} /* Merge_list_low */


/*****************************************************************/
/* Returns the larger keys in list 1.   */

void Merge_list_high (list_size, list1, list2)
int         list_size;
KEY_T*   list1; /* in/out */
KEY_T*   list2;
{
   int  i;
   int  index1 = list_size - 1;
   int  index2 = list_size - 1;
   KEY_T scratch_list[MAX];

   for (i = list_size - 1; i >= 0; i--)
      if (list1[index1] >= list2[index2])
      {
         scratch_list[i] = list1[index1];
         index1--;
      }
      else
      {
         scratch_list[i] = list2[index2];
         index2--;
      }

   for (i = 0; i < list_size; i++)
      list1[i] = scratch_list[i];
} /* Merge_list _high */
```

Appendix B

B. MPI C Binding Reference

int MPI_Abort(comm, errorcode) MPI_Comm comm; int errorcode;	Terminates MPI execution environment
int MPI_Address(location, address) void *location; MPI_Aint *address;	Gets the address of a location in memory
int MPI_Allgatherv (sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs, recvtype, comm) void *sendbuf; int sendcount; MPI_Datatype sendtype; void *recvbuf; int *recvcounts; int *displs; MPI_Datatype recvtype; MPI_Comm comm;	Gathers data from all tasks and deliver it to all
int MPI_Allgather (sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm) void *sendbuf; int sendcount; MPI_Datatype sendtype; void *recvbuf; int recvcount; MPI_Datatype recvtype; MPI_Comm comm;	Gathers data from all tasks and distribute it to all
int MPI_Allreduce (sendbuf, recvbuf, count, datatype, op, comm)	Combines values from all processes and distribute the result back to all processes

void *sendbuf; void *recvbuf; int count; MPI_Datatype datatype; MPI_Op op; MPI_Comm comm;	
int MPI_Alltoallv (sendbuf, sendcnts, sdispls, sendtype, recvbuf, recvcnts, rdispls, recvtype, comm) void *sendbuf; int *sendcnts; int *sdispls; MPI_Datatype sendtype; void *recvbuf; int *recvcnts; int *rdispls; MPI_Datatype recvtype; MPI_Comm comm;	Sends data from all to all processes, with a Displacement
int MPI_Alltoall(sendbuf, sendcount, sendtype, recvbuf, recvcnt, recvtype, comm) void *sendbuf; int sendcount; MPI_Datatype sendtype; void *recvbuf; int recvcnt; MPI_Datatype recvtype; MPI_Comm comm;	Sends data from all to all processes
int MPI_Attr_delete (comm, keyval) MPI_Comm comm; int keyval;	Deletes attribute value associated with a key
int MPI_Attr_get (comm, keyval, attr_value, flag) MPI_Comm comm;	Retrieves attribute value by key

int keyval; void *attr_value; int *flag;	
int MPI_Attr_put (comm, keyval, attr_value) MPI_Comm comm; int keyval; void *attr_value;	Stores attribute value associated with a key
int MPI_Barrier (comm) MPI_Comm comm;	Blocks until all process have reached this routine.
int MPI_Bcast (buffer, count, datatype, root, comm) void *buffer; int count; MPI_Datatype datatype; int root; MPI_Comm comm;	Broadcasts a message from the process with rank "root" to all other processes of the group.
int MPI_Bsend_init(buf, count, datatype, dest, tag, comm, request) void *buf; int count; MPI_Datatype datatype; int dest; int tag; MPI_Comm comm; MPI_Request *request;	Builds a handle for a buffered send
int MPI_Bsend(buf, count, datatype, dest, tag, comm) void *buf; int count, dest, tag; MPI_Datatype datatype; MPI_Comm comm; Basic send with user-specified buffering int MPI_Buffer_attach(buffer, size) void *buffer; int size;	Attaches a user-defined buffer for sending
int MPI_Buffer_detach(bufferptr, size)	Removes an existing buffer (for use in

void *bufferptr; int *size;	MPI_Bsend etc)
int MPI_Cancel(request) MPI_Request *request;	Cancels a communication request
int MPI_Cart_coords (comm, rank, maxdims, coords) MPI_Comm comm; int rank; int maxdims; int *coords;	Determines process coords in cartesian topology given rank in group
int MPI_Cart_create (comm_old, ndims, dims, periods, reorder, comm_cart) MPI_Comm comm_old; int ndims; int *dims; int *periods; int reorder; MPI_Comm *comm_cart;	Makes a new communicator to which topology information has been attached
int MPI_Cart_get (comm, maxdims, dims, periods, coords) MPI_Comm comm; int maxdims; int *dims, *periods, *coords;	Retrieves Cartesian topology information associated with a communicator
int MPI_Cart_map (comm_old, ndims, dims, periods, newrank) MPI_Comm comm_old; int ndims; int *dims; int *periods; int *newrank;	Maps process to Cartesian topology information
int MPI_Cart_rank (comm, coords, rank) MPI_Comm comm; int *coords; int *rank;	Determines process rank in communicator given Cartesian location
int MPI_Cart_shift (comm, direction, displ, source, dest)	Returns the shifted source and destination ranks, given a shift direction and amount

MPI_Comm comm; int direction; int displ; int *source; int *dest;	
int MPI_Cart_sub (comm, remain_dims, comm_new) MPI_Comm comm; int *remain_dims; MPI_Comm *comm_new;	Partitions a communicator into subgroups which form lower-dimensional cartesian subgrids
int MPI_Cartdim_get (comm, ndims) MPI_Comm comm; int *ndims;	Retrieves Cartesian topology information associated with a communicator
int MPI_Comm_compare (comm1, comm2, result) MPI_Comm comm1; MPI_Comm comm2; int *result;	Compares two communicators
int MPI_Comm_create (comm, group, comm_out) MPI_Comm comm; MPI_Group group; MPI_Comm *comm_out;	Creates a new communicator
int MPI_Comm_dup (comm, comm_out) MPI_Comm comm, *comm_out;	Duplicates an existing communicator with all its cached information
int MPI_Comm_free (commp) MPI_Comm *commp;	Marks the communicator object for deallocation
int MPI_Comm_group (comm, group) MPI_Comm comm; MPI_Group *group;	Accesses the group associated with given Communicator
int MPI_Comm_rank (comm, rank) MPI_Comm comm; int *rank;	Determines the rank of the calling process in the Communicator
int MPI_Comm_remote_group (comm, group) MPI_Comm comm; MPI_Group *group;	Accesses the remote group associated with the given intercommunicator
int MPI_Comm_remote_size (comm, size)	Determines the size of the remote group

MPI_Comm comm; int *size;	associated with an intercommunictor
int MPI_Comm_size (comm, size) MPI_Comm comm; int *size;	Determines the size of the group associated with a communictor
int MPI_Comm_split (comm, color, key, comm_out) MPI_Comm comm; int color, key; MPI_Comm *comm_out;	Creates new communicators based on colors and Keys
int MPI_Comm_test_inter (comm, flag) MPI_Comm comm; int *flag;	Tests to see if a comm is an intercommunicator
int MPIR_dup_fn (comm, keyval, extra_state, attr_in, attr_out, flag) MPI_Comm comm; int keyval; void *extra_state; void *attr_in; void *attr_out; int *flag;	A function to simplemindedly copy attributes
int MPI_Dims_create(nnodes, ndims, dims) int nnodes; int ndims; int *dims;	Creates a division of processors in a cartesian Grid
int MPI_Errhandler_create(function, errhandler) MPI_Handler_function *function; MPI_Errhandler *errhandler;	Creates an MPI-style errorhandler
int MPI_Errhandler_free(errhandler) MPI_Errhandler *errhandler;	Frees an MPI-style errorhandler
int MPI_Errhandler_get(comm, errhandler) MPI_Comm comm; MPI_Errhandler *errhandler;	Gets the error handler for a communicator
int MPI_Errhandler_set(comm, errhandler	Sets the error handler for a communicator

) MPI_Comm comm; MPI_Errhandler errhandler;	
int MPI_Error_class(errorcode, errorclass) int errorcode, *errorclass;	Converts an error code into an error class
int MPI_Error_string(errorcode, string, resultlen) int errorcode, *resultlen; char *string;	Return a string for a given error code
int MPI_Finalize()	Terminates MPI execution environment
int MPI_Gatherv (sendbuf, sendcnt, sendtype, recvbuf, recvcnts, displs, recvtype, root, comm) void *sendbuf; int sendcnt; MPI_Datatype sendtype; void *recvbuf; int *recvcnts; int *displs; MPI_Datatype recvtype; int root; MPI_Comm comm;	Gathers into specified locations from all processes in a group
int MPI_Gather (sendbuf, sendcnt, sendtype, recvbuf, recvcount, recvtype, root, comm) void *sendbuf; int sendcnt; MPI_Datatype sendtype; void *recvbuf; int recvcount; MPI_Datatype recvtype; int root; MPI_Comm comm;	Gathers together values from a group of processes
int MPI_Get_count(status, datatype,	Gets the number of "top level" elements

count) MPI_Status *status; MPI_Datatype datatype; int *count;	
int MPI_Get_elements (status, datatype, elements) MPI_Status *status; MPI_Datatype datatype; int *elements;	Returns the number of basic elements in a Datatype
int MPI_Get_processor_name(name, resultlen) char *name; int *resultlen;	Gets the name of the processor
int MPI_Graph_create (comm_old, nnodes, index, edges, reorder, comm_graph) MPI_Comm comm_old; int nnodes; int *index; int *edges; int reorder; MPI_Comm *comm_graph;	Makes a new communicator to which topology information has been attached
int MPI_Graph_get (comm, maxindex, maxedges, index, edges) MPI_Comm comm; int maxindex, maxedges; int *index, *edges;	Retrieves graph topology information associated with a communicator
int MPI_Graph_map (comm_old, nnodes, index, edges, newrank) MPI_Comm comm_old; int nnodes; int *index; int *edges; int *newrank;	Maps process to graph topology information
int MPI_Graph_neighbors_count (comm, rank, nneighbors) MPI_Comm comm;	Returns the number of neighbors of a node associated with a graph topology

int rank; int *nneighbors;	
int MPI_Graph_neighbors (comm, rank, maxneighbors, neighbors) MPI_Comm comm; int rank; int maxneighbors; int *neighbors;	Returns the neighbors of a node associated with a graph topology
int MPI_Graphdims_get (comm, nnodes, nedges) MPI_Comm comm; int *nnodes; int *nedges;	Retrieves graph topology information associated with a communicator
int MPI_Group_compare (group1, group2, result) MPI_Group group1; MPI_Group group2; int *result;	Compares two groups
int MPI_Group_difference (group1, group2, group_out) MPI_Group group1, group2, *group_out;	Makes a group from the difference of two groups
int MPI_Group_excl (group, n, ranks, newgroup) MPI_Group group, *newgroup; int n, *ranks;	Produces a group by reordering an existing group and taking only unlisted members
int MPI_Group_free (group) MPI_Group *group;	Frees a group
int MPI_Group_incl (group, n, ranks, group_out) MPI_Group group, *group_out; int n, *ranks;	Produces a group by reordering an existing group and taking only listed members
int MPI_Group_intersection (group1, group2, group_out) MPI_Group group1, group2, *group_out;	Produces a group as the intersection of two existing groups
int MPI_Group_range_excl (group, n, ranges, newgroup) MPI_Group group, *newgroup;	Produces a group by excluding ranges of processes from an existing group

int n, ranges[][3];	
int MPI_Group_range_incl (group, n, ranges, newgroup) MPI_Group group, *newgroup; int n, ranges[][3];	Creates a new group from ranges of ranks in an existing group
int MPI_Group_rank (group, rank) MPI_Group group; int *rank;	Returns the rank of this process in the given group
int MPI_Group_size (group, size) MPI_Group group; int *size;	Returns the size of a group
int MPI_Group_translate_ranks (group_a, n, ranks_a, group_b, ranks_b) MPI_Group group_a; int n; int *ranks_a; MPI_Group group_b; int *ranks_b;	Translates the ranks of processes in one group to those in another group
int MPI_Group_union (group1, group2, group_out) MPI_Group group1, group2, *group_out;	Produces a group by combining two groups
int MPI_Ibsend(buf, count, datatype, dest, tag, comm, request) void *buf; int count; MPI_Datatype datatype; int dest; int tag; MPI_Comm comm; MPI_Request *request;	Starts a nonblocking buffered send
int MPI_Initialized(flag) int *flag;	Indicates whether 'MPI_Init' has been called.
int MPI_Init(argc,argv) int *argc; char ***argv;	Initialize the MPI execution environment
int MPI_Intercomm_create (local_comm, local_leader, peer_comm,	Creates an intercommuncator from two Intracommunicators

remote_leader, tag, comm_out) MPI_Comm local_comm; int local_leader; MPI_Comm peer_comm; int remote_leader; int tag; MPI_Comm *comm_out;	
int MPI_Intercomm_merge (comm, high, comm_out) MPI_Comm comm; int high; MPI_Comm *comm_out;	Creates an intracommuncator from an Intercommunicator
int MPI_Iprobe(source, tag, comm, flag, status) int source; int tag; int *flag; MPI_Comm comm; MPI_Status *status;	Nonblocking test for a message
int MPI_Irecv(buf, count, datatype, source, tag, comm, request) void *buf; int count; MPI_Datatype datatype; int source; int tag; MPI_Comm comm; MPI_Request *request;	Begins a nonblocking receive
int MPI_Irsend(buf, count, datatype, dest, tag, comm, request) void *buf; int count; MPI_Datatype datatype; int dest; int tag; MPI_Comm comm;	Starts a nonblocking ready send

MPI_Request *request;	
int MPI_Isend(buf, count, datatype, dest, tag, comm, request) void *buf; int count; MPI_Datatype datatype; int dest; int tag; MPI_Comm comm; MPI_Request *request;	Begins a nonblocking send
int MPI_Issend(buf, count, datatype, dest, tag, comm, request) void *buf; int count; MPI_Datatype datatype; int dest; int tag; MPI_Comm comm; MPI_Request *request;	Starts a nonblocking synchronous send
int MPI_Keyval_create (copy_fn, delete_fn, keyval, extra_state) MPI_Copy_function *copy_fn; MPI_Delete_function *delete_fn; int *keyval; void *extra_state;	Generates a new attribute key
int MPI_Keyval_free (keyval) int *keyval;	Frees attribute key for communicator cache attribute
int MPI_Op_create(function, commute, op) MPI_User_function *function; int commute; MPI_Op *op; Creates a user-defined combination function handle int MPI_Op_free(op) MPI_Op *op;	Frees a user-defined combination function handle

int MPI_Pack_size (incount, datatype, comm, size) int incount; MPI_Datatype datatype; MPI_Comm comm; int *size;	Returns the upper bound on the amount of space needed to pack a message
int MPI_Pack (inbuf, incount, type, outbuf, outcount, position, comm) void *inbuf; int incount; MPI_Datatype type; void *outbuf; int outcount; int *position; MPI_Comm comm;	Packs a datatype into contiguous memory
int MPI_Pcontrol(level) int level;	Controls profiling
int MPI_Probe(source, tag, comm, status) int source; int tag; MPI_Comm comm; MPI_Status *status;	Blocking test for a message
int MPI_Recv_init(buf, count, datatype, source, tag, comm, request) void *buf; int count; MPI_Request *request; MPI_Datatype datatype; int source; int tag; MPI_Comm comm;	Builds a handle for a receive
int MPI_Recv(buf, count, datatype, source, tag, comm, status) void *buf; int count, source, tag;	Basic receive

MPI_Datatype datatype; MPI_Comm comm; MPI_Status *status;	
int MPI_Reduce_scatter (sendbuf, recvbuf, recvcnts, datatype, op, comm) void *sendbuf; void *recvbuf; int *recvcnts; MPI_Datatype datatype; MPI_Op op; MPI_Comm comm;	Combines values and scatters the results
int MPI_Reduce (sendbuf, recvbuf, count, datatype, op, root, comm) void *sendbuf; void *recvbuf; int count; MPI_Datatype datatype; MPI_Op op; int root; MPI_Comm comm; Reduces values on all processes to a single value int MPI_Request_free(request) MPI_Request *request;	Frees a communication request object
int MPI_Rsend_init(buf, count, datatype, dest, tag, comm, request) void *buf; int count; MPI_Datatype datatype; int dest; int tag; MPI_Comm comm; MPI_Request *request;	Builds a handle for a ready send
int MPI_Rsend(buf, count, datatype, dest, tag, comm) void *buf;	Basic ready send

int count, dest, tag; MPI_Datatype datatype; MPI_Comm comm;	
int MPI_Scan (sendbuf, recvbuf, count, datatype, op, comm) void *sendbuf; void *recvbuf; int count; MPI_Datatype datatype; MPI_Op op; MPI_Comm comm;	Computes the scan (partial reductions) of data on a collection of processes
int MPI_Scatterv (sendbuf, sendcnts, displs, sendtype, recvbuf, recvcnt, recvtype, root, comm) void *sendbuf; int *sendcnts; int *displs; MPI_Datatype sendtype; void *recvbuf; int recvcnt; MPI_Datatype recvtype; int root; MPI_Comm comm;	Scatters a buffer in parts to all tasks in a group
int MPI_Scatter (sendbuf, sendcnt, sendtype, recvbuf, recvcnt, recvtype, root, comm) void *sendbuf; int sendcnt; MPI_Datatype sendtype; void *recvbuf; int recvcnt; MPI_Datatype recvtype; int root; MPI_Comm comm;	Sends data from one task to all other tasks in a Group
int MPI_Send_init(buf, count, datatype,	Builds a handle for a standard send

dest, tag, comm, request) void *buf; int count; MPI_Datatype datatype; int dest; int tag; MPI_Comm comm; MPI_Request *request;	
int MPI_Sendrecv_replace(buf, count, datatype, dest, sendtag, source, recvtag, comm, status) void *buf; int count, dest, sendtag, source, recvtag; MPI_Datatype datatype; MPI_Comm comm; MPI_Status *status;	Sends and receives using a single buffer
int MPI_Sendrecv(sendbuf, sendcount, sendtype, dest, sendtag, recvbuf, recvcount, recvtype, source, recvtag, comm, status) void *sendbuf; int sendcount; MPI_Datatype sendtype; int dest, sendtag; void *recvbuf; int recvcount; MPI_Datatype recvtype; int source, recvtag; MPI_Comm comm; MPI_Status *status;	Sends and receives a message
int MPI_Send(buf, count, datatype, dest, tag, comm) void *buf; int count, dest, tag;	Performs a basic send

MPI_Datatype datatype; MPI_Comm comm;	
int MPI_Ssend_init(buf, count, datatype, dest, tag, comm, request) void *buf; int count; MPI_Datatype datatype; int dest; int tag; MPI_Comm comm; MPI_Request *request;	Builds a handle for a synchronous send
int MPI_Ssend(buf, count, datatype, dest, tag, comm) void *buf; int count, dest, tag; MPI_Datatype datatype; MPI_Comm comm;	Basic synchronous send
int MPI_Startall(count, array_of_requests) int count; MPI_Request array_of_requests[];	Starts a collection of requests
int MPI_Start(request) MPI_Request *request;	Initiates a communication with a persistent request handle
int MPI_Test_cancelled(status, flag) MPI_Status *status; int *flag;	Tests to see if a request was canceled
int MPI_Testall(count, array_of_requests, flag, array_of_statuses) int count; MPI_Request array_of_requests[]; int *flag; MPI_Status *array_of_statuses;	Tests for the completion of all previously initiated communications
int MPI_Testany(count, array_of_requests, index, flag, status)	Tests for completion of any previdously initiated Communication

int count; MPI_Request array_of_requests[]; int *index, *flag; MPI_Status *status;	
int MPI_Testsome(incount, array_of_requests, outcount, array_of_indices, array_of_statuses) int incount, *outcount, array_of_indices[]; MPI_Request array_of_requests[]; MPI_Status array_of_statuses[];	Tests for some given communications to complete
int MPI_Test (request, flag, status) MPI_Request *request; int *flag; MPI_Status *status; Tests for the completion of a send or receive int MPI_Topo_test (comm, top_type) MPI_Comm comm; int *top_type;	Determines the type of topology (if any) associated with a communicator
int MPI_Type_commit (datatype) MPI_Datatype *datatype; Commits the datatype int MPI_Type_contiguous(count, old_type, newtype) int count; MPI_Datatype old_type; MPI_Datatype *newtype;	Creates a contiguous datatype
int MPI_Type_extent(datatype, extent) MPI_Datatype datatype; MPI_Aint *extent; Returns the extent of a datatype int MPI_Type_free (datatype) MPI_Datatype *datatype;	Frees the datatype
int MPI_Type_hindexed(count, blocklens, indices, old_type, newtype)	Creates an indexed datatype with offsets in bytes

int count; int blocklens[]; MPI_Aint indices[]; MPI_Datatype old_type; MPI_Datatype *newtype;	
int MPI_Type_hvector(count, blocklen, stride, old_type, newtype) int count; int blocklen; MPI_Aint stride; MPI_Datatype old_type; MPI_Datatype *newtype;	Creates a vector (strided) datatype with offset in Bytes
int MPI_Type_indexed(count, blocklens, indices, old_type, newtype) int count; int blocklens[]; int indices[]; MPI_Datatype old_type; MPI_Datatype *newtype;	Creates an indexed datatype
int MPI_Type_lb (datatype, displacement) MPI_Datatype datatype; MPI_Aint *displacement; Returns the lower-bound of a datatype int MPI_Type_size (datatype, size) MPI_Datatype datatype; int *size;	Return the number of bytes occupied by entries in the datatype
int MPI_Type_struct(count, blocklens, indices, old_types, newtype) int count; int blocklens[]; MPI_Aint indices[]; MPI_Datatype old_types[]; MPI_Datatype *newtype;	Creates a struct datatype
int MPI_Type_ub (datatype, displacement) MPI_Datatype datatype;	Returns the upper bound of a datatype

MPI_Aint *displacement;	
int MPI_Type_vector(count, blocklen, stride, old_type, newtype) int count; int blocklen; int stride; MPI_Datatype old_type; MPI_Datatype *newtype;	Creates a vector (strided) datatype
int MPI_Unpack (inbuf, insize, position, outbuf, outcount, type, comm) void *inbuf; int insize; int *position; void *outbuf; int outcount; MPI_Datatype type; MPI_Comm comm;	Unpack a datatype into contiguous memory
int MPI_Waitall(count, array_of_requests, array_of_statuses) int count; MPI_Request array_of_requests[]; MPI_Status array_of_statuses[];	Waits for all given communications to complete
int MPI_Waitany(count, array_of_requests, index, status) int count; MPI_Request array_of_requests[]; int *index; MPI_Status *status;	Waits for any specified send or receive to Complete
int MPI_Waitsome(incount, array_of_requests, outcount, array_of_indices, array_of_statuses) int incount, *outcount, array_of_indices[]; MPI_Request array_of_requests[]; MPI_Status array_of_statuses[];	Waits for some given communications to Complete

int MPI_Wait (request, status) MPI_Request *request; MPI_Status *status;	Waits for an MPI send or receive to complete
double MPI_Wtick()	Returns the resolution of MPI_Wtime
double MPI_Wtime()	Returns an elapsed time on the calling processor

Bibliography

[1] E. Anderson, et al. "MPI: Message Passing Interface Standard", *Message Passing Interface Forum*, 1997.

[2] H. U. T. Nguyen, "CPSS: A Flexible and Efficent Simulator for Wormhole-Routed Multicomputers", *Master Degree Thesis at Concordia University*, 1997.

[3] B. Meek. "Fortran, PL/1, and the Algols," *Macmillan Press, London*, 1978.

[4] P. Pierce. 'The NX/2 Operating System," *Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications, pages 384-390. ACM Press*, 1988.

[5] G.A. Geist, et.al. "A User's Guide to PICL: a Portable Instrumented Communication Library," *Technical Report TM-11616, Oak Ridge National Laboratory*, October 1990.

[6] Parasoft Corporation, "Express User's Guide, version 3.2.5 edition," *Pasadena, CA*,1992.

[7] R. Butler, E. Lusk. "User's Guide to the p4 Programming System." *Technical report TM-ANL-92/17, Argonne National Laboratory*, 1992.

[8] J. Dongarra, et al. "Integrated PVM Framework Supports Heterogenous Network Computing." *Computers in Physics, 7(2):166-75*, April 1993.

[9] D. Walker. "Standards for Message Passing in a Distributed Memory Environment." *Technical Report TM-12147, Oak Ridge National Laboratory*, August 1992.

[10] J. J. Dongarra, R. Hempel, A. J. G. Hey, and D. W. Walker. "A Proposal for a User-level, Message Passing Interface in a Distributed Memory Environment." *Technical Report TM-12231, Oak Ridge National Laboratory*, February 1993.

[11] Message Passing Interface Forum. "MPI: A Message-passing Interface Standard." *Technical Report CS-94-230, Computer Science Dept., University of Tennessee, Knoxville, TN*, 1994.

[12] Message Passing Interface Forum. MPI: "A Message-passing Interface Standard." *International Journal of Supercomputer Applications, 8(3/4)*, 1994.

[13] P. Bridges, et.al. "User's Guide to MPICH, a Portable Implementation of MPI." *Available by ftp from info.mcs.anl.gov in /pub/mpi/guide.ps.Z*, November 1994.

[14] J. Zhang, "A Visual Performance Debugger for Concordia Parallel Programming Environment," *Master Degree Thesis at Concordia University*, 1999.

[15] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. "A high-performance, portable implementation of the MPI message passing interface standard," *Parallel Computing, 22(6):789-828, See: http://www.mcs.anl.gov/mpich/*. September 1996.

[16] Al Geist, Adam Begeulin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam. "PVM: Parallel Virtual Machine. A Users' Guide and Tutorial for Networked Parallel Computing," *MIT Press*, See also *http://www.epm.ornl.gov/pvm/*. 1994.

[17] Ohio Supercomputer Center. "MPI Primer/Developing with LAM." *Ohio Supercomputer Center. http://www.osc.edu/lam.html.* 1995.

[18] R. Alasdair, A. Bruce, James. G. Mills, and A. Gordon Smith. "CHIMP/MPI User Guide." *Ohio Supercomputer Center. http://www.osc.edu/chimp.html.* 1996.

[19] IBM, "IBM Parallel Environment for AIX: MPI Programming and Subroutine Reference," *Version 2, Release 2, Document Number GC23-3894-01, http://www.rs6000.ibm.com/resource/aix resource/sp books/pe/index.html.* November 1996.

[20] Hewlett Packard. "HP MPI User's Guide," *HP Part No. B6011-90001, Hewlett Packard. http://www.hp.com/wsg/ssa/mpi/mpihome.html.* November 1996.

[21] Digital Equipment Corporation. "Digital MPI User Guide," *Digital Equipment Corporation. http://www.digital.com/hpc/software/dmpi.html.* February 1997.

[22] R. Hempel, H. Ritzdorf, and F. Zimmermann. "Implementation of MPI on NEC's SX-4 multi-node architecture," *In Proceedings of the Euro PVM-MPI Workshop, http://www.ccrl-nece.technopark.gmd.de/ mpich/mpich nec.html.* 1997.

[23] Pallas. "Vampir and vampirtrace," *Pallas. http://www.pallas.de.* 1997.

[24] Lloyd J. Lewins. "MPI for the mercury race processor," *For more information, mail to llewins@ccgate.hac.com.* 1998.

[25] David Sitsky. "Implementing mpi using interrupts and remote copying on the ap1000/ap1000+," *In Proceedings of the Fourth Parallel Computing Workshop, London, England, http://cap.anu.edu.au/cap/projects/mpi/mpi.html.* October 1995.