1. Message Passing Fundamentals

Message Passing Fundamentals

As a programmer, you may find that you need to solve ever larger, more memory intensive problems, or simply solve problems with greater speed than is possible on a serial computer. You can turn to parallel programming and parallel computers to satisfy these needs. Using parallel programming methods on parallel computers gives you access to greater memory and Central Processing Unit (CPU) resources not available on serial computers. Hence, you are able to solve large problems that may not have been possible otherwise, as well as solve problems more quickly.

One of the basic methods of programming for parallel computing is the use of message passing libraries. These libraries manage transfer of data between instances of a parallel program running (usually) on multiple processors in a parallel computing architecture.

The topics to be discussed in this chapter are

- The basics of parallel computer architectures.
- The difference between domain and functional decomposition.
- The difference between data parallel and message passing models.
- A brief survey of important parallel programming issues.

1.1. Parallel Architectures

Parallel Architectures

Parallel computers have two basic architectures: **distributed memory** and **shared memory**.

Distributed memory parallel computers are essentially a collection of serial computers (nodes) working together to solve a problem. Each node has rapid access to its own local memory and access to the memory of other nodes via some sort of communications network, usually a proprietary high-speed communications network. Data are exchanged between nodes as messages over the network.

In a **shared memory** computer, multiple processor units share access to a global memory space via a high-speed memory bus. This global memory space allows the processors to efficiently exchange or share access to data. Typically, the number of processors used in shared memory architectures is limited to only a handful (2 - 16) of processors. This is because the amount of data that can be processed is limited by the bandwidth of the memory bus connecting the processors.

The latest generation of parallel computers now uses a mixed shared/distributed memory architecture. Each node consists of a group of 2 to 16 processors connected via local, shared memory and the multiprocessor nodes are, in turn, connected via a high-speed communications fabric.

1.2. Problem Decomposition

Problem Decomposition

The first step in designing a parallel algorithm is to decompose the problem into smaller problems. Then, the smaller problems are assigned to processors to work on simultaneously. Roughly speaking, there are two kinds of decompositions.

1. Domain decomposition
2. Functional decomposition

These are discussed in the following two sections.

1.2.1. Domain Decomposition

Domain Decomposition

In domain decomposition or "data parallelism", data are divided into pieces of approximately the same size and then mapped to different processors. Each processor then works only on the portion of the data that is assigned to it. Of course, the processes may need to communicate periodically in order to exchange data.

Data parallelism provides the advantage of maintaining a single flow of control. A data parallel algorithm consists of a sequence of elementary instructions applied to the data: an instruction is initiated only if the previous instruction is ended. Single-Program-Multiple-Data (SPMD) follows this model where the code is identical on all processors.

Such strategies are commonly employed in finite differencing algorithms where processors can operate independently on large portions of data, communicating only the much smaller shared border data at each iteration. An example of using data parallelism to solve the Poisson equation is provided.

1.2.2. Functional Decomposition

Functional Decomposition

Frequently, the domain decomposition strategy turns out **not** to be the most efficient algorithm for a parallel program. This is the case when the pieces of data assigned to the different processes require greatly different lengths of time to process. The performance of the code is then limited by the speed of the slowest process. The remaining idle processes do no useful work. In this case, functional decomposition or "task parallelism" makes more sense than domain decomposition. In task parallelism, the problem is decomposed into a large number of smaller tasks and then, the tasks are assigned to the processors as they become available. Processors that finish quickly are simply assigned more work.

Task parallelism is implemented in a client-server paradigm. The tasks are allocated to a group of slave processes by a master process that may also perform some of the tasks. The client-server paradigm can be implemented at virtually any level in a program. For example, if you simply wish to run a program with multiple inputs, a parallel client-server implementation might just run multiple copies of the code serially with the server assigning the different inputs to each client process. As each processor finishes its task, it is assigned a new input. Alternately, task parallelism can be implemented at a deeper level within the code. Later in this section, you will see a client-server implementation of a matrix-vector multiplication.

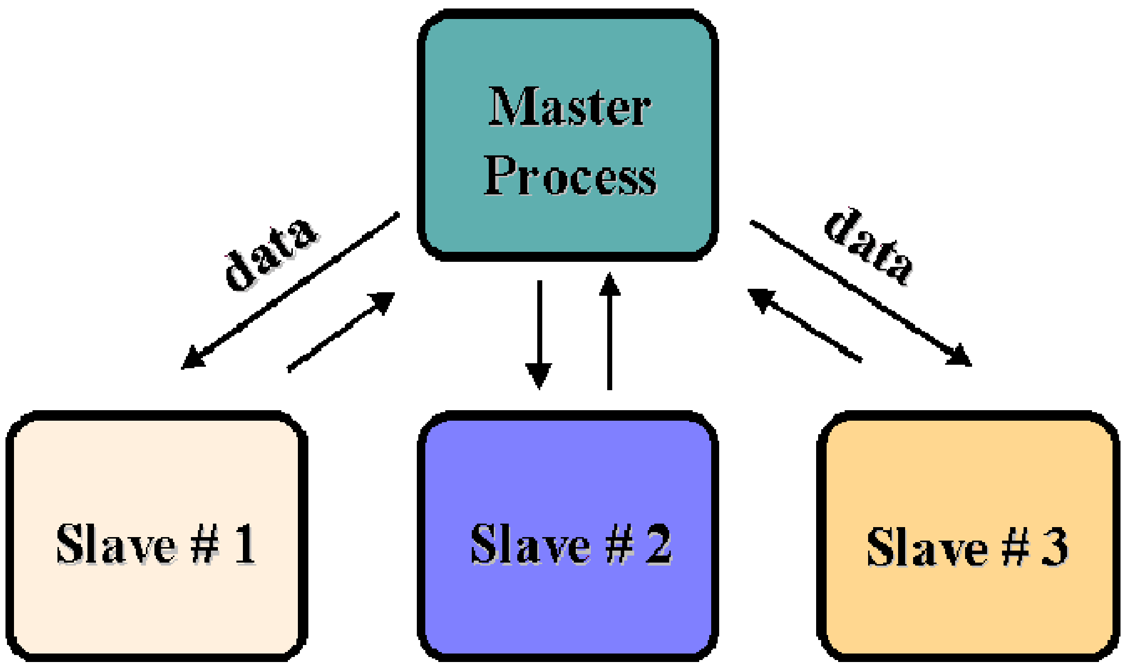


Figure 1.2 The client-server paradigm.

1.3. Data Parallel and Message Passing Models

Data Parallel and Message Passing Models

Historically, there have been two approaches to writing parallel programs. They are

1. use of a **directives-based data-parallel language**, and
2. explicit **message passing** via library calls from standard programming languages.

In a **directives-based data-parallel language**, such as High Performance Fortran (HPF) or OpenMP, a serial code is made parallel by adding directives (which appear as comments in the serial code) that tell the compiler how to distribute data and work across the processors. The details of how data distribution, computation, and communications are to be done are left to the compiler. Data parallel languages are usually implemented on shared memory architectures because the global memory space greatly simplifies the writing of compilers.

In the **message passing approach**, it is left up to the programmer to explicitly divide data and work across the processors as well as manage the communications among them. This approach is very flexible.

1.4. Parallel Programming Issues

Parallel Programming Issues

The main goal of writing a parallel program is to get better performance over the serial version. With this in mind, there are several issues that you need to consider when designing your parallel code to obtain the best performance possible within the constraints of the problem being solved. These issues are

- load balancing
- minimizing communication
- overlapping communication and computation

Each of these issues is discussed in the following sections.

1.4.1. Load Balancing

Load Balancing

Load balancing is the task of equally dividing work among the available processes. This can be easy to do when the same operations are being performed by all the processes (on different pieces of data). It is not trivial when the processing time depends upon the data values being worked on. When there are large variations in processing time, you may be required to adopt a different method for solving the problem.

1.4.2. Minimizing Communication

Minimizing Communication

Total execution time is a major concern in parallel programming because it is an essential component for comparing and improving all programs. Three components make up execution time:

1. Computation time
2. Idle time
3. Communication time

Computation time is the time spent performing computations on the data. Ideally, you would expect that if you had N processors working on a problem, you should be able to finish the job in 1/Nth the time of the serial job. This would be the case if all the processors' time was spent in computation.

Idle time is the time a process spends waiting for data from other processors. During this time, the processors do no useful work. An example of this is the ongoing problem of dealing with with input and output (I/O) in parallel programs. Many message passing libraries do not address parallel I/O, leaving all the work to one process while all other processes are in an idle state.

Finally, **communication time** is the time it takes for processes to send and receive messages. The cost of communication in the execution time can be measured in terms of latency and bandwidth. Latency is the time it takes to set up the envelope for communication, where bandwidth is the actual speed of transmission, or bits per unit time. Serial programs do not use interprocess communication. Therefore, you must minimize this use of time to get the best performance improvements.

1.4.3. Overlapping Communication and Computation

Overlapping Communication and Computation

There are several ways to minimize idle time within processes, and one example is overlapping communication and computation. This involves occupying a process with one or more new tasks while it waits for communication to finish so it can proceed on another task. Careful use of nonblocking communication and data unspecific computation make this possible. It is very difficult in practice to interleave communication with computation.

1.5. Self Test

Message Passing Fundamentals Self Test

Now that you've finished this chapter, test yourself on what you've learned by taking the Self Test provided. Simply click on the Self Test link in the ACTION MENU above to get started.

1.6. Course Problem

Chapter 1 Course Problem

At the end of each chapter of this course, you will be given the opportunity to work on a programming exercise related to the material presented. This exercise, called the "Course Problem", will get increasingly more sophisticated as the chapters progress. As you learn more of the complexities of MPI programming, you will see the initial simple, serial program grow into a parallel program containing most of MPI's salient features.

Description

The initial problem implements a parallel search of an extremely large (several thousand elements) integer array. The program finds all occurrences of a certain integer, called the target, and writes all the array indices where the target was found to an output file. In addition, the program reads both the target value and all the array elements from an input file.

Exercise

This first chapter provided an introduction to the concepts of parallel programming. Using these concepts, write a description of a parallel approach to solving the Course Problem described above. (No coding is required for this exercise.)

Solution

When you think you have described your approach adequately, view the solution description.

2. Getting Started with MPI

Getting Started with MPI

This chapter will familiarize you with some basic concepts of MPI programming, including the basic structure of messages and the main modes of communication.

The topics that will be discussed are

- The basic message passing model
- What is MPI?
- The goals and scope of MPI
- A first program: Hello World!

- Point-to-point communications and messages
- Blocking and nonblocking communications
- Collective communications

2.1. The Message Passing Model

The Message Passing Model

MPI is intended as a standard implementation of the "message passing" model of parallel computing.

- A parallel computation consists of a number of **processes**, each working on some local data. Each process has purely local variables, and there is no mechanism for any process to **directly** access the memory of another.
- Sharing of data between processes takes place by message passing, that is, by explicitly sending and receiving data between processes.

Note that the model involves **processes**, which need not, in principle, be running on different **processors**. In this course, it is generally assumed that different processes are running on different processors and the terms "processes" and "processors" are used interchangeably (e.g., by speaking of processors communicating with one another).

A primary reason for the usefulness of this model is that it is extremely general. Essentially, any type of parallel computation can be cast in the message passing form. In addition, this model

- can be implemented on a wide variety of platforms, from shared-memory multiprocessors to networks of workstations and even single-processor machines.
- generally allows more control over data location and flow within a parallel application than in, for example, the shared memory model. Thus programs can often achieve higher performance using explicit message passing. Indeed, performance is a primary reason why message passing is unlikely to ever disappear from the parallel programming world.

2.2. What is MPI?

What is MPI?

MPI stands for "Message Passing Interface". It is a library of functions (in C) or subroutines (in Fortran) that you insert into source code to perform data communication between processes.

MPI was developed over two years of discussions led by the MPI Forum, a group of roughly sixty people representing some forty organizations.

The MPI-1 standard was defined in Spring of 1994.

- This standard specifies the names, calling sequences, and results of subroutines and functions to be called from Fortran 77 and C, respectively. All implementations of MPI must conform to these rules, thus ensuring portability. MPI programs should compile and run on any platform that supports the MPI standard.
- The detailed implementation of the library is left to individual vendors, who are thus free to produce optimized versions for their machines.
- Implementations of the MPI-1 standard are available for a wide variety of platforms.

An MPI-2 standard has also been defined. It provides for additional features not present in MPI-1, including tools for parallel I/O, C++ and Fortran 90 bindings, and dynamic process management. At present, some MPI implementations include portions of the MPI-2 standard but the full MPI-2 is not yet available.

This course covers MPI-1 except for Chapter 9 - Parallel I/O.

2.3. Goals of MPI

Goals of MPI

The primary goals addressed by MPI are to

- Provide source code portability. MPI programs should compile and run as-is on any platform.
- Allow efficient implementations across a range of architectures.

MPI also offers

- A great deal of functionality, including a number of different types of communication, special routines for common "collective" operations, and the ability to handle user-defined data types and topologies.
- Support for heterogeneous parallel architectures.

Some things that are explicitly outside the scope of MPI-1 are

- The precise mechanism for launching an MPI program. In general, this is platform-dependent and you will need to consult your local documentation to find out how to do this.
- Dynamic process management, that is, changing the number of processes while the code is running.
- Debugging

- Parallel I/O

Several of these issues are addressed in MPI-2.

2.4. Why (Not) Use MPI?

Why (Not) Use MPI?

You should use MPI when you need to

- Write **portable** parallel code.
- Achieve high performance in parallel programming, e.g. when writing parallel libraries.
- Handle a problem that involves irregular or dynamic data relationships that do not fit well into the "data-parallel" model.

You should not use MPI when you

- Can achieve sufficient performance and portability using a data-parallel (e.g., High-Performance Fortran) or shared-memory approach (e.g., OpenMP, or proprietary directive-based paradigms).
- Can use a pre-existing library of parallel routines (which may themselves be written using MPI). For an introduction to using parallel numerical libraries, see Chapter 10 - Parallel Mathematical Libraries.
- Don't need parallelism at all!

2.5. Basic Features of Message Passing Programs

Basic Features of Message Passing Programs

Message passing programs consist of multiple instances of a serial program that communicate by library calls. These calls may be roughly divided into four classes:

1. Calls used to initialize, manage, and finally terminate communications.
2. Calls used to communicate between pairs of processors.
3. Calls that perform communications operations among groups of processors.
4. Calls used to create arbitrary data types.

The first class of calls consists of calls for starting communications, identifying the number of processors being used, creating subgroups of processors, and identifying which processor is running a particular instance of a program.

The second class of calls, called point-to-point communications operations, consists of different types of send and receive operations.

The third class of calls is the collective operations that provide synchronization or certain types of well-defined communications operations among groups of processes and calls that perform communication/calculation operations.

The final class of calls provides flexibility in dealing with complicated data structures.

The following sections of this chapter will focus primarily on the calls from the second and third classes: point-to-point communications and collective operations.

2.6. A First Program: Hello World!

A First Program: Hello World!

C:

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

void main (int argc, char *argv[]) {

  int err;
  err = MPI_Init(&argc, &argv);
  printf("Hello world!\n");
  err = MPI_Finalize();

}
```

Fortran:

```
PROGRAM hello

  INCLUDE 'mpif.h'
  INTEGER err

  CALL MPI_INIT(err)
  PRINT *, "Hello world!"
  CALL MPI_FINALIZE(err)

END
```

For the moment note from the example that

- MPI functions/subroutines have names that begin with **MPI_**.
- There is an MPI header file (mpi.h or mpif.h) containing definitions and function prototypes that is imported via an "include" statement.

- MPI routines return an error code indicating whether or not the routine ran successfully. A fuller discussion of this is given in Section 3.4. - MPI Routines and Return Values
- Each process executes a copy of the entire code. Thus, when run on four processors, the output of this program is

```
Hello world!
Hello world!
Hello world!
Hello world!
```

- However, different processors can be made to do different things using program branches, e.g.

```
if (I am processor 1)
   ...do something...
if (I am processor 2)
   ...do something else...
```

2.7. Point-to-Point Communications and Messages

Point-to-Point Communications and Messages

The elementary communication operation in MPI is "point-to-point" communication, that is, direct communication between two processors, one of which **sends** and the other **receives**.

Point-to-point communication in MPI is "two-sided", meaning that both an explicit send and an explicit receive are required. Data are not transferred without the participation of both processors.

In a generic send or receive, a **message** consisting of some block of data is transferred between processors. A message consists of an **envelope**, indicating the source and destination processors, and a **body**, containing the actual data to be sent.

MPI uses three pieces of information to characterize the message body in a flexible way:

1. **Buffer*** - the starting location in memory where outgoing data is to be found (for a send) or incoming data is to be stored (for a receive).
2. **Datatype** - the type of data to be sent. In the simplest cases this is an elementary type such as float/REAL, int/INTEGER, etc. In more advanced

applications this can be a user-defined type built from the basic types. These can be thought of as roughly analogous to C structures, and can contain data located anywhere, i.e., not necessarily in contiguous memory locations. This ability to make use of user-defined types allows complete flexibility in defining the message content. This is discussed further in Section 5 - Derived Datatypes.
3. **Count** - the number of items of type datatype to be sent.

Note that MPI standardizes the designation of the elementary types. This means that you don't have to explicitly worry about differences in how machines in heterogeneous environments represent them, e.g., differences in representation of floating-point numbers.

---

* In C, **buffer** is the actual address of the array element where the data transfer begins. In Fortran, it is just the name of the array element where the data transfer begins. (Fortran actually gets the address behind the scenes.)

2.8. Communication Modes and Completion Criteria

Communication Modes and Completion Criteria

MPI provides a great deal of flexibility in specifying how messages are to be sent. There are a variety of **communication modes** that define the procedure used to transmit the message, as well as a set of criteria for determining when the communication event (i.e., a particular send or receive) is **complete**. For example, a **synchronous send** is defined to be complete when receipt of the message at its destination has been acknowledged. A **buffered send**, however, is complete when the outgoing data has been copied to a (local) buffer; nothing is implied about the arrival of the message at its destination. In all cases, completion of a send implies that it is safe to overwrite the memory areas where the data were originally stored.

There are four communication modes available for sends:

- Standard
- Synchronous
- Buffered
- Ready

These are discussed in detail in Section 4 - Point-to-Point Communications.

For receives there is only a single communication mode. A receive is complete when the incoming data has actually arrived and is available for use.

2.9. Blocking and Nonblocking Communication

Blocking and Nonblocking Communication

In addition to the communication mode used, a send or receive may be **blocking** or **nonblocking**.

A **blocking** send or receive does not return from the subroutine call until the operation has actually completed. Thus it insures that the relevant completion criteria have been satisfied before the calling process is allowed to proceed.

- With a blocking send, for example, you are sure that the variables sent can safely be overwritten on the sending processor. With a blocking receive, you are sure that the data has actually arrived and is ready for use.

A **nonblocking** send or receive returns immediately, with no information about whether the completion criteria have been satisfied. This has the advantage that the processor is free to do other things while the communication proceeds "in the background." You can test later to see whether the operation has actually completed.

- For example, a nonblocking synchronous send returns immediately, although the send will not be complete until receipt of the message has been acknowledged. The sending processor can then do other useful work, testing later to see if the send is complete. Until it is complete, however, you can not assume that the message has been received or that the variables to be sent may be safely overwritten.

2.10. Collective Communications

Collective Communications

In addition to point-to-point communications between individual pairs of processors, MPI includes routines for performing **collective communications**. These routines allow larger groups of processors to communicate in various ways, for example, one-to-several or several-to-one.

The main advantages of using the collective communication routines over building the equivalent out of point-to-point communications are

- The possibility of error is significantly reduced. One line of code -- the call to the collective routine -- typically replaces several point-to-point calls.
- The source code is much more readable, thus simplifying code debugging and maintenance.
- Optimized forms of the collective routines are often faster than the equivalent operation expressed in terms of point-to-point routines.

Examples of collective communications include **broadcast operations**, **gather and scatter operations**, and **reduction operations**. These are briefly described in the following two sections.

2.10.1. Broadcast Operations

Broadcast Operations

The simplest kind of collective operation is the **broadcast**. In a broadcast operation a single process sends a copy of some data to all the other processes in a group. This operation is illustrated graphically in the figure below. Each row in the figure represents a different process. Each colored block in a column represents the location of a piece of the data. Blocks with the same color that are located on multiple processes contain copies of the same data.

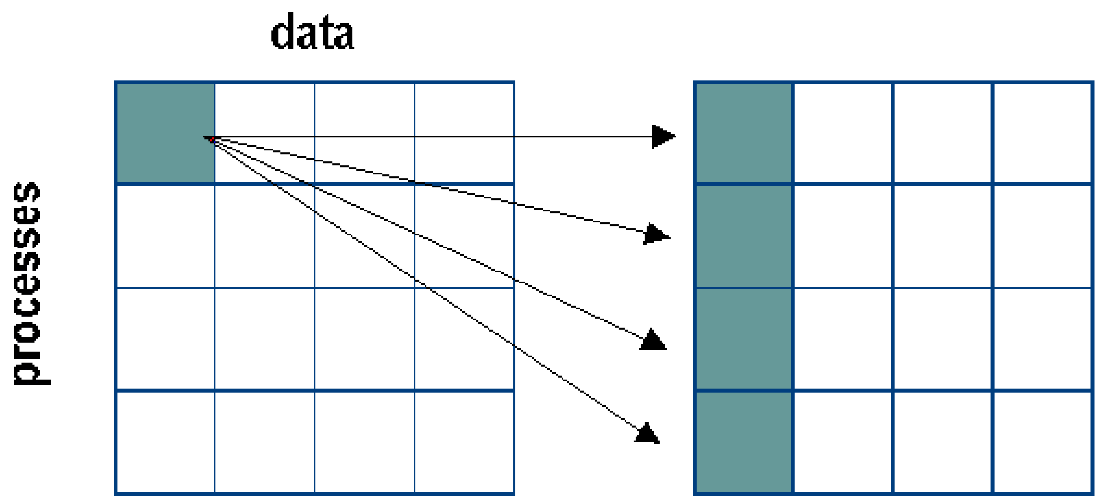


Figure 2.1. A broadcast operation

2.10.2. Gather and Scatter Operations

Gather and Scatter Operations

Perhaps the most important classes of collective operations are those that distribute data from one processor onto a group of processors or vice versa. These are called **scatter** and **gather** operations. MPI provides two kinds of scatter and gather operations, depending upon whether the data can be evenly distributed across processors. These scatter and gather operations are illustrated below.

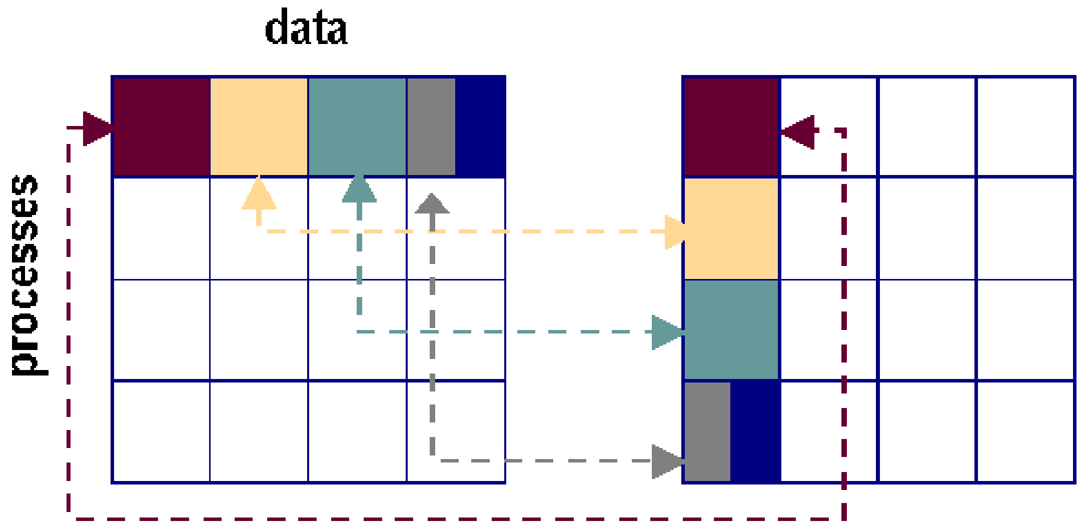


Figure 2.2 Scatter and gather operations

In a scatter operation, all of the data (an array of some type) are initially collected on a single processor (the left side of the figure). After the scatter operation, pieces of the data are distributed on different processors (the right side of the figure). The multicolored box reflects the possibility that the data may not be evenly divisible across the processors. The gather operation is the inverse operation to scatter: it collects pieces of the data that are distributed across a group of processors and reassembles them in the proper order on a single processor.

2.10.3. Reduction Operations

Reduction Operations

A **reduction** is a collective operation in which a single process (the **root** process) collects data from the other processes in a group and combines them into a single data item. For example, you might use a reduction to compute the sum of the elements of an array that is distributed over several processors. Operations other than arithmetic ones are also possible, for example, maximum and minimum, as well as various logical and bitwise operations.

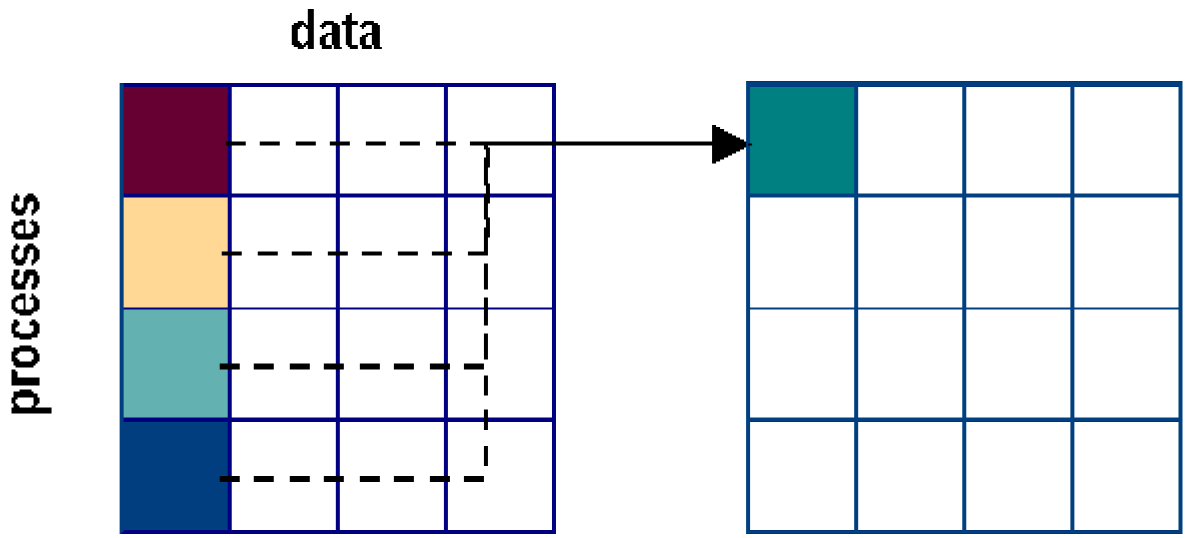


Figure 2.3 A reduction operation

In Figure 2.3, the data, which may be array or scalar values, are initially distributed across the processors. After the reduction operation, the reduced data (array or scalar) are located on the root processor.

2.11. Compiling and Running MPI Programs

Compiling and Running MPI Programs

The MPI standard does not specify how MPI programs are to be started. Thus, implementations vary from machine to machine.

When compiling an MPI program, it may be necessary to link against the MPI library. Typically, to do this, you would include the option

```
-lmpi
```

to the loader.

To run an MPI code, you commonly use a "wrapper" called mpirun or mpprun. The following command would run the executable a.out on four processors:

```
$ mpirun -np 4 a.out
```

For further detail on using MPI on some of the Alliance platforms, see

Ohio Supercomputer Center

- SGI Origin 2000
- Cray T3E
- Linux Cluster

National Center for Supercomputing Applications

- SGI Origin 2000/HP Exemplar
- NT Cluster

Boston University

- SGI Origin 2000/PCA

2.12. Self Test

Getting Started Self Test

Now that you've finished this chapter, test yourself on what you've learned by taking the Self Test provided. Simply click on the Self Test link in the ACTION MENU above to get started.

2.13. Course Problem

Chapter 2 Course Problem

In this chapter, you learned the terminology and concepts of MPI but not the syntax of the library routines. For this reason, the Course Problem is the same as the one described in Chapter 1 but with a different task to be performed.

Description

The initial problem implements a parallel search of an extremely large (several thousand elements) integer array. The program finds all occurrences of a certain integer, called the target, and writes all the array indices where the target was found to an output file. In addition, the program reads both the target value and all the array elements from an input file.

Exercise

Before writing a parallel version of a program, you must first write a serial version (that is, a version that runs on one processor). That is the task for this chapter. You can use either Fortran or C/C++ and should confirm that the program works by using a test input array.

Solution

After you have written your own code, view our version of the serial code for this problem.

---

3. MPI Program Structure

MPI Program Structure

This chapter introduces the basic structure of an MPI program. After sketching this structure using a generic pseudo-code, specific program elements are described in detail for both C and Fortran. These include

- Header files
- MPI naming conventions
- MPI routines and return values
- MPI handles
- MPI datatypes
- Initializing and terminating MPI
- Communicators
- Getting communicator information: rank and size

3.1. Generic MPI Program

A Generic MPI Program

All MPI programs have the following general structure:

```
include MPI header file
variable declarations
initialize the MPI environment

...do computation and MPI communication calls...

close MPI communications
```

The MPI header file contains MPI-specific definitions and function prototypes.

Then, following the variable declarations, each process calls an MPI routine that initializes the message passing environment. All calls to MPI communication routines must come after this initialization.

Finally, before the program ends, each process must call a routine that terminates MPI. No MPI routines may be called after the termination routine is called. Note that if any process does not reach this point during execution, the program will appear to hang.

3.2. MPI Header Files

MPI Header Files

MPI header files contain the prototypes for MPI functions/subroutines, as well as definitions of macros, special constants, and datatypes used by MPI. An appropriate "include" statement must appear in any source file that contains MPI function calls or constants.

C:

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

Fortran:

```
INCLUDE 'mpif.h'
```

3.3. MPI Naming Conventions

MPI Naming Conventions

The names of all MPI entities (routines, constants, types, etc.) begin with **MPI_** to avoid conflicts.

Fortran routine names are conventionally all upper case:

```
MPI_XXXXX(parameter, ... , IERR)
```
Example: `MPI_INIT(IERR)`.

C function names have a mixed case:

```
MPI_Xxxxx(parameter, ... )
```
Example: `MPI_Init(&argc, &argv)`.

The names of **MPI constants** are all upper case in both C and Fortran, for example,

```
MPI_COMM_WORLD, MPI_REAL, ...
```

In C, specially defined types correspond to many MPI entities. (In Fortran these are all integers.) Type names follow the C function naming convention above; for example,

```
MPI_Comm
```

is the type corresponding to an MPI "communicator".

3.4. MPI Routines and Return Values

MPI Routines and Return Values

MPI routines are implemented as **functions** in C and **subroutines** in Fortran. In either case generally an error code is returned, enabling you to test for the successful operation of the routine.

In C, MPI functions return an int, which indicates the exit status of the call.

```
   int err;
   ...
   err = MPI_Init(&argc, &argv);
   ...
```

In Fortran, MPI subroutines have an additional INTEGER argument -- always the last one in the argument list -- that contains the error status when the call returns.

```
   INTEGER IERR
   ...
   CALL MPI_INIT(IERR)
   ...
```

The error code returned is MPI_SUCCESS if the routine ran successfully (that is, the integer returned is equal to the pre-defined integer constant MPI_SUCCESS). Thus, you can test for successful operation with

C:

```
   if (err == MPI_SUCCESS) {

      ...routine ran correctly...

   }
```

Fortran:

```
   if (IERR.EQ.MPI_SUCCESS) THEN

       ...routine ran correctly...
```

```
END IF
```

If an error occurred, then the integer returned has an implementation-dependent value indicating the specific error.

3.5. MPI Handles

MPI Handles

MPI defines and maintains its own internal data structures related to communication, etc. You reference these data structures through **handles**. Handles are returned by various MPI calls and may be used as arguments in other MPI calls.

In C, handles are pointers to specially defined datatypes (created via the C typedef mechanism). Arrays are indexed starting at 0.

In Fortran, handles are integers (possibly arrays of integers) and arrays are indexed starting at 1.

Examples:

- MPI_SUCCESS - An integer in both C and Fortran. Used to test error codes.
- MPI_COMM_WORLD - In C, an object of type MPI_Comm (a "communicator"); in Fortran, an integer. In either case it represents a pre-defined communicator consisting of all processors.

Handles may be copied using the standard assignment operation in both C and Fortran.

3.6. MPI Datatypes

MPI Datatypes

MPI provides its own reference datatypes corresponding to the various elementary datatypes in C and Fortran.

- Variables are normally declared as C/Fortran types.
- MPI type names are used as arguments in MPI routines when a type is needed.

MPI hides the details of, e.g., the floating-point representation, which is an issue for the implementor.

MPI allows automatic translation between representations in a heterogeneous environment.

As a general rule, the MPI datatype given in a receive must match the MPI datatype specified in the send.

In addition, MPI allows you to define arbitrary data types built from the basic types. This is discussed in detail in Section 5 - Derived Datatypes.

3.7. Basic MPI Datatypes - C

Basic MPI Data Types - C

In C, the basic MPI datatypes and their corresponding C types are

MPI Datatype	C Type
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	(none)
MPI_PACKED	(none)

3.8. Basic MPI Datatypes - Fortran

Basic MPI Datatypes - Fortran

In Fortran, the basic MPI datatypes and their corresponding Fortran types are

MPI Datatype	Fortran Type
MPI_INTEGER	integer
MPI_REAL	real
MPI_DOUBLE_PRECISION	double precision
MPI_COMPLEX	complex
MPI_CHARACTER	character(1)
MPI_LOGICAL	logical
MPI_BYTE	(none)
MPI_PACKED	(none)

3.9. Special MPI Datatypes (C)

Special MPI Datatypes (C)

In C, MPI provides several special datatypes (structures). Examples include

- MPI_Comm - a communicator
- MPI_Status - a structure containing several pieces of status information for MPI calls
- MPI_Datatype

These are used in variable declarations, for example,

```
MPI_Comm some_comm;
```

declares a variable called some_comm, which is of type MPI_Comm (i.e. a communicator).

In Fortran, the corresponding types are all INTEGERs.

3.10. Initializing MPI

Initializing MPI

The first MPI routine called in any MPI program must be the initialization routine MPI_INIT. This routine establishes the MPI environment, returning an error code if there is a problem.

MPI_INIT may be called only <u>once</u> in any program!

C:

```
int err;
...
err = MPI_Init(&argc, &argv);
```

Note that the arguments to MPI_Init are the addresses of argc and argv, the variables that contain the command-line arguments for the program.

Fortran:

```
INTEGER IERR
...
CALL MPI_INIT(IERR)
```

3.11. Communicators

Communicators

A **communicator** is a handle representing a group of processors that can communicate with one another.

The **communicator name** is required as an argument to all point-to-point and collective operations.

- The communicator specified in the send and receive calls must agree for communication to take place.
- Processors can communicate only if they share a communicator.

There can be many communicators, and a given processor can be a member of a number of different communicators. Within each communicator, processors are numbered consecutively (starting at 0). This identifying number is known as the **rank** of the processor in that communicator.

- The rank is also used to specify the source and destination in send and receive calls.
- If a processor belongs to more than one communicator, its rank in each can (and usually will) be different!

MPI automatically provides a basic communicator called MPI_COMM_WORLD. It is the communicator consisting of all processors. Using MPI_COMM_WORLD, every processor can communicate with every other processor. You can define additional communicators consisting of subsets of the available processors.

3.11.1. Getting Communicator Information: Rank

Getting Communicator Information: Rank

A processor can determine its rank in a communicator with a call to MPI_COMM_RANK.

- Remember: ranks are consecutive and start with 0.
- A given processor may have different ranks in the various communicators to which it belongs.

C:

```
int MPI_Comm_rank(MPI_Comm comm, int *rank);
```

- The argument comm is a variable of type MPI_Comm, a communicator. For example, you could use MPI_COMM_WORLD here. Alternatively, you could pass the name of another communicator you have defined elsewhere. Such a variable would be declared as

  ```
  MPI_Comm some_comm;
  ```

- Note that the second argument is the *address* of the integer variable rank.

Fortran:

```
MPI_COMM_RANK(COMM, RANK, IERR)
```

- In this case the arguments COMM, RANK, and IERR are all of type INTEGER.

3.11.2. Getting Communicator Information: Size

Getting Communicator Information: Size

A processor can also determine the **size**, or number of processors, of any communicator to which it belongs with a call to MPI_COMM_SIZE.

C:

```
int MPI_Comm_size(MPI_Comm comm, int *size);
```

- The argument comm is of type MPI_Comm, a communicator.
- Note that the second argument is the address of the integer variable size.

Fortran:

```
MPI_COMM_SIZE(COMM, SIZE, IERR)
```

- The arguments COMM, SIZE, and IERR are all of type INTEGER.

3.12. Terminating MPI

Terminating MPI

The last MPI routine called should be MPI_FINALIZE which

- cleans up all MPI data structures, cancels operations that never completed, etc.
- **must** be called by all processes; if any one process does not reach this statement, the program will appear to hang.

Once MPI_FINALIZE has been called, no other MPI routines (including MPI_INIT) may be called.

C:

```
int err;
  ...
err =  MPI_Finalize();
```

Fortran:

```
INTEGER IERR
...
call MPI_FINALIZE(IERR)
```

3.13. Hello World! mk. 2 (C version)

Sample Program: Hello World! mk. 2

In this modified version of the "Hello World" program, each processor prints its rank as well as the total number of processors in the communicator MPI_COMM_WORLD.

C:

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

```
  void main (int argc, char *argv[]) {

    int myrank, size;

    MPI_Init(&argc, &argv);                 /* Initialize MPI       */
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank); /* Get my rank          */
    MPI_Comm_size(MPI_COMM_WORLD, &size);   /* Get the total
                                               number of processors */
    printf("Processor %d of %d: Hello World!\n", myrank, size);

    MPI_Finalize();                         /* Terminate MPI        */

   }
```

Notes:

- Makes use of the pre-defined communicator MPI_COMM_WORLD.
- Not testing for error status of routines!

3.14. Hello World! mk. 2 (Fortran version)

Sample Program: Hello World! mk. 2

Fortran:

```
  PROGRAM hello

    INCLUDE 'mpif.h'

    INTEGER myrank, size, ierr

C Initialize MPI:

    call MPI_INIT(ierr)

C Get my rank:

    call MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)

C Get the total number of processors:

    call MPI_COMM_SIZE(MPI_COMM_WORLD, size, ierr)

    PRINT *, "Processor", myrank, "of", size, ": Hello World!"

C Terminate MPI:

    call MPI_FINALIZE(ierr)
```

```
END
```

Notes:

- Makes use of the pre-defined communicator MPI_COMM_WORLD.
- Not testing for error status of routines!

3.15. Sample Program Output

Sample Program: Output

Running this code on four processors will produce a result like:

```
Processor 2 of 4: Hello World!
Processor 1 of 4: Hello World!
Processor 3 of 4: Hello World!
Processor 0 of 4: Hello World!
```

Each processor executes the same code, including probing for its rank and size and printing the string.

The order of the printed lines is essentially random!

- There is no intrinsic synchronization of operations on different processors.
- Each time the code is run, the order of the output lines may change.

3.16. Self Test

Program Structure Self Test

Now that you've finished this chapter, test yourself on what you've learned by taking the Self Test provided. Simply click on the Self Test link in the ACTION MENU above to get started.

3.17. Course Problem

Chapter 3 Course Problem

In this chapter, you learned the overall structure of an MPI program and a set of ubiquitous MPI routines for initializing and terminating the MPI library. The critical routine each processor can call to get its identifying rank number was also discussed.

You will use these routines in the exercise described below. Again, the problem description is the same as the one given in Chapter 1.

Description

The initial problem implements a parallel search of an extremely large (several thousand elements) integer array. The program finds all occurrences of a certain integer, called the target, and writes all the array indices where the target was found to an output file. In addition, the program reads both the target value and all the array elements from an input file.

Exercise

You now have enough knowledge to write pseudo-code for the parallel search algorithm introduced in Chapter 1. In the pseudo-code, you should correctly initialize MPI, have each processor determine **and use** its rank, and terminate MPI. By tradition, the Master processor has rank 0. Assume in your pseudo-code that the real code will be run on 4 processors.

Solution

When you have finished writing the code for this exercise, view our version of the Parallel Pseudo-Code.

---

4. Point-to-Point Communication

Point-to-Point Communication

Point-to-point communication is the fundamental communication facility provided by the MPI library.

Point-to-point communication is conceptually simple: one process sends a message and another process receives it. However, it is less simple in practice. For example, a process may have many messages waiting to be received. In that case, a crucial issue is how MPI and the receiving process determine what message to receive.

Another issue is whether send and receive routines initiate communication operations and return immediately, or wait for the initiated communication operation to complete before returning. The underlying communication operations are the same in both cases, but the programming interface is very different.

The topics to be discussed in this chapter are:

- Fundamentals of point-to-point communication
- Blocking send and receive

- Nonblocking send and receive
- Send modes

4.1. Fundamentals

Fundamentals

The issues discussed in the next three sections -- Source and Destination, Messages, and Sending and Receiving Messages -- are fundamental to point-to-point communication in MPI. These apply to all versions of send and receive, both blocking and nonblocking, and to all send modes.

4.1.1. Source and Destination

Source and Destination

The point-to-point communication facilities discussed here are two-sided and require active participation from the processes on both sides. One process (the source) sends, and another process (the destination) receives.

In general, the source and destination processes operate asynchronously. Even the sending and receiving of a single message is typically not synchronized. The source process may complete sending a message long before the destination process gets around to receiving it, and the destination process may initiate receiving a message that has not yet been sent.

Because sending and receiving are typically not synchronized, processes often have one or more messages that have been sent but not yet received. These sent, but not yet received messages are called **pending** messages. It is an important feature of MPI that pending messages are **not** maintained in a simple FIFO queue. Instead, each pending message has several attributes and the destination process (the receiving process) can use the attributes to determine which message to receive.

4.1.2. Messages

Messages

Messages consist of 2 parts: the **envelope** and the **message body**.

The **envelope** of an MPI message is analogous to the paper envelope around a letter mailed at the post office. The envelope of a posted letter typically has the destination address, the return address, and any other information needed to transmit and deliver the letter, such as class of service (airmail, for example).

The envelope of an MPI message has 4 parts:

1. **source** - the sending process;
2. **destination** - the receiving process;
3. **communicator** - specifies a group of processes to which both source and destination belong (See Section 3.11 - Communicators.);
4. **tag** - used to classify messages.

The tag field is required, but its use is left up to the program. A pair of communicating processes can use tag values to distinguish classes of messages. For example, one tag value can be used for messages containing data and another tag value for messages containing status information.

The **message body** was previously described in Section 2.7 - Point-to-Point Communications and Messages. It has 3 parts:

1. **buffer** - the message data;
2. **datatype** - the type of the message data;
3. **count** - the number of items of type datatype in buffer.

Think of the buffer as an array; the dimension is given by count, and the type of the array elements is given by datatype. Using datatypes and counts, rather than bytes and bytecounts, allows structured data and noncontiguous data to be handled smoothly. It also allows transparent support of communication between heterogeneous hosts.

4.1.3. Sending and Receiving Messages

Sending and Receiving Messages

Sending messages is straightforward. The source (the identity of the sender) is determined implicitly, but the rest of the message (envelope and body) is given explicitly by the sending process.

Receiving messages is not quite so simple. As mentioned in Section 4.1.1 - Source and Destination, a process may have several pending messages.

To receive a message, a process specifies a message envelope that MPI compares to the envelopes of pending messages. If there is a match, a message is received. Otherwise, the receive operation cannot be completed until a matching message is sent.

In addition, the process receiving a message must provide storage into which the body of the message can be copied. The receiving process must be careful to provide enough storage for the entire message.

4.2. Blocking Send and Receive

Blocking Send and Receive

The two functions, MPI_SEND and MPI_RECV, are the basic point-to-point communication routines in MPI. Their calling sequences are presented and discussed in the following sections.

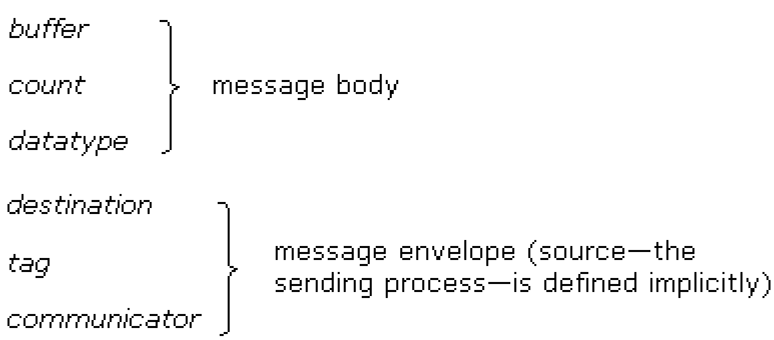
Both functions block the calling process until the communication operation is completed. Blocking creates the possibility of deadlock, a key issue that is explored by way of simple examples. In addition, the meaning of completion is discussed.

The nonblocking analogues of MPI_SEND and MPI_RECV are presented in Section 4.3 - Nonblocking Sends and Receives.

4.2.1. Sending a Message: MPI_SEND

Sending a Message: MPI_SEND

MPI_SEND takes the following arguments:



The message body contains the data to be sent: *count* items of type *datatype*. The message envelope tells where to send it. In addition, an error code is returned. Both C and Fortran bindings are shown below.

C:

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

- All arguments are input arguments.
- An error code is returned by the function.

Fortran:

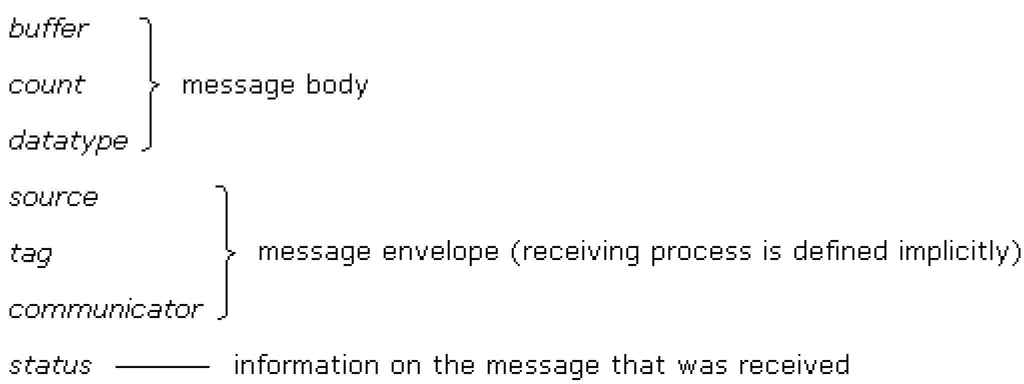
```
MPI_SEND(BUF, COUNT, DTYPE, DEST, TAG, COMM, IERR)
```

- The input argument BUF is an array; its type should match the type given in DTYPE.
- The input arguments COUNT, DTYPE, DEST, TAG, COMM are of type INTEGER.
- The output argument IERR is of type INTEGER; it contains an error code when MPI_SEND returns.

4.2.2. Receiving a Message: MPI_RECV

Receiving a Message: MPI_RECV

MPI_RECV takes a set of arguments similar to MPI_SEND, but several of the arguments are used in a different way.



The message envelope arguments determine what messages can be received by the call. The *source*, *tag*, and *communicator* arguments must match those of a pending message in order for the message to be received. (See Section 4.1.3 - Sending and Receiving Messages).

Wildcard values may be used for the source (accept a message from any process) and the tag (accept a message with any tag value). If wildcards are not used, the call can accept messages from only the specified sending process, and with only the specified tag value. Communicator wildcards are not available.

The message body arguments specify where the arriving data are to be stored, what type it is assumed to be, and how much of it the receiving process is prepared to accept. If the received message has more data than the receiving process is prepared to accept, it is an error.

In general, the sender and receiver must agree about the message datatype, and it is the programmer's responsibility to guarantee that agreement. If the sender and receiver use incompatible message datatypes, the results are undefined.

The status argument returns information about the message that was received. The source and tag of the received message are available this way (needed if wildcards were used); also available is the actual count of data received.

In addition, an error code is returned.

Both C and Fortran bindings are shown below.

C:

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

- buf and status are output arguments; the rest are inputs.
- An error code is returned by the function.

Fortran:

```
MPI_RECV(BUF, COUNT, DTYPE, SOURCE, TAG, COMM, STATUS, IERR)
```

- The output argument BUF is an array; its type should match the type in DTYPE.
- The input arguments COUNT, DTYPE, SOURCE, TAG, COMM are of type INTEGER.
- The output argument STATUS is an INTEGER array with MPI_STATUS_SIZE elements.
- The output argument IERR is of type INTEGER; it contains an error code when MPI_RECV returns.

Notes:

- A maximum of COUNT items of type DTYPE are accepted; if the message contains more, it is an error.
- The sending and receiving processes must agree on the datatype; if they disagree, results are undefined (MPI does not check).
- When this routine returns, the received message data have been copied into the buffer; and the tag, source, and actual count of data received are available via the status argument.

4.2.3. Example: Send and Receive

Example: Send and Receive

In this program, process 0 sends a message to process 1, and process 1 receives it. Note the use of myrank in a conditional to limit execution of code to a particular process.

Fortran:

```
  PROGRAM simple_send_and_receive

    INCLUDE 'mpif.h'
    INTEGER myrank, ierr, status(MPI_STATUS_SIZE)
    REAL a(100)

C Initialize MPI:

    call MPI_INIT(ierr)

C Get my rank:

    call MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)

C Process 0 sends, process 1 receives:

    if( myrank.eq.0 )then
       call MPI_SEND( a, 100, MPI_REAL, 1, 17, MPI_COMM_WORLD, ierr)

    else if ( myrank.eq.1 )then
       call MPI_RECV( a, 100, MPI_REAL, 0, 17, MPI_COMM_WORLD, status,
ierr )
    endif


C Terminate MPI:

    call MPI_FINALIZE(ierr)

  END
```

C:

```
  /* simple send and receive */
  #include <stdio.h>
  #include <mpi.h>

  void main (int argc, char **argv) {

    int myrank;
    MPI_Status status;
    double a[100];

    MPI_Init(&argc, &argv);  /* Initialize MPI */
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank); /* Get rank */
    if( myrank == 0 )        /* Send a message */
      MPI_Send( a, 100, MPI_DOUBLE, 1, 17, MPI_COMM_WORLD );
```

```
    else if( myrank == 1 )    /* Receive a message */
      MPI_Recv( a, 100, MPI_DOUBLE, 0, 17, MPI_COMM_WORLD, &status );

    MPI_Finalize();           /* Terminate MPI */

  }
```

4.2.4. What Happens at Runtime

What Happens at Runtime

It is useful to keep in mind the following model for the runtime behavior of MPI_SEND. According to the model, when a message is sent using MPI_SEND one of two things may happen:

1. The message may be copied into an MPI internal buffer and transferred to its destination later, in the background, or
2. The message may be left where it is, in the program's variables, until the destination process is ready to receive it. At that time, the message is transferred to its destination.

The first option allows the sending process to move on to other things after the copy is completed. The second option minimizes copying and memory use, but may result in extra delay to the sending process. The delay can be significant.

Surprisingly, in 1., a call to MPI_SEND may return before any non-local action has been taken or even begun, i.e., before anything has happened that might naively be associated with sending a message. In 2., a synchronization between sender and receiver is implied.

To summarize, according to the model sketched above, when a message is sent using MPI_SEND, the message is either buffered immediately and delivered later asynchronously, or the sending and receiving processes synchronize.

4.2.5. Blocking and Completion

Blocking and Completion

Both MPI_SEND and MPI_RECV block the calling process. Neither returns until the communication operation it invoked is completed.

The meaning of completion for a call to MPI_RECV is simple and intuitive -- a matching message has arrived, and the message’s data have been copied into the output arguments of the call. In other words, the variables passed to MPI_RECV contain a message and are ready to be used.

For MPI_SEND, the meaning of completion is simple but not as intuitive. A call to MPI_SEND is completed when the message specified in the call has been handed off to MPI. In other words, the variables passed to MPI_SEND can now be overwritten and reused. Recall from the previous section that one of two things may have happened: either MPI copied the message into an internal buffer for later, asynchronous delivery; or else MPI waited for the destination process to receive the message. Note that if MPI copied the message into an internal buffer, then the call to MPI_SEND may be officially completed, even though the message has not yet left the sending process.

If a message passed to MPI_SEND is larger than MPI's available internal buffer, then buffering cannot be used. In this case, the sending process must block until the destination process begins to receive the message, or until more buffer is available. In general, messages that are copied into MPI internal buffer will occupy buffer space until the destination process begins to receive the message.

Note that a call to MPI_RECV matches a pending message if it matches the pending message's envelope (source, tag, communicator). Datatype matching is also required for correct execution but MPI does not check for it. Instead, it is the obligation of the programmer to guarantee datatype matching.

4.2.6. Deadlock

Deadlock

Deadlock occurs when 2 (or more) processes are blocked and each is waiting for the other to make progress. Neither process makes progress because each depends on the other to make progress first. The program shown below is an example -- it fails to run to completion because processes 0 and 1 deadlock.

In the program, process 0 attempts to exchange messages with process 1. Each process begins by attempting to receive a message sent by the other; each process blocks pending receipt. Process 0 cannot proceed until process 1 sends a message; process 1 cannot proceed until process 0 sends a message.

The program is erroneous and deadlocks. No messages are ever sent, and no messages are ever received.

C Example:

```
/* simple deadlock */
#include <stdio.h>
#include <mpi.h>

void main (int argc, char **argv) {

  int myrank;
```

```
    MPI_Status status;
    double a[100], b[100];

    MPI_Init(&argc, &argv);  /* Initialize MPI */
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank); /* Get rank */
    if( myrank == 0 ) {
      /* Receive, then send a message */
      MPI_Recv( b, 100, MPI_DOUBLE, 1, 19, MPI_COMM_WORLD, &status );
      MPI_Send( a, 100, MPI_DOUBLE, 1, 17, MPI_COMM_WORLD );
    }
    else if( myrank == 1 ) {
      /* Receive, then send a message */
      MPI_Recv( b, 100, MPI_DOUBLE, 0, 17, MPI_COMM_WORLD, &status );
      MPI_Send( a, 100, MPI_DOUBLE, 0, 19, MPI_COMM_WORLD );
    }

    MPI_Finalize();          /* Terminate MPI */

  }
```

Fortran Example:

```
  PROGRAM simple_deadlock

    INCLUDE 'mpif.h'
    INTEGER myrank, ierr, status(MPI_STATUS_SIZE)
    REAL a(100), b(100)

C Initialize MPI:

    call MPI_INIT(ierr)

C Get my rank:

    call MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)

C Process 0 receives and sends; same for process 1

    if( myrank.eq.0 )then
       call MPI_RECV( b, 100, MPI_REAL, 1, 19, MPI_COMM_WORLD, status,
ierr )
       call MPI_SEND( a, 100, MPI_REAL, 1, 17, MPI_COMM_WORLD, ierr)

    else if ( myrank.eq.1 )then
       call MPI_RECV( b, 100, MPI_REAL, 0, 17, MPI_COMM_WORLD, status,
ierr )
       call MPI_SEND( a, 100, MPI_REAL, 0, 19, MPI_COMM_WORLD, ierr)

    endif


C Terminate MPI:
```

```
   call MPI_FINALIZE(ierr)

 END
```

4.2.6.1. Avoiding Deadlock

Avoiding Deadlock

In general, avoiding deadlock requires careful organization of the communication in a program. The programmer should be able to explain why the program does not (or does) deadlock.

The program shown below is similar to the program in the preceding section, but its communication is better organized and the program does not deadlock. Once again, process 0 attempts to exchange messages with process 1. Process 0 receives, then sends; process 1 sends, then receives. The protocol is safe. Barring system failures, this program always runs to completion.

Note that increasing array dimensions and message sizes have no effect on the safety of the protocol. The program still runs to completion. This is a useful property for application programs -- when the problem size is increased, the program still runs to completion.

C Example:

```
/* safe exchange */
#include <stdio.h>
#include <mpi.h>
void main (int argc, char **argv) {

  int myrank;
  MPI_Status status;
  double a[100], b[100];

  MPI_Init(&argc, &argv);  /* Initialize MPI */
  MPI_Comm_rank(MPI_COMM_WORLD, &myrank); /* Get rank */
  if( myrank == 0 ) {
    /* Receive a message, then send one */
    MPI_Recv( b, 100, MPI_DOUBLE, 1, 19, MPI_COMM_WORLD, &status );
    MPI_Send( a, 100, MPI_DOUBLE, 1, 17, MPI_COMM_WORLD );
  }
  else if( myrank == 1 ) {
    /* Send a message, then receive one */
    MPI_Send( a, 100, MPI_DOUBLE, 0, 19, MPI_COMM_WORLD );
    MPI_Recv( b, 100, MPI_DOUBLE, 0, 17, MPI_COMM_WORLD, &status );
```

```
    }

    MPI_Finalize();          /* Terminate MPI */

  }
```

Fortran Example:

```
   PROGRAM safe_exchange

   INCLUDE 'mpif.h'
   INTEGER myrank, ierr, status(MPI_STATUS_SIZE)
   REAL a(100), b(100)

C Initialize MPI:

   call MPI_INIT(ierr)

C Get my rank:

   call MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)

C Process 0 receives and sends; process 1 sends and receives

   if( myrank.eq.0 )then
      call MPI_RECV( b, 100, MPI_REAL, 1, 19, MPI_COMM_WORLD, status,
ierr )
      call MPI_SEND( a, 100, MPI_REAL, 1, 17, MPI_COMM_WORLD, ierr)

   else if ( myrank.eq.1 )then
      call MPI_SEND( a, 100, MPI_REAL, 0, 19, MPI_COMM_WORLD, ierr )
      call MPI_RECV( b, 100, MPI_REAL, 0, 17, MPI_COMM_WORLD, status,
ierr)

   endif


C Terminate MPI:

  call MPI_FINALIZE(ierr)

  END
```

4.2.6.2. Avoiding Deadlock (Sometimes but Not Always)

Avoiding Deadlock (Sometimes but Not Always)

The program shown below is similar to preceding examples. Again, process 0 attempts to exchange messages with process 1. This time, both processes send first, then receive. Success depends on the availability of buffering in MPI. There must be enough MPI internal buffer available to hold at least one of the messages in its entirety.

Under most MPI implementations, the program shown will run to completion. However, if the message sizes are increased, sooner or later the program will deadlock. This behavior is sometimes seen in computational codes -- a code will run to completion when given a small problem, but deadlock when given a large problem. This is inconvenient and undesirable. The inconvenience is increased when the original authors of the code are no longer available to maintain it.

In general, depending on MPI internal buffer to avoid deadlock makes a program less portable and less scalable. The best practice is to write programs that run to completion regardless of the availability of MPI internal buffer.

C Example:

```
/* depends on buffering */
#include <stdio.h>
#include <mpi.h>

void main (int argc, char **argv) {

int myrank;
MPI_Status status;
double a[100], b[100];

MPI_Init(&argc, &argv);  /* Initialize MPI */
MPI_Comm_rank(MPI_COMM_WORLD, &myrank); /* Get rank */
if( myrank == 0 ) {
  /* Send a message, then receive one */
  MPI_Send( a, 100, MPI_DOUBLE, 1, 17, MPI_COMM_WORLD );
  MPI_Recv( b, 100, MPI_DOUBLE, 1, 19, MPI_COMM_WORLD, &status );
}
else if( myrank == 1 ) {
  /* Send a message, then receive one */
  MPI_Send( a, 100, MPI_DOUBLE, 0, 19, MPI_COMM_WORLD );
  MPI_Recv( b, 100, MPI_DOUBLE, 0, 17, MPI_COMM_WORLD, &status );
}

MPI_Finalize();          /* Terminate MPI */

}
```

Fortran Example:

```
PROGRAM depends_on_buffering

INCLUDE 'mpif.h'
INTEGER myrank, ierr, status(MPI_STATUS_SIZE)
REAL a(100), b(100)
```

```
C Initialize MPI:

   call MPI_INIT(ierr)

C Get my rank:

   call MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)

C Process 0 sends and receives; same for process 1

   if( myrank.eq.0 )then
      call MPI_SEND( a, 100, MPI_REAL, 1, 17, MPI_COMM_WORLD, ierr)
      call MPI_RECV( b, 100, MPI_REAL, 1, 19, MPI_COMM_WORLD, status,
ierr )

   else if ( myrank.eq.1 )then
      call MPI_SEND( a, 100, MPI_REAL, 0, 19, MPI_COMM_WORLD, ierr )
      call MPI_RECV( b, 100, MPI_REAL, 0, 17, MPI_COMM_WORLD, status,
ierr)

   endif

C Terminate MPI:

   call MPI_FINALIZE(ierr)

   END
```

4.2.6.3. Probable Deadlock

Probable Deadlock

The only significant difference between the program shown below and the preceeding one is the size of the messages. This program will deadlock under the default configuration of nearly all available MPI implementations.

C Example:

```
   /* probable deadlock */
   #include <stdio.h>
   #include <mpi.h>

   void main (int argc, char **argv) {

   int myrank;
   MPI_Status status;
   #define N 100000000
   double a[N], b[N];

   MPI_Init(&argc, &argv);  /* Initialize MPI */
   MPI_Comm_rank(MPI_COMM_WORLD, &myrank); /* Get rank */
```

```
   if( myrank == 0 ) {
     /* Send a message, then receive one */
     MPI_Send( a, N, MPI_DOUBLE, 1, 17, MPI_COMM_WORLD );
     MPI_Recv( b, N, MPI_DOUBLE, 1, 19, MPI_COMM_WORLD, &status );
   }
   else if( myrank == 1 ) {
     /* Send a message, then receive one */
     MPI_Send( a, N, MPI_DOUBLE, 0, 19, MPI_COMM_WORLD );
     MPI_Recv( b, N, MPI_DOUBLE, 0, 17, MPI_COMM_WORLD, &status );
   }

   MPI_Finalize();          /* Terminate MPI */

   }
```

Fortran Example:

```
   PROGRAM probable_deadlock

   INCLUDE 'mpif.h'
   INTEGER myrank, ierr, status(MPI_STATUS_SIZE)
   INTEGER n
   PARAMETER (n=100000000)
   REAL a(n), b(n)

C Initialize MPI:

   call MPI_INIT(ierr)

C Get my rank:

   call MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)

C Process 0 sends, then receives; same for process 1

   if( myrank.eq.0 )then
      call MPI_SEND( a, n, MPI_REAL, 1, 17, MPI_COMM_WORLD, ierr)
      call MPI_RECV( b, n, MPI_REAL, 1, 19, MPI_COMM_WORLD, status,
ierr )

   else if ( myrank.eq.1 )then
      call MPI_SEND( a, n, MPI_REAL, 0, 19, MPI_COMM_WORLD, ierr )
      call MPI_RECV( b, n, MPI_REAL, 0, 17, MPI_COMM_WORLD, status,
ierr)
   endif

C Terminate MPI:

   call MPI_FINALIZE(ierr)

   END
```

4.3. Nonblocking Sends and Receives

Nonblocking Sends and Receives

Recall that both MPI_SEND and MPI_RECV block the calling process. Neither returns until the communication operation it invoked is completed. As discussed in Section 4.2.5 - Blocking and Completion, the requirement for the communication operation to complete can cause delays and even deadlock.

MPI provides another way to invoke send and receive operations. It is possible to separate the initiation of a send or receive operation from its completion by making two separate calls to MPI. The first call initiates the operation, and the second call completes it. Between the two calls, the program is free to do other things.

The underlying communication operations are the same whether they are invoked by a single call, or by two separate calls -- one to initiate the operation and another to complete it. The communication operations are the same, but the interface to the library is different.

4.3.1. Posting, Completion, and Request Handles

Posting, Completion, and Request Handles

The nonblocking interface to send and receive requires two calls per communication operation: one call to initiate the operation, and a second call to complete it. Initiating a send operation is called *posting a send*. Initiating a receive operation is called *posting a receive*.

Once a send or receive operation has been posted, MPI provides two distinct ways of completing it. A process can test to see if the operation has completed, without blocking on the completion. Alternately, a process can wait for the operation to complete.

After posting a send or receive with a call to a nonblocking routine, the posting process needs some way to refer to the posted operation. MPI uses request handles for this purpose (See Section 3.6 - MPI Handles). Nonblocking send and receive routines all return request handles, which are used to identify the operation posted by the call.

In summary, sends and receives may be posted (initiated) by calling nonblocking routines. Posted operations are identified by request handles. Using request handles, processes can check the status of posted operations or wait for their completion.

4.3.2. Posting Sends without Blocking

Posting Sends without Blocking

A process calls the routine MPI_ISEND to post (initiate) a send without blocking on completion of the send operation. The calling sequence is similar to the calling sequence for the blocking routine MPI_SEND but includes an additional output argument, a request handle. The request handle identifies the send operation that was posted. The request handle can be used to check the status of the posted send or to wait for its completion.

None of the arguments passed to MPI_ISEND should be read or written until the send operation it invokes is completed.

Nonblocking C and Fortran versions of the standard mode send are given below.

C:

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

- An error code is returned by the function.

Fortran:

```
MPI_ISEND(BUF, COUNT, DTYPE, DEST, TAG, COMM, REQ, IERR)
```

- The input argument BUF is an array; its type should match the type in DTYPE.
- The input arguments COUNT, DTYPE, DEST, TAG, COMM have type INTEGER.
- The output argument REQ has type INTEGER; it is used to identify a request handle.
- The output argument IERR has type INTEGER; it contains an error code when MPI_SEND returns.

Notes:

- The message body is specified by the first three arguments (BUF, COUNT and DTYPE in Fortran) and the message envelope by the second three (DEST, TAG and COMM in Fortran).
- The source of the message, the sending process, is determined implicitly.
- When this routine returns, a send has been posted (but not yet completed).
- Another call to MPI is required to complete the send operation posted by this routine.
- None of the arguments passed to MPI_ISEND should be read or written until the send operation is completed.

4.3.3. Posting Receives without Blocking

Posting Receives without Blocking

A process calls the routine MPI_IRECV to post (initiate) a receive without blocking on its completion. The calling sequence is similar to the calling sequence for the blocking routine MPI_RECV, but the status argument is replaced by a request handle; both are output arguments. The request handle identifies the receive operation that was posted and can be used to check the status of the posted receive or to wait for its completion.

None of the arguments passed to MPI_IRECV should be read or written until the receive operation it invokes is completed.

Nonblocking C and Fortran versions of the standard mode send are given below.

C:

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

- An error code is returned by the function.

Fortran:

```
MPI_IRECV(BUF, COUNT, DTYPE, SOURCE, TAG, COMM, REQUEST, IERR)
```

- The output argument BUF is an array; its type should match the type in DTYPE.
- The input arguments COUNT, DTYPE, SOURCE, TAG, COMM have type INTEGER.
- The output argument REQUEST has type INTEGER; it is used to identify a request handle.
- The output argument IERR has type INTEGER; it contains an error code when MPI_RECV returns.

Notes:

- The message body is specified by the first three arguments (BUF, COUNT, and DTYPE in Fortran) and the message envelope by the second three (DEST, TAG and COMM in Fortran).
- A maximum of count items of type DTYPE is accepted; if the message contains more, it is an error.
- The sending and receiving processes must agree on the datatype; if they disagree, it is an error.
- When this routine returns, the receive has been posted (initiated) but not yet completed.
- Another call to MPI is required to complete the receive operation posted by this routine.
- None of the arguments passed to MPI_IRECV should be read or written until the receive operation is completed.

4.3.4. Completion

Completion: Waiting and Testing

Posted sends and receives must be completed. If a send or receive is posted by a nonblocking routine, then its completion status can be checked by calling one of a family of completion routines. MPI provides both blocking and nonblocking completion routines. The blocking routines are MPI_WAIT and its variants. The nonblocking routines are MPI_TEST and its variants. These routines are discussed in the following two sections.

4.3.4.1. Completion: Waiting

Completion: Waiting

A process that has posted a send or receive by calling a nonblocking routine (for instance, MPI_ISEND or MPI_IRECV) can subsequently wait for the posted operation to complete by calling MPI_WAIT. The posted send or receive is identified by passing a request handle.

The arguments for the MPI_WAIT routine are:

```
request    a request handle (returned when the send or receive was
posted

status     for receive, information on the message received; for send,
may
           contain an error code
```

In addition, an error code is returned.

C and Fortran versions of MPI_WAIT are given below.

C:

```
int MPI_Wait( MPI_Request *request, MPI_Status *status );
```

- An error code is returned.

Fortran:

```
MPI_WAIT(REQUEST, STATUS, IERR )
```

- The in/out argument REQUEST has type INTEGER.

- The output argument STATUS is an INTEGER array with MPI_STATUS_SIZE elements.
- The output argument IERR has type INTEGER and contains an error code when the call returns.

Notes:

- The request argument is expected to identify a previously posted send or receive.
- MPI_WAIT returns when the send or receive identified by the request argument is complete.
- If the posted operation was a receive, then the source, tag, and actual count of data received are available via the status argument.
- If the posted operation was a send, the status argument may contain an error code for the send operation (different from the error code for the call to MPI_WAIT).

4.3.4.2. Completion: Testing

Completion: Testing

A process that has posted a send or receive by calling a nonblocking routine can subsequently test for the posted operation's completion by calling MPI_TEST. The posted send or receive is identified by passing a request handle.

The arguments for the MPI_TEST routine are:

```
request    a request handle (returned when the send or receive was
posted).
flag       true if the send or receive has completed.
status     undefined if flag equals false. Otherwise, like MPI_WAIT.
```

In addition, an error code is returned.

C and Fortran versions of MPI_TEST are given below.

C:

```
int MPI_Test( MPI_Request *request, int *flag, MPI_Status *status );
```

- An error code is returned.

Fortran:

```
MPI_TEST(REQUEST, FLAG, STATUS, IERR)
```

- The in/out argument REQUEST has type INTEGER.

- The output argument FLAG has type LOGICAL.
- The output argument STATUS is an INTEGER array with MPI_STATUS_SIZE elements.
- The output argument IERR has type INTEGER and contains an error code when the call returns.

Notes:

- The request argument is expected to identify a previously posted send or receive.
- MPI_TEST returns immediately.
- If the flag argument is **true**, then the posted operation is complete.
- If the flag argument is **true** and the posted operation was a receive, then the source, tag, and actual count of data received are available via the status argument.
- If the flag argument is **true** and the posted operation was a send, then the status argument may contain an error code for the send operation (not for MPI_TEST).

4.3.5. Advantages and Disadvantages

Nonblocking Sends and Receives: Advantages and Disadvantages

Selective use of nonblocking routines makes it much easier to write deadlock-free code. This is a big advantage because it is easy to unintentionally write deadlock into programs.

On systems where latencies are large, posting receives early is often an effective, simple strategy for masking communication overhead. Latencies tend to be large on physically distributed collections of hosts (for example, clusters of workstations) and relatively small on shared memory multiprocessors. In general, masking communication overhead requires careful attention to algorithms and code structure.

On the downside, using nonblocking send and receive routines may increase code complexity, which can make code harder to debug and harder to maintain.

4.3.6. Send/Receive Example

Send/Receive Example

This program is a revision of the earlier example given in Section 4.2.3. This version runs to completion.

Process 0 attempts to exchange messages with process 1. Each process begins by posting a receive for a message from the other. Then, each process blocks on a send. Finally, each process waits for its previously posted receive to complete.

Each process completes its send because the other process has posted a matching receive. Each process completes its receive because the other process sends a message that matches. Barring system failure, the program runs to completion.

C Example:

```
/* deadlock avoided */
#include
#include

void main (int argc, char **argv) {

int myrank;
MPI_Request request;
MPI_Status status;
double a[100], b[100];

MPI_Init(&argc, &argv);  /* Initialize MPI */
MPI_Comm_rank(MPI_COMM_WORLD, &myrank); /* Get rank */
if( myrank == 0 ) {
  /* Post a receive, send a message, then wait */
  MPI_Irecv( b, 100, MPI_DOUBLE, 1, 19, MPI_COMM_WORLD, &request );
  MPI_Send( a, 100, MPI_DOUBLE, 1, 17, MPI_COMM_WORLD );
  MPI_Wait( &request, &status );
}
else if( myrank == 1 ) {
  /* Post a receive, send a message, then wait */
  MPI_Irecv( b, 100, MPI_DOUBLE, 0, 17, MPI_COMM_WORLD, &request );
  MPI_Send( a, 100, MPI_DOUBLE, 0, 19, MPI_COMM_WORLD );
  MPI_Wait( &request, &status );
}

MPI_Finalize();          /* Terminate MPI */

}
```

Fortran Example:

```
      PROGRAM simple_deadlock_avoided

      INCLUDE 'mpif.h'
      INTEGER myrank, ierr, status(MPI_STATUS_SIZE)
      INTEGER request
      REAL a(100), b(100)

C Initialize MPI:
```

```
   call MPI_INIT(ierr)

C Get my rank:

   call MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)

C Process 0 posts a receive, then sends; same for process 1

   if( myrank.eq.0 )then
      call MPI_IRECV( b, 100, MPI_REAL, 1, 19, MPI_COMM_WORLD, request,
ierr )
      call MPI_SEND( a, 100, MPI_REAL, 1, 17, MPI_COMM_WORLD, ierr)
      call MPI_WAIT( request, status, ierr )

   else if ( myrank.eq.1 )then
      call MPI_IRECV( b, 100, MPI_REAL, 0, 17, MPI_COMM_WORLD, request,
ierr )
      call MPI_SEND( a, 100, MPI_REAL, 0, 19, MPI_COMM_WORLD, ierr)
      call MPI_WAIT( request, status, ierr )

   endif


C Terminate MPI:

   call MPI_FINALIZE(ierr)

   END
```

4.4. Send Modes

Send Modes

MPI provides the following four send modes:

1. Standard Mode Send
2. Synchronous Mode Send
3. Ready Mode Send
4. Buffered Mode Send

This section describes these send modes and briefly indicates when they are useful. Standard mode, used in all example code so far in this chapter, is the most widely used.

Although there are four send modes, there is only one receive mode. A receiving process can use the same call to MPI_RECV or MPI_IRECV, regardless of the send mode used to send the message.

Both blocking and nonblocking calls are available for each of the four send modes.

4.4.1. Standard Mode Send

Standard Mode Send

Standard mode send is MPI's general-purpose send mode. The other three send modes are useful in special circumstances, but none have the general utility of standard mode.

Recall the discussion of Sections 4.2.4 - What Happens at Runtime and 4.2.5 - Blocking and Completion. When MPI executes a standard mode send, one of two things happens. Either the message is copied into an MPI internal buffer and transferred asynchronously to the destination process, or the source and destination processes synchronize on the message. The MPI implementation is free to choose (on a case-by-case basis) between buffering and synchronizing, depending on message size, resource availability, etc.

If the message is copied into an MPI internal buffer, then the send operation is formally completed as soon as the copy is done. If the two processes synchronize, then the send operation is formally completed only when the receiving process has posted a matching receive and actually begun to receive the message.

The preceding comments apply to both blocking and nonblocking calls, i.e., to both MPI_SEND and MPI_ISEND. MPI_SEND does not return until the send operation it invoked has completed. Completion can mean the message was copied into an MPI internal buffer, or it can mean the sending and receiving processes synchronized on the message. In contrast, MPI_ISEND initiates a send operation and then returns immediately, without waiting for the send operation to complete. Completion has the same meaning as before: either the message was copied into an MPI internal buffer or the sending and receiving processes synchronized on the message.

Note: the variables passed to MPI_ISEND cannot be used (should not even be read) until the send operation invoked by the call has completed. A call to MPI_TEST, MPI_WAIT or one of their variants is needed to determine completion status.

One of the advantages of standard mode send is that the choice between buffering and synchronizing is left to MPI on a case-by-case basis. In general, MPI has a clearer view of the tradeoffs, especially since low-level resources and resources internal to MPI are involved.

4.4.2. Synchronous, Ready Mode, and Buffered Send

Synchronous, Ready Mode, and Buffered Send

Synchronous mode send requires MPI to synchronize the sending and receiving processes.

When a synchronous mode send operation is completed, the sending process may assume the destination process has begun receiving the message. The destination process need not be done receiving the message, but it must have begun receiving the message.

The nonblocking call has the same advantages the nonblocking standard mode send has: the sending process can avoid blocking on a potentially lengthy operation.

Ready mode send requires that a matching receive has already been posted at the destination process before ready mode send is called. If a matching receive has not been posted at the destination, the result is undefined. It is your responsibility to make sure the requirement is met.

In some cases, knowledge of the state of the destination process is available without doing extra work. Communication overhead may be reduced because shorter protocols can be used internally by MPI when it is known that a receive has already been posted.

The nonblocking call has advantages similar to the nonblocking standard mode send: the sending process can avoid blocking on a potentially lengthy operation.

Buffered mode send requires MPI to use buffering. The downside is that you must assume responsibility for managing the buffer. If at any point, insufficient buffer is available to complete a call, the results are undefined. The functions MPI_BUFFER_ATTACH and MPI_BUFFER_DETACH allow a program to make buffer available to MPI.

4.4.3. Naming Conventions and Calling Sequences

Naming Conventions and Calling Sequences

There are eight send functions in MPI: four send modes, each available in both blocking and nonblocking forms.

Synchronous, buffered, and ready mode sends are indicated by adding the letters S, B, and R, respectively, to the function name. Nonblocking calls are indicated by adding an I to the function name. The table below shows the eight function names.

Send Mode	Blocking Function	Nonblocking Function
Standard	MPI_SEND	MPI_ISEND
Synchronous	MPI_SSEND	MPI_ISSEND
Ready	MPI_RSEND	MPI_IRSEND
Buffered	MPI_BSEND	MPI_IBSEND

The blocking send functions take the same arguments (in the same order) as MPI_SEND. The nonblocking send functions take the same arguments (in the same order) as MPI_ISEND.

4.5. Self Test

Point-to-Point Communications Self Test

Now that you've finished this chapter, test yourself on what you've learned by taking the Self Test provided. Simply click on the Self Test link in the ACTION MENU above to get started.

4.6. Course Problem

Chapter 4 Course Problem

In this chapter, you learned about the heart of MPI: point-to-point message passing routines. Both their blocking and non-blocking forms were discussed as well as the various modes of communication. Armed with the knowledge of this chapter, you can now write the real (not pseudo) parallel MPI code to solve the first version of the course problem. Again, the problem description is the same as the one given in Chapter 1.

Description

The initial problem implements a parallel search of an extremely large (several thousand elements) integer array. The program finds all occurrences of a certain integer, called the target, and writes all the array indices where the target was found to an output file. In addition, the program reads both the target value and all the array elements from an input file.

Exercise

Go ahead and write the real parallel code for the search problem! Using the pseudo-code from the previous chapter as a guide, fill in all the sends and receives with calls to the actual MPI send and receive routines. For this task, use **only** the blocking routines. If you have access to a parallel computer with the MPI library installed, run your parallel code using 4 processors. See if you get the same results as those obtained with the serial version of Chapter 2. Of course, you should.

Solution

When you have finished writing the code for this exercise, view our version of the MPI parallel search code.

5. Derived Datatypes

Derived Datatypes and Related Features

In previous sections, you learned how to send and receive messages in which all the data was of a single type intrinsic to MPI and stored contiguously in memory. While your data may occasionally be that well behaved, it is likely that you will need to transmit collections of data of mixed types defined by the program, or data that are scattered in memory. In this chapter, you will learn strategies and features in MPI that allow you to address these circumstances.

5.1. Multiple Messages

The Simplest Strategy - Multiple Messages

Conceptually, the simplest approach is to identify the largest pieces of your data that individually meet the requirements of being of homogeneous intrinsic type and contiguous in memory and send each of those pieces as a separate message.

For example, consider the following problem: You have a large matrix stored in a two-dimensional array and you wish to send a rectangular submatrix to another processor. In Fortran, arrays are stored so that successive elements of a column are contiguous in memory, with the last element in one column followed by the first element of the next column. Since you are not sending full columns of the original matrix, the columns of your submatrix will not be contiguous. However, the elements within a column will be contiguous, so you can send one message for each column of the submatrix. If the array is declared to be **NN** by **MM** and you are sending the **N** by **M** portion whose upper left corner is at position **(K,L)**, this might look like

```
      DO 10 J=1,M
      CALL MPI_SEND(A(K,L+J-1), N, MPI_DOUBLE,
     & DEST, TAG, MPI_COMM_WORLD, IERROR)
10    CONTINUE
```

In C, the approach would be similar but because arrays are stored with the elements of rows contiguous in memory rather than columns, you would send **N** messages containing the **M** elements in a row of the submatrix. Whereas in the Fortran example, you sent **M** messages containing the **N** elements of a column.

```
for (i=0; i<n; ++i) {
```

```
   MPI_Send(&a[k+i][l], m, MPI_DOUBLE,
      dest, tag, MPI_COMM_WORLD);
}
```

In either language, if the receiving processor doesn't know the values of **N**, **M**, **K**, or **L** , they can be sent in a separate message.

- The principal advantage of this approach is that you don't need to learn anything new to apply it.
- The principal disadvantage of this approach is its overhead. A fixed overhead is associated with the sending and receiving of a message, however long or short it is. If you replace one long message with multiple short messages, you slow down your program by greatly increasing your overhead.

If you are working in a portion of your program that will be executed infrequently and the number of additional messages is small, the total amount of added overhead may be small enough to ignore. However, for most programs there won't be any advantage in limiting yourself to this strategy. You will have to learn other techniques for the heavily executed portions.

5.2. Copying Data into a Buffer

Another Simple Strategy - Copying Data into a Buffer

If your data isn't stored in contiguous memory, why not copy it into a contiguous buffer?

For our submatrix example, this might look like

```
p = &buffer;
for (i=0; i<n; ++i) {
   for(j=0; j<m; ++j) {
      *(p++) = a[i][j];
   }
}

MPI_Send(p, n*m, MPI_DOUBLE, dest, tag, MPI_COMM_WORLD)
```

Notes:

- This approach eliminates the excessive messages of the previous approach, at the cost of extra memory for the buffer and extra CPU time to perform the copy into the buffer.
- The obvious limitation of this approach is that it still handles only one type of data at a time.

5.3. A Tempting Wrong Way to Extend Buffering

A Tempting Wrong Way to Extend Buffering

It is often possible to encode the values of one type as values of another type. In our submatrix example, we could convert the values of **N**, **M**, **K**, and **L** to floating point in order to include them in our buffer. However, such conversions generally take more CPU time than a simple copy and, in most cases, the result will occupy more memory.

At this point, you may be tempted to use a programming trick (e.g., EQUIVALENCE, the TRANSFER function, or casting the type of a pointer) to put the bit patterns for values of one type into a buffer declared to be of some other type. This approach can be very dangerous. If you code up a test program to try it out, it is highly likely that it will "work" for you. However, if you use this approach extensively, especially in programs that are run in a variety of environments, it is almost inevitable that it will eventually fail. If you are lucky, it will fail spectacularly. If you are **not** lucky, you may just get incorrect results without realizing that something has gone wrong.

The fundamental problem here is that MPI transmits values, not just bit patterns. As long as you are using a set of processors that all represent values the same way, MPI optimizes its communications by simply transmitting the bit patterns in your message and tricks like this will "work". If there is any chance of communicating with a processor that uses a different representation for some or all of the values, MPI translates the values in your message into a standard intermediate representation, transmits the bits of the intermediate representation, and then translates the intermediate representation back into values on the other processor. This extra translation ensures that the same value is received as was sent. However, on the receiving processor that value may no longer have the same bit pattern as the value in the original type.

5.4. Buffering the Right Way

Buffering the Right Way - Pack Up Your Troubles

The MPI_PACK routine allows you to fill a buffer "the right way". You call MPI_PACK with arguments that describe the buffer you are filling and with most of the arguments you would have provided to MPI_SEND in our simplest approach. MPI_PACK copies your data into the buffer and, if necessary, translates it into a standard intermediate representation. After all the data you want to send have been placed in the buffer by MPI_PACK, you can send the buffer (giving its type as MPI_PACKED) and no further translations will be performed.

With MPI_PACK, the submatrix example becomes

```
      COUNT = 0
      DO 10 I=1,M
      CALL MPI_PACK(A(K,L+I-1), N, MPI_DOUBLE,
     & BUFFER, BUFSIZE, COUNT,
     & MPI_COMM_WORLD, IERROR)
10    CONTINUE
      CALL MPI_SEND(BUFFER, COUNT, MPI_PACKED,
     & DEST, TAG, MPI_COMM_WORLD, IERROR)
```

COUNT is initially set to zero to indicate you are starting the construction of a new message and have an empty buffer. The successive calls to MPI_PACK update COUNT to reflect the data that have been added to the buffer. The final value of COUNT is then used in the call to MPI_SEND as the amount of data to send.

On the receiving side, you can similarly specify type MPI_PACKED to receive a buffer without translation and then use MPI_UNPACK (which bears the same resemblance to MPI_RECV as MPI_PACK bears to MPI_SEND) to translate and copy the data from the buffer to where you really want it.

Because of translation, data may occupy a different amount of space in the buffer than it does natively. You can make your buffer large enough by using the routine MPI_PACK_SIZE to calculate how much buffer space is required for the different types of data you plan to place in your buffer.

Nothing in the content of a message indicates it was or was not built with MPI_PACK. If, as in our example, all the data packed into the buffer is of a single type, the message could be received in a buffer of that type rather than receiving it as MPI_PACKED and using MPI_UNPACK to decode it. Conversely, a message that was sent as an ordinary intrinsic type could be received as MPI_PACKED and distributed using calls to MPI_UNPACK.

Use of MPI_PACK and MPI_UNPACK provides great flexibility. In addition to allowing messages that include arbitrary mixtures of datatypes, its incremental construction and interpretation of messages allows the values of data early in a message to affect the type, size, or destination of data appearing later in the same message. The principal costs of this flexibility are the memory used for the buffers and CPU time used in copying data to and from those buffers. If constructing a message requires a large number of calls to MPI_PACK (or interpreting a message requires a large number of calls to MPI_UNPACK), the added procedure call overhead may also be significant.

5.5. Packing

Packing "On-the-Fly" - MPI Derived Types

You can look at the MPI derived type facility as a way to get MPI to do packing and unpacking "on-the-fly" as part of the send and receive operations. The packing and unpacking can then be done directly to and from its internal communications buffers, eliminating the need for

- the explicit intermediate buffer used when you do the packing and unpacking and
- the copying between the intermediate buffer and the communications buffer.

Thus, using MPI derived types in place of explicit packing and unpacking will generally make your program more efficient.

When sending, instead of building up a list of already packed data, you build up a list of locations from which data need to be packed. This list is used to define a type and the type is used in the send. The submatrix example might become

```
      DO 10 I=1,M
      LENA(I) = N
      CALL MPI_ADDRESS(A(K,L+I-1), LOCA(I), IERROR)
      TYPA(I) = MPI_DOUBLE
10      CONTINUE
      CALL MPI_TYPE_STRUCT(M, LENA, LOCA, TYPA,
     & MY_MPI_TYPE, IERROR)
      CALL MPI_TYPE_COMMIT(MY_MPI_TYPE,IERROR)
      CALL MPI_SEND(MPI_BOTTOM, 1, MY_MPI_TYPE,
     & DEST, TAG, MPI_COMM_WORLD, IERROR)
      CALL MPI_TYPE_FREE(MY_MPI_TYPE,IERROR)
```

The three arrays LENA, LOCA, and TYPA are used to record the length, location, and type of the data that in the previous version were fed to MPI_PACK to put in the buffer. MPI_ADDRESS is used to obtain the location of data relative to the magic address MPI_BOTTOM. After the three arrays have been filled, MPI_TYPE_STRUCT is used to convert that information into a new MPI type indicator stored in the variable MY_MPI_TYPE. MPI_TYPE_COMMIT is used to inform MPI that MY_MPI_TYPE will be used in a send or receive. MY_MPI_TYPE is then used as the type indicator in an actual send operation. The special fixed address, MPI_BOTTOM, is specified here as the nominal location of the data to send. That is because the locations in an MPI derived type specification are always interpreted relative to the data location and the location values obtained from MPI_ADDRESS are relative to MPI_BOTTOM. Finally, MPI_TYPE_FREE is used to inform MPI that you don't intend to use this particular type again, so the resources used to represent this type inside MPI may be release or reused.

When receiving, you must similarly build up a list of locations to receive the data from a message, convert those locations to a committed type, and use that type in a receive operation.

Note:

- As a direct replacement for pack and unpack operations, MPI derived type operations are usually more efficient but somewhat more verbose, because of the need to explicitly create, commit, and free MPI type indicators.
- If one is packing data that doesn't remain in existence until the time the packed buffer is sent (e.g., if successive values to be packed are computed into the same variables), derived type operations lose their efficiency advantage because you must institute some other form of buffering to retain those values until they can be sent.
- Similarly, if the locations to which data are to be unpacked are not disjoint (e.g., if successive values from a message are processed in the same variables), derived type operations lose their efficiency advantage. This is because you will need to buffer those values somewhere else until they can be processed. Buffering is also necessary if the locations to receive the data cannot be determined in advance.
- Derived type operations cannot be used to replace unpacking in those cases where values in the early part of a message determine the structure of the later part of the message. In such cases, explicitly typed buffering will not work and you need the flexibility of piecemeal unpacking of an MPI_PACKED buffer.

5.6. Using MPI Derived Types for User-Defined Types

Using MPI Derived Types for User-Defined Types

Creating an MPI derived type to use it just once before freeing it can be a bit verbose. It is far more effective to create MPI derived types that describe recurring patterns of access and then reuse such types for each occurrence of that pattern of access. The classic example of this is the use of an MPI derived type to describe the access associated with a user-defined datatype in the language you are using. This technique is called **mapping**.

Here is a simple example of mapping a C struct type:

```
   struct SparseElt {     /* representation of a sparse matrix element
*/
      int    location[2]; /* where the element belongs in the overall
matrix */
      double value;       /* the value of the element */
   };

   struct SparseElt anElement; /* a representative variable of this
type */

   int          lena[2]; /* the three arrays used to describe an MPI
derived type */
   MPI_Aint     loca[2]; /* their size reflects the number of
components in SparseElt */
   MPI_Datatype typa[2];
```

```
   MPI_Aint     baseaddress;

   MPI_Datatype MPI_SparseElt; /* a variable to hold the MPI type
indicator for SparseElt */

   /* set up the MPI description of SparseElt */

   MPI_Address(&anElement, &baseaddress);
   lena[0] = 2; MPI_Address(&anElement.location,&loca[0]);
   loca[0] -= baseaddress; typa[0] = MPI_INT;
   lena[1] = 1; MPI_Address(&anElement.value   ,&loca[1]);
   loca[1] -= baseaddress; typa[1] = MPI_DOUBLE;
   MPI_Type_struct(2, lena, loca, typa, &MPI_SparseElt);
   MPI_Type_commit(&MPI_SparseElt);
```

As in our earlier example, we construct three arrays containing the length, location, and types of the components to be transferred when this type is used. Unlike our earlier example, we subtract the address of the whole variable from the addresses of its components, so the locations are specified relative to the variable rather than relative to MPI_BOTTOM. This allows us to use the type indicator MPI_SparseElt to describe a variable of type SparseElt anywhere in the program.

Once a type has been created and committed, it may be used **anywhere** an intrinsic type indicator can be used, not just in send and receive operations. In particular, this includes its use in defining another MPI derived type that might have a SparseElt component or in performing pack or unpack operations on variables of type SparseElt.

5.7. Other Ways of Defining MPI Derived Types

Other Ways of Defining MPI Derived Types

MPI_TYPE_STRUCT is the most general way to construct an MPI derived type because it allows the length, location, and type of each component to be specified independently. Less general procedures are available to describe common patterns of access, primarily within arrays. These are

- MPI_TYPE_CONTIGUOUS
- MPI_TYPE_VECTOR
- MPI_TYPE_HVECTOR
- MPI_TYPE_INDEXED
- MPI_TYPE_HINDEXED

MPI_TYPE_CONTIGUOUS is the simplest of these, describing a contiguous sequence of values in memory. For example,

```
   MPI_Type_contiguous(2,MPI_DOUBLE,&MPI_2D_POINT);
```

```
   MPI_Type_contiguous(3,MPI_DOUBLE,&MPI_3D_POINT);
```

creates new type indicators MPI_2D_POINT and MPI_3D_POINT. These type indicators allow you to treat consecutive pairs of doubles as point coordinates in a 2-dimensional space and sequences of three doubles as point coordinates in a 3-dimensional space.

MPI_TYPE_VECTOR describes several such sequences evenly spaced but not consecutive in memory. With it, you can reduce the submatrix example to

```
   CALL MPI_TYPE_VECTOR(M, N, NN, MPI_DOUBLE,
  & MY_MPI_TYPE, IERROR)
   CALL MPI_TYPE_COMMIT(MY_MPI_TYPE,IERROR)
   CALL MPI_SEND(A(K,L), 1, MY_MPI_TYPE,
  & DEST, TAG, MPI_COMM_WORLD, IERROR)
   CALL MPI_TYPE_FREE(MY_MPI_TYPE,IERROR)
```

The consecutive blocks are the columns of the submatrix, each **N** elements long. There are **M** of them. These columns start **NN** elements apart because that is the declared column size of the array containing the submatrix.

MPI_TYPE_HVECTOR is similar to MPI_TYPE_VECTOR except that the distance between successive blocks is specified in bytes rather than elements. The most common reason for specifying this distance in bytes would be that elements of some other type are interspersed in memory with the elements of interest. For example, if you had an array of type SparseElt, you could use MPI_TYPE_HVECTOR to describe the "array" of value components.

MPI_TYPE_INDEXED describes sequences that may vary both in length and in spacing. Because the location of these sequences is measured in elements rather than bytes, it is most appropriate for identifying arbitrary parts of a single array.

MPI_TYPE_HINDEXED is similar to MPI_TYPE_INDEXED except that the locations are specified in bytes rather than elements. It allows the identification of arbitrary parts of arbitrary arrays, subject only to the requirement that they all have the same type.

5.8. Message Matching and Mismatching

Message Matching and Mismatching

Just as there is nothing in the content of a message to indicate whether it was built with MPI_PACK, there is nothing to indicate whether or what kind of MPI derived types may have been used in its construction. All that matters is that the sender and receiver agree on the nature of the sequence of primitive values the message represents. Thus, a message constructed and sent using MPI_PACK could be received using MPI derived types, or a message sent using using MPI derived types could be received as MPI_PACKED and distributed using MPI_UNPACK. Similarly, a message could be sent using one MPI derived type and received using a different type.

This leads to one significant difference between MPI derived types and its primitive types. If you send data using the same MPI derived type with which you will be receiving it, the message will necessarily contain an integral number of that type, and MPI_GET_COUNT will work for that type in the same way it does for primitive types. However, if the types are different, you could end up with a partial value at the end. For example, if the sending processor sends an array of four MPI_3D_POINTs (or a total of twelve MPI_DOUBLEs) and the receiving processor receives the message as an array of MPI_2D_POINTs, MPI_GET_COUNT will report that six MPI_2D_POINTs were received. If instead five MPI_3D_POINTs were sent (i.e., fifteen MPI_DOUBLEs), seven and a half MPI_2D_POINTs will be changed on the receiving side, but MPI_GET_COUNT cannot return "7.5". Rather than return a potentially misleading value such as "7" or "8", MPI_GET_COUNT returns the flag value MPI_UNDEFINED. If you need more information about the size of the transfer, you can still use MPI_GET_ELEMENTS to learn that nine primitive values were transferred.

5.9. Controlling the Extent of a Derived Type

Controlling the Extent of a Derived Type

The concept of several derived type values being contiguous in memory can be somewhat problematic in the general case of transferring arbitrary sequences of data. MPI's rules for this are designed to work as you might expect for the common case of mapping an MPI derived type onto a user-defined type. First, MPI computes the lower and upper bounds of the type. By default, the lower bound is the beginning of the component that appears first in memory, and the upper bound is the end of the component that appears last in memory (with "end" possibly including adjustments to reflect alignment or padding rules). The distance between the lower bound and upper bound of a type is called its **extent**. Two elements of that type are considered to be contiguous in memory if the distance between them matches the extent. In other words, they are contiguous if the lower bound of the second element exactly coincides with the upper bound of the first. This approach to defining the extent of a derived type element usually produces the "right" results. However, there are cases where it does not.

- The MPI library can implement only one set of padding and alignment rules. If your compiler has options to control these rules or if the compilers for different languages use different rules, then MPI may occasionally compute the upper bound using the "wrong" padding and alignment rules.
- If your MPI derived type maps only part of the components in the user-defined type, MPI may not know about the real first component or last component and thus underestimate the extent.
- If the components of your derived type are arbitrary storage sequences, the default extent will nearly always lack any useful meaning.

In these cases, you can control the bounds of the type, and thus the extent, by inserting components of type MPI_LB and MPI_UB into your derived type definition. The locations of these components are then treated as the lower and upper bounds, regardless of the locations of the other components. One of the simplest solutions is to base the lower and upper bounds on an array of the type you are mapping, specifying the location of the first element in the array as the lower bound of the type and the location of the second element in the array as the upper bound of the type. For example, consider a program in which you have arrays **X**, **Y**, and **Z**. At some point, you would like to send the first **N** values from **X**, **Y**, and **Z** to another processor. If you do things in the obvious way, transmitting first **N** values from **X**, then **N** values from **Y**, and finally **N** values from **Z**, the receiving processor won't know where the values from **X** end and the values from **Y** begin until it has received all the values and can use the length of the message to determine the value of **N**. Putting the value of **N** at the beginning of the message doesn't help, because the receiving processor must define where it wants the values delivered before it receives any part of the message. The solution to this problem is to rearrange the values in the message so first you transfer the first **X**, the first **Y**, and the first **Z**, then the second **X**, **Y**, and **Z**, then the third, etc. This arrangement allows the receiving processor to know which value is which without knowing the total number of values in advance.

```
  LENA(1) = 1
  CALL MPI_ADDRESS(X(1), LOCA(1), IERROR)
  TYPA(1) = MPI_DOUBLE
  LENA(2) = 1
  CALL MPI_ADDRESS(Y(1), LOCA(2), IERROR)
  TYPA(2) = MPI_DOUBLE
  LENA(3) = 1
  CALL MPI_ADDRESS(Z(1), LOCA(3), IERROR)
  TYPA(3) = MPI_DOUBLE
  LENA(4) = 1
  CALL MPI_ADDRESS(X(1), LOCA(4), IERROR)
  TYPA(4) = MPI_LB
  LENA(5) = 1
  CALL MPI_ADDRESS(X(2), LOCA(5), IERROR)
  TYPA(5) = MPI_UB
  CALL MPI_TYPE_STRUCT(5, LENA, LOCA, TYPA, MY_TYPE, IERROR)
  CALL MPI_TYPE_COMMIT(MY_TYPE,IERROR)
  CALL MPI_SEND(MPI_BOTTOM, N, MY_TYPE,
 & DEST, TAG, MPI_COMM_WORLD,IERROR)
  CALL MPI_TYPE_FREE(MY_TYPE,IERROR)
```

This formulation of MY_TYPE works because **X**, **Y**, and **Z** are of the same type, so **Y(2)** is at the same position relative to **X(2)** as **Y(1)** is to **X(1)**, etc. Note that **N** is used only in the send, not in the definition of MY_TYPE, so you can define MY_TYPE once and use it for multiple sends rather than freeing it after each use and redefining it for the next send.

5.10. Obtaining Information About Your Derived Types

Obtaining Information About Your Derived Types

Once you have defined a derived type, several utility procedures can provide you with information about that type.

- MPI_TYPE_LB and MPI_TYPE_UB can provide the lower and upper bounds of the type.
- MPI_TYPE_EXTENT can provide the extent of the type. In most cases, this is the amount of memory a value of the type will occupy.
- MPI_TYPE_SIZE can provide the size of the type in a message. If the type is scattered in memory, this may be significantly smaller than the extent of the type.

5.11. Self Test

Derived Datatypes Self Test

Now that you've finished this chapter, test yourself on what you've learned by taking the Self Test provided. Simply click on the Self Test link in the ACTION MENU above to get started.

5.12. Course Problem

Chapter 5 Course Problem

This chapter discussed a powerful feature of MPI: the ability to make up one's own data type to match the collection of different kinds of data that are being sent from one processor to another. For this chapter, the initial Course Problem is modified so that each slave must send an integer *and* a real value back to the master. A new datatype will be created and used to send/receive both of these data as a single entity.

Description

The new problem still implements a parallel search of an integer array. The program should find all occurences of a certain integer which will be called the target. It should then calculate the average of the target value and its index. Both the target location and the average should be written to an output file. In addition, the program should read both the target value and all the array elements from an input file.

Exercise

Modify your code from Chapter 4 to create a program that solves the new Course Problem. Use the techniques/routines of this chapter to make a new derived type called MPI_PAIR that will contain both the target location and the average. All of the slave sends and the master receives **must use** the MPI_PAIR type.

Solution

When you have finished writing your code for this exercise, view our version of Derived Type Code.

---

6. Collective Communications

Collective Communications

Collective communication involves the sending and receiving of data among processes. In general, all movement of data among processes can be accomplished using MPI send and receive routines. However, some sequences of communication operations are so common that MPI provides a set of collective communication routines to handle them. These routines are built using point-to-point communication routines. Even though you could build your own collective communication routines, these "blackbox" routines hide a lot of the messy details and often implement the most efficient algorithm known for that operation.

Collective communication routines transmit data among all processes in a group. It is important to note that collective communication calls do not use the tag mechanism of send/receive for associating calls. Rather they are associated by order of program execution. Thus, the user must ensure that all processors execute the same collective communication calls and execute them in the same order.

The collective communication routines allow data motion among all processors or just a specified set of processors. The notion of communicators that identify the set of processors specifically involved in exchanging data was introduced in Section 3.11 - Communicators. The examples and discussion for this chapter assume that all of the processors participate in the data motion. However, you may define your own communicator that provides for collective communication between a subset of processors.

MPI provides the following collective communication routines:

- Barrier sychronization across all processes
- Broadcast from one process to all other processes
- Global reduction operations such as *sum, min, max* or user-defined reductions
- Gather data from all processes to one process
- Scatter data from one process to all processes
- Advanced operations where all processes receive the same result from a gather, scatter, or reduction. There is also a *vector* variant of most collective operations where each message can be a different size.

Each of these routines is described in more detail in the following sections.

Note:

- For many implementations of MPI, calls to collective communication routines will sychronize the processors. However, this synchronization is not guaranteed and you should not depend on it. More detail can be found in Section 11 - Portability Issues.
- One routine, MPI_BARRIER, synchronizes the processes but does not pass data. It is nevertheless often categorized as one of the collective communications routines.

6.1. Barrier Synchronization

Barrier Synchronization

There are occasions when some processors cannot proceed until other processors have completed their current instructions. A common instance of this occurs when the root process reads data and then transmits these data to other processors. The other processors must wait until the I/O is completed and the data are moved.

The MPI_BARRIER routine blocks the calling process until all group processes have called the function. When MPI_BARRIER returns, all processes are synchronized at the barrier.

MPI_BARRIER is done in software and can incur a substantial overhead on some machines. In general, you should only insert barriers when they are needed.

C:

```
int MPI_Barrier ( comm  )
MPI_Comm comm
```

Fortran:

```
MPI_BARRIER ( COMM, ERROR )
INTEGER COMM, ERROR
```

6.2. Broadcast

Broadcast

The MPI_BCAST routine enables you to copy data from the memory of the root processor to the same memory locations for other processors in the communicator.

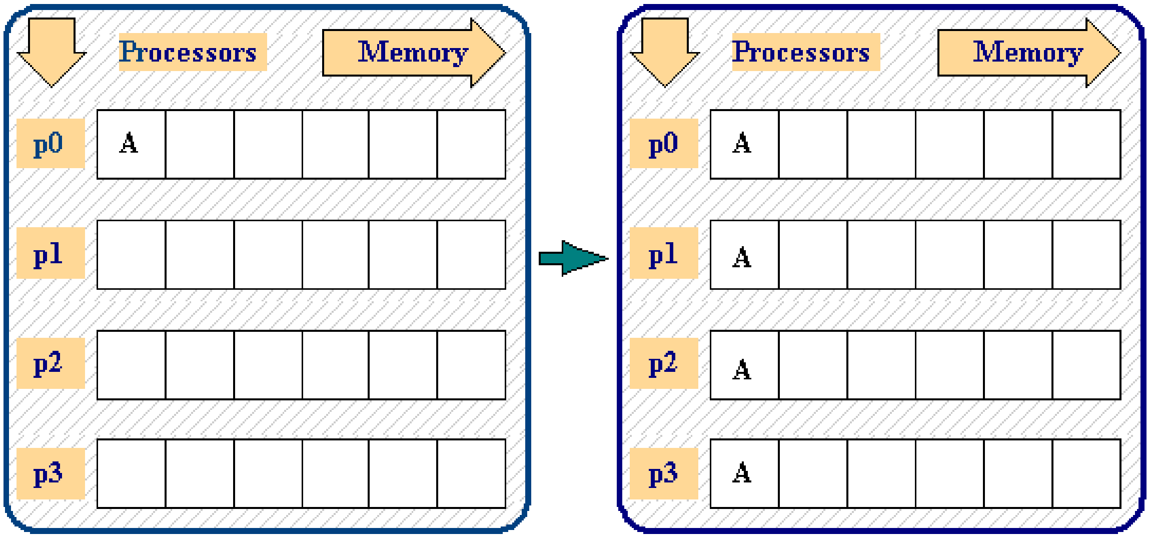


Figure 6.1: A Simple Broadcast Example

In this example, one data value in processor 0 is broadcast to the same memory locations in the other 3 processors. Clearly, you could send data to each processor with multiple calls to one of the send routines. The broadcast routine makes this data motion a bit easier.

All processes call the following:

```
   o
   o
   o

send_count = 1;
root = 0;
MPI_Bcast ( &a, &send_count, MPI_INT, root, comm )

   o
   o
   o
```

Syntax:

```
MPI_Bcast ( send_buffer, send_count, send_type, rank, comm )
```

The arguments for this routine are:

```
send_buffer   in/out   starting address of send buffer
```

```
send_count     in       number of elements in send buffer
send_type      in       data type of elements in send buffer
rank           in       rank of root process
comm           in       mpi communicator
```

C:

```
int MPI_Bcast ( void* buffer, int count,
                MPI_Datatype datatype, int rank,
                MPI_Comm comm )
```

Fortran:

```
MPI_BCAST ( BUFFER, COUNT, DATATYPE, ROOT, COMM, ERROR )

INTEGER COUNT, DATATYPE, ROOT, COMM, ERROR
<type> BUFFER
```

Sample Code

6.3. Reduction

Reduction

The MPI_REDUCE routine enables you to

- collect data from each processor
- reduce these data to a single value (such as a sum or max)
- and store the reduced result on the root processor

The example shown in Figure 6.2 sums the values of ***A*** on each processor and stores the results in ***X*** on processor 0.

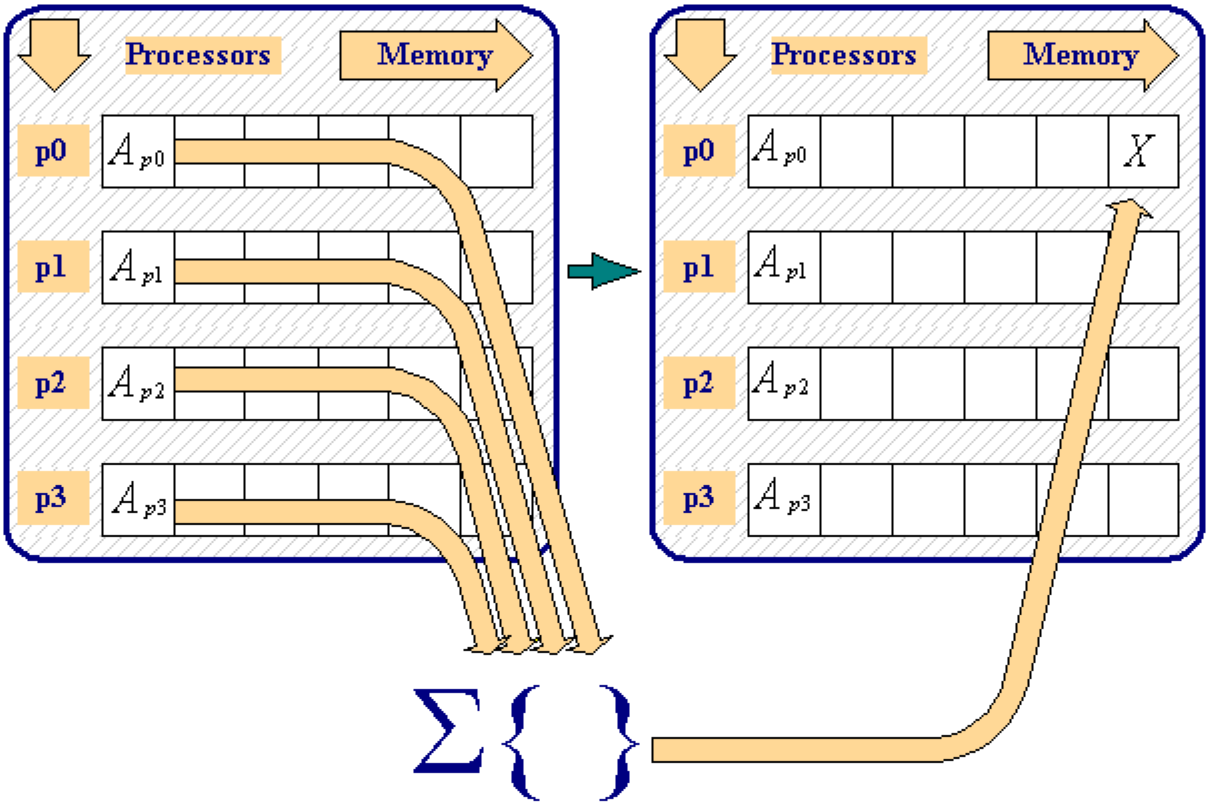


Figure 6.2: A Simple Reduction Example

The routine calls for this example are

```
   count = 1;
   rank = 0;
   MPI_Reduce ( &a, &x, count, MPI_REAL,
                MPI_SUM, rank, MPI_COMM_WORLD );

   COUNT = 1
   RANK = 0
   CALL MPI_REDUCE ( A, X, COUNT, MPI_REAL,
*                    MPI_SUM, RANK, MPI_COMM_WORLD )
```

In general, the calling sequence is

```
   MPI_Reduce( send_buffer, recv_buffer, count, data_type,
               reduction_operation, rank_of_receiving_process,
communicator )
```

MPI_REDUCE combines the elements provided in the send buffer, applies the specified operation (sum, min, max, ...), and returns the result to the receive buffer of the root process.

The send buffer is defined by the arguments send_buffer, count, and datatype.

The receive buffer is defined by the arguments recv_buffer, count, and datatype.

Both buffers have the same number of elements with the same type. The arguments count and datatype must have identical values in all processes. The argument rank, which is the location of the reduced result, must also be the same in all processes.

The following are predefined operations available for MPI_REDUCE.

Operation	Description
MPI_MAX	maximum
MPI_MIN	minimum
MPI_SUM	sum
MPI_PROD	product
MPI_LAND	logical and
MPI_BAND	bit-wise and
MPI_LOR	logical or
MPI_BOR	bit-wise or
MPI_LXOR	logical xor
MPI_BXOR	logical xor
MPI_MINLOC	computes a global minimum and an index attached to the minimum value -- can be used to determine the rank of the process containing the minimum value
MPI_MAXLOC	computes a global maximum and an index attached to the rank of the process containing the minimum value

Syntax:

```
   MPI_Reduce ( send_buffer, recv_buffer, count, datatype, operation,
rank, comm )
```

The arguments for this routine are:

```
send_buffer     in    address of send buffer
recv_buffer     out   address of receive buffer
count           in    number of elements in send buffer
datatype        in    data type of elements in send buffer
operation       in    reduction operation
rank            in    rank of root process
comm            in    mpi communicator
```

C:

```
   int MPI_Reduce ( void* send_buffer, void* recv_buffer, int count,
MPI_Datatype datatype, MPI_Op operation, int rank, MPI_Comm comm )
```

Fortran:

```
   MPI_Reduce ( SEND_BUFFER, RECV_BUFFER, COUNT, DATATYPE,
                OPERATION, RANK, COMM, ERROR )

   INTEGER COUNT, DATATYPE, OPERATION, COMM, ERROR
   <datatype> SEND_BUFFER,RECV_BUFFER
```

Sample Code

6.4. Gather

Gather

This section describes two gather routines: MPI_GATHER and MPI_ALLGATHER.

MPI_GATHER

The MPI_GATHER routine is an *all-to-one* communication. MPI_GATHER has the same arguments as the matching scatter routines. The receive arguments are only meaningful to the root process.

When MPI_GATHER is called, each process (including the root process) sends the contents of its send buffer to the root process. The root process receives the messages and stores them in rank order.

The gather also could be accomplished by each process calling MPI_SEND and the root process calling MPI_RECV ***N*** times to receive all of the messages.

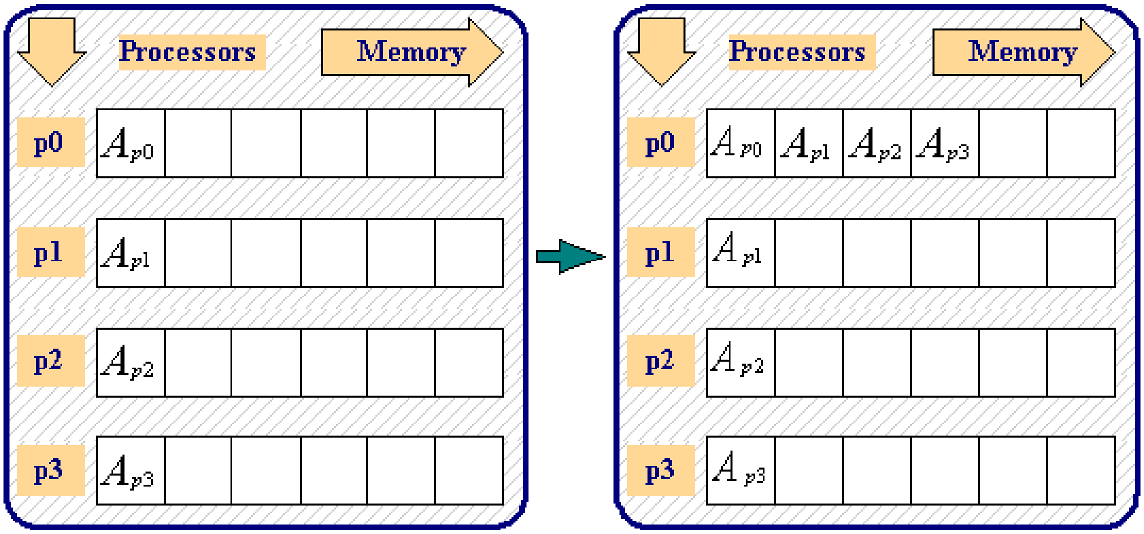


Figure 6.3: A Simple Gather Example

In this example, data values ***A*** on each processor are *gathered* and moved to processor 0 into contiguous memory locations.

The function calls for this example are

```
   send_count = 1;
   recv_count = 1;
   recv_rank = 0;
   MPI_Gather ( &a, send_count, MPI_REAL,
                &a, recv_count, MPI_REAL,
                recv_rank, MPI_COMM_WORLD );


   SEND_COUNT = 1
   RECV_COUNT = 1
   RECV_RANK = 0
   CALL MPI_GATHER ( A, SEND_COUNT, MPI_REAL,
*                    A, RECV_COUNT, MPI_REAL,
*                    RECV_RANK, MPI_COMM_WORLD )
```

MPI_GATHER requires that all processes, including the root, send the same amount of data, and the data are of the same type. Thus send_count = recv_count.

Syntax:

```
   MPI_Gather ( send_buffer, send_count, send_type,
                recv_buffer, recv_count, recv_count,
                recv_rank, comm )
```

The arguments for this routine are:

```
   send_buffer   in   starting address of send buffer
   send_count    in   number of elements in send buffer
   send_type     in   data type of send buffer elements
   recv_buffer   out  starting address of receive buffer
   recv_count    in   number of elements in receive buffer for a single
receive
   recv_type     in   data type of elements in receive buffer
   recv_rank     in   rank of receiving process
   comm          in   mpi communicator
```

C:

```
   int MPI_Gather ( void* send_buffer, int send_count, MPI_datatype
                    send_type, void* recv_buffer, int recv_count,
                    MPI_Datatype recv_type, int rank, MPI_Comm comm )
```

Fortran:

```
   MPI_GATHER ( SEND_BUFFER, SEND_COUNT, SEND_TYPE, RECV_BUFFER,
                RECV_COUNT, RECV_TYPE, RANK, COMM, ERROR )

   INTEGER SEND_COUNT, SEND_TYPE, RECV_COUNT, RECV_TYPE, RANK, COMM,
ERROR
   <datatype> SEND_BUFFER, RECV_BUFFER
```

Sample Code

MPI_ALLGATHER

In the previous example, after the data are *gathered* into processor 0, you could then MPI_BCAST the gathered data to all of the other processors. It is more convenient and efficient to *gather* and *broadcast* with the single MPI_ALLGATHER operation.

The result is the following:

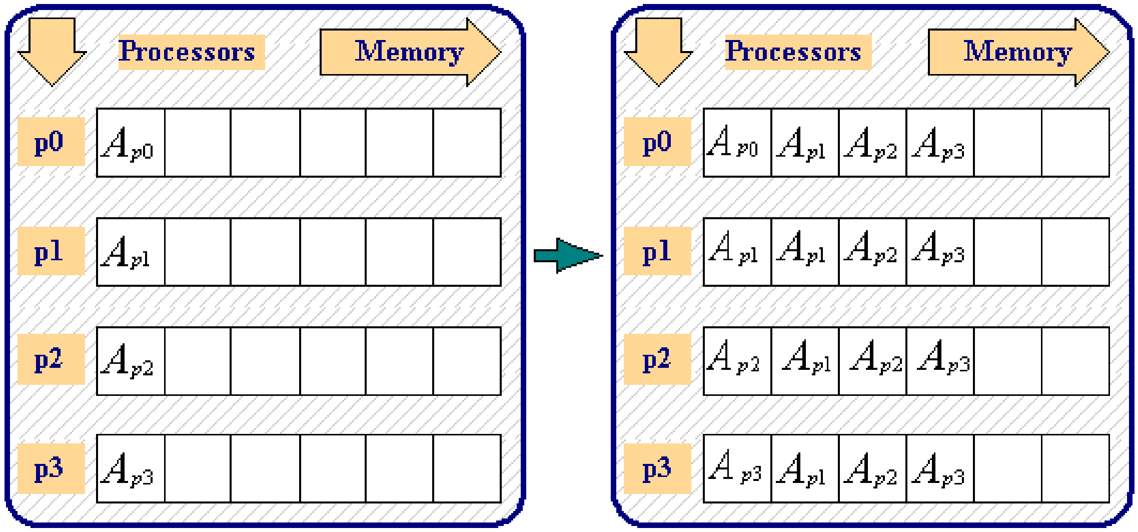


Figure 6.4: An AllGather Example

The calling sequence for MPI_ALLGATHER is exactly the same as the calling sequence for MPI_GATHER.

6.5. Scatter

Scatter

The MPI_SCATTER routine is a *one-to-all* communication. Different data are sent from the root process to each process (in rank order).

When MPI_SCATTER is called, the root process breaks up a set of contiguous memory locations into equal chunks and sends one chunk to each processor. The outcome is the same as if the root executed ***N*** MPI_SEND operations and each process executed an MPI_RECV.

The send arguments are only meaningful to the root process.

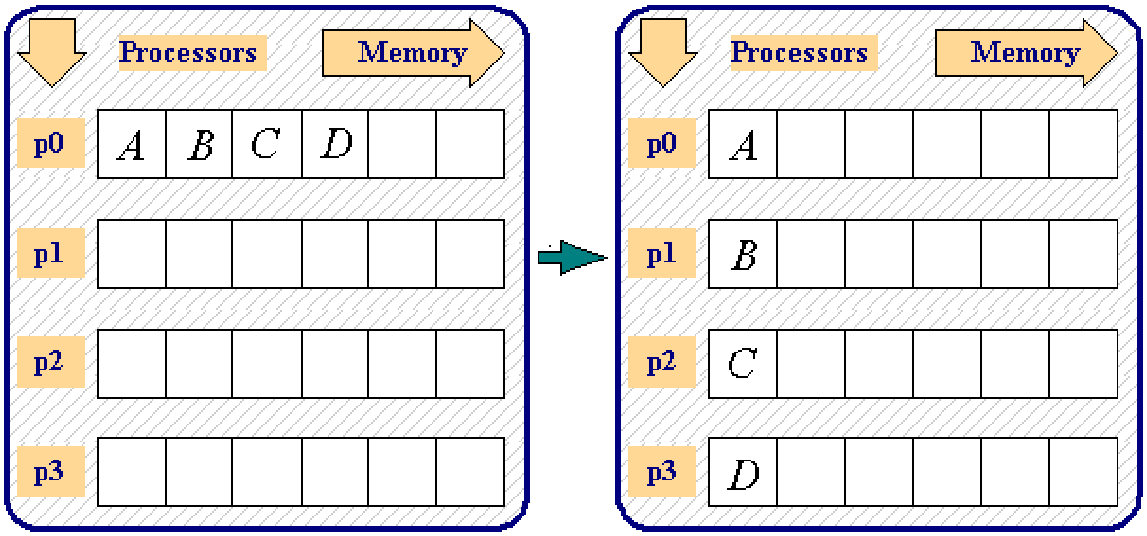


Figure 6.5. A Simple Scatter Example

In this example shown in Figure 6.5, four contiguous data values, elements of processor 0 beginning at ***A*** , are copied with one element going to each processor at location ***A***.

The function calls for this example are

```
   send_count = 1;
   recv_count = 1;
   send_rank = 0;
   MPI_Scatter ( &a, send_count, MPI_REAL,
                 &a, recv_count, MPI_REAL,
                 send_rank, MPI_COMM_WORLD );

   SEND_COUNT = 1
   RECV_COUNT = 1
   SEND_RANK = 0
   CALL MPI_SCATTER ( A, SEND_COUNT, MPI_REAL,
*                     A, RECV_COUNT, MPI_REAL,
*                     SEND_RANK, MPI_COMM_WORLD )
```

Syntax:

```
   MPI_Scatter ( send_buffer, send_count, send_type, recv_buffer,
recv_count,
                 recv_type, rank, comm )
```

The arguments for this routine are:

```
   send_buffer   in   starting address of send buffer
```

```
   send_count     in   number of elements in send buffer to send to each
process
                       (not the total number sent)
   send_type      in   data type of send buffer elements
   recv_buffer    out  starting address of receive buffer
   recv_count     in   number of elements in receive buffer
   recv_type      in   data type of elements in receive buffer
   rank           in   rank of receiving process
   comm           in   mpi communicator
```

C:

```
   int MPI_Scatter ( void* send_buffer, int send_count, MPI_datatype
send_type, void* recv_buffer, int recv_count, MPI_Datatype recv_type,
int rank, MPI_Comm comm )
```

Fortran :

```
   MPI_Scatter ( SEND_BUFFER, SEND_COUNT, SEND_TYPE, RECV_BUFFER,
                 RECV_COUNT, RECV_TYPE, RANK, COMM, ERROR )

   INTEGER SEND_COUNT, SEND_TYPE, RECV_COUNT, RECV_TYPE, RANK, COMM,
ERROR <datatype> SEND_BUFFER, RECV_BUFFER
```

Sample Code

6.6. Advanced Operations

Advanced Operations

MPI_ALLREDUCE

- MPI_ALLREDUCE is used to combine the elements of each process's input buffer
- Stores the combined value on the receive buffer of all group members

User Defined Reduction Operations

- Reduction can be defined to be an arbitrary operation

Gather / Scatter Vector Operations

- MPI_GATHERV and MPI_SCATTERV allow a varying count of data from/to each process

Other Gather / Scatter Variations

- MPI_ALLGATHER and MPI_ALLTOALL
- No root process specified: **all** processes get gathered or scattered data
- Send and receive arguments are meaningful to all processes

MPI_SCAN

- MPI_SCAN is used to carry out a prefix reduction on data throughout the group
- Returns the reduction of the values of all of the processes

MPI_REDUCE_SCATTER

- MPI_REDUCE_SCATTER combines an MPI_REDUCE and an MPI_SCATTERV

6.7. Self Test

Collective Communications Self Test

Now that you've finished this chapter, test yourself on what you've learned by taking the Self Test provided. Simply click on the Self Test link in the ACTION MENU above to get started.

6.8. Course Problem

Chapter 6 Course Problem

In almost every MPI program there are instances where *all* the proccessors in a communicator need to perform some sort of data transfer or calculation. These "collective communication" routines have been the subject of this chapter and our parallel search program is no exception. There are two obvious places where the code we have created so far can be simplified (and perhaps sped up) by the use of the correct collective communication routines. This problem description is the same as the one used in Chapter 5 but you will utilize collective communication routines to modify the code you wrote for that exercise.

Description

The new problem still implements a parallel search of an integer array. The program should find all occurrences of a certain integer which will be called the target. It should then calculate the average of the target value and its index. Both the target location and the average should be written to an output file. In addition, the program should read both the target value and all the array elements from an input file.

Exercise

Modify your code from Chapter 5, to change how the master first sends out the target and subarray data to the slaves. Use the MPI broadcast routines to give each slave the target. Use the MPI scatter routine to give all processors a section of the array b it will search.

When you use the standard MPI scatter routine you will see that the global array b is now split up into four parts and the master process now has the first fourth of the array to search. So you should add a search loop (similar to the slaves') in the master section of code to search for the target and calculate the average and then write the result to the output file. This is actually an improvement in performance since *all* the processors perform part of the search in parallel.

Solution

When you have finished writing the code for this exercise, view our version of the Collective Communication Code.

---

7. Communicators

Communicators

So far, the communicator that you are familiar with is MPI_COMM_WORLD (See Section 3.11 - Communicators). This is a communicator defined by MPI to permit all processes of your program to communicate with each other at run time, either between two processes (point-to-point) or among all processes (collective). For some applications however, communications among a selected subgroup of processes may be desirable or required. In this section, you will learn how to create new communicators for these situations.

Two types of communicators exist within MPI: intra-communicators and inter-communicators. This chapter will focus on intra-communicators that deal with communications among processes within individual communicators. Inter-communicators, on the other hand, deal with communications between intra-communicators. Essentially, intra-communicators are subsets of processes of MPI_COMM_WORLD. The need for these new communicators is often driven by the need to deal with, for instance, rows, columns or subblocks of a matrix. These communicators are often used in conjunction with a virtual topology -- more often than not a Cartesian topology -- to facilitate implementation of parallel operations. Furthermore, the use of communicators, and quite frequently together with virtual topology, generally enhances the readability and maintainability of a program.

To this end, many routines are available in the MPI library to perform various communication-related tasks. These are

- MPI_COMM_GROUP
- MPI_GROUP_INCL

- MPI_GROUP_EXCL
- MPI_GROUP_RANK
- MPI_GROUP_FREE
- MPI_COMM_CREATE
- MPI_COMM_SPLIT

These routines are described in the following sections.

7.1. MPI_COMM_GROUP

MPI_COMM_GROUP

The MPI_COMM_GROUP routine determines the group handle of a communicator.

C:

```
int MPI_Comm_group( MPI_Comm comm, MPI_Group *group )
```

Fortran:

```
MPI_COMM_GROUP( comm, group, ierr )
```

Variable Name	C Type	Fortran Type	In/Out	Description
comm	MPI_Comm	INTEGER	Input	Communicator handle
group	MPI_Group *	INTEGER	Output	Group handle
ierr	See (*)	INTEGER	Output	Error flag

* For C, the function returns an int error flag.

C Example:

```
#include "mpi.h"
MPI_Comm  comm_world;
MPI_Group group_world;

comm_world = MPI_COMM_WORLD;
MPI_Comm_group(comm_world, &group_world);
```

Fortran Example:

```
include "mpif.h"
integer comm_world, ierr, group_world

comm_world = MPI_COMM_WORLD
call MPI_Comm_group(comm_world, group_world, ierr)
```

Associated with a communicator is its group identity, or handle. In the above example, we used MPI_COMM_GROUP to obtain the group handle of the communicator MPI_COMM_WORLD. This handle can then be used as input to the routine

- MPI_GROUP_INCL to select among the processes of one group to form another (new) group;
- MPI_COMM_CREATE to create a new communicator whose members are those of the new group;
- MPI_GROUP_RANK to find the current process rank's equivalent process rank in a group.

7.2. MPI_GROUP_INCL

MPI_GROUP_INCL

The MPI_GROUP_INCL routine creates a new group from an existing group and specifies member processes.

C:

```
   int MPI_Group_incl( MPI_Group old_group, int count, int *members,
MPI_Group *new_group )
```

Fortran:

```
   MPI_GROUP_INCL( OLD_GROUP, COUNT, MEMBERS, NEW_GROUP, IERR )
```

Variable Name	C Type	Fortran Type	In/Out	Description
old_group	MPI_Group	INTEGER	Input	Group handle
count	int	INTEGER	Input	Number of processes in new_group
members	int *	INTEGER	Input	Array of size count defining process ranks (in old_group) to be included (in new_group)
new_group	MPI_Group *	INTEGER	Output	Group handle

ierr	See (*)	INTEGER	Output	Error flag

* For C, the function returns an int error flag.

C Example:

```
   #include "mpi.h"
   MPI_Group group_world, odd_group, even_group;
   int i, p, Neven, Nodd, members[8], ierr;

   MPI_Comm_size(MPI_COMM_WORLD, &p);
   MPI_Comm_group(MPI_COMM_WORLD, &group_world);

   Neven = (p+1)/2;    /* processes of MPI_COMM_WORLD are divided */
   Nodd = p - Neven;   /* into odd- and even-numbered groups */
   for (i=0; i<Neven; i++) {   /* "members" determines members of
even_group */
     members[i] = 2*i;
   };

   MPI_Group_incl(group_world, Neven, members, &even_group);
```

Fortran Example:

```
   include "mpif.h"
   implicit none
   integer group_world, odd_group, even_group
   integer i, p, Neven, Nodd, members(0:7), ierr

   call MPI_Comm_size(MPI_COMM_WORLD, p, ierr)
   call MPI_Comm_group(MPI_COMM_WORLD,group_world,ierr)

   Neven = (p + 1)/2   ! processes of MPI_COMM_WORLD are divided
   Nodd = p - Neven    ! into odd- and even-numbered groups
   do i=0,Neven - 1    ! "members" determines members of even_group
     members(i) = 2*i
   enddo

   call MPI_Group_incl(group_world, Neven, members, even_group, ierr)
```

In the above example, a new group is created whose members are the even-numbered processes of the communicator MPI_COMM_WORLD. In the new communicator, the group members are ordered, with stride 1, in ascending order (0, 1, 2, ..., Neven-1). They are associated, one on one, with processes of the old group as specified by the array (members(0), members(1), ..., members(Neven-1)). In this example, the old group consists of all processes of MPI_COMM_WORLD while the members array picks out the old group's even processes, i.e., members(0) = 0, members(1) = 2, members(2) = 4, and so on.

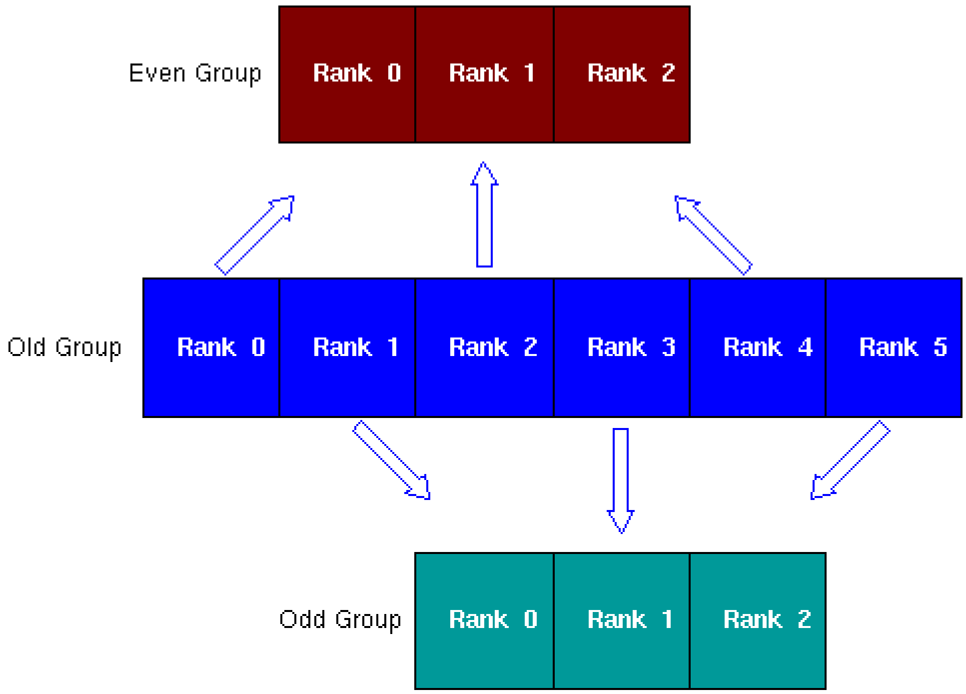


Figure 7.1. Graphical representation of MPI_GROUP_INCL example

Note:

- The position of the calling process in the new group is defined by the members array. Specifically, calling process number "members(i)" has rank "i" in new_group with "i" ranging from "0" to "count - 1".
- The same task could also be accomplished by a similar routine, MPI_GROUP_EXCL.
- If count = 0, new_group has the value MPI_GROUP_EMPTY.
- Two groups may have identical members but in different orders by how the members array is defined.

7.3. MPI_GROUP_EXCL

MPI_GROUP_EXCL

The MPI_GROUP_EXCL routine creates a new group from an existing group and specifies member processes (by exclusion).

C:

```
   int MPI_Group_excl( MPI_Group group, int count, int *nonmembers,
MPI_Group *new_group )
```

Fortran:

```
   MPI_GROUP_EXCL(GROUP, COUNT, NONMEMBERS, NEW_GROUP, IERR)
```

Variable Name	C Type	Fortran Type	In/Out	Description
group	MPI_Group	INTEGER	Input	Group handle
count	int	INTEGER	Input	Number of processes in nonmembers
nonmembers	int *	INTEGER	Output	Array of size count defining process ranks to be excluded
new_group	MPI_Group *	INTEGER	Output	Group handle
ierr	See (*)	INTEGER	Output	Error flag

* For C, the function returns an int error flag.

C Example:

```
  #include "mpi.h"
  MPI_Group group_world, odd_group, even_group;
  int i, p, Neven, Nodd, nonmembers[8], ierr;

  MPI_Comm_size(MPI_COMM_WORLD, &p);
  MPI_Comm_group(MPI_COMM_WORLD, &group_world);

  Neven = (p+1)/2;    /* processes of MPI_COMM_WORLD are divided */
  Nodd = p - Neven;   /* into odd- and even-numbered groups */
  for (i=0; i<Neven; i++) {   /* "nonmembers" are even-numbered procs
*/
    nonmembers[i] = 2*i;
  };

 MPI_Group_excl(group_world, Neven, nonmembers, &odd_group);
```

Fortran Example:

```
  include "mpif.h"
  implicit none
```

```
integer group_world, odd_group, even_group
integer i, p, Neven, Nodd, nonmembers(0:7), ierr

call MPI_Comm_size(MPI_COMM_WORLD, p, ierr)
call MPI_Comm_group(MPI_COMM_WORLD, group_world, ierr)

Neven = (p + 1)/2    ! processes of MPI_COMM_WORLD are divided
Nodd = p - Neven     ! into odd- and even-numbered groups
do i=0,Neven - 1     ! "nonmembers" are even-numbered procs
  nonmembers(i) = 2*i
enddo

call MPI_Group_excl(group_world, Neven, nonmembers, odd_group, ierr)
```

In the above example, a new group is created whose members are the odd-numbered processes of the communicator MPI_COMM_WORLD. Contrary to MPI_GROUP_INCL, this routine takes the **complement** of nonmembers (from MPI_COMM_WORLD) as group members. The new group's members are ranked in ascending order (0, 1, 2, ..., Nodd-1). In this example, these correspond to the process ranks (1, 3, 5, ...) in the old group. Count is the size of the exclusion list, ***i.e.,*** size of nonmembers.

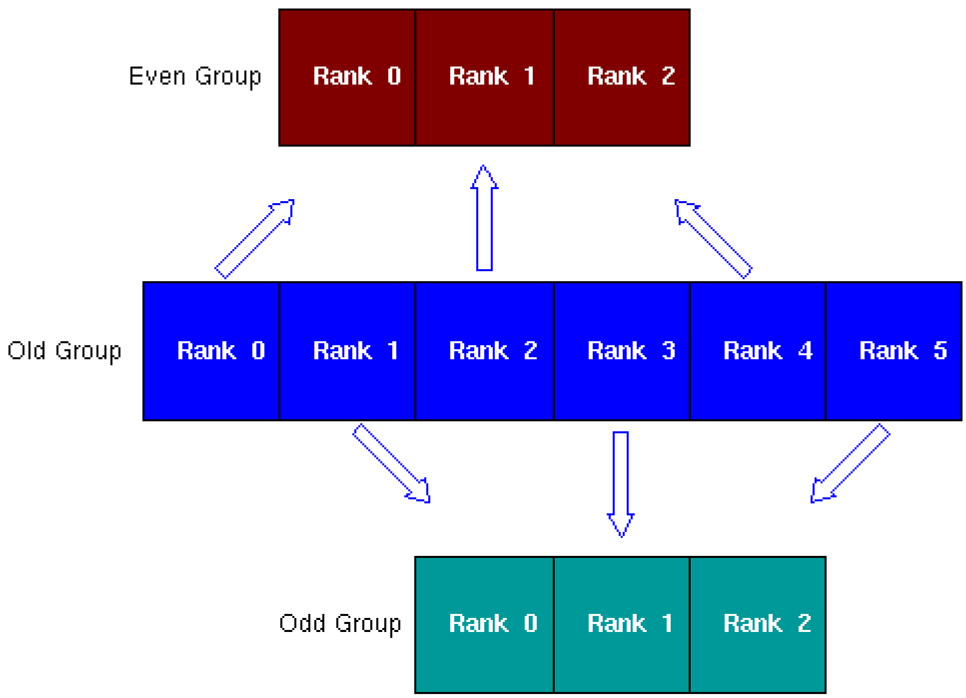


Figure 7.2. Graphical representation of MPI_Group_excl example

Note:

- The position of the calling process in the new group is in accordance with its relative ranking in the old group - skipping over the ones to be excluded.

Unlike MPI_GROUP_INCL, the order of nonmembers has no effect on the ranking order of the NEW_GROUP.

- The same task could also be accomplished by a similar routine, MPI_GROUP_INCL.
- If count = 0, ***i.e.,*** no process is excluded, NEW_GROUP is identical to OLD_GROUP.
- Ranks to be excluded (as defined by nonmembers) must be valid rank numbers in the OLD_GROUP. Otherwise, error will result.
- Nonmembers array must contain only distinct ranks in OLD_GROUP. Otherwise, error will result.

7.4. MPI_GROUP_RANK

MPI_GROUP_RANK

The MPI_GROUP_RANK routine queries the group rank of the calling process.

C:

```
int MPI_Group_rank( MPI_Group group, int *rank )
```

Fortran:

```
MPI_Group_rank( GROUP, RANK, IERR )
```

Variable Name	C Type	Fortran Type	In/Out	Description
group	MPI_Group	INTEGER	Input	Group handle
rank	int *	INTEGER	Output	Calling process rank
ierr	See (*)	INTEGER	Output	Error flag

* For C, the function returns an int error flag.

C Example:

```
#include "mpi.h"
MPI_Group group_world, worker_group;
int i, p, ierr, group_rank;

MPI_Comm_size(MPI_COMM_WORLD, &p);
MPI_Comm_group(MPI_COMM_WORLD, &group_world);
MPI_Group_excl(group_world, 1, 0, &worker_group);
```

```
MPI_Group_rank(worker_group, &group_rank);
```

Fortran Example:

```
include "mpif.h"
implicit none
integer group_world, worker_group
integer i, p, ierr, group_rank

call MPI_Comm_size(MPI_COMM_WORLD, p, ierr)
call MPI_Comm_group(MPI_COMM_WORLD, group_world, ierr)
call MPI_Group_excl(group_world, 1, 0, worker_group, ierr)

call MPI_Group_rank(worker_group, group_rank, ierr)
```

In the above example, first a new worker group is created whose members are all but process 0 of the group MPI_COMM_WORLD. Then a query is made for the group rank. The ranks of MPI_COMM_WORLD have the range (0, 1, 2, ..., p-1). For this simple example, the rank range of the new group, worker_group, is (0, 1, 2, ..., p-2) as it has one less member (i.e., process 0) than MPI_COMM_WORLD. Consequently, the calling process' corresponding rank number in the new group would be 1 smaller. For instance, if the calling process is "i" (which is the rank number in the MPI_COMM_WORLD group), the corresponding rank number in the new group would be "i-1". Note, however, that if the calling process is process 0 which does not belong to worker_group, MPI_GROUP_RANK would return the value of MPI_UNDEFINED for group_rank, indicating that it is not a member of the worker_group. For other arrangements, the rank number of the calling process in the new group may be less straightforward. The use of MPI_GROUP_RANK eliminates the need to keep track of that.

Note:

- It returns MPI_UNDEFINED (a negative number) if current process does not belong to group.
- MPI_UNDEFINED is implementation-dependent. For instance
    - MPI_UNDEFINED = -3 for SGI's MPI
    - MPI_UNDEFINED = -32766 for MPICH
- To check if calling process belongs to group; use code something like this to ensure portability:

```
      if (group_rank .eq. MPI_UNDEFINED) then
c  group_rank does not belong to group
        ...
      else
c  group_rank belongs to group
        ...
      endif
```

7.5. MPI_GROUP_FREE

MPI_GROUP_FREE

The MPI_GROUP_FREE routine returns a group to the system when it is no longer needed.

C:

```
int MPI_Group_free( MPI_Group *group )
```

Fortran:

```
MPI_GROUP_FREE( GROUP, IERR )
```

Variable Name	C Type	Fortran Type	In/Out	Description
group	MPI_Group *	INTEGER	Output	Group handle
ierr	See (*)	INTEGER	Output	Error flag

* For C, the function returns an int error flag.

C Example:

```
#include "mpi.h"
MPI_Group group_world, worker_group;
int i, p, ierr, group_rank;

MPI_Comm_size(MPI_COMM_WORLD, &p);
MPI_Comm_group(MPI_COMM_WORLD, &group_world);
MPI_Group_excl(group_world, 1, 0, &worker_group);
MPI_Group_rank(worker_group, &group_rank);

MPI_Group_free(worker_group);
```

Fortran Example:

```
include "mpif.h"
implicit none
integer group_world, worker_group
integer i, p, ierr, group_rank
```

```
   call MPI_Comm_size(MPI_COMM_WORLD, p, ierr)
   call MPI_Comm_group(MPI_COMM_WORLD, group_world, ierr)
   call MPI_Group_excl(group_world, 1, 0, worker_group, ierr)
   call MPI_Group_rank(worker_group, group_rank, ierr)

   call MPI_Group_free(worker_group, ierr)
```

In the above example, first a new worker group is created whose members are all but process 0 of the original group (MPI_COMM_WORLD). Then a query is made for the group rank. Finally, MPI_GROUP_FREE is called to return worker_group to the system.

Note:

- Freeing a group does not free the communicator to which it belongs.
- An MPI_COMM_FREE routine exists to free a communicator.

7.6. MPI_COMM_CREATE

MPI_COMM_CREATE

The MPI_COMM_CREATE routine creates a new communicator to include specific processes from an existing communicator.

C:

```
   int MPI_Cart_create( MPI_Comm old_comm, MPI_Group group, MPI_Comm
*new_comm )
```

Fortran:

```
   MPI_COMM_CREATE( OLD_COMM, GROUP, NEW_COMM, IERR )
```

Variable Name	C Type	Fortran Type	In/Out	Description
old_comm	MPI_Comm	INTEGER	Input	Communicator handle
group	MPI_Group	INTEGER	Input	Group handle
new_comm	MPI_Comm *	INTEGER	Output	Communicator handle
ierr	See (*)	INTEGER	Output	Error flag

* For C, the function returns an int error flag.

C Example:

```
  #include "mpi.h"
  MPI_Comm  comm_world, comm_worker;
  MPI_Group group_world, group_worker;
  int ierr;

  comm_world = MPI_COMM_WORLD;
  MPI_Comm_group(comm_world, &group_world);
  MPI_Group_excl(group_world, 1, 0, &group_worker);  /* process 0 not
member */

  MPI_Comm_create(comm_world, group_worker, &comm_worker);
```

Fortran Example:

```
  include "mpif.h"
  integer comm_world, group_world, comm_worker, group_worker, ierr

  comm_world = MPI_COMM_WORLD
  call MPI_Comm_group(comm_world, group_world, ierr)
  call MPI_Group_excl(group_world, 1, 0, group_worker, ierr)  ! process
0 not member

  call MPI_Comm_create(comm_world, group_worker, comm_worker, ierr)
```

In the above example, first the MPI_COMM_WORLD communicator's group handle is identified. Then a new group, group_worker, is created. This group is defined to include all but process 0 of MPI_COMM_WORLD as members by way of MPI_GROUP_EXCL. Finally, MPI_COMM_CREATE is used to create a new communicator whose member processes are those of the new group just created. With this new communicator, message passing can now proceed among its member processes.

Note:

- MPI_COMM_CREATE is a collective communication routine; it must be called by all processes of old_comm and all arguments in the call must be the same for all processes. Otherwise, error results.
- MPI_COMM_CREATE returns MPI_COMM_NULL to processes that are not in group.
- Upon completion of its task, the created communicator may be released by calling MPI_COMM_FREE.

7.7. MPI_COMM_SPLIT

MPI_COMM_SPLIT

The MPI_COMM_SPLIT routine forms new communicators from an existing one.

Many scientific and engineering computations deal with matrices or grids - especially Cartesian grids - consisting of rows and columns. This, in turn, precipitates a need to map processes logically into similar grid geometries. Furthermore, they may need to be dealt with in less traditional ways. For example, instead of dealing with individual rows, it may be advantageous or even necessary to deal with groups of rows or even more generally, other arbitrary configurations. MPI_COMM_SPLIT permits the creation of new communicators with such flexibilities.

The input variable *color* identifies the group while the *key* variable specifies a group member.

C:

```
   int MPI_Comm_split( MPI_Comm old_comm, int color, int key, MPI_Comm
*new_comm )
```

Fortran:

```
   MPI_COMM_SPLIT( OLD_COMM, COLOR, KEY, NEW_COMM, IERR )
```

Variable Name	C Type	Fortran Type	In/Out	Description
old_comm	MPI_Comm	INTEGER	Input	Communicator handle
color	int	INTEGER	Input	Provides process grouping classification; processes with same color in same group
key	int	INTEGER	Input	Within each group (color), "key" provides control for rank designation within group
new_comm	MPI_Comm *	INTEGER	Output	Communicator handle
ierr	See (*)	INTEGER	Output	Error flag

* For C, the function returns an int error flag.

Example: For a 2D logical grid, create subgrids of rows and columns

C :

```
/* logical 2D topology with nrow rows and mcol columns */
      irow = Iam/mcol;        /* logical row number */
      jcol = mod(Iam, mcol);  /* logical column number */
      comm2D = MPI_COMM_WORLD;

      MPI_Comm_split(comm2D, irow, jcol, row_comm);
      MPI_Comm_split(comm2D, jcol, irow, col_comm);
```

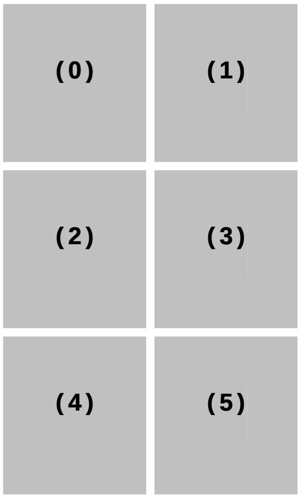
Fortran :

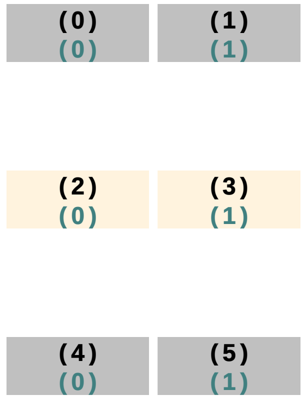
```
c**logical 2D topology with nrow rows and mcol columns
      irow = Iam/mcol       !! logical row number
      jcol = mod(Iam, mcol) !! logical column number
      comm2D = MPI_COMM_WORLD

      call MPI_Comm_split(comm2D, irow, jcol, row_comm, ierr)
      call MPI_Comm_split(comm2D, jcol, irow, col_comm, ierr)
```

To demonstrate the results of this example, say that we have 6 processes (0, 1, ..., 5) at our disposal. Mathematically (and topologically), it may be desirable to think of these processes being arranged in a 3-by-2 logical grid, as shown in Figure 7.3(a) below. The number in parentheses represents the rank number associated with the logical grid. Irow and jcol, both functions of the calling process number, **Iam**, are defined as the row and column numbers, respectively. A tabulation of **irow** and **jcol** versus **Iam** is shown below:

Iam	0	1	2	3	4	5
irow	0	0	1	1	2	2
jcol	0	1	0	1	0	1

The first MPI_COMM_SPLIT call specifies **irow** as the "color" (or group) with **jcol** as the "key" (or distinct member identity within group). This results in processes on each row classified as a separate group, as shown in Figure 7.3(b). The second MPI_COMM_SPLIT call, on the other hand, defines **jcol** as the color and **irow** as the key. This joins all processes in a column as belonging to a group, as shown in Figure 7.3(c).





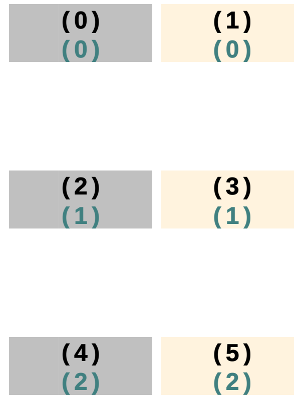


Figure 7.3(a). 2D logical Grid

Figure 7.3(b). 3 Row Subgrids

Figure 7.3(c). 2 Column Subgrids

In this example, the rank numbers (shown above in parentheses) are assigned using C's "row-major" rule. In Figure 7.3 (a), the numbers in black represent the "old" rank numbers (see) while those in green denote the rank numbers within individual row groups, Figure 7.3 (b), and column groups, Figure 7.3 (c).

Note that in the above, we chose to think of the logical grids as two-dimensional; they could very well be thought of as shown in Figures 7.3 (d) - (f).

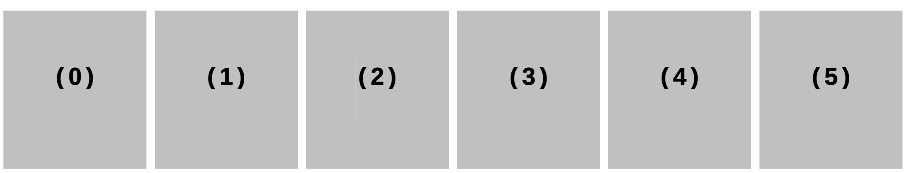


Figure 7.3 (d). 1D logical Grid

Figure 7.3 (e). 3 Subgrids

Figure 7.3 (f). 2 Subgrids

The following example of MPI_COMM_SPLIT splits the rows into two groups of rows: the first consists of rows 1 and 2 while the second represents row 3. The code fragment that does that is

```
   C**MPI_Comm_split is more general than MPI_Cart_sub
   C**simple example of 6 processes divided into 2 groups;
   C**1st 4 belongs to group 0 and remaining two to group 1
         group = Iam/4               ! group0:0,1,2,3;
group1:4,5
         index = Iam - row_group*4   ! group0:0,1,2,3;
group1:0,1
         call MPI_Comm_split(comm2D, group, index,
        &                    row_comm, ierr)
```

The above table is illustrated in the figure below and the output of the example code for this particular arrangement is shown in Figure 7.3 (g).

(0) (0)	(1) (1)
(2) (2)	(3) (3)
(4) (0)	(5) (1)

Figure 7.3 (g). Output for the MPI_COMM_SPLIT example.

Note:

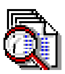
- This routine is similar to MPI_CART_SUB. However, MPI_COMM_SPLIT is more general than MPI_CART_SUB. MPI_COMM_SPLIT creates a logical grid and is referred to by its linear rank number; MPI_CART_SUB creates a Cartesian grid and rank is referred to by Cartesian coordinates. For instance, in a 2D Cartesian grid, a grid cell is known by its (irow,jcol) index pair.
- MPI_COMM_SPLIT is a collective communication routine and hence all processes of old_comm must call routine. Unlike many collective communication routines, however, key and color are allowed to differ among all processes of old_comm.
- Processes in old_comm not part of any new group must have color defined as MPI_UNDEFINED. The corresponding returned NEW_COMM for these processes have value MPI_COMM_NULL.

- If two or more processes have the same key, the resulting rank numbers of these processes in the new communicator are ordered relative to their respective ranks in the old communicator. Recall that ranks in a communicator are by definition unique and ordered as (0, 1, 2, ..., p-1). An example of this is given in the upcoming Section 7.8.2. A corollary to this is that if no specific ordering is required of the new communicator, setting key to a constant results in MPI to order their new ranks following their relative rank order in the old communicator.

7.8. Communicators Examples

Communicators Examples

This section provides examples showing the application of the MPI communicators routines. The first example covers the routines MPI_GROUP_INCL, MPI_GROUP_EXCL, MPI_GROUP_RANK and MPI_GROUP_FREE. The second example demonstrates two different applications of MPI_COMM_SPLIT.

A zip file containing the C and Fortran code for the examples given in this chapter is available for download.

7.8.1. Example on Usages of Group Routines

Example on Usages of Group Routines

The objective of this example is to divide the member processes of MPI_COMM_WORLD into two new groups. One group consists of all odd-numbered processes and the other all even-numbered processes. A table is then printed to show whether a process belongs to the odd or even group.

Fortran Code

```
   PROGRAM Group_example
   implicit none
   integer i, j, Iam, p, ierr, group_world
   integer Neven, Nodd, members(0:7)
   integer even_group, even_rank, odd_group, odd_rank
   include "mpif.h"  !! This brings in pre-defined MPI constants, ...
!**Starts MPI processes ...
   call MPI_Init(ierr)                             !! starts MPI
   call MPI_Comm_rank(MPI_COMM_WORLD, Iam, ierr)  !! get current
process id
   call MPI_Comm_size(MPI_COMM_WORLD, p, ierr)    !! get number of
processes
```

```
   if( p .ne. 6 ) then
     if (Iam .eq. 0) then
       write(*,*)'*****************************************'
       write(*,*)'*** THIS SAMPLE PROGRAM IS VALID FOR ***'
       write(*,*)'*** SIX (6) PROCESSES ONLY           ***'
       write(*,*)'*****************************************'
     endif
     call MPI_Abort(MPI_COMM_WORLD, ierr)
   endif
   Neven = (p + 1)/2     ! processes of MPI_COMM_WORLD are divided
   Nodd = p - Neven      ! into odd- and even-numbered groups
   members(0) = 2        ! proc 2 mapped to rank 0 in even_group
   members(1) = 0        ! proc 0 mapped to rank 1 in even_group
   members(2) = 4        ! proc 4 mapped to rank 2 in even_group
!
   call MPI_Comm_group(MPI_COMM_WORLD, group_world, ierr)
   call MPI_Group_incl(group_world, Neven, members, even_group, ierr)
   call MPI_Group_excl(group_world, Neven, members,  odd_group, ierr)
!
   call MPI_Barrier(MPI_COMM_WORLD, ierr)
   if(Iam .eq. 0) then
     write(*,*)
     write(*,*)'MPI_Group_incl/excl Usage Example'
     write(*,*)
     write(*,*)'Number of processes is     ', p
     write(*,*)'Number of odd processes  is', Nodd
     write(*,*)'Number of even processes is', Neven
     write(*,*)'members(0) is assigned rank',members(0)
     write(*,*)'members(1) is assigned rank',members(1)
     write(*,*)'members(2) is assigned rank',members(2)
     write(*,*)
     write(*,*)'MPI_UNDEFINED is set to    ', MPI_UNDEFINED
     write(*,*)
     write(*,*)' Current     even      odd'
     write(*,*)'    rank     rank     rank'
   endif
   call MPI_Barrier(MPI_COMM_WORLD, ierr)
!
   call MPI_Group_rank(even_group, even_rank, ierr)
   call MPI_Group_rank( odd_group,  odd_rank, ierr)
!
   write(*,'(3i9)')Iam, even_rank, odd_rank
!
   call MPI_Group_free( odd_group, ierr)
   call MPI_Group_free(even_group, ierr)
!
   call MPI_Finalize(ierr)                  !! let MPI finish up ...

   end
```

C Code

```
   #include "mpi.h"
   void main(int argc, char *argv[])
   {
      int Iam, p;
      int Neven, Nodd, members[6], even_rank, odd_rank;
      MPI_Group group_world, even_group, odd_group;
/* Starts MPI processes ... */
      MPI_Init(&argc, &argv);                          /* starts MPI */
      MPI_Comm_rank(MPI_COMM_WORLD, &Iam);  /* get current process id
*/
      MPI_Comm_size(MPI_COMM_WORLD, &p);    /* get number of processes
*/
      Neven = (p + 1)/2;     /* All processes of MPI_COMM_WORLD are
divided */
      Nodd = p - Neven;      /* into 2 groups, odd- and even-numbered
groups */
      members[0] = 2;
      members[1] = 0;
      members[2] = 4;
      MPI_Comm_group(MPI_COMM_WORLD, &group_world);
      MPI_Group_incl(group_world, Neven, members, &even_group);
      MPI_Group_excl(group_world, Neven, members,  &odd_group);

      MPI_Barrier(MPI_COMM_WORLD);
      if(Iam == 0) {
        printf("MPI_Group_incl/excl Usage Example\n");
        printf("\n");
        printf("Number of processes is %d\n", p);
        printf("Number of odd processes is %d\n", Nodd);
        printf("Number of even processes is %d\n", Neven);
        printf("\n");
        printf("     Iam     even      odd\n");
      }
      MPI_Barrier(MPI_COMM_WORLD);

      MPI_Group_rank(even_group, &even_rank);
      MPI_Group_rank( odd_group,  &odd_rank);
      printf("%8d %8d %8d\n",Iam, even_rank, odd_rank);

      MPI_Finalize();                  /* let MPI finish up ...  */
}
```

Output

A negative number indicates that the calling process does not belong to the group. As a matter of fact, in that case the rank number would be set to MPI_UNDEFINED, which is implementation dependent and has a value of "-3" for SGI's MPI and "-32766" for MPICH. It is important to emphasize here that

- The even-numbered group is generated with MPI_GROUP_INCL. The rank of the calling process in the even group is defined by the members array. In other words, calling process number "members(i)" is rank "i" in even_group. Note also that "i" starts from "0".

- The odd-numbered group is generated with MPI_GROUP_EXCL. The position of the calling process in the odd group, however, is in accordance with the relative ranking of the calling process in the old group - skipping over the ones to be excluded.

```
tonka:communicators/codes % mpirun -np 6 group_example

MPI_Group_incl/excl Usage Example

Number of processes is                6
Number of odd processes  is           3
Number of even processes is           3
members(0) is assigned rank           2
members(1) is assigned rank           0
members(2) is assigned rank           4

MPI_UNDEFINED is set to              -3

Current      even      odd
   rank      rank     rank
      0         1       -3
      3        -3        1
      1        -3        0
      2         0       -3
      4         2       -3
      5        -3        2
```

7.8.2. Example on Usages of MPI_COMM_SPLIT

Example on Usages of MPI_COMM_SPLIT

The objective of this example is to demonstrate the usage of MPI_COMM_SPLIT.

Fortran Code

```
   implicit none
   integer nrow, mcol, irow, jcol, i, j, ndim
   parameter (nrow=3, mcol=2, ndim=2)
   integer p, ierr, row_comm, col_comm, comm2D
   integer Iam, me, row_id, col_id, ndim
   integer row_group, row_key, map(0:5)
   data map/2,1,2,1,0,1/
   include "mpif.h"  !! This brings in pre-defined MPI constants, ...
c**Starts MPI processes ...
   call MPI_Init(ierr)                             !! starts MPI
   call MPI_Comm_rank(MPI_COMM_WORLD, Iam, ierr)  !! get current
process id
   call MPI_Comm_size(MPI_COMM_WORLD, p, ierr)    !! get number of
processes
```

```
   if(Iam .eq. 0) then
     write(*,*)
     write(*,*)'Example of MPI_Comm_split Usage'
     write(*,*)'Split 3x2 grid into 2 different communicators'
     write(*,*)'which correspond to 3 rows and 2 columns.'
     write(*,*)
     write(*,*)'    Iam    irow    jcol  row-id  col-id'
   endif
c**virtual topology with nrow rows and mcol columns
   irow = Iam/mcol       !! row number
   jcol = mod(Iam, mcol) !! column number
   comm2D = MPI_COMM_WORLD
   call MPI_Comm_split(comm2D, irow, jcol, row_comm, ierr)
   call MPI_Comm_split(comm2D, jcol, irow, col_comm, ierr)

   call MPI_Comm_rank(row_comm, row_id, ierr)
   call MPI_Comm_rank(col_comm, col_id, ierr)
   call MPI_Barrier(MPI_COMM_WORLD, ierr)

   write(*,'(9i8)')Iam,irow,jcol,row_id,col_id
   call MPI_Barrier(MPI_COMM_WORLD, ierr)

   if(Iam .eq. 0) then
     write(*,*)
     write(*,*)'Next, create more general communicator'
     write(*,*)'which consists of two groups :'
     write(*,*)'Rows 1 and 2 belongs to group 1 and row 3 is group 2'
     write(*,*)
   endif
C**MPI_Comm_split is more general than MPI_Cart_sub
C**simple example of 6 processes divided into 2 groups;
C**1st 4 belongs to group 1 and remaining two to group 2
   row_group = Iam/4     ! this expression by no means general
   row_key = Iam - row_group*4    ! group1:0,1,2,3; group2:0,1
   call MPI_Comm_split(comm2D, row_group, row_key,
  &                    row_comm, ierr)
   call MPI_Comm_rank(row_comm, row_id, ierr)
   write(*,'(9i8)')Iam,row_id
   call MPI_Barrier(MPI_COMM_WORLD, ierr)

   if(Iam .eq. 0) then
     write(*,*)
     write(*,*)'If two processes have same key, the ranks'
     write(*,*)'of these two processes in the new'
     write(*,*)'communicator will be ordered according'
     write(*,*)'to their order in the old communicator'
     write(*,*)' key = map(Iam); map = (2,1,2,1,0,1)'
     write(*,*)
   endif
C**MPI_Comm_split is more general than MPI_Cart_sub
C**simple example of 6 processes dirowided into 2 groups;
C**1st 4 belongs to group 1 and remaining two to group 2
   row_group = Iam/4     ! this expression by no means general
   row_key = map(Iam)
   call MPI_Comm_split(comm2D, row_group, row_key,
```

```
  &                    row_comm, ierr)
   call MPI_Comm_rank(row_comm, row_id, ierr)
   call MPI_Barrier(MPI_COMM_WORLD, ierr)
   write(*,'(9i8)')Iam,row_id

   call MPI_Finalize(ierr)                        !! let MPI finish up
...

   end
```

C code

```
#include "mpi.h"
void main(int argc, char *argv[])
{
      int mcol, irow, jcol, p;
      MPI_Comm row_comm, col_comm, comm2D;
      int Iam, row_id, col_id;
      int row_group, row_key, map[6];

/* Starts MPI processes ... */
      MPI_Init(&argc, &argv);                          /* starts MPI */
      MPI_Comm_rank(MPI_COMM_WORLD, &Iam);  /* get current process id
*/
      MPI_Comm_size(MPI_COMM_WORLD, &p);    /* get number of processes
*/

      map[0]=2; map[1]=1; map[2]=2; map[3]=1; map[4]=0; map[5]=1;
      mcol=2;   /* nrow = 3 */
      if(Iam == 0) {
        printf("\n");
        printf("Example of MPI_Comm_split Usage\n");
        printf("Split 3x2 grid into 2 different communicators\n");
        printf("which correspond to 3 rows and 2 columns.");
        printf("\n");
        printf("     Iam     irow     jcol   row-id   col-id\n");
      }
/* virtual topology with nrow rows and mcol columns */
      irow = Iam/mcol;        /* row number */
      jcol = Iam%mcol;        /* column number */
      comm2D = MPI_COMM_WORLD;
      MPI_Comm_split(comm2D, irow, jcol, &row_comm);
      MPI_Comm_split(comm2D, jcol, irow, &col_comm);

      MPI_Comm_rank(row_comm, &row_id);
      MPI_Comm_rank(col_comm, &col_id);
      MPI_Barrier(MPI_COMM_WORLD);

      printf("%8d %8d %8d %8d %8d\n",Iam,irow,jcol,row_id,col_id);
      MPI_Barrier(MPI_COMM_WORLD);

      if(Iam == 0) {
```

```
        printf("\n");
        printf("Next, create more general communicator\n");
        printf("which consists of two groups :\n");
        printf("Rows 1 and 2 belongs to group 1 and row 3 is group
2\n");
        printf("\n");
      }
/* MPI_Comm_split is more general than MPI_Cart_sub
   simple example of 6 processes divided into 2 groups;
   1st 4 belongs to group 1 and remaining two to group 2 */
      row_group = Iam/4;     /* this expression by no means general */
      row_key = Iam - row_group*4;    /* group1:0,1,2,3; group2:0,1 */
      MPI_Comm_split(comm2D, row_group, row_key, &row_comm);
      MPI_Comm_rank(row_comm, &row_id);
      printf("%8d %8d\n",Iam,row_id);
      MPI_Barrier(MPI_COMM_WORLD);

      if(Iam == 0) {
        printf("\n");
        printf("If two processes have same key, the ranks\n");
        printf("of these two processes in the new\n");
        printf("communicator will be ordered according'\n");
        printf("to their order in the old communicator\n");
        printf(" key = map[Iam]; map = (2,1,2,1,0,1)\n");
        printf("\n");
      }
/* MPI_Comm_split is more general than MPI_Cart_sub
   simple example of 6 processes dirowided into 2 groups;
   1st 4 belongs to group 1 and remaining two to group 2 */
      row_group = Iam/4;     /* this expression by no means general */
      row_key = map[Iam];
      MPI_Comm_split(comm2D, row_group, row_key, &row_comm);
      MPI_Comm_rank(row_comm, &row_id);
      MPI_Barrier(MPI_COMM_WORLD);
      printf("%8d %8d\n",Iam,row_id);

      MPI_Finalize();                  /* let MPI finish up ...  */
}
```

Output

```
   Example of MPI_Comm_split Usage
   Split 3x2 grid into 2 different communicators
   which correspond to 3 rows and 2 columns.

     Iam    irow    jcol   group    rank
       0       0       0       0       0
       2       1       0       0       1
       3       1       1       1       1
       5       2       1       1       2
       1       0       1       1       0
       4       2       0       0       2
```

```
Next, create more general communicator
which consists of two groups :
Rows 1 and 2 belongs to group 1 and row 3 is group 2

           new
  Iam     rank
    0        0
    3        3
    2        2
    1        1
    5        1
    4        0

If two processes have same key, the ranks
of these two processes in the new
communicator will be ordered according
to their order in the old communicator
key = map(Iam); map = (2,1,2,1,0,1)

           new
  Iam     rank
    1        0
    0        2
    4        0
    5        1
    3        1
    2        3
```

Graphically, the last two examples yield the following results where the numbers in black denote MPI_COMM_WORLD rank number and the numbers in green represent the rank number of the two respective new groups.

(0)	(1)	(2)	(3)	(4)	(5)
(0)	(1)	(2)	(3)	(0)	(1)

(0)	(1)	(2)	(3)	(4)	(5)
(2)	(0)	(3)	(1)	(0)	(1)

7.9. Self Test

Communicators Self Test

Now that you've finished this chapter, test yourself on what you've learned by taking the Self Test provided. Simply click on the Self Test link in the ACTION MENU above to get started.

7.10. Course Problem

Chapter 7 Course Problem

This chapter explained that it is possible to create and use different groups of processors and communicators. We will create and use a communicator other than MPI_COMM_WORLD in the next chapter.

For this chapter, a new version of the Course Problem is presented in which the average value each processor calculates when a target location is found, is calculated in a different manner. Specifically, the average will be calculated from the "neighbor" values of the target. This is a classic style of programming (called calculations with a *stencil*) used at important array locations. Stencil calculations are used in many applications including numerical solutions of differential equations and image processesing to name two. This new Course Problem will also entail more message passing between the searching processors because in order to calculate the average they will have to get values of the global array they do not have in their subarray.

Description

Our new problem still implements a parallel search of an integer array. The program should find all occurences of a certain integer which will be called the target. When a processor of a certain rank finds a target location, it should then calculate the average of

- The target value
- An element from the processor with rank one higher (the **"right"** processor). The right processor should send the first element from its local array.
- An element from the processor with rank one less (the **"left"** processor). The left processor should send the first element from its local array.

For example, if processor 1 finds the target at index 33 in its local array, it should get from processors 0 (left) and 2 (right) the first element of *their* local arrays. These three numbers should then be averaged.

In terms of right and left neighbors, you should visualize the four processors connected in a ring. That is, the left neighbor for P0 should be P3, and the right neighbor for P3 should be P0.

Both the target location and the average should be written to an output file. As usual, the program should read both the target value and all the array elements from an input file.

Exercise

Solve this new version of the Course Problem by modifying your code from Chapter 6. Specifically, change the code to perform the new method of calculating the average value at each target location.

Solution

When you have finished writing the code for this exercise, view our version of the Stencil Code.

---

8. Virtual Topologies

Virtual Topologies

Many computational science and engineering problems reduce at the end to either a series of matrix or some form of grid operations, be it through differential, integral or other methods. The dimensions of the matrices or grids are often determined by the physical problems. Frequently in multiprocessing, these matrices or grids are partitioned, or domain-decomposed, so that each partition (or subdomain) is assigned to a process. One such example is an m x n matrix decomposed into p q x n submatrices with each assigned to be worked on by one of the p processes. In this case, each process represents one distinct submatrix in a straightforward manner. However, an algorithm might dictate that the matrix be decomposed into a pxq logical grid, whose elements are themselves each an r x s matrix. This requirement might be due to a number of reasons: efficiency considerations, ease in code implementation, code clarity, to name a few. Although it is still possible to refer to each of these pxq subdomains by a linear rank number, it is obvious that a mapping of the linear process rank to a 2D virtual rank numbering would facilitate a much clearer and natural computational representation. To address the needs of this and other topological layouts, the MPI library provides two types of topology routines: Cartesian and graph topologies. Only Cartesian topology and the associated routines will be discussed in this chapter.

A zip file containing the C and Fortran examples given in this chapter is available for download.

8.1. MPI Topology Routines

Virtual Topology MPI Routines

Some of the MPI topology routines are

- MPI_CART_CREATE
- MPI_CART_COORDS
- MPI_CART_RANK
- MPI_CART_SHIFT
- MPI_CART_SUB
- MPI_CARTDIM_GET
- MPI_CART_GET
- MPI_CART_SHIFT

These routines are discussed in the following sections.

8.1.1. MPI_CART_CREATE

MPI_CART_CREATE

The MPI_CART_CREATE routine creates a new communicator using a Cartesian topology.

C:

```
int MPI_Cart_create(MPI_Comm old_comm, int ndims, int *dim_size,
                    int *periods, int reorder, MPI_Comm *new_comm)
```

Fortran:

```
MPI_CART_CREATE(OLD_COMM, NDIMS, DIM_SIZE, PERIODS, REORDER,
NEW_COMM, IERR)
```

Variable Name	C Type	Fortran Type	In/Out	Description
old_comm	MPI_Comm	INTEGER	Input	Communicator handle
ndims	int	INTEGER	Input	Number of dimensions
dim_size	int *	INTEGER	Output	Array of size ndims providing length in each dimension
periods	int *	LOGICAL	Input	Array of size ndims specifying periodicity status of each dimension
reorder	int	LOGICAL	Input	whether process rank reordering by MPI is permitted
new_comm	MPI_Comm *	INTEGER	Output	Communicator handle

ierr	See (*)	INTEGER	Output	Error flag

* For C, the function returns an int error flag.

C Example:

```
#include "mpi.h"
 MPI_Comm old_comm, new_comm;
 int ndims, reorder, periods[2], dim_size[2];

 old_comm = MPI_COMM_WORLD;
 ndims = 2;           /*  2D matrx/grid */
 dim_size[0] = 3;     /* rows */
 dim_size[1] = 2;     /* columns */
 periods[0] = 1;      /* row periodic (each column forms a ring) */
 periods[1] = 0;      /* columns nonperiodic */
 reorder = 1;         /* allows processes reordered for efficiency */

 MPI_Cart_create(old_comm, ndims, dim_size,
              periods, reorder, &new_comm);
```

Fortran Example:

```
 include "mpif.h"
 integer old_comm, new_comm, ndims, ierr
 integer dim_size(0:1)
 logical periods(0:1), reorder

 old_comm = MPI_COMM_WORLD
 ndims = 2            ! 2D grid
 dim_size(0) = 3      ! rows
 dim_size(1) = 2      ! columns
 periods(0) = .true.  ! row periodic (each column forms a ring)
 periods(1) = .false. ! columns  nonperiodic
 reorder = .true.     ! allows processes reordered for efficiency

 call MPI_Cart_create(old_comm, ndims, dim_size,
&       periods, reorder, new_comm, ierr)
```

In the above example we use MPI_CART_CREATE to map (or rename) 6 processes from a linear ordering (/i> 0,1,2,3,4,5) into a two-dimensional matrix ordering of 3 rows by 2 columns (*i.e.*, (0,0), (0,1), ..., (2,1)). Figure 8.1 (a) below depicts the resulting Cartesian grid representation for the processes. The index pair "i,j" represent row "i" and column "j". The corresponding (linear) rank number is enclosed in parentheses.

0,0 (0)	0,1 (1)
1,0 (2)	1,1 (3)
2,0 (4)	2,1 (5)

Figure 8.1 (a). Cartesian Grid

With processes renamed in a 2D grid topology, we are able to assign or distribute work, or distinguish among the processes by their grid topology rather than by their linear process ranks. Additionally, we have imposed periodicity along the first dimension (periods(0)=.true.), which means that any reference beyond the first or last entry of any column will be wrapped around cyclically. For example, row index *i = -1*, due to periodicity, corresponds to *i = 2*. Similarly, *i = -2* maps onto *i = 1*. Likewise, *i = 3* is the same as *i = 0*. No periodicity is imposed on the second dimension (periods(1)=.false.). Any reference to the column index outside of its defined range (in this case 0 to 1) will result in a negative process rank (equal to MPI_PROC_NULL which is -1), which signifies that it is out of range.

Similarly, if periodicity was defined only for the column index (*i.e.,* periods(0)=.false.; periods(1)=.true.), each row would wrap around itself. Each of the above two 2D cases may be viewed graphically as a cylinder; the periodic dimension forms the circumferential surface while the nonperiodic dimension runs parallel to the cylindrical axis. If both dimensions are periodic, the grid resembles a torus. The effects of periodic columns and periodic rows are depicted in Figures 8.1 (b) and (c), respectively. The tan-colored cells indicate cyclic boundary condition in effect.

	-1,0 (4)	-1,1 (5)	
0,-1(-1)	0,0 (0)	0,1 (1)	0,2(-1)
1,-1(-1)	1,0 (2)	1,1 (3)	1,2 (-1)
2,-1(-1)	2,0 (4)	2,1 (5)	2,2 (-1)
	3,0 (0)	3,1 (1)	

Figure 8.1 (b). periods(0)=.true.;periods(1)=.false.

	-1,0 (-1)	*-1,1 (-1)*	
0,-1 (1)	*0, 0 (0)*	*0, 1 (1)*	*0, 2 (0)*
1,-1 (3)	*1, 0 (2)*	*1, 1 (3)*	*1, 2 (2)*
2,-1 (5)	*2, 0 (4)*	*2, 1 (5)*	*2, 2 (4)*
	3,0 (-1)	*3,1 (-1)*	

Figure 8.1 (c). periods(0)=.false.;periods(1)=.true.

Finally, note that while the processes are arranged logically as a cartesian topology, the processors corresponding to these processes may in fact be scattered physically - even within a shared-memory machine. If reorder

is set to ".true." in Fortran (or "1" in C), MPI *may* reorder the process ranks in the new communicator (for potential gain in performance due to, say, the physical proximities of the processes assigned). If reorder

is ".false." (or "0" in C), the process rank in the new communicator is identical to its rank in the old communicator.

While having the processes laid out in the Cartesian topology help you write code that's conceivably more readable, many MPI routines recognize only rank number and hence knowing the relationship between ranks and Cartesian coordinates (as shown in the figures above) is the key to exploit the topology for computational expediency. In the following sections, we will discuss two subroutines that provide this information. They are

- MPI_CART_COORDS
- MPI_CART_RANK

Note:

- MPI_CART_CREATE is a collective communication function (see Chapter 6 - Collective Communications). It must be called by all processes in the group. Like other collective communication routines, MPI_CART_CREATE uses blocking communication. However, it is not required to be synchronized among processes in the group and hence is implementation dependent.
- If the total size of the Cartesian grid is smaller than available processes, those processes not included in the new communicator will return MPI_COMM_NULL.

- If the total size of the Cartesian grid is larger than available processes, the call results in error.

8.1.2. MPI_CART_COORDS

MPI_CART_COORDS

The MPI_CART_COORDS routine returns the corresponding Cartesian coordinates of a (linear) rank in a Cartesian communicator.

C:

```
   int MPI_Cart_coords( MPI_Comm comm, int rank, int maxdims, int
*coords )
```

Fortran:

```
   MPI_CART_COORDS( COMM, RANK, MAXDIMS, COORDS, IERR )
```

Variable Name	C Type	Fortran Type	In/Out	Description
comm	MPI_Comm	INTEGER	Input	Communicator handle
rank	int	INTEGER	Input	Calling process rank
maxdims	int	INTEGER	Input	Number of dimensions in cartesian topology
coords	int *	INTEGER	Output	Corresponding cartesian coordinates of rank
ierr	See (*)	INTEGER	Output	Error flag

* For C, the function returns an int error flag.

C Example:

```
   MPI_Cart_create(old_comm, ndims, dim_size,
            periods, reorder, &new_comm);  /* creates communicator */

   if(Iam == root) {   /* only want to do this on one process */
      for (rank=0; rank<p; rank++) {
        MPI_Cart_coords(new_comm, rank, coords);
        printf("%d, %d\n ",rank, coords);
      }
```

```
   }
```

Fortran Example:

```
  call MPI_Cart_create(old_comm,ndims,dim_size,
 &       periods,reorder,new_comm,ierr)  ! creates communicator

  if(Iam .eq. root) then    !! only want to do this on one process
    do rank=0,p-1
      call MPI_Cart_coords(new_comm, rank, coords, ierr)
      write(*,*)rank, coords
    enddo
  endif
```

In the above example, a Cartesian communicator is created first. Repeated applications of MPI_CART_COORDS for all process ranks (input) produce the mapping table, shown in Figure 8.2, of process ranks and their corresponding Cartesian coordinates (output).

0,0 (0)	*0,1 (1)*
1,0 (2)	*1,1 (3)*
2,0 (4)	*2,1 (5)*

Figure 8.2. Cartesian Grid

Note:

- This routine is the reciprocal of MPI_CART_RANK.
- Querying for coordinates of ranks in new_comm is not robust; querying for an out-of-range rank results in error.

8.1.3. MPI_CART_RANK

MPI_CART_RANK

The MPI_CART_RANK routine returns the corresponding process rank of the Cartesian coordinates of a Cartesian communicator.

C:

```
int MPI_Cart_rank( MPI_Comm comm, int *coords, int *rank )
```

Fortran:

```
MPI_CART_RANK( COMM, COORDS, RANK, IERR )
```

Variable Name	C Type	Fortran Type	In/Out	Description
comm	MPI_Comm	INTEGER	Input	Cartesian communicator handle
coords	int *	INTEGER	Input	Array of size ndims specifying Cartesian coordinates
rank	int *	INTEGER	Output	Process rank of process specified by its Cartesian coordinates, coords
ierr	See (*)	INTEGER	Output	Error flag

* For C, the function returns an int error flag.

C Example:

```
MPI_Cart_create(old_comm, ndims, dim_size,
        periods, reorder, &new_comm);

if(Iam == root) {       /* only want to do this on one process */
  for (i=0; i<nv; i++) {
    for (j=0; j<mv; j++)  {
      coords[0] = i;
      coords[1] = j;
      MPI_Cart_rank(new_comm, coords, &rank);
      printf("%d, %d, %d\n",coords[0],coords[1],rank);
    }
  }
}
```

Fortran Example:

```
  call MPI_Cart_create(old_comm,ndims,dim_size,
 &       periods,reorder,new_comm,ierr)

  if(Iam .eq. root) then    !! only want to do this on one process
    do i=0,nv-1
      do j=0,mv-1
         coords(0) = i
         coords(1) = j
         call MPI_Cart_rank(new_comm,coords,rank,ierr)
         write(*,*)coords,rank
      enddo
```

```
    enddo
  endif
```

Once a Cartesian communicator has been established, repeated applications of MPI_CART_RANK for all possible values of the cartesian coordinates produce a correlation table of the Cartesian coordinates and their corresponding process ranks.

Shown in Figure 8.3 below is the resulting Cartesian topology (grid) where the index pair "i,j" represent row "i" and column "j". The number in parentheses represents the rank number associated with the Cartesian coordinates.

0,0 (0)	*0,1 (1)*
1,0 (2)	*1,1 (3)*
2,0 (4)	*2,1 (5)*

Figure 8.3. Cartesian Grid

Note:

- This routine is the reciprocal of MPI_CART_COORDS.
- Querying for rank number of out-of-range coordinates along the dimension in which periodicity is not enabled is not safe (i.e., results in error).

8.1.4. MPI_CART_SUB

MPI_CART_SUB

The MPI_CART_SUB routine creates new communicators for subgrids of up to (N-1) dimensions from an N-dimensional Cartesian grid.

Often, after we have created a Cartesian grid, we wish to further group elements of this grid into subgrids of lower dimensions. Typical operations requiring subgrids include reduction operations such as the computation of row sums, column extremums. For instance, the subgrids of a 2D Cartesian grid are 1D grids of the individual rows or columns. Similarly, for a 3D Cartesian grid, the subgrids can either be 2D or 1D.

C:

```
  int MPI_Cart_sub( MPI_Comm old_comm, int *belongs, MPI_Comm
*new_comm )
```

Fortran:

```
MPI_CART_SUB( OLD_COMM, BELONGS, NEW_COMM, IERR )
```

Variable Name	C Type	Fortran Type	In/Out	Description
old_comm	MPI_Comm	INTEGER	Input	Cartesian communicator handle
belongs	int *	INTEGER	Input	Array of size ndims specifying whether a dimension belongs to new_comm
new_comm	MPI_Comm *	INTEGER	Output	Cartesian communicator handle
ierr	See (*)	INTEGER	Output	Error flag

* For C, the function returns an int error flag.

C Example:

For a 2D Cartesian grid, create subgrids of rows and columns. Create Cartesian topology for processes.

```
/* Create 2D Cartesian topology for processes */
  MPI_Cart_create(MPI_COMM_WORLD, ndim, dims,
             period, reorder, &comm2D);
  MPI_Comm_rank(comm2D, &id2D);
  MPI_Cart_coords(comm2D, id2D, ndim, coords2D);
/* Create 1D row subgrids */
  belongs[0] = 0;
  belongs[1] = 1;     ! this dimension belongs to subgrid
  MPI_Cart_sub(comm2D, belongs, &commrow);
/* Create 1D column subgrids */
  belongs[0] = 1;      /* this dimension belongs to subgrid */
  belongs[1] = 0;
  MPI_Cart_sub(comm2D, belongs, &commcol);
```

Fortran Example:

For a 2D Cartesian grid, create subgrids of rows and columns. Create Cartesian topology for processes.

```
!Create 2D Cartesian topology for processes
      call MPI_Cart_create(MPI_COMM_WORLD, ndim, dims,
     &       period, reorder, comm2D, ierr)
      call MPI_Comm_rank(comm2D, id2D, ierr)
      call MPI_Cart_coords(comm2D, id2D, ndim, coords2D, ierr)
!Create 1D row subgrids
      belongs(0) = .false.
      belongs(1) = .true.  ! this dimension belongs to subgrid
```

```
        call MPI_Cart_sub(comm2D, belongs, commrow, ierr)
   !Create 1D column subgrids
        belongs(0) = .true.   ! this dimension belongs to subgrid
        belongs(1) = .false.
        call MPI_Cart_sub(comm2D, belongs, commcol, ierr)
```

Shown in Figure 8.4 (a) below is a 3-by-2 Cartesian topology. Figure 8.4 (b) shows the resulting row subgrids, while Figure 8.4 (c) shows the corresponding column subgrids. In black, the first row of numbers in each cell lists the 2D Cartesian coordinate index pair "i,j" and the associated rank number. On the second row, and in green, are shown the 1D subgrid Cartesian coordinates and the subgrid rank number (in parentheses). Their order is counted relative to their respective subgrid communicator groups.

0,0(0)	**0,1(1)**
1,0(2)	**1,1(3)**
2,0(4)	**2,1(5)**

Figure 8.4 (a). 2D Cartesian Grid

0,0(0) **0(0)**	**0,1(1)** **1(1)**
1,0(2) **0(0)**	**1,1(3)** **1(1)**
2,0(4) **0(0)**	**2,1(5)** **1(1)**

Figure 8.4 (b). 3 Row Subgrids

0,0(0) **0(0)**	**0,1(1)** **0(0)**
1,0(2) **1(1)**	**1,1(3)** **1(1)**
2,0(4) **2(2)**	**2,1(5)** **2(2)**

Figure 8.4 (c). 2 Column Subgrids

Here is a Fortran example demonstrating the column subgrid.

Note:

- MPI_CART_SUB is a collective routine. It must be called by all processes in old_comm.
- MPI_CART_SUB-generated subgrid communicators are derived from Cartesian grid created with MPI_CART_CREATE.
- Full length of each dimension of the original Cartesian grid is used in the subgrids.
- Each subgrid has a corresponding communicator. It inherits properties of the parent Cartesian grid; it remains a Cartesian grid.
- It returns the communicator to which the calling process belongs.
- There is a comparable MPI_COMM_SPLIT to perform similar function.

- MPI_CARTDIM_GET and MPI_CART_GET can be used to acquire structural information of a grid (such as dimension, size, periodicity).

8.1.5. MPI_CARTDIM_GET

MPI_CARTDIM_GET

The MPI_CARTDIM_GET routine determines the number of dimensions of a subgrid communicator.

On occasions, a subgrid communicator may be created in one routine and subsequently used in another routine. If the dimension of the subgrid is not available, it can be determined by MPI_CARTDIM_GET.

C:

```
int MPI_Cartdim_get( MPI_Comm comm, int* ndims )
```

Fortran:

```
MPI_CARTDIM_GET( COMM, NDIMS, IERR )
```

Variable Name	C Type	Fortran Type	In/Out	Description
comm	MPI_Comm	INTEGER	Input	Cartesian communicator handle
ndims	int *	INTEGER	Output	Number of dimensions
ierr	See (*)	INTEGER	Output	Error flag

* For C, the function returns an int error flag.

C Example:

```
/* create column subgrids */
 belongs[0] = 1;
 belongs[1] = 0;
 MPI_Cart_sub(grid_comm, belongs, &col_comm);
/* queries number of dimensions of cartesan grid */
 MPI_Cartdim_get(col_comm, &ndims);
```

Fortran Example:

```
!**create column subgrids
      belongs(0) = .true.
      belongs(1) = .false.
      call MPI_Cart_sub(grid_comm, belongs, col_comm, ierr)
!**queries number of dimensions of cartesian grid
      call MPI_Cartdim_get(col_comm, ndims, ierr)
```

On occasions, detailed information about a grid may not be available, as in the case where a communicator is created in one routine and is used in another. In such a situation, MPI_CARTDIM_GET may be used to find the dimension of the grid associated with the communicator. Armed with this value, additional information may be obtained by calling MPI_CART_GET, which is discussed in the next section.

8.1.6. MPI_CART_GET

MPI_CART_GET

The MPI_CART_GET routine retrieves properties such as periodicity and size of a subgrid.

On occasions, a subgrid communicator may be created in one routine and subsequently used in another routine. If only the communicator is available in the latter, this routine, along with MPI_CARTDIM_GET, may be used to determine the size and other pertinent information about the subgrid.

C:

```
int MPI_Cart_get( MPI_Comm subgrid_comm, int ndims, int *dims, int
*periods, int *coords )
```

Fortran:

```
MPI_CART_GET( SUBGRID_COMM, NDIMS, DIMS, PERIOD, COORDS, IERR )
```

Variable Name	C Type	Fortran Type	In/Out	Description
subgrid_comm	MPI_Comm	INTEGER	Input	Communicator handle
ndims	int	INTEGER	Input	Number of dimensions
dims	int *	INTEGER	Output	Array of size ndims providing length in each dimension
periods	int *	LOGICAL	Output	Array of size ndims specifying periodicity status of each dimension

coords	int *	INTEGER	Output	Array of size ndims providing Cartesian coordinates of calling process
ierr	See (*)	INTEGER	Output	Error flag

* For C, the function returns an int error flag.

C Example:

```
/* create Cartesian topology for processes */
  dims[0] = nrow;
  dims[1] = mcol;
  MPI_Cart_create(MPI_COMM_WORLD, ndim, dims,
              period, reorder, &grid_comm);
  MPI_Comm_rank(grid_comm, &me);
  MPI_Cart_coords(grid_comm, me, ndim, coords);
/* create row subgrids */
  belongs[0] = 1;
  belongs[1] = 0;
  MPI_Cart_sub(grid_comm, remain, &row_comm);
/* Retrieve subgrid dimensions and other info */
  MPI_Cartdim_get(row_comm, &mdims);
  MPI_Cart_get(row_comm, mdims, dims, period,
          row_coords);
```

Fortran Example:

```
create Cartesian topology for processes
  dims(1) = nrow
  dims(2) = mcol
  call MPI_Cart_create(MPI_COMM_WORLD, ndim, dims,
&       period, reorder, grid_comm, ierr)
  call MPI_Comm_rank(grid_comm, me, ierr)
  call MPI_Cart_coords(grid_comm, me, ndim, coords, ierr)
create row subgrids
  belongs(0) = 1
  belongs(1) = 0
  call MPI_Cart_sub(grid_comm, remain, row_comm, ierr)
c**Retrieve subgrid dimensions and other info
  call MPI_Cartdim_get(row_comm, mdims, ierr)
  call MPI_Cart_get(row_comm, mdims, dims, period,
 &     row_coords, ierr)
```

Shown in Figure 8.5 below is a 3-by-2 Cartesian topology (grid) where the index pair "i,j" represents row "i" and column "j". The number in parentheses represents the rank number associated with the Cartesian grid.

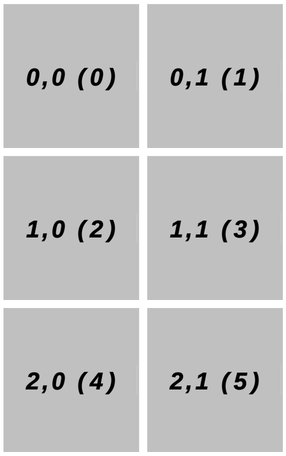


Figure 8.5. Cartesian Grid

This example demonstrated the use of MPI_CART_GET to retrieve information on a subgrid communicator. Often, MPI_CARTDIM_GET needs to be called first since ndims, the dimensions of the subgrid, is needed as input to MPI_CART_GET.

8.1.7. MPI_CART_SHIFT

MPI_CART_SHIFT

The MPI_CART_SHIFT routine finds the resulting source and destination ranks, given a shift direction and amount.

C:

```
   int MPI_Cart_shift( MPI_Comm comm, int direction, int displ, int
*source, int *dest )
```

Fortran:

```
   MPI_CART_SHIFT( COMM, DIRECTION, DISPL, SOURCE, DEST, IERR )
```

Loosely speaking, MPI_CART_SHIFT is used to find two "nearby" neighbors of the calling process along a specific direction of an N-dimensional Cartesian topology. This direction is specified by the input argument, direction, to MPI_CART_SHIFT. The two neighbors are called "source" and "destination" ranks, and the proximity of these two neighbors to the calling process is determined by the input parameter displ. If displ = 1, the neighbors are the two adjoining processes along the specified direction and the source is the process with the lower rank number, while the destination rank is the process with the higher rank. On the other hand, if displ = -1, the reverse is true. A simple code fragment and a complete sample code are shown below to demonstrate the usage. A more practical example of an application is given in Section 8.2.2 - Iterative Solvers.

Variable Name	C Type	Fortran Type	In/Out	Description
comm	MPI_Comm	INTEGER	Input	Communicator handle
direction	int	INTEGER	Input	The dimension along which shift is to be in effect
displ	int	INTEGER	Input	Amount and sense of shift (<0; >0; or 0)
source	int *	INTEGER	Output	The source of shift (a rank number)
dest	int *	INTEGER	Output	The destination of shift (a rank number)
ierr	See (*)	INTEGER	Output	Error flag

* For C, the function returns an int error flag.

C Example:

```
   /* create Cartesian topology for processes */
     dims[0] = nrow;     /* number of rows     */
     dims[1] = mcol;     /* number of columns  */
     period[0] = 1;      /* cyclic in this direction */
     period[1] = 0;      /* no cyclic in this direction */
     MPI_Cart_create(MPI_COMM_WORLD, ndim, dims, period, reorder,
&comm2D);
     MPI_Comm_rank(comm2D, &me);
     MPI_Cart_coords(comm2D, me, ndim, coords);

     index =  0;    /* shift along the 1st index (out of 2) */
     displ =  1;    /* shift by  1 */
     MPI_Cart_shift(comm2D, index, displ, &source, &dest1);
```

Fortran Example:

```
!**create Cartesian topology for processes
      dims(1) = nrow      ! number of rows
      dims(2) = mcol      ! number of columns
      period(0) = .true.  ! cyclic in this direction
      period(1) = .false. ! no cyclic in this direction
      call MPI_Cart_create(MPI_COMM_WORLD, ndim, dims,
     &       period, reorder, comm2D, ierr)
      call MPI_Comm_rank(comm2D, me, ierr)
      call MPI_Cart_coords(comm2D, me, ndim, coords, ierr)
```

```
direction =  0   ! shift along the 1st index (0 or 1)
displ =  1       ! shift by  1

call MPI_Cart_shift(comm2D, direction, displ, source, dest, ierr)
```

In the above example, we demonstrate the application of MPI_CART_SHIFT to obtain the source and destination rank numbers of the calling process, me, resulting from shifting it along the first direction of the 2D Cartesian grid by one.

Shown in Figure 8.6 below is a 3x2 Cartesian topology (grid) where the index pair "i,j" represent row "i" and column "j". The number in parentheses represents the rank number associated with the Cartesian coordinates.

0,0 (0)	***0,1 (1)***
1,0 (2)	***1,1 (3)***
2,0 (4)	***2,1 (5)***

Figure 8.6. Cartesian Grid

With the input as specified above and 2 as the calling process, the source and destination rank would be 0 and 4, respectively, as a result of the shift. Similarly, if the calling process were 1, the source rank would be 5 and destination rank would be 3. The source rank of 5 is the consequence of period(0) = .true. More examples are included in the sample code.

Note:

- Direction, the Cartesian grid dimension index, has range (0, 1, ..., ndim-1). For a 2D grid, the two choices for direction are 0 and 1.
- MPI_CART_SHIFT is a query function. No action results from its call.
- A negative returned value (MPI_UNDEFINED) of source or destination signifies the respective value is out of bound. It also implies that there is no periodicity along that direction.
- If periodic condition is enabled along the shift direction, an out of bound does not result. (See sample code).

8.2. Practical Applications

Practical Applications of Virtual Topologies

The practical applications of virtual topologies listed below are discussed in the following sections.

- Matrix Transposition
- Iterative Solvers

8.2.1. Matrix Transposition

Matrix Transposition

This section demonstrates the use of virtual topologies by way of a matrix transposition. The matrix algebra for a matrix transposition is demonstrated in the following example.

Consider a ***3 x 3*** matrix $\boldsymbol{A}$. This matrix is blocked into sub-matrices $\boldsymbol{A}_{11}$, $\boldsymbol{A}_{12}$, $\boldsymbol{A}_{21}$, and $\boldsymbol{A}_{22}$ as follows:

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} = \begin{bmatrix} \mathbf{A}_{11} & \mathbf{A}_{12} \\ \mathbf{A}_{21} & \mathbf{A}_{22} \end{bmatrix} \qquad (1)$$

where

$$\mathbf{A}_{11} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}; \ \mathbf{A}_{12} = \begin{bmatrix} a_{13} \\ a_{23} \end{bmatrix}$$

$$\mathbf{A}_{21} = \begin{bmatrix} a_{31} & a_{32} \end{bmatrix}; \ \mathbf{A}_{22} = \begin{bmatrix} a_{33} \end{bmatrix} \qquad (2)$$

Next, let $\boldsymbol{B}$ represent the transpose of $\boldsymbol{A}$.

$$\mathbf{B} = \begin{bmatrix} \mathbf{B}_{11} & \mathbf{B}_{12} \\ \mathbf{B}_{21} & \mathbf{B}_{22} \end{bmatrix} = \mathbf{A}^{\mathrm{T}}$$

$$= \begin{bmatrix} \mathbf{A}_{11} & \mathbf{A}_{12} \\ \mathbf{A}_{21} & \mathbf{A}_{22} \end{bmatrix}^{\mathrm{T}} \qquad (3)$$

$$= \begin{bmatrix} \mathbf{A}_{11}^{\mathbf{T}} & \mathbf{A}_{21}^{\mathbf{T}} \\ \mathbf{A}_{12}^{\mathbf{T}} & \mathbf{A}_{22}^{\mathbf{T}} \end{bmatrix}$$

According to Equation (3) above, the element B_{ij} is the blocked submatrix $\mathbf{A}_{ji}^{\mathbf{T}}$. For instance,

$$\mathrm{B}_{12} = \mathbf{A}_{21}^{\mathbf{T}}$$

$$= \begin{bmatrix} a_{31} \\ a_{32} \end{bmatrix} \qquad (4)$$

The parallel algorithm is

1. Select p and q such that the total number of processes, $nprocs = p \times q$.
2. Partition the $n \times m$ matrix into a (blocked) $p \times q$ matrix whose elements are themselves matrices of size $(n/p) \times (m/q)$.
3. Perform a transpose on each of these sub-matrices. These are performed serially because the entire sub-matrix resides locally on a process. No inter-process communication is required.
4. Formally, the $p \times q$ matrix needs to be transposed to obtain the final result. However, in reality this step is often not necessary. If you need to access the element (or sub-matrix) "p,q" of the transposed matrix, all you need to do is access the element "q,p", which has already been transposed locally. Depending on what comes next in the calculation, unnecessary message passing may be avoided.

As an example (see Figure 8.8), a 9×4 matrix with 6 processes is defined. Next, that matrix is mapped into a 3×2 virtual Cartesian grid, i.e., $p=3$, $q=2$. Coincidentally, each element of this Cartesian grid is, in turn, a 3×2 matrix.

i,j (p) a_{ij}

For the physical grid, each square box represents one entry of the matrix. The pair of indices, "i,j", on the first row gives the global Cartesian coordinates, while "(p)" is the process associated with the virtual grid allocated by calling MPI_CART_CREATE or MPI_COMM_SPLIT. On the second row, a_{ij}, is the value of the matrix element.

The 3×2 virtual grid is depicted on the right of Figure 8.8. Each box in this grid represents one process and contains one 3×2 submatrix. Finally, another communicator is created for the transposed virtual grid with dimensions of 2×3. For instance, the element at "$1,0$" of the transposed virtual grid stores the value sent by the element at "$0,1$" of the virtual grid.

Physical Grid

0,0 (0) 100	0,1 (0) 101	0,2 (1) 102	0,3 (1) 103
1,0 (0) 110	1,1 (0) 111	1,2 (1) 112	1,3 (1) 113
2,0 (0) 120	2,1 (0) 121	2,2 (1) 122	2,3 (1) 123
3,0 (2) 130	3,1 (2) 131	3,2 (3) 132	3,3 (3) 133
4,0 (2) 140	4,1 (2) 141	4,2 (3) 142	4,3 (3) 143
5,0 (2) 150	5,1 (2) 151	5,2 (3) 152	5,3 (3) 153
6,0 (4) 160	6,1 (4) 161	6,2 (5) 162	6,3 (5) 163
7,0 (4) 170	7,1 (4) 171	7,2 (5) 172	7,3 (5) 173
8,0 (4) 180	8,1 (4) 181	8,2 (5) 182	8,3 (5) 183

Virtual Grid

0,0 (0)	0,1 (1)
1,0 (2)	1,1 (3)
2,0 (4)	2,1 (5)

Transposed Virtual Grid

0,0 (0)	0,1 (1)	0,2 (2)
1,0 (3)	1,1 (4)	1,2 (5)

Figure 8.8. An example of the use of virtual topologies by way of a matrix transposition using a $9 \; x \; 4$ matrix with 6 processes.

Sample code is provided for this example.

8.2.2. Iterative Solvers (F90)

Example : Iterative Solvers

In this example, we demonstrate an application of the Cartesian topology by way of a simple elliptic (Laplace) equation.

Fundamentals: The Laplace equation, along with prescribed boundary conditions, are introduced. Finite Difference Method is then applied to discretize the PDE to form an algebraic system of equations.

Jacobi Scheme: A very simple iterative method, known as the Jacobi Scheme, is described. A single-process computer code is shown. This program is written in Fortran 90 for its concise but clear array representations. (Parallelism and other improvements will be added to this code as you progress through the example.)

Parallel Jacobi Scheme: A parallel algorithm for this problem is discussed. Simple MPI routines, without the invocations of Cartesian topology, are inserted into the basic, single-process code to form the parallel code.

SOR Scheme: The Jacobi scheme, while simple and hence desirable for demonstration purposes, is impractical for "real" applications because of its slow convergence. Enhancements to the basic technique are introduced leading to the Successive Over Relaxation (SOR) scheme.

Parallel SOR Scheme: With the introduction of a "red-black" algorithm, the parallel algorithm used for Jacobi is employed to parallelize the SOR scheme.

Scalability: The performance of the code for a number of processes is shown to demonstrate its scalability.

Download a zipped file containing the F90 Jacobi and SOR codes (in f77, f90 and C), along with make files and a Matlab m-file for plotting the solution.

8.2.2.1. Fundamentals

Fundamentals

First, some basics.

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0 \qquad (1)$$

where $u=u(x,y)$ is an unknown scalar potential subjected to the following boundary conditions:

$$u(x,0) = sin(\pi x) \qquad 0 \le x \le 1$$
$$u(x,1) = sin(\pi x)e^{-x} \qquad 0 \le x \le 1 \qquad (2)$$
$$u(0,y) = u(1,y) = 0 \qquad 0 \le y \le 1$$

Discretize the equation numerically with centered difference results in the algebraic equation

$$u_{i,j}^{n+1} \cong \frac{u_{i+1,j}^{n} + u_{i-1,j}^{n} + u_{i,j+1}^{n} + u_{i,j-1}^{n}}{4}; i = 1,\ldots,m; j = 1,\ldots,m \qquad (3)$$

where ***n*** and ***n+1*** denote the current and the next time step, respectively, while $u_{i-1,j}^{n}$ represents

$$\begin{aligned} u_{i-1,j}^{n} &= u^{n}(x_{i-1}, y_j); i = 1,\ldots,m; j = 1,\ldots,m \\ &= u^{n}((i-1)\Delta x, j\Delta y) \end{aligned} \qquad (4)$$

and for simplicity, we take $\Delta x = \Delta y = \dfrac{1}{m+1}$.

Note that the analytical solution for this boundary value problem can easily be verified to be

$$u(x,y) = sin(\pi x)e^{-xy}; 0 \le x \le 1; 0 \le y \le 1 \qquad (5)$$

and is shown below in a contour plot with x pointing from left to right and y going from bottom to top.

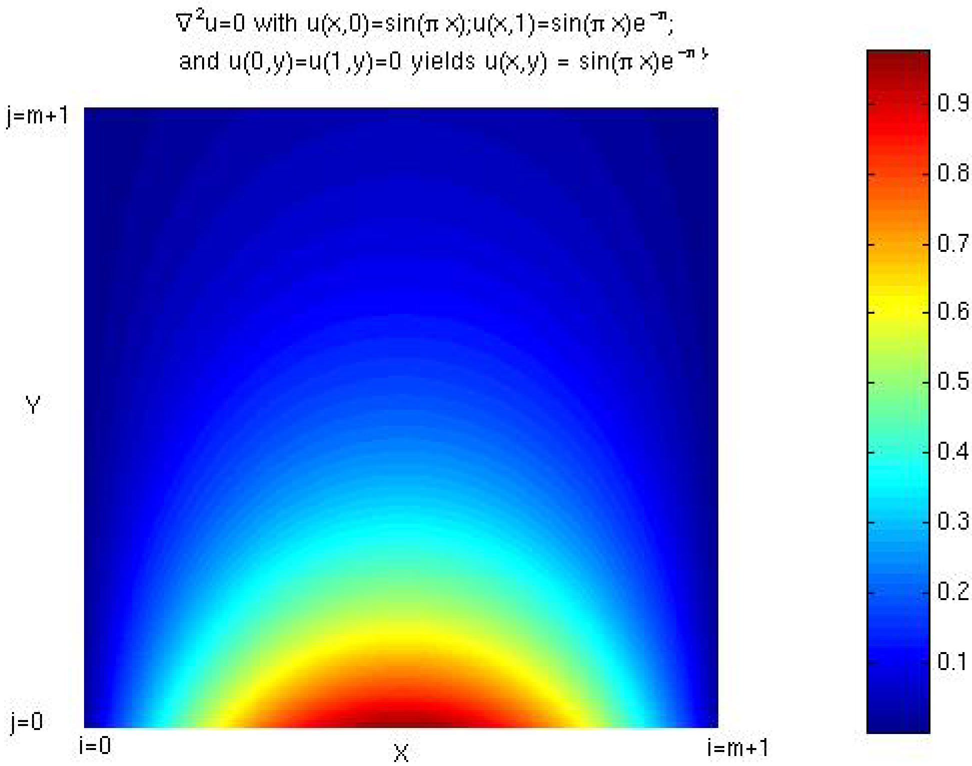


Figure 8.9. Contour plot showing the analytical solution for the boundary value problem.

8.2.2.2. Jacobi Scheme

Jacobi Scheme

While numerical techniques abound to solve PDEs such as the Laplace equation, we will focus on the use of two iterative methods. These methods will be shown to be readily parallelizable, as well as lending themselves to the opportunity to apply MPI Cartesian topology introduced above. The simplest of iterative techniques is the Jacobi scheme, which may be stated as follows:

1. Make initial guess for $u_{i,j}$ at all interior points (i,j) for all $i=1:m$ and $j=1:m$.
2. Use Equation 3 to compute $u^{n+1}_{i,j}$ at all interior points (i,j).
3. Stop if the prescribed convergence threshold is reached, otherwise continue on to the next step.
4. $u^{n}_{i,j} = u^{n+1}_{i,j}$.
5. Go to Step 2.

Single process Fortran 90 code for the Jacobi Scheme

Single process Fortran 77 code for the Jacobi Scheme

Single process C code for the Jacobi Scheme

8.2.2.3. Parallel Jacobi Scheme

Parallel Algorithm for the Jacobi Scheme

First, to enable parallelism, the work must be divided among the individual processes; this is known commonly as domain decomposition. Because the governing equation is two-dimensional, typically the choice is to use a 1D or 2D decomposition. This section will focus on a 1D decomposition, deferring the discussion of a 2D decomposition for later. Assuming that p processes will be used, the computational domain is split into p horizontal strips, each assigned to one process, along the north-south or y-direction. This choice is made primarily to facilitate simpler boundary condition (code) implementations.

For the obvious reason of better load-balancing, we will divide the amount of work, in this case proportional to the grid size, evenly among the processes ($m \times m / p$). For convenience, $m' = m/p$ is defined as the number of cells in the y-direction for each process. Next, Equation 3 is restated for a process k as follows:

$$v_{i,j}^{n+1,k} \simeq \frac{v_{i+1,j}^{n,k} + v_{i-1,j}^{n,k} + v_{i,j+1}^{n,k} + v_{i,j-1}^{n,k}}{4}; \qquad (6)$$
$$i = 1, \ldots, m;\ j = 1, \ldots, m';\ k = 0, \ldots, p-1$$

where v denotes the local solution corresponding to the process k with $m'=m/p$.

The figure below depicts the grid of a typical process k, as well as part of the adjoining grids of $k-1$, $k+1$.

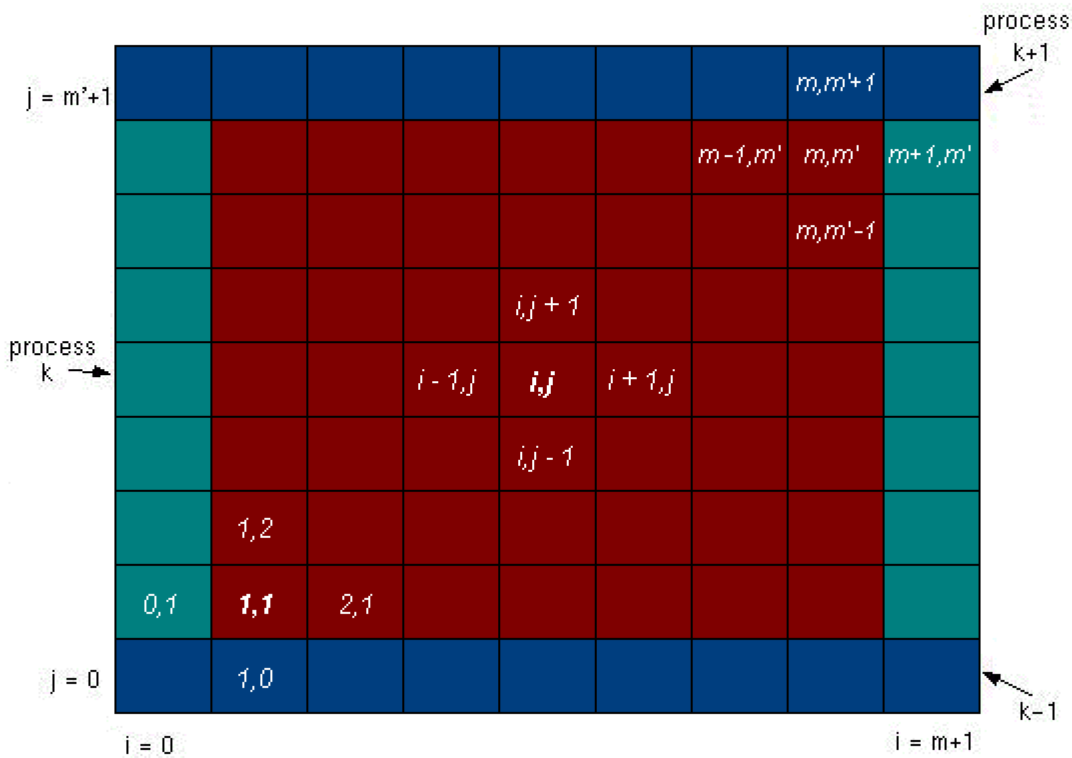


Figure 8.10. The grid of a typical process k as well as part of adjoining grids of k-1, k+1

- The **red** cells represent process k's grid cells for which the solution u is sought through Equation 6.
- The **blue** cells on the top row represent cells belonging to the first row ($j = 1$) of cells of process k+1. The **blue** cells on the bottom row represent the last row ($j = m'$) of cells of process k-1. It is important to note that the u at the blue cells of k belong to adjoining processes (k-1 and k+1) and hence must be "imported" via MPI message passing routines. Similarly, process k's first and last rows of cells must be "exported" to adjoining processes for the same reason.
- For $i = 1$ and $i = m$, Equation 6 again requires an extra cell beyond these two locations. These cells contain the prescribed boundary conditions ($u(0,y) = u(1,y) = 0$) and are colored **green** to distinguish them from the **red** and **blue** cells. Note that no message passing operations are needed for these green cells as they are fixed boundary conditions and are known a priori.
- From the standpoint of process k, the blue and green cells may be considered as additional "boundary" cells around it. As a result, the range of the strip becomes (0:m+1,0:m'+1). Physical boundary conditions are imposed on its green cells, while u is imported to its blue "boundary" cells from two adjoining processes. With the boundary conditions in place, Equation 6 can be applied to all of its interior points. Concurrently, all other processes proceed following the same procedure. It is interesting to note that the grid layout for a typical

process k is completely analogous to that of the original undivided grid. Whereas the orginal problem has fixed boundary conditions, the problem for a process k is subjected to variable boundary conditions.

- These boundary conditions can be stated mathematically as

$$
\begin{aligned}
v^{n,k}_{i,0} &= u(x_i,0) = sin(\pi x_i)\ ; \quad i=0,\ \ldots, m+1; \quad k=0 \\
v^{n,k}_{i,m'+1} &= v^{n,k+1}_{i,1}\ ; \quad i=0,\ \ldots, m+1; \quad k=0 \\
v^{n,k}_{i,0} &= v^{n,k-1}_{i,m'}\ ; \quad i=0,\ \ldots, m+1; \quad 0<k<p-1 \\
v^{n,k}_{i,m'+1} &= v^{n,k+1}_{i,1}\ ; \quad i=0,\ \ldots, m+1; \quad 0<k<p-1 \\
v^{n,k}_{i,0} &= v^{n,k-1}_{i,m'}\ ; \quad i=0,\ \ldots, m+1; \quad k=p-1 \\
v^{n,k}_{i,m'+1} &= u(x_i,1) = sin(\pi x_i)e^{-x}\ ; \quad i=0,\ \ldots, m+1; \quad k=p-1 \\
v^{n,k}_{0,j} &= u(0,y_j) = 0\ ; \quad j=1,\ \ldots, m'; \quad 0\le k\le p-1 \\
v^{n,k}_{m+1,j} &= u(1,y_j) = 0\ ; \quad j=1,\ \ldots, m'; \quad 0\le k\le p-1
\end{aligned}
\qquad (7)
$$

- Note that the interior points of u and v are related by the relationship

$$u^n_{i,j+k*m/p} = v^{n,k}_{i,j}\ ; \quad i=1,\ \ldots, m\ ; \quad j=1,\ \ldots, m'\ ; \quad 0\le k\le p-1 \qquad (8)$$

Jacobi Parallel Implementation. Note that Cartesian topology is not employed in this implementation but will be used later in the parallel SOR example with the purpose of showing alternative ways to solve this type of problems.

8.2.2.4. SOR Scheme

Successive Over Relaxation (SOR)

While the Jacobi iteration scheme is very simple and easily parallelizable, its slow convergent rate renders it impractical for any "real world" applications. One way to speed up the convergent rate would be to "over predict" the new solution by linear extrapolation. This leads to the Successive Over Relaxation (SOR) scheme shown below:

1. Make initial guess for $u_{i,j}$ at all interior points (i,j).
2. Define a scalar w_n $(0 < w_n < 2)$.
3. Apply Equation 3 to all interior points (i,j) and call it $u'_{i,j}$.
4. $u^{n+1}_{i,j} = w_n\, u'_{i,j} + (1 - w_n)\, u^n_{i,j}$.
5. Stop if the prescribed convergence threshold is reached, otherwise continue to the next step.

6. $u^{n}_{i,j} = u^{n+1}_{i,j}$.
7. Go to Step 2.

Note that in the above setting $w_n = 1$ recovers the Jacobi scheme while $w_n < 1$ underrelaxes the solution. Ideally, the choice of w_n should provide the optimal rate of convergence and is not restricted to a fixed constant. As a matter of fact, an effective choice of w_n, known as the Chebyshev acceleration, is defined as

$$\omega_n = \begin{cases} 0 & \text{for } n = 0 \\ \frac{1}{1-\rho^2/2} & \text{for } n = 1 \\ \frac{1}{1-\rho^2\omega_1/4} & \text{for } n = 2 \\ \frac{1}{1-\rho^2\omega_{q-1}/4} & \text{for } n = q > 2 \end{cases}$$

where $\rho = 1 - \left(\frac{\pi}{2(m+1)}\right)^2$ is the spectral radius.

For further detail, see Chapter 19.5 - Relaxation Methods for Boundary Value Problems in the online book *Numerical Recipes*.

We can further speed up the rate of convergence by using u at time level $n+1$ for any or all terms on the right hand side of Equation 6 as soon as they become available. This is the essence of the Gauss-Seidel scheme. A conceptually similar red-black scheme will be used here. This scheme is best understood visually by painting the interior cells alternately in red and black to yield a checkerboard-like pattern as shown in Figure 8.11.

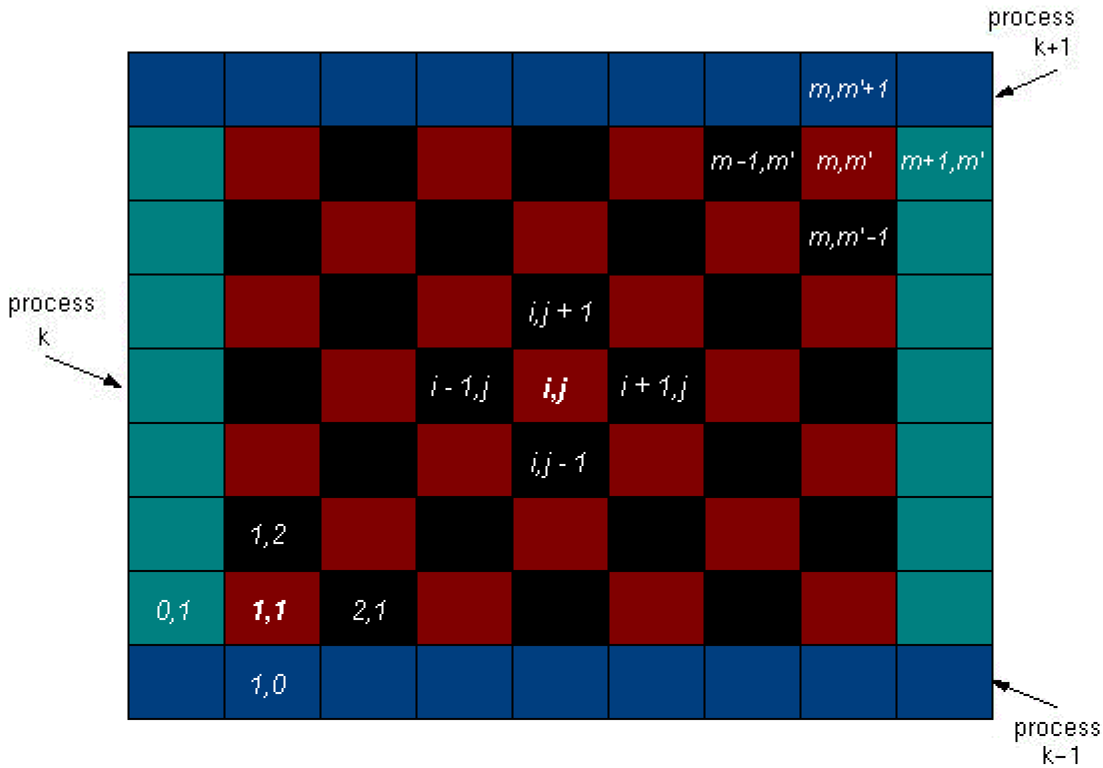


Figure 8.11. Checkerboard-like pattern depicting a parallel SOR red-black scheme.

By using this **red**- **black** group identification strategy and applying the five-point finite-difference stencil to a point ***(i,j)*** located at a red cell, it is immediately apparent that the solution at the red cell depends only on its four immediate black neighbors to the north, east, west, and south (by virtue of Equation 6). On the contrary, a point ***(i,j)*** located at a black cell depends only on its north, east, west, and south red neighbors. In other words, the finite-difference stencil in Equation 6 effects an uncoupling of the solution at interior cells such that the solution at the red cells depends only on the solution at the black cells and vice versa. In a typical iteration, if we first perform an update on all red ***(i,j)*** cells, then when we perform the remaining update on black ***(i,j)*** cells, the red cells that have just been updated could be used. Otherwise, everything that we described about the Jacobi scheme applies equally well here; i.e., the **green** cells represent the physical boundary conditions while the solutions from the first and last rows of the grid of each process are deposited into the **blue** cells of respective process grids to be used as the remaining boundary conditions.

Single process Fortran 90 code for the Successive Over Relaxation Scheme

Single process Fortran 77 code for the Successive Over Relaxation Scheme

Single process C code for the Successive Over Relaxation Scheme

8.2.2.5. Parallel SOR Scheme

A Parallel SOR Red-black Scheme

The parallel aspect of the Jacobi scheme can be used verbatim for the SOR scheme. Figure 8.11, as introduced in the previous section on the single-thread SOR scheme, may be used to represent the layout for a typical thread "k" of the SOR scheme, *i.e.*,

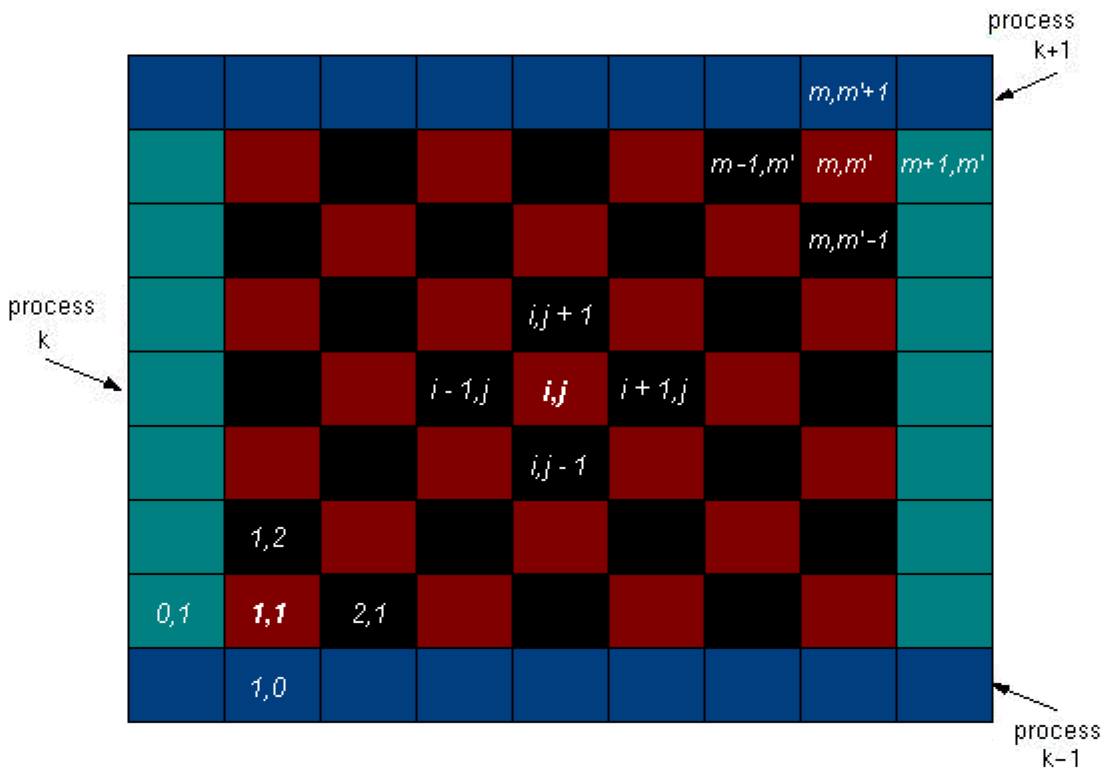


Figure 8.11. Checkerboard-like pattern depicting a parallel SOR red-black scheme.

As before, the green boxes denote boundary cells that are prescribed while the blue boxes represent boundary cells whose values are updated at each iteration by way of message passing.

The parallel SOR red-black scheme has been implemented in F90, F77 and C, respectively:

Parallel F90 SOR code.

Parallel Fortran 77 SOR code.

Parallel C SOR code.

8.2.2.6. Scalability Plot of SOR

Scalability Plot of SOR

The plot in Figure 8.12 below shows the scalability of the MPI implementation of the Laplace equation using SOR on an SGI Origin 2000 shared-memory multiprocessor.

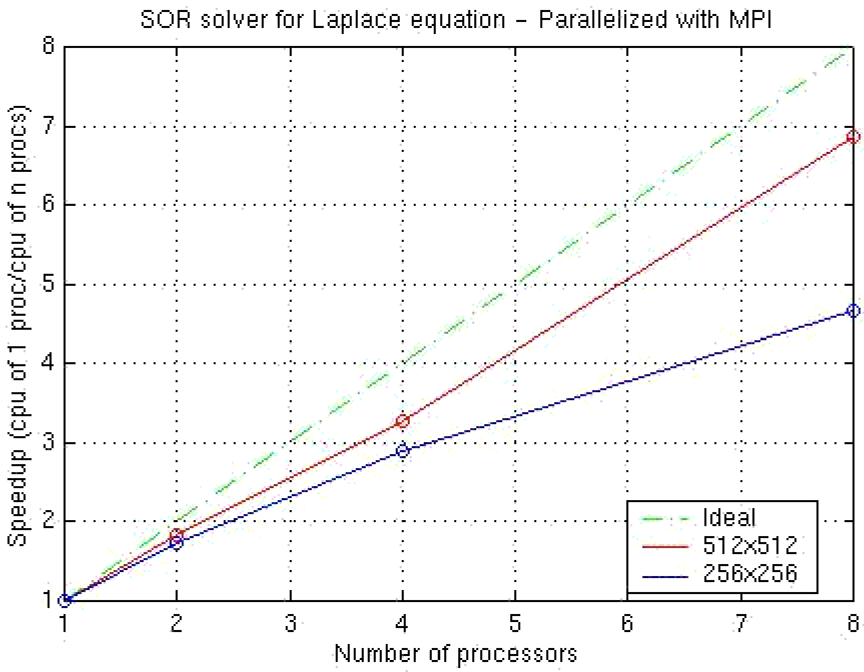


Figure 8.12. Scalability plot using SOR on an SGI Origin 2000.

8.3. Self Test

Virtual Topologies Self Test

Now that you've finished this chapter, test yourself on what you've learned by taking the Self Test provided. Simply click on the Self Test link in the ACTION MENU above to get started.

8.4. Course Problem

Chapter 8 Course Problem

This chapter discussed the creation and use of virtual topologies. Virtual topologies are a powerful feature of MPI because they allow the programmer to create a new communicator that fits the manner in which they have "visualized" how the processors have been connected all along. Once the new communicator is made, utility routines simplify the message passing required among the processors.

For this chapter, the exercise will solve the same problem as described in Chapter 7, but in a different manner. You will create and use a "ring" topology to perform the parallel search and the average the data obtained from neighbor processors.

Description

The new problem still implements a parallel search of an integer array. The program should find all occurrences of a certain integer which will be called the target. When a processor of a certain rank finds a target location, it should then calculate the average of

- The target value
- An element from the processor with rank one higher (the **"right"** processor). The right processor should send the first element from its local array.
- An element from the processor with rank one less (the **"left"** processor). The left processor should send the first element from its local array.

For example, if processor 1 finds the target at index 33 in its local array, it should get from processors 0 (left) and 2 (right) the first element of *their* local arrays. These three numbers should then be averaged.

In terms of right and left neighbors, you should visualize the four processors connected in a ring. That is, the left neighbor for P0 should be P3, and the right neighbor for P3 should be P0.

Both the target location and the average should be written to an output file. As usual, the program should read both the target value and all the array elements from an input file.

Exercise

Modify your code from Chapter 7 to solve this latest version of the Course Problem using a virtual topology. First, create the topology (which should be called MPI_RING) in which the four processors are connected in a ring. Then, use the utility routines to determine which neighbors a given processor has.

Solution

When you have finished writing the code for this exercise, view our version of the Topology Code.

---

9. Parallel I/O

Parallel I/O - Objectives

The material covered to this point discussed how multiple processes can share data stored in separate memory spaces (See Section 1.1 - Parallel Architectures). This is achieved by sending messages between processes.

Parallel I/O covers the issue of how data are distributed among I/O devices. While the memory subsystem can be different from machine to machine, the logical methods of accessing memory are generally common, i.e., the same model should apply from machine to machine. Parallel I/O is complicated in that both the physical and logical configurations often differ from machine to machine.

MPI-2 is the first version of MPI to include routines for handling parallel I/O. As such, much of the material in this chapter is applicable only if the system you are working on has an implementation of MPI that includes parallel I/O. Many MPI implementations are starting to include bits and pieces of MPI-2, but at the time these notes were written there is no known "full" implementation of MPI-2.

This section proposes to introduce you to the general concepts of parallel I/O, with a focus on MPI-2 file I/O.

The material is organized to meet the following three goals:

1. Learn fundamental concepts that define parallel I/O
2. Understand how parallel I/O is different from traditional, serial I/O
3. Gain experience with the basic MPI-2 I/O function calls

The topics to be covered are

- Introduction
- Applications
- Characteristics of Serial I/O
- Characteristics of Parallel I/O
- Introduction to MPI-2 Parallel I/O
- MPI-2 File Structure
- Initializing MPI-2 File I/O
- Defining A View
- Data Access - Reading Data
- Data Access - Writing Data
- Closing MPI-2 File I/O

9.1. Introduction

Introduction

A traditional programming style teaches that computer programs can be broken down by function into three main sections:

1. input
2. computation
3. output

For science applications, much of what has been learned about MPI in this course addresses the computation phase. With parallel systems allowing larger computational models, these applications often produce large amounts of output.

Serial I/O on a parallel machine can have large time penalties for many reasons.

- Larger datasets generated from parallel applications have a serial bottleneck if I/O is only done on one node
- Many MPP machines are built from large numbers of slower processors, which increase the time penalty as the serial I/O gets funneled through a single, slower processor
- Some parallel datasets are too large to be sent back to one node for file I/O

Decomposing the computation phase while leaving the I/O channeled through one processor to one file can cause the time required for I/O to be of the same order or exceed the time required for parallel computation.

There are also non-science applications in which input and output are the dominant processes and significant performance improvement can be obtained with parallel I/O.

9.2. Applications

Applications

The ability to parallelize I/O can offer significant performance improvements. Several applications are given here to show examples.

Large Computational Grids/Meshes

Many new applications are utilizing finer resolution meshes and grids. Computed properties at each node and/or element often need to be written to storage for data analysis at a later time. These large computational grid/meshes

- Increase I/O requirements because of the larger number of data points to be saved.
- Increase I/O time because data is being funneled through slower commodity processors in MPP.

User-Level Checkpointing

User-level checkpointing is contained within the program itself. Like OS checkpointing, user-level checkpointing saves a program's state for a later restart. Below are some reasons you should incorporate checkpointing at the user level in your code.

- Even with massively parallel systems, runtime for large models is often measured in days.
- As the number of processors increases, there is a higher probability that one of the nodes your job is running on will suffer a hardware failure.
- Not all operating systems support OS level checkpointing.
- Larger parallel models require more I/O to save the state of a program.

Out-Of-Core Solvers

In an out-of-core problem the entire amount of data does not fit in the local main memory of a processor and must be stored on disk.

- Needed only if the OS does not support virtual memory.

Data Mining

Applications that have minimal computations, but many references to online databases are excellent candidates for parallel I/O.

- A phone book directory could be broken down to have chunks of the database distributed on each node.

9.3. Characteristics of Serial I/O

Characteristics of Serial I/O

To help understand what parallel I/O entails, it is beneficial to review some of the defining features of serial I/O and then compare the differences.

Physical Structure

- Generally there is one processor connected to one physical disk

Logical Structure

- Traditional view of I/O from high level languages
- Single file pointer
- Access to files can be sequential or random
- File parameters (read/write only, etc.)
- Data can be written raw or formatted, binary or ascii
- Built-in function calls
  - **Fortran** (`READ, WRITE`)
  - **C** (`fprintf, fscanf`)

9.4. Characteristics of Parallel I/O

Characteristics of Parallel I/O

Parallel I/O falls into one of two main categories:

1. Physical decomposition
2. Logical decomposition

Physical Decomposition

- Multiple processors writing data to multiple disks
- Local disk at each node
  - All I/O is local - best performance
  - Each process opens a unique file on the local system
  - Implemented with standard C or Fortran I/O calls
  - Excellent method for storing temporary data that do not need to be kept after the program executes
  - Data from multiple processes can be combined at the end of the run, if desired
  - Beowulf clusters support this type of parallel I/O
  - IBM SPs (in some configurations) support this type of I/O

- I/O nodes
  - Certain number of nodes are reserved for performing I/O
  - I/O performance will increase with the numbers of nodes allocated to I/O
  - Can be implemented with standard C/Fortran I/O calls or parallel I/O library calls
  - IBM SP (in some configurations) support this type of I/O

Logical Decomposition

- Multiple processors write data to a single disk
- Data written to a single file

- o Direct-access I/O in high level language terminology
  - o Data from various processes will be interleaved in the file
  - o MPI-2 implements its parallel I/O in this method

- Data written to multiple files
  - o Data from various processes are written to a file appended with the process rank
  - o At the end of the run the files are combined into one data file

Parallel machines can have I/O systems that fall in either category. It is possible to have a hybrid system that has characteristics from both categories, but these will not be discussed in this chapter.

9.5. Introduction to MPI-2 Parallel I/O

Introduction to MPI-2 Parallel I/O

One of the significant changes from MPI-1 to MPI-2 was the addition of parallel I/O. This is covered in chapter 9 of the MPI-2 standards document. This document, along with other useful information, can be found at the MPI Forum web page.

Before MPI-2, some parallel I/O could be achieved without MPI calls:

Parallel File System

- UNIX-like read/write and logical partitioning of files
- Example: PIOFS (IBM)

If each node had a local disk and data was written using standard I/O routines

- All files independent
- Examples: IBM SP thick nodes, workstation clusters

MPI-2 parallel I/O provides a high-level interface supporting partitioning of file data among processes and a collective interface supporting complete transfers of global data structures between process memories and files. Reading and writing data is now handled much like sending messages to the disk and receiving messages from storage. The standard I/O routines in Fortran, C, and C++ are not used.

Several key features of MPI-2 parallel I/O are

- MPI-2 allows you to perform parallel I/O similar to the way messages are sent from one process to another
- Not all implementations at present implement the full MPI-2 I/O
- Physical decomposition with a certain number of I/O nodes, as discussed on the previous slide, can be configured

- MPI-2 supports both blocking and nonblocking I/O (See Section 2.9 - Blocking and Nonblocking Communication)
- MPI-2 supports both collective and non-collective I/O (See Section 2.10 - Collective Communications)

This section will consider the case in which one node performs the I/O to one file. MPI-2 also offers several features for improving parallel I/O efficiency:

- Support for asynchronous I/O
- Strided accesses
- Control over physical file layout on storage devices (disks)

9.6. MPI-2 File Structure

MPI-2 File Structure

Parallel I/O in MPI-2 defines how multiple processes access and modify data in a shared file. This requires you to think about how data will be partitioned within this file.

MPI-2 partitions data within files similar to the way that derived datatypes define data partitions within memory (see Chapter 5 - Derived Datatypes). While simple datatypes can be used with MPI parallel I/O routines, the concepts of blocksize, memory striding, and offsets are fundamental to understanding the file structure. Also, performance benefits have been seen when parallel I/O is performed with derived datatypes as compared to parallel I/O performed with basic datatypes.

MPI-2 I/O has some of the following characteristics:

- MPI datatypes are written and read
- Concepts to those used to define derived datatypes are used to define how data is partitioned in the file
- Sequential as well as random access to files is supported
- Each process has its own "view" of the file

A **view** defines the current set of data visible and accessible by a process from an open file. Each process has its own view of the file, defined by three quantities:

1. a displacement - describing where in the file to start
2. an etype - the type of data that is to be written or read
3. a filetype - pattern of how the data is partitioned in the file

The pattern described by a filetype is repeated, beginning at the displacement, to define the view. The pattern of repetition is defined to be the same pattern that MPI_TYPE_CONTIGUOUS would produce if it were passed the filetype and an arbitrarily large count. Views can be changed by the user during program execution.

The default view is a linear byte stream (displacement is zero, etype and filetype equal to MPI_BYTE).

9.6.1. Displacement

Displacement

To better understand the file view, each component will be explained in more detail. The first of the parameters that define a view is **displacement**.

A file displacement is an absolute byte position relative to the beginning of a file. The displacement defines the location where a view begins.

Any data before the position indicated by the displacement will not be accessible from that process.

9.6.2. Elementary Datatype

Elementary Datatype - etype

The second parameter that defines a view is **etype**.

An etype (elementary datatype) is the unit of data access and positioning. It can be any MPI predefined or derived datatype. Derived etypes can be constructed using any of the MPI datatype constructor routines, provided all resulting typemap displacements are nonnegative and monotonically nondecreasing. Data access is performed in etype units, reading or writing whole data items of type etype. Offsets are expressed as a count of etypes; file pointers point to the beginning of etypes.

An etype

- Is the unit of data access and positioning
- Can be any MPI predefined datatype or derived datatype
- Is similar to the datatype argument in the MPI_SEND call

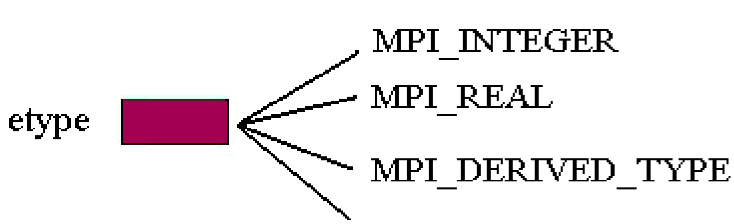


Figure 9.1. Elementary Datatype

9.6.3. Filetype

Filetype

With displacement and etype defined, the last item needed to define a view is a **filetype**.

A filetype is the basis for partitioning a file and defines a template for accessing the file. A filetype is either a single etype or a derived MPI datatype constructed from multiple instances of the same etype. In addition, the extent of any hole in the filetype must be a multiple of the etype's extent. The displacements in the typemap of the filetype are not required to be distinct, but they must be nonnegative and monotonically nondecreasing.

A filetype

- Is the basis for partitioning a file among processes
- Defines a template for accessing the file
- Is a defined sequence of etypes, which can have data or be considered blank
- Is similar to a vector derived datatype (See Section 5.7 - Other Ways of Defining MPI Derived Types)
- A filetype is repeated to fill the view of the file

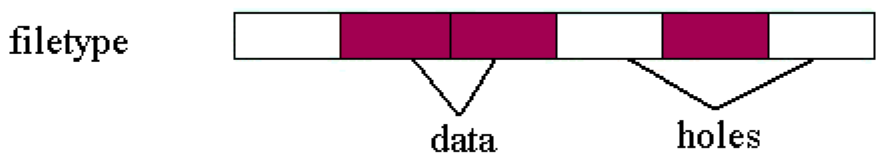


Figure 9.2. Filetype

When a parallel write or read is executed, some number of "filetypes" are passed.

9.6.4. View

View

Each process has its own **view** of the shared file that defines what data it can access.

- A view defines the current set of data, visible and accessible, from an open file
- Each process has its own view of the file

- A view can be changed by the user during program execution
- Starting at the position defined by the displacement, the filetype is repeated to form the view
- A default view is defined with the following values:
    - o displacement = 0
    - o etype = MPI_BYTE
    - o filetype = MPI_BYTE

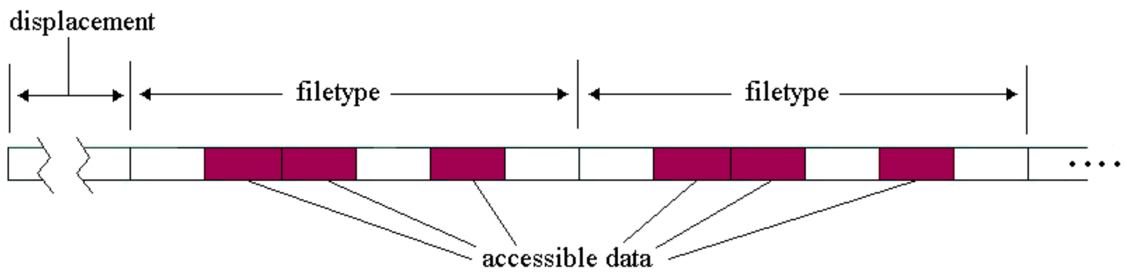


Figure 9.3. Displacement

9.6.5. Combining Views of Multiple Processes

Combining Views of Multiple Processes

A group of processes can use complementary views to achieve a global data distribution.

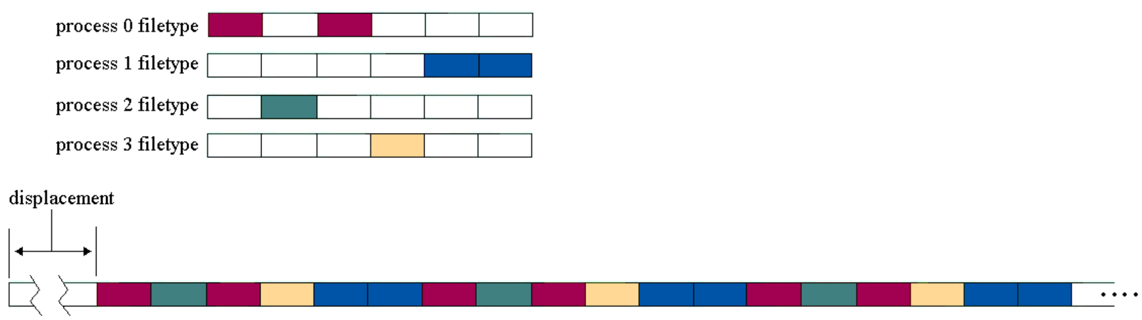


Figure 9.4. Combining Views of Multiple Processes

9.7. Initializing MPI-2 File I/O

Initializing MPI-2 File I/O

File I/O is initialized in MPI-2 with the MPI_FILE_OPEN call. This is a collective routine, and all processes within the specified communicator must provide filenames that reference the same file.

C:

```
int MPI_File_open(MPI_Comm comm, char *filename, int amode,
                  MPI_Info info, MPI_File *fh)
```

Fortran:

```
MPI_FILE_OPEN(COMM, FILENAME, AMODE, INFO, FH, IERROR)
CHARACTER*(*) FILENAME
INTEGER COMM, AMODE, INFO, FH, IERROR
```

The arguments for this routine are

```
COMM          process communicator
FILENAME      name of the file to open
AMODE         file access mode
INFO          information handle that varies with implementation
FH            new file handle by which the file can be accessed
```

The file access mode is set from the list of following options:

- **MPI_MODE_RONLY** - read only
- **MPI_MODE_RDWR** - reading and writing
- **MPI_MODE_WRONLY** - write only
- **MPI_MODE_CREATE** - create the file if it does not exist
- **MPI_MODE_EXCL** - error if creating file that already exists
- **MPI_MODE_DELETE_ON_CLOSE** - delete file on close
- **MPI_MODE_UNIQUE_OPEN** - file will not be concurrently opened elsewhere
- **MPI_MODE_SEQUENTIAL** - file will only be accessed sequentially
- **MPI_MODE_APPEND** - set initial position of all file pointers to end of file

These are all MPI-2 integer constants. AMODE is a bitwise "OR" of all of the above features desired.

- C/C++ - you can use bitwise OR (|) to combine the constants desired
- Fortran 90 - you can use the bitwise IOR intrinsic
- Fortran 77 - you can use the bitwise IOR on system that support it, but it is not portable

---

9.8. Defining a View

Defining a View

We learned earlier that each process has its own view of a shared file. The view is defined by three parameters: displacement, etype, and filetype.

The MPI_FILE_SET_VIEW routine is used to define a view.

C:

```
   int MPI_File_set_view(MPI_File fh, MPI_Offset disp, MPI_Datatype
etype, MPI_Datatype filetype, char *datarep, MPI_Info info)
```

Fortran:

```
   MPI_FILE_SET_VIEW(FH, DISP, ETYPE, FILETYPE, DATAREP, INFO, IERROR)
   INTEGER  FH, ETYPE, FILETYPE, INFO, IERROR
   CHARACTER*(*) DATAREP
   INTEGER(KIND=MPI_OFFSET_KIND) DISP
```

The arguments for this routine are

```
   FH             file handle specified in MPI_FILE_OPEN
   DISP           absolute displacement from the beginning of the file
   ETYPE          unit of data access and positioning
   FILETYPE       sequence of etypes, which is repeated to define the
view of a file
   DATAREP        data representation (see explanation below)
   INFO           information handle that varies with implementation
```

MPI_FILE and MPI_OFFSET are MPI types defined in the header file.

The DATAREP argument refers to the representation of the data within the file. This is important when considering the interoperability of a file, which is the ability to read the information previously written to a file. MPI guarantees full interoperability within a single MPI environment, and supports increased interoperability outside that environment through the external data representation.

While custom data representations can be defined, by default DATAREP can be one of the following values:

Native

- Data in this representation are stored in a file exactly as it is in memory
- Advantages: data precision and I/O performance are not lost in type conversions with a purely homogeneous environment
- Disadvantages: the loss of transparent interoperability within a heterogeneous MPI environment
- MPI implementation dependent

Internal

- Data in this representation can be used for I/O operations in a homogeneous or heterogeneous environment
- The implementation will perform type conversions if necessary
- The implementation is free to store data in any format of its choice, with the restriction that it will maintain constant extents for all predefined datatypes in any one file
- MPI implementation dependent

External32

- This data representation states that read and write operations convert all data from and to the "external32" representation
- All processes reading the file in a heterogeneous MPI environment will automatically have the data converted to their respective native representations
- The file can be exported from one MPI environment and imported into any other MPI environment with the guarantee that the second environment will be able to read all the data in the file
- Disadvantage: data precision and I/O performance may be lost in data type conversions
- external32 is common to all MPI implementations

MPI_FILE_SET_VIEW is a collective call and must be called by all processes within the communicator.

9.9. Data Access - Reading Data

Data Access - Reading Data

There are three aspects to data access - positioning, synchronism, and coordination.

Positioning

- explicit offset
- individual file pointers
- shared file pointers

Synchronism

- blocking communication
- nonblocking communication
- split collective

Coordination

- collective communication
- non-collective I/O

As an introduction to parallel data access, this section will look only at the blocking, non-collective calls. The subset of read routines that will be examined are

Positioning	Blocking, Non-collective Routine
Explicit offsets	MPI_FILE_READ_AT
Individual file pointer	MPI_FILE_READ
Shared file pointer	MPI_FILE_READ_SHARED

9.9.1. MPI_FILE_READ_AT

MPI_FILE_READ_AT

MPI_FILE_READ_AT reads a file beginning at the position specified by an explicit offset.

Explicit offset operations perform data access at the file position given directly as an argument. No file pointer is used or updated.

C:

```
   int MPI_File_read_at(MPI_File fh, MPI_Offset offset, void *buf, int
count, MPI_Datatype datatype, MPI_Status *status)
```

Fortran:

```
   MPI_FILE_READ_AT(FH, OFFSET, BUF, COUNT, DATATYPE, STATUS, IERROR)

   <type> BUF(*)
   INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
   INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
```

The arguments for this routine are:

```
   FH          file handle
   OFFSET      file offset
   BUF         initial address of buffer
   COUNT       number of elements in buffer
   DATATYPE    datatype of each buffer element (handle)
   STATUS      status object (Status)
```

9.9.2. MPI_FILE_READ

MPI_FILE_READ

MPI_FILE_READ reads a file using individual file pointers.

C:

```
int MPI_File_read(MPI_File fh, void *buf, int count,
                  MPI_Datatype datatype, MPI_Status *status)
```

Fortran:

```
MPI_FILE_READ(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
<type> BUF(*)
INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
```

The arguments for this routine are:

```
FH          file handle
BUF         initial address of buffer
COUNT       number of elements in buffer
DATATYPE    datatype of each buffer element (handle)
STATUS      status object (Status)
```

9.9.3. MPI_FILE_READ_SHARED

MPI_FILE_READ_SHARED

MPI_FILE_READ_SHARED reads a file using a shared file pointer.

C:

```
int MPI_File_read_shared(MPI_File fh, void *buf, int count,
                         MPI_Datatype datatype, MPI_Status *status)
```

Fortran:

```
MPI_FILE_READ_SHARED(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)

<type> BUF(*)
INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
```

The arguments for this routine are:

```
FH          file handle
BUF         initial address of buffer
COUNT       number of elements in buffer
DATATYPE    datatype of each buffer element (handle)
```

```
STATUS      status object (status)
```

9.10. Data Access - Writing Data

Data Access - Writing Data

As an introduction to parallel data access, this section will look only at the blocking, non-collective I/O calls. The subset of write routines that will be examined are

Positioning	Blocking, Non-collective Routine
Explicit offsets	MPI_FILE_WRITE_AT
Individual file pointer	MPI_FILE_WRITE
Shared file pointer	MPI_FILE_WRITE_SHARED

9.10.1. MPI_FILE_WRITE_AT

MPI_FILE_WRITE_AT

MPI_FILE_WRITE_AT writes a file beginning at the position specified by an offset.

Explicit offset operations perform data access at the file position given directly as an argument. No file pointer is used or updated.

C:

```
int MPI_File_write_at(MPI_File fh, MPI_Offset offset, void *buf, int
count, MPI_Datatype datatype, MPI_Status *status)
```

Fortran:

```
MPI_FILE_WRITE_AT(FH, OFFSET, BUF, COUNT, DATATYPE, STATUS, IERROR)

<type> BUF(*)
INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
```

The arguments for this routine are:

```
FH          file handle
OFFSET      file offset
BUF         initial address of buffer
COUNT       number of elements in buffer
```

```
DATATYPE   datatype of each buffer element (handle)
STATUS     status object (status)
```

9.10.2. MPI_FILE_WRITE

MPI_FILE_WRITE

MPI_FILE_WRITE writes a file using individual file pointers.

C:

```
int MPI_File_write(MPI_File fh, void *buf, int count,
                   MPI_Datatype datatype, MPI_Status *status)
```

Fortran:

```
MPI_FILE_WRITE(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)

<type> BUF(*)
INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
```

The arguments for this routine are:

```
FH         file handle
OFFSET     file offset
BUF        initial address of buffer
COUNT      number of elements in buffer
DATATYPE   datatype of each buffer element (handle)
STATUS     status object (status)
```

9.10.3. MPI_FILE_WRITE_SHARED

MPI_FILE_WRITE_SHARED

MPI_FILE_WRITE_SHARED writes a file using a shared file pointer.

C:

```
int MPI_File_write_shared(MPI_File fh, void *buf, int count,
                          MPI_Datatype datatype, MPI_Status *status)
```

Fortran:

```
MPI_FILE_WRITE_SHARED(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)

<type> BUF(*)
INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
```

The arguments for this routine are:

```
FH          file handle
BUF         initial address of buffer
COUNT       number of elements in buffer
DATATYPE    datatype of each buffer element (handle)
STATUS      status object (status)
```

9.11. Closing MPI-2 File I/O

Closing MPI-2 File I/O

MPI_FILE_CLOSE is a collective routine. It first synchronizes the state of the file associated with the file handle and then closes the file.

C:

```
int MPI_File_close(MPI_File *fh)
```

Fortran:

```
MPI_FILE_CLOSE(FH, IERROR)

INTEGER FH, IERROR
```

The user is responsible for ensuring that all outstanding nonblocking requests by a process have completed before that process calls MPI_FILE_CLOSE.

If the file is deleted on close due to the AMODE setting, and there are other processes currently accessing the file, the status of the file and the behavior of future accesses by these processes are implementation dependent.

9.12. MPI-2 File I/O - Example Problem 1, Individual File Pointers

MPI-2 File I/O - Example Problem 1, Individual File Pointers

In serial I/O operations, a single file pointer points to the next location in the file that will be read from or written to, depending on the I/O operation being performed. With parallel I/O, two types of file pointers were discussed: individual and shared. Access to data using an individual file pointer means that each MPI process has its own pointer to the file that is updated only when that process performs an I/O operation. Access using a shared file pointer means that each MPI process uses a single file pointer that is updated when any of the processes performs an I/O operation.

A third method, explicit offset, was discussed in Section 9.9.1. - MPI_FILE_READ_AT. With this method, there is no file pointer and you give the exact location in the file from which to read or write a data value.

Below are two sample codes. Both use individual file pointers and one writes out data to a file using MPI parallel I/O calls, while the second reads in these values and outputs them to the screen.

Fortran:

```
      PROGRAM WRITE_INDIVIDUAL_POINTER

      INCLUDE 'MPIF.H'

      INTEGER COMM_SIZE,COMM_RANK,STATUS(MPI_STATUS_SIZE)
      INTEGER AMODE,INFO,FH,IERROR,ETYPE,FILETYPE,SIZE_INT
      INTEGER (KIND=MPI_OFFSET_KIND) DISP

      CALL MPI_INIT(IERROR)
      CALL MPI_COMM_SIZE(MPI_COMM_WORLD,COMM_SIZE,IERROR)
      CALL MPI_COMM_RANK(MPI_COMM_WORLD,COMM_RANK,IERROR)

      AMODE=IOR(MPI_MODE_CREATE,MPI_MODE_WRONLY)
      INFO=0

      CALL MPI_TYPE_EXTENT(MPI_INTEGER,SIZE_INT,IERROR)

      CALL MPI_FILE_OPEN(MPI_COMM_WORLD,'DATA.DAT',AMODE,INFO,FH,IERROR)

      DISP=COMM_RANK*SIZE_INT
      ETYPE=MPI_INTEGER
      FILETYPE=MPI_INTEGER

      CALL MPI_FILE_SET_VIEW(FH,DISP,ETYPE,FILETYPE,'NATIVE',
     &                       INFO,IERROR)

      CALL MPI_FILE_WRITE(FH,COMM_RANK,1,MPI_INTEGER,STATUS,IERROR)

      WRITE(6,*) 'HELLO FROM RANK',COMM_RANK,'I WROTE:',COMM_RANK,'.'

      CALL MPI_FILE_CLOSE(FH,IERROR)
      CALL MPI_FINALIZE(IERROR)

      STOP
      END
```

Some important items to note:

- With individual file pointers, you must call MPI_FILE_SET_VIEW.
- Recall that the displacement argument in MPI_FILE_SET_VIEW wants a displacement in bytes. Because you want to write out integer values, a call is

made to MPI_TYPE_EXTENT to get the size, in bytes, of the MPI_INTEGER datatype.

- Each individual file pointer will initially be set to the position specified by DISP. In this sample problem, only one data value is written. If multiple data values were written, each file pointer would be updated only when that process performed an I/O operation.

Example output from this code might look like

```
mpirun -np 8 write.ex
Hello from rank 4 I wrote: 4 .
Hello from rank 3 I wrote: 3 .
Hello from rank 5 I wrote: 5 .
Hello from rank 1 I wrote: 1 .
Hello from rank 0 I wrote: 0 .
Hello from rank 2 I wrote: 2 .
Hello from rank 6 I wrote: 6 .
Hello from rank 7 I wrote: 7 .
```

Looking at the contents of the file does not provide any insight, so we need to create a second program to read in the data and show that everything was written correctly.

C Code Example

Fortran:

```
      PROGRAM READ_INDIVIDUAL_POINTER
      INCLUDE 'MPIF.H'
      INTEGER COMM_SIZE,COMM_RANK,STATUS(MPI_STATUS_SIZE)
      INTEGER AMODE,INFO,FH,IERROR,ETYPE,FILETYPE,SIZE_INT
      INTEGER (KIND=MPI_OFFSET_KIND) DISP

      CALL MPI_INIT(IERROR)
      CALL MPI_COMM_SIZE(MPI_COMM_WORLD,COMM_SIZE,IERROR)
      CALL MPI_COMM_RANK(MPI_COMM_WORLD,COMM_RANK,IERROR)

      AMODE=MPI_MODE_RDONLY
      INFO=0

      CALL MPI_TYPE_EXTENT(MPI_INTEGER,SIZE_INT,IERROR)

      CALL MPI_FILE_OPEN(MPI_COMM_WORLD,'DATA.DAT',AMODE,INFO,FH,IERROR)

      DISP=COMM_RANK*SIZE_INT
      ETYPE=MPI_INTEGER
      FILETYPE=MPI_INTEGER

      CALL MPI_FILE_SET_VIEW(FH,DISP,ETYPE,FILETYPE,'NATIVE',
     &                       INFO,IERROR)

      CALL MPI_FILE_READ(FH,ITEST,1,MPI_INTEGER,STATUS,IERROR)
```

```
WRITE(6,*) 'HELLO FROM RANK',COMM_RANK,'I READ:',ITEST,'.'

CALL MPI_FILE_CLOSE(FH,IERROR)
CALL MPI_FINALIZE(IERROR)

STOP
END
```

Since in the first program each process set its pointer based on its rank, we can have each rank point to the same location. This should allow the process with rank 5 in this program to read in the value that was written by the process of the write program that had a rank of 5.

Example output from this code might look like

```
mpirun -np 8 read.ex
Hello from rank 6 I read: 6 .
Hello from rank 4 I read: 4 .
Hello from rank 0 I read: 0 .
Hello from rank 1 I read: 1 .
Hello from rank 2 I read: 2 .
Hello from rank 3 I read: 3 .
Hello from rank 5 I read: 5 .
Hello from rank 7 I read: 7 .
```

Notice that even though the processes performed their I/O in a different order than the write program, the correct values were read in because the displacement was calculated by the same formula in each program.

C Code Example

9.13. MPI-2 File I/O - Example Problem 2, Explicit Offset

MPI-2 File I/O - Example Problem 2, Explicit Offset

With individual file pointers, each process maintained its own file pointer that gets updated only when that process performed an I/O operation. Data access via a shared file pointer means that each process references a single file pointer that gets updated when any of the processes performs an I/O operation.

Data do not need to be accessed by file pointers, however. Programs can specify at what location in a file to read or write a value. This is done through explicit offsets. Rather than create and define a file pointer and then have the MPI_READ or MPI_WRITE reference the file pointer for the location at which to actually place the data, no pointer is created and the MPI calls include an additional argument, which is the position from the start of the file to perform the given I/O operation.

Below are two sample codes. Both use explicit offset. One writes out data to a file using MPI parallel I/O calls, while the second reads in these values and outputs them to the screen.

Fortran:

```
  PROGRAM EXPLICIT_WRITE

  INCLUDE 'MPIF.H'

  INTEGER COMM_SIZE,COMM_RANK,STATUS(MPI_STATUS_SIZE)
  INTEGER AMODE,INFO,FH,IERROR,SIZE_INT
  INTEGER (KIND=MPI_OFFSET_KIND) OFFSET

  CALL MPI_INIT(IERROR)
  CALL MPI_COMM_SIZE(MPI_COMM_WORLD,COMM_SIZE,IERROR)
  CALL MPI_COMM_RANK(MPI_COMM_WORLD,COMM_RANK,IERROR)

  AMODE=IOR(MPI_MODE_CREATE,MPI_MODE_WRONLY)
  INFO=0

  CALL MPI_TYPE_EXTENT(MPI_INTEGER,SIZE_INT,IERROR)

  CALL MPI_FILE_OPEN(MPI_COMM_WORLD,'DATA.DAT',AMODE,INFO,FH,IERROR)

  OFFSET=COMM_RANK*SIZE_INT

  CALL MPI_FILE_WRITE_AT(FH,OFFSET,COMM_RANK,1,MPI_INTEGER,
 &                       STATUS,IERROR)

  WRITE(6,*) 'HELLO FROM RANK',COMM_RANK,'I WROTE:',COMM_RANK,'.'

  CALL MPI_FILE_CLOSE(FH,IERROR)
  CALL MPI_FINALIZE(IERROR)

  STOP
  END
```

Some important items to note

- With explicit offsets, you do not define a file pointer with MPI_FILE_SET_VIEW.
- Each write operation has the additional argument OFFSET, which is the offset from the start of file, in bytes, where the data will be written.

Example output from this code might look like

```
  mpirun -np 8 write_explicit.ex
   Hello from rank 2 I wrote: 2 .
   Hello from rank 5 I wrote: 5 .
   Hello from rank 4 I wrote: 4 .
   Hello from rank 7 I wrote: 7 .
```

```
Hello from rank 3 I wrote: 3 .
Hello from rank 1 I wrote: 1 .
Hello from rank 0 I wrote: 0 .
Hello from rank 6 I wrote: 6 .
```

Looking at the contents of the file does not provide any insight, so you need to create a second program to read in the data and show that everything was written correctly.

C Code Example

Fortran:

```
PROGRAM EXPLICIT_READ
INCLUDE 'MPIF.H'

INTEGER COMM_SIZE,COMM_RANK,STATUS(MPI_STATUS_SIZE)
INTEGER AMODE,INFO,FH,IERROR,SIZE_INT
INTEGER (KIND=MPI_OFFSET_KIND) OFFSET

INTEGER STATUS(MPI_STATUS_SIZE)

CALL MPI_INIT(IERROR)
CALL MPI_COMM_SIZE(MPI_COMM_WORLD,COMM_SIZE,IERROR)
CALL MPI_COMM_RANK(MPI_COMM_WORLD,COMM_RANK,IERROR)

AMODE=MPI_MODE_RDONLY
INFO=0

CALL MPI_TYPE_EXTENT(MPI_INTEGER,SIZE_INT,IERROR)

CALL MPI_FILE_OPEN(MPI_COMM_WORLD,'DATA.DAT',AMODE,INFO,FH,IERROR)

OFFSET=COMM_RANK*SIZE_INT

CALL MPI_FILE_READ_AT(FH,OFFSET,ITEST,1,MPI_INTEGER,STATUS,IERROR)

WRITE(6,*) 'HELLO FROM RANK',COMM_RANK, 'I READ:',ITEST,'.'

CALL MPI_FILE_CLOSE(FH,IERROR)
CALL MPI_FINALIZE(IERROR)

STOP
END
```

Because each process in the first program wrote the data at a position based on its rank, we can have each rank point to the same location to read in the data. This should allow the process with rank 2 in this program to read in the value that was written by the process of the write program that had a rank of 2.

Example output from this code might look like

```
mpirun NP 8 read_explicit.ex
 Hello from rank 0 I read: 0 .
 Hello from rank 3 I read: 3 .
 Hello from rank 5 I read: 5 .
 Hello from rank 6 I read: 6 .
 Hello from rank 1 I read: 1 .
 Hello from rank 4 I read: 4 .
 Hello from rank 2 I read: 2 .
 Hello from rank 7 I read: 7 .
```

Notice that even though the processes performed their I/O in a different order than the write program, the correct values were read in because the offset was calculated by the same formula in each program.

C Code Example

9.14. Self Test

Parallel I/O Self Test

Now that you've finished this chapter, test yourself on what you've learned by taking the Self Test provided. Simply click on the Self Test link in the ACTION MENU above to get started.

9.15. Course Problem

Chapter 9 Course Problem

This chapter discussed the advanced topic of parallel I/O where all the processors can perform input and output simultaneously to the same file. Since most parallel programs (and serial for that matter) perform some sort of I/O, the concepts and routines in this chapter can prove quite useful.

Our Course Problem is not exempt from the need for parallel I/O. So far, the master (P0) has done all the input and output. Your task for this chapter will be to have *all* the processors perfom the output to the same binary data file. Since this task is formidable enough, the simpler version of the Course Problem found in Chapter 4 will be used.

Description

The initial problem implements a parallel search of an extremely large (several thousand elements) integer array. The program finds all occurrences of a certain integer, called the target, and writes all the array indices where the target was found to an output file. In addition, the program reads both the target value and all the array elements from an input file.

Exercise

Modify your code from Chapter 4 to implement the parallel writing of the target global indices to a binary file called "found.dat". The master should still read in the target and broadcast the value to the other processors. The master should still read in the entire array b and then scatter its contents to all the processors (including itself). But in this version of the course program, when a processor finds a target location it should write the global index directly to "found.dat" instead of sending it to the master for output as was done in the previous exercise.

Solution

When you have finished writing the code for this exercise, view our version of the Parallel I/O Code.

---

10. Parallel Mathematical Libraries

Parallel Mathematical Libraries

This chapter is a change of pace. In the preceding chapters, you have learned useful MPI routines that will allow you to write your own MPI programs to accomplish mathematical and scientific tasks in parallel. In this chapter, you will learn about existing (and popular and free) mathematical libraries that will implement common linear algebra algorithms for you **in parallel**.

You will not have to write your own MPI code to perform the parallel linear algebra tasks; you will instead call a subroutine.

Besides the obvious ease in just using a library call, it also keeps you from "reinventing the wheel." You will see that these libraries are complete, well written, well optimized (both serially and from a parallel point of view), and designed around excellent parallel algorithms.

The topics to be covered in this chapter are

- Introduction
- Serial and Parallel Mathematical Libraries
- How to Use a ScaLAPACK Routine: Step-by-Step
- Processor Grid Creation
- ScaLAPACK Data Distribution
- Case Study: Matrix-Vector Multiplication

10.1. Introduction to ScaLAPACK

Features of the ScaLAPACK Library

This section focuses on the ScaLAPACK parallel library. You will see in later sections that the ScaLAPACK library is in fact built upon and contains several other major libraries, offering users access to over a hundred mathematical routines that run in parallel.

ScaLAPACK has become the **de-facto** standard parallel numerical library primarily because of its portability and design. ScaLAPACK has been successfully installed on a variety of massively parallel processing (MPP) platforms, including the Cray T3E, SGI Origin 2000, IBM SP2, Intel Paragon, networks of workstations, and the recently developed Beowolf Clusters. But more than this portability, the real reason for the wide acceptance of ScaLAPACK is that it contains an extensive set of parallel routines for common **Linear Algebra** calculations that perform well and are scalable. If the number of processors used by a ScaLAPACK routine increases, the wall clock time for the calculations decreases, which is the entire point of parallel processing. ScaLAPACK's scalablity can also be viewed in a different manner - if the size of the problem (array dimensions) increases, with the same number of processors the parallel efficiency remains constant.

ScaLAPACK's success can also be attributed to the fact that it is built on the reliable, well-developed, LAPACK library which has been of use to programmers for decades. The LAPACK library contains **serial** linear algebra routines optimized for a variety of processors. In addition, ScaLAPACK includes two new libraries: PBLAS and PBLACS. As will be detailed later, PBLAS routines perform optimized, **parallel**, low-level linear algebra tasks. Users are insulated from the particular machine characteristics that the PBLAS routines use for optimization. PBLACS routines are responsible for communication between the processors, replacing the MPI coding you would otherwise have to do.

The ScaLAPACK developers made a helpful decision when they developed the syntax of the actual ScaLAPACK calls. This "user-friendly" idea was to make the parallel version of a serial LAPACK routine have the same name but just prefaced by a 'P'. Also, the names and meaning of the ScaLAPACK routine's arguments were made as similar as possible to the corresponding LAPACK routine arguments.

The final, and by far the most desirable, feature of ScaLAPACK is that by using its routines you **do not** have to write your own parallel processing code and **do not** have to develop or research a good parallel algorithm for performing a particular linear algebra task. (The latter is often a difficult process and is the subject of current research in CIS journals.) Strictly speaking, you need no knowledge of MPI or any other message-passing libraries. However, some message-passing experience is useful as background in understanding the PBLACS routines designed to send entire arrays (or subsections) from one processor to another.

ScaLAPACK Documentation

All the information you need to know about ScaLAPACK and its libraries is available from the documentation in the Netlib Repository.

The Netlib Repository contains not only the **FREE** source code of dozens of mathematical libraires but also documentation in many forms: subroutine descriptions, manuals, man pages, quick-reference cards, etc. The particular manuscripts that would be of use for this section are

- ScaLAPACK User's Guide
- Parallel Basic Linear Algebra Subprograms (PBLAS)
- A User's Guide to the BLACS

Finally, no introduction to Parallel Numerical Libraries would be complete without an acknowledgement to the computer scientists who developed the libraries and support materials of the Netlib Repository. The primary institutions involved are

- University of Tennessee, Knoxville
- Oak Ridge National Laboratory
- University of California, Berkeley

10.2. Serial and Parallel Mathematical Libraries

Serial and Parallel Mathematical Libraries

The Acronyms

First off, let's define some acronyms:

- BLAS:**B**asic **L**inear **A**lgebra **S**ubprograms
- PBLAS:**P**arallel **B**asic **L**inear **A**lgebra **S**ubprograms
- BLACS:**B**asic **L**inear **A**lgebra **C**ommunication **S**ubprograms
- LAPACK:**L**inear **A**lgebra **PACK**age
- ScaLAPACK:**Sca**lable **L**inear **A**lgebra **PACK**age

The Mathematical Library Hierarchy

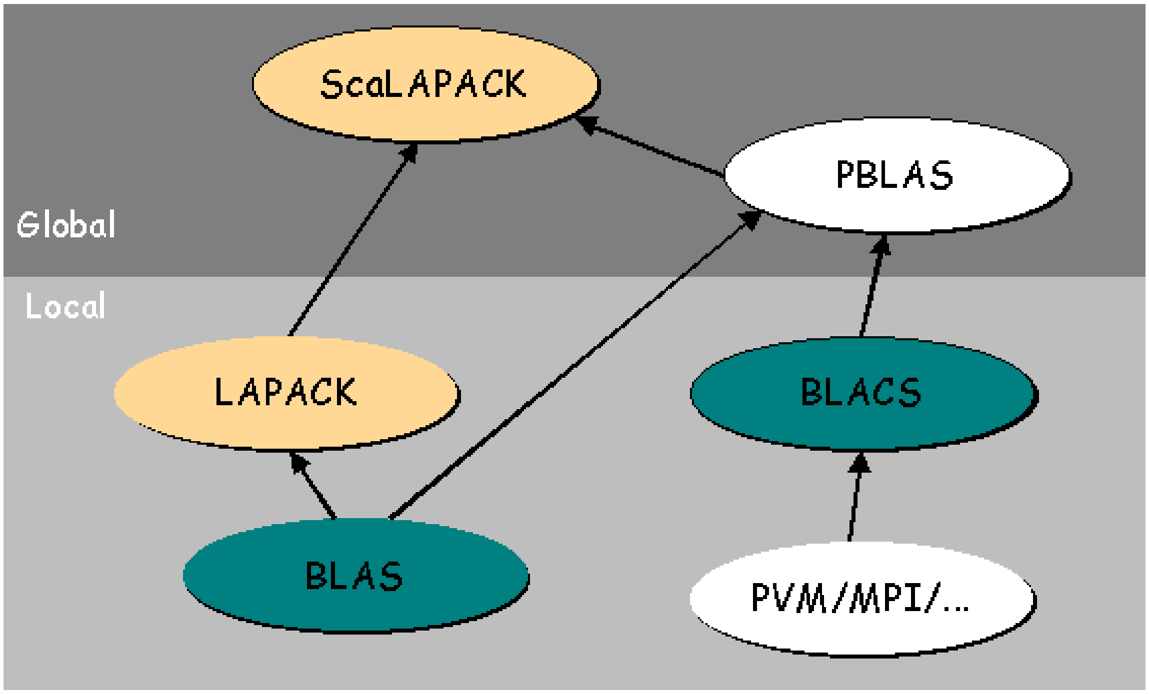


Figure 10.1. The Mathematical Library Hierarchy

In the above figure, the libraries lying below the "Local" keyword are serial libraries. Note that LAPACK contains and is built on BLAS. Each is vendor optimized and works only on **single processor** systems: vector supercomputers, workstations, or one processor of an MPP system. Typically, the optimization is performed through blocking algorithms designed to keep and reuse critical data in the lowest levels of the memory heirarchy: registers -> primary cache -> secondary cache -> local memory.

The libraries lying above the "Global" keyword are the new parallel libraries. In an analogous way to the serial libraries, ScaLAPACK contains and is built on PBLAS. These are the libaries that will be discussed and used in this section because they are used on MPP systems to perform linear algebra calculations in parallel.

The figure also shows that ScaLAPACK is built on BLACS and it must be because the BLACS routines transfer local data from one processor's memory to another. In reality, the BLACS routines are wrapper routines that call a lower-level message-passing library. Typically this is MPI itself! (The figure above is a bit confusing at this point, because BLACS and MPI are parallel libraries. They are drawn in the lower section to illustrate that the parallel libraries are built on top of them).

BLAS/PBLAS

These libraries contain serial and parallel versions of basic linear algebra procedures. Each contains routines that fall into three levels:

Level 1: Vector-Vector Operations

- Examples: swap, copy, addition, dot product, ...

Level 2: Matrix-Vector Operations

- Examples: multiply, rank-updates, outer-product, ...

Level 3: Matrix-Matrix Operations

- Examples: multiply, transpose, rank-updates, ...

All these levels of routines are designed to work with a variety of matrix types such as general, symmetric, complex, Hermetian, and triangular matrices. Note that sparse matrices are not supported.

LAPACK/ScaLAPACK

These libraries contain routines for more sophisticated linear algebra calculations. There are three types of advanced problems these libraries solve:

- Solution to a set of simultaneous linear equations
- Eigenvalue/Eigenvector problems
- Linear Least squares fitting

As with the basic libraries, many matrix types are supported as well as various algorithms for factorizing the matrices involved. These factorizations include LU, Cholesky, QR, LQ, Orthogonal, etc.

BLACS

As mentioned previously, the BLACS routines are used primarily to transfer data between processors. Specifically, the BLACS routines include point-to-point communication, broadcast routines, and "combination" routines for doing "global" calculations - summation, maximum, and minimum - with data residing on different processors. An attractive feature of the BLACS communication routines is that they are array-based: the data that is transferred is always an entire array or subarray.

The BLACS library also contains important routines for creating and examining the **processor grid**, which is expected and used by all PBLAS and ScaLAPACK routines. The processor grid is 2-D with its size and shape controlled by the user. A processor is identified not by its traditional MPI rank, but rather by its row and column number in the processor grid.

As can be seen in the processor grid diagram below (Figure 10.2), the 8 processors used by the program have been arranged in a 2x4 grid. Contained within each element of the grid is the MPI rank for a processor. For example, a processor with rank 6 has grid coordinates (**p,q**)=(1,2). Notice that both the row and column numbering begin with zero. For the example we have chosen to insert the processors "by-row". This grid was created by a call to a simple BLACS routine, which will be described in detail later.

	0	1	2	3
0	0	1	2	3
1	4	5	6	7

Figure 10.2. A grid created by a simple BLACS routine.

10.3. How to Use a ScaLAPACK Routine: Step-by-Step

How to Use a ScaLAPACK Routine: Step-by-Step

This section may be the shortest, but in some sense it is the most important. If you perform each step correctly, your ScaLAPACK routine should work!

ScaLAPACK Checklist

1. Write a scaled-down program using the equivalent serial LAPACK routine
   - Recommended, **not required**
   - Run using only one processor
   - Good for familiarizing the user with proper routine name and arguments. Parallel routine names and arguments will be similiar.
   - Good for syntax and logical debugging
2. Initialize the BLACS library for its use in the program
3. Create and use the BLACS processor grid
4. Distribute pieces of each global array over the processors in the grid
   - User does this by creating an **array descriptor vector** for each global array
   - Global array elements mapped in a 2-D blocked-cyclic manner onto the the processor grid
5. Have each processor initialize its local array with the correct values of the pieces of the global array it owns.
6. Call the ScaLAPACK routine!
7. Confirm/use the output of the ScaLAPACK routine
8. Release the processor grid and exit the BLACS library

10.4. Processor Grid Creation

Processor Grid Creation

BLACS Intialization Routines

The two routines described in this section will initialize the BLACS library by establishing a context within which data will be transferred. This is step 2 of the checklist outlined in Section 10.3. The following convention will be used for all routines in this section: if an argument is underlined, a value for it is returned by the routine; all others are input to the routine. In addition, all routines are subroutines unless their syntax shows a return type.

```
BLACS_PINFO(rank,nprocs)
```

- Primary purpose of calling this routine is to get the number of processors that the program will use (as specified on the command line when the program is run).
- Argument rank contains the normal MPI rank (first introduced in Section 3 - MPI Program Structure) of the processor that called the routine.
- Argument nprocs contains the total number of processors used.

```
BLACS_GET(-1,0,icontxt)
```

- BLACS_GET is actually a general-purpose, utility routine that will retrieve values for a set of BLACS internal parameters.
- With the arguments shown above, the BLACS_GET call will get the context (argument icontxt) for ScaLAPACK use. (Note: the first argument is ignored in this form of the call).
- The context of a parallel data communication library uniquely identifies all the BLACS data transfers as being connected with a particular ScaLAPACK routine. No other message-passing routines in the program can interfere with the BLACS communication. Context is completely equivalent to the MPI Communicator value explained earlier in this course.

BLACS Grid Routines

The first of these two BLACS routines actually creates the processor grid to specs. The second is a tool routine that confirms the grid was created successfully. Together, the calls to these routines make up step 3 of our checklist.

```
BLACS_GRIDINIT(icontxt,order,nprow,npcol)
```

- This routine creates the virtual processor grid according to values to which the arguments are set.

- icontxt= BLACS context for its message passing (first argument of almost every BLACS routine).
- order= determines how the processor ranks are mapped to the grid. If this argument is 'R', the mapping is done by rows; if it is 'C', the mapping is done by column.
- nprow= sets the number of rows in the processor grid.
- npcol= sets the number of clolumns in the processor grid.

For example, the grid shown in the diagram at the end of Section 10.2 was made with this call:

```
CALL BLACS_GRIDINIT(icontxt,'R',2,4)
```

This sample processor grid would be appropriate for a program using 8(2x4) processors. You can experiment with the size and shape of the process grid created in order to maximize the performance of the program.

```
BLACS_GRIDINFO(icontxt,nprow,npcol,myrow,mycol)
```

- The primary uses of this utility routine are to confirm the shape of the processor grid and, more importantly, for each processor to obtain its coordinates (myrow, mycol) in the processor grid. **GET IN THE HABIT OF USING GRID COORDINATES TO IDENTIFY A PROCESSOR!**
- Notice that nprow and npcol are returned to make sure the grid shape is what was desired.
- myrow = calling processor's row number in the processor grid.
- mycol = calling processor's column number in the processor grid.

10.5. ScaLAPACK Data Distribution

ScaLAPACK Data Distribution

Data Distribution Method

ScaLAPACK uses a two-dimensional block-cyclic distribution technique to parse out global array elements onto the processor grid. The block-cyclic technique was chosen because it gives the best load balance and maximum data locality for most of the ScaLAPACK algorithms. (See the "ScaLAPACK User's Guide" for a detailed justification of this choice and a comparison with other distribution methods.)

The 2-D block-cyclic distribution is accomplished by following these steps:

1. Divide up the global array into blocks with **mb** rows and **nb** columns. From now on, think of the global array as composed only of these blocks.

2. Give out the first row of array blocks across the first row of the processor grid in order. If you run out of processor grid columns, cycle back to the first column.
3. Repeat with the second row of array blocks, with the second row of the processor grid.
4. Continue for the remaining rows of array blocks.
5. If you run out of processor grid rows, cycle back to the first processor row and repeat.

The diagram below illustrates a 2-D, block-cyclic distribution of a 9x9 global array with 2x2 array blocks over a 2x3 processor grid. (The colors represent the ranks of the 6 different processors.)

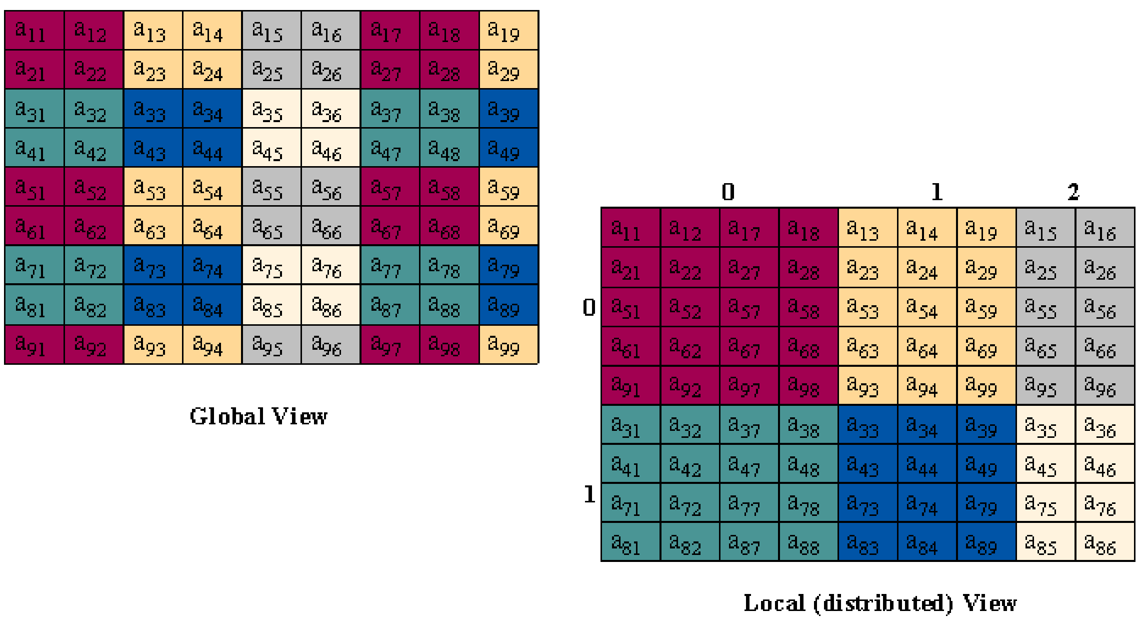


Figure 10.3. A 2-D, block-cyclic distribution of a 9x9 global array with 2x2 array blocks over a 2x3 processor grid

Notice that each processor has a different sized local array it has to fill with the correct global elements (Step 5 on the checklist).

- Processor(0,0): local array dimensions 5x4
- Processor(0,1): local array dimensions 5x3
- Processor(0,2): local array dimensions 5x2
- Processor(1,0): local array dimensions 4x4
- Processor(1,1): local array dimensions 4x3
- Processor(1,2): local array dimensions 4x2

Array Descriptor Vector

The array descriptor vector contains the information by which each ScaLAPACK routine determines the distribution of global array elements into local arrays owned by each processor. Below, each of the 9 elements of this vector is described. Two conventions are used in this description. First, for a global array A, it is traditional to name the array descriptor vector DESCA. Second, "_A" is read as "of the distributed global array A". The symbolic names and definitions of the elements of DESCA are given in the following table.

Symbolic Name	Description
DESCA(1)=dtype_A	type of matrix being distributed (1 for dense, general matrix)
DESCA(2)=icontxt	BLACS context
DESCA(3)=m_A	number of rows in the global array A
DESCA(4)=n_A	number of columns in the global array A
DESCA(5)=mb_A	number of rows in a block of A
DESCA(6)=nb_A	number of columns in a block of A
DESCA(7)=rsrc_A	processor grid row that has the first block of A (typically 0)
DESCA(8)=csrc_A	processor grid column that has the first block of A (typically 0)
DESCA(9)=lld	number of rows of the local array that stores the blocks of A (**l**ocal **l**eading **d**dimension). This element is processor- dependent

So for processor (0,2) in the preceding example, DESCA=(1,icontxt,9,9,2,2,0,0,5).

DESCINIT Utility Routine

Thankfully, you never have to explicitly fill up each element of DESCA. ScaLAPACK has provided a tool routine DESCINIT that will use its arguments to create the array descriptor vector DESCA for you. Once DESCINIT has been called (by each processor), step 4 of the checklist is completed.

```
DESCINIT(desc, m, n, mb, nb, rsrc, csrc, icontxt, lld, info)
```

- desc = the "filled-in" descriptor vector returned by the routine.
- arguments 2-9 = values for elements 2-9 of the descriptor vector (different order).
- info = status value returned to indicate if DESCINIT worked correctly. If info = 0, the routine call was sucessful. If info = -i, the ith argument had an illegal value.

Case Study: Matrix-Vector Multiplication

Problem Description

In the following two Fortran programs, the Matrix-Vector multiplication **Axb** will be performed first serially (step 1 of the checklist) and then in parallel using a ScaLAPACK routine. **A** is a 16x16 matrix and **b** is a 16 element vector. Both **A** and **b** have been filled with random integers in the range -50:50. Below is a diagram showing the operands.

$$A = \begin{bmatrix}
-45 & 49 & -45 & 48 & -38 & -17 & 5 & 17 & 29 & 30 & 0 & 10 & -31 & 43 & -12 & -3 \\
9 & -10 & 0 & -38 & -38 & -31 & 4 & 0 & 28 & -3 & -40 & 11 & -17 & 49 & 7 & -19 \\
45 & 18 & -16 & -6 & -38 & 0 & -24 & 25 & -7 & -31 & -30 & -29 & -45 & 31 & 31 & 33 \\
9 & -12 & 7 & -20 & -11 & -26 & -23 & 1 & 40 & 27 & -33 & -19 & -18 & 22 & -42 & 0 \\
-6 & 18 & -15 & 46 & 22 & 29 & -45 & -25 & -33 & -24 & 1 & -26 & -47 & -30 & -47 & 50 \\
-17 & -7 & -31 & -48 & -10 & 25 & -23 & 18 & 39 & -15 & 37 & 43 & 26 & 24 & 36 & 15 \\
11 & -6 & -23 & 50 & 47 & 48 & -17 & 9 & 40 & -20 & -43 & -38 & -1 & 20 & -11 & 38 \\
5 & -14 & 23 & 23 & 40 & -24 & 18 & 26 & -10 & 17 & 45 & 22 & -32 & -34 & -12 & 11 \\
-43 & -31 & 49 & 48 & 21 & -47 & 30 & -33 & 47 & 26 & -33 & -15 & 43 & 11 & -2 & -25 \\
0 & 15 & -50 & 35 & 18 & 9 & -12 & 45 & -16 & -25 & 11 & -47 & 14 & -2 & 41 & 47 \\
26 & -17 & -7 & 2 & 13 & -45 & 1 & -42 & -28 & 27 & 30 & -45 & 48 & -33 & -44 & -19 \\
21 & -28 & 10 & 25 & -46 & 5 & 7 & -8 & 49 & 41 & 22 & -49 & -6 & 24 & -17 & 8 \\
-41 & -37 & -15 & 26 & 29 & -43 & -4 & -49 & 13 & 7 & 9 & -38 & -10 & 48 & -28 & 50 \\
35 & 25 & 6 & 39 & 42 & -28 & 47 & -34 & 24 & 40 & 44 & 20 & 39 & 26 & -42 & -28 \\
-32 & -50 & -4 & 20 & 21 & -7 & 39 & 0 & 28 & 41 & 5 & 41 & -44 & 18 & 6 & -33 \\
49 & -25 & 0 & 39 & -10 & 41 & 13 & -34 & 23 & 49 & 15 & 27 & -49 & 16 & -40 & 28
\end{bmatrix}$$

$$b = \begin{bmatrix} -40 & 48 & 38 & -32 & -24 & -10 & 49 & -28 & 38 & 30 & -7 & 42 & -4 & -4 & -15 & 10 \end{bmatrix}^T$$

Figure 10.4. Operands for the Matrix-Vector multiplication example

Serial Solution

To perform the multiplication **A x b** on a single processor, you can use the tried-and-true LAPACK routine SGEMV. This is actually a BLAS level 2 routine, and a coding exists in the name itself. The 'S' indicates single-precision, the 'GE' indicates a general matrix, and the 'MV' indicates that the routine performs a matrix-vector multiplication. More specfically, SGEMV performs the operation

$$y = alpha*Ab+beta*y$$

where the *y* vector is the result of the operation and alpha and beta are constants chosen by the user. The syntax for the SGEMV call is

```
   SGEMV(trans,m,n,alpha,A,lda,b,incb,beta,y,incy)
```

where the arguments are

```
   trans   ='N', use normal matrix A in the calculations; ='Y', use
transpose of A
   m       number of rows in A
   n       number of columns of A
   alpha   scaling factor (usually 1.0)
   A       address of the beginning of matrix A [Can put for this
argument A or A(1,1)]
   lda     total number of rows specified when A was declared (leading
dimension)
   b       address of the beginning of vector b [Can put for this
argument b of b(1)]
   incb    stride for determining which elements of b to use (typically
1)
   beta    scaling/initialization factor (typically 0.0)
   y       on exit from routine,  vector y contains the result of the
multiplication
   incy    stride for determining which values of y are meaningful
(typically 1)
```

Serial MV Program

Below is the serial program that performs the matrix-vector multiplication **Ab**. For those unfamiliar, the declaration statements at the beginning of the code are in Fortran 90 syntax. In addition, the elements of **A** and the elements of **b** are stored in the text files "a.data" and "b.data" respectively. Notice how simple this program is: first the variables are declared, the operands are initialized, SGEMV is called, and the result is output.

```
program serial_mv
  real, dimension(16,16) :: a
  real, dimension(16) :: b,y

  open(unit=12,file="a.data")
  read(12,*) a
  open(unit=13,file="b.data")
  read(13,*) b

  call sgemv('n',16,16,1.0,a,16,b,1,0.0,y,1)

  print *,"product is ",y

end program serial_mv
```

The output from this program is

```
product is 4341., 3467., -5259., 1716., -5821., 750., -6561., -
237., 5149., -9212., -2387., 0., -1815., 4601. 3890., 836.
```

[Author's comment: It amazes me that from this large set of random numbers being multiplied and added that one element of the solution vector is exactly 0!]

Parallel Solution

Now we can put in all the proceeding information in this section and write a program that follows the checklist for using a ScaLAPACK routine. First off, the name of the parallel ScaLAPACK is already known: just put a 'P' in front of the serial routine. Thus, we will use the PSGEMV ScaLAPACK routine.

The first decision is what type of processor grid we want. The parallel code will run on 4 processors, which will be arranged on a **2x2** processor grid that looks like the one in Figure 10.5 below. Notice that the color coding is by rank of the processor: 0 is **red**, 1 is **green**, etc.

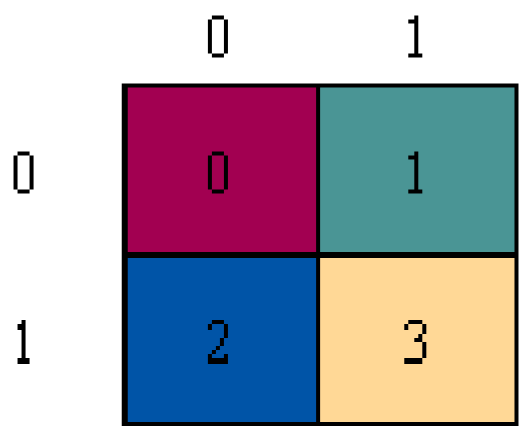


Figure 10.5. A **2x2** processor grid.

The second decision is to pick the blocking desired for the matrix **A** and the vector **b**. For the matrix **A** we choose **8x8** blocks of array elements; for the vector **b**, we choose **8x1** blocks of array elements. If the reader follows the steps involved in the 2-D block-cycle distribution method for the processor grid shown, the distribution of array elements ends up as shown in Figure 10.6, where the matrix **A** is on the left side of the diagram and the vector **b** is on the right. This type of blocking - especially of **A** - is quite common because it gives equal amounts of array elements to the local memories of each of the four processors.

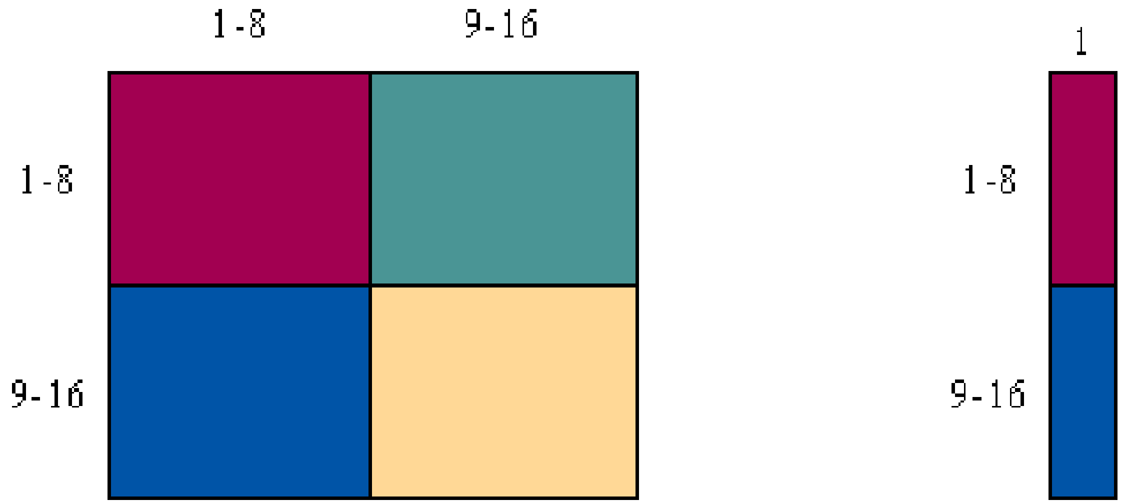


Figure 10.6. Distribution of array elements.

The third piece of knowledge we need is the detailed syntax of the PSGEMV routine, so it can be used correctly. The syntax that is shown below is that which is used when we to multiply the *entire* array **A** with the *entire* array **b**. Both SGEMV and PSGEMV have the capability of working with only parts of **A** and **b**, but that is the subject for another time.

```
PSGEMV(trans,m,n,alpha,la,1,1,desca,lb,1,1,descb,1,beta,ly,1,1,descy,1)
```

where the arguments are

```
   trans   ='N', use normal matrix A in the calculations; ='Y', use
transpose of A
   m       number of rows in the global array A
   n       number of columns in the global array A
   alpha   scaling factor (usually 1.0)
   la      local array containing the some of the elements of the
distrbuted,
           global array A (shape and contents of la change from
processor to processor)
   desca   array descriptor vector for the global array A (used by the
           routine to know what elements of A are on what processor)
   lb      local array containing some blocks of of the distributed,
global vectror b
   descb   array descriptor vector for the global vector b
   beta    scaling/initialization factor (typcially 0.0)
   ly      local array containing some elements of the resultant
product vector y
   descy   array descriptor vector for the global, resultant vector y
```

Because each of the 4 processors **must** call PSGEMV, the entire global arrays end up being given block by block to the routine.

Parallel MV program

The parallel program consists of different parts that perform certain of the steps outlined in the "ScaLAPACK routine checklist" given in Section 10.3. Shown below is the first part of the code in which variables are initialized, the BLACS library is initialized (Step 2), and the processor grid is created (Step 3).

```
program parallel_mv
  ! global arrays
  real, dimension(16,16) :: a
  real, dimension(16) :: b,y

  ! variables for BLACS initialization and processor grid creation
  integer iam,nprocs,ictxt,nprow,npcol,myrow,mycol

  !variables needed for distributing global arrays across the proc grid
  integer desca(9), descb(9),descy(9),m,n,mb,nb,rsrc,csrc
  integer llda,lldb,info

  ! local arrays
  real, dimension(8,8) :: la
  real, dimension(8) :: lb,ly

  !Initializing the BLACS library (STEP 2)
  call blacs_pinfo (iam,nprocs)
  call blacs_get(-1,0,ictxt)

  !Creating and using the processor grid (STEP 3)
  nprow=2; npcol=2
  call blacs_gridinit(ictxt,'r',nprow,npcol)
  call blacs_gridinfo(ictxt,nprow,npcol,myrow,mycol)
```

In this next part of the code DESCINIT is used to create the array descriptors (Step 4).

```
  ! Making the array descriptor vectors (STEP 4)
  m=16; n=16
  mb=8; nb=8
  rsrc=0; csrc=0
  llda=8
  call descinit(desca,m,n,mb,nb,rsrc,csrc,ictxt,llda,info)
  n=1; nb=1; lldb=8
  call descinit(descb,m,n,mb,nb,rsrc,csrc,ictxt,lldb,info)
  call descinit(descy,m,n,mb,nb,rsrc,csrc,ictxt,lldb,info)

  ! Filling the global arrays A,b
  open(unit=12,file="a.data")
  read(12,*) a
  open(unit=13,file="b.data")
  read(13,*) b
```

In this third part of the parallel code, each processor (identified by its processor grid coordinates fills up its own local array with the correct quadrant of **A** and the correct half of **b** (STEP 5).

```
! Each processor fills in its local arrays with correct elements
! from the global arrays (STEP 5)
if(myrow.eq.0.and.mycol.eq.0) then
  do i_loc=1,8
    do j_loc=1,8
      la(i_loc,j_loc)=a(i_loc,j_loc)
    end do
    lb(i_loc)=b(i_loc)
  end do
end if

if(myrow.eq.1.and.mycol.eq.0) then
  do i_loc=1,8
    do j_loc=1,8
      la(i_loc,j_loc)=a(i_loc+llda,j_loc)
    end do
    lb(i_loc)=b(i_loc+lldb)
  end do
end if

if(myrow.eq.0.and.mycol.eq.1) then
  do i_loc=1,8
    do j_loc=1,8
      la(i_loc,j_loc)=a(i_loc,j_loc+llda)
    end do
  end do
end if

if(myrow.eq.1.and.mycol.eq.1) then
  do i_loc=1,8
    do j_loc=1,8
      la(i_loc,j_loc)=a(i_loc+llda,j_loc+llda)
    end do
  end do
end if
```

Finally, PSGEMV is ready to be called and is (STEP 6). This last part of the code then prints out the the resultant vector y, which is contained only on two of the processors (STEP 7) and the last statements exit the BLACS library (Step 8).

```
! Call the ScaLAPACK routine (STEP 6)
n=16
call
psgemv('n',m,n,1.0,la,l,l,desca,lb,l,l,descb,1,0.0,ly,l,l,descy,l)

! Each processor prints out its part of the product vector y (STEP 7)
if(myrow.eq.0.and.mycol.eq.0) then
  do i=1,8
    print *,'PE:',myrow,mycol,' y(',i,')=',ly(i)
```

```
    end do
  end if

  if(myrow.eq.1.and.mycol.eq.0) then
    do i=1,8
      print *,'PE:',myrow,mycol,' y(',i+lldb,')=',ly(i)
    end do
  end if

  ! Release the proc grid and BLACS library (STEP 8)
  call blacs_gridexit(ictxt)
  call blacs_exit(0)

end program parallel_mv
```

The output of the parallel code is shown below and, as you can see, it agrees with the output of the serial code **AS IT SHOULD**.

```
  PE: 0,0  y( 1 )= 4341.
  PE: 0,0  y( 2 )= 3467.
  PE: 0,0  y( 3 )= -5259.
  PE: 0,0  y( 4 )= 1716.
  PE: 0,0  y( 5 )= -5821.
  PE: 0,0  y( 6 )= 750.
  PE: 0,0  y( 7 )= -6561.
  PE: 0,0  y( 8 )= -237.
  PE: 0,0  y( 9 )= 5149.
  PE: 0,0  y( 10 )= -9212.
  PE: 0,0  y( 11 )= -2387.
  PE: 0,0  y( 12 )= 0.
  PE: 0,0  y( 13 )= -1815.
  PE: 0,0  y( 14 )= 4601.
  PE: 0,0  y( 15 )= 3890.
  PE: 0,0  y( 16 )= 836.
```

Coda:

A proper ending to this section is to list the key points you can glean from the parallel matrix-vector program just discussed in terms of using the ScaLAPACK library routines for your own work.

- Need to do all the preparatory steps before simply calling the ScaLAPACK routine!
- Each processor **must** call and contribute to the ScaLAPACK routine.
- When comparing the serial and parallel matrix-vector multiplication codes it can be seen that the global arrays **A**,**b**, and ***y*** are used in the SGEMV call, while it is local arrays **la**, **lb**, **ly** *and* their array descriptor vectors that are used in the PSGEMV call. This is an example of an overall strategy for parallel programming called **data decomposition**, in which each processor holds part of the data and does calculations with part of the data.

- In general, the size, shape, and contents of the local arrays will be different for each processor.

10.7. Self Test

Parallel Mathematical Libraries Self Test

Now that you've finished this chapter, test yourself on what you've learned by taking the Self Test provided. Simply click on the Self Test link in the ACTION MENU above to get started.

10.8. Course Problem

Chapter 10 Course Problem

This chapter is unique in that it is not about MPI. Instead, it describes an alternative to writing your own MPI code which utilizes routines from a parallel library. Specifically, the ScaLAPACK library which provides parallel routines for Linear Algebra calculations. Unfortunately, there is no ScaLAPACK routine for searching an array so there is no exercise for this chapter. Take the chapter off!

---

11. Portability Issues

Portability Issues

The MPI standard was defined in May of 1994. This standardization effort was a response to the many incompatible versions of parallel libraries that were in existence at that time. If you write a parallel program in compliance with the MPI standard, you should be able to compile it on any vendor's MPI compiler. However, slight differences exist among the various implementations of MPI because the standard's authors did not specify how some routines should be implemented. These differences *may* lead to an "unsafe" program that behaves differently on one system than it does on another. Several outcomes are consistent with the MPI specification and the actual outcome depends on the precise timing of events. In order to complete successfully, an "unsafe" program may require resources that are not guaranteed by the MPI implementation of the system on which it is running.

Because of the varying MPI implementations, it is important to be aware of the issues you should consider when writing code that you want to be portable. Some of these issues are

- **Buffering Assumptions.** In standard mode, blocking sends and receives should not be assumed to be buffered. The reason for this is that buffer memory is finite and all computers will fail under sufficiently large communication loads. If you write a program using buffering assumptions, it will work under some conditions and fail under others. Hence, it is not portable.

- **Barrier Synchronization Assumptions for Collective Calls.** In MPI, collective communications are always assumed to be blocking. However, your program should not depend on whether collective communication routines, like broadcast commands, act as barrier synchronizations. An MPI implementation of collective communications may or may not have the effect of barrier synchronization. One obvious exception to this is the MPI_BARRIER routine.
- **Communication Ambiguities.** When writing a program, you should make sure that messages are matched by the intended receive call. Ambiguities in the communication specification can lead to incorrect or non-deterministic programs if race conditions arise. Use the message tags and communicators provided by MPI to avoid these types of problems.

11.1. Course Problem

Chapter 11 Course Problem

This brief chapter points out the most important feature of MPI: that it has source code portability. This means that once you get an MPI program working on one parallel computer, you can transfer the source code to a new parallel computer, recompile and run your code and it will produce the same results.

Exercise

Take any of the programs you have written for the previous chapter exercises and run them on several parallel platforms with MPI installed. You should see the same answers. (Recall the Parallel I/O program used routines that are part of MPI-2).

---

12. Program Performance

MPI Program Performance

Defining the performance of a parallel program is more complex than simply optimizing its execution time. This is because of the large number of variables that can influence a program's behavior. Some of these variables are

- the number of processors
- the size of the data being worked on
- interprocessor communications limits

- available memory

This chapter will discuss various approaches to performance modeling and how to evaluate performance models with empirical performance data taken by data collecting performance tools.

12.1. Introduction to Performance Modeling

Introduction to Performance Modeling

In this section, three metrics that are commonly used to measure performance are introduced . They are

1. execution time
2. efficiency
3. speedup

Then, three simple models used to roughly estimate a parallel program's performance are discussed. These approaches should be taken as approximate measures of a parallel program's performance. More comprehensive ways of measuring performance will be discussed in a later section.

12.1.1. Performance Metrics

Performance Metrics

An obvious performance parameter is a parallel program's **execution time**, or what is commonly referred to as the *wall-clock time*. The execution time is defined as the time elapsed from when the first processor starts executing a problem to when the last processor completes execution.

It is sometimes useful to have a metric that is independent of the problem size. Two measures that are independent of problem size are **relative efficiency** and **relative speedup**. Relative efficiency is defined as $T_1/(P*T_p)$, where T_1 is the execution time on one processor and T_p is the execution time on P processors. Relative speedup is defined as T_1/T_p. Often an algorithm-independent definition of efficiency and speedup is needed for comparison purposes. These measures are called **absolute efficiency** and **absolute speedup** and they can be defined by making T_1 the execution time on one processor of the fastest sequential algorithm. When the terms efficiency and speedup are used without qualifiers, they usually refer to absolute efficiency and absolute speedup, respectively.

Note that it is possible for efficiencies to be greater than 1 and speedups to be greater than P. For example, if your problem size is such that your arrays do not fit in memory and/or cache in the serial code, but do fit in memory and/or cache when run on multiple processors, then you can have an additional speedup because your program is now working with fast memory.

12.1.2. Simple Models

Simple Models

The following simple models can be used to roughly estimate a parallel program's performance:

- **Amdahl's Law.** Stated simply, Amdahl's Law is: *if the sequential component of an algorithm accounts for 1/s of the program's execution time, then the maximum possible speedup that can be achieved on a parallel computer is s.* For example, if the sequential component is 10 percent, then the maximum speedup that can be achieved is 10. Amdahl's Law is not usually relevant for estimating a program's performance because it does not take into account a programmer's ability to overlap computation and communication tasks in an efficient manner.
- **Extrapolation from observation.** This model presents a single number as evidence of enhanced performance. Consider the following example: *An algorithm is implemented on parallel computer X and achieves a speedup of 10.8 on 12 processors with problem size N = 100.* However, a single performance measure serves only to determine performance in a narrow region of the parameter space and may not give a correct picture of an algorithm's overall performance.
- **Asymptotic analysis.** For theoretical ease, performance is sometimes characterized in a large limit. You may encounter the following example: *Asymptotic analysis reveals that the algorithm requires order((N/P)*log(N/P)) time on P processors, where N is some parameter characterizing the problem size.* This analysis is not always relevant, and can be misleading, because you will often be interested in a regime where the lower order terms are significant.

12.2. Developing Better Models

Developing Better Models

Better qualitative models than those described in the previous section can be developed to characterize the performance of parallel algorithms. Such models explain and predict the behavior of a parallel program while still abstracting away many technical details. This gives you a better sense of how a program depends on the many parameters that can be varied in a parallel computation. One such model, **Scalability Analysis**, consists of examining how a given metric (execution time, efficiency, speedup) varies with a program parameter. Questions you might ask from this model are

- How does efficiency vary with increasing problem size? (Fixed number of processors.)
- How does efficiency vary with the number of processors? (Scalability with fixed problem size.) A specific question of this type would be: What is the fastest way to solve problem A on computer X? (In this case one optimizes a given metric, keeping the problem size fixed.)
- How do you vary the number of processors with the problem size to keep the execution time roughly constant?

Although qualitative models are quite useful, quantitative models can provide a more precise description of performance and should be used for serious examinations of performance. The following example describes a quantitative model used to examine the metric *execution time*.

Example of a Quantitative Model:

The execution time, $\mathbf{T_e}$, is given by, $\mathbf{T_e = T_{comp} + T_{comm} + T_{idle}}$, where the execution time is divided between computing, communicating, or sitting idle, respectively. It is important to understand how the execution time depends on programming variables such as the size of the problem, number of processors, etc.

- **Computation time, $\mathbf{T_{comp}}$**: The computation time is the time a single processor spends doing its part of a computation. It depends on the problem size and specifics of the processor. Ideally, this is just $\mathbf{T_{serial}/P}$, but it may be different depending upon the parallel algorithm you are using.
- **Communication time, $\mathbf{T_{comm}}$:** The communication time is the part of the execution time spent on communication between processors. To model this, you start from a model of the time for a single communication operation. This time is usually broken up into two parts, $\mathbf{T_{comm,op} = T_l + T_m}$. The first part is the time associated with initializing the communication and is called the latency, $\mathbf{T_l}$. The second part is the time it takes to send a message of length $\mathbf{m}$, $\mathbf{T_m}$. $\mathbf{T_m}$ is given by $\mathbf{m/B}$ where $\mathbf{B}$ is the physical bandwidth of the channel (usually given in megabytes per second). So a simple model of communications, which assumes that communication cannot be overlapped with other operations would be: $\mathbf{T_{comm}=N_{messages} \times (T_l+<m>/B)}$ where $\mathbf{<m>}$ is the average message size and $\mathbf{N_{messages}}$ is the number of messages required by the algorithm. The last two parameters depend on the size of the problem, number of processors, and the algorithm you are using. Your job is to develop a model for these relationships by analyzing your algorithm. Parallel programs implemented on systems that have a large latency cost should use algorithms that minimize the number of messages sent.
- **Idle time, $\mathbf{T_{idle}}$:** When a processor is not computing or communicating, it is idle. Good parallel algorithms try to minimize a processor's idle time with proper load balancing and efficient coordination of processor computation and communication.

12.3. Evaluating Implementations

Evaluating Implementations

Once you implement a parallel algorithm, its performance can be measured experimentally and compared to the model you developed. When the actual performance differs from the predictions of your model, you should first check to make sure you did both the performance model and the experimental design correctly and that they measure the same thing. If the performance discrepancy persists, you should check for **unaccounted-for overhead** and **speedup anomalies**.

If an implementation has unaccounted-for overhead, then any of the following may be the reason:

- **Load imbalances**: An algorithm may suffer from computation or communication imbalances among processors.
- **Replicated computation**: Disparities between observed and predicted times can signal deficiencies in implementation. For example, you fail to take into account the need to parallelize some portion of a code.
- **Tool/algorithm mismatch**: The tools used to implement the algorithm may introduce inefficiencies. For example, you may call a slow library subroutine.
- **Competition for bandwidth**: Concurrent communications may compete for bandwidth, thereby increasing total communication costs.

If an implementation has speedup anomalies, meaning that it executes *faster* than expected, then any of the following may be the reason:

- **Cache effects**: The cache, or fast memory, on a processor may get used more often in a parallel implementation causing an unexpected decrease in the computation time.
- **Search anomalies**: Some parallel search algorithms have search trees that search for solutions at varying depths. This can cause a speedup because of the fundamental difference between a parallel algorithm and a serial algorithm.

12.4. Performance Tools

Performance Tools

The previous section emphasized the importance of constructing performance models and comparing these models to the actual performance of a parallel program. This section discusses the tools used to collect empirical data used in these models and the issues you must take into account when collecting the data.

You can use several data collection techniques to gather performance data. These techniques are

- **Profiles** show the amount of time a program spends on different program components. This information can be used to identify bottlenecks in a program. Also, profiles performed for a range of processors or problem sizes can identify components of a program that do not scale. Profiles are limited in that they can miss communication inefficiencies.
- **Counters** are data collection subroutines which increment whenever a specified event occurs. These programs can be used to record the number of procedure calls, total number of messages sent, total message volume, etc. A useful variant of a counter is a timer which determines the length of time spent executing a particular piece of code.
- **Event traces** contain the most detailed program performance information. A trace based system generates a file that records the significant events in the running of a program. For instance, a trace can record the time it takes to call a procedure or send a message. This kind of data collection technique can generate huge data files that can themselves perturb the performance of a program.

When you are collecting empirical data you must take into account

- **Accuracy.** In general, performance data obtained using sampling techniques is less accurate than data obtained using counters or timers. In the case of timers, the accuracy of the clock must also be considered.
- **Simplicity.** The best tools, in many circumstances, are those that collect data automatically with little or no programmer intervention and provide convenient analysis capabilities.
- **Flexibility.** A flexible tool can easily be extended to collect additional performance data or to provide different views of the same data. Flexibility and simplicity are often opposing requirements.
- **Intrusiveness.** Unless a computer provides hardware support, performance data collection inevitably introduces some overhead. You need to be aware of this overhead and account for it when analyzing data.
- **Abstraction.** A good performance tool allows data to be examined at a level of abstraction appropriate for the programming model of the parallel program.

12.5. Finding Bottlenecks with Profiling Tools

Finding Bottlenecks with Profiling Tools

Bottlenecks in your code can be of two types:

- computational bottlenecks (slow serial performance)
- communications bottlenecks

Tools are available for gathering information about both. The simplest tool to use is the MPI routine MPI_WTIME which can give you information about the performance of a particular section of your code. For a more detailed analysis, you can typically use any of a number of performance analysis tools designed for either serial or parallel codes. These are discussed in the next two sections.

12.5.1. Serial Profiling Tools

Serial Profiling Tools

We discuss the serial tools first since some of these tools form the basis for more sophisticated parallel performance tools. Also, you generally want to start from a highly optimized serial code when converting to a parallel implementation. Some useful serial performance analysis tools include Speedshop (ssrun, SGI) and Performance Application Programming Interface (PAPI, many platforms). The Speedshop and PAPI tools use hardware event counters within your CPU to gather information about the performance of your code. Thus, they can be used to gather information without recompiling your original code. PAPI also provides the low-level interface for another tool called HPMcount.

ssrun (Speedshop)

ssrun (Speedshop) is a popular performance tool available on SGI platforms that periodically samples the state of the program counter and stack, and writes this information to a file for later analysis. The sampling interval can be based upon system timers, upon hardware events (e.g. instruction or data cache misses), or upon entry and exit into blocks of code. This allows you to determine both the cause of the poor performance and which subroutines are responsible for the poor performance.

Usage:

```
ssrun [ssrun options] command [command arguments]
```

The *ssrun* options for selecting the sampling interval are listed below. The output of *ssrun* has been converted to text using *prof*. Where available, the hypertext links give some sample output for a simple program.

***ssrun* option**	**time base**	**Comments/Description**
-usertime	30 ms timer	Fairly coarse resolution. The experiment runs quickly and the output file is small. Some bugs are noted in *speedshop(1)*.
-pcsamp[x] -fpcsamp[x]	10 ms timer 1 ms timer	Moderately coarse resolution. Emphasizes functions that cause cache misses or page faults. Add suffix x for 32 bit counts.

-fgi_hwc	instructions 6553 instructions	emphasizes functions that execute many instructions.
-cy_hwc -fcy_hwc	16411 clocks 3779 clocks	Fine-grain resolution based on elapsed cycles. This emphasizes functions with cache misses and mispredicted branches.
-ic_hwc -fic_hwc	2053 icache miss 419 icache miss	Emphasizes code that doesn't fit in the L1 cache.
-isc_hwc -fisc_hwc	131 scache miss 29 scache miss	Emphasizes code that doesn't fit in the L2 cache.
-dc_hwc -fdc_hwc	2053 dcache miss 419 dcache miss	Emphasizes code that causes L1 cache data misses.
-dsc_hwc -fdsc_hwc	131 scache miss 29 scache miss	Emphasizes code that causes L2 cache data misses.
-tlb_hwc -ftlb_hwc	257 TLB misses 53 TLB misses	Emphasizes code that causes page faults.
-gfp_hwc -fgfp_hwc	32771 fp instructions 6553 fp instructions	Emphasizes code that performs heavy FP calculation.
-prof_hwc	user-set	Hardware counter and overflow values from counters named in environment variables.
-fpe	floating point exceptions	Creates a trace file that records all floating point exceptions.

PAPI

The Performance API (PAPI) project specifies a standard application programming interface (API) for accessing hardware performance counters available on most modern microprocessors. The PAPI provides two interfaces to the underlying counter hardware; a simple, high level interface for the acquisition of simple measurements and a fully programmable, low level interface directed towards users with more sophisticated needs. The low level PAPI deals with hardware events in groups called EventSets. EventSets reflect how the counters are most frequently used, such as taking simultaneous measurements of different hardware events and relating them to one another. For example, relating cycles to memory references or flops to level 1 cache misses can indicate poor locality and memory management. In addition, EventSets allow a highly efficient implementation which translates to more detailed and accurate measurements. EventSets are fully programmable and have features such as guaranteed thread safety, writing of counter values, multiplexing and notification on threshold crossing, as well as processor specific features. The high level interface simply provides the ability to start, stop and read specific events, one at a time. In addition, the PAPI provides portability across different platforms. It uses the same routines with similar argument lists to control and access the counters for every architecture.

12.5.2. Parallel Profiling Tools

Parallel Profiling Tools

Some MPI aware parallel performance analysis tools include Vampir (multiple platforms), DEEP/MPI and HPMcount (IBM SP3 and Linux). In some cases, profiling information can be gathered without recompiling your code.

In contrast to the serial profiling tools, the parallel profiling tools usually require you to instrument your parallel code in some fashion. Vampir falls into this category. Others can take advantage of hardware event counters.

Vampir is available on all major MPI platforms. It includes a MPI tracing and profiling library (Vampirtrace) that records execution of MPI routines, point-to-point and collective communications, as well as user-defined events such as subroutines and code blocks.

HPMcount, which is based upon PAPI can take advantage of hardware counters to characterize your code. HPMcount is being developed for performance measurement of applications running on IBM Power3 systems but it also works on Linux. It is in early development.

12.6. Self Test

Program Performance Self Test

Now that you've finished this chapter, test yourself on what you've learned by taking the Self Test provided. Simply click on the Self Test link in the ACTION MENU above to get started.

12.7. Course Problem

Chapter 12 Course Problem

In this chapter, the broad subject of parallel code performance is discussed both in terms of theoretical concepts and some specific tools for measuring performance metrics that work on certain parallel machines. Put in its simplest terms, improving code performance boils down to speeding up your parallel code and/or improving how your code uses memory.

As you have learned new features of MPI in this course, you have also improved the performance of the code. Here is a list of performance improvements so far:

- Using Derived Datatypes instead of sending and receiving the separate pieces of data
- Using Collective Communication routines instead of repeating/looping individual sends and receives
- Using a Virtual Topology and its utility routines to avoid extraneous calculations
- Changing the original master-slave algorithm so that the master also searches part of the global array (The slave rebellion: Spartacus!)
- Using "true" parallel I/O so that *all* processors write to the output file simultaneously instead of just one (the master)

But more remains to be done - especially in terms of how the program affects memory. And that is the last exercise for this course. The problem description is the same as the one given in Chapter 9 but you will modify the code you wrote using what you learned in this chapter.

Description

The initial problem implements a parallel search of an extremely large (several thousand elements) integer array. The program finds all occurrences of a certain integer, called the target, and writes all the array indices where the target was found to an output file. In addition, the program reads both the target value and all the array elements from an input file.

Exercise

Modify your code from Chapter 9 so that it uses dynamic memory allocation to use only the amount of memory it needs and only for as long as it needs it. Make both the arrays a and b ALLOCATABLE and connect them to memory properly. You may also assume that the input data file "b.data" now has on its first line the number of elements in the global array b. The second line now has the target value. The remaining lines are the contents of the global array b.

Solution

When you have finished writing the code for this exercise, view our version of the Performance Code.

13. Parallel Algorithms Underlying MPI Implementations

Looking Under the Hood -- Parallel Algorithms Underlying MPI Implementations

This chapter looks at a few of the parallel algorithms underlying the implementations of some simple MPI calls. The purpose of this is not to teach you how to "roll your own" versions of these routines, but rather to help you start thinking about algorithms in a parallel fashion. First, the method of **recursive halving and doubling**, which is the algorithm underlying operations such as broadcasts and reduction operations, is discussed. Then, specific examples of parallel algorithms that implement message passing are given.

13.1. Recursive Halving and Doubling

Recursive Halving and Doubling

To illustrate recursive halving and doubling, suppose you have a vector distributed among **p** processors, and you need the sum of all components of the vector in each processor, i.e., a sum reduction. One method is to use a tree-based algorithm to compute the sum to a single processor and then broadcast the sum to every processor.

Assume that each processor has formed the partial sum of the components of the vector that it has.

Step 1: Processor 2 sends its partial sum to processor 1 and processor 1 adds this partial sum to its own. Meanwhile, processor 4 sends its partial sum to processor 3 and processor 3 performs a similar summation.

Step 2: Processor 3 sends its partial sum, which is now the sum of the components on processors 3 and 4, to processor 1 and processor 1 adds it to its partial sum to get the final sum across all the components of the vector.

At each stage of the process, the number of processes doing work is cut in half. The algorithm is depicted in the Figure 13.1 below, where the solid arrow denotes a send operation and the dotted line arrow denotes a receive operation followed by a summation.

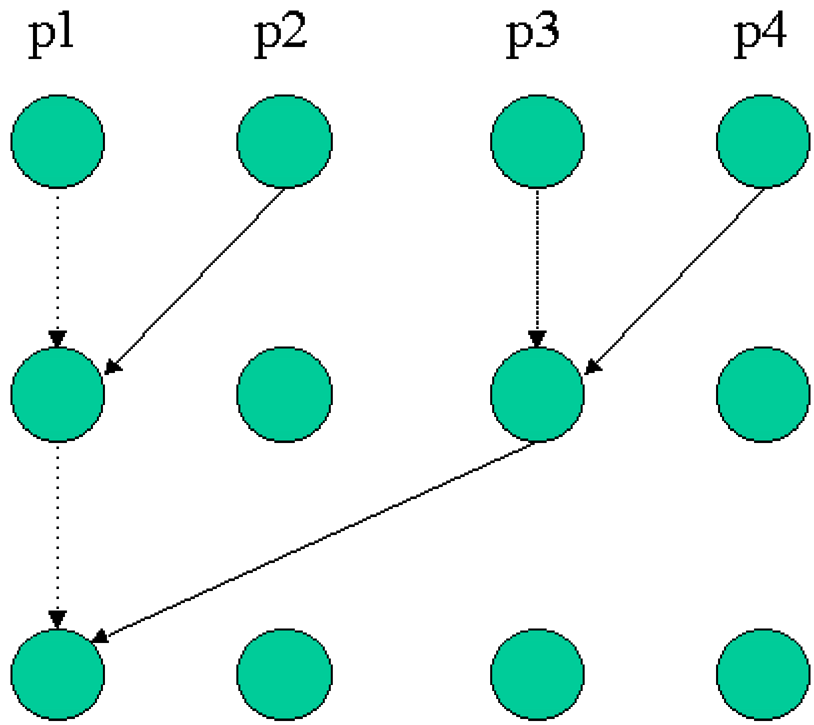


Figure 13.1. Summation in log(N) steps.

Step 3: Processor 1 then must broadcast this sum to all other processors. This broadcast operation can be done using the same communication structure as the summation, but in reverse. You will see pseudocode for this at the end of this section. Note that if the total number of processors is N, then only 2 log(N) (log base 2) steps are needed to complete the operation.

There is an even more efficient way to finish the job in only log(N) steps. By way of example, look at the next figure containing 8 processors. At each step, processor i and processor i+k send and receive data in a pairwise fashion and then perform the summation. k is iterated from 1 through N/2 in powers of 2. If the total number of processors is N, then log(N) steps are needed. As an exercise, you should write out the necessary pseudocode for this example.

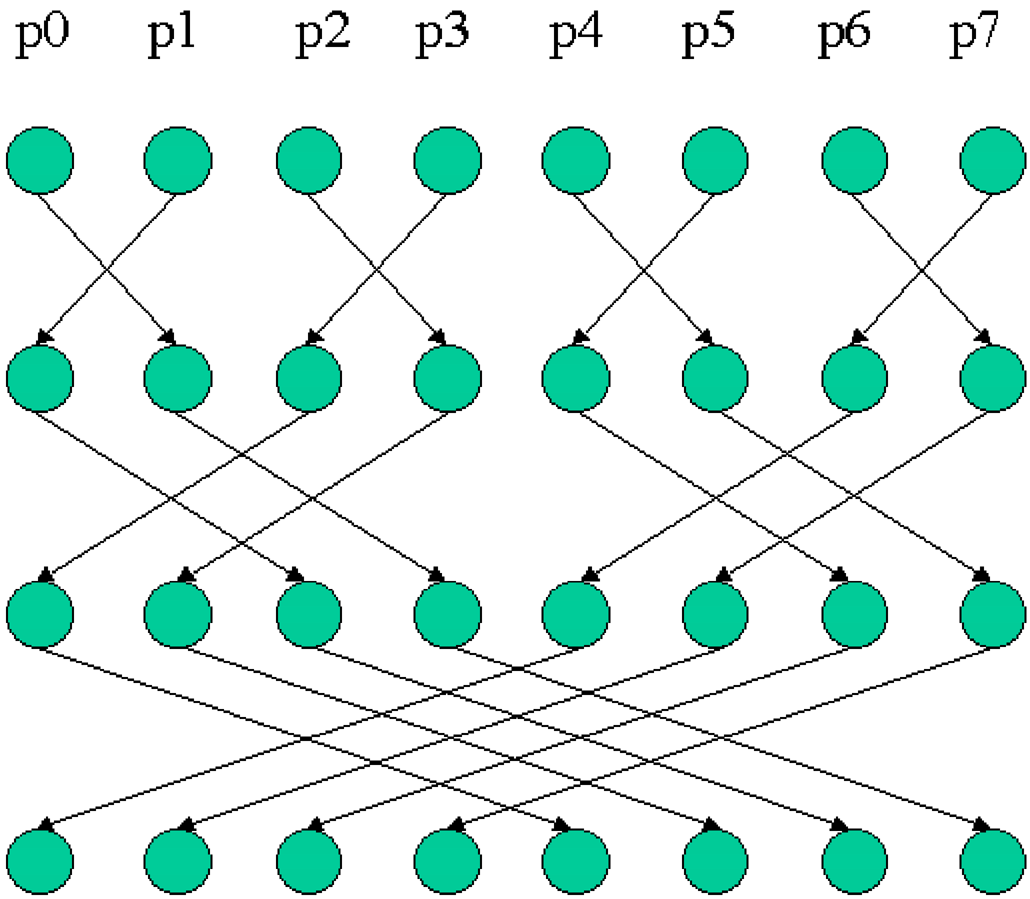


Figure 13.2. Summation to all processors in log(N) steps.

What about adding vectors? That is, how do you add several vectors component-wise to get a new vector? The answer is, you employ the method discussed earlier in a component-wise fashion. This fascinating way to reduce the communications and to avoid abundant summations is described next. This method utilizes the recursive halving and doubling technique and is illustrated in Figure 13.3.

Suppose there are 4 processors and the length of each vector is also 4.

Step 1: Processor p0 sends the first two components of the vector to processor p1, and p1 sends the last two components of the vector to p0. Then p0 gets the partial sums for the last two components, and p1 gets the partial sums for the first two components. So do p2 and p3.

Step 2: Processor p0 sends the partial sum of the third component to processor p3. Processor p3 then adds to get the total sum of the third component. Similarly, processor 1, 2, and 4 find the total sums of the 4th, 2nd, and 1st components, respectively. Now the sum of the vectors are found and the components are stored in different processors.

Step 3: Broadcast the result using the reverse of the above communication process.

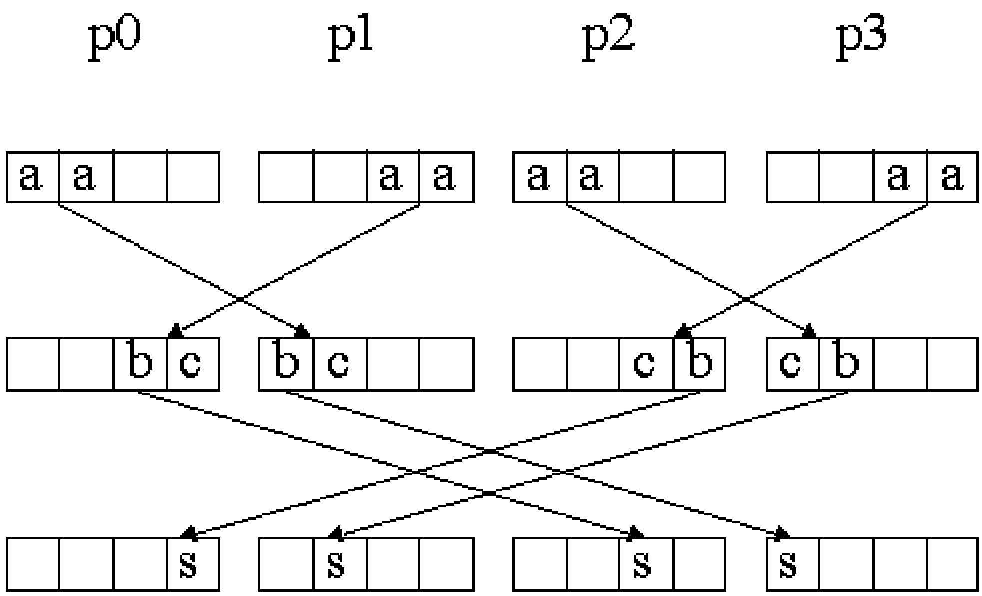


Figure 13.3. Adding vectors

Pseudocode for Broadcast Operation:

The following algorithm completes a broadcast operation in logarithmic time. Figure 13.4 illustrates the idea.

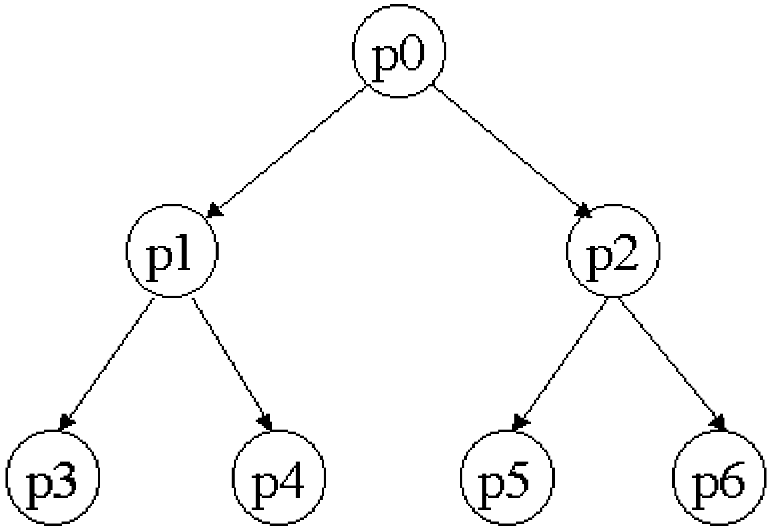


Figure 13.4. Broadcast via recursive doubling.

The first processor first sends the data to only two other processors. Then each of these processors send the data to two other processors, and so on. At each stage, the number of processors sending and receiving data doubles. The code is simple and looks similar to

```
if(myRank==0) {
  send to processors 1 and 2;
}
else
{
  receive from processors int((myRank-1)/2);
  torank1=2*myRank+1;
  torank2=2*myRank+2;
  if(torank1N)
   send to torank2;
}
```

13.2. Parallel Algorithm Examples

Specific Examples

In this section, specific examples of parallel algorithms that implement message passing are given. The first two examples consider simple collective communication calls to parallelize matrix-vector and matrix-matrix multiplication. These calls are meant to be illustrative, because parallel numerical libraries usually provide the most efficient algorithms for these operations. (See Chapter 10 - Parallel Mathematical

Libraries.) The third example shows how you can use ghost cells to construct a parallel data approach to solve Poisson's equation. The fourth example revisits matrix-vector multiplication, but from a client server approach.

- **Example 1:** Matrix-vector multiplication using collective communication.
- **Example 2:** Matrix-matrix multiplication using collective communication.
- **Example 3:** Solving Poisson's equation through the use of ghost cells.
- **Example 4:** Matrix-vector multiplication using a client-server approach.

13.2.1. Example 1: Matrix-vector Multiplication

Example 1: Matrix-vector Multiplication

The figure below demonstrates schematically how a matrix-vector multiplication, **A=B*C**, can be decomposed into four independent computations involving a scalar multiplying a column vector. This approach is different from that which is usually taught in a linear algebra course because this decomposition lends itself better to parallelization. These computations are independent and do not require communication, something that usually reduces performance of parallel code.

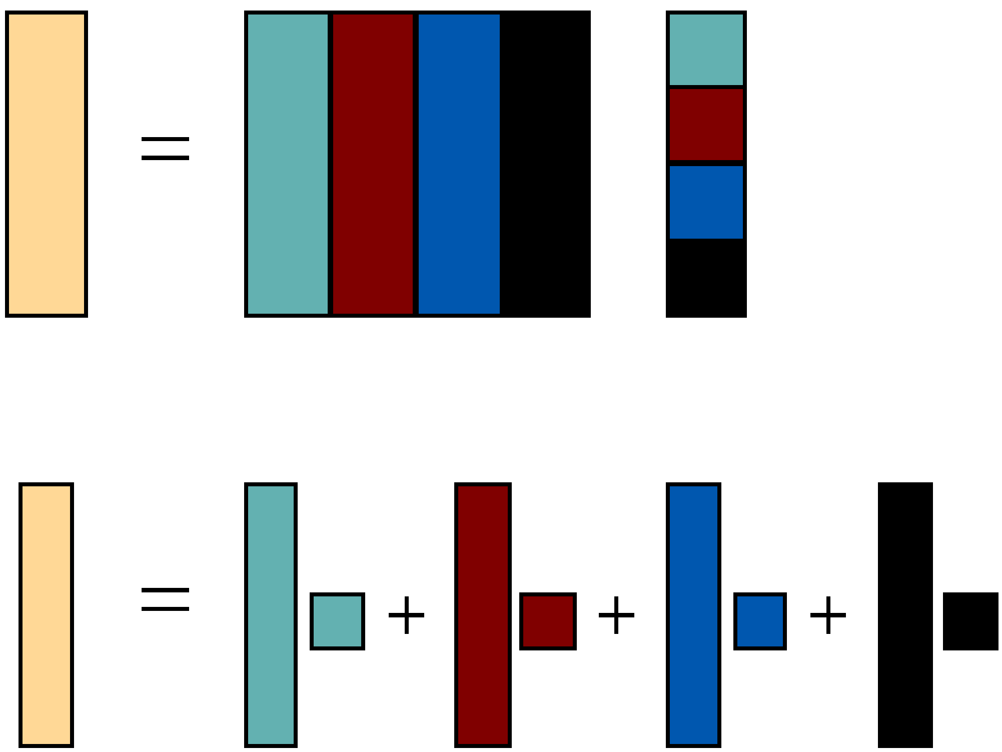


Figure 13.5. Schematic of parallel decomposition for vector-matrix multiplication, **A=B*C**, in Fortran 90. The vector **A** is depicted in yellow. The matrix **B** and vector **C** are depicted in multiple colors representing the portions, columns, and elements assigned to each processor, respectively.

The columns of matrix **B** and elements of column vector **C** must be distributed to the various processors using MPI commands called **scatter** operations. Note that MPI provides two types of scatter operations depending on whether the problem can be divided evenly among the number of processors or not. Each processor now has a column of **B**, called **Bpart**, and an element of **C**, called **Cpart**. Each processor can now perform an independent vector-scalar multiplication. Once this has been accomplished, every processor will have a part of the final column vector **A**, called **Apart**. The column vectors on each processor can be added together with an MPI reduction command that computes the final sum on the root processor. A Fortran 90 code and C code are available for you to examine.

Something for you to think about as you read the next section on matrix-matrix multiplication: How would you generalize this algorithm to the multiplication of a n X 4m matrix by a 4m by M matrix on 4 processors?

It is important to realize that this algorithm would change if the program were written in C. This is because C decomposes arrays in memory by rows while Fortran decomposes arrays into columns. If you translated the above program directly into a C program, the collective MPI calls would fail because the data going to each of the different processors is not contiguous. This problem can be solved with derived datatypes, which are discussed in Chapter 6 - Derived Datatypes. A simpler approach would be to decompose the vector-matrix multiplication into independent scalar-row computations and then proceed as above. This approach is shown schematically in Figure 13.6.

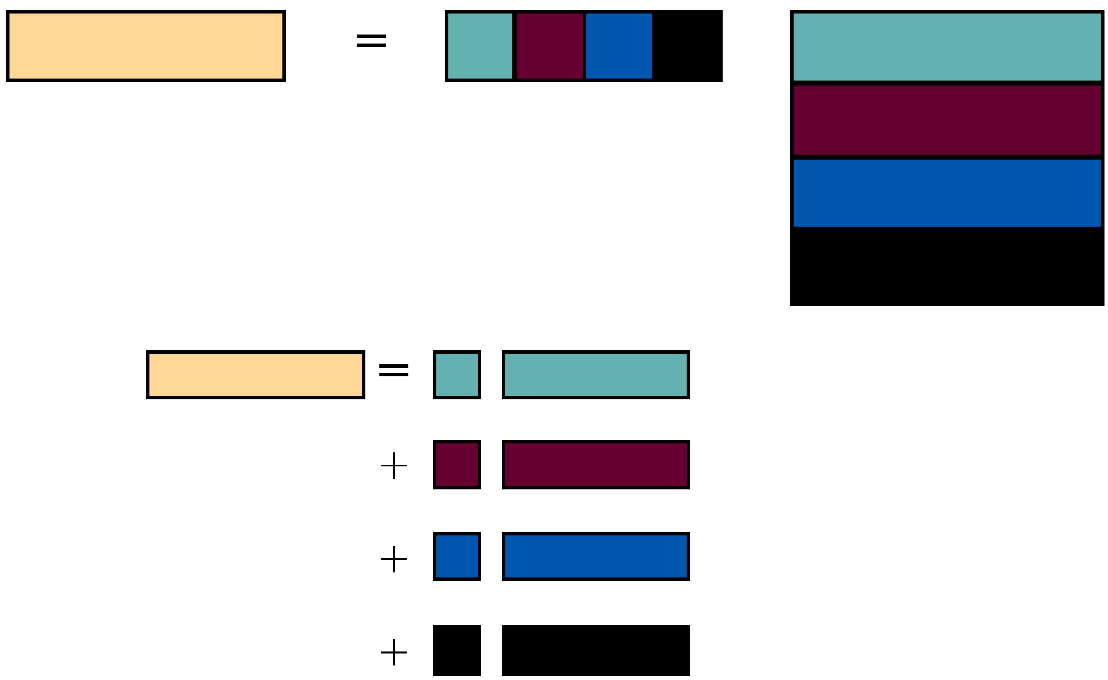


Figure 13.6. Schematic of parallel decomposition for vector-matrix multiplication, **A=B*C**, in the C programming language.

Returning to the Fortran example, another way this problem can be decomposed is to broadcast the column vector **C** to all the processors using the MPI broadcast command. (See Section 6.2 - Broadcast.) Then, scatter the rows of **B** to every processor so that they can form the elements of the result matrix **A** by the usual vector-vector "dot product". This will produce a scalar on each processor, **Apart**, which can then be gathered with an MPI gather command (see Section 6.4 - Gather) back onto the root processor in the column vector **A**.

However, you must be wary because division of a matrix into rows in Fortran 90 causes data in different processors to be noncontiguous. Again, this can be handled with derived datatypes but, in this example, it's simpler to just take the transpose of the vector **B** and scatter the columns. It is left as an exercise for you to write MPI code to implement this.

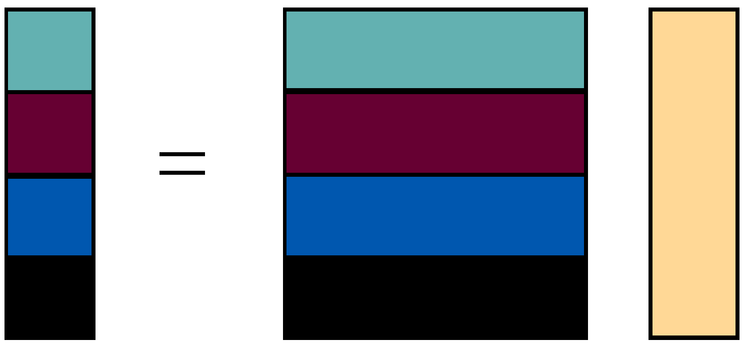


Figure 13.7. Schematic of a different parallel decomposition for vector-matrix multiplication in Fortran 90.

13.2.2. Example 2: Matrix-matrix Multiplication

Example 2: Matrix-matrix Multiplication

A similar, albeit naive, type of decomposition can be achieved for matrix-matrix multiplication, **A=B*C**. The figure below shows schematically how matrix-matrix multiplication of two 4x4 matrices can be decomposed into four independent vector-matrix multiplications, which can be performed on four different processors.

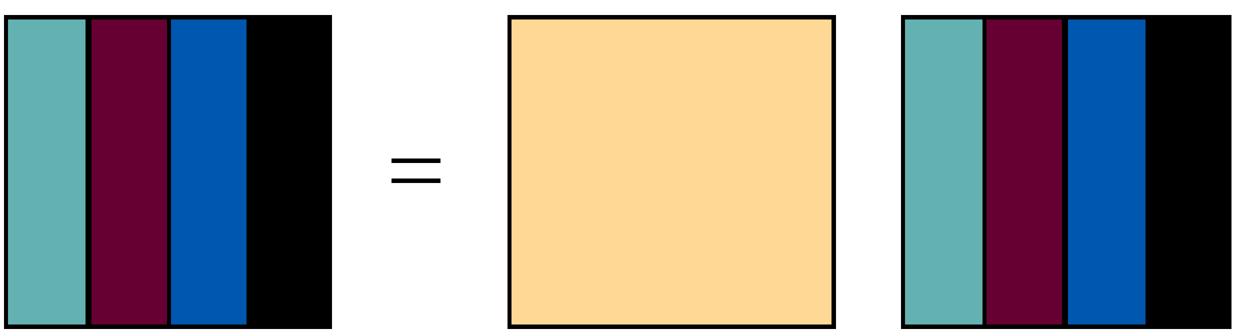


Figure 13.8. Schematic of a decomposition for matrix-matrix multiplication, **A=B*C**, in Fortran 90. The matrices **A** and **C** are depicted as multicolored columns with each color denoting a different processor. The matrix **B,** in yellow, is broadcast to all processors.

The basic steps are

1. Distribute the columns of **C** among the processors using a scatter operation.
2. Broadcast the matrix **B** to every processor.
3. Form the product of **B** with the columns of **C** on each processor. These are the corresponding columns of **A**.
4. Bring the columns of **A** back to one processor using a gather operation.

The complete Fortran 90 code and C code are provided.

Again, in C, the problem could be decomposed in rows. This is shown schematically below.

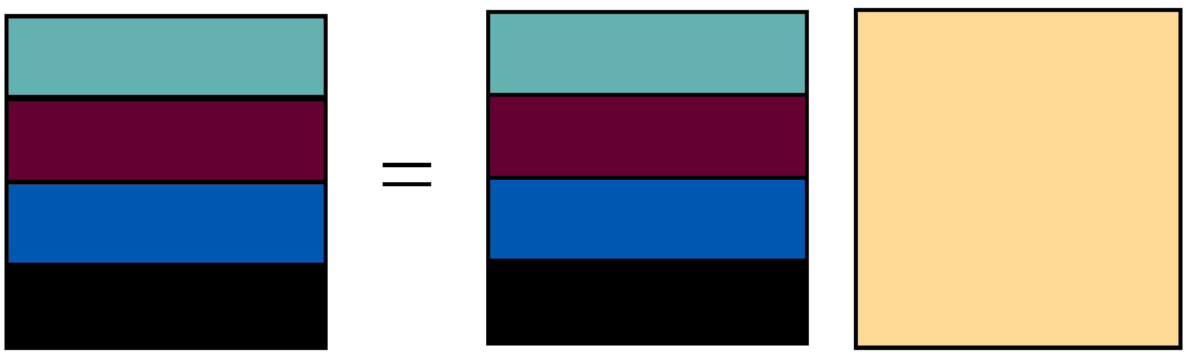


Figure 13.9. Schematic of a decomposition for matrix-matrix multiplication, **A=B*C**, in the C programming language. The matrices **A** and **B** are depicted as multicolored rows with each color denoting a different processor. The matrix **C**, in yellow, is broadcast to all processors.

13.2.3. Example 3: Poisson Equation

Example 3: The Use of Ghost Cells to solve a Poisson Equation

The objective in data parallelism is for all processors to work on a single task simultaneously. The computational domain (e.g., a 2D or 3D grid) is divided among the processors such that the computational work load is balanced. Before each processor can compute on its local data, it must perform communications with other processors so that all of the necessary information is brought on each processor in order for it to accomplish its local task.

As an instructive example of data parallelism, an arbitrary number of processors is used to solve the **2D Poisson Equation** in electrostatics (i.e., Laplace Equation with a source). The equation to solve is

$$\nabla^2 \phi(x,y) = -4\pi\rho(x,y)$$

$$\rho(x,y) = \frac{a}{\pi}\left(e^{-a[(x-L/4)^2 + y^2]} - e^{-a[(x-3L/4)^2 + y^2]} \right)$$

Figure 13.10. Poisson Equation on a 2D grid with periodic boundary conditions.

where ***phi(x,y)*** is our unknown potential function and ***rho(x,y)*** is the known source charge density. The domain of the problem is the box defined by the ***x***-axis, ***y***-axis, and the lines ***x=L*** and ***y=L***.

Serial Code:

To solve this equation, an iterative scheme is employed using finite differences. The update equation for the field ***phi*** at the (***n+1***)th iteration is written in terms of the values at nth iteration via

$$\phi_{i,j} = \pi \Delta x^2 \rho_{i,j} + \frac{1}{4}(\phi_{i+1,j} + \phi_{i-1,j} + \phi_{i,j+1} + \phi_{i,j-1})$$

iterating until the condition

$$\sum_{i,j} |\phi_{i,j}^{new} - \phi_{i,j}^{old}| / \sum_{i,j} |\rho_{i,j}| < \varepsilon$$

has been satisfied.

Parallel Code:

In this example, the domain is chopped into rectangles, in what is often called **block-block** decomposition. In Figure 13.11 below,

Columns	0	1	2	3	4
Row 0	0,0	0,1	0,2	0,3	0,4
Row 1	1,0	1,1	1,2	1,3	1,4
Row 2	2,0	2,1	2,2	2,3	2,4

Figure 13.11. Parallel Poisson solver via domain decomposition on a **3x5** processor grid.

an example **N=64 x M=64** computational grid is shown that will be divided amongst **NP=15** processors. The number of processors, **NP**, is purposely chosen such that it does not divide evenly into either **N** or **M**. Because the computational domain has been divided into rectangles, the 15 processors {**P(0),P(1),...,P(14)**} (which are laid out in row-major order on the processor grid) can be given a 2-digit designation that represents their processor grid row number and processor grid column number. MPI has commands that allow you to do this.

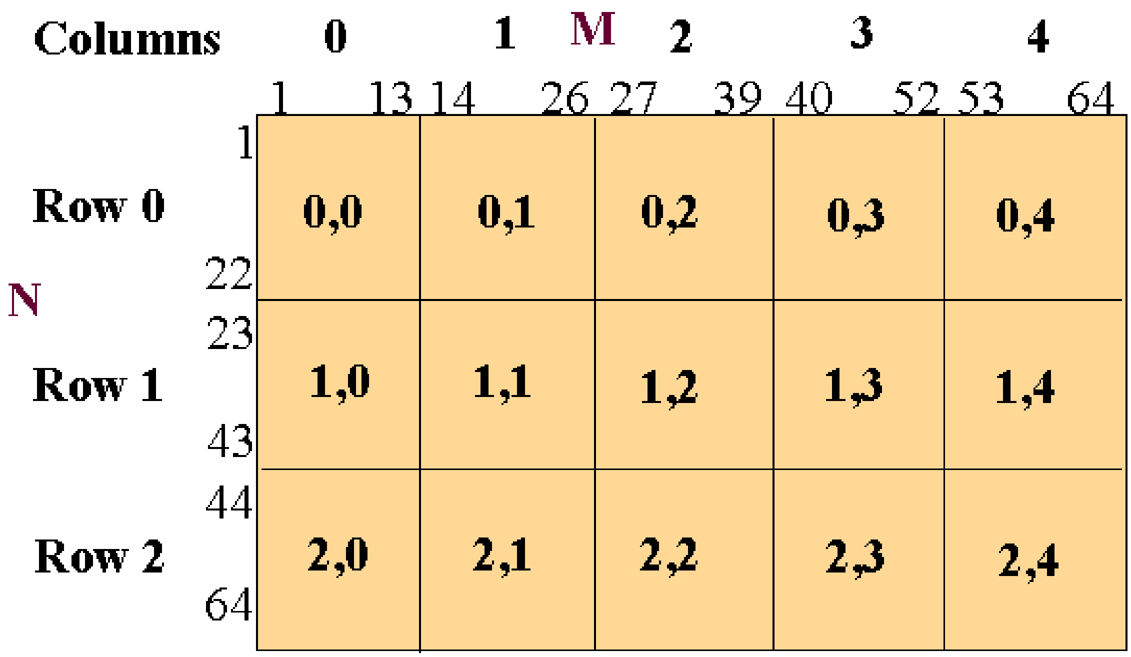


Figure 13.12. Array indexing in a parallel Poisson solver on a **3x5** processor grid.

Note that **P(1,2)** (i.e., **P(7)**) is responsible for indices ***i=23-43*** and ***j=27-39*** in the serial code double do-loop. A parallel speedup is obtained because each processor is working on essentially 1/15 of the total data. However, there is a problem. What does **P(1,2)** do when its 5-point stencil hits the boundaries of its domain (i.e., when ***i=23*** or ***i=43***, or ***j=27*** or ***j=39***)? The 5-point stencil now reaches into another processor's domain, which means that boundary data exists in memory on another separate processor. Because the update formula for ***phi*** at grid point ***(i,j)*** involves neighboring grid indices {***i-1,i,i+1;j-1,j,j+1***}, **P(1,2)** must communicate with its North, South, East, and West (N, S, E, W) neighbors to get one column of boundary data from its E, W neighbors and one row of boundary data from its N,S neighbors. This is illustrated in Figure 13.13 below.

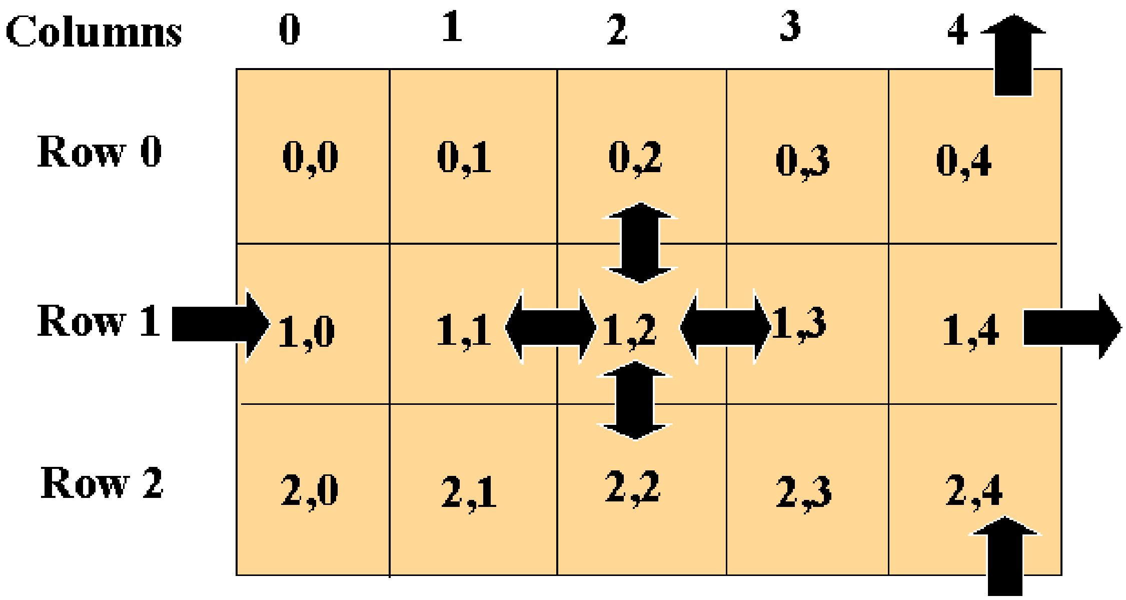


Figure 13.13. Boundary data movement in the parallel Poisson solver following each iteration of the stencil.

In order to accommodate this transference of boundary data between processors, each processor must dimension its **local** array ***phi*** to have two extra rows and 2 extra columns. This is illustrated in Figure 13.14 where the shaded areas indicate the extra rows and columns needed for the boundary data from other processors.

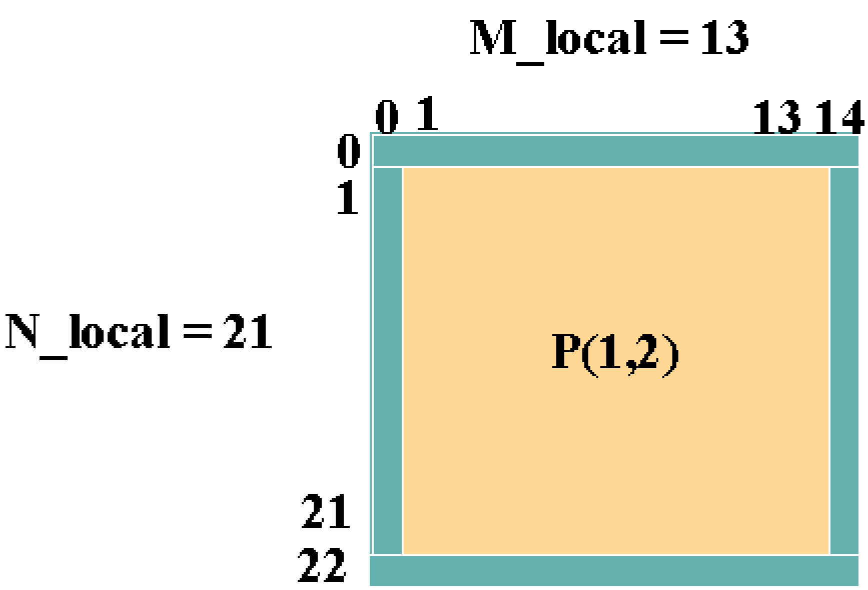


Figure 13.14. Ghost cells: Local indices.

Note that even though this example speaks of global indices, the whole point about parallelism is that no one processor ever has the global ***phi*** matrix on processor. Each processor has only its local version of ***phi*** with its own sub-collection of ***i*** and ***j*** indices. Locally these indices are labeled beginning at either 0 or 1, as in Figure 13.14, rather than beginning at their corresponding global values, as in Figure 13.12. Keeping track of the on-processor local indices and the global (in-your-head) indices is the bookkeeping that you have to manage when using message passing parallelism. Other parallel paradigms, such as High Performance Fortran (HPF) or OpenMP, are directive-based, i.e., compiler directives are inserted into the code to tell the supercomputer to distribute data across processors or to perform other operations. The difference between the two paradigms is akin to the difference between an automatic and stick-shift transmission car. In the directive based paradigm (automatic), the compiler (car) does the data layout and parallel communications (gear shifting) implicitly. In the message passing paradigm (stick-shift), the user (driver) performs the data layout and parallel communications explicitly. In this example, this communication can be performed in a regular prescribed pattern for all processors. For example, all processors could first communicate with their N-most partners, then S, then E, then W. What is happening when all processors communicate with their E neighbors is illustrated in Figure 13.15.

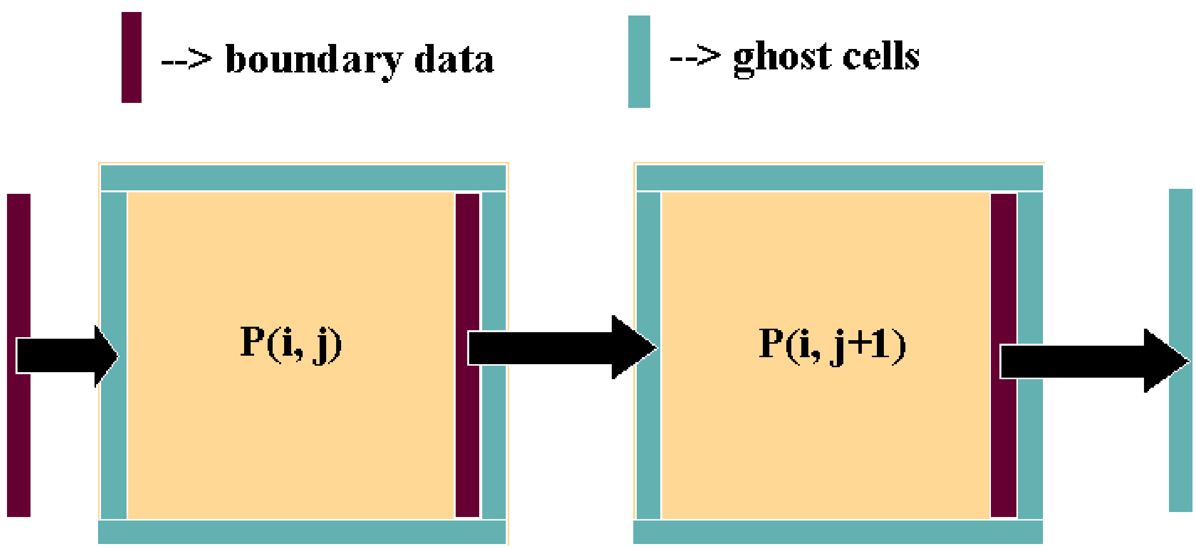


Figure 13.15. Data movement, shift right (East).

Note that in this **shift right** communication, **P(i,j)** places its right-most column of boundary data into the left-most ghost column of **P(i,j+1)**. In addition, **P(i,j)** receives the right-most column of boundary data from **P(i,j-1)** into its own left-most ghost column.

For each iteration, the psuedo-code for the parallel algorithm is thus

```
    t = 0
(0) Initialize psi

(1) Loop over stencil iterations

    (2) Perform parallel N shift communications of boundary data
    (3) Perform parallel S shift communications of boundary data
    (4) Perform parallel E shift communications of boundary data
    (5) Perform parallel W shift communications of boundary data

          in Fortran                                          in C

    (6) do j = 1, M_local
for{i=1;i<=N_local;i++){
           do  i = 1, N_local
for(j=1;j<=M_local;j++){
               update phi(i,j)                               update
phi[i][j]
           enddo                                         }
        enddo                                        }

    End Loop over stencil iterations

(7) Output data
```

Note that initializing the data should be performed in parallel. That is, each processor **P(i,j)** should only initialize the portion of ***phi*** for which it is responsible. (Recall NO processor contains the full global ***phi***). In relation to this point, step (7), Output data, is not such a simple-minded task when performing parallel calculations. Should you reduce all the data from **phi_local** on each processor to one giant **phi_global** on **P(0,0)** and then print out the data? This is certainly one way to do it, but it seems to defeat the purpose of not having all the data reside on one processor. For example, what if **phi_global** is too large to fit in memory on a single processor? A second alternative is for each processor to write out its own **phi_local** to a file "phi.ij", where ij indicates the processor's 2-digit designation (e.g. **P(1,2)** writes out to file "phi.12"). The data then has to be manipulated off processor by another code to put it into a form that may be rendered by a visualization package. This code itself may have to be a parallel code.

As you can see, the issue of parallel I/O is not a trivial one (see Section 9 - Parallel I/O) and is in fact a topic of current research among parallel language developers and researchers.

13.2.4. Example 4: Matrix-vector Multiplication (Client Server)

Matrix-vector Multiplication using a Client-Server Approach

In Section 13.2.1, a simple data decomposition for multiplying a matrix and a vector was described. This decomposition is also used here to demonstrate a "client-server" approach. The code for this example is in the C program, server_client_c.c.

In server_client_c.c, all input/output is handled by the "server" (preset to be processor 0). This includes parsing the command-line arguments, reading the file containing the matrix **A** and vector ***x***, and writing the result to standard output. The file containing the matrix **A** and the vector ***x*** has the form

```
m n
x1  x2  ...
a11 a12 ...
a21 a22 ...
     .
     .
     .
```

where **A** is **m** (rows) by **n** (columns), and ***x*** is a column vector with **n** elements. After the server reads in the size of **A**, it broadcasts this information to all of the clients. It then checks to make sure that there are fewer processors than columns. (If there are more processors than columns, then using a parallel program is not efficient and the program exits.) The server and all of the clients then allocate memory locations for **A** and ***x*.** The server also allocates memory for the result. Because there are more columns than client processors, the first "round" consists of the server sending one column to each of the client processors. All of the clients receive a column to process. Upon finishing, the clients send results back to the server. As the server receives a "result" buffer from a client, it sends the next unprocessed column to that client.

The source code is divided into two sections: the "server" code and the "client" code. The pseudo-code for each of these sections is

Server:

1. Broadcast (vector) ***x*** to all client processors.
2. Send a column of **A** to each processor.
3. While there are more columns to process OR there are expected results, receive results and send next unprocessed column.
4. Print result.

Client:

1. Receive (vector) ***x*.**
2. Receive a column of **A** with ***tag = column number.***
3. Multiply respective element of (vector) ***x*** (which is the same as ***tag***) to produce the (vector) result.
4. Send result back to server.

Note that the numbers used in the pseudo-code (for both the server and client) have been added to the source code.

Source code similar to server_client_c.c., server_client_r.c is also provided as an example. The main difference between theses codes is the way the data is stored. Because only contiguous memory locations can be sent using MPI_SEND, server_client_c.c stores the matrix **A** "column-wise" in memory, while server_client_r.c stores the matrix **A** "row-wise" in memory. The pseudo-code for server_client_c.c and server_client_r.c is stated in the "block" documentation at the beginning of the source code.

---

14. Complete Reference Listing

Complete Reference Listing

You may view a complete listing of the reference materials for this course by clicking on the References link in the ACTION MENU above.

---

15. Course Evaluation Form

Introduction to MPI Evaluation

We really want to know what you thought of this course. Please complete the evaluation form by clicking on the evaluation icon in the button bar above. Even if you didn't finish the course, your input is welcome.
