Course Title: Introduction to MPI

2 Getting Started with MPI

2.1 Getting Started with MPI

Introduction

MPI was designed to be a standard implementation of the message-passing model of parallel computing and consists of a set of C functions or Fortran subroutines that you insert into source code to perform data communication between processes. Although MPI programs include one or more calls to a library of message-passing functions or subroutines, MPI itself is not a library. Rather, MPI is an interface specification of what such a message-passing library should be in order to provide a standard for the writing of message passing parallel programs.

A MPI program consists of two or more autonomous processes, each executing their own codes, which may or may not be identical on a given pair of processes. These processes communicate via calls to MPI communication routines and are identified according to their relative *rank* within a group (0, 1, . . . , groupsize-1). MPI does not allow for dynamic allocation of processes during the execution of a parallel program. You specify the number of processes at the start of your program and that number remains fixed throughout the entire program.

Objectives

This lesson will familiarize you with the following basic concepts of MPI programming:

- The MPI Standard
- The goals and scope of MPI
- Types of MPI Routines
- Point-to-point communications and messages
- Collective communications
- Compiling and running MPI programs
- How-to write a simple MPI program

At the end of this lesson, you will be able to write serial versions of code in preparation for writing parallel versions later in this tutorial. Two exercises and a self test are provided to let you practice what you have learned.

2.2 The MPI Standard

MPI was developed over two years of discussions led by the MPI Forum, a group of approximately sixty people representing about forty organizations. The MPI-1 standard was defined in the spring of 1994, and it consists of the following:

- It 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, including tools for parallel I/O, C++ and Fortran 90 bindings, and one-sided communication. At present, some MPI implementations do include portions of the MPI-2 standard.

Note: This lesson only covers MPI-1. See the other lessons in this course ("Parallel I/O", "One-Sided Communication", and "Using MPI in C++ Code") to learn about new MPI features now commonly supported.

2.3 MPI Goals

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 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 and support for

heterogeneous parallel architectures.

The following are explicitly outside the scope of MPI-1:

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

2.4 When and When Not to Use MPI

MPI is not necessarily the best option for every parallel computing problem. There are some reasons when you should use MPI and some reasons when you should not.

Use MPI when you need:

- parallel code that is portable across platforms
- higher performance, e.g. when small-scale "loop-level" parallelism does not provide enough *speedup*

Do not use MPI when you:

- can achieve sufficient performance and portability by using the "loop level" parallelism available in such software as High-Performance Fortran or OpenMP, or proprietary machine-dependent directives (*Note:* Typically the performance provided by "loop-level" parallelism is minimum and less than desired.)
- can use a pre-existing library of parallel routines, which may themselves be written using MPI (See the lesson in this course entitled "Parallel Mathematical Libraries.")
- don't need parallelism at all

2.5 Types of MPI Routines

2.5.1 Types of MPI Routines

The MPI standard includes routines for the following operations:

- *Point-to-point communication*
- *Collective communications*
- Process groups
- Process topologies
- Environment management and inquiry

Each of these operations is described further in this lesson but the main focus will be point-to-point and collective communications.

2.5.2 Point-to-Point Communications and Messages

2.5.2.1 Point-to-Point Communications and Messages

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

Point-to-point communication is two-sided, which means 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* that indicates the source and destination processors, and a *body* that contains the actual data to be sent.

MPI uses the following 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). 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.
2. **Datatype** — the type of data to be sent. In the simplest cases this is an elementary type such as float (REAL), int (INTEGER). In more advanced applications this can be a user-defined datatype 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 datatypes allows complete flexibility in defining the message content.

3. **Count** — the number of items of type datatype to be sent.

MPI standardizes the designation of the elementary types, which means that you do not have to explicitly worry about differences in how machines in heterogeneous environments represent them, e.g., differences in representation of floating-point numbers and integers.

2.5.2.2 Communication Modes and Completion Criteria

MPI provides much flexibility in specifying how messages are to be sent. A variety of communication modes 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 considered 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. In this latter case, nothing at all 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

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.

2.5.2.3 Blocking and Nonblocking Communication

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

A blocking send or receive does not return from the subroutine call until the operation has actually completed. Thus it ensures that the relevant completion criteria have been satisfied before the calling process is allowed to proceed. For example, with a blocking send 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 approach has the advantage that the processor is free to do something else while the communication proceeds in the background. You can test later to see whether the communication 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. However, until it is complete you cannot assume that the message has been received or that the variables to be sent may be safely overwritten.

2.5.3 Collective Communications

2.5.3.1 Collective Communications

A *communicator* is an MPI object that defines a group of processes that are permitted to communicate with one another. Every MPI message must specify a communicator via a “name” that is included as an explicit parameter within the argument list of the MPI call. By default, all processes are defined as being members of the communicator MPI_COMM_WORLD.

Collective communication routines, also called collective operations, transmit data among all processes in a group. These routines allow larger groups of processors to communicate in various ways, for example, one-to-several or several-to-one. All collective communication events are blocking.

There are three basic types of collective communication events in MPI. They are:

1. Synchronization — each process waits until all processes included within its group have reached the specified synchronization point.
2. Data movement — data is transferred to all processes included within its group.
3. Collective computation — one process within a group collects data from other processes within that group and performs an operation (addition, multiplication, etc.) on that data.

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

- The possibility of error is significantly reduced. A single 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.

We briefly describe these collective operations:

- Broadcast operations
- Scatter operation and gather operation
- Reduction operations

2.5.3.2 Broadcast Operations

The simplest kind of collective operation is 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 in Figure 2.1 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.

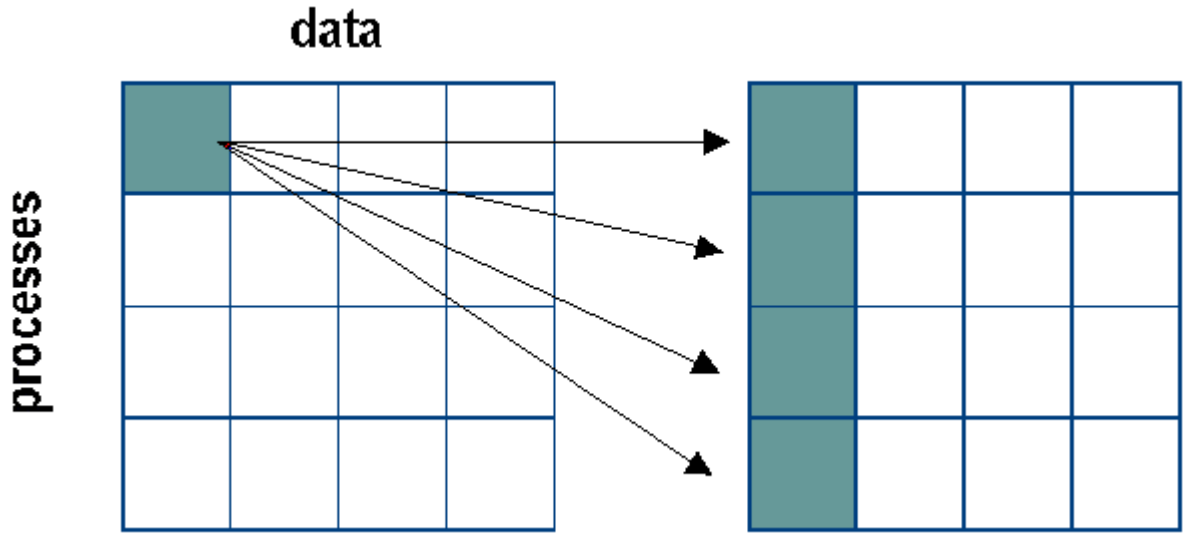

Figure 2.1: Broadcast operations

2.5.3.3 Scatter and Gather Operations

Perhaps the most important classes of collective operations are those that distribute data on one processor across a group of processors or vice versa. These are called *scatter operations* and *gather operation*s. MPI provides two kinds of scatter and gather operations, depending upon whether the data can be evenly distributed across processors. These operations are illustrated below in Figure 2.2.

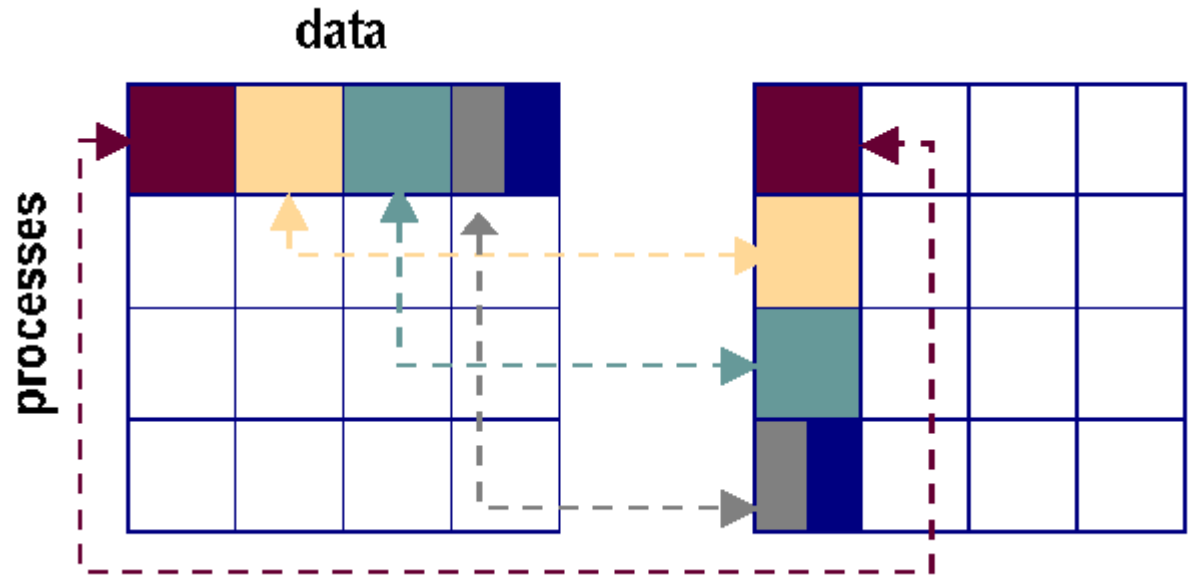

Figure 2.2: Scatter and gather operations

In a scatter operation, all of the data (an array of some type) are initially collected on a single processor (the left side of Figure 2.2). After the scatter operation, pieces of the data are distributed on different processors (the right side of Figure 2.2). 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.

2.5.3.4 Reduction Operations

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

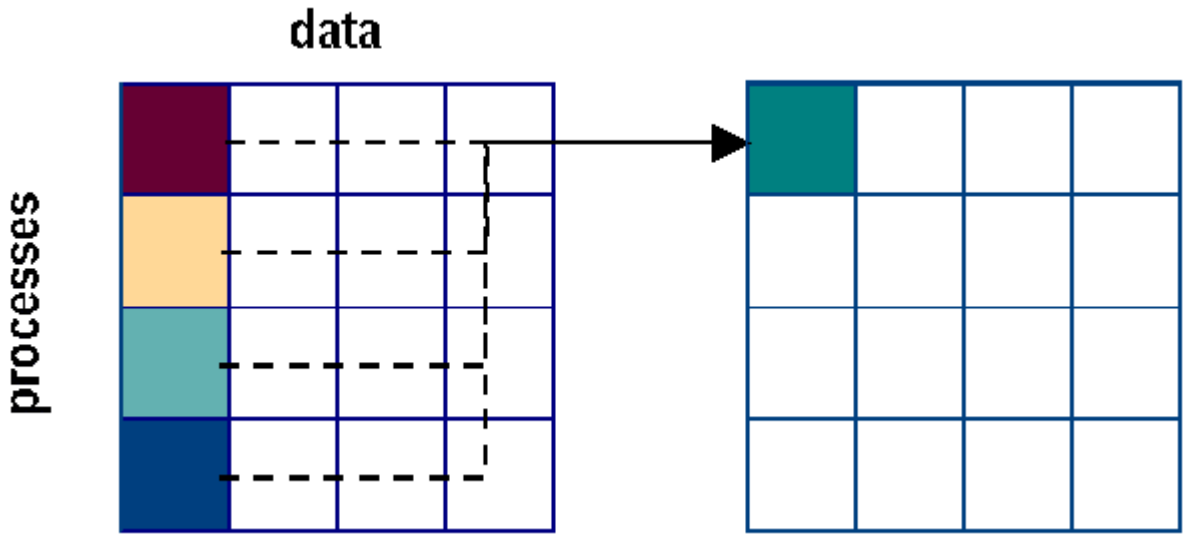

Figure 2.3: Reduction operations

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

2.5.4 Process Groups

A process group is simply an ordered set of processes where each *process* in a group is associated with a unique integer value referred to as the *rank* of the process. A process rank is also referred to as the process "ID." Rank values in MPI always start at zero and run sequentially through *N-1*, where *N* is the number of processes in the group.

Although the number of processes specified in an MPI program remains fixed throughout the program, both groups and communicators can be dynamically created and eliminated during the execution of the program. In addition, a given process can be a member of more than one group or communicator and will have a unique rank within each group or communicator.

2.5.5 Process Topologies

In MPI, a *topology* is a mechanism for associating different identification schemes with the processes belonging to a particular group. That is, a topology describes a mapping or ordering of MPI processes into some geometric shape. MPI supports two main types of topologies – a Cartesian, or grid, topology and a graph topology. All MPI topologies are virtual in that there may be no simple relation between the process structure implicit in the MPI topology and the actual underlying physical arrangement of the processors within the computer itself.

Virtual topologies are used in MPI to provide communication *efficiency* and for programming convenience. For example, a Cartesian or grid topology is likely to be convenient in an application involving nearest-neighbor communication between points on a rectangular grid. Which virtual topology to use in a MPI program is determined by the application developer.

2.5.6 Environment Management and Inquiry

A number of MPI routines are available for managing and inquiring about the state of the environment. They are used for a number of purposes, such as initializing and terminating the MPI execution environment, terminating all processes belonging to a given MPI communicator, determining the number of processes belonging to a given communicator, and determining the rank of the calling process within a given communicator. A number of these routines will be discussed throughout this tutorial.

2.6 Compiling and Running MPI Programs

The MPI standard *does not specify* how MPI programs are to be started; therefore, it is important to note that implementations vary from machine to machine. When compiling a MPI program, it may be necessary to link against the MPI library. Typically, to do this, you include the option "-lmpi " to the loader. Some MPI implementations will include special commands to use when compiling and linking: mpicc, mpif90, ...

To run an MPI code, you commonly use a wrapper specific to the implementation and/or batch environment. In this example the wrapper is called "mpirun". Other implementations might use a command like: aprun, ibrun, mpiexec, poe, or an alternate wrapper. The following command would run the executable "execfile" on four processors:

```
$  mpirun -np 4 execfile
```

2.7 A First Program: ProcessColors

The code listed below illustrates the use of both point-to-point and collective communications. In this code, an array of three colors (white, red, and green) is distributed to all processes created at the start of the MPI program. Each process changes its color and sends its modified color attribute back to process 0, which prints both the original color and the modified color of each process.

After having defined all of the required variables that were to be used in the program, we specified our root process. The root process was used to broadcast colors to all processes and to print the colors of each of our processes. Process 0 was selected as the root process.

Next, we generated the three separate colors used to “color” our processes by dynamically allocating memory for an array, *colorArray*, that contained *ncolor = 3* elements wherein we stored the three colors: white, red, and green.

```
ncolors = 3;
colorArray = (int*) malloc(sizeof(int) * ncolors);

for (k = 0; k < ncolors; k++)
{
     colorArray[k] = k;
}
```

Next we started the primary MPI coding and then accessed (1) the rank of each process and (2) the total number of processes created at runtime.

```
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &procID);
MPI_Comm_size(MPI_COMM_WORLD, &nproc);
```

We used the communicator *MPI_COMM_WORLD*, which consists of all processes created at the start of the MPI run.

We then broadcast the entire color array to each process using the command *MPI_BCAST* and had each process color itself with the color green (the third element in *colorArray*).

```
MPI_Bcast(colorArray, ncolors, MPI_INT, root, MPI_COMM_WORLD);
pcolor = colorArray[2];
```

The rank of the processor calling MPI_BCAST was not specified because all processors in the communicator MPI_COMM_WORLD must make this collective communication call. In this case it was the root process that sent the same data, *colorArray*, to every process in the communicator. Additionally, there was no *tag* parameter included in the MPI_BCAST function. This parameter is not required because in this collective communication operation the number of elements and the datatype of these elements are identical on all processes.

Next, we had each non-root process send its color back to the root process, which then printed the color of each of the *nproc* processes. We used the point-to-point communication operations *MPI_SEND* and *MPI_RECV* to communicate between the processes.

We then changed the color of each process based on whether the process rank was even or odd (even rank white, odd rank red) and had each non-root process send its modified color back to the root process, which again printed out the color of each of the *nproc* processes.

Sample Code

Here we give the code and output for our sample C program, ProcessColors.c.

C:

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

main (int argc, char* argv[])
{
     int procID, nproc, root, source, target, tag;
     int k, ncolors, pcolor;
     int *colorArray;
     char color[10];
     MPI_Status status;

     // Set the rank 0 process as the root process
     root = 0;

     // Generate three colors for color array, where white = 0, red = 1, and green = 2
     ncolors = 3;
     colorArray = (int*) malloc(sizeof(int) * ncolors);

     for (k = 0; k < ncolors; k++)
     {
          colorArray[k] = k;
     }

     // Initialize MPI
     MPI_Init(&argc, &argv);

     // Get process rank
     MPI_Comm_rank(MPI_COMM_WORLD, &procID);

     // Get total number of processes specificed at start of run
     MPI_Comm_size(MPI_COMM_WORLD, &nproc);

     // Broadcast the array of colors to all processes
     MPI_Bcast(colorArray, ncolors, MPI_INT, root, MPI_COMM_WORLD);

     // Color each process 'green' (color = 2)
          pcolor = colorArray[2];

     // Have each process send its color to the root process
```

```
    tag = pcolor;
    target = 0;

    if (procID != root)
    {
         MPI_Send(&pcolor, 1, MPI_INT, target, tag, MPI_COMM_WORLD);
    }
    else
    {
         for (source = 0; source < nproc; source++)
        {
              if (source != 0)
              {
                   MPI_Recv(&pcolor, 1, MPI_INT, source, tag, MPI_COMM_WORLD, &status);
              }

              switch(pcolor)
              {
                case 0: sprintf(color, "white");
                        break;
                case 1: sprintf(color, "red");
                        break;
                case 2: sprintf(color, "green");
                        break;
                default: printf("Invalid color\n");
              }

              printf("proc %d has color %s\n", source, color);
         }
         printf("\n\n");
    }

    pcolor = procID%2;

    if (pcolor == 0)
    {
         sprintf(color, "white");
    }
    else if (pcolor == 1)
    {
         sprintf(color, "red");
    }
    else if (pcolor == 2)
    {
         sprintf(color, "green");
    }

    // Access new process colors
    if (procID != root)
    {
         MPI_Send(color, 10, MPI_CHAR, root, tag, MPI_COMM_WORLD);
    }
    else
    {
         printf("proc %d has color %s\n", root, color);

         for (source = 1; source < nproc; source++)
         {
              MPI_Recv(color, 10, MPI_CHAR, source, tag, MPI_COMM_WORLD, &status);
              printf("proc %d has color %s\n", source, color);
         }
    }

    free(colorArray);

    MPI_Finalize();
}
```

After compiling the code using the command:

```
mpicc -o ProcessColors ProcessColors.c
```

and running it on 8 processes using the command

```
mpirun –np 8 ProcessColors
```

we obtained the the following output:

```
proc 0 has color green
proc 1 has color green
proc 2 has color green
proc 3 has color green
proc 4 has color green
proc 5 has color green
proc 6 has color green
proc 7 has color green

proc 0 has color white
proc 1 has color red
proc 2 has color white
proc 3 has color red
proc 4 has color white
proc 5 has color red
proc 6 has color white
proc 7 has color red
```

2.8 Exercise 2 - Game of Life

In this lesson, you learned the basic terminology and concepts of using MPI, but not the syntax of the library routines. For this reason, in this exercise you will write a serial (i.e., single-processor) "Game of Life" program. You will extend it to a parallel version in a later lesson.

Exercise

Refer to the outline you wrote for Exercise 1 - "Game of Life" and write, compile, and run a serial "Game of Life" code. In later course exercises the arrays will have to be allocated dynamically; so, using dynamic allocation from the start will make it easier later on. For debugging purposes you may want to specify small arrays (say, 4 x 4) and a single time step. Once you are confident the code is working properly, you can increase the array size and the number of time steps to something larger, for example, 200 x 200 and 500.

Show C Solution

Show Fortran Solution

2.9 Exercise 2 - Parallel Search

As described in Exercise 1 - Parallel Search, this 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 lesson provided an introduction to the terminology and concepts of MPI but not the syntax of the library routines. To get a better understanding of these concepts, you will write a serial version (that is, a version that runs on one processor) before writing a parallel version of a program. You can use either Fortran or C/C++ and should confirm that the program works by using a test input array. If you like, you may use the input array used in our solution - b.data

Show Solution

Serial Search Program (Fortran 90)

```
PROGRAM search
   parameter (N=300)
   integer i, target ! local variables
   integer b(N)      ! the entire array of integers
```

```
   ! File b.data has the target value on the first line
   ! The remaining 300 lines of b.data have the values for the b array
   open(unit=10,file="b.data")

   ! File found.data will contain the indices of b where the target is
   open(unit=11,file="found.data")

   ! Read in the target
   read(10,*) target

   ! Read in b array
   do i=1,300
      read(10,*) b(i)
   end do

   ! Search the b array and output the target locations
   do i=1,300
      if (b(i) == target) then
         write(11,*) i
      end if
   end do
END PROGRAM search
```

The results obtained from running this code are in the file "found.data" which contains the following:

```
62
183
271
291
296
```

2.10 Self Test - Getting Started

Now that you have finished this lesson, test yourself on what you have learned by taking the following Self Test.

 Question 1

Is a blocking send necessarily also synchronous?

- ○ Yes
- ○ No

 Question 2

 Question 3

Which of the following is true for all send routines?

- ○ It is always safe to overwrite the sent variable(s) on the sending processor after the send returns.
- ○ Completion implies that the message has been received at its destination.
- ○ It is always safe to overwrite the sent variable(s) on the sending processor after the send is complete.
- ○ All of the above.
- ○ None of the above.

 Question 4

Consider the following fragment of MPI pseudo-code:

```
...

x = fun(y)

MPI_SOME_SEND(the value of x to some other processor)

x = fun(z)

...
```

where MPI_SOME_SEND is a generic send routine. In this case, it would be best to use

- ○ A blocking send
- ○ A nonblocking send
