Running C/C++ Program in parallel using MPI

Eunyoung Seol

April 29, 2003

abstract

MPI (Message Passing Interface) is one of the most popular library for message passing within a parallel program. MPI can be used with Fortran and C/C++, and this paper discusses how to create parallel C/C++ program using MPI.

1 Introduction

Concurrency can be provided to the programmer in the form of explicitly concurrent language, compiler-supported extention to traditional sequential languages, or library package outside the language proper. The latter two alternatives are by far most common: the vast majority of parallel programs currently in use are either annotated Fortran for vector machines or C/C++ code with library calls

The two most popular packages for message passing within a paralle program are PVM and MPI. PVM is richer in the area of creating and managing processes on a heterogeneous distributed network, in which machines of different types may join and leave the computation during execution. MPI provides more control over how communication is implemented, and a richer set of communication primitives, especially for so-called collective communication: one-to-all, all-to-one, or all-to-all patterns of messages among a set of threads. Implementations of PVM and MPI are available for C, C++, and Fortran.

MPI is not a new programming language. It is a simply a library of definition and functions that can be used in C/C++(Fortran) programs. So in order to unserstand MPI, we just need to learn about a collection of special definitions and functions. Thus, this paper is about how to use MPI definitions and functions to run C/C++ program in parallel. Even there are lots of topics with MPI, this paper will simply focus on programming aspect of MPI within C/C++ program. Thus, to the rest of this paper, MPI/C program means "C/C++ program that calls MPI library". And MPI program means "C/C++ or Fortran program that calls MPI library routine".

Chapter 2 provides brief history of MPI, how to obtain, compile and run MPI/C program. Chapter 3 and chapter 4 provide a tutorial to basic MPI.

Chapter 3 shows a very simple MPI/C program and discusses the basic structure of MPI/C program. Chapter 4 describes the details of some MPI routines. Chapter 5 provides an example of MPI/C program based on a tutorial of chapters 3-4. Chapter 6 gives the resource list of MPI for the readers who are interested in more of MPI.

2 Get Ready for MPI

2.1 History of MPI

MPI is a Message Passing Interface Standard defined by a group involving about 60 people from 40 organizations in the United States and Europe, including vendors as well as researchers. It was the first attempt to create a ”standard by consensus” for message passing libraries. MPI is available on a wide variety of platforms, ranging from massively parallel systems to networks of workstations. The main design goals for MPI were to establish a practical, portable, efficient, and flexible standard for message-passing. The document defining MPI is ”MPI: A Message Passing Standard” written by the Message Passing Interface Forum, and should be available from Netlib via `http://www.netlib.org/mpi`.

2.2 Obtaining MPI

MPI is merely a standardized interface, not a specific implementation. There are several implementations of MPI in existence, a few freely available one are `MPICH` from Argonne National Laboratory and Mississippi State University, `LAM` from the Ohio Supercomputer Center, `CHIMP/MPI` from the Edinburgh Parallel Computing Center (EPCC), and `UNIFY` from the Mississippi State University. The Ohio Supercomputer Center tries to maintain a current list of implementations at `http://www.osc.edu/mpi`.

2.3 Compiling and Running MPI Applications

The details of compiling and executing MPI/C protgram depend on the system. Compiling may be as simple as

```
g++ -o executable filename.cc -lmpi
```

However, there may also be a special script or makefile for compiling. Therefore, the most generic way to compile MPI/C program is using `mpicc` script provided by some MPI implementations. These commands appear similarly to the basic `cc` command, however they transparently set the include paths and link to the appropriate libraries. You may naturally write your own compilation commands to accomplish this.

```
mpicc -o executable filename.cc
```

To execute MPI/C program, the most generic way is to use a commonly provided script `mpirun`. Roughly speaking, this script determines machine architecture, which other machines are included in virtual machine and spawns the desired processes on the other machines. The following command spawns 3 copies of executable.

```
mpirun -np 3 executable
```

The actual processors chosen by textttmpirun to take part in parallel program is usually determined by a global configuration file. Choice of processors can be specified by setting a parameter `machinefile`.

```
mpirun -machinefile machine-file-name -np nb-procs executable
```

Please note that the above syntax refers to the `MPICH` implementation of MPI, other implementations may be missing these commands or may have different versions of these commands.

3 Basic of MPI

The complete MPI specification consists of about 129 calls. However, a beginning MPI programmer can get by with very few of them (6 to 24). All that is really required is a way for the processes to exchange data, that is, to be able to send and recieve messages.

The following are basic functions that are used to build most MPI programs.

- All MPI/C programs must include a header file `mpi.h`.
- All MPI programs must call `MPI_INT` as the first MPI call, to initialize themselves.
- Most MPI programs call `MPI_COMM_SIZE` to get the number of processes that are running
- Most MPI programs call `MPI_COMM_RANK` to determine their rank, which is a number between 0 and `size-1`.
- Conditional process and general message passing can take place. For example, using the calls `MPI_SEND` and `MPI_RECV`.
- All MPI programs must call `MPI_FINALIZE` as the last call to an MPI library routine.

So we can write a number of useful MPI programs using just the following 6 calls `MPI_INIT, MPI_COMM_SIZE, MPI_COMM_RANK, MPI_SEND, MPI_RECV, MPI_FINALIZE`. The fo llowing is one of the simple MPI/C programs that makes all involving processors print "Hello, world".

```
#include <iostream>
#include "mpi.h"
int main(int argc, char **argv)
{
  int rank, size, tag, rc, i;
  MPI_Status status;
  char message[20];
  rc = MPI_Init(&argc, &argv);
  rc = MPI_Comm_size(MPI_COMM_WORLD, &size);
  rc = MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  tag=7;
  if (rank==0) {
    strcpy(message, "Hello, world");
    for (int i=1;i<size;++i)
      rc = MPI_SEND(message, 13, MPI_CHAR, i, tag, MPI_COMM_WORLD);
  }
  else
    rc = MPI_RECV(message, 13, MPI_CHAR, 0, tag, MPI_COMM_WORLD,
                  &status);
  std::cout<<"node "<<rank<<": "<<message<<std::endl;
  rc = MPI_Finalize();
}
```

In the sample program, the master process (rank = 0) sends a message consisting of the characters ”Hello, world” to all the other processes (rank > 0). The other processes simply receive this message and print it out. If the sample programs were run using the command `mpirun -np 3 executable`, the output would be

```
node 1: Hello, world
node 2: Hello, world
node 3: Hello, world
```

4 Details about MPI Routines

There are lots topics related with MPI program. For example,

- Data types
- Communication - point-to-point and collective
- Timing
- Grouping data for communications
- Communicators and Topologies

- I/O in parallel
- Debugging parallel program
- Performance
- Etc.

Even though they are all important topics in making *good* parallel MPI/C program, we will introduce only the first three topics of them, which are essencial in making parallel program. If the reader is interested in the others, please refer to [2].

4.1 Data Types with C/C++ Binding

MPI has constants defining the basic datatypes. Each basic datatype in C/C++ has its MPI equivalent which is of type `MPI_Datatype` and should be used in MPI calls in C/C++.

MPI datatype	C/C++ datatype
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	
MPI_PACKED	

`[Table 1] Predefined MPI daratypes`

4.2 Communicators

A communicator handle defines which processes a particular command will apply to. All MPI communication calls take a communicator handle as a parameter, which is effectively the context in which the communication will take place. One of the uses for communicators is to enable software writers to write a message passing parallel libraries which will run in a different, system-assigned context the programs, so the message passing will not crash with in the program. For the most part, whenever a communicator handle is required, beginning MPI programmers can use the predefined value `MPI_COMM_WORLD` which is the global context and includes all the processes in the program.

4.3 Initial MPI Calls

The first MPI call in a program must be MPI_INIT to initialize the environment. This usually followed by a call to MPI_COMM_SIZE to determine the number of processes taking part in communication (the size of the ”virtual machine”), and a call to MPI_COMM_RANK to find out the rank of the calling process within the ”virtual machine”. Following are more explicit details of the initial MPI function calls and associated parameters and syntax in C/C++. In the MPI routine notation, an IN parameter is an input parameter which is not altered by the caller, an OUTPUT parameter is an output parameter which is set by the caller, and an INOUT parameter is used by the routine both for input and output.

```
int MPI_Init(int* argc, char ***argv)
```

This initialization routine must be called once only before any other MPI routine is called.

```
int MPI_Comm_size(MPI_Comm comm, int *size)
  IN comm: communicator (handle)
  OUT size: number of processes in the group of comm (integer)
```

This call returns the number of processes involved in a communicator. When the communicator used is the predefined global communicator MPI_COMM_WORLD, then this function indicates the total number of processes involved in the program.

```
int MPI_Comm_rank(MPI_Comm comm, int *rank)
  IN comm: communicator (handle)
  OUT rank: rank of the calling process in group of comm (integer)
```

This call returns the ”rank” of the current process within a communicator. Every communicator can be considered to contain a group of processes, each of which has a unique integer ”rank” identifier starting from 0 and increasing (0, 1, ..., size $-$ 1).

4.4 Point-to-Point Communication Calls

Point-to-Point communication calls involve sends and receives between two processes. There are two basic categories of sends and receives, which are either *blocking* or *nonblocking*. A blocking call is one that returns when the send (or recieve) is complete. A non-blocking call returns immediately and it is up to the programmer to check for the completion of the call.

There are also four different types of communication modes which are correspond to four versions of send: standard MPI_SEND, buffered MPI_BSEND, synchronous MPI_SSEND, and ready MPI_RSEND. Other very useful calls include the non-blocking standard send MPI_ISEND, the non-blocking receive MPI_IRECV, and their tests for completion MPI_TEST, and MPI_WAIT.

Following are C implementations of MPI_SEND and MPI_RECV, with associated parameters and syntax.

```
int MPI_Send(void* buf, int count, MPI_Datatype datatype,
             int dest, int tag, MPI_Comm comm)
  IN buf: initial address of send buffer (choice)
  IN count: number of elements in send buffer (nonnegative integer)
  IN datatype: datatype of each send buffer element (handle)
  IN dest: rank of destination process (integer)
  IN tag: message tag (integer)
  IN comm: communacator (handle)
```

`MPI_Send` specifies that a message containing `count` elements of a specified `datatype` starting at address `buf` is to be sent using the message tag `tag` to the process ranked `dest` in the communicator `comm`. `MPI_Send` will not return until it can use send buffer.

```
int MPI_Recv(void* buf, int count, MPI_Datatype datatype,
             int source, int tag, MPI_Comm comm, MPI_Status *status)
  IN buf: initial address of send buffer (choice)
  IN count: number of elements in send buffer (nonnegative integer)
  IN datatype: datatype of each send buffer element (handle)
  IN source: rank of sourc or MPI_ANY_SOURCE (integer)
  IN tag: message tag or MPI_ANY_TAG (integer)
  IN comm: communacator (handle)
  OUT status: status object (Status)
```

`MPI_Recv` blocks a process until it receives a message from the process ranked `source` in the communicator `comm` with message tag `tag`. Wild cards `MPI_ANY_SOURCE` and `MPI_ANY_FLAG` can be used to receive messages. If a wild card is used, the returned `status` can be used to determine the actual source and tag. The recieved message is placed in a receive buffer, which consists of the storage containing `count` consecutive elements of the type specified by `datatype`, starting at address `buf`. The length of the received message must be less than or equal to the length of the available receive buffer.

4.5 Collective Communication Calls

Collective communication calls enable a programmer to give command to a subgroup of the processors in the virtual machine. Members of a subgroup are identified by their communicatorm and a processor may be a member of more than one subgroup. Collective communication calls make it easier to perform tasks such as process synchronization, global summation, scattering and gathering data.

The following are detail of `MPI_BARRIER`, `MPI_BCAST`, and `MPI_REDUCE`. Other useful collective communication calls include `MPI_SCATTER`, `MPI_GATHER`, `MPI_ALLREDUCE`, `MPI_SCAN`.

```
int MPI_Barrier(MPI_Comm comm)
  IN comm: communicator (handle)
```

MPI_Barrier blocks the caller until all group members have called it. The call returns at an process only after all group members have entered the call.

```
int MPI_Bcast(void *buffer, int count, MPI_Datatype datatype,
              int root, MPI_Comm comm)
  INOUT buffer: starting address of buffer (choice)
  IN count: number of entries in buffer (integer)
  IN datatype: data type of buffer (handle)
  IN root: rank of broadcast root (integer)
  IN comm: communicator (handle)
```

MPI_Bcast broadcasts a message from the process with rank root to all processes of the group, itself included. Every process gets a copy of count elements of datatype which they put in a local buffer starting at address buffer. MPI_Bcast must be called by all processes in the communicator using the same arguments for comm, root.

```
int MPI_Reduce(void *sendbuf, void *recvbuf, int count,
               MPI_Datatype datatype, MPI_Op op, int root,
               MPI_Comm comm)
  IN sendbuf: address of send buffer (choice)
  OUT recvbuf: address of receive buffer (choice, significant
               only at root)
  IN count: number of elements in send buffer (integer)
  IN datatype: data type of elements in sendbuffer (handle)
  IN op: reduce operation (handle)
  IN root: rank of root process (integer)
  IN comm: communicator (handle)
```

MPI_Reduce combines the elements in the input sendbuf of each processor, and returns the combined value at the root process in recvbuf. There are several predefined operations op including MPI_MAX, MPI_MIN, MPI_SUM, and MPI_PROD.

4.6 Leaving MPI

The MPI_Finalize routine cleans up all MPI states.

```
int MPI_Finalize(void)
```

The user must ensure that all communications are completed before calling MPI_Finalize. Once this routine is called, no other MPI routine may be called (including MPI_Init).

4.7 Timing

MPI defines a timer which is very convenient for performance debugging which provides a portable timing facility.

```
double MPI_Wtime(void)
```

MPI_Wtime returns a floating point number of seconds, representing wall-clock time. To time a process, MPI_Wtime can be called just before the process starts, and again just after the process ends; then calculate the difference between the two times.

5 An Example of MPI/C Program

Now we will see the more interesting MPI/C program that sums an integer array.

5.1 A non-parallel program that sums the values in an array

The following program calculates the sum of the elements of a array. It will be followed by a parallel version of the same program using MPI calls.

```
#include <stdio.h>
#define max_rows 10000000

int array[max_rows];

main(int argc, char **argv)
{
  int i, num_rows;
  long int sum;

  printf("please enter the number of numbers to sum: ");
  scanf("%i", &num_rows);

  if(num_rows > max_rows) {
     printf("Too many numbers.\n");
     exit(1);
  }

  /* initialize an array */

  for(i = 0; i < num_rows; i++)
    array[i] = i;
  /* compute sum */

  sum = 0;
  for(i = 0; i < num_rows; i++)
    sum += array[i];

  printf("The grand total is: %i\n", sum);
}
```

5.2 Design for a parallel program to sum an array

The code below shows a common program structure for including both master and slave segments in the parallel version of the example program just presented. It is composed of a short set-up section followed by a single `if...else` loop where the master process executes the statments between the brackets after the if statement, and the slave processes execute the statements between the brackets after the else statement.

```
/* This program sums all rows in an array using MPI parallelism.
 * The root process acts as a master and sends a portion of the
 * array to each child process.  Master and child processes then
 * all calculate a partial sum of the portion of the array assigned
 * to them, and the child processes send their partial sums to
 * the master, who calculates a grand total.
 **/

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

int main()
{
  int my_id, root_process, ierr, num_procs, an_id;
  MPI_Status status;

  root_process = 0;

  /* Now replicate this process to create parallel processes.
  ierr = MPI_Init(&argc, &argv);

  /* find out MY process ID, and how many processes were started */
  ierr = MPI_Comm_rank(MPI_COMM_WORLD, &my_id);
  ierr = MPI_Comm_size(MPI_COMM_WORLD, &num_procs);

  if(my_id == root_process) {
    /* I must be the root process, so I will query the user
     * to determine how many numbers to sum.
```

```
       * initialize an array,
       * distribute a portion of the array to each child process,
       * and calculate the sum of the values in the segment assigned
       * to the root process,
       * and, finally, I collect the partial sums from slave processes,
       * print them, and add them to the grand sum, and print it */
    }
    else {
      /* I must be slave process, so I must receive my array segment,
       * calculate the sum of my portion of the array,
       * and, finally, send my portion of the sum to the root process. */
    }

    /* Stop this process */
    ierr = MPI_Finalize();
  }
```

5.3 The complete parallel program to sum a array

Here is the expanded parallel version of the same program using MPI calls.

```
/* This program sums all rows in an array using MPI parallelism.
 * The root process acts as a master and sends a portion of the
 * array to each child process.  Master and child processes then
 * all calculate a partial sum of the portion of the array assigned
 * to them, and the child processes send their partial sums to
 * the master, who calculates a grand total.
 **/

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

#define max_rows 100000
#define send_data_tag 2001
#define return_data_tag 2002

int array[max_rows];
int array2[max_rows];

main(int argc, char **argv)
{
  long int sum, partial_sum;
  MPI_Status status;
  int my_id, root_process, ierr, i, num_rows, num_procs,
      an_id, num_rows_to_receive, avg_rows_per_process,
      sender, num_rows_received, start_row, end_row, num_rows_to_send;
```

```
/* Now replicte this process to create parallel processes.
 * From this point on, every process executes a seperate copy
 * of this program */

ierr = MPI_Init(&argc, &argv);

root_process = 0;

/* find out MY process ID, and how many processes were started. */

ierr = MPI_Comm_rank(MPI_COMM_WORLD, &my_id);
ierr = MPI_Comm_size(MPI_COMM_WORLD, &num_procs);

if(my_id == root_process) {
  /* I must be the root process, so I will query the user
   * to determine how many numbers to sum. */

  printf("please enter the number of numbers to sum: ");
  scanf("%i", &num_rows);

  if (num_rows > max_rows) {
    printf("Too many numbers.\n");
    exit(1);
  }

  avg_rows_per_process = num_rows / num_procs;

  /* initialize an array */
  for (i = 0; i < num_rows; i++)
    array[i] = i + 1;

  /* distribute a portion of the bector to each child process */
  for (an_id = 1; an_id < num_procs; an_id++) {
    start_row = an_id*avg_rows_per_process + 1;
    end_row   = (an_id + 1)*avg_rows_per_process;

    if ((num_rows - end_row) < avg_rows_per_process)
      end_row = num_rows - 1;

    num_rows_to_send = end_row - start_row + 1;

    ierr = MPI_Send( &num_rows_to_send, 1 , MPI_INT,
           an_id, send_data_tag, MPI_COMM_WORLD);

    ierr = MPI_Send( &array[start_row], num_rows_to_send, MPI_INT,
               an_id, send_data_tag, MPI_COMM_WORLD);
  }

  /* and calculate the sum of the values in the segment assigned
   * to the root process */

  sum = 0;
  for (i = 0; i < avg_rows_per_process + 1; i++)
    sum += array[i];

  printf("sum %i calculated by root process\n", sum);

  /* and, finally, I collet the partial sums from the slave processes,
   * print them, and add them to the grand sum, and print it */

  for(an_id = 1; an_id < num_procs; an_id++) {
    ierr = MPI_Recv( &partial_sum, 1, MPI_LONG, MPI_ANY_SOURCE,
             return_data_tag, MPI_COMM_WORLD, &status);

    sender = status.MPI_SOURCE;

    printf("Partial sum %i returned from process %i\n", partial_sum, sender);

    sum += partial_sum;
  }

  printf("The grand total is: %i\n", sum);
}

else {
  /* I must be a slave process, so I must receive my array segment,
   * storing it in a "local" array, array1. */

  ierr = MPI_Recv( &num_rows_to_receive, 1, MPI_INT,
           root_process, send_data_tag, MPI_COMM_WORLD, &status);

  ierr = MPI_Recv( &array2, num_rows_to_receive, MPI_INT,
           root_process, send_data_tag, MPI_COMM_WORLD, &status);

  num_rows_received = num_rows_to_receive;

  /* Calculate the sum of my portion of the array */

  partial_sum = 0;
  for(i = 0; i < num_rows_received; i++)
    partial_sum += array2[i];
```

```
      /* and finally, send my partial sum to hte root process */

      ierr = MPI_Send( &partial_sum, 1, MPI_LONG, root_process,
               return_data_tag, MPI_COMM_WORLD);
    }
    ierr = MPI_Finalize();
  }
```

Table 2 shows the values of several variables during the execution of sumarray_mpi. The information comes from a two-processor parallel run, and the values of program variables are shown in both processor memory spaces. Note that there is only one process active prior to the call to MPI_Init.

Program location	Before MPI_Init	After MPI_Init	Before MPI_Send to slave	After MPI_Recv by slave	After MPI_Recv by master
variable name	proc0	proc0 / proc1	proc0 / proc1	proc0 / proc1	proc0 / proc1
root process	0	0 / 0	0 / 0	0 / 0	0 / 0
my_id	·	0 / 1	0 / 1	0 / 1	0 / 1
num_procs	·	2 / 2	2 / 2	2 / 2	2 / 2
num_rows	·	· / ·	6 / ·	6 / ·	6 / ·
avg_rows_per_process	·	· / ·	3 / ·	3 / ·	3 / ·
num_rows_received	·	· / ·	· / ·	· / 3	· / 3
array[0]	·	· / ·	1.0 / ·	1.0 / ·	1.0 / ·
array[1]	·	· / ·	2.0 / ·	2.0 / ·	2.0 / ·
array[2]	·	· / ·	3.0 / ·	3.0 / ·	3.0 / ·
array[3]	·	· / ·	4.0 / ·	4.0 / ·	4.0 / ·
array[4]	·	· / ·	5.0 / ·	5.0 / ·	5.0 / ·
array[5]	·	· / ·	6.0 / ·	6.0 / ·	6.0 / ·
array2[0]	·	· / ·	· / ·	· / 4.0	· / 4.0
array2[1]	·	· / ·	· / ·	· / 5.0	· / 5.0
array2[2]	·	· / ·	· / ·	· / 6.0	· / 6.0
array2[3]	·	· / ·	· / ·	· / ·	· / ·
array2[4]	·	· / ·	· / ·	· / ·	· / ·
array2[5]	·	· / ·	· / ·	· / ·	· / ·
partial_sum	·	· / ·	· / ·	· / ·	6.0 / 15.0
sum	·	· / ·	· / ·	· / ·	21.0 / ·

```
[Table 2] Value histories of selected variables within
          the master(proc0) and slave(proc1) processes during
          a 2-process execution of program sumarray_mpi
```

6 MPI Resourses

- MPI implementations
  - MPICH
    ftp://info.mcs.anl.gov/pub/mpi
  - LAM
    http://www.mpi.nd.edu/lam/download
  - CHIMP
    ftp://ftp.epcc.ed.ac.uk/pub/chimp/release
  - WinMPI: MPI running under Windows 3.1
    ftp://csftp.unomaha.edu/pub/rewini/WinMPI
  - W32MPI: MPI implementation for Win32
    http://dsg.dei.uc.pt/wmpi/intro.html
- Online Tutorials and User's Guide
  - an introductoty MPI Tutorials
    http://www.tc.cornell.edu/Edu/Tutor/MPI
  - Cornell Tutorials
    http://www.tc.cornell.edu/Edu/Tutor/MPI/
  - University of New Maxico
    http://www.arc.unm.edu/workshop/mpi/mpi.html
  - Gropp's Tutorial
    http://www.mcs.anl.gov/mpi/tutorial/gropp/talk.html
  - MPICH User's Guide (HTML)
    http://www.mcs.anl.gov/mpi/mpiuserguide/paper.html
- MPI FAQ
  - http://www.erc.msstate.edu/mpi/mpi-faq.html
- MPI Web Pages
  - Argonne National Labs
    http://www-unix.mcs.anl.gov/mpi
  - Mississippi State Univ
    http://www.erc.msstate.edu/mpi
  - Oak Ridge National Labs
    http://www.epm.ornl.gov/w̃alker/mpi
- MPI Newsgroup
  - comp.parallel.mpi
- MPI Forum

  - http://www.mpi-forum.org

- MPI Example Programs
  - http://www.abo.fi/ mats/HPC1999/examples
  - http://www.npac.syr.edu/projects/cpsedu/summer98summary/examples/mpi-c/mpi-c.html
  - http://www.cacr.caltech.edu/resources/v2500/SystemSoftware/docs/MPI/a_examples.html

7 References

[1] Micheal L. Scott, Programming Language Pragmatics, Morgan Kaufmann Publisher, 2000

[2] Peter S. Pacheo, Parallel Programming with MPI, Morgan Kaufmann Publisher, 1997

[3] Joint Institute for Computational Science, Beginner's Guid to MPI Message Passing Interface, University of Tennessee, 1997