Open MPI User Manual

Open MPI Document Project

September 17, 2013

CONTENTS

LISTINGS

PREFACE

TODO:The preface goes here.

1 INTRODUCTION TO OPEN MPI

1.1 Open MPI The Message Passing Interface (MPI) is a set of definitions for managing multiple processes on a wide variety of computational platforms. Open MPI is an implementation of this set of definitions. The MPI definition is based on the idea of multiple processes running in parallel, and the set of definitions provides a way to create multiple processes, initiate multiple processes, and share information between them.

At its most basic level Open MPI provides a framework in which the computational platform can be treated as a array of independent, distributed memory, distributed processor machines. This framework allows for the creation of programs that can exploit this assumed structure, and it allows for the creation and execution of multiple processes that can be thought of as acting independently of one another. Any information shared between the processes must done so explicitly making use of the Open MPI library.

TODO:Provide a low level introduction to MPI

1.2 Getting Started TODO:Give an example like a mini-tutorial.

1.3 Compilation and Running TODO:Discuss in more detail how to compile and execute a process.

1.3.1 Compilation TODO:How to compile a code

1.3.2 Execution TODO:How to start and execute a code and creating new processes.

mpirun
mpiexec

1.4 High Level Introduction TODO:Give a more detailed introduction. Include some information about the MPI functionality.

From Joshua Hursey:

> A high level introduction to some MPI functionally would be nice, but I would not worry too much about the uglier corner cases of the API.

2 PROCESS MANAGEMENT

2.1 Basic TODO:Introduce the basic ideas.

Basic commands include the following:

- MPI_Init
- MPI_Comm_size
- MPI_Comm_rank
- MPI_Abort
- MPI_Initialized
- MPI_Finalize

Listing 2.1: Basic Process Information

```
1 #include <fstream>
2 #include <iostream>
3 #include <mpi.h>
4
5 // mpic++ -o  mpiManagementExample mpiManagementExample.cpp
6 // mpirun -np 4 --host localhost mpiManagementExample
7
8 int main(int argc,char **argv)
9 {
10   // MPI job information.
11   int  mpiResult;      // Used to check the results from MPI library calls
12   int  numtasks;       // Total number of processes spawned for this job.
13   int  rank;           // The unique number associated with this process.
14
15   // Host Information
16   char hostname[MPI_MAX_PROCESSOR_NAME];
17   int  len;
18
19   // Initialize the session
20   mpiResult = MPI_Init (&argc,&argv);
21   if(mpiResult!= MPI_SUCCESS)
22     {
23       std::cout << "MPI not started. Terminating the process." << std::endl;
24       MPI_Abort(MPI_COMM_WORLD, mpiResult);
25     }
26
27   // Get information about this session and this process
28   MPI_Comm_size(MPI_COMM_WORLD,&numtasks);  // get the number of processes
29   MPI_Comm_rank(MPI_COMM_WORLD,&rank);      // get the rank of this process
30   MPI_Get_processor_name(hostname, &len);   // Get the host name for
31                                             // this process
32
```

```
33   // Print out the information about this process.
34   std::cout << "Number of tasks= " <<  numtasks
35             << " My rank= " << rank
36             << " Running on " << hostname
37             << std::endl;
38
39   // All done. Time to wrap it up.
40   MPI_Finalize();
41   return(0);
42 }
```

2.2 Intermediate TODO:Intermediate ideas and commands associated include the following:

- `MPI_Get_processor_name`
- `MPI_Get_version`
- `MPI_Initialized`

TODO:Talk about groups and communicators.
TODO:Error handling and status.

2.3 Advanced TODO:Intermediate ideas and commands associated include the following:

- `MPI_Wtime`
- `MPI_Wtick`
- Go into details about communicators?

TODO:Talk about groups and communicators.
TODO:Error handling and status.

3 SHARING INFORMATION BETWEEN PROCESSES

The Open MPI libraries provide for a wide variety of approaches for sharing information between processes. The details of how the information is managed and passed between the physical machines is hidden, and the programmer is allowed to focus on the information and how to organize it from the point of view of an individual process. The ideas shared here are divided into three broad sections.

First the most basic functions that allow the programmer to share data are given. The focus in this first section is on synchronous (blocking) communication and non-synchronous (non-blocking) communication. At the most fundamental level synchronous communications will wait until the request to share information is complete from the point of view of the processes that send and receive the information. Asynchronous communications, on the other hand, do not wait, rather they submit the request to either send or receive the information and the process is then allowed to go on and perform other tasks.

In the second section some of the intermediate ideas and functions are given. These build on the basic ideas of synchronous and asynchronous communications but explore the details on ways to manage the buffers used to shared information sent between processes. This section also explores more ways to decide if requests have been completed.

Finally, in the third section some of the more advanced ideas and functions are given. This section builds on the second section and goes a little further into the organization of buffers. Additional details are also explored in how to track the status of a request.

TODO:I am not familiar with the broadcast, scatter and gather routines. Where do they go?

3.1 Basic In this section we introduce the basic ideas of sharing information between processes. The focus is on passing messages, and messages can be passed using synchronous (blocking) and asynchronous (non-blocking) approaches. Each of these two ideas are given in separate subsections starting with synchronous methods.

3.1.1 Synchronous communications TODO:Discuss the idea of blocking sends and receives. Basic introduction to the idea.

TODO:Relies on the process chapter.

Ideas and library calls:

- MPI_Send
- MPI_Recv

Listing 3.1: Example of blocking communication.

```
1 #include <fstream>
2 #include <iostream>
3 #include <mpi.h>
4
5 // mpic++ -o  mpiManagementExample mpiManagementExample.cpp
6 // mpirun -np 4 --host localhost mpiManagementExample
7
8 // The size of the array to be sent to the next process.
9 #define NUMBER 10
10
11 int main(int argc,char **argv)
12 {
13   // MPI job information.
14   int  mpiResult;      // Used to check the results from MPI library calls
15   int  numtasks;       // Total number of processes spawned for this job.
16   int  rank;           // The unique number associated with this process.
17
18   // Information about the host this process is running on.
19   char hostname[MPI_MAX_PROCESSOR_NAME];
20   int len;
21
22   // Information to be shared between processes.
23   double val[NUMBER];
24   int lupe;
25   MPI_Status theStatus;
26
27   // Initialize the session
28   mpiResult = MPI_Init (&argc,&argv);
29   if(mpiResult!= MPI_SUCCESS)
30     {
31       std::cout << "MPI not started. Terminating the process." << std::endl;
32       MPI_Abort(MPI_COMM_WORLD, mpiResult);
33     }
34
35
36   // Get information about this session and this process
37   MPI_Comm_size(MPI_COMM_WORLD,&numtasks);  // get the number of processes
38   MPI_Comm_rank(MPI_COMM_WORLD,&rank);      // get the rank of this process
39   MPI_Get_processor_name(hostname, &len);   // Get the host name for
40                                             // this process
41
42
43   // Decide to start the communication or wait to hear something.
44   if(rank == 0)
45     {
46       // This is the first process. Send my information to the second process.
47
48       // First initialize the buffer
49       for(lupe=0;lupe<NUMBER;++lupe)
50         {
51           val[lupe] = (double)lupe;
```

```
52          }
53
54        // Now pass the information along to the next process.
55        std::cout << "Sending message from the zero process" << std::endl;
56        MPI_Send(val,NUMBER,MPI_DOUBLE,1,10,MPI_COMM_WORLD);
57
58        // Now wait to get the information from the last process in the ring.
59        std::cout << "Zero process waiting to hear the message from " << numtasks-1 << std::endl;
60        MPI_Recv(val,NUMBER,MPI_DOUBLE,MPI_ANY_SOURCE,MPI_ANY_TAG,MPI_COMM_WORLD,&theStatus);
61        std::cout << "Zero process heard from  " << theStatus.MPI_SOURCE << " with tag "
62                  << theStatus.MPI_TAG << std::endl;
63      }
64
65    else
66      {
67        // I am not the first process. Wait until I get the information
68        // before passing along what I have.
69        std::cout << "Process " << rank << " waiting to hear the message" << std::endl;
70        MPI_Recv(val,NUMBER,MPI_DOUBLE,MPI_ANY_SOURCE,MPI_ANY_TAG,MPI_COMM_WORLD,&theStatus);
71        std::cout << "Process " << rank << " heard from  " << theStatus.MPI_SOURCE << " with tag "
72                  << theStatus.MPI_TAG << std::endl;
73
74        // update the information that was sent to me.
75        for(lupe=0;lupe<NUMBER;++lupe)
76          val[lupe] += 1.0;
77
78        // Finally pass it along to the next process in the ring.
79        std::cout << "Process " << rank << " sending message to "
80                  << (rank+1)%numtasks << " process." << std::endl;
81        MPI_Send(val,NUMBER,MPI_DOUBLE,(rank+1)%numtasks,10,MPI_COMM_WORLD);
82      }
83
84    // print out the information that I have.
85    std::cout << "Process " << rank << " heard: ";
86    for(lupe=0;lupe<NUMBER;++lupe)
87      std::cout << val[lupe] << ", ";
88    std::cout << std::endl;
89
90    // All done. Time to wrap it up.
91    MPI_Finalize();
92    return(0);
93  }
```

3.1.2 Aynchronous communications

TODO:Discuss the basic idea of asynchronous communications. Give basic examples. Next subsection goes into more details once richer data structures can be used.

Ideas and library calls:

- `MPI_Isend`
- `MPI_Irecv`

- MPI_Wait*
- MPI_Test*

Listing 3.2: Example of nonblocking communication.

```
1 #include <fstream>
2 #include <iostream>
3 #include <mpi.h>
4
5 // mpic++ -o  mpiManagementExample mpiManagementExample.cpp
6 // mpirun -np 4 --host localhost mpiManagementExample
7
8 // The size of the array to be sent to the next process.
9 #define NUMBER 10
10
11 int main(int argc,char **argv)
12 {
13   // MPI job information.
14   int  mpiResult;      // Used to check the results from MPI library calls
15   int  numtasks;       // Total number of processes spawned for this job.
16   int  rank;           // The unique number associated with this process.
17
18   // Information about the host this process is running on.
19   char hostname[MPI_MAX_PROCESSOR_NAME];
20   int len;
21
22   // Information to be shared between processes.
23   double val[NUMBER];
24   int lupe;
25
26   // Variables to keep track of the status and the requests to send
27   // info.
28   MPI_Status  theStatus[2];
29   MPI_Request theRequests[2];
30
31   // Variables used to determine if the info has come through.
32   int test1;
33   int test2;
34
35   // Initialize the session
36   mpiResult = MPI_Init (&argc,&argv);
37   if(mpiResult!= MPI_SUCCESS)
38     {
39       std::cout << "MPI not started. Terminating the process." << std::endl;
40       MPI_Abort(MPI_COMM_WORLD, mpiResult);
41     }
42
43   // Get information about this session and this process
44   MPI_Comm_size(MPI_COMM_WORLD,&numtasks);  // get the number of processes
45   MPI_Comm_rank(MPI_COMM_WORLD,&rank);      // get the rank of this process
46   MPI_Get_processor_name(hostname, &len);   // Get the host name for
47                                             // this process
48
49   // Initialize the buffer to be sent.
50   for(lupe=0;lupe<NUMBER;++lupe)
51     {
```

```
52        val[lupe] = (double)lupe + rank;
53      }
54
55
56    // pass the info along to the next process
57    std::cout << "Process " << rank << " sending message to "
58              << (rank+1)%numtasks << " process." << std::endl;
59    MPI_Isend(val,NUMBER,MPI_DOUBLE,(rank+1)%numtasks,10,MPI_COMM_WORLD,&theRequests[0]);
60
61    // Wait to hear the message.
62    std::cout << "Process " << rank << " waiting to hear the message" << std::endl;
63    MPI_Irecv(val,NUMBER,MPI_DOUBLE,MPI_ANY_SOURCE,MPI_ANY_TAG,MPI_COMM_WORLD,&theRequests[1]);
64
65    // keep checking to see when everything is done.
66    // This is not a good way to do it, and it is only for demonstration purposes.
67    test1 = 0;
68    test2 = 0;
69    while(!test1 && !test2)
70      {
71        MPI_Test (&theRequests[0],&test1,&theStatus[0]);
72        MPI_Test (&theRequests[1],&test2,&theStatus[1]);
73      }
74
75    // Alternatively, just wait for them all at once....
76    MPI_Waitall(2, theRequests, theStatus);
77
78    // print out the info that was passed
79    std::cout << "Process " << rank << " heard: ";
80    for(lupe=0;lupe<NUMBER;++lupe)
81      std::cout << val[lupe] << ", ";
82    std::cout << std::endl;
83
84    // All done. Time to wrap it up.
85    MPI_Finalize();
86    return(0);
87  }
```

3.1.3 Data and Memory Management

TODO:Discuss memory organization. Discuss the ideas of buffers. Need to discuss the differences between synchronous and asynchronous communications.

- Memory management and buffers
- Data types

3.2 Intermediate

3.2.1 Synchronous communications

TODO:Ideas and library calls:

- `MPI_Ssend`

- `MPI_Bsend`
- `MPI_Buffer_attach`
- `MPI_Buffer_detach`

3.2.2 Aynchronous communications TODO:Ideas and library calls:

- `MPI_Iprobe`
- `MPI_Probe`

3.2.3 Data and Memory Management

- Memory management and buffers
- Data types

3.3 Advanced

3.3.1 Synchronous communications TODO:Ideas and library calls:

- `MPI_Rsend`
- `MPI_Sendrecv`

3.3.2 Aynchronous communications TODO:Ideas and library calls:

- `MPI_Issend`
- `MPI_Ibsend`

3.3.3 Data and Memory Management TODO:Ideas and library calls:

- Memory management and buffers
- Data types

4 FILE OPERATIONS

4.1 Basic TODO:Basic file handling. File pointers. Binary data. Single process examples.

TODO:Partitioning a file and avoiding overlapping data.

TODO:Blocking file operations. Opening a file. Writing basic data to the file. Closing the file.

Listing 4.1: Example of writing information to a file.

```
1 #include <fstream>
2 #include <iostream>
3 #include <string.h>
4 #include <mpi.h>
5
6 // mpic++ -o  mpiManagementExample mpiManagementExample.cpp
7 // mpirun -np 4 --host localhost mpiManagementExample
8
9 // Set the name of the data file and the number of items to write.
10 #define FILE_NAME "fileExample-01.dat"
11 #define NUMBER 10
12
13 int main(int argc,char **argv)
14 {
15   // MPI job information.
16   int  mpiResult;      // Used to check the results from MPI library calls
17   int  numtasks;       // Total number of processes spawned for this job.
18   int  rank;           // The unique number associated with this process.
19
20   // Information to be written to the file.
21   struct output
22   {
23     double x;
24     int    i;
25   };
26
27   // buffers used to write the information.
28   output basicInfo;
29   char   buffer[1024];
30
31   // File related stuffs
32   MPI_File mpiFileHandle;
33
34
35   // Initialize the session
36   mpiResult = MPI_Init (&argc,&argv);
37   if(mpiResult!= MPI_SUCCESS)
38     {
39       std::cout << "MPI_not_started._Terminating_the_process." << std::endl;
40       MPI_Abort(MPI_COMM_WORLD, mpiResult);
41     }
42
43   // Get information about this session and this process
44   MPI_Comm_size(MPI_COMM_WORLD,&numtasks);  // get the number of processes
45   MPI_Comm_rank(MPI_COMM_WORLD,&rank);      // get the rank of this process
46
```

```
47   // open the file
48   std::cout << "opening " << FILE_NAME << std::endl;
49   MPI_Status status;
50   char err_buffer[MPI_MAX_ERROR_STRING];
51   int resultlen;
52   int ierr = MPI_File_open(MPI_COMM_WORLD, FILE_NAME,
53                            MPI_MODE_WRONLY | MPI_MODE_CREATE | MPI_MODE_EXCL,
54                            MPI_INFO_NULL,
55                            &mpiFileHandle);
56
57   // Print out the status of the request.
58   std::cout << "Open: " << ierr << "," << MPI_SUCCESS << std::endl;
59   std::cout << "Status: " << status.MPI_ERROR << ","
60             << status.MPI_SOURCE << ","
61             << status.MPI_TAG << std::endl;
62   MPI_Error_string(ierr,err_buffer,&resultlen);
63   std::cout << "Error: " << err_buffer << std::endl;
64
65
66   if(ierr != MPI_SUCCESS)
67     {
68       // There was an error in trying to create the file. Stop
69       // everything and shut down.
70       std::cout << "Could not open the file. Terminating the process." << std::endl;
71       MPI_Abort(MPI_COMM_WORLD, mpiResult);
72     }
73
74   // Write out the basic information to the file.
75   // First move the file pointer to the correct location.
76   MPI_File_seek(mpiFileHandle,rank*sizeof(basicInfo),MPI_SEEK_SET);
77
78   // Set the data and  copy it to the buffer
79   basicInfo.i = 2*rank;
80   basicInfo.x = 1.0+(float)basicInfo.i;
81   memset(buffer,0,1024);
82   memcpy(buffer,&basicInfo,sizeof(basicInfo));
83
84   // Let everybody know what will be written to the file.
85   std::cout << "rank: " << rank << " moving pointer to "
86             << rank*sizeof(basicInfo)
87             << " writing " <<  basicInfo.i
88             << " and " << basicInfo.x << std::endl;
89
90   // write the date at the current pointer
91   MPI_File_write(mpiFileHandle,buffer,sizeof(basicInfo)/sizeof(char),
92                  MPI_CHAR, &status );
93
94   // close the file
95   MPI_File_close(&mpiFileHandle);
96
97   // All done. Time to wrap it up.
98   MPI_Finalize();
99   return(0);
100 }
```

4.2 Intermediate TODO:Defining a view of a file.

TODO:non-blocking file operations.

4.3 Advanced

TODO:Shared file pointers.

5 TUNING AND OPTIONS

5.1 Introduction TODO:It needs to be made clear that this is specific to Open MPI.

TODO:Discuss what the tuning options are and provide an overview of what they can do.

TODO:Discuss why you might want to change the defaults.

5.2 Basic TODO:I need help here.

5.3 Intermediate TODO:I need help here.

5.4 Advanced TODO:I need help here.