LEARNING mpi

#mpi

Table of Contents

About

Chapter 1: Getting started with mpi

Remarks

MPI is a standard for communication among a group of distributed (or local) processes. It includes routines to send and receive data, communicate collectively, and other more complex tasks.

The standard provides an API for C and Fortran, but bindings to various other languages also exist.

Versions

Version	Standard	Release Date
1	mpi-report-1.3-2008-05-30.pdf	1994-05-05
2.0	mpi2-report.pdf	2003-09-15
2.2	mpi22-report.pdf	2009-09-04
3.0	mpi30-report.pdf	2012-09-21
3.1	mpi31-report.pdf	2015-06-04

Examples

Rank and size

To get the size of a communicator (e.g. `MPI_COMM_WORLD`) and the local process' rank inside it:

```
int rank, size;
int res;
MPI_Comm communicator = MPI_COMM_WORLD;

res = MPI_Comm_rank (communicator, &rank);
if (res != MPI_SUCCESS)
{
  fprintf (stderr, "MPI_Comm_rank failed\n");
  exit (0);
}
res = MPI_Comm_size (communicator, &size);
if (res != MPI_SUCCESS)
{
  fprintf (stderr, "MPI_Comm_size failed\n");
  exit (0);
}
```

Init/Finalize

Before any MPI commands can be run, the environment needs to be initialized, and finalized in the end:

```
int main(int argc, char** argv)
{
    int res;
    res = MPI_Init(&argc,&argv);
    if (res != MPI_SUCCESS)
    {
      fprintf (stderr, "MPI_Init failed\n");
      exit (0);
    }
    ...
    res = MPI_Finalize();
    if (res != MPI_SUCCESS)
    {
      fprintf (stderr, "MPI_Finalize failed\n");
      exit (0);
    }
}
```

Hello World!

Three things are usually important when starting to learn to use MPI. First, you must initialize the library when you are ready to use it (you also need to finalize it when you are done). Second, you will want to know the size of your communicator (the thing you use to send messages to other processes). Third, you will want to know your rank within that communicator (which process number are you within that communicator).

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

int main(int argc, char **argv) {
    int size, rank;
    int res;

    res = MPI_Init(&argc, &argv);
    if (res != MPI_SUCCESS)
    {
        fprintf (stderr, "MPI_Init failed\n");
        exit (0);
    }

    res = MPI_Comm_size(MPI_COMM_WORLD, &size);
    if (res != MPI_SUCCESS)
    {
        fprintf (stderr, "MPI_Comm_size failed\n");
        exit (0);
    }
    res = MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    if (res != MPI_SUCCESS)
    {
        fprintf (stderr, "MPI_Comm_rank failed\n");
        exit (0);
    }

    fprintf(stdout, "Hello World from rank %d of %d~\n", rank, size);

    res = MPI_Finalize();
    if (res != MPI_SUCCESS)
    {
        fprintf (stderr, "MPI_Finalize failed\n");
        exit (0);
    }
}
```

If you run this program like this:

```
mpiexec -n 2 ./hello
```

You would expect to get output like this:

```
Hello World from rank 0 of 2!
Hello World from rank 1 of 2!
```

You could also get that output backward (see http://stackoverflow.com/a/17571699/491687) for more discussion of this:

```
Hello World from rank 1 of 2!
Hello World from rank 0 of 2!
```

Return values of MPI calls

Almost any MPI call returns an integer error code, which signifies the success of the operation. If no error occurs, the return code is `MPI_SUCCESS`:

```
if (MPI_Some_op(...) != MPI_SUCCESS)
{
   // Process error
}
```

If an error occurs, MPI calls an error handler associated with the communicator, window or file object before returning to the user code. There are two predefined error handlers (the user can define additional error handlers):

- `MPI_ERRORS_ARE_FATAL` - errors result in termination of the MPI program
- `MPI_ERRORS_RETURN` - errors result in the error code being passed back to the user

The default error handler for communicators and windows is `MPI_ERRORS_ARE_FATAL`; for file objects it is `MPI_ERRORS_RETURN`. The error handler for `MPI_COMM_WORLD` also applies to all operations that are not specifically related to an object (e.g., `MPI_Get_count`). Thus, checking the return value of non-I/O operations without setting the error handler to `MPI_ERRORS_RETURN` is redundant as erroneous MPI calls will not return.

```
// The execution will not reach past the following line in case of error
int res = MPI_Comm_size(MPI_COMM_WORLD, &size);
if (res != MPI_SUCCESS)
{
    // The following code will never get executed
    fprintf(stderr, "MPI_Comm_size failed: %d\n", res);
    exit(EXIT_FAILURE);
}
```

To enable user error processing, one must first change the error handler of `MPI_COMM_WORLD`:

```
MPI_Comm_set_errhandler(MPI_COMM_WORLD, MPI_ERRORS_RETURN);

int res = MPI_Comm_size(MPI_COMM_WORLD, &size);
if (res != MPI_SUCCESS)
{
    fprintf(stderr, "MPI_Comm_size failed: %d\n", res);
    exit(EXIT_FAILURE);
}
```

The MPI standard does not require that MPI implementations are able to recover from errors and continue the program execution.

Read Getting started with mpi online: https://riptutorial.com/mpi/topic/1943/getting-started-with-mpi

Chapter 2: Collectives

Remarks

Collectives operations are MPI calls designed communicate the processes pointed out by a communicator in a single operation or to perform a synchronization among them. These are often used to calculate one or more values based on data contributed by other processes or to distribute or collect data from all other processes.

Note that all the processes in the communicator should invoke the same collective operations in order, otherwise the application would block.

Examples

Broadcast

The following code broadcasts the contents in `buffer` among all the processes belonging to the `MPI_COMM_WORLD` communicator (i.e. all the processes running in parallel) using the `MPI_Bcast` operation.

```
int rank;
int res;

res = MPI_Comm_rank (MPI_COMM_WORLD, &rank);
if (res != MPI_SUCCESS)
{
    fprintf (stderr, "MPI_Comm_rank failed\n");
    exit (0);
}

int buffer[100];
if (rank == 0)
{
    // Process with rank id 0 should fill the buffer structure
    // with the data it wants to share.
}

res = MPI_Bcast (buffer, 100, MPI_INT, 0, MPI_COMM_WORLD);
if (res != MPI_SUCCESS)
{
    fprintf (stderr, "MPI_Bcast failed\n");
    exit (0);
}
```

Barrier

The `MPI_Barrier` operation performs a synchronization among the processes belonging to the given communicator. That is, all the processes from a given communicator will wait within the `MPI_Barrier` until all of them are inside, and at that point, they will leave the operation.

```
int res;

res = MPI_Barrier (MPI_COMM_WORLD); /* all processes will wait */
if (res != MPI_SUCCESS)
{
    fprintf (stderr, "MPI_Barrier failed\n");
    exit (0);
}
```

Scatter

The root process scatters the contents in `sendbuf` to all processes (including itself) using the `MPI_Scatter` operation.

```
int rank;
int size;
int sendcount = 1;
int recvcount = sendcount;
int sendbuf[3];
int recvbuf;
int root = 0;

MPI_Comm_size (MPI_COMM_WORLD, &size);

if (size != 3)
{
    fprintf (stderr, "Number of processes must be 3\n");
    exit (0);
}

MPI_Comm_rank (MPI_COMM_WORLD, &rank);

if (rank == 0)
{
    sendbuf[0] = 3;
    sendbuf[1] = 5;
    sendbuf[2] = 7;
}

MPI_Scatter (sendbuf, sendcount, MPI_INT, &recvbuf, recvcount, MPI_INT, root, MPI_COMM_WORLD);

printf ("rank: %i, value: %i\n", rank, recvbuf);

/* Output:
 * rank: 0, value: 3
 * rank: 1, value: 5
 * rank: 2, value: 7
 */
```

Read Collectives online: https://riptutorial.com/mpi/topic/2602/collectives

Chapter 3: Compiling an MPI Program

Remarks

MPI needs to add extra libraries and include directories to your compilation line when compiling your program. Rather than tracking all of them yourself, you can usually use one of the compiler wrappers.

Examples

C Wrapper

```
mpicc -o my_prog my_prog.c
```

Read Compiling an MPI Program online: https://riptutorial.com/mpi/topic/3650/compiling-an-mpi-program

Chapter 4: MPI Implementations

Remarks

MPI is a standard, not a programming library. There are many implementations of the standard. The most common open source ones are MPICH and Open MPI. There are many derivatives of these two libraries that are either open source or commercial (or both).

It's important to know which implementation you have because the way you compile or run your program might change subtly.

Examples

Installing MPICH on Mac with Homebrew

```
brew install mpich
```

Installing Open MPI on Mac with Homebrew

```
brew install open-mpi
```

Read MPI Implementations online: https://riptutorial.com/mpi/topic/2870/mpi-implementations

Chapter 5: Process creation and management

Examples

Spawn

Master process spawns two worker processes and scatters `sendbuf` to workers.

master.c

```
#include "mpi.h"

int main(int argc, char *argv[])
{
   int n_spawns = 2;
   MPI_Comm intercomm;

   MPI_Init(&argc, &argv);

   MPI_Comm_spawn("worker_program", MPI_ARGV_NULL, n_spawns, MPI_INFO_NULL, 0, MPI_COMM_SELF,
&intercomm, MPI_ERRCODES_IGNORE);

   int sendbuf[2] = {3, 5};
   int recvbuf; // redundant for master.

   MPI_Scatter(sendbuf, 1, MPI_INT, &recvbuf, 1, MPI_INT, MPI_ROOT, intercomm);

   MPI_Finalize();
   return 0;
}
```

worker.c

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

int main(int argc, char *argv[])
{
   MPI_Init(&argc, &argv);

   MPI_Comm intercomm;
   MPI_Comm_get_parent(&intercomm);

   int sendbuf[2]; // redundant for worker.
   int recvbuf;

   MPI_Scatter(sendbuf, 1, MPI_INT, &recvbuf, 1, MPI_INT, 0, intercomm);
   printf("recvbuf = %d\n", recvbuf);

   MPI_Finalize();
   return 0;
}
```

Execution

```
mpicc master.c -o master_program
mpicc worker.c -o worker_program
mpirun -n 1 master_program
```

Establishing connection between two independent applications

Master process spawns server and client applications with a single process for each application. Server opens a port and client connects to that port. Then client sends data to server with `MPI_Send` to verify that the connection is established.

master.c

```
#include "mpi.h"

int main(int argc, char *argv[])
{
   MPI_Init(&argc, &argv);

   MPI_Comm intercomm;
   // Spawn two applications with a single process for each application.
   // Server must be spawned before client otherwise the client will complain at
MPI_Lookup_name().
   MPI_Comm_spawn("server", MPI_ARGV_NULL, 1, MPI_INFO_NULL, 0, MPI_COMM_SELF, &intercomm,
MPI_ERRCODES_IGNORE);
   MPI_Comm_spawn("client", MPI_ARGV_NULL, 1, MPI_INFO_NULL, 0, MPI_COMM_SELF, &intercomm,
MPI_ERRCODES_IGNORE);

   MPI_Finalize();
   return 0;
}
```

server.c

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

int main(int argc, char *argv[])
{
   MPI_Init(&argc, &argv);

   // Open port.
   char port_name[MPI_MAX_PORT_NAME];
   MPI_Open_port(MPI_INFO_NULL, port_name);

   // Publish port name and accept client.
   MPI_Comm client;
   MPI_Publish_name("name", MPI_INFO_NULL, port_name);
   MPI_Comm_accept(port_name, MPI_INFO_NULL, 0, MPI_COMM_WORLD, &client);

   // Receive data from client.
   int recvbuf;
   MPI_Recv(&recvbuf, 1, MPI_INT, 0, 0, client, MPI_STATUS_IGNORE);
   printf("recvbuf = %d\n", recvbuf);

   MPI_Unpublish_name("name", MPI_INFO_NULL, port_name);

   MPI_Finalize();
```

```
    return 0;
}
```

client.c

```
#include "mpi.h"

int main(int argc, char *argv[])
{
    MPI_Init(&argc, &argv);

    // Look up for server's port name.
    char port_name[MPI_MAX_PORT_NAME];
    MPI_Lookup_name("name", MPI_INFO_NULL, port_name);

    // Connect to server.
    MPI_Comm server;
    MPI_Comm_connect(port_name, MPI_INFO_NULL, 0, MPI_COMM_SELF, &server);

    // Send data to server.
    int sendbuf = 3;
    MPI_Send(&sendbuf, 1, MPI_INT, 0, 0, server);

    MPI_Finalize();
    return 0;
}
```

Command line

```
mpicc master.c -o master_program
mpicc server.c -o server
mpicc client.c -o client
mpirun -n 1 master_program
```

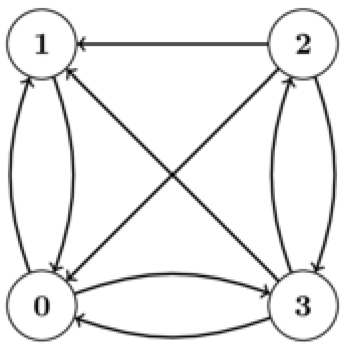
Read Process creation and management online: https://riptutorial.com/mpi/topic/9416/process-creation-and-management

Chapter 6: Process Topologies

Examples

Graph topology creation and communication

Creates a graph topology in a distributed manner so that each node defines its neighbors. Each node communicates its rank among neighbors with `MPI_Neighbor_allgather`.



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

#define nnode 4

int main()
{
    MPI_Init(NULL, NULL);

    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    int source = rank;
    int degree;
    int dest[nnode];
    int weight[nnode] = {1, 1, 1, 1};
    int recv[nnode] = {-1, -1, -1, -1};
    int send = rank;

    // set dest and degree.
    if (rank == 0)
    {
        dest[0] = 1;
        dest[1] = 3;
        degree = 2;
    }
    else if(rank == 1)
    {
        dest[0] = 0;
        degree = 1;
    }
    else if(rank == 2)
    {
        dest[0] = 3;
        dest[1] = 0;
        dest[2] = 1;
        degree = 3;
    }
    else if(rank == 3)
    {
        dest[0] = 0;
        dest[1] = 2;
        dest[2] = 1;
        degree = 3;
    }

    // create graph.
    MPI_Comm graph;
    MPI_Dist_graph_create(MPI_COMM_WORLD, 1, &source, &degree, dest, weight, MPI_INFO_NULL, 1,
&graph);

    // send and gather rank to/from neighbors.
    MPI_Neighbor_allgather(&send, 1, MPI_INT, recv, 1, MPI_INT, graph);

    printf("Rank: %i, recv[0] = %i, recv[1] = %i, recv[2] = %i, recv[3] = %i\n", rank,
recv[0], recv[1], recv[2], recv[3]);

    MPI_Finalize();
    return 0;
}
```

Read Process Topologies online: https://riptutorial.com/mpi/topic/9423/process-topologies

Chapter 7: Running an MPI Program

Parameters

Parameter	Details
`-n <num_procs>`	The number of MPI processes to start up for the job

Examples

Execute your job

The simplest way to run your job is to use `mpiexec` or `mpirun` (they are usually the same thing and aliases of each other).

```
mpiexec -n 2 ./my_prog
```

Read Running an MPI Program online: https://riptutorial.com/mpi/topic/2877/running-an-mpi-program

Credits

S. No	Chapters	Contributors
1	Getting started with mpi	Community, foxcub, Harald, haraldkl, Hristo Iliev, Wesley Bland
2	Collectives	foxcub, Harald, Shibli, Wesley Bland
3	Compiling an MPI Program	Harald, Wesley Bland
4	MPI Implementations	Wesley Bland
5	Process creation and management	Shibli
6	Process Topologies	Shibli
7	Running an MPI Program	Wesley Bland