Introduction to the Message Passing Interface (MPI)

CPS343

Parallel and High Performance Computing

Spring 2018

Outline

Acknowledgements

Some material used in creating these slides comes from

- MPI Programming Model: Desert Islands Analogy by Henry Neeman, University of Oklahoma Supercomputing Center.
- An Introduction to MPI by William Gropp and Ewing Lusk, Argonne National Laboratory.

Outline

What is MPI?

- MPI stands for *Message Passing Interface* and is a library specification for message-passing, proposed as a standard by a broadly based committee of vendors, implementors, and users.
- MPI consists of
  1. a header file `mpi.h`
  2. a library of routines and functions, and
  3. a runtime system.
- MPI is for parallel computers, clusters, and heterogeneous networks.
- MPI is full-featured.
- MPI is designed to provide access to advanced parallel hardware for
  - end users
  - library writers
  - tool developers
- MPI can be used with C/C++, Fortran, and many other languages.

MPI is an API

MPI is actually just an *Application Programming Interface* (API). As such, MPI

- specifies what a call to each routine should look like, and how each routine should behave, but
- does not specify how each routine should be implemented, and sometimes is intentionally vague about certain aspects of a routines behavior;
- implementations are often platform vendor specific, and
- has multiple open-source and proprietary implementations.

Example MPI routines

The following routines are found in nearly every program that uses MPI:

- `MPI_Init()` starts the MPI runtime environment.
- `MPI_Finalize()` shuts down the MPI runtime environment.
- `MPI_Comm_size()` gets the number of processes, N_p.
- `MPI_Comm_rank()` gets the process ID of the current process which is between 0 and $N_p - 1$, inclusive.

(These last two routines are typically called right after `MPI_Init()`.)

More example MPI routines

Some of the simplest and most common communication routines are:

- `MPI_Send()` sends a message from the current process to another process (the *destination*).
- `MPI_Recv()` receives a message on the current process from another process (the *source*).
- `MPI_Bcast()` broadcasts a message from one process to all of the others.
- `MPI_Reduce()` performs a *reduction* (e.g. a global sum, maximum, etc.) of a variable in all processes, with the result ending up in *a single process*.
- `MPI_Allreduce()` performs a *reduction* of a variable in all processes, with the result ending up in *all processes*.

Outline

MPI Hello world: hello.c

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

int main ( int argc, char *argv[] )
{
    int rank;
    int number_of_processes;

    MPI_Init( &argc, &argv );
    MPI_Comm_size( MPI_COMM_WORLD, &number_of_processes );
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );

    printf( "hello from process %d of %d\n",
                                rank, number_of_processes );

    MPI_Finalize();

    return 0;
}
```

MPI Hello world output

Running the program produces the output

```
hello from process 3 of 8
hello from process 0 of 8
hello from process 1 of 8
hello from process 7 of 8
hello from process 2 of 8
hello from process 5 of 8
hello from process 6 of 8
hello from process 4 of 8
```

Note:

- All MPI processes (normally) run the same executable
- Each MPI process knows which rank it is
- Each MPI process knows how many processes are part of the same job
- The processes run in a non-deterministic order

Communicators

Recall the MPI initialization sequence:

```
MPI_Init( &argc, &argv );
MPI_Comm_size( MPI_COMM_WORLD, &number_of_processes );
MPI_Comm_rank( MPI_COMM_WORLD, &rank );
```

- MPI uses communicators to organize how processes communicate with each other.
- A single communicator, `MPI_COMM_WORLD`, is created by `MPI_Init()` and all the processes running the program have access to it.
- Note that process ranks are relative to a communicator. A program may have multiple communicators; if so, a process may have multiple ranks, one for each communicator it is associated with.

MPI is (usually) SPMD

- Usually MPI is run in SPMD (Single Program, Multiple Data) mode.
- (It is possible to run multiple programs, i.e. MPMD).
- The program can use its rank to determine its role:

```
const int SERVER_RANK = 0;

if ( rank == SERVER_RANK )
{
  /* do server stuff */
}
else
{
  /* do compute node stuff */
}
```

- as shown here, often the rank 0 process plays the role of server or process coordinator.

A second MPI program: greeting.c

The next several slides show the source code for an MPI program that works on a client-server model.

- When the program starts, it initializes the MPI system then determines if it is the server process (rank 0) or a client process.
- Each client process will construct a string message and send it to the server.
- The server will receive and display messages from the clients one-by-one.

greeting.c: main

```c
#include <stdio.h>
#include <mpi.h>
const int SERVER_RANK = 0;
const int MESSAGE_TAG = 0;

int main ( int argc, char *argv[] )
{
    int rank, number_of_processes;
    MPI_Init( &argc, &argv );
    MPI_Comm_size( MPI_COMM_WORLD, &number_of_processes );
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );

    if ( rank == SERVER_RANK )
        do_server_work( number_of_processes );
    else
        do_client_work( rank );

    MPI_Finalize();
    return 0;
}
```

greeting.c: server

```
void do_server_work( int number_of_processes )
{
    const int max_message_length = 256;
    char message[max_message_length];
    int src;
    MPI_Status status;

    for ( src = 0; src < number_of_processes; src++ )
    {
        if ( src != SERVER_RANK )
        {
            MPI_Recv( message, max_message_length, MPI_CHAR,
                      src, MESSAGE_TAG, MPI_COMM_WORLD,
                      &status );
            printf( "Received: %s\n", message );
        }
    }
}
```

greeting.c: client

```
void do_client_work( int rank )
{
    const int max_message_length = 256;
    char message[max_message_length];
    int message_length;

    message_length =
      sprintf( message, "Greetings from process %d", rank );
    message_length++;    /* add one for null char */

    MPI_Send( message, message_length, MPI_CHAR,
              SERVER_RANK, MESSAGE_TAG, MPI_COMM_WORLD );
}
```

Outline

Compiling an MPI program

- Compiling a program for MPI is almost just like compiling a regular C or C++ program
- Compiler "front-ends" have been defined that know where the `mpi.h` file and the proper libraries are found.
- The C compiler is `mpicc` and the C++ compiler is `mpic++`.
- For example, to compile `hello.c` you would use a command like

```
mpicc -O2 -o hello hello.c
```

Running an MPI program

Here is a sample session compiling and running the program `greeting.c`.

```
$ mpicc -O2 -o greeting greeting.c
$ mpiexec -n 1 greeting
$ mpiexec -n 2 greeting
Greetings from process 1
$ mpiexec -n 4 greeting
Greetings from process 1
Greetings from process 2
Greetings from process 3
```

Note:

- the server process (rank 0) does not send a message, but does display the contents of messages received from the other processes.
- `mpirun` can be used rather than `mpiexec`.
- the arguments to `mpiexec` vary between MPI implementations.
- `mpiexec` (or `mpirun`) may not be available.

Deterministic operation?

You may have noticed that in the four-process case the greeting messages were printed out in-order. Does this mean that the order the messages were sent is deterministic? Look again at the loop that carries out the server's work:

```
for ( src = 0; src < number_of_processes; src++ )
{
    if ( src != SERVER_RANK )
    {
        MPI_Recv( message, max_message_length, MPI_CHAR,
                  src, MESSAGE_TAG, MPI_COMM_WORLD, &status );
        printf( "Received: %s\n", message );
    }
}
```

The server loops over values of `src` from 0 to `number_of_processes`, skipping the server's own rank. The fourth argument to `MPI_Recv()` is the rank of the process the message is received from. Here the messages are received in increasing rank order, regardless of the sending order.

Non-deterministic receive order

By making one small change, we can allow the messages to be received in any order. The constant `MPI_ANY_SOURCE` can be used in the `MPI_Recv()` function to indicate that the next available message with the correct tag should be read, regardless of its source.

```
for ( src = 0; src < number_of_processes; src++ )
{
    if ( src != SERVER_RANK )
    {
        MPI_Recv( message, max_message_length, MPI_CHAR,
                  MPI_ANY_SOURCE, MESSAGE_TAG, MPI_COMM_WORLD,
                  &status );
        printf( "Received: %s\n", message );
    }
}
```

Note: it is possible to use the data returned in `status` to determine the message's source.

Outline

MPI function return values

- Nearly all MPI functions return an integer status code:
  - `MPI_SUCCESS` if function completed without error,
  - otherwise an error code is returned,
- Most examples you find on the web and in textbooks do not check the MPI function return status value...
- ...but this should be done in production code.
- It can certainly help to avoid errors during development.

Sample MPI error handler

Here is a sample MPI error handler. If called with a non-successful status value, it displays a string corresponding to the error and then exits – all errors are treated as fatal.

```
void mpi_check_status( int mpi_status )
{
  if ( mpi_status != MPI_SUCCESS )
  {
    int len;
    char err_string[MPI_MAX_ERROR_STRING];
    MPI_Error_string( mpi_status, err_string, &len );
    fprintf( stderr, "MPI Error: (%d) %s\n",
                              mpi_status, err_string );
    exit( EXIT_FAILURE );
  }
}
```

Outline

MPI_Send()

The calling sequence for `MPI_Send()` is

```
int MPI_Send(
  void *buf,                /* pointer to send buffer */
  int count,                /* number of items to send */
  MPI_Datatype datatype,    /* datatype of buffer elements */
  int dest,                 /* rank of destination process */
  int tag,                  /* message type identifier */
  MPI_Comm comm)            /* MPI communicator to use */
```

- The `MPI_Send()` function initiates a *blocking send*. Here "blocking" does not indicate that the sender waits for the message to be received, but rather that *the sender waits for the message to be accepted by the MPI system*.
- It does mean that once this function returns the send buffer may be changed with out impacting the send operation.

MPI_Recv()

The calling sequence for `MPI_Recv()` is

```
int MPI_Recv(
  void *buf,                /* pointer to send buffer */
  int count,                /* number of items to send */
  MPI_Datatype datatype,    /* datatype of buffer elements */
  int source,               /* rank of sending process */
  int tag,                  /* message type identifier */
  MPI_Comm comm,            /* MPI communicator to use */
  MPI_Status *status)       /* MPI status object */
```

- The `MPI_Recv()` function initiates a *blocking receive*. It will not return to its caller until a message with the specified tag is received from the specified source.
- `MPI_ANY_SOURCE` may be used to indicate the message should be accepted from any source.
- `MPI_ANY_TAG` may be used to indicate the message should be accepted regardless of its tag.

MPI Datatypes

- Here is a list of the most commonly used MPI datatypes. There are others and users can construct their own datatypes to handle special situations.

C/C++ datatype	MPI datatype
`char`	`MPI_CHAR`
`int`	`MPI_INT`
`float`	`MPI_FLOAT`
`double`	`MPI_DOUBLE`

- The `count` argument in both `MPI_Send()` and `MPI_Recv()` specifies the number of elements of the indicated type in the buffer, *not the number of bytes in the buffer.*

MPI Tags

MPI uses tags to identify messages. Why is this necessary? Isn't just knowing the source or destination sufficient?

Often processes need to communicate different types of messages. For example, one message might contain a domain specification and another message might contain domain data.

Some things to keep in mind about tags:

- message tags can be any integer.
- use `const int` to name tags; it's poor programming style to use numbers for tags in send and receive function calls.
- some simple programs (like `greeting.c`) can use a single tag.

Three more communication functions

Send and Receive carry out *point-to-point* communication; one process sending a message to one other process.

MPI supports many other communication functions. Three others that we'll examine today are

1. `MPI_Bcast()`: single process sends message to all processes.
2. `MPI_Reduce()`: data values in all processes are reduced via some operation (e.g. sum, max, etc.) to a single value on a single process.
3. `MPI_Allreduce()`: data values in all processes are reduced via some operation to a single value available on all processes.

These are examples of *collective communication*.

MPI_Bcast()

The calling sequence for `MPI_Bcast()` is

```
int MPI_Bcast(
  void *buf,                /* pointer to send buffer */
  int count,                /* number of items to send */
  MPI_Datatype datatype,    /* datatype of buffer elements */
  int root,                 /* rank of broadcast root */
  MPI_Comm comm)            /* MPI communicator to use */
```

- all processes call `MPI_Bcast()`
- the data pointed to by `buf` in the process with rank `root` is sent to all other processes
- upon return, the data pointed to by `buf` in each process is the same.

MPI_Bcast() example

What will be displayed by the code segment below when run on 4 processors?

```
MPI_Comm_size( MPI_COMM_WORLD, &number_of_processes );
MPI_Comm_rank( MPI_COMM_WORLD, &rank );
data = 111 * rank;
printf( "Before: process %d has data %03d\n", rank, data );
root = 6 % number_of_processes;
MPI_Bcast( &data, 1, MPI_INT, root, MPI_COMM_WORLD );
printf( "After:  process %d has data %03d\n", rank, data );
```

MPI_Bcast() example output

Here is some example output when run on 4 processors:

```
Before: process 1 has data 111
Before: process 3 has data 333
After : process 3 has data 222
After : process 1 has data 222
Before: process 0 has data 000
After : process 0 has data 222
Before: process 2 has data 222
After : process 2 has data 222
```

Of course, the statements could appear in any order. We see that the data from the process with rank 2 (since `root` $= 2$) was broadcast to all processes.

MPI_Reduce()

The calling sequence for `MPI_Reduce()` is

```
int MPI_Reduce(
  void *sendbuf,           /* pointer to send buffer */
  void *recvbuf,           /* pointer to receive buffer */
  int count,               /* number of items to send */
  MPI_Datatype datatype,   /* datatype of buffer elements */
  MPI_Op op,               /* reduce operation */
  int root,                /* rank of root process */
  MPI_Comm comm)           /* MPI communicator to use */
```

- Contents of `sendbuf` on all processes are combined via the reduction operation specified by op (e.g. `MPI_SUM`, `MPI_MAX`, etc.) with the result being placed in `recvbuf` on the process with rank `root`.
- The contents of `sendbuf` on all processes is unchanged.

MPI_Allreduce()

The calling sequence for `MPI_Allreduce()` is

```
int MPI_Allreduce(
  void *sendbuf,           /* pointer to send buffer */
  void *recvbuf,           /* pointer to receive buffer */
  int count,               /* number of items to send */
  MPI_Datatype datatype,   /* datatype of buffer elements */
  MPI_Op op,               /* reduce operation */
  MPI_Comm comm)           /* MPI communicator to use */
```

- Contents of `sendbuf` on all processes are combined via the reduction operation specified by op (e.g. `MPI_SUM`, `MPI_MAX`, etc.) with the result being placed in `recvbuf` on all processes.

MPI_Reduce() and MPI_Allreduce() example

What will be displayed by the code segment below when run on 4 processors?

```
const int MASTER_RANK = 0;
sum = 0;
MPI_Reduce( &rank, &sum, 1, MPI_INT,
            MPI_SUM, MASTER_RANK, MPI_COMM_WORLD );
printf( "Reduce: process %d has %3d\n",
        rank, sum );

sum = 0;
MPI_Allreduce( &rank, &sum, 1, MPI_INT,
               MPI_SUM, MPI_COMM_WORLD );
printf( "Allreduce: process %d has %3d\n",
        rank, sum );
```

MPI_Reduce() and MPI_Allreduce() example output

Here is some example output when run on 4 processors:

```
Reduce: process 1 has   0
Reduce: process 2 has   0
Reduce: process 3 has   0
Reduce: process 0 has   6
Allreduce: process 1 has   6
Allreduce: process 2 has   6
Allreduce: process 0 has   6
Allreduce: process 3 has   6
```

As in the last example, the statements could appear in any order.