Part B - Languages

MPI - Fundamentals

Introduce distributed memory programming
Define the MPI primitives
Outline the structure of Point-to-Point Communications
Show a multiple program multiple data solution using MPI

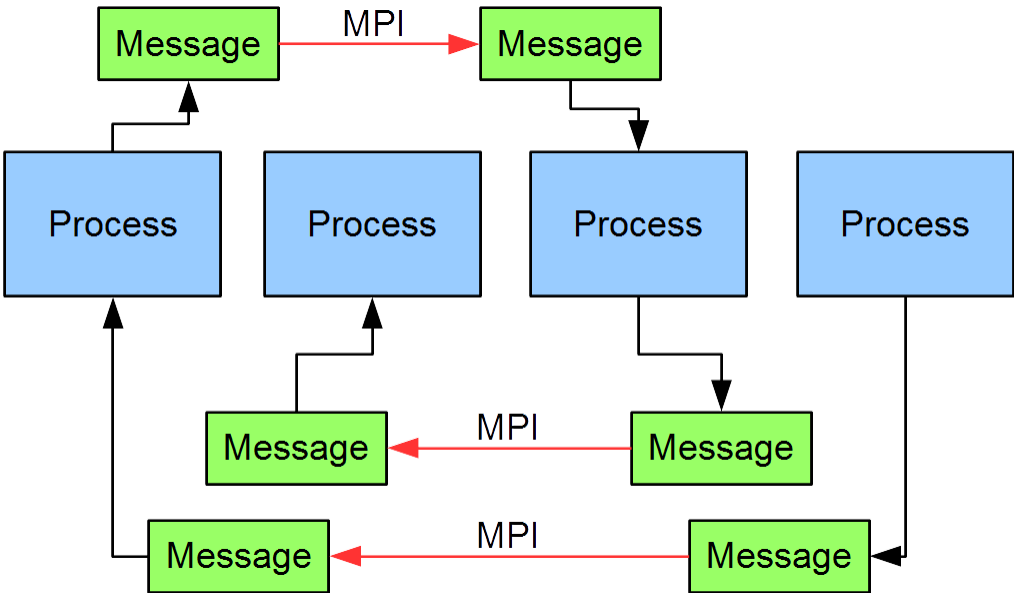
MPI stands for Message-Passing Interface. The MPI specifications define the rules for implementing a message-passing library and environment for parallel programming of distributed memory compututing devices. MPI has a sufficiently flexible design to map processes to multiple cores as well as to multiple hardware nodes. MPI is suitable for MIMD programs as well as SPMD programs.

This chapter introduces the MPI programming model, its primitives and how to define point-to-point communications between processes. This chapter includes a complete example of an MPMD program that illustrates the use of the MPI primitives.

INTRODUCTION

The latest version of the MPI standard is 3.0 (MPI-3), published in 2012. The original standard was proposed in 1994. MPI envisages distributed memory computing as a set of processes that exchange messages with one another. The hardware does not need to be homogeneous; that is, each process can run on differently designed hardware. The MPI run-time handles the mapping of the processes to the hardware - be it to the different cores of a processor, to the different processors within a compute node or to the different compute nodes of a cluster.

The MPI specification publishes a set of standards for implementing message passing between processes. The specification is written in a language-egnostic manner.



Message Passing in Distributed Memory Computing

Implementations of the MPI standard offer bindings for C and Fortran programs. The standard deprecated C++ bindings in MPI 2.2 and completely removed them from MPI-3. Boost.MPI provides an alternative message-passing solution (to MPI) for C++ programmers.

The installation instructions for a few of the current implementations are listed in the MPI section of the chapter entitled Compiler Support.

Communicators

MPI communicates across the different processes using one or more communicators.

Each communicator identifies a context for its communications between the processes. All messages remain within their own context and do not cross contexts. Messages passed in one context do not interfere with those passed in another context.

MPI predefines a global communicator called **MPI_COMM_WORLD**. Programs with only one communication context can use **MPI_COMM_WORLD** throughout. MPI lets us define our own communicators for more advanced applications.

Hello World

The following MPI program launches the number of processes specified on the command line and prints a message from each process.

```
// MPI Program - Hello World
// mpi_hello.c

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

int main(int argc, char** argv) {

    int rank, np;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &np);
    printf("Hello from process %d of %d\n",
     rank, np);
    MPI_Finalize();

    return 0;
}
```

```
On a Windows platform:

C:\...\MPI\Debug>mpiexec -n 8 MPI
Hello from process 5 of 8
Hello from process 7 of 8
Hello from process 6 of 8
Hello from process 4 of 8
Hello from process 2 of 8
Hello from process 0 of 8
Hello from process 1 of 8
Hello from process 3 of 8
```

- **MPI_Comm_rank()** - returns the rank of the executing process.
- **MPI_Comm_size()** - returns the total number of processes.

PRIMITIVES

The MPI library and the MPI environment handle four aspects of communications between processes:

1. identify process - rank of a process within the communicator (a unique non-negative number identifying the process in the communicator)
2. message routing - delivering the message through sockets or shared buffers
3. message buffering - on dispatch and/or while awaiting collection
4. data marshalling - converting processor specific data representations to MPI representations and vice versa

Message Structure

An MPI message consists of two parts: the data being transferred and the envelope describing the transfer. The data itself consists of one of more data values. The envelope identifies the

- source process
- destination process
- message tag

- communicator

The source and destination fields hold the respective ranks in the communication. The tag field identifies the type of message.

Tags allow the communicators to filter or distinguish the different messages being transferred. A tag value can be any non-negative integer within the implementation defined range. The value **`MPI_ANY_TAG`** identifies any incoming message from the source process regardless of its tag at the source end.

Function Primitives

The MPI primitive functions for non-collective communications include:

- Blocking
  - **`MPI_Send()`, `MPI_Bsend()`, `MPI_Ssend()`, `MPI_Rsend()`** - send a message
  - **`MPI_Recv()`** - receive a message
  - **`MPI_Wait()`** - block until communication completes
- Non-Blocking
  - **`MPI_Isend()`, `MPI_Ibsend()`, `MPI_Issend()`, `MPI_Irsend()`** - send a message
  - **`MPI_Irecv()`** - receive a message
  - **`MPI_Test()`** - test if the communication has completed

The section below entitled Point-to-Point Communication describes these functions in detail.

The MPI functions use parameter types that are either standard C types or predefined MPI paramter types.

Predefined Parameter Types

The predefined parameter types include:

- **`MPI_Datatype`** - an MPI data type
- **`MPI_Comm`** - a communicator type
- **`MPI_Status`** - a structure that holds a message's parameters
- **`MPI_Request`** - a handle to polling information
- **`MPI_Errhandler`** - a structure that identifies the error handler

Predefined MPI_Datatype Values

The admissible values for an **`MPI_Datatype`** include the following. The corresponding C type is listed on the right:

MPI_Datatype	C type
MPI_CHAR	signed char
MPI_UNSIGNED_CHAR	unsigned char
MPI_INT	signed int
MPI_LONG	signed long
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double

MPI_Status

An **`MPI_Status`** object holds the information that describes a communication. Its fields are:

```
struct MPI_Status {
    int MPI_SOURCE;    // the rank of the source
    int MPI_TAG;       // the tag set by the source
    int MPI_ERROR;     // the error code value if not 0
    int _count;        // size of the message in bytes
```

```
    int _cancelled;    //
};
```

POINT-TO-POINT COMMUNICATIONS

The MPI send and receive primitives provide point-to-point communication functionality. The communications can be

- blocking, or
- non-blocking

In a blocking communications the function primitive does not return control until MPI has stored the message away safely. In a non-blocking communication the function primitive initiates the communication and may return control before MPI has stored the message away safely.

Blocking Send and Receive

The **MPI_Send** and **MPI_Recv** primitives are the most common.

MPI_Send

The **MPI_Send** primitive implements the standard mode of communication in MPI. It returns control to the caller as soon as MPI copies the message into a local buffer in the case of a short message and as soon as the destination process collects the message in the case of a long one. The former mode is called local blocking. In other words, **MPI_Send** may block locally or globally.

The prototype for the **MPI_Send** primitive is

```
int                    // error code if not 0
MPI_Send(
    // message description
    void *in,          // address of the send buffer
    int count,         // number of items in the message
    MPI_Datatype type, // MPI type being sent

    // destination process
    int dest,          // rank of the destination process
    int tag,           // type of message
    MPI_Comm comm      // destination's communicator context
)
```

MPI_Recv

The prototype for the **MPI_Recv** primitive is

```
int                    // error code if not 0
MPI_Recv(
    // message description
    void *out,         // address of the receive buffer
    int count,         // number of items in the message
    MPI_Datatype type, // MPI type being received

    // source process
    int source,        // rank of the source process
    int tag,           // type of message
    MPI_Comm comm,     // source's communicator context

    // message parameters
    MPI_Status* status // address of the object that holds message information
)
```

MPI holds the programmer responsible for memory allocation. The receipt buffer (`*out`) must be large enough to accommodate the data received. The source process' identifier (`source`) may be either its rank in `comm` or `MPI_ANY_SOURCE` in `comm`. The message type (`tag`) may be a non-negative number or `MPI_ANY_TAG`, which stands for any incoming message from rank `source` in `comm`.

MPI_Get_count

A call to the `MPI_Get_count()` function primitive returns the number of items in the incoming message:

```
int                     // error code if not 0
MPI_Get_count(
    MPI_Status* status, // address of the status structure
    MPI_Datatype type,  // MPI data type expected
    int* count          // number of items read
)
```

Alternative Modes of Blocking Communication

The alternatives modes of communication include

- buffered (`MPI_Bsend()`) - always locally blocking - MPI always copies the message to a local buffer - the user provides the local buffer
- synchronous (`MPI_Ssend()`) - returns only after the destination process has initiated receipt - this mode is called globally blocking
- ready (`MPI_Rsend()`) - the send operation succeeds only if the the destination process has initiated receipt of the message - returns an error code otherwise

Buffered Communication

For buffered communications, we allocate enough space to hold the entire message. We attach the allocated buffer to MPI using

```
int                // error code if not 0
MPI_Buffer_attach(
    void* buffer,  // address of the allocated buffer
    int size       // buffer size in bytes
)
```

We detach the allocated buffer from MPI using

```
int                 // error code if not 0
MPI_Buffer_detach(
    void** buffer,  // address of the buffer's pointer
    int* size       // address of the buffer's size
)
```

The detach reduces the buffer size to zero, but does not deallocate the memory.

For example,

```
#define BUFFER_SIZE (1024 * 1024)

char* buffer = (char*)malloc(BUFFER_SIZE * sizeof(char));

MPI_Buffer_attach(buffer, BUFFER_SIZE);
// ...
MPI_Bsend(msg, ...);
// ...
int size;
```

```
MPI_Buffer_detach(&buffer, &size);
// ...
// can still reuse the buffer - memory is still allocated
// ...

free(buffer);
```

Non-Blocking Send and Receive

Performance is optimized if no blocking occurs. The **MPI_Isend** and **MPI_Irecv** primitives provide no-blocking functionality. Non-blocking communication requires polling to determine its status.

MPI_Isend

The prototype for the **MPI_Isend** primitive is

```
int                     // error code if not 0
MPI_Isend(
    // message description
    void *in,           // address of the send buffer
    int count,          // number of items in the message
    MPI_Datatype type, // MPI type being sent

    // destination process
    int dest,           // rank of the destination process
    int tag,            // type of message
    MPI_Comm comm,      // destination's communicator context

    // polling information
    MPI_Request* req    // address of request object
)
```

Multiple sends require an array of **MPI_Request** handles to access individual state.

MPI_Irecv

The prototype for the **MPI_Irecv** primitive is

```
int                     // error code if not 0
MPI_Irecv(
    // message description
    void *out,          // address of the receive buffer
    int count,          // number of items in the message
    MPI_Datatype type, // MPI type being received

    // source process
    int source,         // rank of the source process
    int tag,            // type of message,
    MPI_Comm comm,      // source's communicator context

    // polling information
    MPI_Request* req    // address of request object
)
```

MPI_Wait

A call to the **MPI_Wait()** function primitive blocks until the communication finishes:

```
int                     // error code if not 0
MPI_Wait(
    MPI_Request* req,   // address of the handle to the operation
    MPI_Status* status  // address of the object that holds comm information
)
```

MPI_Test

A call to the `MPI_Test()` function primitive returns the completion state of the operation. This primitive is the non-blocking version of `MPI_Wait()`. Its prototype is until the communication finishes:

```
int                     // error code if not 0
MPI_Test(
    MPI_Request* req,   // address of the handle to the operation
    int* flag,          // true is operation has completed
    MPI_Status* status  // address of the object that holds comm information
)
```

Alternative Modes of Non-Blocking Communication

The alternatives modes of communication are the same as for blocking

- buffered (`MPI_Bsend()`) - always locally blocking - MPI always copies the message to a local buffer - the user provides the local buffer
- synchronous (`MPI_Ssend()`) - returns only after the destination process has initiated receipt - this mode is called globally blocking
- ready (`MPI_Rsend()`) - the send operation succeeds only if the the destination process has initiated receipt of the message - returns an error code otherwise

MPMD EXAMPLE

Multiple Program Version

The following example uses one program to send messages and a separate program to receive them.

Sender

The sender transmits a message through a call to the `MPI_Send()` primitive:

```
// MPI MPMD
// mpi_sender.c
#include <stdio.h>
#include <string.h>
#include <mpi.h>

#define MAX_DATA 1000
#define MAX_MSG (MAX_DATA+20)

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

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    char msg[MAX_MSG + 1];
    char data[MAX_DATA + 1] = "Good Morning Sunshine!";
    sprintf_s(msg, MAX_MSG, "Process %d says %s", rank, data);
    int len = strlen(msg);
```

```
    MPI_Send(msg, len, MPI_CHAR, 0, 0, MPI_COMM_WORLD);
    MPI_Finalize();
    return 0;
}
```

Receiver

The receiver collects the message through separate calls to the **`MPI_Recv()`** primitive:

```
// MPI MPMD
// mpi_receiver.c
#include <stdio.h>
#include <mpi.h>

#define MAX_DATA 1000
#define MAX_MSG (MAX_DATA+20)

int main(int argc, char** argv) {
    MPI_Status status;
    int ip, np;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &np);
    char msg[MAX_MSG + 1];
    for (ip = 1; ip < np; ip++) {
        MPI_Recv(msg, MAX_MSG, MPI_CHAR, MPI_ANY_SOURCE, MPI_ANY_TAG,
            MPI_COMM_WORLD, &status);
        int nchar;
        MPI_Get_count(&status, MPI_CHAR, &nchar);
        msg[nchar] = '\0';
        printf("%s\n", msg);
    }
    printf("You bet!!\n");
    MPI_Finalize();
    return 0;
}
```

Executing an MPMD Solution - Linux

To compile and execute this MPMD solution on a Linux platform, enter the following commands

```
>mpicc receiver.c -o Receive
>mpicc sender.c -o Send
>mpirun -np 1 Receive : -np 3 Send
```

```
Process 3 says Good Morning Sunshine!
Process 1 says Good Morning Sunshine!
Process 2 says Good Morning Sunshine!
You bet!!
```

Executing an MPMD Solution - Visual Studio

To compile this MPMD solution on Visual Studio with the Microsoft compiler, enter the following Project settings for each Project (Sender and Receiver):

- add **`$(MSMPI_INC);$(MSMPI_INC)\x64`** to the Additional Include Directories under C/C++->General
- add **`$(MSMPI_LIB64)`** to the Additional Library Directories under Linker->General
- add **`msmpi.lib`** to the Additional Dependencies under Linker->Input

Compile each Project.

To run the two executables (**`Receiver.exe Send.exe`**) from the command line, open the command line prompt as an administrator, change to the directory that contains these executables, and enter the following command:

```
>"%MSMPI_BIN%"\mpiexec -n 1 Receiver : Send     Process 4 says Good Morning Sunshine!
                                                Process 5 says Good Morning Sunshine!
                                                Process 7 says Good Morning Sunshine!
                                                Process 6 says Good Morning Sunshine!
                                                Process 3 says Good Morning Sunshine!
                                                Process 1 says Good Morning Sunshine!
                                                Process 2 says Good Morning Sunshine!
                                                You bet!!
```

Executing an MPMD Solution - Intel Compiler

To compile this MPMD solution on Visual Studio with the Intel compiler, enter the following Project settings for each Project (Sender and Receiver):

- add **`$(I_MPI_ROOT)\intel64\include`** to the Additional Include Directories under C/C++->General
- add **`$(I_MPI_ROOT)\intel64\lib\release`** to the Additional Library Directories under Linker->General
- add **`impi.lib`** to the Additional Dependencies under Linker->Input

Compile each project.

The run the two (**`Receive.exe Send.exe`**) executables, open a command-line prompt as an administrator, and change the directory to that of the executables. Enter the following command

```
>mpiexec -n 1 Receive : -n 7 Send       Process 4 says Good Morning Sunshine!
                                        Process 5 says Good Morning Sunshine!
                                        Process 7 says Good Morning Sunshine!
                                        Process 6 says Good Morning Sunshine!
                                        Process 3 says Good Morning Sunshine!
                                        Process 1 says Good Morning Sunshine!
                                        Process 2 says Good Morning Sunshine!
                                        You bet!!
```

Single Program Version

The single program version of the above solution collects the code into one source file that distinguishes execution paths based on rank.

The process with rank 0 receives the messages, while all other processes transmit them:

```
// MPI SPMD
// mpi_sender_receiver.c
#include <stdio.h>
#include <string.h>
#include <mpi.h>

#define MAX_DATA 1000
#define MAX_MSG (MAX_DATA+20)

int main(int argc, char** argv) {
    MPI_Status status;
    int rank, ip, np;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &np);
    if (rank == 0) {
        char msg[MAX_MSG + 1];
        for (ip = 1; ip < np; ip++) {
            MPI_Recv(msg, MAX_MSG, MPI_CHAR, MPI_ANY_SOURCE, MPI_ANY_TAG,
                MPI_COMM_WORLD, &status);
```

```
            int nchar;
            MPI_Get_count(&status, MPI_CHAR, &nchar);
            msg[nchar] = '\0';
            printf("%s\n", msg);
        }
        printf("You bet!!\n");
    }
    else {
        char msg[MAX_MSG + 1];
        char data[MAX_DATA + 1] = "Good Morning Sunshine!";
        sprintf_s(msg, MAX_MSG, "Process %d says %s", rank, data);
        int len = strlen(msg);
        MPI_Send(msg, len, MPI_CHAR, 0, 0, MPI_COMM_WORLD);
    }
    MPI_Finalize();
    return 0;
}
```

Executing the Single Source Solution

To compile and execute this solution on a Linux platform, enter the following commands

```
>mpicc sender-receiver.c -o Send-Receive
>mpirun -np 4 Send-Receive
```

```
Process 3 says Good Morning Sunshine!
Process 1 says Good Morning Sunshine!
Process 2 says Good Morning Sunshine!
You bet!!
```

To execute this solution on a Visual Studio platform, enter the following at the command prompt

```
>mpiexec /np 8 Send-Receive
```

```
Process 4 says Good Morning Sunshine!
Process 5 says Good Morning Sunshine!
Process 7 says Good Morning Sunshine!
Process 6 says Good Morning Sunshine!
Process 3 says Good Morning Sunshine!
Process 1 says Good Morning Sunshine!
Process 2 says Good Morning Sunshine!
You bet!!
```

ERROR HANDLING

MPI provides simple error reporting and handling facilities. Calls to MPI primitive functions return an integer value. If the value is 0 (or MPI_SUCCESS), the function did not detect any error. Otherwise, the value is the error code, which identifies the type of error. A function that detects an error may abort the program.

Custom Handling and Reporting

MPI-2 lets us define our own error handler. To specify an error handler, call the following function

```
int                     // error code if not 0
MPI_Comm_set_errorhandler(
    MPI_Comm comm,      // communicator
    MPI_Errhandler err  // custom error handler
)
```

The predefined handlers include

- **MPI_ERRORS_ARE_FATAL** - abort on error (default)
- **MPI_ERRORS_RETURN** - return on error (do not abort)

Custom Handler

The prototype for a custom handler is:

```
void
error(
    MPI_Comm *comm, // communicator
    int *err,       // error code
    ...             // variadic
)
```

Registration

To register a custom handler call the following function:

```
int
MPI_Comm_create_errhandler(
    MPI_Comm_errhandler_fn *f, // points to handler function
    MPI_Errhandler *errhandler // points to error handler struct
)
```

Example

The following code snippet defines a custom handler, registers it, and notifies MPI to use it. The uninitialized communicator causes this handler to be called:

```
// custom handler
void err(MPI_Comm *comm, int *err, ...) {
    printf("Error code %d\n", *err);
}

int main(int argc, char** argv) {
    MPI_Comm context;

    MPI_Init(&argc, &argv);
    MPI_Errhandler error;
    MPI_Comm_create_errhandler(err, &error);
    MPI_Comm_set_errhandler(MPI_COMM_WORLD, error);
    // ...
    MPI_Send(..., context);  // uninitialized context calls err()
    // ...
    MPI_Finalize();

    return 0;
)
```

EXERCISES

- MPI Forum (2012). MPI: A Message-Passing Interface Standard, Version 3.0
- Barney, B. (2014). "Message Passing Interface (MPI)". Lawrence Livermore National Laboratory. Retrieved Jan 19 2015. Last Modified Nov 12 2014.
- Microsoft (2015). Microsoft MPI
- Barlas, G. (2015). Multicore and GPU Programming - An Integrated Approach. Morgan-Kaufmann. 978-0-12-417137-4. pp.239-261.