High-Performance Computing

Lecture 6 Introduction to MPI

CITS5507

Zeyi Wen

Computer Science and Software Engineering

School of Maths, Physics and Computing

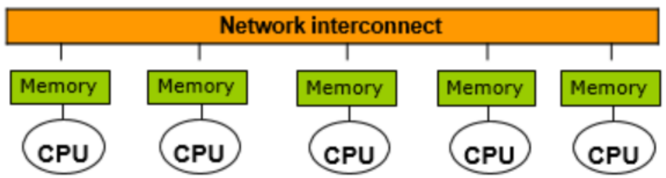
Acknowledgement: The lecture slides are adapted from many online sources.

Outline

- MPI Related Background
- MPI Basics
  - ✓ Hello World
  - ✓ Procedure Specification
  - ✓ Error Handling
  - ✓ Message Passing Model
  - ✓ Other Interesting Features
- Point-to-Point Communication
  - ✓ Sending and Receiving Routine
  - ✓ MPI Tag and Datatype
  - ✓ Blocking vs Non-blocking
  - ✓ Message Ordering
  - ✓ Extended Examples

Distributed Memory Machines

- Each processor has its own address space
- Communication between processes by explicit data exchange
  - Sockets (a term in computer network)
  - Message passing
  - Remote procedure call/remote method invocation



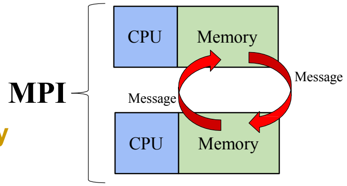
What is MPI

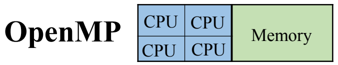
- **M**essage **P**assing **I**nterface
- All machines run the same code
- Messages are sent between them to guide computation

- MPI is a standard not a library itself
  - ➢ OpenMPI, MPICH are libraries/implementations
- MPI is portable
- MPI can work with heterogenous clusters
- MPI code can work on various configurations of machines

OpenMP and MPI

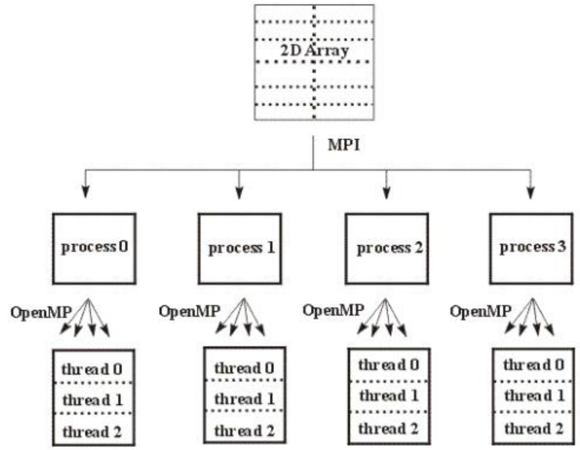
- **MPI – Designed for distributed memory**
  - Multiple systems
  - Send/receive messages
- **OpenMP – Designed for shared memory**
  - Single system with multiple cores
  - One thread/core sharing memory
- **C, C++, and Fortran**
- **There are other options**
  - Interpreted languages with multithreading
    - Python, R, matlab (have OpenMP & MPI underneath)
  - CUDA, OpenACC (GPUs)
  - Pthreads, Intel Cilk Plus (multithreading)
  - OpenCL, Chapel, Co-array Fortran, Unified Parallel C (UPC)





OpenMP and MPI

Example of "OpenMP vs MPI":



(Lecture 2) Process and Thread

- A process can be considered as an independent execution environment in a computer system.
- There are usually many processes in a system at any time, each with its own memory space.
- Each process executes a sequence of instructions (the machine language program).
- Threads are also independent execution environments, but with a shared memory space (or address space).

(Lecture 2) Process vs. Thread

- MPI = Process, OpenMP = Thread
- Program starts with a single process
- Processes have their own (private) memory space
- A process can create one or more threads
- Threads created by a process share its memory space
  - ✓ Read and write to same memory addresses
  - ✓ Share same process ids and file descriptors
- Each thread has a unique instruction counter and stack pointer
  - ✓ A thread can have private storage on the stack

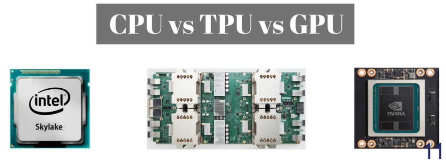
Classification of Parallel Architectures

Flynn's Taxonomy

- SISD: Single instruction single data
  - Classical von Neumann architecture
- SIMD: Single instruction multiple data
- MISD: Multiple instructions single data
  - Non existent, just listed for completeness
- MIMD: Multiple instructions multiple data
  - Most common and general parallel machine
  - Our focus in MPI and OpenMP

Current Trend in HPC

- **Single machine is getting faster and cheaper**
  - ✓ Graphics Processing Units (GPUs)
  - ✓ Multi/Many core CPUs
  - ✓ AI Accelerators (e.g. Google TPUs)
- **HPC cluster with multiple machines**
  - ✓ Enormous data sizes
  - ✓ On-demand HPC infrastructure
  - ✓ Much faster networking capabilities



Why is MPI?

What happens when you run out of compute power?

- Too much data
- Too many steps

Solution: Staple a number of computers together

- Also called 'building a super-computer'
- **MPI** allows you to do problems in parallel using message-passing to communicate between "computers", or more precisely, processes.

Outline

- MPI Related Background
- MPI Basics
  - ✓ Hello World
  - ✓ Procedure Specification
  - ✓ Error Handling
  - ✓ Message Passing Model
  - ✓ Other Interesting Features
- Point-to-Point Communication
  - ✓ Sending and Receiving Routine
  - ✓ MPI Tag and Datatype
  - ✓ Blocking vs Non-blocking
  - ✓ Message Ordering
  - ✓ Extended Examples

MPI Program Basics

Include MPI Header File

Start of Program

(Non-interacting Code)

Initialize MPI

Run Parallel Code & Pass Messages

End MPI Environment

(Non-interacting Code)

End of Program

```
#include <mpi.h>

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

.
.     // Run parallel code
.

MPI_Finalize(); // End MPI Envir

return 0;
}
```


Hello World

```
#include "mpi.h"
#include <stdio.h>
int main(int argc, char *argv[])
{
    MPI_Init(&argc, &argv);
    printf("Hello world\n");
    MPI_Finalize();
    return 0;
}
```

All MPI programs need:

- MPI_Init
- MPI_Finalize()

Hello World (Continued)

- **MPI_INIT**
  This routine must be the first MPI routine you call (it does not have to be the first statement). It sets things up and might do a lot of behind-the-scenes work on some cluster-type systems (like start daemons and such).

- **MPI_FINALIZE**
  This is the companion to MPI_Init. It must be the last MPI call. It may do a lot of housekeeping, or it may not.

Hello World (Continued)

Compiling MPI Programs

```
$ mpicc -o helloWorld helloworld.c
```

- mpicc → calls the compilers
- Then standard flags as usual
  - -o
  - -Wall
  - -O1 -O2 -O3 (numerical optimisation)

Hello World (Continued)

THE UNIVERSITY OF WESTERN AUSTRALIA

Running MPI Programs

```
//1
$ mpiexec -n 4 helloWorld
//2
$ mpirun -np 4 helloWorld
```

- Both work, but mpiexec is generally preferable because it is standardised
- "np" or "n" is the number of processes. In this case np = 4, so there will be four MPI processes run.
- will run multiple processes on one machine

MPI Hello World

THE UNIVERSITY OF WESTERN AUSTRALIA

```
#include "mpi.h"
#include <stdio.h>
int main(int argc, char *argv[])
{
    MPI_Init(&argc, &argv);
    printf("Hello world\n");
    MPI_Finalize();
    return 0;
}
```

All MPI programs need:
- MPI_Init
- MPI_Finalize()

Hello World (Continued)

THE UNIVERSITY OF WESTERN AUSTRALIA

Complex Version:

```
#include "mpi.h"
#include <stdio.h>
int main(int argc, char *argv[])
{
int rank, size;
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);
printf("Hello from %d of %d processes.\n", rank, size);
MPI_Finalize();
return 0;
}
```

- Communicators
  - ✓ Rank
  - ✓ Size

Hello World (Continued)

THE UNIVERSITY OF WESTERN AUSTRALIA

```
int MPI_Comm_rank(MPI_Comm comm, int *rank)
```

Returns the rank of the calling process in that communicator.
- **comm** is the "communicator" and can be found in many of the MPI routines.

```
int MPI_COMM_SIZE(MPI_Comm comm, int *size)
```

Returns the number of processes in the communicator.

Communicators

```
int MPI_Comm_rank(MPI_Comm comm, int *rank)
```

- **comm** is the "communicator" and can be found in many of the MPI routines.
- One can divide up the processes into subsets for various algorithmic purposes using "communicator".
  - ✓ If we had a matrix - distributed across the processes – for which we wished to find the determinant, we could define a subset of the processes that holds a certain column of the matrix so that we could read that column conveniently.
  - ✓ One may define a communicator for just the odd processes.

Communicators (Continued)

- Processes exist as part of a communicator
  - ✓ Communicator is a group of processes
- All processes are part of the MPI_COMM_WORLD communicator
  - ✓ MPI_COMM_WORLD is all the processes
- Rank – The 'id' of this process in that communicator
- Size – The number of processes in that communicator

Often important for processes to work out what job they should do. Rank 0 is often the 'root' process.

```
int MPI_Comm_rank(MPI_Comm comm, int *rank)
```

```
int MPI_COMM_SIZE(MPI_Comm comm, int *size)
```


Outline

- MPI Related Background
- MPI Basics
  - ✓ Hello World
  - ✓ Procedure Specification
  - ✓ Error Handling
  - ✓ Message Passing Model
  - ✓ Other Interesting Features
- Point-to-Point Communication
  - ✓ Sending and Receiving Routine
  - ✓ MPI Tag and Datatype
  - ✓ Blocking vs Non-blocking
  - ✓ Message Ordering
  - ✓ Extended Examples

Procedure Specification

MPI procedures are specified using a language-independent notation. The arguments of procedure calls are marked as IN, OUT or INOUT. The meanings of these are:

- IN – Used but not updated
  (e.g. comm in `MPI_Comm_rank`)
- OUT – May be updated
  (e.g. rank in `MPI_Comm_rank`)
- INOUT - Both used and updated
  (less common but very important)

Note: procedure = function in C

```
int MPI_Comm_rank(MPI_Comm comm, int *rank)
```

Procedure Specification

- A common occurrence for MPI functions is an argument that is used as IN by some processes and OUT by other processes. Such an argument is, syntactically, an INOUT argument and is marked as such, although, semantically, it is not used in one call both for input and for output on a single process.
- Another frequent situation arises when an argument value is needed only by a subset of the processes. When an argument is not significant at a process, then an arbitrary value can be passed as an argument.

Procedure Specification (Example)

An argument of type OUT or INOUT cannot be aliased with any other argument passed to an MPI procedure.

- An example of argument aliasing in C appears below:

```
void copyIntBuffer(int *pin, int *pout, int len)
{
        int i;
        for (i=0; i<len; ++i)
                *pout++ = *pin++;
}
```

A call to it in the following code fragment has aliased arguments.

```
int a[10];
copyIntBuffer( a, a+3, 7);
```

Although the C language allows this, such usage of MPI procedures is forbidden unless otherwise specified.

Outline

- MPI Basics
  - ✓ Hello world (continued)
  - ✓ Procedure Specification
  - ✓ **Error handling**
  - ✓ Message Passing Model
  - ✓ Other Interesting Features
- Point-to-Point Communication
  - ✓ Sending and Receiving Routine
  - ✓ MPI Tag and Datatype
  - ✓ Blocking vs Non-blocking
  - ✓ Message Ordering
  - ✓ Extended Examples

Error Handling

The most common error code is MPI_SUCCESS

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

int main(int argc, char *argv[])
{
  int rank, size;
  int status;
  MPI_Init(&argc, &argv);
  status = MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  error_handle(status);
  status =MPI_Comm_size(MPI_COMM_WORLD, &size);
  error_handle(status);
  printf("I am process %d of %d\n", rank, size);
  MPI_Finalize();
  return 0;
}
```

We can write our own error handlers:

```
void error_handle(int status)
{
   switch(status)
   {
     case MPI_SUCCESS: break;
     case 1:
       //printf("help\n");
       break;
     default:
 //printf("call ghost-busters\n");
       break;
   }
}
```

**Also, the meaning of an error code (status) can be extracted by calling function:* MPI_Error_string.

Error Handler

- The predefined default error handler, which is called MPI_ERRORS_ARE_FATAL, for a newly created communicator or for MPI_COMM_WORLD is to abort the whole parallel program as soon as any MPI error is detected.

- There is another predefined error handler, which is called MPI_ERRORS_RETURN. The default error handler can be replaced with this one by calling function MPI_Errhandler_set.

```
/*---------------------------------------------------------------------------*/
/* MPI_Errhandler                                                            */
/*---------------------------------------------------------------------------*/

typedef int MPI_Errhandler;
#define MPI_ERRHANDLER_NULL  ((MPI_Errhandler)0x14000000)

#define MPI_ERRORS_ARE_FATAL ((MPI_Errhandler)0x54000000)
#define MPI_ERRORS_RETURN    ((MPI_Errhandler)0x54000001)
```


Error Handler (Continued)

- The predefined default error handler, which is called MPI_ERRORS_ARE_FATAL, for a newly created communicator or for MPI_COMM_WORLD is to abort the whole parallel program as soon as any MPI error is detected.

- There is another predefined error handler, which is called MPI_ERRORS_RETURN. The default error handler can be replaced with this one by calling function MPI_Errhandler_set, for example:

```
MPI_Errhandler_set(MPI_COMM_WORLD, MPI_ERRORS_RETURN);
```

Once you've done this in your MPI code, the program will not longer abort on having detected an MPI error, instead the error will be returned and you will have to handle it.

MPI Error Class

- To make it possible for an application to interpret an error code (more than 50 error codes), the routine MPI_ERROR_CLASS converts any error code into one of a small set of standard error codes, called *error classes*. Valid error classes include

```
/*---------------------------------------------------------------------------*/
/* MPI ERROR CLASS                                                           */
/*---------------------------------------------------------------------------*/

#define MPI_SUCCESS          0      /* Successful return code */

#define MPI_ERR_BUFFER       1      /* Invalid buffer pointer */
#define MPI_ERR_COUNT        2      /* Invalid count argument */
#define MPI_ERR_TYPE         3      /* Invalid datatype argument */
#define MPI_ERR_TAG          4      /* Invalid tag argument */
#define MPI_ERR_COMM         5      /* Invalid communicator */
#define MPI_ERR_RANK         6      /* Invalid rank */
#define MPI_ERR_ROOT         7      /* Invalid root */
#define MPI_ERR_GROUP        8      /* Invalid group */
#define MPI_ERR_OP           9      /* Invalid operation */
#define MPI_ERR_TOPOLOGY    10      /* Invalid topology */
#define MPI_ERR_DIMS        11      /* Invalid dimension argument */
#define MPI_ERR_ARG         12      /* Invalid argument */
#define MPI_ERR_UNKNOWN     13      /* Unknown error */
#define MPI_ERR_TRUNCATE    14      /* Message truncated on receive */
#define MPI_ERR_OTHER       15      /* Other error; use Error_string */
#define MPI_ERR_INTERN      16      /* Internal error code */
#define MPI_ERR_IN_STATUS   17      /* Error code is in status */
#define MPI_ERR_PENDING     18      /* Pending request */
#define MPI_ERR_REQUEST     19      /* Invalid request (handle) */
#define MPI_ERR_ACCESS      20      /* Premission denied */
#define MPI_ERR_AMODE       21      /* Error related to amode passed to MPI_File_open
#define MPI_ERR_BAD_FILE    22      /* Invalid file name (e.g., path name too long) */
```


Error Handling Example

Example:

```
MPI_Errhandler_set(MPI_COMM_WORLD, MPI_ERRORS_RETURN);
error_code = MPI_Send(send_buffer, strlen(send_buffer) + 1,
MPI_CHAR, addressee, tag, MPI_COMM_WORLD);
if (error_code != MPI_SUCCESS)
{
    char error_string[BUFSIZ];
    int length_of_error_string;
    MPI_Error_string(error_code, error_string, &length_of_error_string);
    fprintf(stderr, "%3d: %s\n", my_rank, error_string);
    send_error = TRUE;
}
```

- the function MPI_ERROR_STRING can be used to compute the error string associated with an error class.

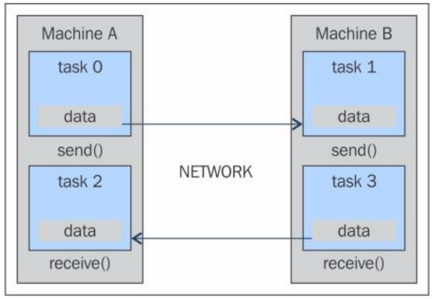
Outline

- MPI Related Background
- MPI Basics
  - ✓ Hello world
  - ✓ Procedure Specification
  - ✓ Error handling
  - ✓ Message Passing Model
  - ✓ Other Interesting Features
- Point-to-Point Communication
  - ✓ Sending and Receiving Routine
  - ✓ MPI Tag and Datatype
  - ✓ Blocking vs Non-blocking
  - ✓ Message Ordering
  - ✓ Extended Examples

Message Passing Model

- **Simple goals**
  - ✓ Portability
  - ✓ Efficiency
  - ✓ Functionality
- **So we want a message passing model**
  - ✓ Each process has separate address space
  - ✓ A message is one process copying some of its address space to another
    - Send
    - Receive

Message Passing Model



*Remember that a **distributed-memory computer** is effectively a collection of separate machines, each called a **node**, connected by some network cables. It is not possible for one node to directly read or write to the memory of another node, so there is no concept of **shared memory**, but **sender** and **receiver**.*

Minimal MPI- Sender and Receiver

- What does the sender send?
  - **Data** starting address + length (in bytes)
  - **Destination** destination address (an int is enough)
- What does the receiver receive?
  - **Data** starting address + length (in bytes)
  - **Source** source address (filled when received)

Message = **data** + **envelope**

$$\text{MPI\_Send}\ (\underbrace{\text{startbuf, count, datatype}}_{\text{DATA}},\ \underbrace{\text{dest, tag, comm}}_{\text{ENVELOPE}})$$

Minimal MPI - Tag

- So we can send and receive messages
- Might be enough for some applications but there's something missing

Message selection

- Currently all processes receive all messages
- If we add a tag field, processes is able to ignore messages not intended for them

Minimal MPI - Basic Model

- Our model now becomes
  - ✓ Send this information: address, length, destination, tag
  - ✓ Receive this info: address, length, source, tag, actual length
- We can make the source and tag arguments wildcards to go back to our original model
- This is a complete model for Message-Passing
- Most MPI functions are built by combining these two

Minimal MPI – Problems

There are still some issues that MPI solves

1. Describing message buffers
2. Separating families of messages
3. Naming processes
4. Communicators

MPI – Describing Buffers

(address, length) is not sufficient for two main reasons

- Assumes data is contiguous
  - ✓ Often not the case
  - ✓ E.g. sending the row of a matrix stored column-wise
- Assumes data representation is always known
  - ✓ Does not handle heterogenous clusters
  - ✓ E.g. CPU + GPU machines for example
- **MPI's solution**
  - ✓ MPI_datatypes → Abstract one layout up → Allow users to specify their own
  - ✓ (address, length, datatype)

MPI – Separating Families of Messages

- Consider using a 3rd party library written with MPI
  - ✓ They can have their own naming of tags, etc.
  - ✓ Your code may interact with the library.
- **MPI's solution**
  - ✓ Exploit contexts → Think if this as super-tags
  - ✓ Provides one more layer of separation between codes running in one application

MPI – Naming Processes

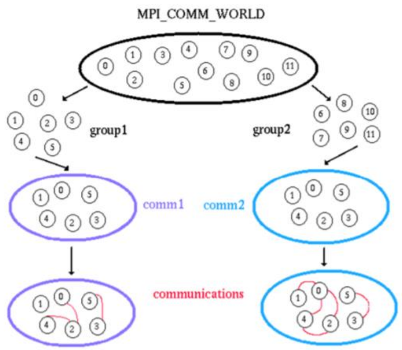
- Processes belong to **groups**
- **A rank** is associates with **each group**
- Using an int is actually sufficient in this case

```
int MPI_Comm_rank(MPI_Comm comm, int *rank)
```


MPI – Communicators

```
int MPI_Comm_rank(MPI_Comm comm, int *rank)
```

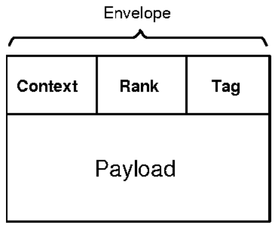
- Combines contexts and groups into a single structure
- Destination and source ranks are specified relative to a communicator



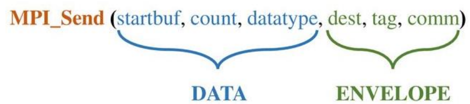
Sender

```
MPI_Send(start, count, datatype, dest, tag, comm)
```

- Message buffer described by
  - Start
  - Count
  - Data types
- Target process given by
  - Dest
  - Comm
- Tag can be used to create different 'types' of messages



Format of MPI Message



Receiver

```
MPI_Recv(start, count, datatype, source, tag, comm, status)
```

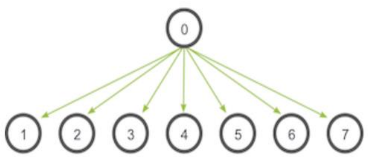
- Waits until a matching (source, tag) message is available
- Reads into the buffer
  - Start
  - Count
  - Datatype
- Target process specified by
  - Source
  - Comm
- Status contains more information
- Receiving fewer than count occurrences of datatype is okay, more is an error

Outline

- MPI Related Background
- MPI Basics
  - ✓ Hello world
  - ✓ Procedure Specification
  - ✓ Error handling
  - ✓ Message Passing Model
  - ✓ **Other Interesting Features**
- Point-to-Point Communication
  - ✓ Sending and Receiving Routine
  - ✓ MPI Tag and Datatype
  - ✓ Blocking vs Non-blocking
  - ✓ Message Ordering
  - ✓ Extended Examples

MPI – Other Interesting Features

- **Collective communication**
  - ✓ Get all of your friends involved – light up the group chat
  - ✓ Two flavours
    - Data movement – E.g. broadcast
    - Collective computation – min, max, average, logical OR, etc.



MPI – Other Interesting Features

- Virtual topologies
  - ✓ Allow graphs and grid connections to be imposed on processes
  - ✓ 'Send to my neighbours'
- Debugging and profiling
  - ✓ race conditions, deadlocks,
  - ✓ workload balancing, costs of communications
- Communication modes
  - ✓ Blocking vs. Non-blocking
- Support for Libraries
  - ✓ Communicators allow libraries to exist in their own space
- Support for heterogenous networks
  - ✓ MPI_Send/Recv implementation independent

MPI – Other Interesting Features

- **Processes vs. Processors**
  - ✓ **A process** is a software concept
  - ✓ **A processor** or CPU, is a circuit board inside a computer that executes instructions on behalf of programs.
  - ✓ Some implementations limit one process per processor

Outline

- MPI Related Background
- MPI Basics
  - ✓ Hello world
  - ✓ Procedure Specification
  - ✓ Error handling
  - ✓ Message Passing Model
  - ✓ Other Interesting Features
- **Point-to-Point Communication**
  - ✓ Sending and Receiving Routine
  - ✓ MPI Tag and Datatype
  - ✓ Blocking vs Non-blocking
  - ✓ Message Ordering
  - ✓ Extended Examples

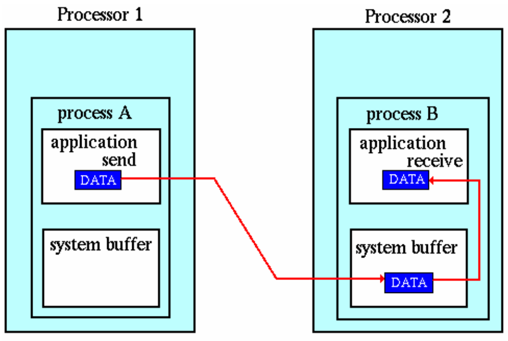
Point-to-Point Communication

Why need point-to-point communication?

- The fundamental mechanism in MPI is the **transmission of data** between a pair of processes
  - ✓ **One** sender
  - ✓ **One** receiver
- Almost all other MPI constructs are short-hand versions of tasks you could achieve with point-to-point methods
- We will learn more on this topic in the rest of the unit.
  - ✓ Many idiosyncrasies in MPI come from how point-to-point communication is achieved in-code

Point-to-Point Communication

- Remember that
  - ✓ Rank → ID of each process in a communicator
  - ✓ Communicator → Collection of processes
  - ✓ MPI_COMM_WORLD → The communicator for all processes



Path of a message buffered at the receiving process

Sending and Receiving Routines (Review)

Usage:

MPI_Send(**start**, **count**, **datatype**, **dest**, **tag**, **comm**)

- Message buffer described by
  - **Start**
  - **Count**
  - **Data types**
- Target process given by
  - **Dest**
  - **Comm**
- **Tag** can be used to create different 'types' of messages

Sending and Receiving Routines (Review)

Usage:

MPI_Recv(start, count, datatype, source, tag, comm, status)

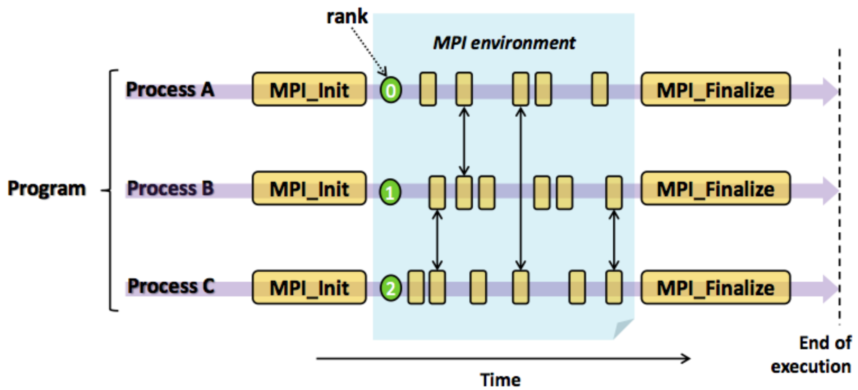
- Waits until a matching (source, tag) message is available
- Reads into the buffer
  - Start
  - Count
  - Datatype
- Target process specified by
  - Source
  - Comm
- Status contains more information
- Receiving fewer than count occurrences of datatype is okay, more is an error

Example1: Knock-Knock

```
#include "mpi.h"
#include <stdio.h>
#include <string.h>
int main(int argc, char* argv[]){
 MPI_Init(&argc, &argv);
 char msg[20];
 int myrank, tag = 99;
 MPI_Status status;
 MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
 if(myrank == 0){
   strcpy_s(msg, "Knock knock");
   MPI_Send(msg, strlen(msg)+1, MPI_CHAR, 1, tag, MPI_COMM_WORLD);
   printf("I am process %d sending \"%s\" to process 1. Over!\n", myrank, msg);
  }else if (myrank == 1){
   MPI_Recv(msg, 20, MPI_CHAR, 0, tag,   MPI_COMM_WORLD, &status);
   printf("I am process %d receiving \"%s\". Over!\n", myrank, msg);
  }
  MPI_Finalize();
  return 0;
}
```

- This code sends a single string from process **0** to process **1**

MPI Execution Model



Example1: Knock-Knock (Review)

```
#include "mpi.h"
#include <stdio.h>
#include <string.h>
int main(int argc, char* argv[]){
 MPI_Init(&argc, &argv);
 char msg[20];
 int myrank, tag = 99;
 MPI_Status status;
 MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
 if(myrank == 0){
   strcpy_s(msg, "Knock knock");
   MPI_Send(msg, strlen(msg)+1, MPI_CHAR, 1, tag, MPI_COMM_WORLD);
   printf("I am process %d sending \"%s\" to process 1. Over!\n", myrank, msg);
  }else if (myrank == 1){
   MPI_Recv(msg, 20, MPI_CHAR, 0, tag,   MPI_COMM_WORLD, &status);
   printf("I am process %d receiving \"%s\". Over!\n", myrank, msg);
  }
  MPI_Finalize();
  return 0;
}
```

- This code sends a single string from process **0** to process **1**

Example1: Knock-Knock

MPI_Send:

- **msg**: The **buffer** (location in memory) to send from
- **strlen(msg)+1**
  - ✓ The **number of items** to send
  - ✓ + 1 to include the null-byte '\0' → Only relevant when sending strings
- **MPI_CHAR**
  - The **MPI datatype** (more on this later) indicates the size of each element in your buffer

```
if(myrank == 0){
  strcpy(msg, "Knock knock");
  MPI_Send(msg, strlen(msg)+1, MPI_CHAR, 1, tag, MPI_COMM_WORLD);
}
```


Example1: Knock-Knock

MPI_Send (Continued):

- **1:** The **rank** of the destination process
- **Tag**
  - ✓ The '**topic**' of the message (will only be received if process 1 Recv's on tag 99)
- **MPI_COMM_WORLD**
  - ✓ The **communicator** on which we are sending through
  - ✓ Each communicator (with two processes) has a rank 0 process and a rank 1 process

```
if(myrank == 0){
  strcpy(msg, "Knock knock");
  MPI_Send(msg, strlen(msg)+1, MPI_CHAR, 1, tag, MPI_COMM_WORLD);
}
```


Example1: Knock-Knock

MPI_Recv:

- **msg:** The **buffer** to receive from
- **20:** The **maximum** number of elements we want
- **MPI_CHAR:** The **size** of each **element**
- **0:** The **process** we want to **receive** from
- **Tag:** The 'topic' we want to receive on(more on later)
- **MPI_COMM_WORLD**
  - ✓ The **communicator** we are communicating on
- **&status:** The error code info in this case passed to the function (since it returns how many elements was received)

```
else if (myrank == 1){
  MPI_Recv(msg, 20, MPI_CHAR, 0, tag,   MPI_COMM_WORLD, &status);
}
```

Outline

- MPI Related Background
- MPI Basics
  - ✓ Hello world
  - ✓ Procedure Specification
  - ✓ Error handling
  - ✓ Message Passing Model
  - ✓ Other Interesting Features
- Point-to-Point Communication
  - ✓ Sending and Receiving Routine
  - ✓ **MPI Tag and Datatype**
  - ✓ Blocking vs Non-blocking
  - ✓ Message Ordering
  - ✓ Extended Examples

MPI Tags

- A rather good idea at the time
- Rarely used in practice
- Allow processes to provide '**topics**' for communication
  - E.g. '42' refers to all communication for a particular sub-task etc.
- **MPI_ANY_TAG** renders specifying tags useless

MPI Datatype

- MPI defines its **own** data type that correspond to typical datatypes in C or Fortran
- This allows to code to be **portable** between systems
- Users are allowed to build their own datatypes in MPI
- Since all data is given an MPI type, an MPI implementation can communicate between **very different machines**
- Specifying **application-oriented** data layout
  - ✓ Reduces memory-to-memory copies in implementation
- Allows the use of special hardware where available

MPI Datatype vs C Datatype

MPI Datatype	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	n/a
MPI_PACKED	n/a

MPI Datatype – BYTE and PACKED

MPI_BYTE / MPI_PACKED:

- **MPI_BYTE** is precisely a byte (eight bits)
  - ✓ Un-interpreted and may be different to a character
  - ✓ Some machines may use two bytes for a character for instance
- MPI_PACKED is a much more complicated
  - ✓ Used to send structs(noncontiguous) through MPI
  - ✓ The user explicitly packs data into a contiguous buffer before sending it and unpacks it from a contiguous buffer after receiving it.

MPI Datatype - Pack

```
MPI_PACK(inbuf, incount, datatype, outbuf, outsize, position, comm)
```

IN	inbuf	input buffer start (choice)
IN	incount	number of input data items (non-negative integer)
IN	datatype	datatype of each input data item (handle)
OUT	outbuf	output buffer start (choice)
IN	outsize	output buffer size, in bytes (non-negative integer)
INOUT	position	current position in buffer, in bytes (integer)
IN	comm	communicator for packed message (handle)

- Used by repeatedly calling MPI_PACK with changed inbuf and outbuf values

MPI Datatype - Unpack

```
MPI_UNPACK(inbuf, insize, position, outbuf, outcount, datatype, comm)
```

IN	inbuf	Input buffer start (choice)
IN	insize	size of input buffer, in bytes (non-negative integer)
INOUT	position	current position in bytes (integer)
OUT	outbuf	output buffer start (choice)
IN	outcount	number of items to be unpacked (integer)
IN	datatype	datatype of each output data item (handle)
IN	comm	communicator for packed message (handle)

The exact inverse of MPI_PACK. Used by repeatedly calling unpack, extracting each subsequent element

Example – Pack/Unpack

```
#include "mpi.h"
#include <stdio.h>
int main(int argc, char *argv[]) {
   int rank, size, i, position = 0;
//Position needs to be assigned a non-negative number, otherwise an error will be reported
   char c[100], buffer[110];// The buffer size is 110 bytes
   MPI_Init(&argc, &argv);
   MPI_Comm_size(MPI_COMM_WORLD, &size);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   if (rank == 0) {
     //Process 0 packs an int variable and a char array into the same memory
     //and sends it to process 1
     for (i = 0; i < 100; i++)
           c[i] = i;
     i = 2020;
     MPI_Pack(&i, 1, MPI_INT, buffer, 110, &position, MPI_COMM_WORLD);
     // Specify size (in bytes) when packing and unpacking buffers
     MPI_Pack(c, 100, MPI_CHAR, buffer, 110, &position, MPI_COMM_WORLD);
     MPI_Send(buffer, position, MPI_PACKED, 1, 0, MPI_COMM_WORLD);
   }
   if (rank == 1) {
      MPI_Recv(buffer, 110, MPI_PACKED, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
      MPI_Unpack(buffer, 110, &position, &i, 1, MPI_INT, MPI_COMM_WORLD);
      MPI_Unpack(buffer, 110, &position, c, 100, MPI_CHAR, MPI_COMM_WORLD);
      printf("i=%d, c[0] = %d, c[99] = %d\n", i, (int)c[0], (int)c[99]);
      fflush(stdout);
   }
   MPI_Finalize();
   return 0;
}
```

Outline

- MPI Related Background
- MPI Basics
  - ✓ Hello world
  - ✓ Procedure Specification
  - ✓ Error handling
  - ✓ Message Passing Model
  - ✓ Other Interesting Features
- Point-to-Point Communication
  - ✓ Sending and Receiving Routine
  - ✓ MPI Tag and Datatype
  - ✓ **Blocking vs Non-blocking**
  - ✓ Message Ordering
  - ✓ Extended Examples

Important Notes

- Communication requires **cooperation**; You need to know:
  - ✓ **Who** you are sending/receiving from/to
  - ✓ **What** you are sending/receiving
  - ✓ **When** you want to send/receive
  - ✓ Very specific, requires careful reasoning about algorithms
- All nodes (in general) will run the **same executable**
  - ✓ Very **different style** of programming
  - ✓ The '**root**' (usually rank **0**) may have very different tasks to all other nodes
  - ✓ Rank becomes very important to dividing the bounds of a problem

Example 2: Knock-knock, who's there

```
char msg[20];
int myrank, tag = 99;
MPI_Status status;
...
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
if(myrank == 0){
  strcpy_s(msg, "Knock knock");
  MPI_Send(msg, strlen(msg)+1, MPI_CHAR, 1, tag, MPI_COMM_WORLD);//1
  MPI_Recv(msg, 20, MPI_CHAR, 1, tag, MPI_COMM_WORLD, &status);//2
}
else if (myrank == 1){
  strcpy_s(msg, "Who's there?");
  MPI_Send(msg, strlen(msg)+1, MPI_CHAR, 0, tag, MPI_COMM_WORLD);//3
  MPI_Recv(msg, 20, MPI_CHAR, 0, tag, MPI_COMM_WORLD, &status);//4
}
```

May have a problem between 1, 3 and 2, 4.

Blocking vs Non-blocking

- Depending on the implementation you use this may cause a **deadlock**
  - ✓ If you have enough buffer space it might be okay (but don't rely on this)
- We have been using the blocking send/receive functions
  - ✓ Halt execution **until completed**
- There exist non-blocking versions of send/recv
  - ✓ **MPI_Isend** – Same arguments
  - ✓ **MPI_Irecv** – Same arguments but replace MPI_Status with MPI_Request
- Return **immediately** and continue with computation

When to use Non-blocking

- Should only be used where performance **improves**
  - ✓ E.g. sending a large amount of data when a large amount of compute is also available
  - ✓ Using non-blocking communication will parallelise a little more
- To check for a communication's success, need to use
  - ✓ **MPI_Wait()**
  - ✓ **MPI_Test()**
- An alternate interpretation
  - ✓ MPI_Send/Recv is just MPI_Isend/Irecv + MPI_WAIT()

Example 3: Knock-Who's Knock-There

Non-blocking Code:

```
if(rank == 0){
  strcpy(msg, "Knock knock");
  MPI_Irecv(msg2, 20, MPI_CHAR, 1, MPI_ANY_TAG, MPI_COMM_WORLD, &request);
  MPI_Send(msg, strlen(msg)+1, MPI_CHAR, 1, MPI_ANY_TAG, MPI_COMM_WORLD);
  MPI_Wait(&request, &status);
}
else if (rank == 1){
  strcpy(msg2, "Who's there?");
  MPI_Irecv(msg, 20, MPI_CHAR, 0, MPI_ANY_TAG, MPI_COMM_WORLD, &request);
  MPI_Send(msg2, strlen(msg)+1, MPI_CHAR, 0, MPI_ANY_TAG, MPI_COMM_WORLD);
  MPI_Wait(&request, &status);
}
MPI_Finalize();
return 0;
```


Write Safe Code

- A safe MPI program should not rely on **system buffering** for success.
- Any system will eventually run out of buffer space as message sizes are increased.
- User should design proper send/receive orders to avoid **deadlock**

Safe Code

```
#include <stdio.h>
#include "mpi.h"
/* process 0 send a number to and receive a number from process 1.
   process 1 receive a number from and send a number to process 0
*/
int main(int argc, char** argv)
{
  int my_rank, numbertoreceive, numbertosend = -16;
  MPI_Status status;
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
  if (my_rank==0){
    MPI_Send( &numbertosend, 1, MPI_INT, 1, 10, MPI_COMM_WORLD);
    MPI_Recv( &numbertoreceive, 1, MPI_INT, 1, 20, MPI_COMM_WORLD,
&status);
  }
  else if(my_rank == 1)
  {
    MPI_Recv( &numbertoreceive, 1, MPI_INT, 0, 10, MPI_COMM_WORLD,
&status);
    MPI_Send( &numbertosend, 1, MPI_INT, 0, 20, MPI_COMM_WORLD);
  }
  MPI_Finalize();
  return 0;
}
```

Deadlock Code

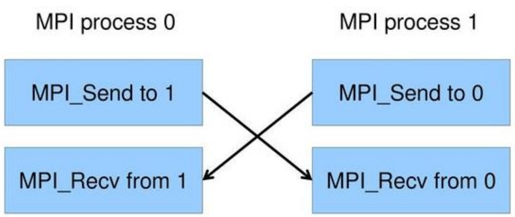
```c
#include <stdio.h>
#include "mpi.h"
/* process 0 receive a number from and send a number from process 1.
   process 1 receive a number from and send a number to process 0
*/
int main(int argc, char** argv)
{
  int my_rank, numbertoreceive, numbertosend = -16;
  MPI_Status status;
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
  if (my_rank==0){
    MPI_Recv( &numbertoreceive, 1, MPI_INT, 1, 20, MPI_COMM_WORLD, &status);
    MPI_Send( &numbertosend, 1, MPI_INT, 1, 10, MPI_COMM_WORLD);
  }
  else if(my_rank == 1)
  {
    MPI_Recv( &numbertoreceive, 1, MPI_INT, 0, 10, MPI_COMM_WORLD, &status);
    MPI_Send( &numbertosend, 1, MPI_INT, 0, 20, MPI_COMM_WORLD);
  }
  MPI_Finalize();
  return 0;
}
```


Buffering Dependent Code

```c
#include <stdio.h>
#include "mpi.h"
/* process 0 receive a number from and send a number from process 1.
   process 1 receive a number from and send a number to process 0
*/
int main(int argc, char** argv)
{
  int my_rank, numbertoreceive, numbertosend = -16;
  MPI_Status status;
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
  if (my_rank==0){
    MPI_Send( &numbertosend, 1, MPI_INT, 1, 10, MPI_COMM_WORLD);
    MPI_Recv( &numbertoreceive, 1, MPI_INT, 1, 20, MPI_COMM_WORLD, &status);
  }
  else if(my_rank == 1){
    MPI_Send( &numbertosend, 1, MPI_INT, 0, 20, MPI_COMM_WORLD);
    MPI_Recv( &numbertoreceive, 1, MPI_INT, 0, 10, MPI_COMM_WORLD, &status);
  }
  MPI_Finalize();
  return 0;
}
```

- Success of this code is dependent on **buffering**. One of the send must buffer and return. Otherwise, **deadlock** occurs.

Possible Deadlock

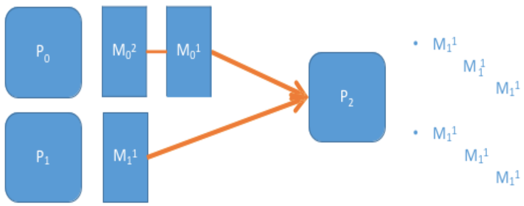


Outline

- MPI Related Background
- MPI Basics
  - ✓ Hello world
  - ✓ Procedure Specification
  - ✓ Error handling
  - ✓ Message Passing Model
  - ✓ Other Interesting Features
- Point-to-Point Communication
  - ✓ Sending and Receiving Routine
  - ✓ MPI Tag and Datatype
  - ✓ Blocking vs Non-blocking
  - ✓ **Message Ordering**
  - ✓ Extended Examples

Message Ordering

- Messages are **non-overtaking**
- The order a process sends messages is the order another process receives them
- The order multiple processes send messages in does not matter



Can be received at P2 as:

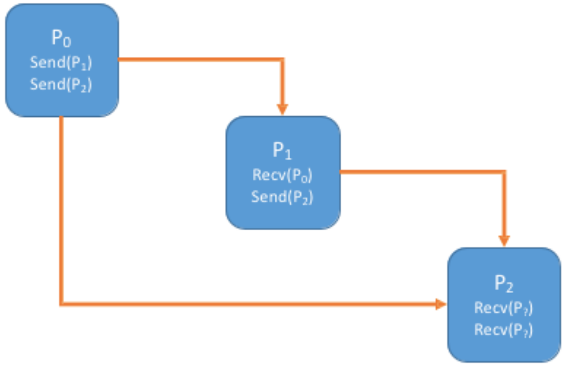
- $M1^1$,$M0^1$,$M0^2$
- $M0^1$, $M1^1$, $M0^2$
- $M0^1$, $M0^2$, $M1^1$

But not:

- $M1^1$,$M0^2$,$M0^1$
- $M0^2$, $M1^1$, $M0^1$
- $M0^2$, $M0^1$, $M1^1$

Message Ordering

- Another important note: Ordering is not transitive
  - Sounds goofy, but easy to make this mistake
  - Be careful when using **MPI_ANY_SOURCE**



Message Ordering

- One goal of MPI is to encourage **deterministic communication** patterns
- Using exact addresses, exact buffer sizes, enforced ordering etc.
- Makes code predictable
- Sources of non-determinism
  - MPI_ANY_SOURCE as source argument
  - MPI_CANCEL()
  - MPI_WAITANY()
  - Threading

Outline

- MPI Related Background
- MPI Basics
  - ✓ Hello world
  - ✓ Procedure Specification
  - ✓ Error handling
  - ✓ Message Passing Model
  - ✓ Other Interesting Features
- Point-to-Point Communication
  - ✓ Sending and Receiving Routine
  - ✓ MPI Tag and Datatype
  - ✓ Blocking vs Non-blocking
  - ✓ Message Ordering
  - ✓ **Extended Examples**

Extended Example1 : Computing Pi

Method:

- Divide [0,1] by some value n
- Each forms a rectangle of height $f(n)$ and width 1/n
- Add up all the rectangles to get an approximation of the integration
- This gives us an approximation to pi

Extended Example1 : Computing Pi

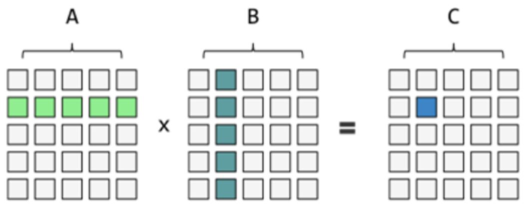
Parallel Strategy

- One process (the **root**, **rank 0**) obtains n from the user and **broadcast** this value to all others
- All other processes determine how many points they each compute
- All other processes compute their **sub-approximations**
- All other processes send **back** their approximations
- The **root** displays the **final result**

Extended Example 2 - Matrix Multiplication

- We introduce one of the most common structures for a parallel program
  - Self-scheduling
  - Master-worker
  - In this code, the master task distributes a matrix multiply * operation to numtasks-1 worker tasks

Matrix Multiplication - Definition




```
for(int i=0;i<ROW;i++)
{
     for(int j=0;j<ROW;j++)
     {
          for(int z=0;z<COL;z++)
          {
               C[i][j] += A[i][z]*B[z][j];
          }
     }
}
```

References

- Readings
  - Estimating Pi using the Monte Carlo Method
  - MPI Tutorial

Copyright Notice