Introduction to Parallel Programming with MPI

Difference between shared memory and distributed memory computer architectures.

- The price of communication: the time needed to exchange a certain amount of data between two or more processors is much faster in shared memory computers

- The second difference, is in the number of processors that can cooperate efficiently, is in favor of distributed memory computers.

- Usually, our primary choice when computing complex tasks will be to engage a large number of fastest available processors, but the communication among them poses additional limitations.
- Cooperation among processors implies communication or data exchange among them.
- When the number of processors must be high (e.g., more than eight) to reduce the execution time, the speed of communication becomes a crucial performance factor.

- There is a difference in the speed of data movement between two computing cores within a single multi-core computer, depending on the location of data to be communicated.
- This is because the data can be stored in registers, cache memory, or system memory, which can differ by up to two orders of magnitude if their access times are considered

- The differences in the communication speed get even more pronounced in the interconnected computers, again by orders of magnitude, but this now depends on the technology and topology of the interconnection networks and on the geographical distance of the cooperating computers.

- Complex tasks can be executed efficiently either
  - (i) on a small number of extremely fast computers or
  - (ii) on a large number of potentially slower interconnected computers.

Message Passing Interface (MPI)

- Enables **system independent** parallel programming.
- The MPI standard includes process creation and management, language bindings for C, point-to-point and collective communications, group and communicator concepts.
- Programmers have to be aware that the cooperation among processes implies the data exchange.
  - The total execution time is consequently a sum of computation and communication time.

- Algorithms with only local communication between neighboring processors are faster and more scalable than the algorithms with the global communication among all processors.

- Therefore, the programmer's view of a problem that will be parallelized has to incorporate a wide number of aspects
  - e.g., data independency, communication type and frequency, balancing the load among processors, balancing between communication and computation, overlapping communication and computation, synchronous or asynchronous program flow, stopping criteria, and others.

Message Passing Interface (MPI)

- The MPI is not a language
- All MPI "**operations**" are expressed as functions, subroutines, or methods
- The MPI standard defines the syntax and semantics of library operations that support the message passing model, independently of program language or compiler specification.
- An MPI program consists of autonomous processes that are able to execute their own code in the sense of multiple instruction multiple data (MIMD) paradigm.
- An MPI "**process**" can be interpreted in this sense as a program counter that addresses their program instructions in the system memory, which implies that the program codes executed by each process need not to be the same.

- The processes communicate via calls to MPI communication operations, independently of operating system.
- Based on the MPI library specifications, several efficient MPI library implementations have been developed, either in open-source or in a proprietary domain.
- Based on the MPI library specifications, several efficient MPI library implementations have been developed, either in open-source or in a proprietary domain.
- The basic MPI communication is characterized by two fundamental MPI operations
  - MPI_SEND and MPI_RECV that provide sends and receives of process data
  - They are represented by numerous data types.
  - Besides the data transfer these two operations synchronize the cooperating processes in time instants where communication has to be established
  - e.g., a process cannot proceed if the expected data has not arrived.

Message Passing Interface (MPI)

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

int main(int argc, char **argv)
//int main(argc, argv)
//int  argc;
//char **argv;
{
  int rank, size;
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  printf("Hello world from process %d of %d processes.\n", rank, size);
  MPI_Finalize();
  return 0;
}
```

Listing 4.1 "Hello world" MPI program `MSMPIHello.ccp` in C programming syntax.

Message Passing Interface (MPI)

- The program has to be compiled only once to be executed on all active processes. Such a methodology could simplify the development of parallel programs.
- #include "stdafx.h" is needed because the MS Visual Studio compiler has been used
- #include <stdio.h> is needed because of printf, which is used later in the program
- #include "mpi.h" provides basic MPI definition of named constants, types, and function prototypes, and must be included in any MPI program.

- The **number of processes** will be determined by parameter **-n** of the MPI execution utility ***mpiexec***, usually provided by the MPI library implementation.
- MPI_Init initializes the MPI execution environment and
- MPI_Finalize exits the MPI.
- MPI_Comm_size(MPI_COMM_WORLD, & size) returns size, which is the number of started processes.
- MPI_Comm_rank(MPI_COMM_WORLD, & rank) returns rank, i.e., an ID of each process.
- MPI operations return a status of the execution success; in C routines as the value of the function, which is not considered in the above C program

Message Passing Interface (MPI)

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

int main(int argc, char **argv)
//int main(argc, argv)
//int  argc;
//char **argv;
{
  int rank, size;
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  printf("Hello world from process %d of %d processes.\n", rank, size);
  MPI_Finalize();
  return 0;
}
```

Listing 4.1 "Hello world" MPI program `MSMPIHello.ccp` in C programming syntax.

Message Passing Interface (MPI)

- The “Hello World” code is the same for all processes.
- It has to be compiled only once to be executed on all active processes.
- Run the program with:

  $ mpiexec -n 3 MSMPIHello

  from Command prompt of the host process, at the path of directory where MSMPIHello.exe is located.

  The program should output three “Hello World” messages, each with a process identification data.

- All non-MPI procedures are local, e.g., printf in the above example.
- It runs on each process and prints separate “Hello World” notice.
- If one would prefer to have only a notice from a specific process, e.g., 0, an extra if(rank == 0) statement should be inserted.
- ***Note also that in this simple example no communication between processes has been required.***

- Depending on the number of processes, the printf function will run on each process, which will print a separate “Hello World” notice.

- If all processes will print the output, we expect size lines with “HelloWorld” notice, one from each process.

- Note that the order of the printed notices is not known in advance, because there is no guaranty about the ordering of the MPI processes.

MPI Operation Syntax

- The MPI standard is independent of specific programming languages.
- Capitalized MPI operation names will be used in the definition of MPI operations.

  MPI operation arguments, in a language-independent notation, are marked as:

  - IN—for input values that may be used by the operation, but not updated;
  - OUT—for output values that may be updated by the operation, but not used as input value;
  - INOUT—for arguments that may be used and/or updated by the MPI operation.

  - IN arguments are in normal text, e.g., buf, sendbuf, MPI_COMM_WORLD, etc.
  - OUT arguments are in underlined text, e.g., rank, recbuf, etc.
  - INOUT arguments are in underlined italic text, e.g., *inbuf*, *request*, etc.

Some terms and conventions that are implemented with C program language binding:

- Function names are equal to the MPI definitions but with the MPI_ prefix and the first letter of the function name in uppercase, e.g., **MPI_Finalize().**

- The status of execution success of MPI operations is returned as integer return codes, e.g., **ierr = MPI_Finalize().**
  - The return code can be an error code or **MPI_SUCCESS** for successful competition, defined in the file mpi.h.
  - Note that all predefined constants and types are fully capitalized.

- Operation arguments IN are usually passed by value with an exception of the send buffer, which is determined by its initial address. All OUT and INOUT arguments are passed by reference (as pointers)
  - e.g., MPI_Comm_size (MPI_COMM_WORLD, & size).

MPI Data Types

- MPI standard defines its own basic data types that can be used for the specification of message data values, and correspond to the basic data types of the host language.

- As MPI does not require that communicating processes use the same representation of data, i.e., it needs to keep track of possible data types through the build-in basic MPI data types

- For more specific applications, MPI offers operations to construct custom data types, e.g., array of (int, float) pairs, and many other options

- A value of type MPI_BYTE consists of a byte, i.e., 8 binary digits.
- A byte is uninterpreted and is different from a character.
- Different machines may have different representations for characters or may use more than one byte to represent characters.
- On the other hand, a byte has the same binary value on all machines. If the size and representation of data are known, the fastest way is the transmission of raw data, for example, by using an elementary MPI data type MPI_BYTE.

MPI data type	C data type	MPI data type
MPI_INT	int	MPI_INTEGER
MPI_SHORT	short int	MPI_REAL
MPI_LONG	long int	MPI_DOUBLE_PRECISION
MPI_FLOAT	float	MPI_COMPLEX
MPI_DOUBLE	double	MPI_LOGICAL
MPI_CHAR	char	MPI_CHARACTER
MPI_BYTE	/	MPI_BYTE
MPI_PACKED	/	MPI_PACKED

- The MPI communication operations have involved only buffers containing a continuous sequence of identical basic data types.
- Often, one wants to pass messages that contain values with different data types,
  - e.g., a number of integers followed by a sequence of real numbers;
  - or one wants to send noncontiguous data, e.g., a subblock of a matrix.
- The type MPI_PACKED is maintained by MPI_PACK or MPI_UNPACK operations, which enable to pack different types of data into a contiguous send buffer and to unpack it from a contiguous receive buffer

- A user specifies in advance the layout of data types to be sent or received and the communication library can directly access a noncontinuous data.
  - The simplest noncontiguous data type is the vector type, constructed with MPI_Type_vector.
  - For example, a sender process has to communicate the main diagonal of an *N* × *N* array of integers, declared as: int matrix[N][N]; which is stored in a row-major layout.

- A continuous derived data type diagonal can be constructed:
  - MPI_Datatype MPI_diagonal; specifies the main diagonal as a set of integers: MPI_Type_vector (N, 1, N+1, MPI_INT, & diagonal); where their count is N, block length is 1, and stride is N+1.
  - The receiver process receives the data as a contiguous block.

Advantages:

- If all data of an MPI program is specified by MPI types it will support data transfer between processes on computers with different memory organization and different interpretations of elementary data items,
  - e.g., in heterogeneous platforms.
- The parallel programs, designed with MPI data types, can be easily ported even between computers with unknown representations of data.
- Further, the custom application oriented data types can reduce the number of memory-to-memory copies or can be tailored to a dedicated hardware for global communication.

MPI Environment Management Routines:

MPI_Init: Initializes the MPI execution environment. This function must be called in every MPI program, must be called before any other MPI functions and must be called only once in an MPI program.

```
MPI_Init (&argc,&argv);
```

MPI_INIT (int *argc, char ***argv)

MPI_Finalize: Terminates the MPI execution environment. This function should be the last MPI routine called in every MPI program. No other MPI routines may be called after it.

```
MPI_Finalize ();
```

MPI_FINALIZE ()

Note: The arguments argc and argv are required in C language binding only, where they are parameters of the main C program.

No MPI routine can be called before MPI_INIT or after MPI_FINALIZE, with one exception MPI_INITIALIZED (flag), which queries if MPI_INIT has been called.

MPI Environment Management Routines:

MPI_Comm_rank: Returns the rank of the calling MPI process within the specified communicator. Each process will be assigned a unique integer rank between 0 and size - 1 within the communicator MPI_COMM_WORLD. This rank is often referred to as a process ID.

```
MPI_Comm_rank Comm,&rank);
```

MPI_COMM_RANK (comm, rank)

MPI_Comm_size: Returns the total number of MPI processes to the variable size in the specified communicator, such as MPI_COMM_WORLD. **Returns the number of processes in the current communicator**

```
MPI_Comm_size(Comm,&size);
```

MPI_COMM_SIZE (comm, size)

- The input argument comm is the handle of communicator; the output argument size returned by the operation MPI_COMM_SIZE is the number of processes in the group of comm.
- If comm is MPI_COMM_WORLD, then it represents the number of all active MPI processes.

MPI Error Handling

- The MPI standard assumes a reliable and error-free underlying communication platform; therefore, it does not provide mechanisms for dealing with failures in the communication system.
  - For example, a message sent is always received correctly, and the user need not check for transmission errors, time-outs, or similar.
- MPI does not provide mechanisms for handling processor failures. A program error can follow an MPI operation call with incorrect arguments,
  - e.g., non-existing destination in a send operation, exceeding available system resources, or similar

- Most of MPI operation calls return an error code that indicates the completion status of the operation.
- Before the error value is returned, the current MPI error handler is called, which, by default, aborts all MPI processes.
- One can specify that no MPI error is fatal, and handle the returned error codes by custom error-handling routines.

MPI Environment Management Routines:

Solved Example:

Write a program in MPI to print total number of process and rank of each process.

```
#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("My rank is %d in total %d processes", rank, size);
        MPI_Finalize();
        return 0;
}
```

Process-to-Process Communication or Point to point Communication in MPI

Objectives:

1. Understand the different APIs used for point to point communication in MPI
2. Learn the different modes available in case of blocking send operation

- The process-to-process communication has to implement two essential tasks:
  - data movement and
  - Synchronization of processes;

  Therefore, it requires cooperation of sender and receiver processes.

Process-to-Process Communication or Point to point Communication in MPI

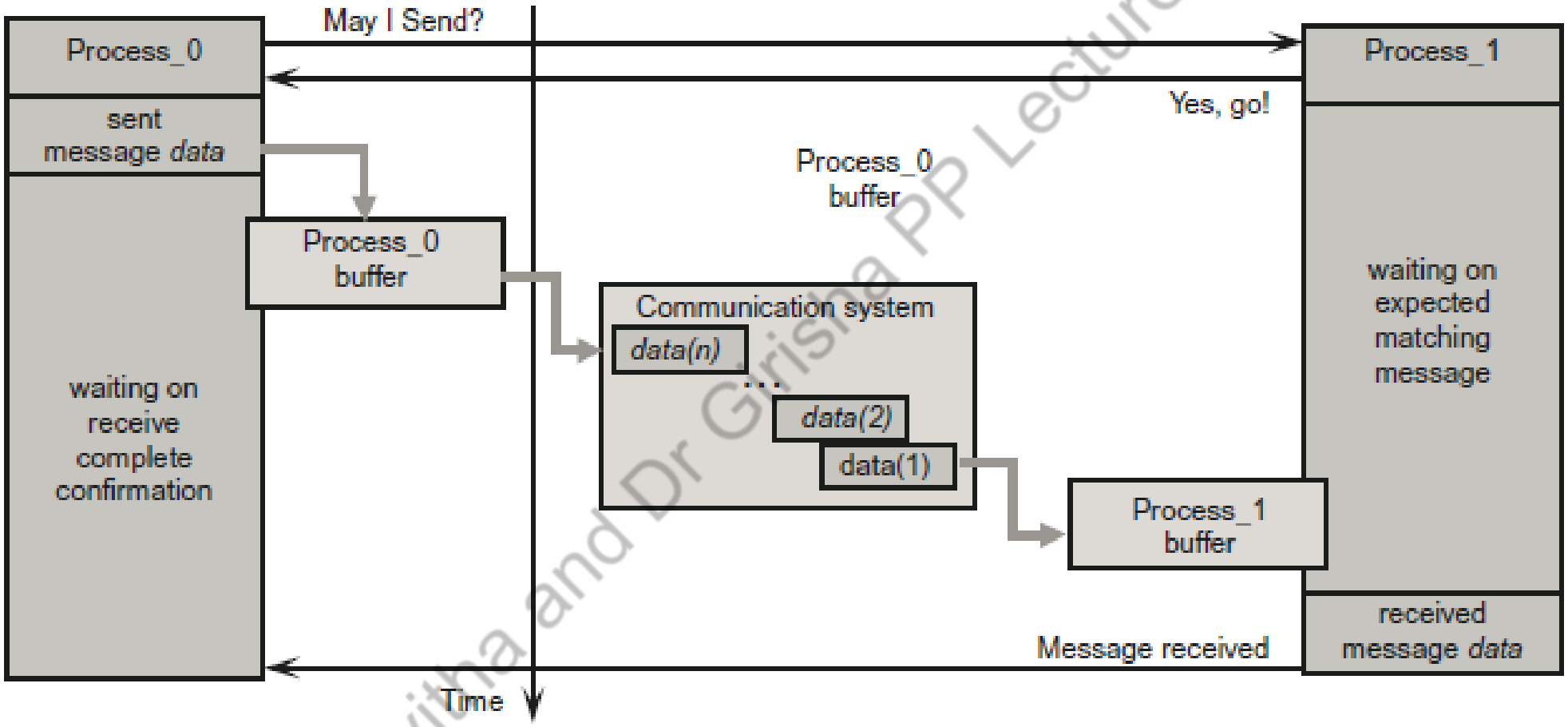


Fig. 4.1 Communication between two processes awakes both of them while transferring data from sender `Process_0` to receiver `Process_1`, possibly with a set of shorter sub-messages

Point to Point Communication in MPI

- Optional **intermediate message buffers** are used in order to enable sender Process_0 to continue immediately after it initiates the send operation.
- However, Process_0 will have to wait on the return from the previous call, before it can send a new message.
- On the receiver side, Process_1 can do some useful work instead of idling while waiting on the matching message reception.
- It is a communication system that must ensure that the message will be reliably transferred between both processes.
- If the processes have been created on a single computer, the actual communication will be probably implemented through a shared memory.

- If the processes reside on two distant computers, then the actual communication might be performed through an existing interconnection network using, e.g., TCP/IP communication protocol.
- Although that blocking send/receive operations enable a simple way for **synchronization of processes**, they could introduce **unnecessary delays** in cases where sender and receiver do not reach communication point at the same real time.
- For example, if Process_0 issues a send call significantly before the matching receives call in Process_1, Process_0 will start waiting to the actual message data transfer.
- In the same way, processes' idling can happen if a process that produces many messages is much faster than the consumer process. **Message buffering may alleviate the idling** to some extent, but if the amount of data exceeds the capacity of the message buffer, which can always happen, Process_0 will be blocked again.

Point to Point Communication in MPI

- The next concern of the blocking communication are **deadlocks**.
- For example, if Process_0 and Process_1 initiate their send calls in the same time, they will be blocked forever by waiting matching receive calls.

Point to Point Communication in MPI

Point to Point communication in MPI

- MPI point-to-point operations typically involve message passing between two, and only two, different MPI tasks. One task is performing a send operation and the other task is performing a matching receive operation.
- MPI provides both blocking and non-blocking send and receive operations.

MPI_SEND (buf, count, datatype, dest, tag, comm)

- The send buffer is specified by the following arguments
  - buf - pointer to the send buffer,
  - count - number of data items,
  - datatype - type of data items.
- The receiver process is addressed by an envelope that consists of arguments
  - dest, which is the rank of receiver process within all processes in the communicator comm, and of a message tag.
  - **tag** provide a mechanism for distinguishing between different messages for the same receiver process identified by destination rank

Point to Point Communication in MPI

MPI_RECV (buf, count, datatype, source, tag, comm, status)

This operation waits until the communication system delivers a message with matching datatype, source, tag, and comm.

- The entire set of arguments: count, datatype, source, tag and comm, must match between the sender process and the receiver process to initiate actual message passing.
- When a message, posted by a sender process, has been collected by a receiver process, the message is said to be completed, and the program flows of the receiver and the sender processes may continue.

Point to Point Communication in MPI

Sending message in MPI

- **Blocked Send** sends a message to another processor and waits until the receiver has received it before continuing the process. Also called as **Synchronous send**.
- **Send** sends a message and continues without waiting. Also called as **Asynchronous send.**

There are multiple communication modes used in blocking send operation:

- **Standard mode**
- **Synchronous mode**
- **Buffered mode**

Point to Point Communication in MPI

Standard mode

This mode blocks until the message is buffered.

```
MPI_Send(&Msg, Count, Datatype, Destination, Tag,  Comm);
```

- First 3 parameters together constitute message buffer. The **Msg** could be any address in sender's address space. The **Count** indicates the number of data elements of a particular type to be sent. The **Datatype** specifies the message type. Some Data types available in MPI are: MPI_INT, MPI_FLOAT, MPI_CHAR, MPI_DOUBLE, MPI_LONG
- Next 3 parameters specify message envelope. The **Destination** specifies the rank of the process to which the message is to be sent.
- **Tag:** The **tag** is an integer used by the programmer to label different types of messages and to restrict message reception.

Point to Point Communication in MPI

- **Communicator:** Major problem with tags is that they are specified by users who can make mistakes. **Context** are allocated at run time by the system in response to user request and are used for matching messages. The notions of **context and group** are combined in a single object called a communicator (**Comm**).
- The default process group is **MPI_COMM_WORLD**.

Point to Point Communication in MPI

Synchronous mode

This mode requires a send to block until the corresponding receive has occurred.

```
MPI_Ssend(&Msg, Count, Datatype, Destination, Tag,  Comm);
```

Buffered mode

```
MPI_Bsend(&Msg, Count, Datatype, Destination, Tag,  Comm);
```

In this mode a send assumes availability of a certain amount of buffer space, which must be previously specified by the user program through a routine call that allocates a user buffer.

Point to Point Communication in MPI

```
MPI-Buffer_attach(buffer, size);
```

This buffer can be released by

```
MPI_Buffer_detach(*buffer, *size);
```

Point to Point Communication in MPI

Receiving message in MPI

```
MPI_Recv(&Msg, Count, Datatype, Source, Tag, Comm, &status);
```

- Receive a message and block until the requested data is available in the application buffer in the receiving task.
- The **Msg** could be any address in receiver's address space. The **Count** specifies number of data items. The **Datatype** specifies the message type. The **Source** specifies the rank of the process which has sent the message. The **Tag** and **Comm** should be same as that is used in corresponding send operation. The status is a structure of type status which contains following information: Sender's rank, Sender's tag and number of items received

Point to Point Communication in MPI

Finding execution time in MPI

MPI_Wtime: Returns an elapsed wall clock time in seconds (double precision) on the calling processor.

- **MPI_Wtime ()**

```
double start, finish;
start = MPI_Wtime ();
   ... //MPI program segment to be clocked
finish = MPI_Wtime ();
printf ("Elapsed time is %f\n", finish - start);
```

Solved Example:

Write a MPI program using standard send. The sender process sends a number to the receiver. The second process receives the number and prints it.

```
#include "mpi.h"
#include <stdio.h>
int main(int argc, char *argv[])
{
        int rank,size,x;
        MPI_Init(&argc,&argv);
        MPI_Comm_rank(MPI_COMM_WORLD,&rank);
        MPI_Comm_size(MPI_COMM_WORLD, &size);
        MPI_Status status;
        if(rank==0)
         {
          Printf("Enter a value in master process:");
          scanf("%d",&x);
           MPI_Send(&x,1,MPI_INT,1,1,MPI_COMM_WORLD);
           fprintf(stdout,"I have sent %d from process 0\n",x);
           fflush(stdout);
         }
        else
        {
         MPI_Recv(&x,1,MPI_INT,0,1,MPI_COMM_WORLD,&status);
         fprintf(stdout,"I have received %d in process 1\n",x);
         fflush(stdout);
         }
       MPI_Finalize();
       return 0;
}
```

Point to Point Communication in MPI

Seven basic MPI operations

Many parallel programs can be written and evaluated just by using the following seven MPI operations that have been overviewed in the previous sections:

```
MPI_INIT,
MPI_FINALIZE,
MPI_COMM_SIZE,
MPI_COMM_RANK,
MPI_SEND,
MPI_RECV,
MPI_WTIME.
```

Enable MPI in Visual Studio

- Download MPI for Windows(Microsoft MPI)
- https://www.microsoft.com/en-s/download/details.aspx?id=57467
- Run both **.exe** and **.msi** file, they will install Microsoft MPI under C:\Program Files\Microsoft MPI by default.(But if you have changed the register manually, the path might be changed)

Configure MPI in Visual Studio 2019

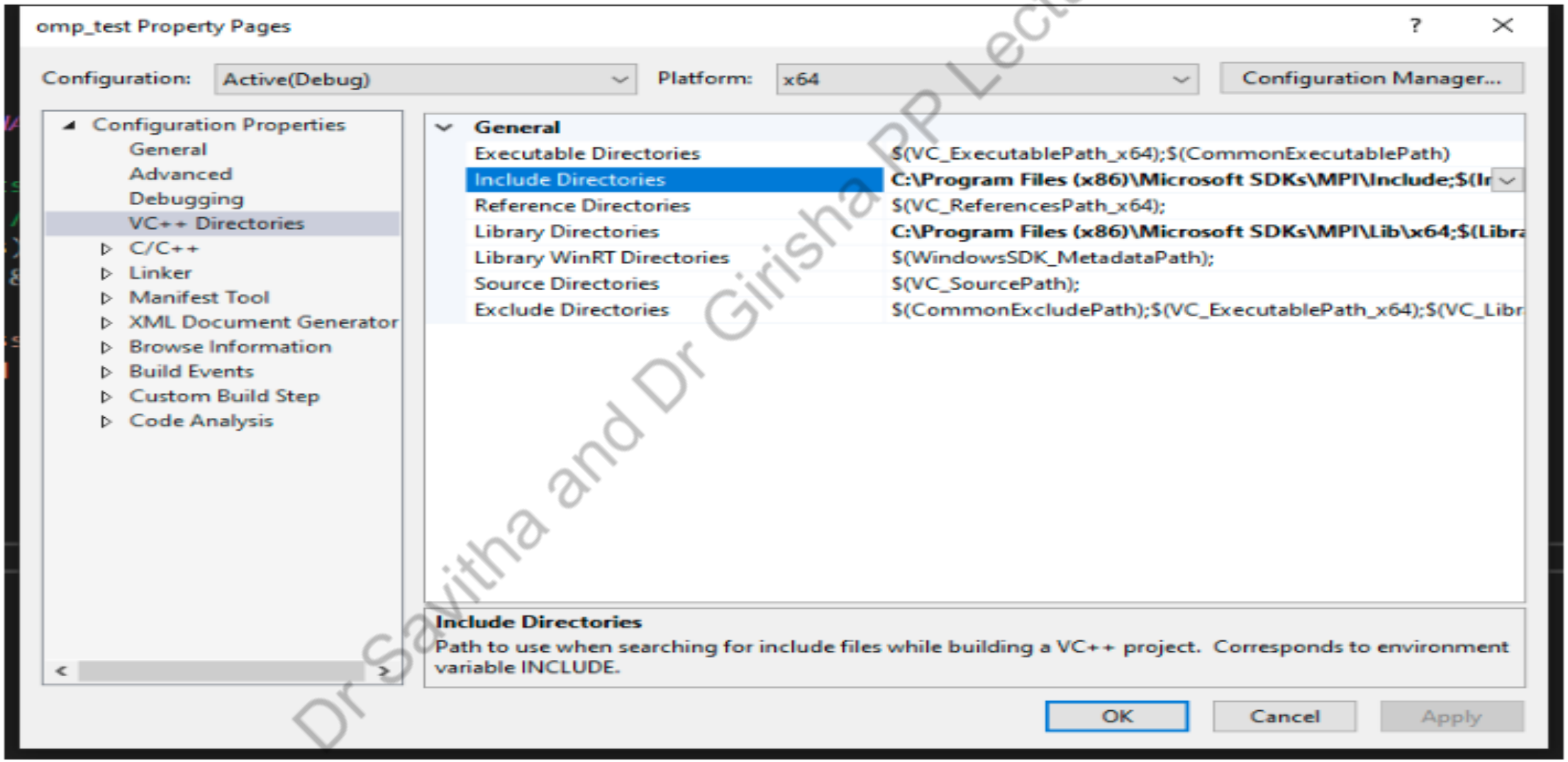
Open Project -> Project_name Properties

Under VC++ Directories

Add C:\Program Files (x86)\MicrosoftSDKs\MPI\Include in Include Directories

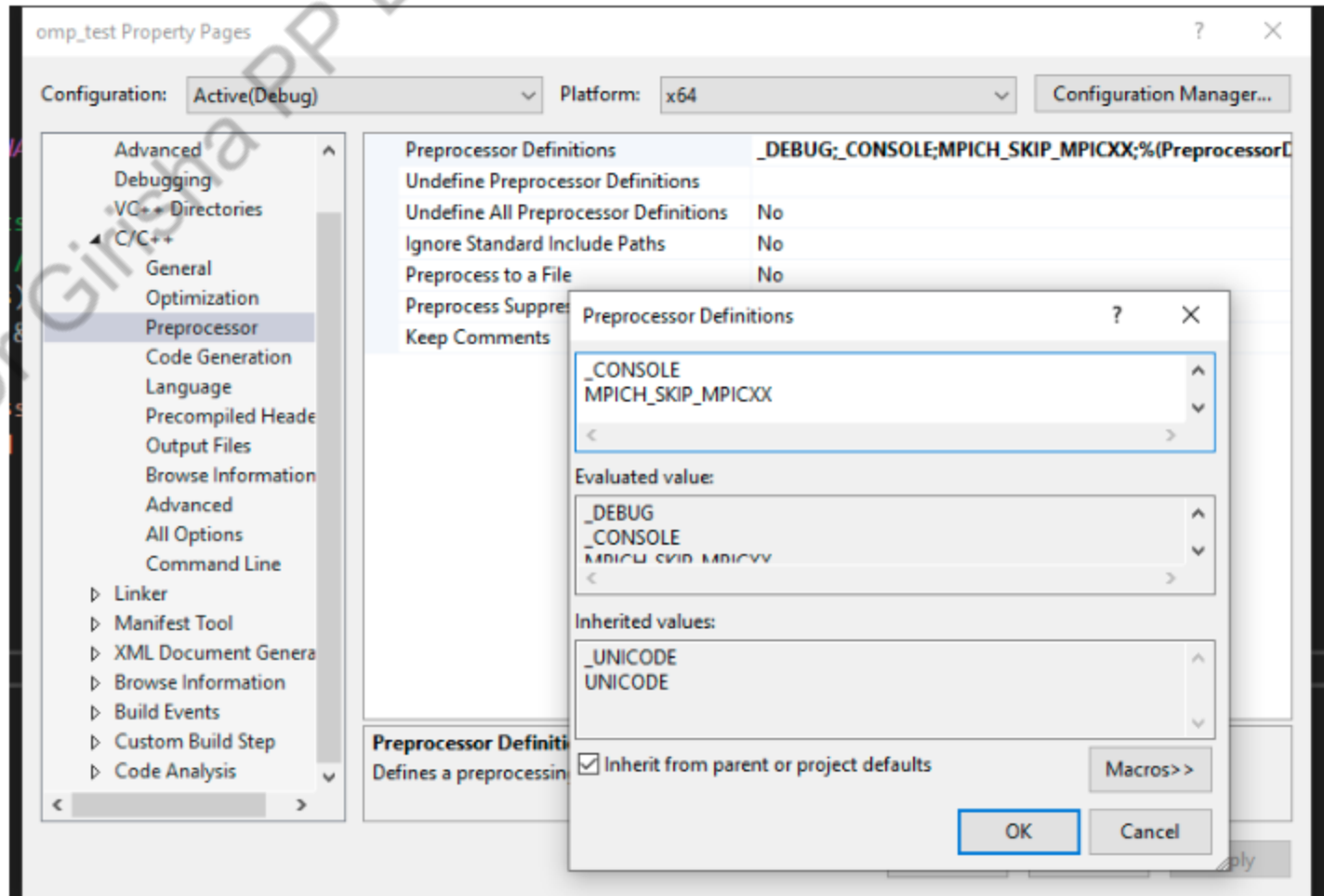
Add C:\Program Files(x86)\MicrosoftSDKs\MPI\Lib\x86 in Library Directories

Configure MPI in Visual Studio 2019



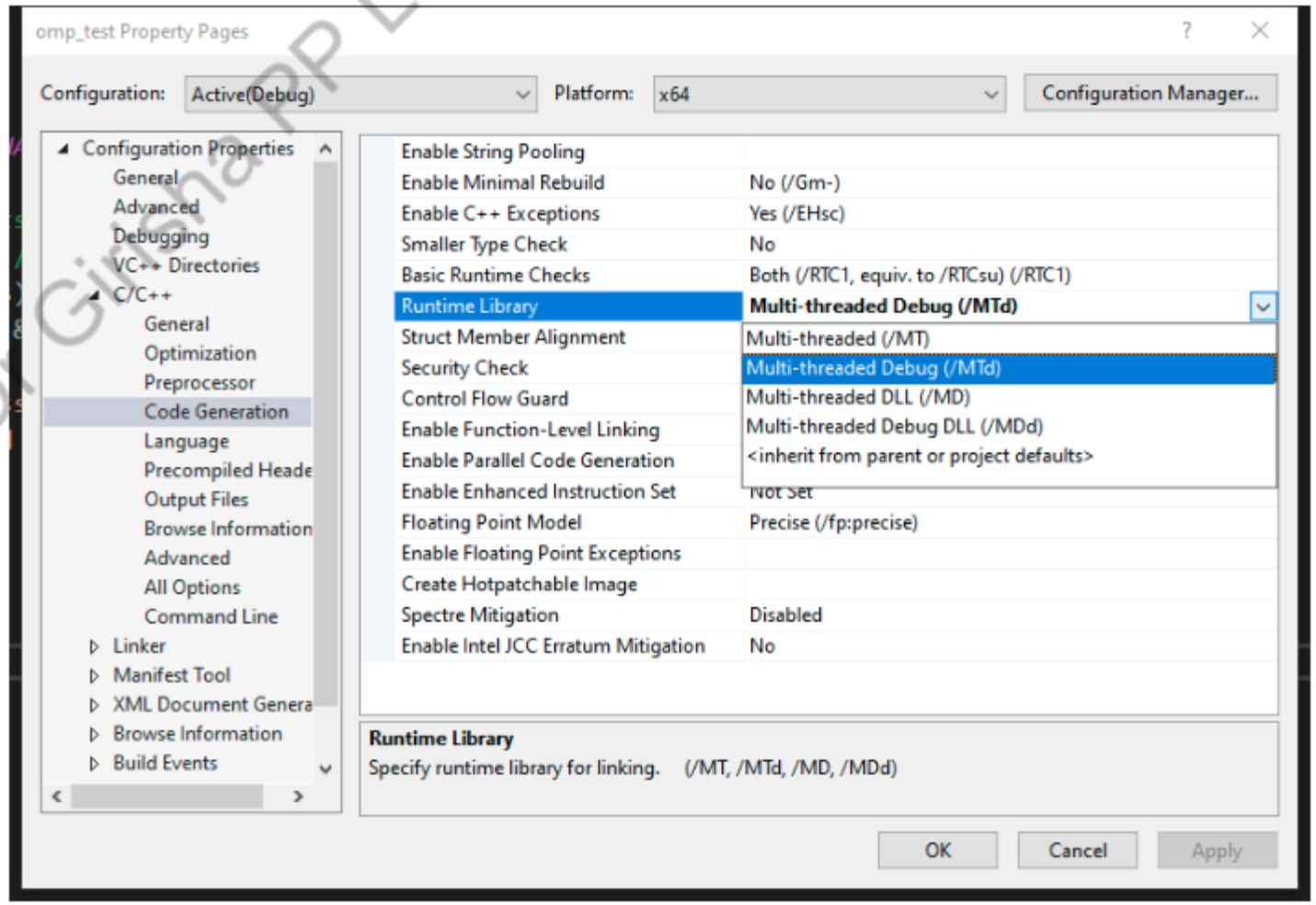
Configure MPI in Visual Studio 2019

- In C/C++ -> Preprocessor -> Preprocessor Definitions
- Add MPICH_SKIP_MPICXX



Configure MPI in Visual Studio 2019

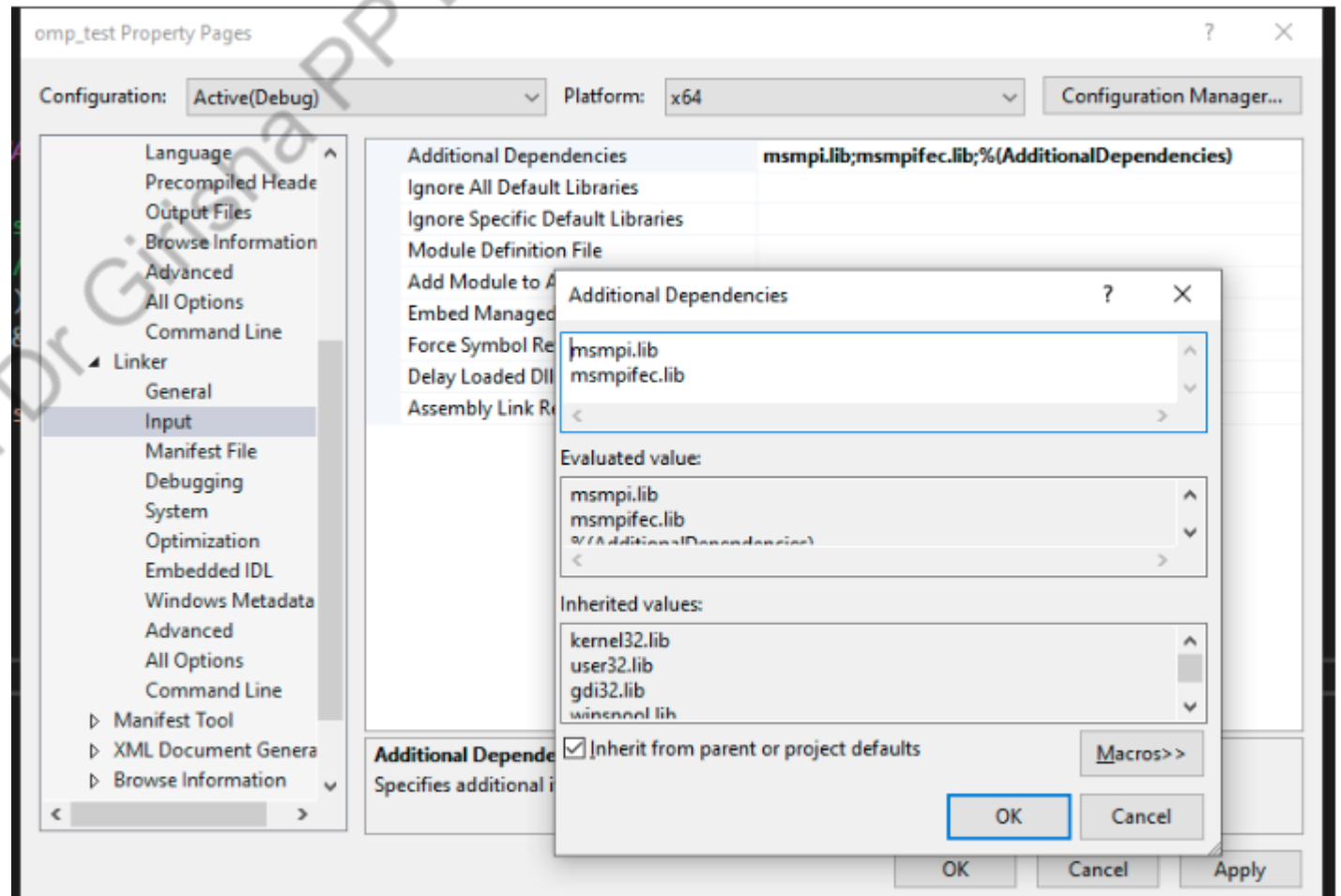
- In **C/C++** -> **Code Generation**
- Change Runtime Library to MTd



Configure MPI in Visual Studio 2019

In Linker -> Input -> Additional Dependencies

Add msmpi.lib and msmpifec.lib



Testing

```
#include<stdio.h> #include<mpi.h> #include<stdlib.h>
int main(int argc, char* argv[])
{
        int myid, numprocs, namelen;
        char processor_name[MPI_MAX_PROCESSOR_NAME];

        MPI_Init(&argc, &argv);         // starts MPI
        MPI_Comm_rank(MPI_COMM_WORLD, &myid);  // get current process id
        MPI_Comm_size(MPI_COMM_WORLD, &numprocs);       // get number of processeser
        MPI_Get_processor_name(processor_name, &namelen);

        if (myid == 0) printf("number of processes: %d\n...", numprocs);
        printf("%s: Hello world from process %d \n", processor_name, myid);

        MPI_Finalize();

        return 0;
}
```

Click Build -> Build Solution

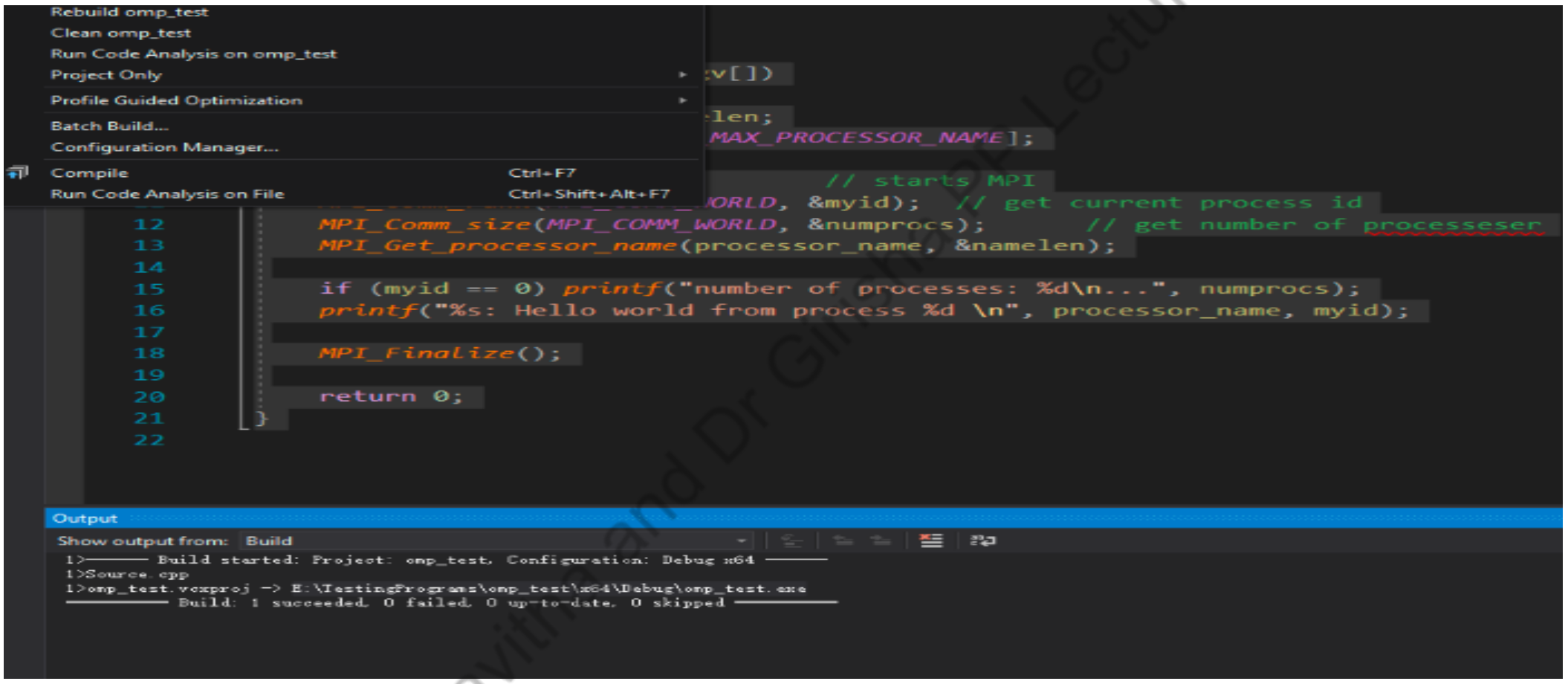
And your terminal will look like the following screenshot.

Please make sure you can build .exe file successfully without error.

You can see the .exe file path inside the terminal, for me, it is

E:\TestingPrograms\omp_test\x64\Debug\omp_test.exe

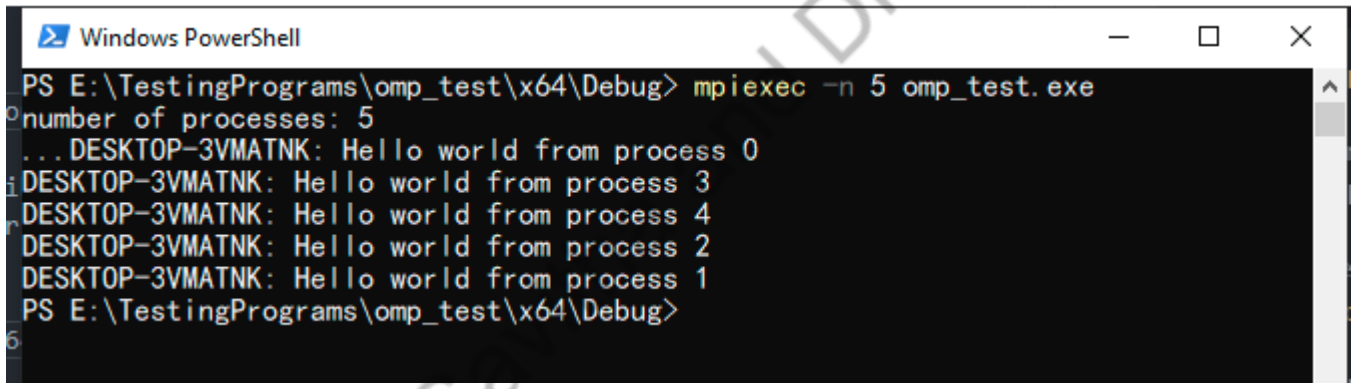
Testing



Open file explorer, E:\TestingPrograms\omp_test\x64\Debug\ folder (Your path will be different!)
Do right-click while pressing Shift button

Testing

- Enter mpiexec -n 5 omp_test.exe in powershell or command-line window(depending on your Windows version, for Win10 you will see powershell, but for early version you will see command-line)
- You can replace 5 with number of process you want,
- omp_test.exe must be replaced by the name of your builded .exe file



https://610yilingliu.github.io/2020/07/21/ConfigureOpenMPI/

Collective MPI Communication

Objectives:

1. Understand the usage of collective communication in MPI
2. Learn how to broadcast messages from root
3. Learn and use the APIs for distributing values from root and gathering the values in the root

Collective MPI Communication

- Collective Communication routines
  - When **all processes** in a group participate in a global communication operation, the resulting communication is called a collective communication.
- The MPI collective operations are called by all processes in a communicator
  - MPI_BARRIER (comm)
  - MPI_BCAST (inbuf, incnt, intype, root, comm)
  - MPI_GATHER (inbuf, incnt, intype, outbuf, outcnt, outtype, root, comm)
  - MPI_SCATTER (inbuf, incnt, intype, outbuf, outcnt, outtype, root, comm)
- Tasks that can be elegantly implemented in this way are as follows:
  - global synchronization
  - reception of a local data item from all cooperating processes in the communicator

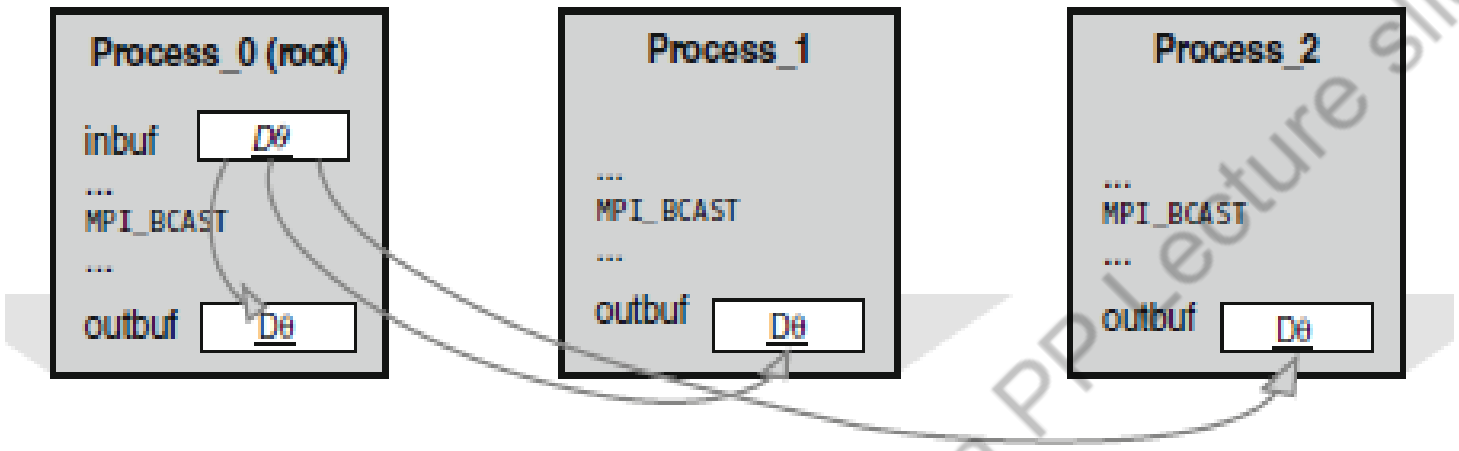
MPI_BARRIER (comm)

- This operation is used to synchronize the execution of a group of processes specified within the communicator ***comm***
- When a process reaches this operation, it has to wait until all other processes have reached the MPI_BARRIER.
  - No process returns from MPI_BARRIER until all processes have called it.
- Programmer is responsible that all processes from communicator comm will really call MPI_BARRIER.
- The barrier is a simple way of separating two phases of a computation to ensure that messages generated in different phases do not interfere.

- MPI_BARRIER is a global operation that invokes all processes; therefore, it could be time-consuming. In many cases, the call to MPI_BARRIER should be avoided by an appropriate use of explicit addressing options, e.g., tag, source, or comm.

MPI_BCAST (inbuf, incnt, intype, root, comm)

- The operation implements a ***one-to-all broadcast operation*** whereby a single named ***process root*** sends its data to ***all other processes*** in the communicator, including to ***itself***
- Each process receives this data from the root process, which can be of any rank
- At the time of call, the input data are located in ***inbuf*** of process root and consists of ***incnt*** data items of a specified ***intype***
- After the call, the data are replicated in ***inbuf*** as output data of all remaining processes.
- As inbuf is used as an *input argument at the root process, but as an output argument in all remaining processes*, it is of the INOUT type.

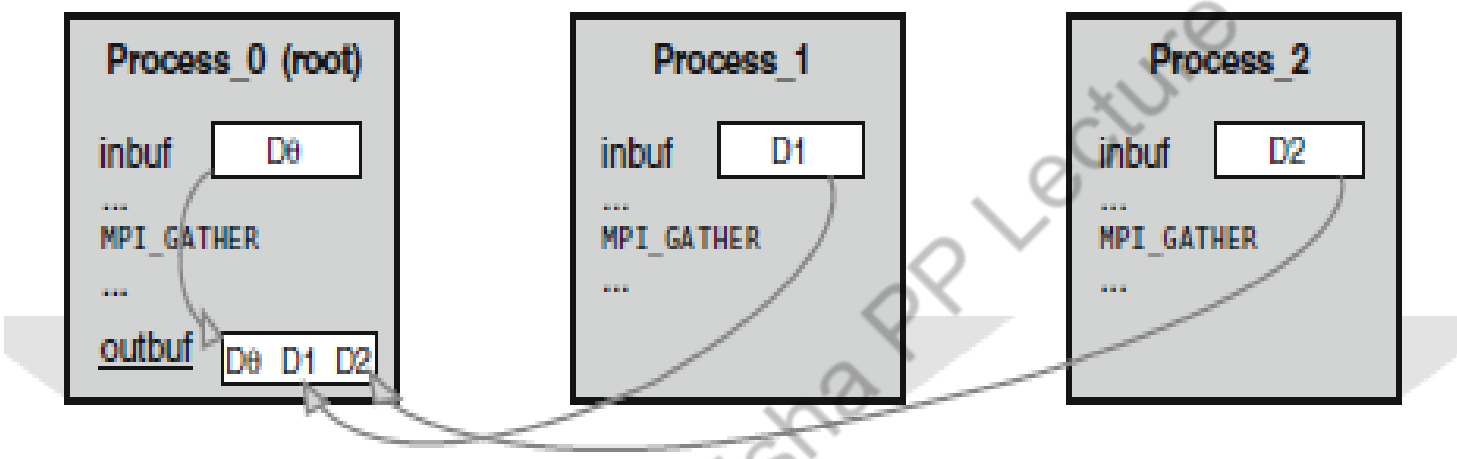


- A simple case of three processes is depicted above.
- The process with rank = 0 is the root process.
- Arrows symbolize the required message transfer.
- Note that all processes have to call MPI_BCAST to complete the requested data relocation

- The functionality of MPI_BCAST could be implemented, in the above example, by three calls to MPI_SEND in the root process and by a single corresponding MPI_RECV call in any remaining process.
- Usually, such an implementation will be less efficient than the original MPI_BCAST.
- All collective communications could be *time-consuming*.
- Their *efficiency* is strongly related with the *topology and performance of interconnection network.*
- *The process ranked **Root** send the same message whose content is identified by the triple (Address, Count, Datatype) to **all** processes (including itself) in the communicator **Comm.***

MPI_GATHER (inbuf, incnt, intype, outbuf, outcnt, outtype, root, comm)

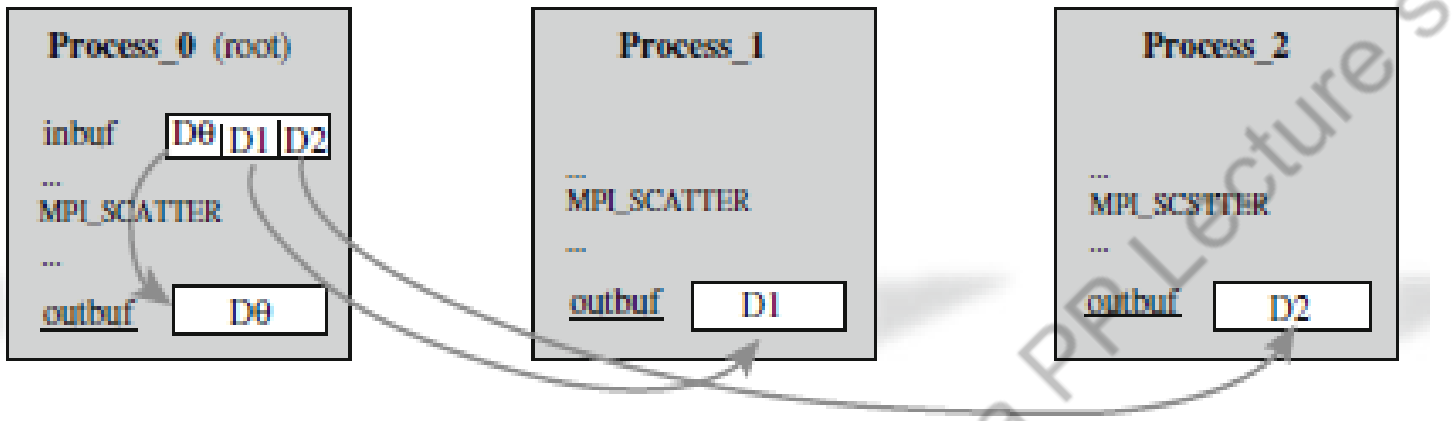
- *All-to-one collective communication* is implemented by MPI_GATHER.
- This operation is also called *by all processes* in the communicator.
- Each process, including *root process*, sends its *input data located in inbuf that consists of incnt data items of a specified intype, to the root process*, which can be of any rank.
- Note that the communication data can be different in count and type for each process.
- However, the root process has *to allocate enough space, through its output buffer, that suffices for all expected data*.
- After the return from MPI_GATHER in all processes, the data are collected in *outbuf of the root processes*.



- A schematic presentation of data relocation after the call to MPI_GATHER is shown in here.
- Example is for the case of three processes, where process with rank = 0 is the root process.
- Note again that all processes have to call MPI_GATHER to complete the requested data relocation.

MPI_SCATTER (inbuf, incnt, intype, outbuf, outcnt, outtype, root, comm)

- This operation works *inverse to MPI_GATHER*
  - i.e., it scatters data from *inbuf of process root to outbuf of all remaining processes, including itself*.
- Note that the count *outcnt and type outtype of the data in each of the receiver processes are the same, so, data is scattered into equal segments*.



- A schematic presentation of data relocation after the call to MPI_SCATTER is shown in Figure, for the case of three processes, where process with rank = 0 is the root process.
- Note again that all processes have to call MPI_SCATTER to complete the requested data relocation.

- There are also more complex collective operations
  - e.g., **MPI_GATHERV** and **MPI_SCATTERV** that allow a varying count of process data from each process and permit some options for process data placement on the root process.
- Such extensions are possible by changing the ***incnt*** and ***outcnt*** arguments from a single integer to an array of integers, and by providing a new array argument ***displs*** for specifying the displacement relative to *root* buffers at which to place the processes data.

Collective MPI Data Manipulations

- MPI provides a set of operations that perform several simple manipulations on the transferred data.

- These operations represent a combination of *collective communication* and *computational manipulation* in a single call and therefore simplify MPI programs.

- Collective MPI operations for data manipulation are based on data *reduction paradigm* that involves *reducing a set of numbers into a smaller set of numbers via a data manipulation*.

Example:

- Three pairs of numbers: {5, 1}, {3, 2}, {7, 6}, each representing the local data of a process
- Can be reduced in a pair of maximum numbers, i.e., {7, 6}
- Or in a sum of all pair numbers, i.e., {15, 9}

- Reduction operations defined by MPI:
  - ***MPI_MAX, MPI_MIN***; return either maximum or minimum data item;
  - ***MPI_SUM, MPI_PROD***; return either sum or product of aligned data items;
  - ***MPI_LAND, MPI_LOR, MPI_BAND, MPI_BOR***; return logical or bitwise AND or OR operation across the data items;
  - ***MPI_MAXLOC, MPI_MINLOC***; return the maximum or minimum value and the rank of the process that owns it;

- The MPI operation that implements all kind of data reductions is

MPI_REDUCE (inbuf, outbuf, count, type, op, root, comm).

  - The ***MPI_REDUCE*** operation implements manipulation ***op*** on matching data items in input buffer ***inbuf*** from all processes in the communicator comm.
  - The results of the manipulation are stored in the output buffer ***outbuf*** of process root.
  - The functionality of ***MPI_REDUCE*** is in fact an ***MPI_GATHER*** followed by manipulation op in process root.
  - Reduce operations are implemented on a per-element basis, i.e., ***i***th elements from each process ***inbuf*** are combined into the ***i***th element in ***outbuf*** of process root.

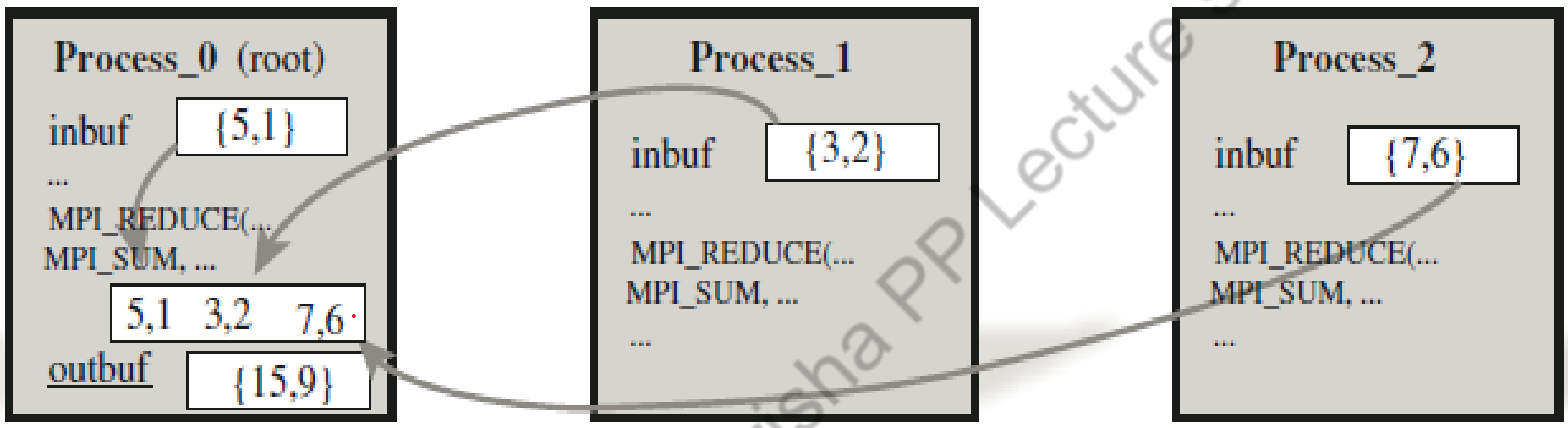


Fig. 4.7 Root process collects the data from input buffers of all processes, performs per-element `MPI_SUM` manipulation, and saves the result in its output buffer

- In many parallel calculations, a global problem domain is divided into subdomains that are assigned to corresponding processes.
- Often, an algorithm requires that all processes take a decision based on the global data.
  - For example, an iterative calculation can stop when the maximal solution error reaches a specified value.
  - An approach to the implementation of the stopping criteria could be the calculation of maximal sub-domain errors and collection of them in a root process, which will evaluate stopping criteria and broadcast the final result/decision to all processes

- MPI provides a specialized operation for this task, i.e.:

MPI_ALLREDUCE (inbuf, outbuf, count, type, op, comm)

- It improves simplicity and efficiency of MPI programs.
- It works as MPI_REDUCE followed by MPI_BCAST.
- Note that the argument root is not needed anymore because the final result has to be available to all processes in the communicator.
- For the same ***inbuf*** data as in Fig. 4.7 and with ***MPI_SUM*** manipulation, a call to ***MPI_ALLREDUCE*** will produce the result {15, 9}, in output buffers of all processes in the communicator.

- MPI_REDUCE (inbuf,outbuf,2,MPI_INT, MPI_SUM,0,MPI_COMM_ WORLD)
  - Before the call, inbuf of three processes with ranks 0, 1, and 2 were: {5, 1}, {3, 2}, and {7, 6}, respectively.
  - After the call to the MPI_REDUCE the value in outbuf of root process is {15, 9}

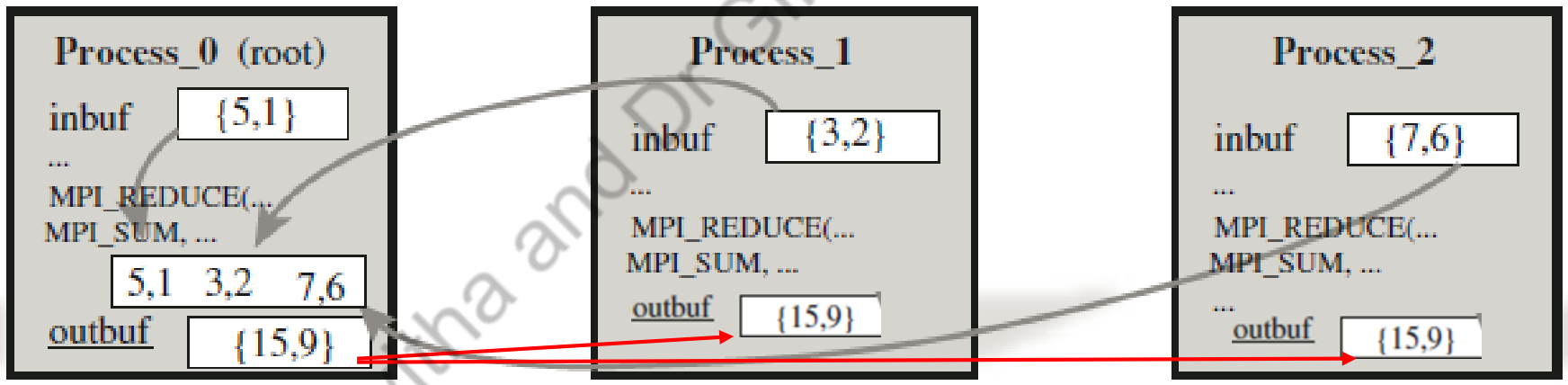


Fig. 4.7 Root process collects the data from input buffers of all processes, performs per-element `MPI_SUM` manipulation, and saves the result in its output buffer

List of MPI Operations :

Basic MPI operations:

```
MPI_INIT, MPI_FINALIZE,
MPI_COMM_SIZE, MPI_COMM_RANK,
MPI_SEND, MPI_RECV,
```

MPI operations for collective communication:

```
MPI_BARRIER,
MPI_BCAST, MPI_GATHER, MPI_SCATTER,
MPI_REDUCE, MPI_ALLREDUCE,
```

Control MPI operations:

```
MPI_WTIME, MPI_STATUS,
MPI_INITIALIZED.
```

END