



Lecture 5

Programming Using the Message-Passing Paradigm II

MPI: the Message Passing Interface; Unicast

Ceng471 *Parallel Computing* at November 25, 2010

MPI: the Message Passing Interface

Starting and Terminating
the MPI Library

Communicators

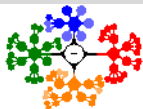
Getting Information

Sending and Receiving
Messages

Avoiding Deadlocks

Sending and Receiving
Messages Simultaneously

Dr. Cem Özdoğan
Computer Engineering Department
Çankaya University



1 MPI: the Message Passing Interface

Starting and Terminating the MPI Library

Communicators

Getting Information

Sending and Receiving Messages

Avoiding Deadlocks

Sending and Receiving Messages Simultaneously

MPI: the Message Passing Interface

Starting and Terminating
the MPI Library

Communicators

Getting Information

Sending and Receiving
Messages

Avoiding Deadlocks

Sending and Receiving
Messages Simultaneously

MPI: the Message Passing Interface I

- Many early generation commercial parallel computers were based on the message-passing architecture due to its lower cost relative to shared-address-space architectures.
- Message-passing became the modern-age form of assembly language, in which every hardware vendor provided its own library.
- Performed very well on its own hardware, but was incompatible with the parallel computers offered by other vendors.
- Many of the differences between the various vendor-specific message-passing libraries were only syntactic.
- However, often enough there were some *serious semantic differences* that required significant re-engineering to port a message-passing program from one library to another.
- The message-passing interface (MPI) was created to essentially solve this problem.



MPI: the Message Passing Interface

Starting and Terminating
the MPI Library

Communicators

Getting Information

Sending and Receiving
Messages

Avoiding Deadlocks

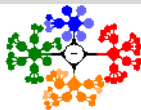
Sending and Receiving
Messages Simultaneously

MPI: the Message Passing Interface II

- MPI defines
 - a standard library for message-passing,
 - can be used to develop **portable** message-passing programs.
- The MPI standard defines both the syntax as well as the semantics of a core set of library routines.
- The MPI library contains over 125 routines, but the number of key concepts is much smaller.
- In fact, it is possible to write fully-functional message-passing programs by using only six routines (see table 1).

Table: The minimal set of MPI routines.

MPI_Init	Initializes MPI
MPI_Finalize	Terminates MPI
MPI_Comm_size	Determines the number of processes
MPI_Comm_rank	Determines the label of the calling process
MPI_Send	Sends a message
MPI_Recv	Receives a message



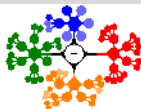


- **MPI_Init** is called prior to any calls to other MPI routines.
 - Its purpose is to initialize the mpi environment.
 - Calling **MPI_Init** more than once during the execution of a program will lead to an error.
- **MPI_Finalize** is called at the end of the computation.
 - It performs various clean-up tasks to terminate the MPI environment.
 - No MPI calls may be performed after **MPI_Finalize** has been called, not even **MPI_Init**.
- Upon successful execution, **MPI_Init** and **MPI_Finalize** return *MPI_SUCCESS*; otherwise they return an implementation-defined error code.

- A key concept used throughout MPI is that of the communication domain.
- A communication domain is a set of processes that are allowed to communicate with each other.
- Information about communication domains is stored in variables of type *MPI_Comm*, that are called communicators.
- These communicators are used as arguments to all message transfer MPI routines.
- They uniquely identify the processes participating in the message transfer operation.



- In general, all the processes may need to communicate with each other.
- For this reason, MPI defines a default communicator called *MPI_COMM_WORLD* which includes all the processes involved.
- However, in many cases we want to perform communication only within (possibly overlapping) groups of processes.
- By using a different communicator for each such group, we can ensure that no messages will ever interfere with messages destined to any other group.



- **MPI_Comm_size** function \implies number of processes
- **MPI_Comm_rank** function \implies label of the calling process
- The calling sequences of these routines are as follows:

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

- Note that each process that calls either one of these functions must belong in the supplied communicator, otherwise an error will occur.
- The function `MPI_Comm_size` returns in the variable `size` the number of processes that belong to the communicator `comm`.
- So, when there is a single process per processor, the call `MPI_Comm_size(MPI_COMM_WORLD, &size)` will return in `size` the number of processors used by the program.





- Every process that belongs to a communicator is uniquely identified by its rank.
- The rank of a process is an integer that ranges from zero up to the size of the communicator minus one.
- A process can determine its rank in a communicator by calling

```
MPI_Comm_rank(MPI_COMM_WORLD, &rank)
```

that takes two arguments:

- 1 the communicator,
 - 2 an integer variable rank.
- Up on return, the variable *rank* stores the rank of the process.

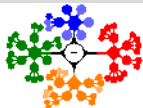
Sending and Receiving Messages I

- The basic functions for sending and receiving messages in MPI are the **MPI_Send** and **MPI_Recv**, respectively.
- The calling sequences of these routines are as follows:

```
int MPI_Send(void *buf, int count,
             MPI_Datatype datatype,
             int dest, int tag,
             MPI_Comm comm)
```

```
int MPI_Recv(void *buf, int count,
             MPI_Datatype datatype,
             int source, int tag,
             MPI_Comm comm,
             MPI_Status *status)
```

- **MPI_Send** sends the data stored in the buffer pointed by *buf*.
- This buffer consists of consecutive entries of the type specified by the parameter *datatype*.
- The number of entries in the buffer is given by the parameter *count*.



Sending and Receiving Messages II

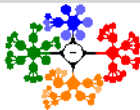


Table: Correspondence between the datatypes supported by MPI and those supported by C.

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	
MPI_PACKED	

MPI: the Message Passing Interface

Starting and Terminating
the MPI Library

Communicators

Getting Information

Sending and Receiving
Messages

Avoiding Deadlocks

Sending and Receiving
Messages Simultaneously

Note that for all C datatypes, an equivalent MPI datatype is provided.

Sending and Receiving Messages III

- MPI allows two additional datatypes that are not part of the C language.
- These are *MPI_BYTE* and *MPI_PACKED*.
 - *MPI_BYTE* corresponds to a byte (8 bits)
 - *MPI_PACKED* corresponds to a collection of data items that has been created by packing non-contiguous data.
- Note that the length of the message in **MPI_Send**, as well as in other MPI routines, is specified *in terms of the number of entries* being sent and *not in terms of the number of bytes*.
- Specifying the length in terms of the number of entries has the advantage of making the MPI code portable,
- since the number of bytes used to store various datatypes can be different for different architectures.



Sending and Receiving Messages IV

- The destination of the message sent by **MPI_Send** is uniquely specified by
 - dest argument. This argument is the *rank* of the destination process in the communication domain specified by the communicator *comm*.
 - comm argument.
- Each message has an integer-valued tag associated with it.
- This is used to distinguish different types of messages.
- The message-tag can take values ranging from zero up to the MPI defined constant *MPI_TAG_UB* (implementation specific, at least 32767).



Sending and Receiving Messages V

- **MPI_Recv** receives a message sent by a process whose *rank* is given by the *source* in the communication domain specified by the *comm* argument.
- The *tag* of the sent message must be that specified by the tag argument.
- If there are many messages with identical tag from the same process, then any one of these messages is received.
- MPI allows specification of wild card arguments for both source and tag.
 - If source is set to *MPI_ANY_SOURCE*, then any process of the communication domain can be the source of the message.
 - Similarly, if tag is set to *MPI_ANY_TAG*, then messages with any tag are accepted.



MPI: the Message Passing Interface

Starting and Terminating
the MPI Library

Communicators

Getting Information

Sending and Receiving
Messages

Avoiding Deadlocks

Sending and Receiving
Messages Simultaneously

Sending and Receiving Messages VI

- The received message is stored in continuous locations in the buffer pointed to by *buf*.
- The *count* and *datatype* arguments of **MPI_Recv** are used to specify the length of the supplied buffer.
- The received message should be of length equal to or less than this length.
- This allows the receiving process to not know the exact size of the message being sent.
- If the received message is larger than the supplied buffer, then an overflow error will occur, and the routine will return the error *MPI_ERR_TRUNCATE*.



Sending and Receiving Messages VII

- After a message has been received, the status variable can be used to get information about the **MPI_Recv** operation.
- In C, status is stored using the *MPI_Status* data-structure.
- This is implemented as a structure with three fields, as follows:

```
typedef struct MPI_Status {  
    int MPI_SOURCE;  
    int MPI_TAG;  
    int MPI_ERROR;  
};
```

- *MPI_SOURCE* and *MPI_TAG* store the source and the tag of the received message.
- They are particularly useful when *MPI_ANY_SOURCE* and *MPI_ANY_TAG* are used for the source and tag arguments.
- *MPI_ERROR* stores the error-code of the received message.





- The status argument also returns information about the length of the received message.
- This information is not directly accessible from the status variable, but it can be retrieved by calling the **MPI_Get_count** function.
- The calling sequence:

```
int MPI_Get_count(MPI_Status *status,  
                 MPI_Datatype datatype,  
                 int *count)
```

- **MPI_Get_count** takes as arguments the status returned by **MPI_Recv** and the type of the received data in *datatype*, and returns the number of entries that were actually received in the *count* variable.



- The **MPI_Recv** returns only after the requested message has been received and copied into the buffer.
- That is, **MPI_Recv** is a **blocking** receive operation.
- However, MPI allows two different implementations for **MPI_Send**.
 - 1 **MPI_Send** returns only after the corresponding **MPI_Recv** have been issued and the message has been sent to the receiver.
 - 2 **MPI_Send** first copies the message into a **buffer** and then returns, without waiting for the corresponding **MPI_Recv** to be executed.
- In either implementation, the buffer that is pointed by the *buf* argument of **MPI_Send** *can be safely reused and overwritten*.

Sending and Receiving Messages X



- MPI programs must be able to run correctly regardless of which of the two methods is used for implementing **MPI_Send**.
- Such programs are called safe.
- In writing safe MPI programs, sometimes it is helpful to forget about the alternate implementation of **MPI_Send** and just think of it as being a blocking send operation.

MPI: the Message Passing Interface

Starting and Terminating
the MPI Library

Communicators

Getting Information

Sending and Receiving
Messages

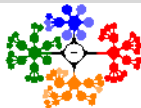
Avoiding Deadlocks

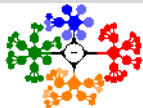
Sending and Receiving
Messages Simultaneously

Avoiding Deadlocks I

- The semantics of **MPI_Send** and **MPI_Recv** place some restrictions on how we can mix and match send and receive operations.
- Consider the following not complete code in which process 0 sends two messages with different tags to process 1, and process 1 receives them in the reverse order.

```
1  int a[10], b[10], myrank;
2  MPI_Status status;
3  ...
4  MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
5  if (myrank == 0) {
6      MPI_Send(a, 10, MPI_INT, 1, 1, MPI_COMM_WORLD);
7      MPI_Send(b, 10, MPI_INT, 1, 2, MPI_COMM_WORLD);
8  }
9  else if (myrank == 1) {
10     MPI_Recv(b, 10, MPI_INT, 0, 2, MPI_COMM_WORLD);
11     MPI_Recv(a, 10, MPI_INT, 0, 1, MPI_COMM_WORLD);
12 }
13 ...
```



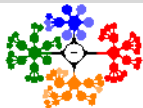


- If **MPI_Send** is implemented using buffering, then this code will run correctly (if sufficient buffer space is available).
- However, if **MPI_Send** is implemented by blocking until the matching receive has been issued, then neither of the two processes will be able to proceed.
- This code fragment is not safe, as its behavior is implementation dependent.
- It is up to the programmer to ensure that his or her program will run correctly on any MPI implementation.
- The problem in this program can be corrected by matching the order in which the send and receive operations are issued.
- Similar deadlock situations can also occur when a process sends a message to itself.

Avoiding Deadlocks III

- Improper use of **MPI_Send** and **MPI_Recv** can also lead to deadlocks in situations when each processor needs to send and receive a message in a circular fashion.
- Consider the following not complete code, in which
 - process i sends a message to process $i + 1$ (modulo the number of processes),
 - process i receives a message from process $i - 1$ (module the number of processes).

```
1  int a[10], b[10], npes, myrank;
2  MPI_Status status;
3  ...
4  MPI_Comm_size(MPI_COMM_WORLD, &npes);
5  MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
6  MPI_Send(a, 10, MPI_INT, (myrank+1)%npes, 1,
           MPI_COMM_WORLD);
7  MPI_Recv(b, 10, MPI_INT, (myrank-1+npes)%npes, 1,
           MPI_COMM_WORLD);
8  ...
```





- When **MPI_Send** is implemented using buffering, the program will work correctly,
 - since every call to **MPI_Send** will get buffered, allowing the call of the **MPI_Recv** to be performed, which will transfer the required data.
- However, if **MPI_Send** blocks until the matching receive has been issued,
 - all processes will enter an infinite wait state, waiting for the neighbouring process to issue a **MPI_Recv** operation.
- Note that the deadlock still remains even when we have only two processes.
- Thus, when pairs of processes need to exchange data, the above method leads to an unsafe program.

MPI: the Message Passing Interface

Starting and Terminating
the MPI Library

Communicators

Getting Information

Sending and Receiving
Messages

Avoiding Deadlocks

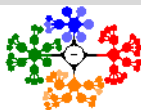
Sending and Receiving
Messages Simultaneously

Avoiding Deadlocks V

- The above example can be made safe, by rewriting it as follows:

```
1  int a[10], b[10], npes, myrank;
2  MPI_Status status;
3  ...
4  MPI_Comm_size(MPI_COMM_WORLD, &npes);
5  MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
6  if (myrank%2 == 1) {
7      MPI_Send(a, 10, MPI_INT, (myrank+1)%npes, 1,
               MPI_COMM_WORLD);
8      MPI_Recv(b, 10, MPI_INT, (myrank-1+npes)%npes, 1,
               MPI_COMM_WORLD);
9  }
10 else {
11     MPI_Recv(b, 10, MPI_INT, (myrank-1+npes)%npes, 1,
              MPI_COMM_WORLD);
12     MPI_Send(a, 10, MPI_INT, (myrank+1)%npes, 1,
              MPI_COMM_WORLD);
13 }
14 ...
```

- This new implementation partitions the processes into two groups.
- One consists of the odd-numbered processes and the other of the even-numbered processes.



Sending and Receiving Messages Simultaneously I

- The above communication pattern appears frequently in many message-passing programs,
- For this reason MPI provides the **MPI_Sendrecv** function that both sends and receives a message.
- **MPI_Sendrecv** does not suffer from the circular deadlock problems of **MPI_Send** and **MPI_Recv**.
- You can think of **MPI_Sendrecv** as allowing data to travel for both send and receive simultaneously.
- The calling sequence of **MPI_Sendrecv** is as the following:

```
int MPI_Sendrecv(void *sendbuf, int sendcount,
                 MPI_Datatype senddatatype, int dest,
                 int sendtag,
                 void *recvbuf, int recvcount,
                 MPI_Datatype recvdatatype, int source,
                 int recvtag, MPI_Comm comm,
                 MPI_Status *status)
```

- The arguments of **MPI_Sendrecv** are essentially the combination of the arguments of **MPI_Send** and **MPI_Recv**.



MPI: the Message Passing Interface

Starting and Terminating
the MPI Library

Communicators

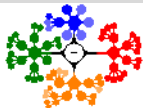
Getting Information

Sending and Receiving
Messages

Avoiding Deadlocks

Sending and Receiving
Messages Simultaneously

Sending and Receiving Messages Simultaneously II



- The send and receive buffers must be disjoint, and the source and destination of the messages can be the same or different.
- The safe version of our previous example using **MPI_Sendrecv** is as the following;

```
1  int a[10], b[10], npes, myrank;
2  MPI_Status status;
3  ...
4  MPI_Comm_size(MPI_COMM_WORLD, &npes);
5  MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
6  MPI_Sendrecv(a, 10, MPI_INT, (myrank+1)%npes, 1,
7              b, 10, MPI_INT, (myrank-1+npes)%npes, 1,
8              MPI_COMM_WORLD, &status);
9  ...
```

MPI: the Message Passing Interface

Starting and Terminating
the MPI Library

Communicators

Getting Information

Sending and Receiving
Messages

Avoiding Deadlocks

Sending and Receiving
Messages Simultaneously

Sending and Receiving Messages Simultaneously III



- In many programs, the requirement for the send and receive buffers of **MPI_Sendrecv** be disjoint may force us to use a temporary buffer.
- This increases the amount of memory required by the program and also increases the overall run time due to the extra copy.
- This problem can be solved by using that **MPI_Sendrecv_replace** MPI function.
- This function performs a blocking send and receive, but it uses a single buffer for both the send and receive operation.
- That is, the received data replaces the data that was sent out of the buffer.

MPI: the Message Passing Interface

Starting and Terminating
the MPI Library

Communicators

Getting Information

Sending and Receiving
Messages

Avoiding Deadlocks

Sending and Receiving
Messages Simultaneously