MPI – Message Passing Interface

CHAPTER 4

Outline

1. **MPI Overview**
2. Basic Structure of a MPI program
3. Messages and Point-to-Point Communication
4. Non-blocking Communication
5. Collective Communication
6. Derived Data Types

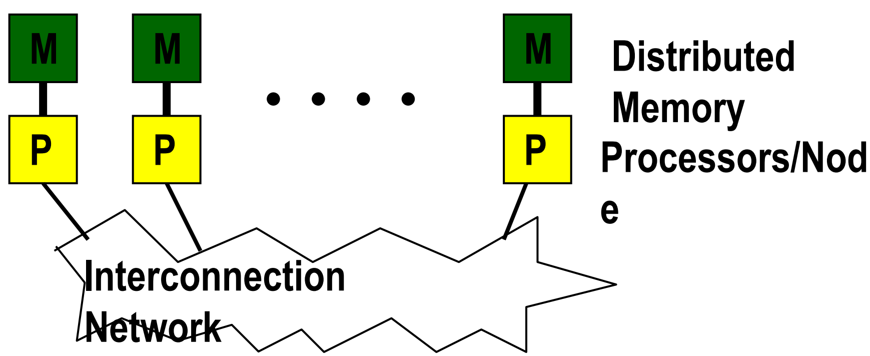
MPI Overview

OpenMP can only be used on **shared memory** systems with a **single address space** used by all threads.

Distributed memory systems require a different approach. (clusters of computers, supercomputers, heterogeneous Networks)

MPI - Message Passing Programming Paradigm:

- written in a conventional sequential language, e.g., C or Fortran.
- SPMD model: All processors execute same program, but with different data.
- Program manages memory by placing data in processes (all variables are private).
- Data must be shared via special send&receive routines (***message passing***)

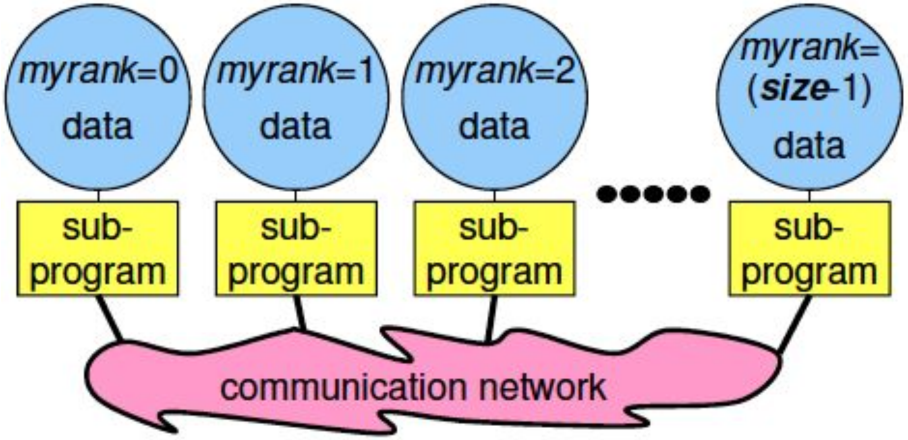


MPI Overview

The value of ***myrank*** identifies each sub-program and is returned by special library routines.

The ***size*** (number of processors) is started by special MPI initialization program (mpirun or mpiexec)

All distribution decisions are based on ***myrank***



MPI Overview

- Parallel programs typically implement SPMD (**S**ingle **P**rogram **M**ultiple **D**ata)
  Same sub-program runs on each processor

```
int main (int argc, char **argv)
{
    . . .
    if (myrank == 0) {
        /*  Master  role:  data  distribution,  collect  results,
            workers coordination, etc. */
        Master_function(/*arguments*/);
    }
   else {
        /* Worker role */
        Worker_function(/*arguments*/);
    }
}
```

MPI Overview

- MPI allows also MPMD (**M**ultiple **P**rogram **M**ultiple **D**ata)
  MPMD can be emulated by SPMD

```
int main (int argc, char **argv)
{
    . . .
    if (myrank == 0) {
        /* process should run the function 0 */
        function_0(/*arguments*/);
    }
    else if (myrank == i) {
        /* process should run the function i */
        function_i(/*arguments*/);
    }
    else {
        function_n (/*arguments*/);
    }
}
```

MPI Overview

- Access:
  - A message passing system is similar to:
    Post-office, Phone line, Fax, E-mail, etc.
  - Each sub-program needs to be connected to the same message passing system Each sub-program must be started with the MPI start-up tool.
- Addressing:
  - Messages need to have addresses to be sent to
  - MPI addresses are ranks of the MPI processes (sub-programs)
- Reception:
  - It is important that the receiving processes be able to manage and receive all sent messages
- Communication models:
  - Point-to-Point, Collective, Synchronous (telephone)/Asynchronous (Postal)

MPI Overview

Benefits

+ No new language is required

+ Source-code portability

+ Efficient implementations

Disadvantages

- Explicitly forces programmer to deal with local/global access

- Harder to program than shared memory – requires larger program/algorithm changes

- Harder debugging

Outline

1. MPI Overview
2. **Basic Structure of a MPI program**
3. Messages and Point-to-Point Communication
4. Non-blocking Communication
5. Collective Communication
6. Derived Data Types

Basic Structure of an MPI program

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

int main( int argc, char **argv )
{
    int nproc,iproc;
    MPI_Init( &argc, &argv );

    printf( "Hello World\n" );

    MPI_Finalize();
    return 0;
}
```

Always need to	mpi.h
Start with	MPI_Init()
End with	MPI_Finalize()

* **MPI_** namespace is reserved for MPI constants and routines.

Basic Structure of an MPI program

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

int main( int argc, char **argv )
{
     int nproc,iproc;
     MPI_Init( &argc, &argv );

     if (myrank == 0) {
          /* process should run the function 0 */
          function_0(/*arguments*/);
     }
     else if (myrank == i) {
          /* process should run the function i */
          function_i(/*arguments*/);
     }
    else {
          function_n (/*arguments*/);
     }

     MPI_Finalize();
     return 0;
}
```

Basic Structure of an MPI program

The **mpi module "mpi.h"** includes

Subroutines such as `mpi_init`, `mpi_comm_size`, `mpi_comm_rank`, …

Handles: Global variables which refer to internal MPI data structures

`MPI_COMM_WORLD`: a communicator
`MPI_INTEGER`: a type used in a MPI_routine
`MPI_SUM`: used to specify a type of reduction operation
etc.

The object accessed by the predefined constant handle exist and does not change only between `MPI_Init()` and `MPI_Finalize()`

Basic Structure of an MPI program

```
int MPI_Init(int *argc, char **argv)
```

- Must be the first MPI routine that is called.

```
int MPI_Finalize()
```

- Must be called last by all processes
- User must ensure the completion of all pending communications (locally) before calling finalize
- After MPI_Finalize:
  - Further MPI-calls are forbidden
  - Re-initialization with **MPI_Init** is forbidden
  - May abort all processes except “rank==0” in MPI_COMM_WORLD

Basic Structure of an MPI program

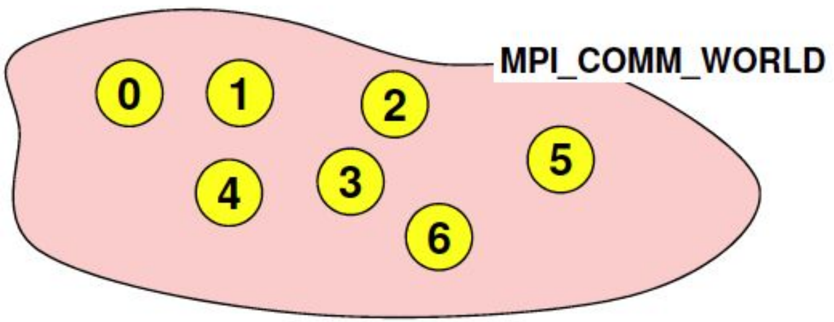
- All processes (=sub-programs) of one MPI program are combined in the communicator **MPI_COMM_WORLD**.
- MPI_COMM_WORLD is a predefined **handle**

```
MPI_Comm_rank(MPI_COMM_WORLD, &iproc);
```

  - the **rank** identifies different processes
  - the **rank** is the basis for any work an data distribution
  - the **rank** is in the range 0 to size-1

```
MPI_Comm_size(MPI_COMM_WORLD, &nproc);
```

  - The **size** is the number of processes within a communicator



Basic Structure of an MPI program

Compilation and Execution

• **Compilation of a MPI program:**	mpicc (for programms in C) mpif77 (Fortran 77). mpif90 (Fortran 90) mpiCC (C++)
• **Program start on num PEs:**	mpiexec -n ***num*** prog_exec (***num*** = number of processes)

Exemple:

```
$mpicc -o hello hello.c
$mpiexec -n 4 hello
```

For more complicated applications, we can also use a Makefile.

Basic Structure of an MPI program

Activity 1: Hello World

1. Write a minimal MPI program which prints “Hello World!” by each MPI process

    – Every process writes its **rank** and the **size** of MPI_COMM_WORLD

    – Only **process 0** in MPI_COMM_WORLD **prints “Hello World!”**

    – Run it on several processors in parallel

Outline

1. MPI Overview
2. Basic Structure of a MPI program
3. **Messages and Point-to-Point Communication**
4. Non-blocking Communication
5. Collective Communication
6. Derived Data Types

Messages and Point-to-Point Communication

Messages

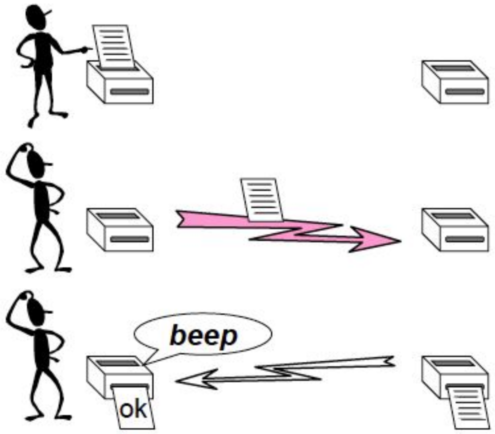
- Messages are sent by packing up all necessary data, this packages are referred as envelopes.
    - **Destination** (Send)
        - to route the message to the appropriate process
    - **Source** (Recv)
        - indicates the message source process, only messages coming from that source can be accepted
        - MPI_ANY_SOURCE to receive from any resource
    - **tag**
        - distinguish messages received from a single process
        - user-specified integer in the range (0-32567)
        - MPI_ANY_TAG to receive from any tag
    - **communicator**
        - Both communicators must be identical

Messages and Point-to-Point Communication

Point-to-Point Communication Variations

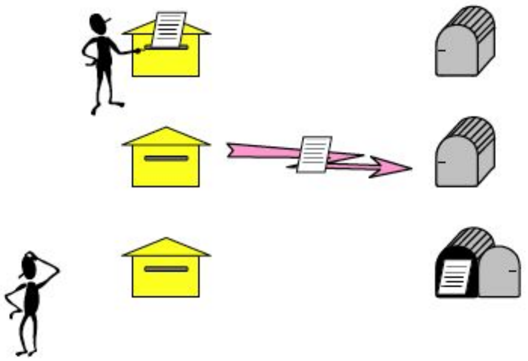
•synchronous send

- The sender gets an information that the message is received.
  (Analogue to the beep or okay-sheet of a fax)



•buffered = asynchronous send

- Only know when the message has left
  (Analoge to the postal service)

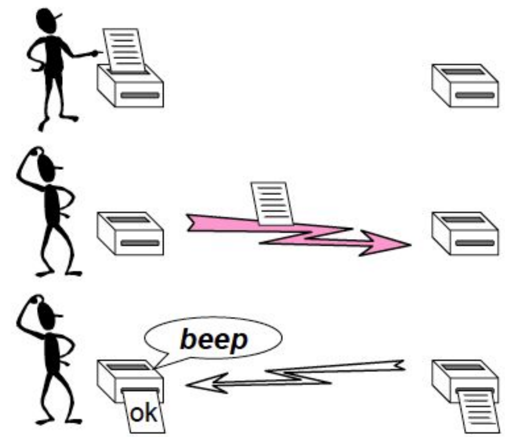


Messages and Point-to-Point Communication

Point-to-Point Communication Variations

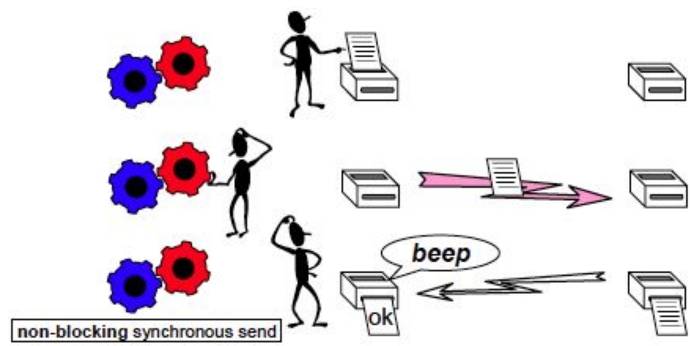
•Blocking operations

- only return from the call when operation has completed
  - synchronous **send** operation blocks until receive is posted;
  - **receive** operation blocks until message is sent.



•Non-blocking operations

- Non-blocking operation: returns immediately and allow the sub-program to perform other work.
- At some later time the sub-program must check the completion of the non-blocking operation.
- A non-blocking operation immediately followed by a matching wait is equivalent to a blocking operation.



Messages and Point-to-Point Communication

Point-to-Point Communication – Communication modes

Mode	Definition	Notes
Standard send **MPI_SEND**	Either synchronous or buffered	depends on the implementation
Synchronous send **MPI_SSEND**	Only completes when the receive has started	
Buffered send **MPI_BSEND**	Always completes irrespective of receiver	Needs application-defined buffer to be declared with **MPI_BUFFER_ATTACH**
Ready send **MPI_RSEND**	May be started only if the matching receive is already posted	May be the fastest. But highly dangerous! You must guarantee that Recv is already called.

Receive **MPI_RECV**	Completes when a message has arrived	same routine for all communication modes
Receive **MPI_IRECV**	Does not wait for the messages	To know if the message has been received, you must use MPI_Wait or MPI_Test

Messages and Point-to-Point Communication

Point-to-Point Communication – Communication modes issues

- Standard send (**MPI_SEND**)
  - May issue deadlocks
- Synchronous send (**MPI_SSEND**)
  - Risk of deadlock
  - Risk of serialization
  - Risk of waiting (idle time)
  - High latency / best bandwidth
- Buffered send (**MPI_BSEND**)
  - Low latency/bad bandwidth
- Ready send (**MPI_RSEND**)
  - May be the fastest
  - Highly dangerous

Messages and Point-to-Point Communication

Point-to-Point Communication – STANDARD SEND

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

`buf`	the address of the data to be sent
`count`	the number of elements of datatype
`datatype`	the MPI datatype
`dest`	rank of destination in communicator comm
`tag`	a marker used to distinguish different message types
`comm`	the communicator shared by sender and receiver

Messages and Point-to-Point Communication

Messages

- A message contains a number of elements of some particular datatype.
- MPI datatype
  - Basic datatype
  - Derived datatypes
- Datatype handles are used to describe the type of data in the memory.
- Derived datatypes can be built up from basic or derived datatypes.

MPI Basic Datatypes

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_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOBLE	long double
MPI_BYTE	
MPI_PACKED	

Messages and Point-to-Point Communication

Point-to-Point Communication – STANDARD BLOCKING RECEIVE

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

- ***`Buf/count/datatype`*** describe the receive buffer
- Receiving the message sent by process with rank ***source*** in ***comm***.
- Envelope information is returned in ***status***
- One can pass MPI_STATUS_IGNORE instead of a status argument
- Only messages with matching ***tag*** are received

Messages and Point-to-Point Communication

Point-to-Point Communication - MPI_STATUS struct

hpc-course/Examples/status_struct.c

- Envelope information is returned from MPI_RECV in *status*

```
MPI_Status status;
status.MPI_SOURCE
status.MPI_TAG
status.MPI_ERROR
```

- In the case that the size of the message was not the same that the size of the buffer, the number of elements received can be obtained by

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

The parameter *count* gives back the number of elements received of the type datatype.

Messages and Point-to-Point Communication

Point-to-Point Communication – For a communication to succeed:

•Sender must specify a valid destination rank

•Receiver must specify a valid source rank

•The communicator must be the same

•Tags must match

•Buffer's type must match with the datatype handle

•Message datatypes must match

•Receivers buffer must be large enough

Messages and Point-to-Point Communication

Activity 2: Point-to-Point communication “Hello World”

1. Write a program in which each process send a message to process 0 :

   "Hello, world. I'am <rank> of <numproc> on <name>"

2. The proces 0 will print the same message for itself and for all messages received:

   "Received from <source> : <message received>

3. Make the output deterministic

Messages and Point-to-Point Communication

Activity 3 (2): Ping pong

1. Write a program in which two processes repeatedly pass a message back (ping) and forth (pong).

2. Repeat this ping-pong with a loop of length 50

3. Add timing calls to measure the time taken for one message.

4. Investigate how the time taken to exchange messages varies with the size of the message
   - 8 bytes (1 double), 512 bytes (64 double), 32Kbytes (4096 double), 2 Mbytes (262144 double)

5. Print out the following results
   - Print out the transfer time of one message: total_time/(2*50)*1e3
   - Print out the Bandwidth: message size (in bytes) / transfer time

6. Exclude startup time problems from measurements
   - Execute a first ping-pong outside of the measurement loop

Explain the results.

(To get the time use MPI_Wtime();)