Distributed Computing

MPI

Mahmoud Badry, mma18@fayoum.edu.eg

AGENDA

1. MPICH
2. PTP communication
3. PTP examples
4. PTP deadlocks
5. End
   Questions

MPICH

Knowing and installing MPICH

Mahmoud Badry, mma18@fayoum.edu.eg

MPICH (www.mpich.org)

- MPICH is a **freely** available, **portable** implementation of **MPI**, a standard for message-passing for **distributed-memory** applications used in **parallel** computing.
- The CH part of the name was derived from "**Chameleon**", which was a portable parallel programming library developed by **William Gropp**, one of the **founders** of MPICH.
- History:
  - **Before 2001**: MPICH1 which implements MPI-1
  - **Between 2001-2012**: MPICH2 which implements MPI-2
  - **After 2012**: MPICH v3.0 which implements MPI-3

Mahmoud Badry, mma18@fayoum.edu.eg

MPICH installation (For our lab)

- MPICH installation requirements:
  - Windows XP.
  - .NET framework 2.0.
  - VS2005.
- To install MPICH in your machine use the instructions that can be found in this link: http://www.cs.utah.edu/~delisi/vsmpi/
- After installation and compiling your MPI use this command on CMD:
  - `mpiexec -np 4 yourprogram.exe`

Mahmoud Badry, mma18@fayoum.edu.eg

PTP Communication

Point to point commuincation in MPI

Communicators

- In MPI, all communication operations are executed using a **communicator**. A communicator represents a communication domain which is essentially a **set** of **processes** that exchange messages between each other.
- The MPI default communicator MPI COMM WORLD is used for the communication. This **communicator captures** all **processes** executing a **parallel** program

Mahmoud Badry, mma18@fayoum.edu.eg

MPI_Send operation

- ```
  int MPI_Send(void *smessage,
               int count,
               MPI_Datatype datatype,
               int dest,
               int tag,
               MPI_Comm comm).
  ```
- **smessage** specifies a send buffer which contain the data elements.
- **count** is the number of elements to be sent from the send buffer.
- **datatype** is the datatype of each entry in the send buffer.
- **dest** specifies the rank of the target process.
- **tag** is a message tag which can be used by the receiver to distinguish different messages from the same sender.
  ```

MPI_Recv operation

- ```
  int MPI_Recv(void *rmessage,
               int count,
               MPI_Datatype datatype,
               int source,
               int tag,
               MPI_Comm comm,
               MPI_Status *status).
  ```
- Like MPI_Send except that:
  - **rmessage** specifies the receive buffer in which the message should be stored.
  - **status** specifies a data structure which contains information about a message after the completion of the receive operation. Can be ignored using MPI_STATUS_IGNORE

Mahmoud Badry, mma18@fayoum.edu.eg
  ```

MPI_Recv operation Cont.

- Like MPI_Send except that:
  - By using **`source`** `= MPI_ANY_SOURCE`, a process can receive a message from any arbitrary process.
  - Similarly, by using **`tag`** `= MPI_ANY_TAG`, a process can receive a message with an arbitrary tag.
- After completion of the receive operation, status variable will contain these information:
  - status.MPI_SOURCE specifies the rank of the sending process.
  - status.MPI_TAG specifies the tag of the message received.
  - status.MPI_ERROR contains an error code.

MPI Datatypes

MPI Datentyp	C-Datentyp
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_LONG_LONG_INT	long long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_UNSIGNED_LONG_LONG	unsigned long long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_WCHAR	wide char
MPI_PACKED	special data type for packing
MPI_BYTE	single byte value

What happends while sending and receiving ?

1. The data elements to be sent are **copied** from the send buffer **smessage** specified as parameter into a **system buffer** of the MPI runtime system. The **message** is **assembled** by adding a **header** with information on the **sending process**, the **receiving process**, the **tag**, and the **communicator** used.
2. The **message** is sent via the **network** from the **sending** process to the **receiving** process.
3. At the **receiving** side, the data entries of the **message** are **copied** from the **system buffer** into the **receive buffer** rmessage specified by MPI_Recv().

Mahmoud Badry, mma18@fayoum.edu.eg

Send and receive operations nature

- Both MPI_Send() and MPI_Recv() are blocking, asynchronous operations.
  - MPI_Recv() operation can also be **started** when the **corresponding** MPI_Send() operation has not yet been **started**.
  - The **process** executing the MPI_Recv() operation is **blocked** until the specified receive **buffer** contains the data elements sent.
  - Similarly, an MPI_Send() **operation** can also be **started** when the corresponding MPI_Recv() **operation** has not yet been **started**.

Mahmoud Badry, mma18@fayoum.edu.eg

E

Point to point operation examples

Mahmoud Badry, mma18@fayoum.edu.eg

Deadlocks

Deadlocks with Point-to-Point Communications

Mahmoud Badry, mma18@fayoum.edu.eg

Deadlocks with Point-to-Point Communications

- Send and receive operations must be used with care, since deadlocks can occur in ill-constructed programs.

```
/* program fragment which always causes a deadlock */
MPI_Comm_rank (comm, &my_rank);
if (my_rank == 0) {
   MPI_Recv (recvbuf, count, MPI_INT, 1, tag, comm, &status);
   MPI_Send (sendbuf, count, MPI_INT, 1, tag, comm);
}
else if (my_rank == 1) {
   MPI_Recv (recvbuf, count, MPI_INT, 0, tag, comm, &status);
   MPI_Send (sendbuf, count, MPI_INT, 0, tag, comm);
}
```


Mahmoud Badry, mma18@fayoum.edu.eg

Deadlocks with Point-to-Point Communications cont.

```
/* program fragment for which the occurrence of a deadlock
   depends on the implementation */
MPI_Comm_rank (comm, &my_rank);
if (my_rank == 0) {
   MPI_Send (sendbuf, count, MPI_INT, 1, tag, comm);
   MPI_Recv (recvbuf, count, MPI_INT, 1, tag, comm, &status);
}
else if (my_rank == 1) {
   MPI_Send (sendbuf, count, MPI_INT, 0, tag, comm);
   MPI_Recv (recvbuf, count, MPI_INT, 0, tag, comm, &status);
}
```

- Message transmission is performed correctly here without deadlock, if the MPI runtime system uses system buffers. But a deadlock occurs, if the runtime system does not use system buffers or if the system buffers used are too small.

Secure implementation

- A secure implementation which does not cause deadlocks even if no system buffers are used.

```
/* program fragment that does not cause a deadlock */
MPI_Comm_rank (comm, &myrank);
if (my_rank == 0) {
   MPI_Send (sendbuf, count, MPI_INT, 1, tag, comm);
   MPI_Recv (recvbuf, count, MPI_INT, 1, tag, comm, &status);
}
else if (my_rank == 1) {
   MPI_Recv (recvbuf, count, MPI_INT, 0, tag, comm, &status);
   MPI_Send (sendbuf, count, MPI_INT, 0, tag, comm);
}
```


Mahmoud Badry, mma18@fayoum.edu.eg

Secure implementation Cont.

- An MPI program is called secure if the correctness of the program does not depend on assumptions about specific properties of the MPI runtime system, like the existence of system buffers or the size of system buffers.

Mahmoud Badry, mma18@fayoum.edu.eg

Send and recieve operation?!

- Suppose that we made a program which has some processes. Each process is sending and receiving a message in the same time. If we've imagined a secure implementation for this, the phases would be like this:

Phase	Process 0	Process 1	Process 2
1	MPI_Send() to 1	MPI_Recv() from 0	MPI_Send() to 0
2	MPI_Recv() from 2	MPI_Send() to 2	-wait-
3		-wait-	MPI_Recv() from 1

Mahmoud Badry, mma18@fayoum.edu.eg

MPI_Sendrecv operation

- ```
  int MPI Sendrecv (void *sendbuf,
                    int sendcount,
                    MPI Datatype sendtype,
                    int dest,
                    int sendtag,
                    void *recvbuf,
                    int recvcount,
                    MPI Datatype recvtype,
                    int source,
                    int recvtag,
                    MPI Comm comm,
                    MPI Status *status);
  ```


Mahmoud Badry, mma18@fayoum.edu.eg
  ```

MPI_Sendrecv operation cont.

- Using MPI_Sendrecv(), the programmer does not need to worry about the order of the send and receive operations. The MPI runtime system guarantees deadlock freedom, also for the case that no internal system buffers are used.
- The parameters **sendbuf** and **recvbuf**, specifying the send and receive buffers of the executing process, must be disjoint, non-overlapping memory locations.
- There is a variant of MPI_Sendrecv() for which the send buffer and the receive buffer are identical.

Mahmoud Badry, mma18@fayoum.edu.eg

MPI_Sendrecv_replace operation

```
int MPI_Sendrecv_replace (void *buffer,
                          int count,
                          MPI Datatype type,
                          int dest,
                          int sendtag,
                          int source,
                          int recvtag,
                          MPI Comm comm,
                          MPI Status *status).
```

- Here, buffer specifies the buffer that is used as both send and receive buffer. For this function, count is the number of elements to be sent and to be received; these elements now should have identical type.

E

Running „Send recieve operation" example

Mahmoud Badry, mma18@fayoum.edu.eg

?

QUESTIONS

Mahmoud Badry, mma18@fayoum.edu.eg

THANK YOU!

Mahmoud Badry, mma18@fayoum.edu.eg