Elementary MPI

Sayan Ghosh

Washington State University
EECS

2 Mar 2015
(HPC) Compute clusters

- Tightly knit cluster of computers with special-purpose networking cards for optimal network latency and bandwidth for data communication.

Homogeneous system

Distributed computers

Commodity computers connected via internet – cost effective

Heterogeneous system – End-points may differ substantially – one end-point could be a drone in Saturn, Solar System and another a laptop in Houston, TX

Performance might not be as important as fault tolerance (h/w and s/ws are built to provide multi-layered fault tolerance in certain cases)

Middleware rules! Know a middleware, forget about underlying system level tweaks.
(HPC) Compute clusters

- Tightly knit cluster of computers with special-purpose networking cards for optimal network latency and bandwidth for data communication
- Typically runs some variant of GNU/Linux as O/S

Distributed computers

- Commodity computers connected via internet — cost effective
- Heterogeneous system — End-points may differ substantially — one end-point could be a drone in Saturn, Solar System and another a laptop in Houston, TX
- Performance might not be as important as fault tolerance (h/w and s/ws are built to provide multi-layered fault tolerance in certain cases)

Middleware rules! Know a middleware, forget about underlying system level tweaks
(HPC) Compute clusters

- Tightly knit cluster of computers with special-purpose networking cards for optimal network latency and bandwidth for data communication
- Typically runs some variant of GNU/Linux as O/S
- Node structure – 4-16 multicore processors, 12 MB L3-cache, at 16-64 GB of main memory, co-processor cards such as Nvidia GPUs or Intel Xeon Phi
- Many of these systems are diskless
- Mandates an optimal cooling infrastructure
- Parallel programming skills a must to work on such a machine

Distributed computers

- Commodity computers connected via internet – cost effective
- Heterogeneous system – End-points may differ substantially – one end-point could be a drone in Saturn, Solar System and another a laptop in Houston, TX
- Performance might not be as important as fault tolerance (h/w and s/ws are built to provide multi-layered fault tolerance in certain cases)
- Middleware rules! Know a middleware, forget about underlying system level tweaks
(HPC) Compute clusters

- Tightly knit cluster of computers with special-purpose networking cards for optimal network latency and bandwidth for data communication.
- Typically runs some variant of GNU/Linux as O/S.
- Node structure – 4-16 multicore processors, 12 MB L3-cache, at 16-64 GB of main memory, co-processor cards such as Nvidia GPUs or Intel Xeon Phi.
- Many of these systems are diskless.
(HPC) Compute clusters

- Tightly knit cluster of computers with special-purpose networking cards for optimal network latency and bandwidth for data communication
- Typically runs some variant of GNU/Linux as O/S
- Node structure – 4-16 multicore processors, 12 MB L3-cache, at 16-64 GB of main memory, co-processor cards such as Nvidia GPUs or Intel Xeon Phi
- Many of these systems are diskless
- Mandates an optimal cooling infrastructure
(HPC) Compute clusters

- Tightly knit cluster of computers with special-purpose networking cards for optimal network latency and bandwidth for data communication
- Typically runs some variant of GNU/Linux as O/S
- Node structure – 4-16 multicore processors, 12 MB L3-cache, at 16-64 GB of main memory, co-processor cards such as Nvidia GPUs or Intel Xeon Phi
- Many of these systems are diskless
- Mandates an optimal cooling infrastructure
- Parallel programming skills a must to work on such a machine

Distributed computers

- Commodity computers connected via internet – cost effective
- Heterogeneous system – End-points may differ substantially – one end-point could be a drone in Saturn, Solar System and another a laptop in Houston, TX
- Performance might not be as important as fault tolerance (h/w and s/ws are built to provide multi-layered fault tolerance in certain cases)

Middleware rules! Know a middleware, forget about underlying system level tweaks
Compute clusters and distributed computers

(HPC) Compute clusters

- Tightly knit cluster of computers with special-purpose networking cards for optimal network latency and bandwidth for data communication
- Typically runs some variant of GNU/Linux as O/S
- Node structure – 4-16 multicore processors, 12 MB L3-cache, at 16-64 GB of main memory, co-processor cards such as Nvidia GPUs or Intel Xeon Phi
- Many of these systems are diskless
- Mandates an optimal cooling infrastructure
- Parallel programming skills a must to work on such a machine
- Homogeneous system

Distributed computers

- Commodity computers connected via internet – cost effective
- Heterogeneous system – End-points may differ substantially – one end-point could be a drone in Saturn, Solar System and another a laptop in Houston, TX
- Performance might not be as important as fault tolerance (h/w and s/ws are built to provide multi-layered fault tolerance in certain cases)
- Middleware rules! Know a middleware, forget about underlying system level tweaks
(HPC) Compute clusters

- Tightly knit cluster of computers with special-purpose networking cards for optimal network latency and bandwidth for data communication
- Typically runs some variant of GNU/Linux as O/S
- Node structure – 4-16 multicore processors, 12 MB L3-cache, at 16-64 GB of main memory, co-processor cards such as Nvidia GPUs or Intel Xeon Phi
- Many of these systems are diskless
- Mandates an optimal cooling infrastructure
- Parallel programming skills a must to work on such a machine
- Homogeneous system
Compute clusters and distributed computers

(HPC) Compute clusters
- Tightly knit cluster of computers with special-purpose networking cards for optimal network latency and bandwidth for data communication
- Typically runs some variant of GNU/Linux as O/S
- Node structure – 4-16 multicore processors, 12 MB L3-cache, at 16-64 GB of main memory, co-processor cards such as Nvidia GPUs or Intel Xeon Phi
- Many of these systems are diskless
- Mandates an optimal cooling infrastructure
- Parallel programming skills a *must* to work on such a machine
- Homogeneous system

Distributed computers
- Commodity computers connected via internet – cost effective
(HPC) Compute clusters

- Tightly knit cluster of computers with special-purpose networking cards for optimal network latency and bandwidth for data communication
- Typically runs some variant of GNU/Linux as O/S
- Node structure – 4-16 multicore processors, 12 MB L3-cache, at 16-64 GB of main memory, co-processor cards such as Nvidia GPUs or Intel Xeon Phi
- Many of these systems are diskless
- Mandates an optimal cooling infrastructure
- Parallel programming skills a *must* to work on such a machine
- Homogeneous system

Distributed computers

- Commodity computers connected via internet – cost effective
- Heterogeneous system – End-points may differ substantially – one end-point could be a drone in Saturn, Solar System and another a laptop in Houston, TX
Compute clusters and distributed computers

(HPC) Compute clusters

- Tightly knit cluster of computers with special-purpose networking cards for optimal network latency and bandwidth for data communication
- Typically runs some variant of GNU/Linux as O/S
- Node structure – 4-16 multicore processors, 12 MB L3-cache, at 16-64 GB of main memory, co-processor cards such as Nvidia GPUs or Intel Xeon Phi
- Many of these systems are diskless
- Mandates an optimal cooling infrastructure
- Parallel programming skills a must to work on such a machine
- Homogeneous system

Distributed computers

- Commodity computers connected via internet – cost effective
- Heterogeneous system – End-points may differ substantially – one end-point could be a drone in Saturn, Solar System and another a laptop in Houston, TX
- Performance might not be as important as fault tolerance (h/w and s/ws are built to provide multi-layered fault tolerance in certain cases)
(HPC) Compute clusters

- Tightly knit cluster of computers with special-purpose networking cards for optimal network latency and bandwidth for data communication
- Typically runs some variant of GNU/Linux as O/S
- Node structure – 4-16 multicore processors, 12 MB L3-cache, at 16-64 GB of main memory, co-processor cards such as Nvidia GPUs or Intel Xeon Phi
- Many of these systems are diskless
- Mandates an optimal cooling infrastructure
- Parallel programming skills a must to work on such a machine
- Homogeneous system

Distributed computers

- Commodity computers connected via internet – cost effective
- Heterogeneous system – End-points may differ substantially – one end-point could be a drone in Saturn, Solar System and another a laptop in Houston, TX
- Performance might not be as important as fault tolerance (h/w and s/ws are built to provide multi-layered fault tolerance in certain cases)
- Middleware rules! Know a middleware, forget about underlying system level tweaks
Compute clusters and distributed computers

(HPC) Compute clusters
- Tightly knit cluster of computers with special-purpose networking cards for optimal network latency and bandwidth for data communication
- Typically runs some variant of GNU/Linux as O/S
- Node structure – 4-16 multicore processors, 12 MB L3-cache, at 16-64 GB of main memory, co-processor cards such as Nvidia GPUs or Intel Xeon Phi
- Many of these systems are diskless
- Mandates an optimal cooling infrastructure
- Parallel programming skills a must to work on such a machine
- Homogeneous system

Distributed computers
- Commodity computers connected via internet – cost effective
- Heterogeneous system – End-points may differ substantially – one end-point could be a drone in Saturn, Solar System and another a laptop in Houston, TX
- Performance might not be as important as fault tolerance (h/w and s/ws are built to provide multi-layered fault tolerance in certain cases)
- Middleware rules! Know a middleware, forget about underlying system level tweaks
(HPC) Compute clusters

- Tightly knit cluster of computers with special-purpose networking cards for optimal network latency and bandwidth for data communication
- Typically runs some variant of GNU/Linux as O/S
- Node structure – 4-16 multicore processors, 12 MB L3-cache, at 16-64 GB of main memory, co-processor cards such as Nvidia GPUs or Intel Xeon Phi
- Many of these systems are diskless
- Mandates an optimal cooling infrastructure
- Parallel programming skills a *must* to work on such a machine
- Homogeneous system

Distributed computers

- Commodity computers connected via internet – cost effective
- Heterogeneous system – End-points may differ substantially – one end-point could be a drone in Saturn, Solar System and another a laptop in Houston, TX
- Performance might not be as important as fault tolerance (h/w and s/ws are built to provide multi-layered fault tolerance in certain cases)
- Middleware rules! Know a middleware, forget about underlying system level tweaks
What is MPI?

MPI

A **message-passing library interface specification** for massively parallel systems

- Few open source implementations of MPI are MPICH (Argonne Nat’l Lab), foMPI (ETH Zurich), MVAPICH (OSU), OpenMPI (Sandia Nat’l Lab, CISCO, IU, et al), FG-MPI (UBC), et al
What is MPI?

MPI

A *message-passing library interface specification* for massively parallel systems

- Few open source implementations of MPI are MPICH (Argonne Nat’l Lab), foMPI (ETH Zurich), MVAPICH (OSU), OpenMPI (Sandia Nat’l Lab, CISCO, IU, et al), FG-MPI (UBC), et al
- Some vendor implementations are Cray-MPI, IBM-MPI, Intel-MPI, SGI-MPI, MS-MPI (yes, there is one!)

Where is MPI used?

- Oil and Gas exploration
- Climate Modeling
- Biological sequence analysis
- Computer Animation
- Any large scale simulation
What is MPI?

MPI

A message-passing library interface specification for massively parallel systems

- Few open source implementations of MPI are MPICH (Argonne Nat’l Lab), foMPI (ETH Zurich), MVAPICH (OSU), OpenMPI (Sandia Nat’l Lab, CISCO, IU, et al), FG-MPI (UBC), et al

- Some vendor implementations are Cray-MPI, IBM-MPI, Intel-MPI, SGI-MPI, MS-MPI (yes, there is one!)

- Most MPI implementations are derivatives of MPICH [QED]
What is MPI?

A message-passing library interface specification for massively parallel systems

- Few open source implementations of MPI are MPICH (Argonne Nat’l Lab), foMPI (ETH Zurich), MVAPICH (OSU), OpenMPI (Sandia Nat’l Lab, CISCO, IU, et al), FG-MPI (UBC), et al
- Some vendor implementations are Cray-MPI, IBM-MPI, Intel-MPI, SGI-MPI, MS-MPI (yes, there is one!)
- Most MPI implementations are derivatives of MPICH [QED]
What is MPI?

**MPI**

A *message-passing library interface specification* for massively parallel systems

- Few open source implementations of MPI are MPICH (Argonne Nat’l Lab), foMPI (ETH Zurich), MVAPICH (OSU), OpenMPI (Sandia Nat’l Lab, CISCO, IU, et al), FG-MPI (UBC), et al
- Some vendor implementations are Cray-MPI, IBM-MPI, Intel-MPI, SGI-MPI, MS-MPI (yes, there is one!)
- Most MPI implementations are derivatives of MPICH [QED]

What are some of the problem MPI tackles (elegantly)?

Different data communication scenarios arising in scientific computing, non-contiguous data transfers, file I/O, et al
What is MPI?

**MPI**

A **message-passing library interface specification** for massively parallel systems

- Few open source implementations of MPI are MPICH (Argonne Nat’l Lab), foMPI (ETH Zurich), MVAPICH (OSU), OpenMPI (Sandia Nat’l Lab, CISCO, IU, et al), FG-MPI (UBC), et al
- Some vendor implementations are Cray-MPI, IBM-MPI, Intel-MPI, SGI-MPI, MS-MPI (yes, there is one!)
- Most MPI implementations are derivatives of MPICH [QED]

What are some of the problem MPI tackles (elegantly)?

Different data communication scenarios arising in scientific computing, non-contiguous data transfers, file I/O, et al

Where is MPI used?

Oil and Gas exploration, Climate Modeling, Biological sequence analysis, Computer Animation, and any large scale simulation
What is MPI?

MPI

A message-passing library interface specification for massively parallel systems

- Few open source implementations of MPI are MPICH (Argonne Nat’l Lab), foMPI (ETH Zurich), MVAPICH (OSU), OpenMPI (Sandia Nat’l Lab, CISCO, IU, et al), FG-MPI (UBC), et al
- Some vendor implementations are Cray-MPI, IBM-MPI, Intel-MPI, SGI-MPI, MS-MPI (yes, there is one!)
- Most MPI implementations are derivatives of MPICH [QED]

What are some of the problem MPI tackles (elegantly)?

Different data communication scenarios arising in scientific computing, non-contiguous data transfers, file I/O, et al

Where is MPI used?

Oil and Gas exploration, Climate Modeling, Biological sequence analysis, Computer Animation, and any large scale simulation
Processes and MPI

- An MPI program consists of autonomous processes, executing their own code (independently), in a Multiple Instruction Multiple Data style
- Processes synchronize by exchanging messages

What is a message in MPI?

Message = {Envelope + Data}

Envelope = [source rank, destination rank, tag, communicator]

Message envelope is used to distinguish messages and selectively receiving them.

MPI processes exposed as an integer value to the user, the process manager maps the integer value to an actual processor core (Eg: MPICH uses HYDRA process manager for starting parallel programs).
Processes and MPI

- An MPI program consists of autonomous processes, executing their own code (independently), in a Multiple Instruction Multiple Data style
- Processes synchronize by exchanging messages

What is a message in MPI?

Message = \{Envelope + Data\}
Envelope = [source rank, destination rank, tag, communicator]
Message envelope is used to distinguish messages and selectively receiving them
Processes and MPI

- An MPI program consists of autonomous processes, executing their own code (independently), in an Multiple Instruction Multiple Data style
- Processes synchronize by exchanging messages

What is a message in MPI?

Message = \{Envelope + Data\}
Envelope = [source rank, destination rank, tag, communicator]
Message envelope is used to distinguish messages and selectively receiving them

MPI processes

Exposed as an integer value to the user, the process manager maps the integer value to an actual processor core (Eg: MPICH uses HYDRA process manager for starting parallel programs)
**Processes and MPI**

- An MPI program consists of autonomous processes, executing their own code (independently), in a **Multiple Instruction Multiple Data** style.
- Processes *synchronize* by exchanging messages.

**What is a message in MPI?**

- **Message** =  
  \[
  \{\text{Envelope } + \text{ Data}\}
  \]

- **Envelope** =  
  \[
  \{\text{source rank, destination rank, tag, communicator}\}
  \]

- Message envelope is used to *distinguish* messages and *selectively receiving* them.

**MPI processes**

- Exposed as an integer value to the user, the *process manager* maps the integer value to an actual processor core (Eg: MPICH uses HYDRA process manager for starting parallel programs).
### MPI Tag

An integer value *message tag* is used to distinguish different messages.
An integer value *message tag* is used to distinguish different messages.

Provides a communication context – Consider them as *sets* of processes, every `Send/Recv` function needs a communicator object...the target/source process id (or rank) passed to a `Send/Recv` function is like an index to the elements of a particular set:

- `{0,1,4,8}` – com1
- `{0,2,4,6}` – com2
- `{1,2,3,4}` – com3
- `{0,3,6,8}` – com4
MPI Tag

An integer value *message tag* is used to distinguish different messages

MPI Communicator object

- Provides a communication context – Consider them as *sets* of processes, every `Send/Recv` function needs a communicator object...the target/source process id (or rank) passed to a `Send/Recv` function is like an index to the elements of a particular set
  - `{0,1,4,8}` – com1
  - `{0,2,4,6}` – com2
  - `{1,2,3,4}` – com3
  - `{0,3,6,8}` – com4

- Each communication context provides a separate “communication universe”: messages are always received within the context they were sent, and messages sent in different contexts do not interfere
### MPI Tag

An integer value *message tag* is used to distinguish different messages.

### MPI Communicator object

- Provides a communication context – Consider them as sets of processes, every Send/Recv function needs a communicator object...the target/source process id (or rank) passed to a Send/Recv function is like an index to the elements of a particular set
  - \{0,1,4,8\} – com1
  - \{0,2,4,6\} – com2
  - \{1,2,3,4\} – com3
  - \{0,3,6,8\} – com4

- Each communication context provides a separate “communication universe”: messages are always received within the context they were sent, and messages sent in different contexts do not interfere

- MPI provides a pre-defined communicator of all accessible processes – MPI_COMM_WORLD
### MPI Tag

An integer value *message tag* is used to distinguish different messages.

### MPI Communicator object

- Provides a communication context – Consider them as *sets* of processes, every `Send/Recv` function needs a communicator object...the target/source process id (or rank) passed to a `Send/Recv` function is like an index to the elements of a particular set:
  - `{0,1,4,8}` – com1
  - `{0,2,4,6}` – com2
  - `{1,2,3,4}` – com3
  - `{0,3,6,8}` – com4

- Each communication context provides a separate “communication universe”: messages are always received within the context they were sent, and messages sent in different contexts do not interfere.

- MPI provides a pre-defined communicator of all accessible processes – MPI_COMM_WORLD
**MPI Tag**

An integer value *message tag* is used to distinguish different messages.

**MPI Communicator object**

- Provides a communication context – Consider them as sets of processes, every Send/Recv function needs a communicator object...the target/source process id (or rank) passed to a Send/Recv function is like an index to the elements of a particular set
  
  \{0,1,4,8\} – com1  
  \{0,2,4,6\} – com2  
  \{1,2,3,4\} – com3  
  \{0,3,6,8\} – com4

- Each communication context provides a separate “communication universe”: messages are always received within the context they were sent, and messages sent in different contexts do not interfere

- MPI provides a pre-defined communicator of all accessible processes – MPI_COMM_WORLD

**MPI status object**

- Source rank or tag of a message may not be known *apriori* to a receiver (they are wildcards), *status* object stores the source rank, tag and count of data item in the message queue
**MPI Tag**

An integer value *message tag* is used to distinguish different messages.

**MPI Communicator object**

- Provides a communication context – Consider them as *sets* of processes, every `Send/Recv` function needs a communicator object...the target/source process id (or rank) passed to a `Send/Recv` function is like an index to the elements of a particular set:
  - `{0,1,4,8}` – com1
  - `{0,2,4,6}` – com2
  - `{1,2,3,4}` – com3
  - `{0,3,6,8}` – com4

- Each communication context provides a separate “communication universe”: messages are always received within the context they were sent, and messages sent in different contexts do not interfere.

- MPI provides a pre-defined communicator of all accessible processes – `MPI_COMM_WORLD`.

**MPI status object**

- Source rank or tag of a message may not be known *apriori* to a receiver (they are wildcards), *status* object stores the source rank, tag and count of data item in the message queue.

- It is possible to probe for a message prior to receiving (if the incoming data size is unknown, we cannot post a `Recv`) – `MPI_Probe(..., status)`.
MPI Tag

An integer value *message tag* is used to distinguish different messages

MPI Communicator object

- Provides a communication context – Consider them as sets of processes, every *Send/Recv* function needs a communicator object...the target/source process id (or rank) passed to a *Send/Recv* function is like an index to the elements of a particular set
  - \{0,1,4,8\} – com1
  - \{0,2,4,6\} – com2
  - \{1,2,3,4\} – com3
  - \{0,3,6,8\} – com4

- Each communication context provides a separate “communication universe”: messages are always received within the context they were sent, and messages sent in different contexts do not interfere

- MPI provides a pre-defined communicator of all accessible processes – MPI_COMM_WORLD

MPI status object

- Source rank or tag of a message may not be known *apriori* to a receiver (they are wildcards), *status* object stores the source rank, tag and count of data item in the message queue

- It is possible to probe for a message prior to receiving (if the incoming data size is unknown, we cannot post a Recv) – MPI_Probe(...)
MPI crucial concepts - II

MPI Tag

An integer value message tag is used to distinguish different messages

MPI Communicator object

- Provides a communication context – Consider them as sets of processes, every Send/Recv function needs a communicator object...the target/source process id (or rank) passed to a Send/Recv function is like an index to the elements of a particular set
  - \{0,1,4,8\} – com1
  - \{0,2,4,6\} – com2
  - \{1,2,3,4\} – com3
  - \{0,3,6,8\} – com4
- Each communication context provides a separate “communication universe”: messages are always received within the context they were sent, and messages sent in different contexts do not interfere
- MPI provides a pre-defined communicator of all accessible processes – MPI_COMM_WORLD

MPI status object

- Source rank or tag of a message may not be known apriori to a receiver (they are wildcards), status object stores the source rank, tag and count of data item in the message queue
  - It is possible to probe for a message prior to receiving (if the incoming data size is unknown, we cannot post a Recv) – MPI_Probe(...)
Every MPI application could be written using these 5 MPI functions:

- `MPI_Init(int *argc, int *argv[])`
- `MPI_Comm_rank(MPI_Comm comm, int *rank)`
- `MPI_Send( send_buf, count, datatype, dest_rank, tag, comm )`
- `MPI_Recv( recv_buf, count, datatype, source_rank, tag, comm, status )`
- `MPI_Finalize()`
A simple MPI program

```c
#include "mpi.h"
int main(int argc, char *argv[])
{
  char message[20];
  int myrank;
  MPI_Status status;
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
  if (myrank == 0)
    /* code for process zero */
  {
    strcpy(message, "Hello, there");
    MPI_Send(message, strlen(message)+1, MPI_CHAR, 1, 99, MPI_COMM_WORLD);
  }
  else if (myrank == 1) /* code for process one */
  {
    MPI_Recv(message, 20, MPI_CHAR, 0, 99, MPI_COMM_WORLD, &status);
    printf("received :%s:\n", message);
  }
  MPI_Finalize();
  return 0;
}
```
How does my data go from one process to another?

Two cases, *buffered* and *unbuffered*.

- Send buffer $\rightarrow$ Intermediate buffer $\rightarrow$ **Network** $\rightarrow$ Intermediate buffer $\rightarrow$ Receive buffer

Better, but now **MPI** Send needs to wait on delivery

Or **MPI** Send returns before the data has reached the other end, we will need some other function to wait for data transfer later
Two cases, *buffered* and *unbuffered*.

- Send buffer $\rightarrow$ Intermediate buffer $\rightarrow$ **Network** $\rightarrow$ Intermediate buffer $\rightarrow$ Receive buffer
- Send buffer $\rightarrow$ **Network** $\rightarrow$ Receive buffer
How does my data go from one process to another?

Two cases, *buffered* and *unbuffered*.

- Send buffer $\rightarrow$ Intermediate buffer $\rightarrow$ **Network** $\rightarrow$ Intermediate buffer $\rightarrow$ Receive buffer
- Send buffer $\rightarrow$ **Network** $\rightarrow$ Receive buffer
  - Better, but now MPI\_Send needs to wait on *delivery*
How does my data go from one process to another?

Two cases, *buffered* and *unbuffered*.

- Send buffer $\rightarrow$ Intermediate buffer $\rightarrow$ **Network** $\rightarrow$ Intermediate buffer $\rightarrow$ Receive buffer
- Send buffer $\rightarrow$ **Network** $\rightarrow$ Receive buffer
  - Better, but now `MPI_Send` needs to wait on delivery
  - Or `MPI_Send` returns before the data has reached the other end, we will need some other function to *wait* for data transfer later
How does my data go from one process to another?

Two cases, \textit{buffered} and \textit{unbuffered}.

- Send buffer $\rightarrow$ Intermediate buffer $\rightarrow$ \textbf{Network} $\rightarrow$ Intermediate buffer $\rightarrow$ Receive buffer
- Send buffer $\rightarrow$ \textbf{Network} $\rightarrow$ Receive buffer
  - Better, but now MPI\_Send needs to wait on \textit{delivery}
  - Or MPI\_Send returns before the data has reached the other end, we will need some other function to \textit{wait} for data transfer later
Message passing protocols

- **Eager** protocol – Assumption here is that the receiving process can receive the sending process’s data right away (without sending an acknowledgement to sender process)
Message passing protocols

- **Eager** protocol – Assumption here is that the receiving process can receive the sending process’s data right away (without sending an acknowledgement to sender process)

- **Rendezvous** protocol – For larger data sizes, sender first sends the *message envelope* to receiver, receiver checks the envelope and *allocates space* for data, and *sends notification* to the sender process that it is OK now to start sending the data
### Message passing protocols

- **Eager** protocol – Assumption here is that the receiving process can receive the sending process’s data right away (without sending an acknowledgement to sender process)

- **Rendezvous** protocol – For larger data sizes, sender first sends the *message envelope* to receiver, receiver checks the envelope and *allocates space* for data, and *sends notification* to the sender process that it is OK now to start sending the data
Message passing protocols

- **Eager** protocol – Assumption here is that the receiving process can receive the sending process’s data right away (without sending an acknowledgement to sender process)

- **Rendezvous** protocol – For larger data sizes, sender first sends the *message envelope* to receiver, receiver checks the envelope and *allocates space* for data, and *sends notification* to the sender process that it is OK now to start sending the data

Why not one-sided communication?

Yes, definitely! MPI-3.0 improved the one-sided semantics significantly
Message passing protocols

- **Eager** protocol – Assumption here is that the receiving process can receive the sending process’s data right away (without sending an acknowledgement to sender process)

- **Rendezvous** protocol – For larger data sizes, sender first sends the message envelope to receiver, receiver checks the envelope and allocates space for data, and sends notification to the sender process that it is OK now to start sending the data

Why not one-sided communication?

Yes, definitely! MPI-3.0 improved the one-sided semantics significantly
Categories of communication routines in MPI

- **Pair-wise point-to-point** – Send and Receive
Categories of communication routines in MPI

- **Pair-wise point-to-point** – Send and Receive
- **Collective** – Involves all processes in a communicator; examples are moving data from one to all/some processes (broadcast, scatter, gather, etc), reduction operations across processes
Categories of communication routines in MPI

- **Pair-wise point-to-point** – Send and Receive
- **Collective** – Involves all processes in a communicator; examples are moving data from one to all/some processes (broadcast, scatter, gather, etc), reduction operations across processes
- **On-sided communication** – Remote Memory Access extends MPI by allowing one process to specify all communication parameters, for sending side as well as receiving side (e.g, MPI_Put, MPI_Get)
Categories of communication routines in MPI

- **Pair-wise point-to-point** – Send and Receive
- **Collective** – Involves all processes in a communicator; examples are moving data from one to all/some processes (broadcast, scatter, gather, etc), reduction operations across processes
- **On-sided communication** – Remote Memory Access extends MPI by allowing one process to specify all communication parameters, for sending side as well as receiving side (e.g, MPI_Put, MPI_Get)
- **Synchronization** – Barrier, wait till all processes in a communicator reaches a point in program execution
Categories of communication routines in MPI

- **Pair-wise point-to-point** – Send and Receive
- **Collective** – Involves all processes in a communicator; examples are moving data from one to all/some processes (broadcast, scatter, gather, etc), reduction operations across processes
- **On-sided communication** – Remote Memory Access extends MPI by allowing one process to specify all communication parameters, for sending side as well as receiving side (e.g, MPI_Put, MPI_Get)
- **Synchronization** – Barrier, wait till all processes in a communicator reaches a point in program execution
Architecture of a typical MPI library
Error Handling

MPI standard leaves the behavior of an MPI implementation following process or network failures *undefined*. 
MPI standard leaves the behavior of an MPI implementation following process or network failures *undefined*.

**Expected fault tolerance**
Program must carry on even if one or more processes in the *communicator* has failed.
## MPI Fault Tolerance

### Error Handling

MPI standard leaves the behavior of an MPI implementation following process or network failures *undefined*.

### Expected fault tolerance

Program must carry on even if one or more processes in the *communicator* has failed.

### Role of MPI *communicator* in providing fault tolerance

- Prior to MPI-3.0 creation of MPI communicator object was a collective operation.
MPI Fault Tolerance

Error Handling
MPI standard leaves the behavior of an MPI implementation following process or network failures *undefined*

Expected fault tolerance
Program must carry on even if one or more processes in the *communicator* has failed

Role of MPI *communicator* in providing fault tolerance
- Prior to MPI-3.0 creation of MPI communicator object was a collective operation
- Supporting collective operations after a failure (in one or more processes) is challenging
## Error Handling

MPI standard leaves the behavior of an MPI implementation following process or network failures *undefined*.

## Expected fault tolerance

Program must carry on even if one or more processes in the *communicator* has failed.

## Role of MPI *communicator* in providing fault tolerance

- Prior to MPI-3.0 creation of MPI communicator object was a collective operation.
- Supporting collective operations after a failure (in one or more processes) is challenging.
- Currently MPI supports non-collective communicator creation routines (MPI_Comm_create_group(...)).
MPI Fault Tolerance

Error Handling

MPI standard leaves the behavior of an MPI implementation following process or network failures *undefined*.

Expected fault tolerance

Program must carry on even if one or more processes in the *communicator* has failed.

Role of MPI *communicator* in providing fault tolerance

- Prior to MPI-3.0 creation of MPI communicator object was a collective operation.
- Supporting collective operations after a failure (in one or more processes) is challenging.
- Currently MPI supports non-collective communicator creation routines (MPI_Comm_create_group(...)).
### MPI Fault Tolerance

#### Error Handling

MPI standard leaves the behavior of an MPI implementation following process or network failures *undefined*.

#### Expected fault tolerance

Program must carry on even if one or more processes in the *communicator* has failed.

#### Role of MPI *communicator* in providing fault tolerance

- Prior to MPI-3.0 creation of MPI communicator object was a collective operation.
- Supporting collective operations after a failure (in one or more processes) is challenging.
- Currently MPI supports non-collective communicator creation routines (MPI_Comm_create_group(...)).
Current and future fault tolerance strategy

Reliability

- Some network modules provide limited reliability – packets dropped on network are retransmitted

MPI user level failure mitigation

- Don’t do recovery for an application, provide relevant info to application/library and let it handle failures

- Only handle process failures currently

- If target rank not available for MPI Send, then notify with MPI ERR PROC FAILED

- If source rank failed, revoke the subsequent MPI Recv s

Failure Recovery – shrink the communicator object to remove the failed processes
Reliability

- Some network modules provide limited reliability – packets dropped on network are retransmitted
- When hardware cannot handle messages, enforce a *flow control* – pause and unpause message transmission, reissue packets for which acknowledgement not received
Current and future fault tolerance strategy

Reliability

- Some network modules provide limited reliability – packets dropped on network are retransmitted.
- When hardware cannot handle messages, enforce a *flow control* – pause and unpause message transmission, reissue packets for which acknowledgement not received.
Reliability

- Some network modules provide limited reliability – packets dropped on network are retransmitted.
- When hardware cannot handle messages, enforce a flow control – pause and unpause message transmission, reissue packets for which acknowledgement not received.

MPI user level failure mitigation

- Don’t do recovery for an application, provide relevant info to application/library and let it handle failures.
Current and future fault tolerance strategy

Reliability

- Some network modules provide limited reliability – packets dropped on network are retransmitted.
- When hardware cannot handle messages, enforce a flow control – pause and unpause message transmission, reissue packets for which acknowledgement not received.

MPI user level failure mitigation

- Don’t do recovery for an application, provide relevant info to application/library and let it handle failures.
- Only handle process failures currently.
Current and future fault tolerance strategy

Reliability
- Some network modules provide limited reliability – packets dropped on the network are retransmitted.
- When hardware cannot handle messages, enforce a flow control – pause and unpause message transmission, reissue packets for which acknowledgement not received.

MPI user level failure mitigation
- Don’t do recovery for an application, provide relevant info to application/library and let it handle failures.
- Only handle process failures currently.
  - If target rank not available for MPI_Send, then notify with MPI_ERR_PROC_FAILED.
Current and future fault tolerance strategy

Reliability

- Some network modules provide limited reliability – packets dropped on network are retransmitted
- When hardware cannot handle messages, enforce a *flow control* – pause and unpause message transmission, reissue packets for which acknowledgement not received

MPI *user level* failure mitigation

- Don’t do recovery for an application, provide relevant info to application/library and let it handle failures
- Only handle *process failures* currently
  - If target rank not available for MPI_Send, then notify with MPI_ERR_PROC_FAILED
  - If source rank failed, *revoke* the subsequent MPI_Recvs
Current and future fault tolerance strategy

Reliability
- Some network modules provide limited reliability – packets dropped on the network are retransmitted.
- When hardware cannot handle messages, enforce a *flow control* – pause and unpause message transmission, reissue packets for which acknowledgement not received.

MPI user level failure mitigation
- Don’t do recovery for an application, provide relevant info to application/library and let it handle failures.
- Only handle *process failures* currently.
  - If target rank not available for MPI_Send, then notify with MPI_ERR_PROC_FAILED.
  - If source rank failed, *revoke* the subsequent MPI_Recv.
  - Failure Recovery – shrink the communicator object to remove the failed processes.
Current and future fault tolerance strategy

**Reliability**
- Some network modules provide limited reliability – packets dropped on network are retransmitted.
- When hardware cannot handle messages, enforce a flow control – pause and unpause message transmission, reissue packets for which acknowledgement not received.

**MPI user level failure mitigation**
- Don’t do recovery for an application, provide relevant info to application/library and let it handle failures.
- Only handle process failures currently:
  - If target rank not available for MPI_Send, then notify with MPI_ERR_PROC_FAILED.
  - If source rank failed, revoke the subsequent MPI_Recvs.
  - Failure Recovery – shrink the communicator object to remove the failed processes.
Reliability

- Some network modules provide limited reliability – packets dropped on network are retransmitted.
- When hardware cannot handle messages, enforce a flow control – pause and unpause message transmission, reissue packets for which acknowledgement not received.

MPI user level failure mitigation

- Don’t do recovery for an application, provide relevant info to application/library and let it handle failures.
- Only handle process failures currently.
  - If target rank not available for MPI_Send, then notify with MPI_ERR_PROC_FAILED.
  - If source rank failed, revoke the subsequent MPI_Recv.
  - Failure Recovery – shrink the communicator object to remove the failed processes.
Thanks

Most of the materials were taken from the current MPI specification (http://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf), and presentations/papers of Pavan Balaji (ANL), Rajeev Thakur (ANL) and Jeff Hammond (Intel Labs).

sghosh1@eecs.wsu.edu