CC755: Distributed and Parallel Systems

Lecture 9: Programming Using the Message Passing Paradigm

Slides by: Ananth Grama, Anshul Gupta, George Karypis, and Vipin Kumar

To accompany the text "Introduction to Parallel Computing", Addison Wesley, 2003.
With some edits from other sources

Dr. Manal Helal, Spring 2016

moodle.manalhelal.com

Topic Overview

- Motivation
- Principles of Message-Passing Programming
- The Building Blocks: Send and Receive Operations
- MPI: the Message Passing Interface

Motivation

- Synchronisation
- Load balancing
- Resource management
- ...in all cases communication between nodes is required.

MPI: the Message Passing Interface

- MPI defines a standard library for message-passing that can be used to develop portable message-passing programs using either C or Fortran.
- The MPI standard defines both the syntax as well as the semantics of a core set of library routines.
- It is possible to write fully-functional message-passing programs by using only six basic routines.

Principles of Message-Passing Programming

- The logical view of a machine supporting the message-passing paradigm consists of p processes, each with its own exclusive address space.
- Each data element must belong to one of the partitions of the space; hence, data must be explicitly partitioned and placed.
- All interactions (read-only or read/write) require cooperation of two processes - the process that has the data and the process that wants to access the data.
- These two constraints, while onerous, make underlying costs very explicit to the programmer.

Principles of Message-Passing Programming

- Message-passing programs are often written using the *asynchronous* or *loosely synchronous* paradigms.
- In the asynchronous paradigm, all concurrent tasks execute asynchronously.
- In the loosely synchronous model, tasks or subsets of tasks synchronise to perform interactions. Between these interactions, tasks execute completely asynchronously.
- Most message-passing programs are written using the *single program multiple data* (SPMD) model.

MPI Implementations

- MPICH: www-unix.mcs.anl.gov/mpi/mpich/
- LAM/MPI: www.lam-mpi.org
- Open MPI: www.open-mpi.org
- MVAPICH2: mvapich.cse.ohio-state.edu/overview/mvapich2/

» The MPI interface is standard, so programming in all is identical: http://www-unix.mcs.anl.gov/mpi/

» Booting and terminating nodes, and running jobs on the system, differs slightly.

MPI Commands

- MPI is simple, but complex; or is it complex, but simple?
- How many MPI commands are there?
  - Basic Commands: 6+1
  - 128+
    - 52 Point-to-Point Communication
    - 16 Collective Communication
    - 30 Groups, Contexts, and Communicators
    - 16 Process Topologies
    - 13 Environmental Inquiry
    - 1 Profiling

Six Basic MPI commands via three fingers

Pointer Finger – Setup: starting & terminating the MPI Library

- `1) MPI_Init` is called prior to any calls to other MPI routines. Its purpose is to initialize the MPI environment.
- `2) MPI_Finalize` is called at the end of the computation, and it performs various clean-up tasks to terminate the MPI environment.
- The prototypes of these two functions are:

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

- `MPI_Init` also strips off any MPI related command-line arguments.
- All MPI routines, data-types, and constants are prefixed by "`MPI_`". The return code for successful completion is `MPI_SUCCESS`.

Six Basic MPI commands via three fingers

Rule of Thumb – Know thy self : Querying Information

- The 3) `MPI_Comm_size` and 4) `MPI_Comm_rank` functions are used to determine the number of processes and the label of the calling process, respectively.
- 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)
```

- The rank of a process is an integer that ranges from zero up to the size of the communicator minus one.
- + 1) `MPI_Get_processor_name(char * name, int *length)` – External processor name

Hello MPI Program

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

int main(int argc, char ** argv) {
   int size,rank, length;
   char name[80];
   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD,&rank);
   MPI_Comm_size(MPI_COMM_WORLD,&size);
   MPI_Get_processor_name(name,&length);
   printf("Hello MPI! Process %d of %d on %s\n”,rank, size,name);
   MPI_Finalize();
   return 0;
}
```

```
$ mpicc hello.c  -o hello
$ mpiexec -n 2 ./hello
Hello MPI! Process 0 of 2 on Manals-MacBook-Pro.local
Hello MPI! Process 1 of 2 on Manals-MacBook-Pro.local
```

Six Basic MPI commands via three fingers

Middle Finger – Message Passing

- 5) MPI Send()
- 6) MPI Recv()

MPI: the Message Passing Interface

The minimal set of MPI routines.

1	`MPI_Init`	Initializes MPI.
2	`MPI_Finalize`	Terminates MPI.
3	`MPI_Comm_size`	Determines the number of processes.
4	`MPI_Comm_rank`	Determines the label of calling process.
5	`MPI_Send`	Sends a message.
6	`MPI_Recv`	Receives a message.

The Building Blocks: Send and Receive Operations

- The prototypes of these operations are as follows:

```
send(void *sendbuf, int nelems, int dest)
receive(void *recvbuf, int nelems, int source)
```

- Consider the following code segments:

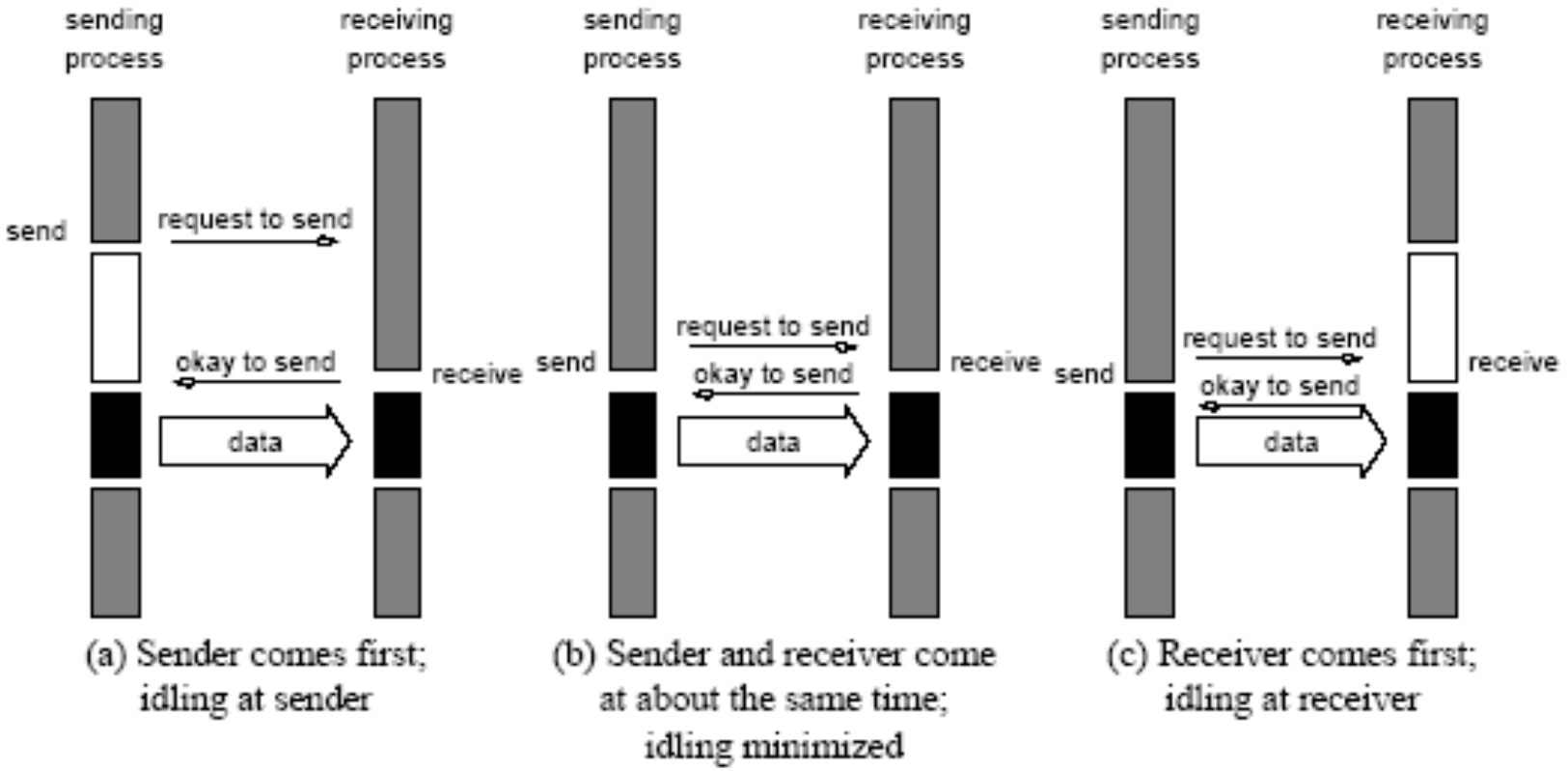
```
P0                          P1
a = 100;                    receive(&a, 1, 0)
send(&a, 1, 1);             printf("%d\n", a);
a = 0;
```

- The semantics of the send operation require that the value received by process P1 must be 100 as opposed to 0.
- This motivates the design of the send and receive protocols.

Non-Buffered Blocking Message Passing Operations

- A simple method for forcing send/receive semantics is for the send operation to return only when it is safe to do so.
- In the non-buffered blocking send, the operation does not return until the matching receive has been encountered at the receiving process.
- Idling and deadlocks are major issues with non-buffered blocking sends.
- In buffered blocking sends, the sender simply copies the data into the designated buffer and returns after the copy operation has been completed. The data is copied at a buffer at the receiving end as well.
- Buffering alleviates idling at the expense of copying overheads.

Non-Buffered Blocking Message Passing Operations



(a) Sender comes first; idling at sender

(b) Sender and receiver come at about the same time; idling minimized

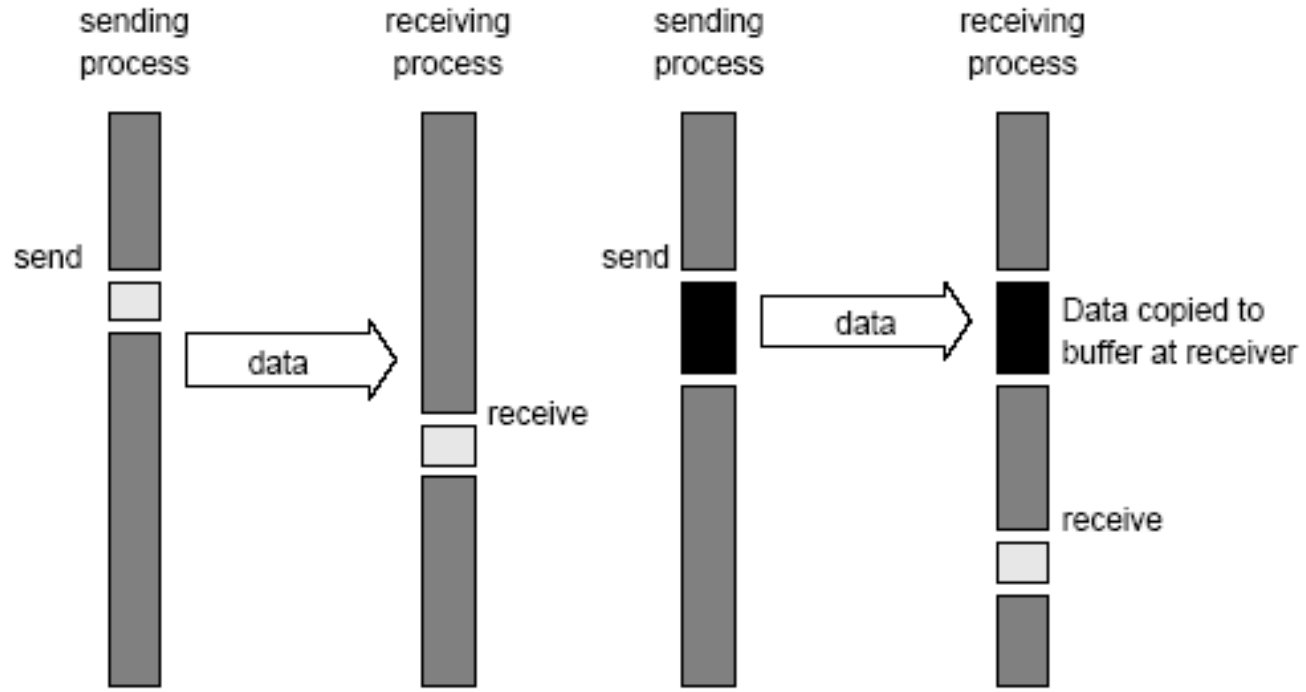
(c) Receiver comes first; idling at receiver

Handshake for a blocking non-buffered send/receive operation. It is easy to see that in cases where sender and receiver do not reach communication point at similar times, there can be considerable idling overheads.

Buffered Blocking Message Passing Operations

- A simple solution to the idling and deadlocking problem outlined above is to rely on buffers at the sending and receiving ends.
- The sender simply copies the data into the designated buffer and returns after the copy operation has been completed.
- The data must be buffered at the receiving end as well.
- Buffering trades off idling overhead for buffer copying overhead.

Buffered Blocking Message Passing Operations



Blocking buffered transfer protocols: (a) in the presence of communication hardware with buffers at send and receive ends; and (b) in the absence of communication hardware, sender interrupts receiver and deposits data in buffer at receiver end.

Buffered Blocking Message Passing Operations

Bounded buffer sizes can have significant impact on performance.

P0

```
for (i = 0; i < 1000; i++){
     produce_data(&a);
     send(&a, 1, 1);
}
```

P1

```
for (i = 0; i < 1000; i++){
     receive(&a, 1, 0);
     consume_data(&a);
}
```

What if consumer was much slower than producer?

Buffered Blocking Message Passing Operations

Deadlocks are still possible with buffering since receive operations block.

P0

```
receive(&a, 1, 1);
send(&b, 1, 1);
```

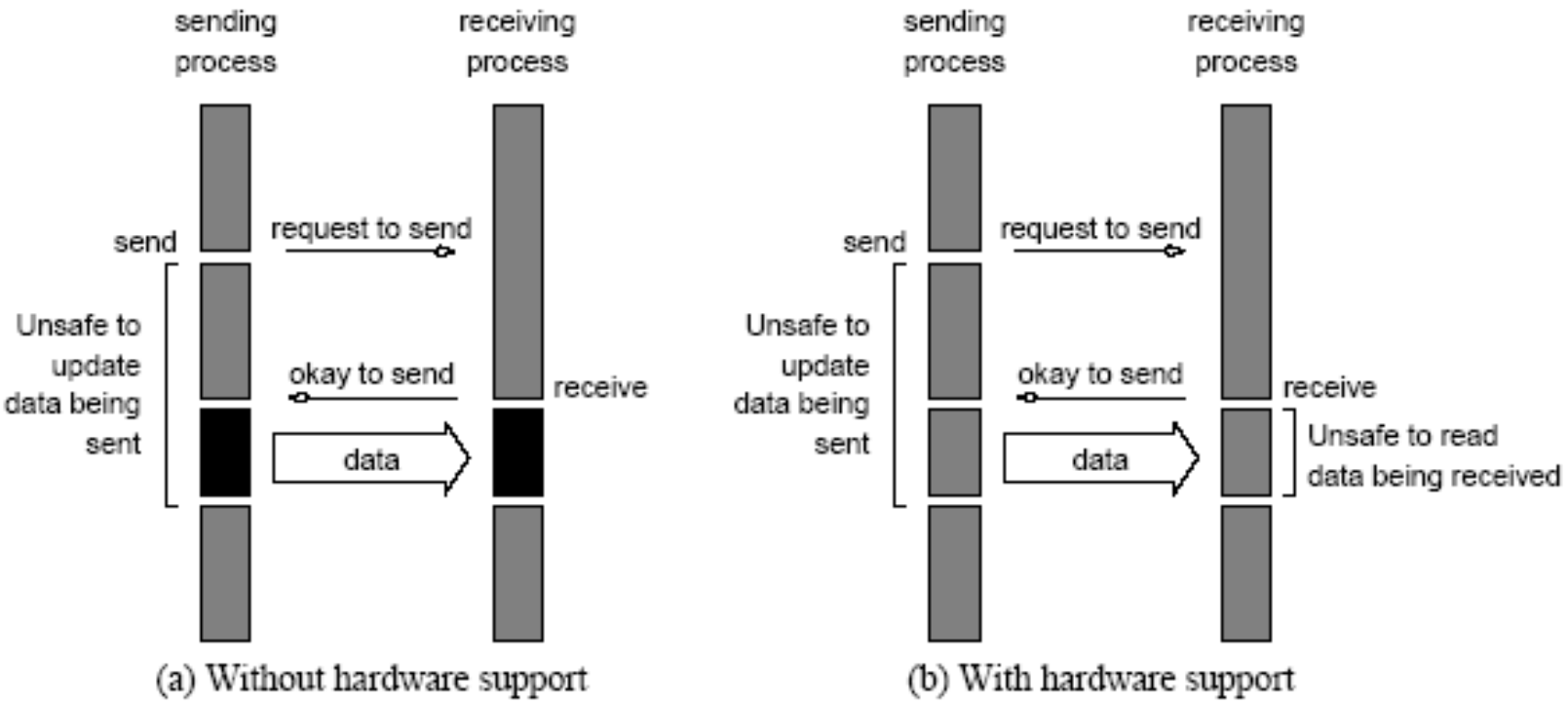
P1

```
receive(&a, 1, 0);
send(&b, 1, 0);
```

Non-Blocking Message Passing Operations

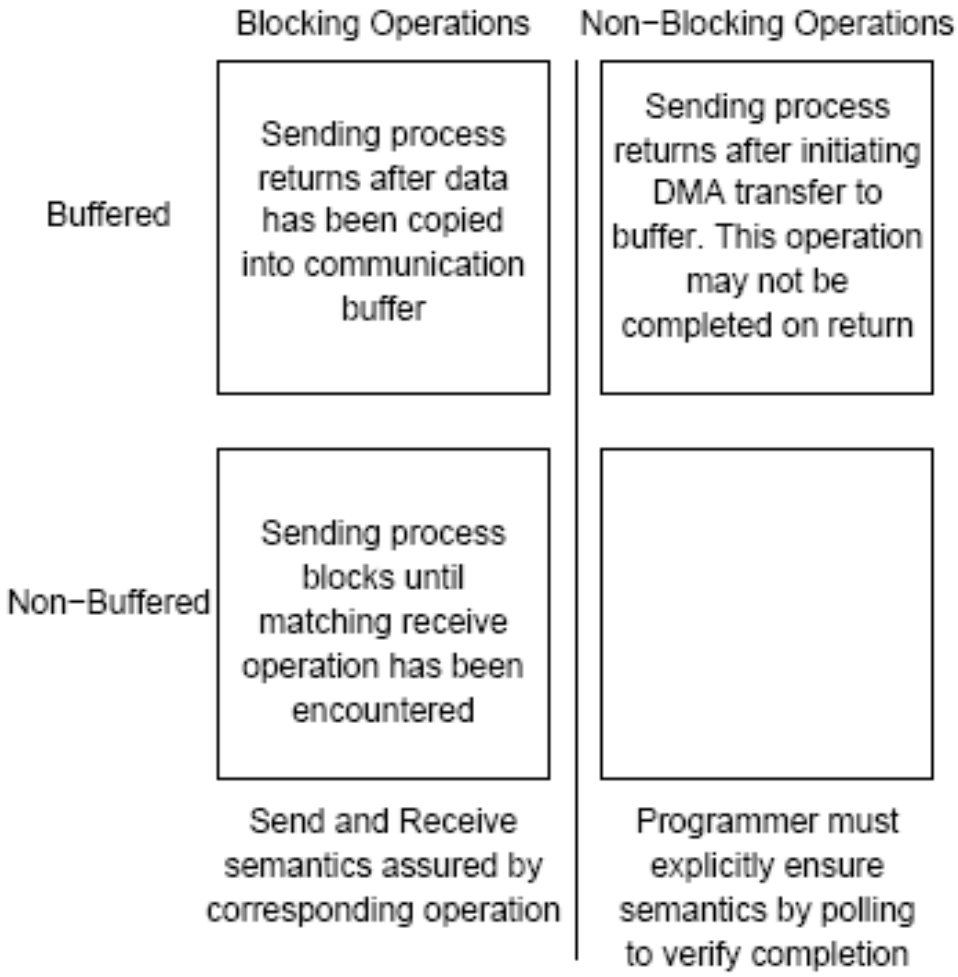
- The programmer must ensure semantics of the send and receive.
- This class of non-blocking protocols returns from the send or receive operation before it is semantically safe to do so.
- Non-blocking operations are generally accompanied by a check-status operation.
- When used correctly, these primitives are capable of overlapping communication overheads with useful computations.
- Message passing libraries typically provide both blocking and non-blocking primitives.

Non-Blocking Message Passing Operations



Non-blocking non-buffered send and receive operations (a) in absence of communication hardware; (b) in presence of communication hardware.

Send and Receive Protocols



Space of possible protocols for send and receive operations.

Blocking Send Modes

- **Synchronous** – **Stoplight Intersection**
  - No buffer, but both sides wait for other
- **Buffered** – **The roundabout You construct**
  - Explicit user buffer, alls well as long as within buffer
- **Ready** – **Fire truck Stoplight Override**
  - No buffer, no handshake, Send is the firetruck
- **Standard** – **The Roundabout**
  - Not so standard blend of Synchronous and Buffered
  - Internal buffer?

Send/Receive in Different modes

- Synchronous communication
  - MPI_Send - Performs a blocking send
  - MPI_Ssend - Blocking synchronous send
  - MPI_Rsend - Blocking ready send
  - MPI_Recv - Blocking receive for a message
- Asynchronous communication
  - MPI_Isend - Begins a nonblocking send
  - MPI_Irecv - Begins a nonblocking receive
- Buffered
  - MPI_Bsend - Basic send with user-provided buffering
- Collective communication
  - MPI_Broadcast, MPI_Gather, MPI_Reduce