1 Programming Using the Message-Passing Paradigm

- A message passing architecture uses a set of primitives that allows processes to communicate with each other.
- i.e., *send*, *receive*, *broadcast*, and *barrier*.
- Numerous programming languages and libraries have been developed for explicit parallel programming.These differ in
  - their view of the address space that they make available to the programmer,
  - the degree of synchronization imposed on concurrent activities, and the multiplicity of programs.
- Some links; Scientific Applications on Linux, Parallel Programming Laboratory.

1.1 Principles of Message-Passing Programming

There are two key attributes that characterize the message-passing programming paradigm.

1. the first is that it assumes a partitioned address space,
2. the second is that it supports only explicit parallelization.

- Each data element must **belong to one of the partitions** of the space;
  - hence, data must be explicitly partitioned and placed.
  - Adds complexity, encourages data locality, NUMA architecture.
- 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).
  - process that has the data must participate in the interaction,
  - for dynamic and/or unstructured interactions, the *complexity of the code* can be very high,
  - primary advantage of explicit two-way interactions is that the programmer is fully aware of all the costs of non-local interactions

- more likely to think about algorithms (and mappings) that minimize interactions.

- The programmer is responsible for analyzing the underlying serial algorithm/application.
- Identifying ways by which he or she can decompose the computations and extract concurrency.
- As a result, programming using the message-passing paradigm tends to be hard and intellectually demanding.
- However, on the other hand, **properly written** message-passing programs can often *achieve very high performance* and *scale to a very large* number of processes.

1.2 Structure of Message-Passing Programs

- Message-passing programs are often written using the *asynchronous* or *loosely synchronous* paradigms.
- In the ***asynchronous*** paradigm, all concurrent tasks execute asynchronously.
  - However, such programs can be harder and can have *non-deterministic behavior* due to race conditions.
- ***Loosely synchronous*** programs are a good compromise between two extremes.
  - In such programs, tasks or subsets of tasks synchronize to perform interactions.
  - However, between these interactions, tasks execute completely asynchronously.
- In its most general form, the message-passing paradigm supports execution of a different program on each of the p processes.
- This provides the ultimate flexibility in parallel programming, but makes the job of writing parallel programs effectively unscalable.
- For this reason, most message-passing programs are written using the single program multiple data *(SPMD)*.

- In SPMD programs the code executed by different processes is identical except for a small number of processes (e.g., the ”root” process).
- In an extreme case, even in an SPMD program, each process could execute a different code (many case statements).
- But except for this degenerate case, most processes execute the same code.
- SPMD programs can be loosely synchronous or completely asynchronous.

1.3 The Building Blocks: Send and Receive Operations

- Since interactions are accomplished by *sending* and *receiving* messages, the basic operations in the message-passing programming paradigm are **send** and **receive**.
- In their simplest form, the prototypes of these operations are defined as follows:

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

  - *sendbuf* points to a buffer that stores the data to be sent,
  - *recvbuf* points to a buffer that stores the data to be received,
  - *nelems* is the number of data units to be sent and received,
  - *dest* is the identifier of the process that receives the data,
  - *source* is the identifier of the process that sends the data.

```
P0                              P1

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

- Process P_0 sends a message to process P_1 which receives and prints the message.

- The important thing to note is that process P_0 changes the value of a to 0 immediately following the send.

- The semantics of the send operation require that the value received by process P_1 must be 100 (not 0).

- That is, the value of a at the time of the send operation must be the value that is received by process P_1.

- It may seem that it is quite straightforward to ensure the semantics of the send and receive operations.

- However, based on how the send and receive operations are implemented this may not be the case.

- Most message passing platforms have additional hardware support for sending and receiving messages.

- They may support DMA (direct memory access) and asynchronous message transfer using network interface hardware.

- Network interfaces allow the transfer of messages from buffer memory to desired location *without CPU intervention*.

- Similarly, DMA allows copying of data from one memory location to another (e.g., communication buffers) *without CPU support* (once they have been programmed).

- As a result, *if the send operation programs the communication hardware and returns before the communication operation has been accomplished*, process P_1 might receive the value 0 in a instead of 100!

1.3.1 Blocking Message Passing Operations

- A simple solution to the dilemma presented in the code fragment above is for the send operation to return only when it is semantically safe to do so.

- Note that this is not the same as saying that the send operation returns only after the receiver has received the data.

- It simply means that the sending operation blocks until it can guarantee that the semantics will not be violated on return irrespective of what happens in the program subsequently.

- There are two mechanisms by which this can be achieved.
    1. Blocking Non-Buffered Send/Receive
    2. Blocking Buffered Send/Receive

1 Blocking Non-Buffered Send/Receive

- The send operation does not return until the matching receive has been encountered at the receiving process.
- When this happens, the message is sent and the send operation returns upon completion of the communication operation.
- Typically, this process involves a *handshake* between the sending and receiving processes (see Fig. 1).

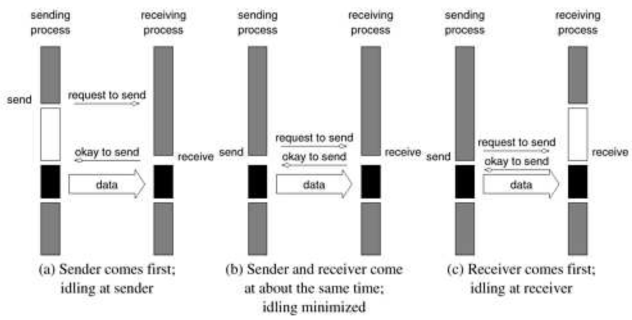

Figure 1: Handshake for a blocking non-buffered send/receive operation.

- The sending process sends a request to communicate to the receiving process.
- When the receiving process encounters the target receive, it responds to the request.
- The sending process upon receiving this response initiates a transfer operation.
- Since there are no buffers used at either sending or receiving ends, this is also referred to as a **non-buffered blocking** operation.
- *Idling Overheads in Blocking Non-Buffered Operations:* It is clear from the figure that a blocking non-buffered protocol is suitable when the send and receive are posted at roughly the same time (middle in the figure).

- However, in an asynchronous environment, this may be impossible to predict.
- This idling overhead is one of the major drawbacks of this protocol.
- *Deadlocks in Blocking Non-Buffered Operations:* Consider the following simple exchange of messages that can lead to a deadlock:

```
  P0                            P1

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

- The code fragment makes the values of a available to both processes P_0 and P_1.
- However, if the send and receive operations are implemented using a blocking non-buffered protocol,
  - the send at P_0 waits for the matching receive at P_1
  - whereas the send at process P_1 waits for the corresponding receive at P_0,
  - resulting in an infinite wait.
- Deadlocks are very easy in blocking protocols and care must be taken to break cyclic waits.

2 Blocking Buffered Send/Receive

  - A simple solution to the *idling* and *deadlocking* problems outlined above is to rely on **buffers** at the sending and receiving ends.

Figure 2Left

- On a send operation, the sender simply *copies the data into* the designated buffer and *returns after the copy operation has been completed.*
- The sender process can now continue with the program knowing that any changes to the data will not impact program semantics.
- If the hardware supports asynchronous communication (independent of the CPU), then a network transfer can be initiated after the message has been copied into the buffer.

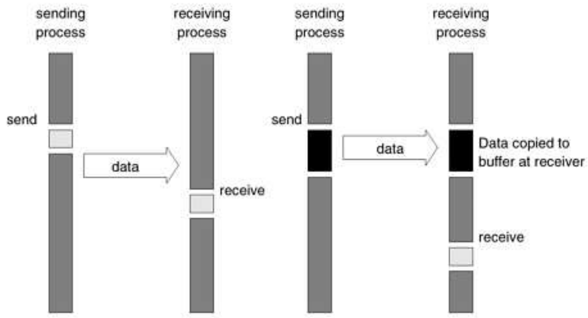

Figure 2: Blocking buffered transfer protocols: *Left:* in the presence of communication hardware with buffers at send and receive ends; and *Right:* in the absence of communication hardware, sender interrupts receiver and deposits data in buffer at receiver end.

- Note that at the receiving end, the data cannot be stored directly at the target location since this would violate program semantics.
- Instead, the data is copied into a buffer at the receiver as well.
- When the receiving process encounters a receive operation, it checks to see if the message is available in its receive buffer. If so, the data is copied into the target location.

Figure 2Right

- In Fig. 2Left, **buffers** are used at both sender and receiver and communication is handled by dedicated hardware.
- Sometimes machines do not have such communication hardware.
- In this case, some of the overhead can be saved by buffering only on one side.
- For example, on encountering a send operation, the sender interrupts the receiver, both processes participate in a communication operation and the message is deposited in a buffer at the receiver end.
- When the receiver eventually encounters a receive operation, the message is copied from the buffer into the target location.
- In general, if the parallel program is highly synchronous, non-buffered sends may perform better than buffered sends.

- However, generally, this is not the case and buffered sends are desirable unless buffer capacity becomes an issue.

- Impact of finite buffers in message passing; consider the following code fragment:

```
   P0                                    P1

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

- In this code fragment, process P_0 produces 1000 data items and process P_1 consumes them.

- However, if process P_1 was slow getting to this loop, process P_0 might have sent all of its data.

- If there is enough buffer space, then both processes can proceed;

- however, if the buffer is not sufficient (i.e., buffer overflow), the sender would have to be blocked until some of the corresponding receive operations had been posted, thus freeing up buffer space.

- This can often *lead to unforeseen overheads and performance degradation.*

- In general, it is a good idea to write programs that have bounded buffer requirements.

- *Deadlocks in Buffered Send and Receive Operations:*

- While buffering relieves many of the deadlock situations, it is still possible to write code that deadlocks.

- This is due to the fact that as in the non-buffered case, receive calls are always blocking (*to ensure semantic consistency*).

- Thus, a simple code fragment such as the following deadlocks since both processes wait to receive data but nobody sends it.

- Once again, such circular waits have to be broken.

- However, deadlocks are caused only by waits on receive operations in this case.

```
    P0                                  P1

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

1.3.2 Non-Blocking Message Passing Operations

- In blocking protocols, the *overhead of guaranteeing* semantic correctness was paid in the form of idling (non-buffered) or buffer management (buffered).
- It is possible to require the programmer
  - to ensure semantic correctness,
  - to provide a fast send/receive operation that incurs little overhead.
- This class of **non-blocking protocols** returns from the send or receive operation before it is semantically safe to do so.
- Consequently, the user must be careful not to alter data that may be potentially participating in communication.
- Non-blocking operations are generally accompanied by a check-status operation,
- which indicates whether the semantics of a previously initiated transfer may be violated or not.
- Upon return from a non-blocking operation, the process is free to perform any computation that does not depend upon the completion of the operation.
- Later in the program, the process can check whether or not the non-blocking operation has completed,
- and, if necessary, wait for its completion.
- Non-blocking operations can be buffered or non-buffered.
- In the non-buffered case, a process wishing to send data to another simply posts a pending message and returns to the user program.
- The program can then do other useful work.
- At some point in the future, *when the corresponding receive is posted*, the communication operation is initiated.

- When this operation is completed, the *check-status operation indicates* that it is safe to touch this data.
- This transfer is indicated in Fig. 3Left.
- The benefits of non-blocking operations are further enhanced by the presence of dedicated communication hardware.
- In this case, the communication overhead can be almost entirely masked by non-blocking operations.
- However, the data being received is unsafe for the duration of the receive operation.
- This is illustrated in Fig. 3Right.

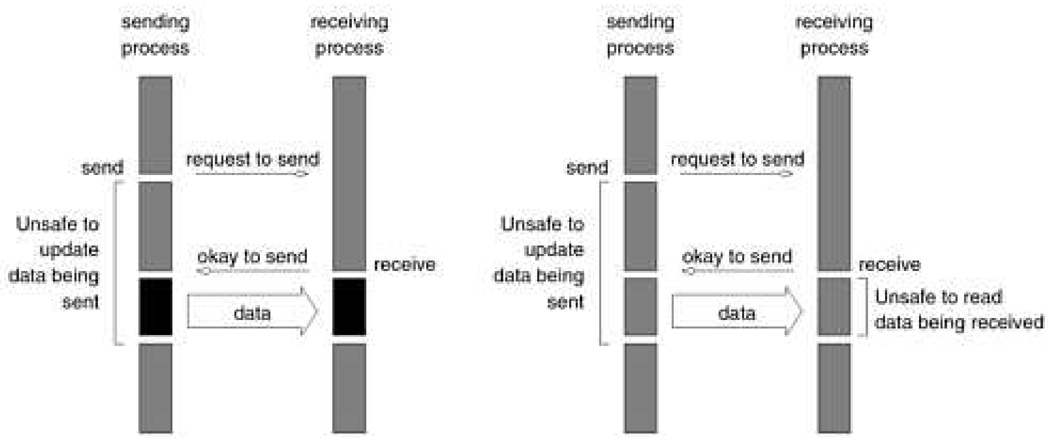

Figure 3: Non-blocking non-buffered send and receive operations *Left:* in absence of communication hardware; *Right:* in presence of communication hardware.

- Comparing Figures 3Left and 1a, it is easy to see that the idling time when the process is waiting for the corresponding receive in a blocking operation can now be utilized for computation (provided it does not update the data being sent).
- This removes the major bottleneck associated with the former at the expense of some program restructuring.

- Typical message-passing libraries such as Message Passing Interface (MPI) and Parallel Virtual Machine (PVM) implement both blocking and non-blocking operations.

- Blocking operations facilitate safe and easier programming.

- Non-blocking operations are useful for performance optimization by masking communication overhead.

- One must, however, be careful using non-blocking protocols since errors can result from <u>unsafe access</u> to data that is in the process of being communicated.

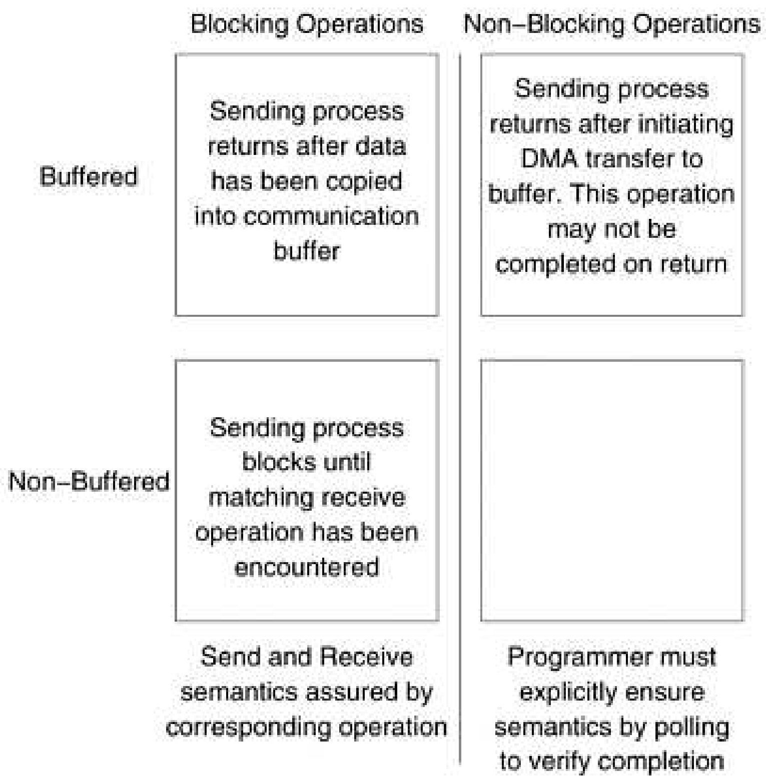

Figure 4: Space of possible protocols for send and receive operations.