Lecture 5

Programming Using the Message-Passing Paradigm I

Principles of Message-Passing Programming

IKC-MH.57 *Introduction to High Performance and Parallel Computing* at November 10, 2023

Dr. Cem Özdoğan
Engineering Sciences Department
İzmir Kâtip Çelebi University

Contents

Principles of Message-Passing Programming I

Set of Primitives: Allows processes to communicate with each other.

- A message passing architecture uses a set of primitives that allows processes to communicate with each other.
- i.e., *send*, *receive*, *broadcast*, and *barrier*.

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.**

Principles of Message-Passing Programming II

- All interactions (read-only or read/write) require **cooperation of two processes**:
  1. the process that has the data,
  2. the process that wants to access the data.
- Primary advantage of explicit two-way interactions is that the programmer is fully aware of all the costs of non-local interactions
- The programmer is responsible for analyzing the underlying serial algorithm/application.
- 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.

Structure of Message-Passing Programs I

- 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.
- Most message-passing programs are written using the single program multiple data *(SPMD)*.
- SPMD programs can be loosely synchronous or completely asynchronous.

The Building Blocks: Send and Receive Operations I

- 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.

The Building Blocks: Send and Receive Operations II

```
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.*

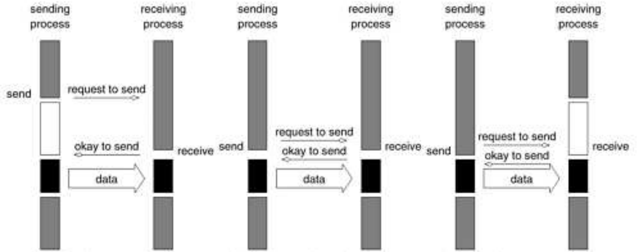
Blocking Message Passing Operations I

- 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!
- A simple solution to the problem 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

Blocking Message Passing Operations II

❶ 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 Figure).



(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

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

Blocking Message Passing Operations III

• 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 (see Figure(b)).
- However, in an asynchronous environment, this may be impossible to predict.
- This idling overhead is one of the major drawbacks of this protocol.

Blocking Message Passing Operations IV

- *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.

Blocking Message Passing Operations V

❷ 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.

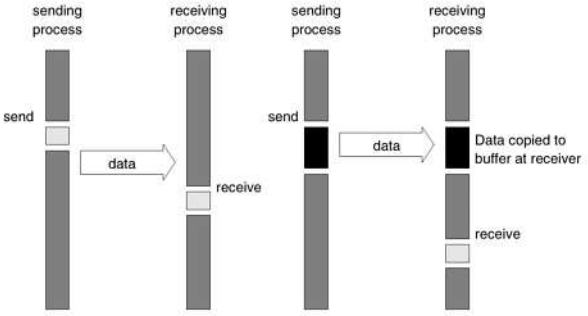


Figure: Blocking buffered transfer protocols: *Left:* in the presence of communication hardware with buffers at send and receive ends; and *Right:* in the absence of communication hardware, sender interrupts receiver and deposits data in buffer at receiver end.

Blocking Message Passing Operations VI

- 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.
- 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.
- 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.

Blocking Message Passing Operations VII

- *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.

```
    P0                               P1

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

- Once again, such circular waits have to be broken.
- However, deadlocks are caused only by waits on receive operations in this case.

Non-Blocking Message Passing Operations I

- In blocking protocols, the *overhead of guaranteeing* <u>semantic correctness</u> was paid in the form of <u>idling</u> (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 <u>check-status operation</u>,
- which indicates whether the semantics of a previously initiated transfer may be violated or not.

Non-Blocking Message Passing Operations II

- 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.

Non-Blocking Message Passing Operations III

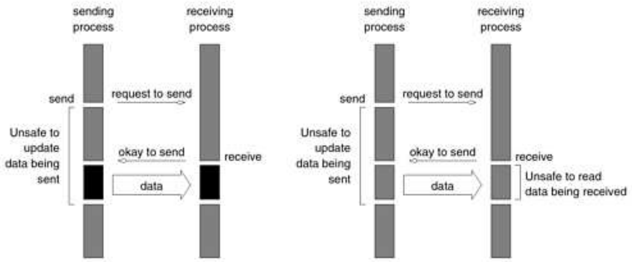


Figure: Non-blocking non-buffered send and receive operations *Left:* in absence of communication hardware; *Right:* in presence of communication hardware.

- This transfer is indicated in Figure(Left)
- Comparing Figures (Left) and (a), 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.
- This *removes the major bottleneck* associated with the former at the expense of some program restructuring.

Non-Blocking Message Passing Operations IV

- 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 unsafe access to data that is in the process of being communicated.