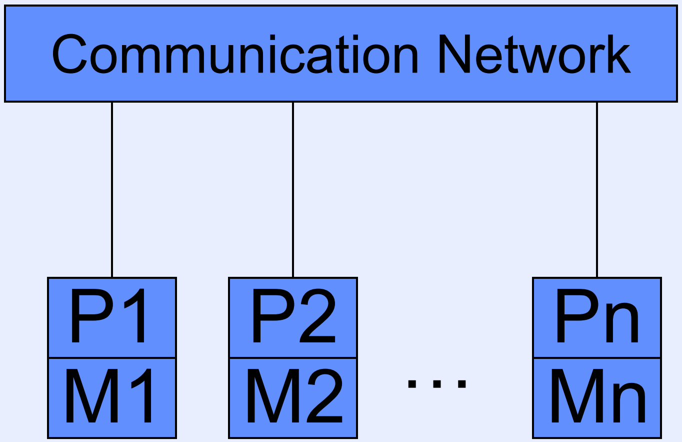
Message Passing Paradigm
MPI : Distributed memory systems

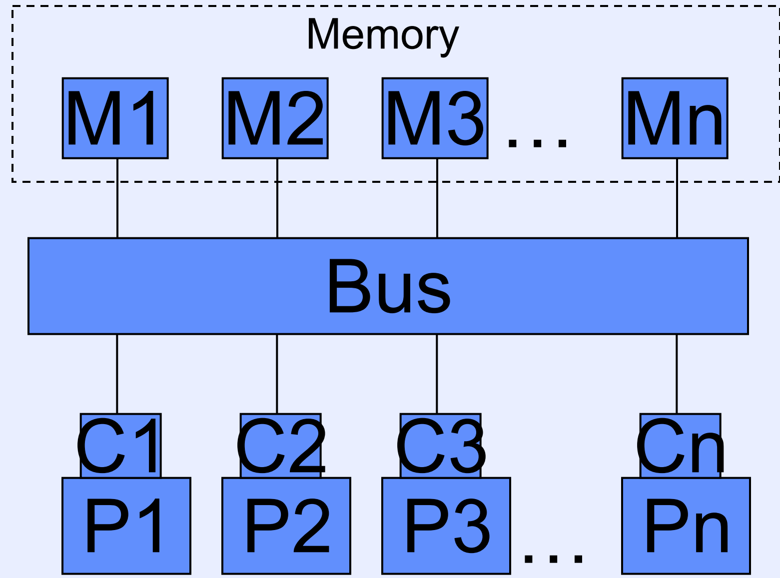
Topics to be covered

- Distributed-Memory Parallel Computer
- MP Programming Model
- MPI Program structure – first Example
- MPI Communications
- One Case Study using MPI

Parallel Systems



Distributed-memory computer



Shared-memory computer

P – processor; C – cache; M – memory.

- Data flows : memory →cache →processors
- Performance depends dramatically on reuse of data in cache

Parallel Systems

Based on address space/ memory organization :

Shared-memory parallel computers	Distributed-memory parallel computers
Processors/cores can access all memories	Processor can only access local memory. Remote memory access through explicit communication
Processors are all the same and have equal access to machine resources - symmetric multiprocessor (SMP)	All processors may not be same.
Uniform Memory Access machines	Access time to a memory location is not uniform, hence they are also known as Non-Uniform Memory Access machines.
	Performance of network connection crucial to performance of applications. Ideal: low latency, high bandwidth
	High scalability - No memory contention like shared-memory machines

Parallel Programming Paradigms

- Parallel Computing Paradigms
  - *Directives (OpenMP)*
    - Shared memory only
  - *Message Passing (MPI)*
    - Distributed or shared memory
  - *Multi-Level/ hybrid Parallel programming (MPI + OpenMP)*
    - Shared (and distributed) memory

SPMD vs. MPMD

- SPMD: Single program that performs same operation on multiple sets of data
- MPMD: Different programs perform different operations on multiple sets of data
- Hybrid program in which some processes perform same task

Programming Model

SPMD: Single program multiple data

All processors execute same program (executable a.out) on multiple data sets - domain decomposition.

```
if(my_cpu_id == k){}
else {}
```

MPMD: multiple programs multiple data

Different processors execute different programs on different data

master-slave model

- Master CPU creates & dispatches jobs to slave CPUs running a different program.
- Can be converted into SPMD model
  - *If (my_cpu_id==k) run program_1;*
  - *else run program_2;*

Distributed-Memory Computer

- the Rules
  - *Problem needs to be broken up into independent tasks with independent memory*
  - *Each task is assigned to a processor*
  - *Domain decomposition*

- Message passing: tasks explicitly exchange data by **message passing.**
  - *Transfers all data using explicit instructions*
  - *User must optimize communications*

MPI

MPI stands for **Message Passing Interface.**

- Library of subroutines/functions - not a language.
- Programmer insert appropriate MPI subroutine/ function calls, compile and finally link with MPI message passing library.

What is Message!

1. Collection of data (say array)
   Basic data types such as integer, float/real
   Derived data types
2. Message "envelope" – source, destination, tag, communicator

Advantages of MPI

- Provides efficient communication (message passing) among clusters of nodes
- Helps in more analyses in a given amount of time.
- Reduce time required for one analysis.
- To have access to more memory.
- To enhance code portability; works for both shared- and distributed-memory.

Limitations

➢ Introduces an additional overhead because of inter-processor communication

➢ Low latency and high bandwidth for inter-processor communication → key to higher performance

MPI Programming Model

- Message passing model: data exchange through explicit communications.

- For distributed memory, as well as **shared-memory parallel machines**

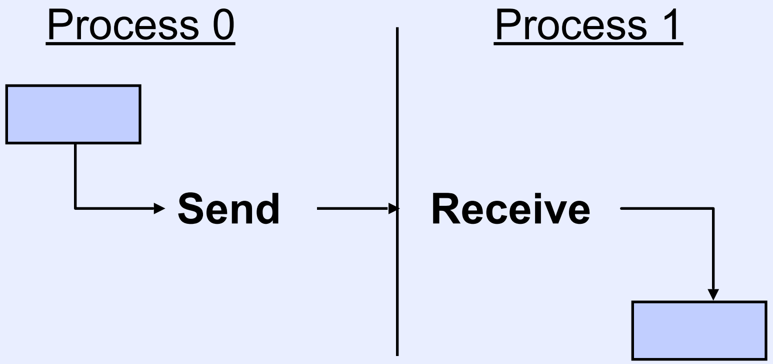
- User has full control (data partition, distribution): needs to identify parallelism and implement parallel algorithms using MPI function calls.

Principles of Message-Passing Programming

- The logical view of a parallel 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.

Sending and Receiving Messages

- Basic message passing process.
- Send data from one process to another



- <u>issues</u>
  - *Who will send data?*
  - *To whom is data sent or who will receive data?*
  - *Where is the data?*
  - *What type of data?*
  - *How much of data?*
  - *How to make sure send/ receive has been completed successfully?*

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 synchronize to perform interactions. Between these interactions, tasks execute completely asynchronously.
- Most message-passing programs are written using the *single program multiple data* (SPMD) model.

Message Organization in MPI

As discussed earlier - Message is divided into data and envelope

- data
  - *buffer*
  - *count*
  - *datatype*
- envelope
  - *process identifier (source and destination rank)*
  - *message tag*
  - *communicator*
- Follows standard argument order for most functions
  - *Call MPI_SEND (buf,count,datatype, destination, tag, communicator, error)*

The Building Blocks: Send and Receive Operations

- The prototypes of these operations are as follows:

```
send(void *send_buf, int no_elems, int dest)
receive(void *recv_buf, int no_elems, 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.

Traditional Buffer Specification

Sending and receiving only a contiguous array of bytes:

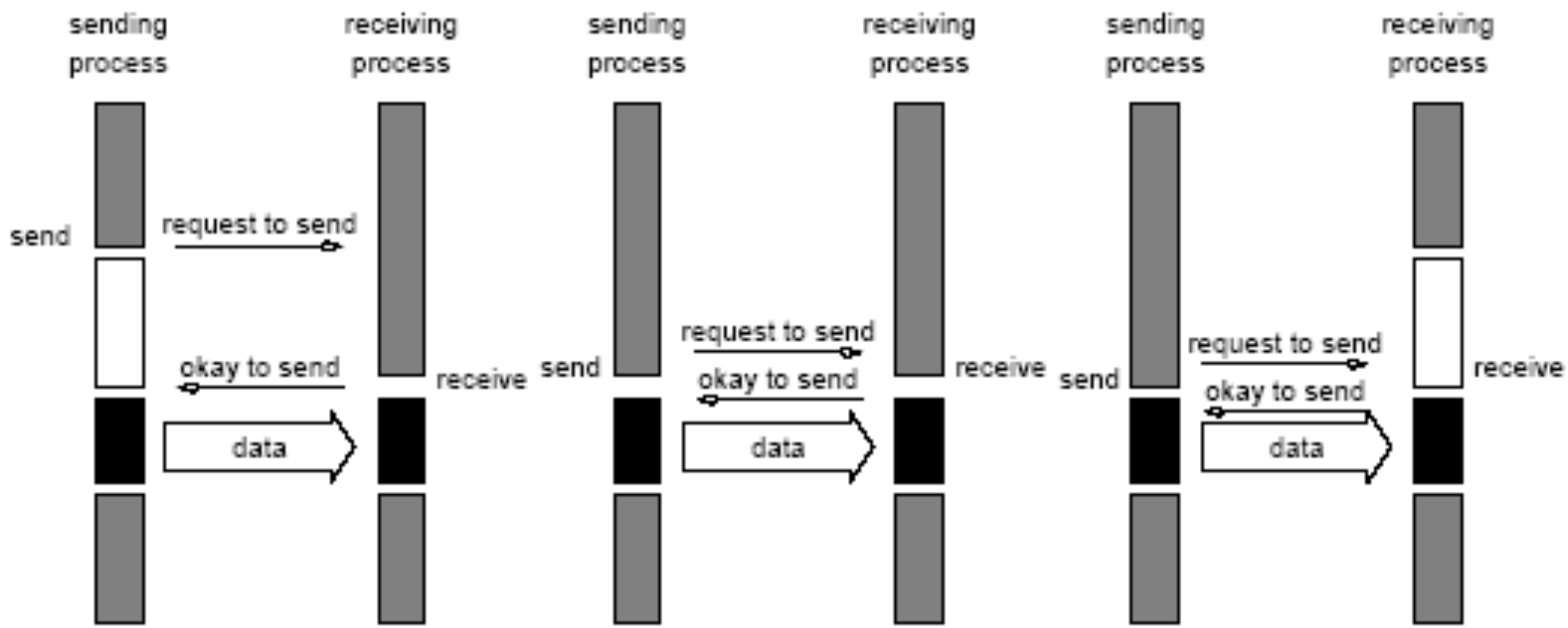
- ➢ Requires pre-packing dispersed data
- ➢ *Buffer* in MPI documentation can refer to:
  - *User defined variable, array, or structure*
  - *MPI system memory used to process data (hidden from user)*

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

Ref: Introduction to Parallel Programming; Ananth Grama, A. Gupta, G. Karypis and V Kumar



(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

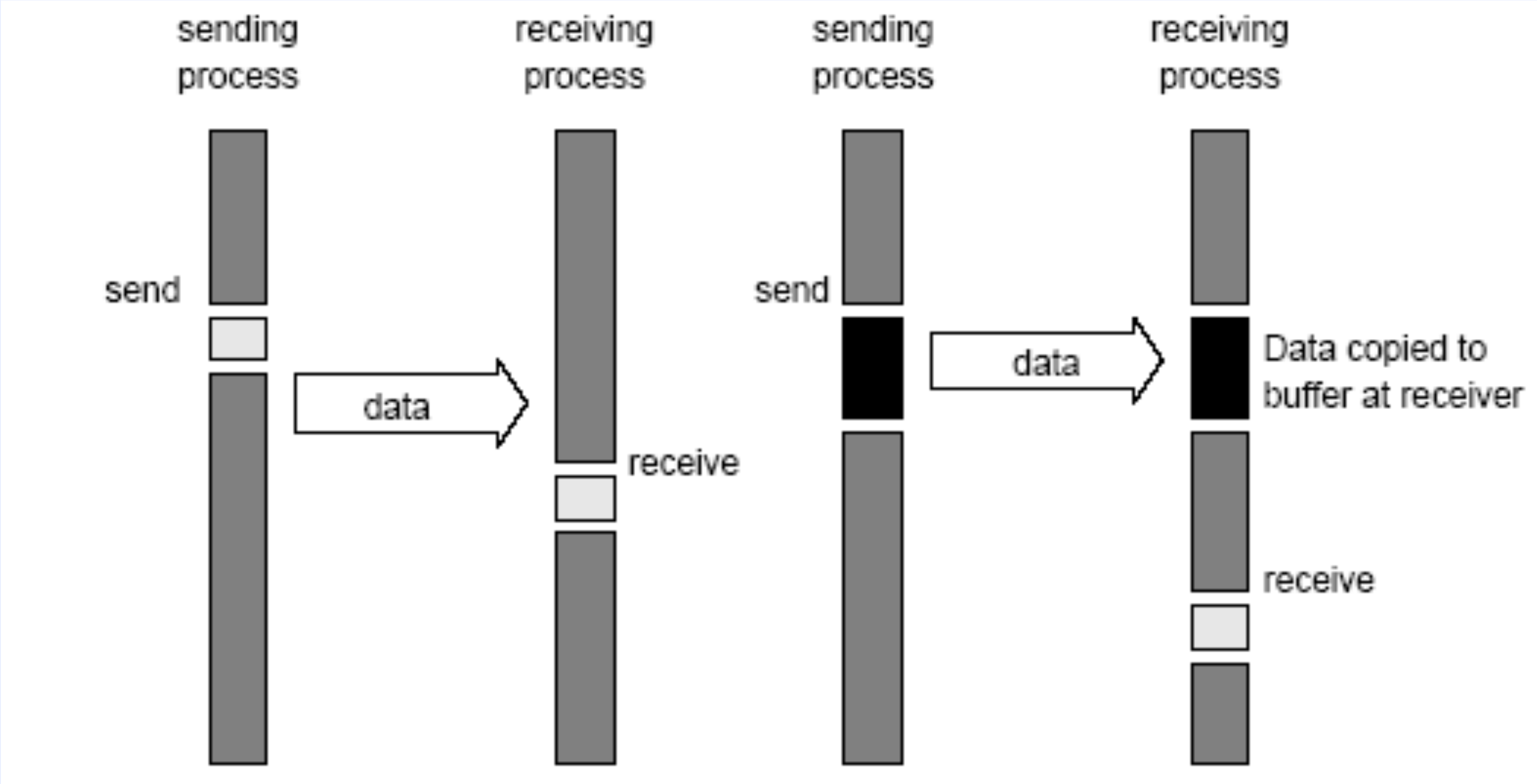
Handshake for a blocking non-buffered send/receive operation. It is easy to see that in cases where sender and receiver do not reach communication point at similar times, there can be considerable idling overheads.

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

Ref: Introduction to Parallel Programming; Ananth Grama, A. Gupta, G. Karypis and V Kumar



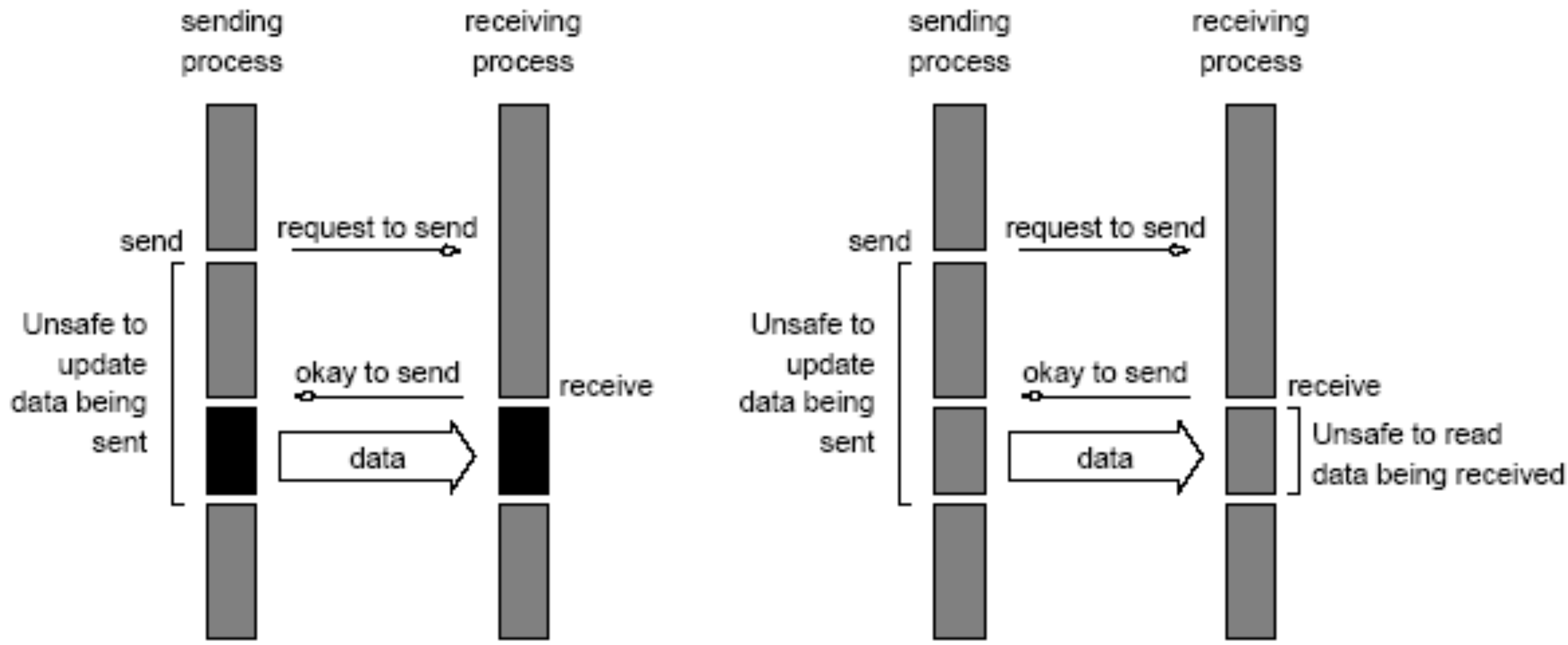
Blocking buffered transfer protocols: (a) in the presence of communication hardware with buffers at send and receive ends; and (b) in the absence of communication hardware, sender interrupts receiver and deposits data in buffer at receiver end.

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

Ref: Introduction to Parallel Programming; Ananth Grama, A. Gupta, G. Karypis and V Kumar

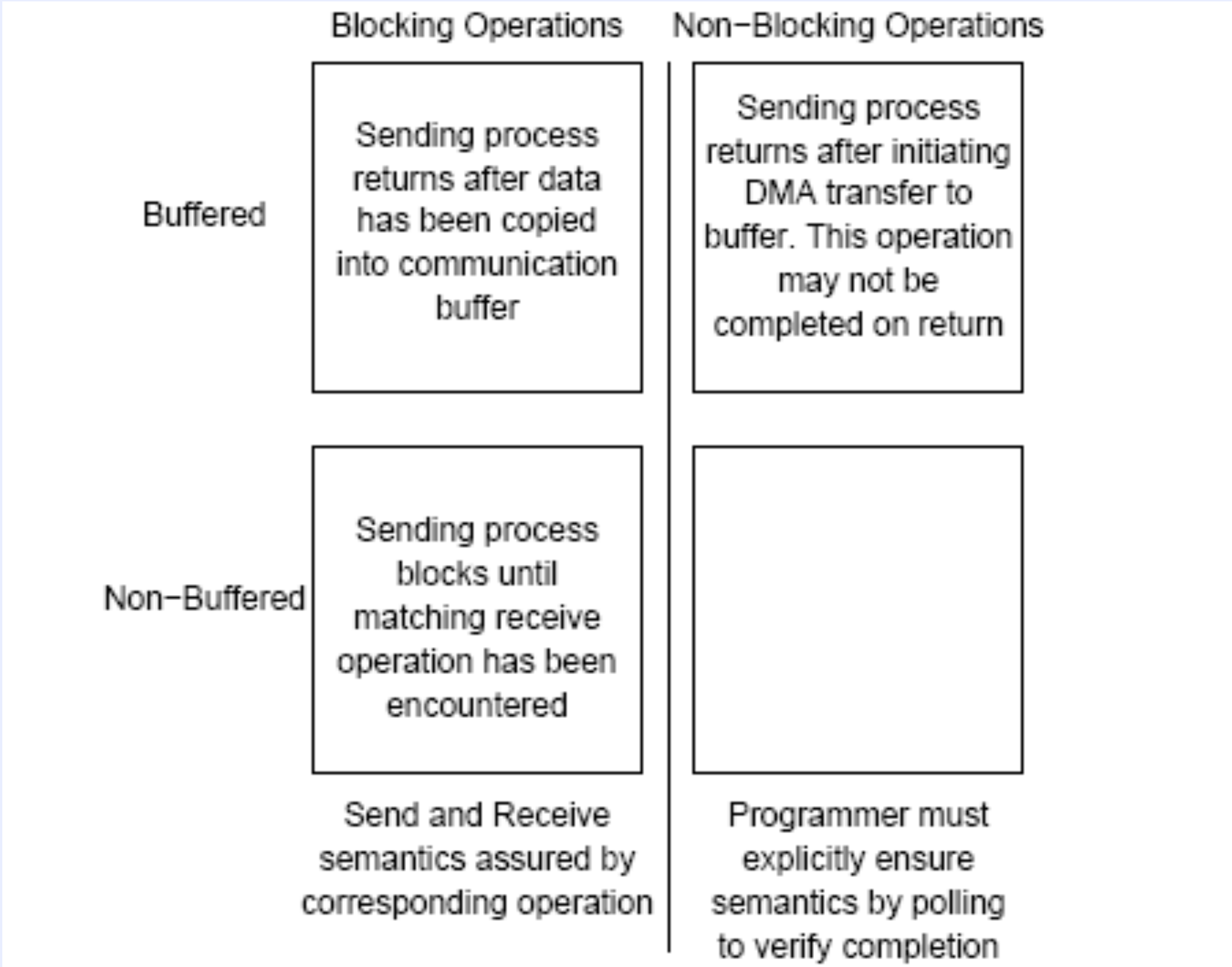


(a) Without hardware support (b) With hardware support

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

Send and Receive Protocols

Ref: Introduction to Parallel Programming; Ananth Grama, A. Gupta, G. Karypis and V Kumar



Space of possible protocols for send and receive operations.