The most difficult approach.

# Message-Passing Programming

Jingke Li

Portland State University



Advantage - Cluster parallelization

## Overview

*Hardware Characteristics of Message-Passing Systems:*

- Nodes are independent computers with private memory → full capacity
- Processes communicate via message passing through an interconnection network

*Programming Approaches:*

- Explicit Message-Passing Programming (*traditional*)
  - Partition and map data and computation
  - Explicit message passing

  *e.g.* MPI

This involves →

(there is no shared memory!) no unified system view

EP - require partition, but no message passing.

New approach

- Implicit Message-Passing Programming (*experimental*)
  - Based on the Partitioned Global Address Space (PGAS) model

  *e.g.* Chapel, CAF, UPC, X10 (CRAY ? ? IBM)

The final executable will contain MPI derived from the source code by the compiler (like openMP) User must be aware that Global Space is partitioned.



Cheap parallel system: Connect any number of PC's into a cluster

Difference between TOP-500 and simple approach - the way machines are connected. (Interconnection is very important)

Message Passing System - Set of computing units connected together.

## Explicit Message-Passing Programming

Hard to acomplish

Theoretically, each node on a message-passing system can execute a completely independent program. In the real world, however, user typically write programs with one of the following styles:

Dominating approach

- *SPMD Style:* (SPMP = Same Program Multiple Data) Kinda Like GPU

  All processes execute the *same* program; control statements in the program customize the code for individual processes. For example,

```
compute(my_nid)
if (my_nid == source)
  send data to dest
if (my_nid == dest)
  receive data from source
```

Alternative approach

- *Master-Slave Style:*

  One copy of a master program and many copies of a slave program.

Do some special task like IO



## Basic Programming Issues

data ≠ computation (computation desccibes operations on data)!

#1 ▸ Partition and Mapping — (go together)

Partition data and computation, and map them to processes.

- Which partition first, data or computation?
- How to select a mapping strategy?
- There are usually multiple choices need to pick the one that will perform the best.

# ▸ Communication — Message passing itself

Pass messages between processes to facilitate data sharing and computation synchronization.

- Figure out time and location for each communication
- Figure out senders and receivers for each communication

Mostly used

- Select proper communication routines
  - which one is better?

e.g. send multiple small or one big message?
- need to figure out correct pair

Approach Depends on application!

## Partition and Mapping

*Approach 1: Partitioning Computation First*

- Decompose the computation workload into disjoint tasks, and map the tasks to the processes first. Partition and distribute data later.
- Suitable for applications with task-based parallelism.
- However, it is not suitable for large-scale message-passing systems — The tasks from the same program are likely to access the same set of data. Hence a large amount of data may have to be replicated or passed from node to node.

In General doesn't work very well for large scale applications

Not many applications fit into this model!



## Partition and Mapping (cont.)

*Approach 2: Partitioning Data First*

- Decompose the data into small portions, and map them to the processes. For each data portion, the associated computation is carried out on the assigned process.
- For many scientific applications, data and computation are tightly coupled. For these applications, computation mapping can be derived from data mapping.

Natural rule → The "Owner Computes" Rule: [Most of the cases used as default approach]

Every data item has a "host" process (the "owner"). The owner is responsible for updating the data's value. → writing into



Most of the cases you don't even need to worry about type of computation!

Observation: in most of the cases some computation is the same for all elements of a data set

Extract and separate reads and writes. Process responsible for write is responsible for the data item (it owns it)

If there are multiple owners then they all update it's own copy in Parallel
Only owners can execute update.

Second piece → which way to go?

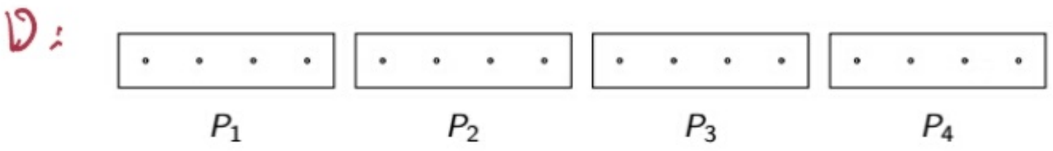
e.g. Mapping 3D array to 2d can be done in multiple ways.

If block partitioning is used then at some point some cpus will run out of work since data converges in a triangle shape

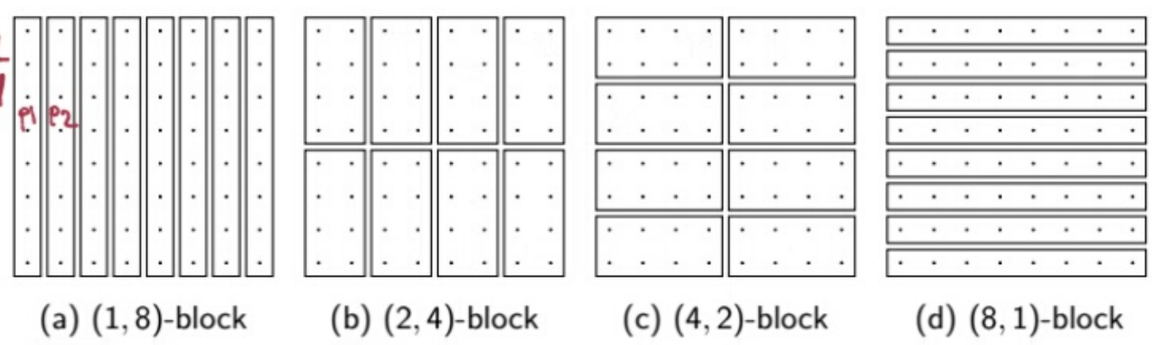
## Data Mapping Strategies

- *Block* — Partition data into blocks of contiguous elements, and assign each block to a different process.
  - simple, easier to implement, but needs to know # of processes

eg. mapping 2D to 1D:

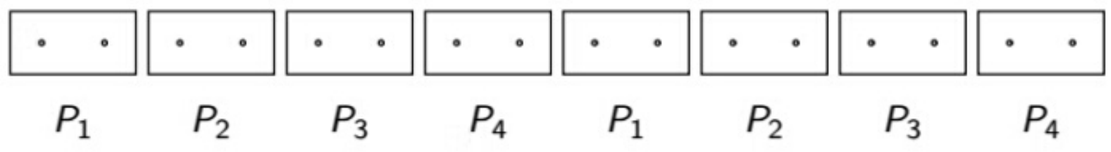



ways it can be done
Each strip can be assigned to a cpu or process




(a) $(1, 8)$-block (b) $(2, 4)$-block (c) $(4, 2)$-block (d) $(8, 1)$-block

- *Cyclic* — Partition data into small blocks, then assign them to processes in a round-robin fashion.
  - more adaptable to environment, and better for load balancing






## How to Select Data Mapping Strategies?

Data mapping can have a big impact on a program's overall performance.
However, there is no easy way to find the best mapping strategy. (No theoretical way) Run and see!

Many issues are involved:

But there are some patterns that can be used.

#1
- Communication patterns
  - Different mappings may result in different communication patterns, which in turn induce different costs

Try to avoid communication if possible

- Amount of data to communicate
  - *Boarder vs interior* — Boarder data need to be communicated to neighbors, while interior data are used only for local computation
  - Different mappings may affect the ratio between boarder data and local data (*a.k.a* surface-to-volume effect)

Border / Interior ratio

- Data alignment
  - Coordinating the mappings of multiple data objects

multiple arrays

- Load balancing
  - Computation may not always be uniform across a data domain



For a concrete application pick the one that works the best!

## Communication

- *Point-to-Point Communication* —
  A single pair of send and receive.
- *Collective Communication* —
  Multiple pairs of send and receive working synchronously and collectively.
- *One-Sided Communication* —
  One-sided communication requires involvement of only one side, either the sender or the receiver, but not both.

Used when its hard to predict who is receives (very expensive and complex)



## Send/Receive Primitives (There are dosens of routines available)

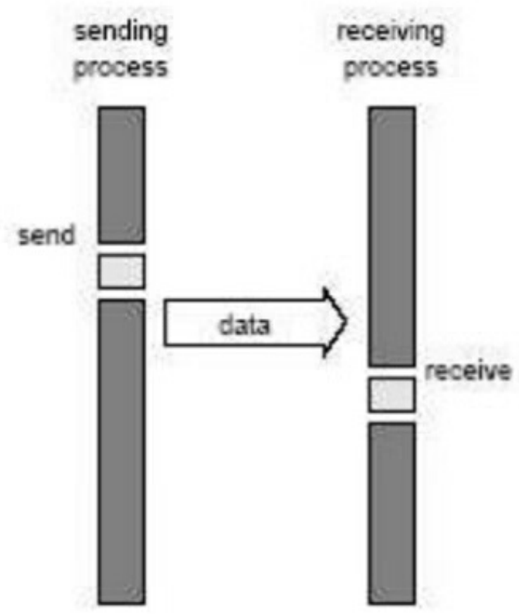
Send and receive are two message-passing primitives. Their general control flows are described below:

*Send* —

- User program issues a send() call;
- System copies data from user space to system buffer;
- System establishes connect to the receiver;
- System sends data to the receiver.

*Receive* —

- User program issues a receive() call;
- System checks for the expected message in system buffer; if it's not there, block and wait;
- System copies data to user space.



## Blocking vs. Non-Blocking

From the users perspective it's ok to overwrite content of the buffer

A send or receive routine can be either *blocking* or *non-blocking*.

A blocking send/receive routine will block until it is "safe" to return.

Safe =

- For a blocking `send()`, safe means that the message data can be modified and the communication buffer can be reused.

Safe receive =

- For a blocking `receive()`, safe means the message has been received and is available for use. If the message has not arrived at the time the receive routine is issued, it will wait until the message arrives.

*Note:* If not careful, blocking sends and receives can lead to deadlock.



## Blocking vs. Non-Blocking (cont.)

A non-blocking routine always returns immediately. It does wait for the send or receive action to finish. The benefit is that the user program can quickly move on to do something useful, instead of just waiting.

- When a non-blocking `send()` returns, it is not safe to alter the message data or to reuse the communication buffer.

  *Note:* No matter how long one waits after the routine returns, there is no guarantee that the send is finished. (A companion check routine is often provided to test for the completion of the send.)

- A non-blocking `receive()` returns after checking local buffer, regardless whether the expected message has arrived or not; if it's the latter case, then process will *not* get the message.

Receiver → Nonblocking gives more control → receive message from multiple senders, no order!



Sender → Do some other job if receiver is not ready.

There exists a routine that allows to check if buffer reuse is safe

Send doesn't block, but receiver blocks. (strongest version, not as efficient as other versions).
Advantage: no need to use system buffer (safer, but slower due to waiting).

## Synchronous Send-Receive Pairs

While a blocking receive waits for its message to arrive before returning, a blocking send does not. However, a stronger form of send can be defined.

A *synchronous send-receive pair* works in a strictly synchronous form —

- Neither routine would return unless the message is received.

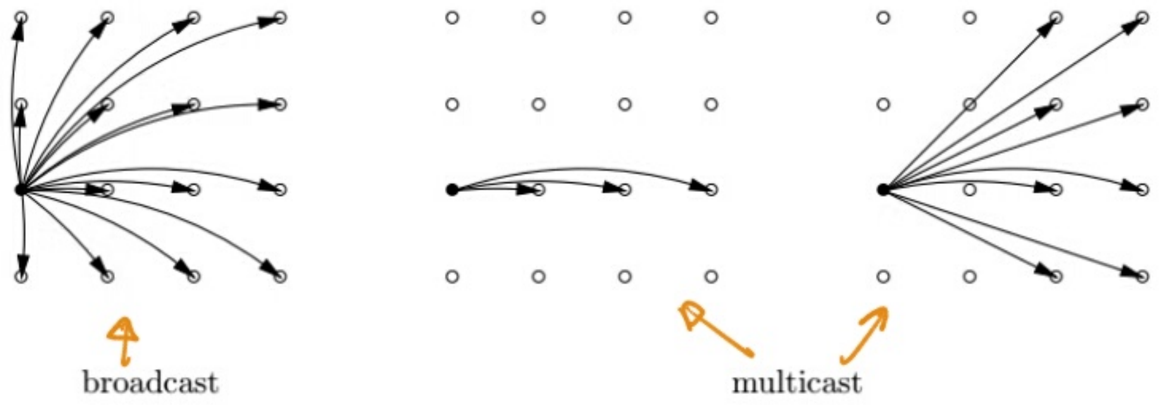
One advantage of this form is that no message buffer is needed.



## Collective Communications

Synchronous concurrent messages implementing global communication patterns. All processes need to participate!

- *One-to-Many* — Spread data from one node to many other nodes.
  - *Broadcast:* send same data to every other node.
  - *Multicast:* send same data to a set of nodes. (subset of nodes)
  - *Scatter:* send different data to different nodes.
    - used same pattern as Broadcast






sometimes processes don't send/receive but help to propagate

Gather is still usually faster then individual messages, because gather avoids collisions.

Gather is complex requires all messages to be the same size e.g. quicksort won't work.

## Collective Communications (cont.)

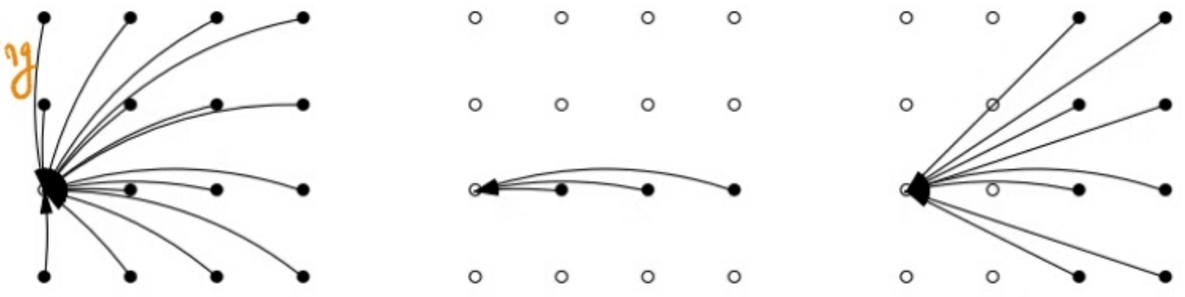
One receiver multiple senders data should be different! (usually used for reduce)

- *Many-to-One* — Combine messages from many nodes to one node.
  - *Reduce:* combine multiple data by a reduction function to a single data. (+, −, ×, /, *min*, *max*, etc.)
  - *Gather:* collect multiple data to a single node.

Reduce: O(log n)

Gather: O(n) receive at least n pieces

Gather – no combining each individual data item is saved.



- *Many-to-Many* — Concurrent, disjoint send/receive pairs.

parallel → shifting → Broadcasting → Receiving.

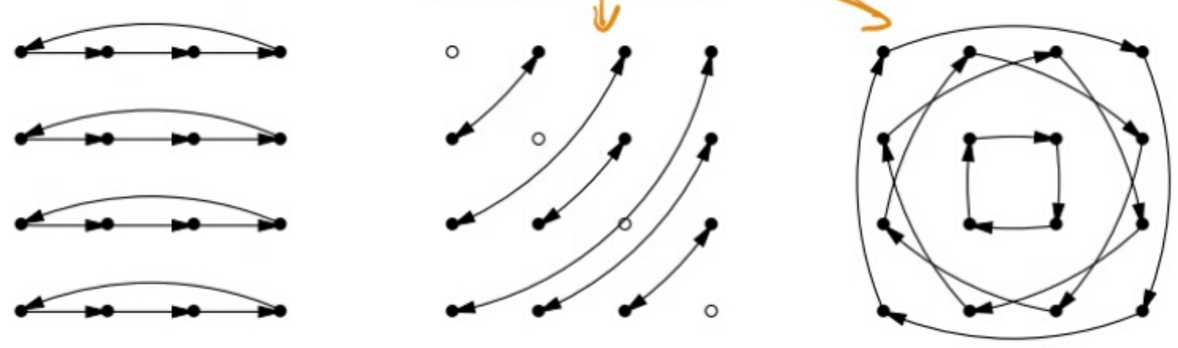
MPI has small support for these types



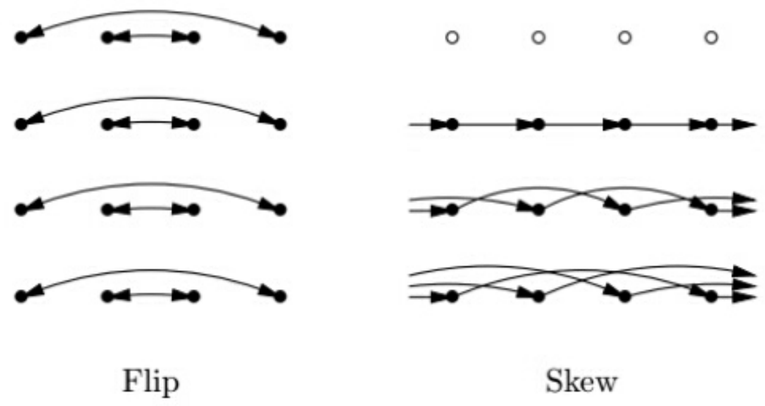
## Collective Communications (cont.)

- *Many-to-Many* — More examples.

not supported



Shift Transpose Rotate



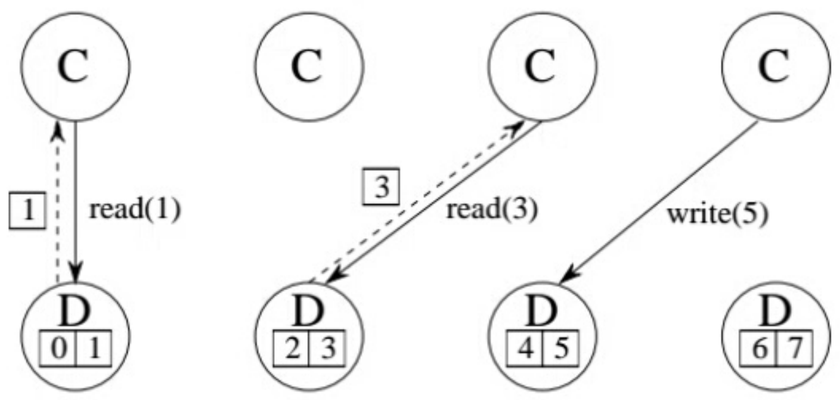
Flip Skew

Remote read and remote write.

## One-Sided Communications

The data producers (senders) are passive, they only respond to requests from the consumers (receivers).

"Remote reads" and "remote writes." Both forms are supported




Very complex multiple problems
many edge cases
complex to use
e.g.
what if data is not available on remote machine to read.

The semantics of one-sided communications can be tricky to define.

High overhead

- Acccess conflicts becomes an issue.
- Memory consistency becomes an issue.
- Operation granularity also becomes an issue.

where is safe to write to remote location?



## How to Reduce Communication Overhead?

General guide: how to create fast performing MPI applications.

- *Increasing locality* — localized communication (i.e. sender and receiver are neighbors) has a lower chance to interfere with other communications
- *Vectorizing messages* — grouping small messages together; sending fewer, larger messages Avoid startup overhead by grouping.
- *Utilizing collective communications* — collective communications are often implemented as library routines, which often are optimized on the given architecture

generally good choice →

- *Overlapping communication with computation* — hiding communication latency; this can be very effective (harder to do)

↓ For example use non-blocking.