Message-Passing Programming

Jingke Li

Portland State University

Overview

Hardware Characteristics of Message-Passing Systems:

- Nodes are independent computers with private memory
- 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
- Implicit Message-Passing Programming (*experimental*)
  - Based on the Partitioned Global Address Space (PGAS) model

  e.g. Chapel, CAF, UPC, X10

Explicit Message-Passing Programming

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:

- *SPMD Style:* (SPMP = Same Program Multiple Data)
  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
```

- *Master-Slave Style:*
  One copy of a master program and many copies of a slave program.

Basic Programming Issues

- Partition and Mapping —
  Partition data and computation, and map them to processes.
  - Which partition first, data or computation?
  - How to select a mapping strategy?
- Communication —
  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
  - Select proper communication routines

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.

Partition and Mapping (cont.)

Approach 2: Partitioning Data First

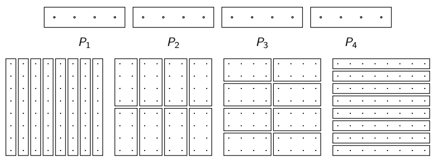
- Decompose the data into small portions, and map them to the processes. For each data portion, the associated computation is carried out on the assigned process.
- For many scientific applications, data and computation are tightly coupled. For these applications, computation mapping can be derived from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

The "Owner Computes" Rule:

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

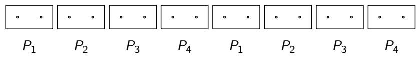
Data Mapping Strategies

- *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



(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.

Many issues are involved:

- Communication patterns
  - Different mappings may result in different communication patterns, which in turn induce different costs
- 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)
- Data alignment
  - Coordinating the mappings of multiple data objects
- Load balancing
  - Computation may not always be uniform across a data domain

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.

Send/Receive Primitives

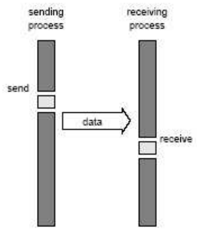
Send and receive are two message-passing primitives. Their general control flows are described below:

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

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.

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

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 —

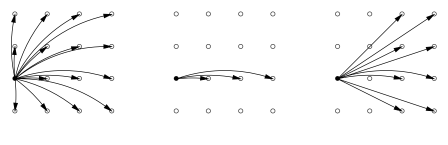
- Neither routine would return unless the message is received.

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

Collective Communications

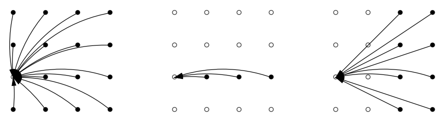
Synchronous concurrent messages implementing global communication patterns.

- *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.
  - *Scatter:* send different data to different nodes.

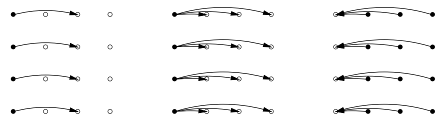


Collective Communications (cont.)

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

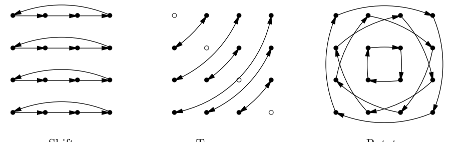


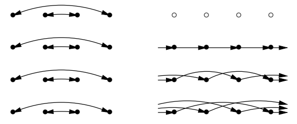
- *Many-to-Many* — Concurrent, disjoint send/receive pairs.



Collective Communications (cont.)

- *Many-to-Many* — More examples.

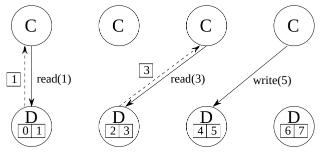




One-Sided Communications

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

"Remote reads" and "remote writes."



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

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

How to Reduce Communication Overhead?

- *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
- *Utilizing collective communications* — collective communications are often implemented as library routines, which often are optimized on the given architecture
- *Overlapping communication with computation* — hiding communication latency; this can be very effective