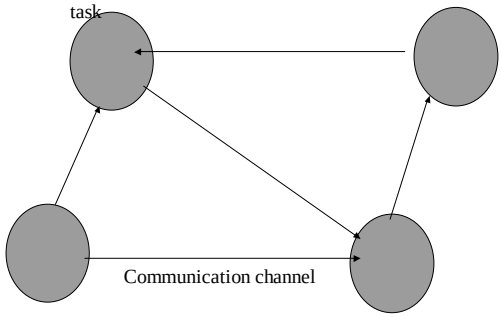
Message Passing Model

Task/Channel Model

- Developed by Ian Foster (look at his book)
- Represents a parallel computation as a set of tasks and interacts through channels with messages.
- For distributed memory machines
- Each task contains:
  - Local memory
    - Program instructions
    - Private data
  - Collection of I/O ports

A task can send local data values to other tasks via output port s and receive data via input ports

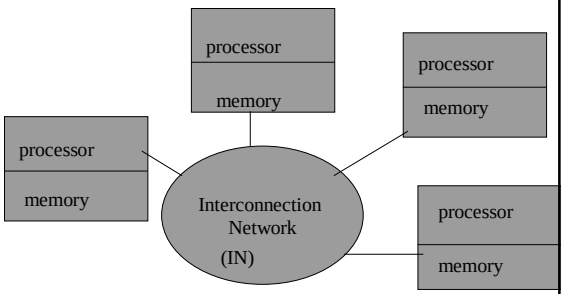


A channel is a message queue that connects one tasks output port with another tasks input port.

- Data values appear at the input port in the same order it was placed in the output port at the other end.
- Blocked: When a task tries to receive a value at an input port and no value is available, the task must wait until the value appears.
  - Receiving is synchronous
- However, a process sending a message never blocks even if previously sent messages along channel have not been received.
  - Sending is asynchronous
- Local data accesses easily distinguished from non-local data accesses.

Message Passing Model

- Based on Task/Channel model



hardware

- It is a collection of processors
- Process → task
- Local memory
- Private data
- IN supports message passing
- Processor A may send some of its local data values to processor B → indirect access to these values
- several processes run in parallel and communicate with one another by sending and receiving messages. The processes do not have access to shared memory.

Note

- IN: Implicit channel between every pair of processors
- Every process can communicate with every other process
- User specifies number of concurrent processes at the beginning of the program and remains constant throughout the execution of the program.
- Each process executes the same program
- Each process has a unique ID number
- Each process may perform different instructions as the program unfolds (asynchronous computations)
- Processes perform computations on its local variables.

- In the message passing model, messages are passed for both communication and synchronization
- Messages tell you the state of the process.
- PVM: Parallel Virtual Machine
  - Oak Ridge National Laboratory
  - Vaidy Sunderam (now at Emory Univ, Atlanta)
  - Facilitated the execution of parallel programs across heterogeneous collections of computers
  - In 1993 it became popular.
  - 1992—CRPC→ wanted a standard and MPI was standarized.

Message passing Computing

- Two Primary Methods:
  - A method of creating separate processes for execution on different computers
  - A method of sending and receiving messages
- Message passing library routine linked to conventional sequential programs(s) for message passing

Process in MPI

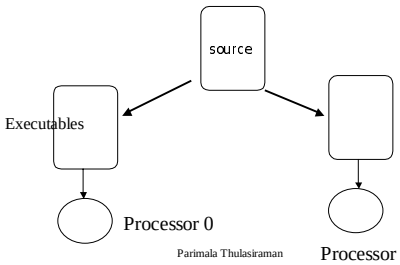
- Independent parallelizable subparts of a program
- A problem is divided into a number of concurrent processes
- Processes maybe executed on individual computers
- If there are more processes than computers, more than one process is executed on one computer in a time-shared fashion
  - Does not mean fast execution speed
  - Maybe hide network latencies
- Processes communicate by sending messages
  - Overhead in send and receive

Create processes

- Two methods of process creation:
  - Static process creation : all processes are specified before execution and the system will execute a fixed number of processes. One **master** process, remainder are **slaves**
  - Dynamic process creation: Processes can be created and their execution initiated during execution of other processes; Number of processes varies during execution (creation/destruction); More powerful technique; Incurs overhead (ParMetis)

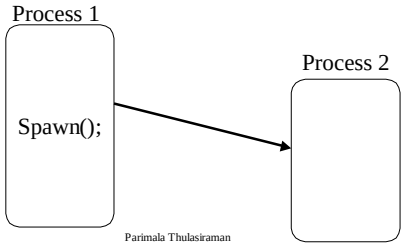
SPMD model (MPI)

- Static
- Different processes merged into one program



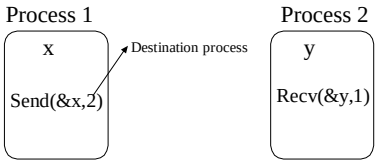
MPMD(PVM)

- Dynamic
- Completely separate and different program written for different processors



Message Passing Routines

- Send(): source process, originating the message
- Recv(): destination process to collect the messages being sent



Central communication routines

- *Send* and *receive*
- Always two processes involved in communication
- Send and receive exist in several variants

Purpose

- Message exchange may serve different purposes:
  - Exchange of data between sender and receiver that know each other and interaction planned in advance
  - Connection between sender and receiver was not planned in advance-receiver must be ready to receive and check the buffer for message arrival periodically
  - Synchronization: a process may send a message to notify that it has reached a certain point of program execution

Consequences

- Programmability: heavy burden on programmer
  - Details in data distribution
  - Task scheduling
  - Communication between tasks
  - Load balancing
  - Data replication
  - Maintenance of coherency

Error prone
Time consuming

- Efficiency:under programmers control: distribute and redistribute data and processes with any pattern, replicate data in variable sizes, aggregate communications
- Portability

Synchronous Message passing

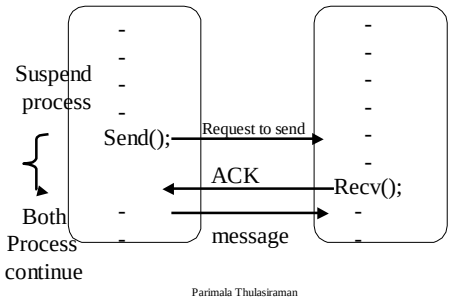
- No need for buffer storage
- Synchronous send routine could wait until the complete message can be accepted by the receiving process before sending the message
- Synchronous receive routine will wait until the message it is expecting arrives
- Synchronous routines perform 2 operations: transfer data and synchronize processes
- 3 way protocol:
  - Source sends a “request to send” message to destination
  - Destination is ready to accept, it returns an ACK
  - Upon receiving this ACK, source send the actual message

3 Way Protocol

- Source sends a “request to send” message to destination
- Destination is ready to accept, it returns an ACK
- Upon receiving this ACK, source sends the actual message

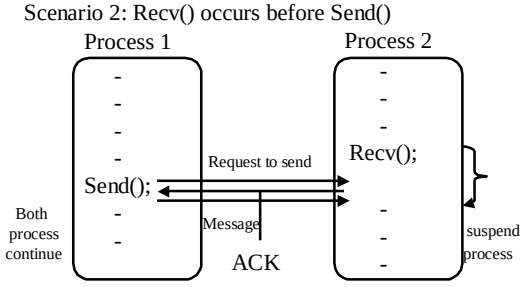
3 Way protocol

Scenario 1: Send( ) occurs before Recv( )



3 Way protocol

Scenario 2: Recv() occurs before Send()

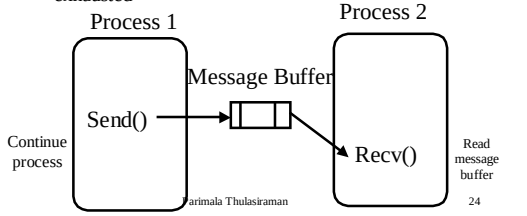


Blocking and Non-Blocking Message passing

- Blocking: Synonymous to synchronous message passing– Do not allow the process to continue with the next statement until the transfer is completed.
- Non-Blocking:Routines that return whether or not the message has been received
  - Message Buffer: between source and destination to store the message
  - Receive routine: message has to have been received if we want the message. If recv() is reached before send(), the message buffer will be empty and recv() waits for message

Message buffer

- Send routine: once the local actions have been completed and the message is safely on its way, the process can continue with subsequent work. Decreases overall execution time. But, buffer is of finite length and the send() routine can be held up because all the buffer space has been exhausted



Message Tag

- Used to differentiate between different types of messages
  - Send(&x, 2, 5);
  - Recv(&y,1,5);
- Wild Card: Destination accepts any message from any source (-1)-no special matching between destination and source is required

Broadcast,Scatter, Gather

- Broadcast: Sending the same message to all process concerned with the problem
  - Root process broadcasts the data to some processes
- Scatter: Sending each element of an array of data in the root to a separate process. The contents of the *ith* location of the array is sent to the *ith* process
  - Identify the group of processes and root process
  - Root process also receives a data element
- Gather: Having one process collect individual values from a set of processes
- Reduce: Gather + arithmetic/logical operation
  - Gathered and added by the root