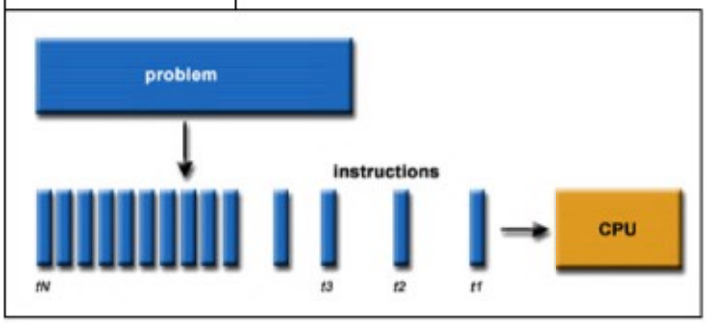
Lecture 1: Introduction

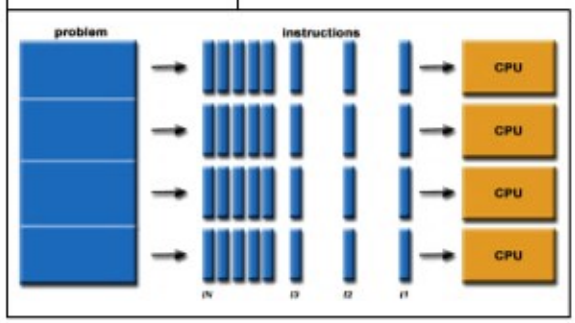
Definition

- Parallel Computer
  - Situation 1
    - + Single Computer
    - + Multiple Internal Processors
  - Situation 2
    - + Multiple Computer
    - + Interconnected
  - → To form a coherent high performance computing platform
- Parallel Programming
  - Programming multiple computers
  - A group of Computers with multiple internal processors
  - → To solve a problem at a greater computational speed than is possible with a single computer

Serial Computing



Parallel Computing



Demand for Computational Speed

- → Continual Demand for greater computational speed from a computer system than is currently possible.
- Areas requiring great computational speed
  - + Numeric Modelling
  - + Simulation of scientific and engineering problems
- Computations must be completed within a "reasonable" time period.
- Grand Challenge Problems
  - → one that cannot be solved in a reasonable amount of time with today's computers.
  - → Example
    - + Modelling large DNA Structures
    - + Global weather forecasting
    - + Modelling motion of astronomical bodies

Shazwan

Lecture 1: Introduction

Reasons

Not Using Existing Serial Machines

May not be able to reach the solution within a reasonable amount of time if using a single Von Neumann machine.

Limits

+ Physical constraints such as the speed of light
+ Cost of designing and manufacturing new chips.
+ Current speed of chips.

Using Parallel Machines

Job Level Parallelism

Embarrassingly Parallel Computation

Advantages

+ Much greater speeds can be achieved.
+ Used for enormous sized problems
+ Carrying out huge numbers of computations

The fastest computer in the world has a peak speed of about 1.1 Exaflops

Domains

+ Science
+ Engineering
+ Economics

Cannot be done by any existing serial machine but feasible to attack using parallel processing

Ideal Parallel Computation

In this context, a network is a collection of integrated processors that are communicating and sharing information to speed up the solution to a single task.

Immediately divided into completely **independent** parts that can be executed simultaneously

Parallel computers Classification

+ Type and number of processors.
+ Interconnection scheme of the processors.
+ Communication scheme.
+ Input/output operations.

Issues

How big of a collection?

+ A few very powerful, special purpose computers? (2-64)
+ Many smaller, standard microprocessors? (100-10,000)
+ A huge number of simple, one bit processors? (billions, trillions)

What kind of processors?

+ Special purpose?
+ Standard microprocessors or Simple?
+ One-bit processors?

How closely integrated is the collection of processors?

+ **Tightly coupled?**
Designed as a single architecture and a single system.
+ **Loosely coupled?**
Systems that can operate as a single logical system but can also function independently. Not designed as a single architecture.

Focus on a single task

+ focus more on task level parallelism rather than job-level parallelism
+ interested in situations where all the processors work together to speed up the solution of a single task.

What type of information is shared?

+ Data values, inputs, results?
+ Synchronizing information?

How do the processors communicate?

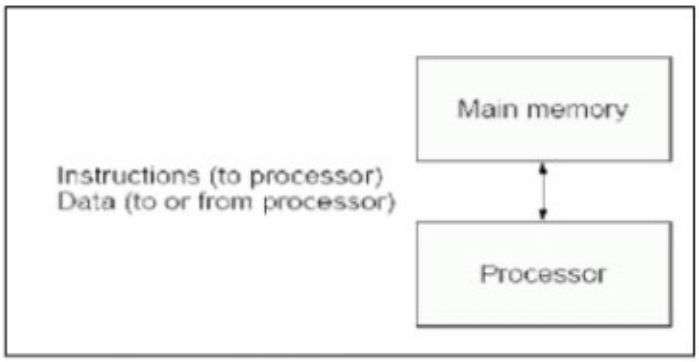
+ Shared memory?
+ External communications channel?

Shazwan

Lecture 1: Multi-Processor Computer Architecture

Conventional Computer → Consists of a processor executing a program stored in a (main) memory.

Each main memory location located by its address. Addresses start at 0 and extend to 2n-1 when there are n bits (binary digits) in the address.



Multi-Processor Computer Architecture

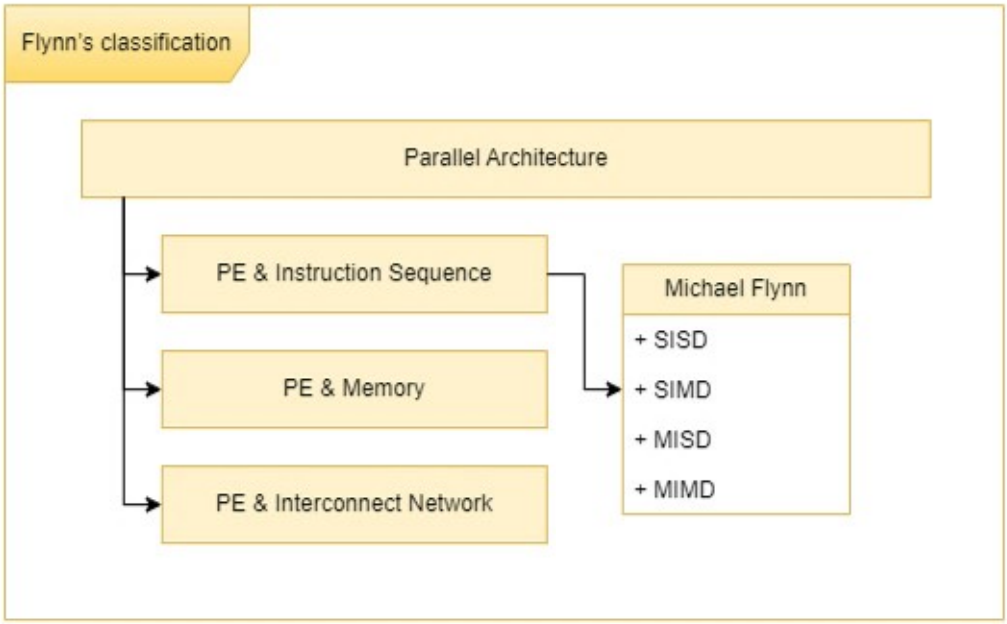
+ Flynn's Classification
+ Shared memory model
+ Distributed memory model
+ Distributed shared memory model

Taxonomy → **Three relationships** Relationship between PE and

+ the instruction sequence executed
+ the memory
+ the interconnection network

PE: Processing Element

Flynn's classification



Flynn Taxonomy

Michael Flynn (1996) developed a taxonomy of parallel systems based on the number of independent instruction and data streams.

Michael Flynn

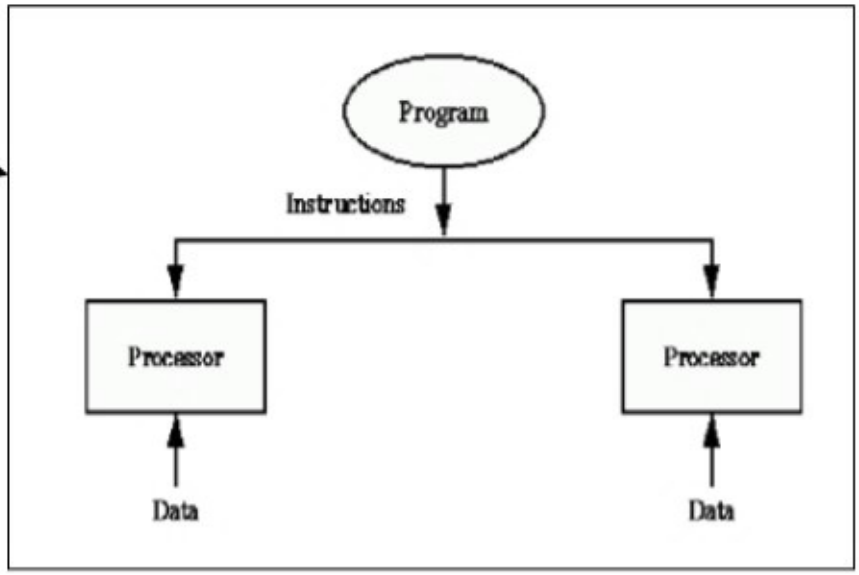
SISD	Single Instruction Stream - Single Data Stream
MIMD	Multiple Instruction Stream - Multiple Data Stream
MISD	Single Instruction Stream - Multiple Data Stream
SIMD	Multiple Instruction Stream - Single Data Stream

This is a good conventional sequential Von Neumann machine.

Shazwan

Lecture 1: Multi-Processor Computer Architecture

Single Instruction Stream-Multiple Data Stream (SIMD)



A **single instruction stream** is fetched by a control processor and broadcast to all other processors who execute it at the same time on their own local data values.

The processors executes the same instruction in synchronism, but using different data

This was the first form of parallel architecture developed because there are a number of important applications that mostly operate upon arrays of data

There is a **single instruction stream** executed by all PE.

Processors must either execute that instruction or be disabled and do nothing.

Characteristics of the SIMD Model of Parallelism

The instruction is fetched from control memory and broadcast to all PE by the control processor.

All PE have a unique identifier called their ID which they know and which can be used to determine the actions they are to take.

PE synchronize after each instruction. The control unit will not broadcast the next instruction until all PE are done with the current one.

Parallel [boolean expression]
<statement sequence>
End Parallel

For I = a **to** b **do in parallel**
<statement sequence>
End For

Meaning: Each processor evaluates the boolean selection expression. If it is true for your ID then execute the statement sequence, otherwise do nothing.

Typical Language Construct for SIMD Parallelism

Meaning: All processors whose ID is in the range a to b inclusive execute the statement sequence. Otherwise, they do nothing.

```
// Here will be sequential
Parallel [ ID<=9 ]
 X[ ID ] = 1;
 // sets the first 9 elements of X to 1
End Parallel
// Here will be sequential
```

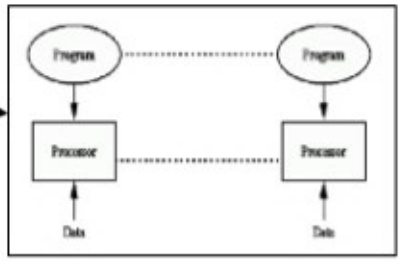
Note: Anything done outside a Parallel statement is done sequentially

```
// Here will be sequential
For ID = 1 to 9 do in parallel
 X[ ID ] = 1;
 // sets the first 9 elements of X to 1
End Parallel
// Here will be sequential
```

Shazwan

Lecture 1: Multi-Processor Computer Architecture

Single Instruction Stream-Multiple Data Stream (SIMD)



P0		P1
Input N values		Wait for inputs from P0
Send N/2 values to P1	→	
Sum up your values		Sum up your values
Wait for result from P1	←	Send sum to P0
Add sum to your own sum		Halt
Print results		
Halt		

General-purpose multiprocessor system - each processor has a separate program and one instruction stream is generated from each program for each processor.

Each **instruction operates upon different data.**

Both the shared memory and the message-passing multiprocessors so far described are in the MIMD classification.

Each processing element executes its **own program stream.**

There is **no central control**. All processors execute independently of all other processors.

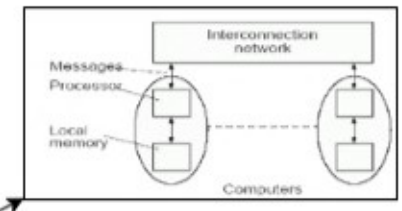
Characteristics of the MIMD Model of Parallelism

There is **no central clock**. Each processor executes at its own rate of speed.

There is **no automatic hardware synchronization**. **Synchronization** must be done in software by the programs themselves.

Each computer consists of a processor and a **local memory** that is not accessible by other processors.

The **messages** carry data from one processor to another as dictated by the program



Memory is **distributed** among the computers and each computer has its own address space.

Multi-Computers or Message-Passing Computers

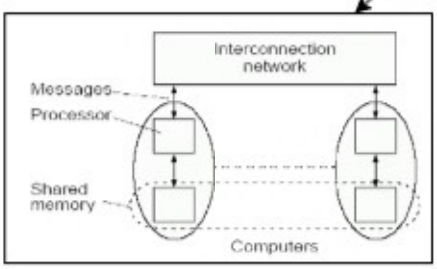
Each processor has access to the whole memory using a **single memory address space**.

The **interconnection network** is provided for processors to send **messages** to other processors which can include data that other processors may require for their computations.

A processor can only access a location in its own memory.

Distributed Shared Memory

This idea is called **shared virtual memory**, which gives the illusion of global shared memory even when it is distributed in different computers.



For a processor to access a location not in its local memory, message passing occurs to pass data between the processor and the memory location but in some automated way that hides the fact that the memory is distributed.

Shazwan

Lecture 2: Parallel Programming Models

Within the MIMD classification, each processor will have its own program to execute. This could be described as MPMD.

Another programming structure we may use is the Single Program Multiple Data (SPMD) structure.

In this structure, a single source program is written and each processor will execute its personal copy of this program; independently, and not synchronized.

In this case, some of the programs to be executed could be copies of the same program.

Multiple Program Multiple Data (MPMD) Structure

Single Program Multiple Data (SPMD) Structure

This source program can be constructed so that parts of the program are executed by certain computers and not others depending on the identity of the computer.

Typically only two source programs are written, **one for the designated master processor**, and **one for the remaining processors, which are called slave processors**.

For a master-slave structure, the programs could have parts for the master and parts for slaves.

MPMD & SPMD

You can execute those models in both

+ Shared Memory Multiprocessor Systems, or in
+ Programming Message-Passing between Multi-computers

With special parallel programming constructs and statements that allow shared variables and parallel code sections to be declared.

Executing on Shared Memory Multiprocessor Systems

Then the compiler is responsible for producing the final executable code.

Parallel Programming Languages

Threads

Sequential programming

Contain regular high-level language code sequences for individual processors. These code sequences can then access shared locations.

Use a regular sequential programming language and modify the syntax to specify parallelism.

Dividing the problem into parts that are intended to be executed simultaneously to solve the problem

Programming Message-Passing Multi-computers

A problem is divided into a number of concurrent processes that may be executed on a different computer.

Common approach is to use message-passing library routines that are inserted into a conventional sequential program for message passing.

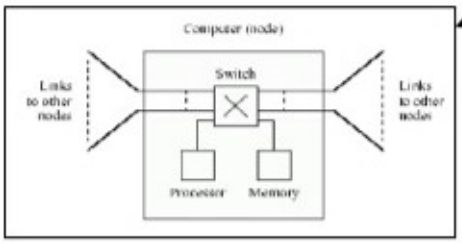
Processes communicate by sending messages; this will be the only way to distribute data and results between processes.

Shazwan

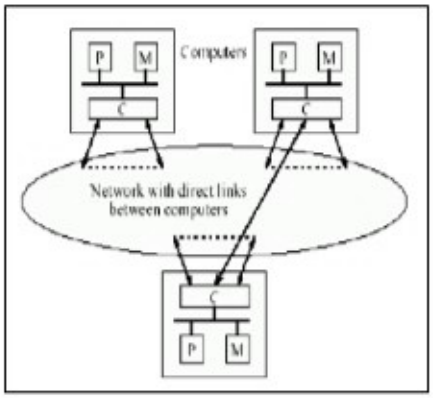
Lecture 2: Interconnection network for multi-computers or message-passing computers

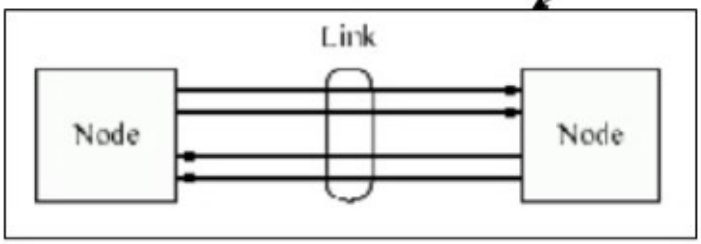
The Architectural Features of Message-Passing Multi-Computers → Static Network Message-Passing Multi-computers → Static interconnection networks are those that have direct fixed physical links between computers(nodes) for message passing as shown in the next slide.

Specially Designed Interface: Fully Integrated, Tightly Arranged Multicomputer → Here, each node has a switch that enables packets to be routed through the switch to other nodes without processors being disturbed.



A link between two nodes could be bidirectional or there may be two separate unidirectional links, one for each direction.





Network for Multi-Computers → **Purpose:** To provide a physical path for messages sent from one computer to another computer.

Network Criteria	
Bandwidth	number of bits that can be transmitted in unit time, given as bits/ sec.
Network latency	time to make a message transfer through network.
Communication latency	total time to send message, including software overhead and interface delays.
Cost	indicated by number of links in network*. (Ease of construction is also important.)

Network Criteria (Con't)	
Message latency / startup time	time to send a zero- length message. Essentially the software & hardware overhead in sending a message and the actual transmission time.
Diameter	minimum number of links between 2 farthest nodes in the network. Only shortest routes used. Used to determine worst case delays.
Bisection width of a network	number of links (or sometimes wires) that must be cut to divide network into two equal parts. Can provide a lower bound for messages in a parallel algorithm.

Each node has a link to every other node. E.g.: N nodes could be exhaustively interconnected with N-1 links. There are N(N-1)/2 links in all.

Completely Connected Network → Hence, static networks with restricted interconnections are used.

Feasible for a small system. As the size increases, the number of interconnection becomes impractical for economic and engineering reasons.

Types of such networks.	
+ Line/Ring	+ Mesh
+ Hypercube	+ Tree

Shazwan

Lecture 2: Interconnection network for multi-computers or message-passing computers

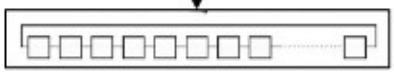
Completely Connected Network Line / Ring

A row of nodes with interconnections limited to adjacent nodes.

The two end nodes are the farthest away in a line and hence the diameter is N-1.

The line can be formed into a RING structure by connecting the free ends as shown in the next slide.

For RING where messages can pass in either direction, the diameter is N/2.



Each node requires two links, one to each neighboring node, and hence a N node ring requires N links.

Routing algorithms are used to find routes between nodes.

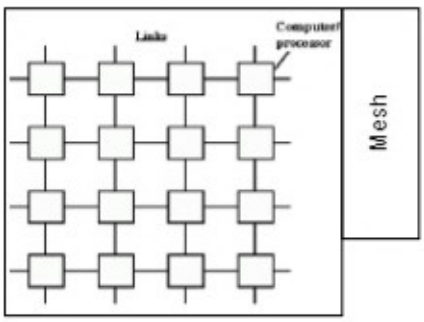
Completely Connected Network Mesh

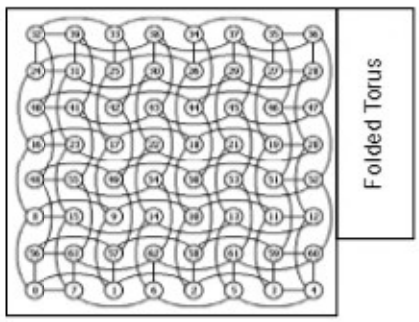
A 2D mesh can be created by having each node in a 2D array connect to all its four nearest neighbors.

The free ends of a mesh might circulate back to the opposite sides. Then the network is called a TORUS. Each node in a sqrt(N)* sqrt(N) torus links and there are 2N links in all.

The diameter of a sqrt(N) * sqrt(N) mesh is 2(sqrt(N-1)) since to reach one corner from the opposite corner requires a path to made across (sqrt(N-1)) nodes and down(sqrt(N-1)) nodes.

Three dimensional meshes can also be formed.





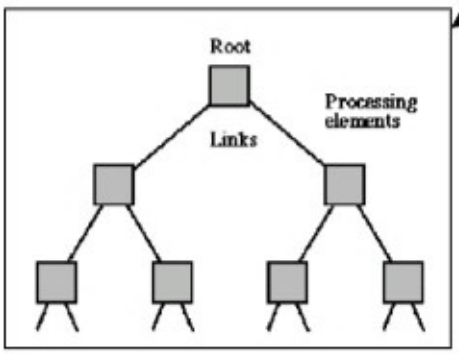
Meshes are convenient for many scientific and engineering problems in which solution points are arranged in 2D or 3D arrays.

Completely Connected Network Tree

A binary tree network is shown.

This particular tree is a complete binary tree as every level is fully occupied.

The height of the tree is the number of links from the root to the lowest leaves.



The tree network need not be complete nor binary.

In an m-ary tree, each node connects to m nodes beneath it.

Shazwan

Lecture 2: Interconnection network for multi-computers or message-passing computers

Completely Connected Network Hypercube

In a D-dimensional hypercube network, each node connects to one node in each of the dimensions of the network.

Each node is assigned a D-bit binary address when there are D dimensions. Each bit corresponds to one of the dimensions and can be 0 or 1, for the two nodes in that dimension.

In a 3D hypercube, the connections in the x, y, and z-direction form a cube as shown in the next slide.

Nodes in a 3D hypercube would have 3 bit address. Node 000 would connect to nodes with addresses 001, 010, 100. Node 111 would connect to nodes with addresses 110, 101, 011.

In a 5D hypercube, node 11101 connects to 11100, 11111, 11001, 10101, and 01101.

Routing in Hypercubes

Diameter of the network is given by log2n.

To describe a routing algorithm, let us route a message from a node S having a nodal address S = sn-1 sn-2...s1 s0 to a destination node having a nodal address D = dn-1dn-2...d1d0.

Each bit of D that is different from that of S identifies one hyper-cube dimension that the route should take and can be found by performing the Exclusive-OR function R=S xor D, operating on pairs of bits.

At each node in the path, the Exclusive OR function between the **current nodal** address and the **destination nodal** address is performed. Usually the dimension identified by the most significant 1 in R is chosen for the route.

The dimensions to use in the routing are given by those bits of R that are 1.

This routing algorithm is called the e-cube routing , or left to right routing algorithm.

Routing Example

The route taken from node 13 (001101) to node 42 (101010) in a six dimensional hypercube would be node 13 (001101) to node 45 (101101) to node 41 (101001) to node 43 (101011) to node 42 (101010).

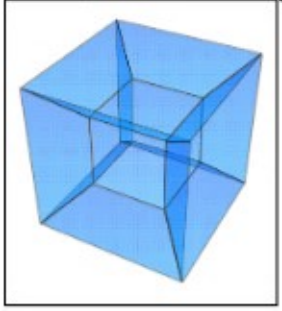
001101 xor 101010 → 100111

(The dimensions to use in the routing are given by those bits of R that are 1)

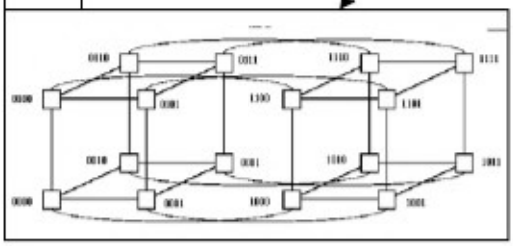
The next slide shows a 4-dimensional hypercube drawn as TWO 3-dimensional hypercubes with 8 connections between them. Hence the bisection width is eight (the bisection width is n/2 for an n-node hypercube).

A D-dimensional hypercube actually consists of two D-1 dimensional hypercubes with D-th dimension links between them.

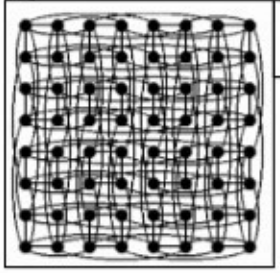
4D



4D



6D



Shazwan

Lecture 2: Interconnection network for multi-computers

Embedding

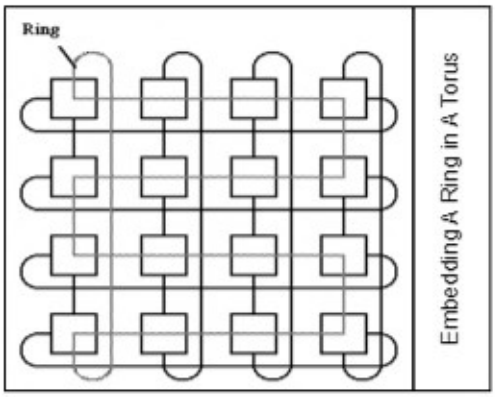
The term embedding, when applied to static networks, describes the mapping of nodes of one network on to another network, e.g.

It is important because we may have an algorithm that requires only communication to its nearest neighbours in a specified network.

A RING can be embedded in a TORUS. This particular embedding is perfect in that each link in the ring exists in the mesh.

Similarly a LINE can be embedded into a MESH.

The actual computer may use a different interconnection network. In that case we would want to arrange for an embedding that is perfect to minimize the communication delays.



Embedding A Ring in A Torus

A mesh/torus can be embedded into a hypercube by numbering the nodes of the mesh/torus.

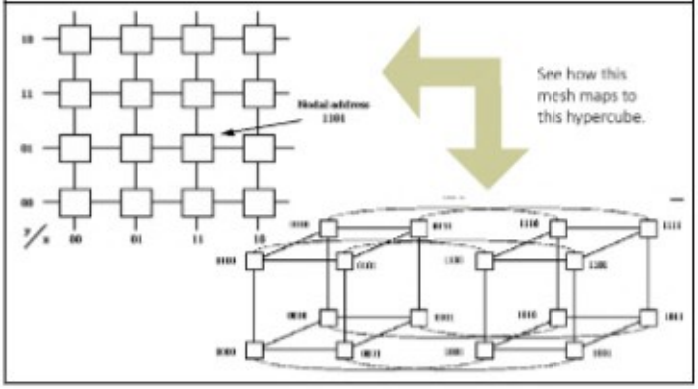
Each node has an address composed of an x-coordinate and a y-coordinate. Each coordinate is labelled in a gray code. In a gray code, adjacent numbers differ by one bit.

Since each node connects to nodes with addresses that differ by one bit, it follows that the mesh will embed a 4*4 mesh and all the links in the hypercube would be used shown in next slide.

Larger meshes requires larger hypercubes and in all cases only four links are needed at each node.

E.g. The number adjacent to 10 are 00(lower) and 11(higher).

Embedding a Mesh into a Hypercube



Dilation

Used to indicate the quality of the embedding.

Does the maximum number of links in the "embedding" network corresponding to one link in the "embedded" network?

Perfect embeddings, such as a line/ ring into mesh/ torus or a mesh onto a hypercube, have a dilation of 1.

Shazwan

Lecture 2: Communication methods and deadlocks

2 basic ways	
+ Circuit Switching	+ Packet Switching

messages can be transferred from Source to Destination

Communication Methods

Circuit Switching

Involves establishing path and maintaining all links in path for a message to pass, uninterrupted. All links are reserved for the transfer until the message transfer is complete.

Once a telephone connection is made, the connection is maintained until the completion of the telephone call.

It suffers from forcing all the links in the path to be reserved for the complete transfer. None of links can be used for other messages until the transfer is completed.

A simple telephone system (not using advanced digital techniques) is an example of a circuit-switched system.

Virtual Cut-Through

Can eliminate storage latency.

If the outgoing link is available, the message is immediately passed forward without being stored in the nodal buffer; i.e., it is "cut through."

If complete path were available, the message would pass immediately through to the destination.

However, if path is blocked, storage is needed for the complete message/ packet being received.

Packet Switching

In Packet Switching, buffers are extensively used.

Buffers also introduces latency due to storage.

Message divided into "packets" of information, each includes source and destination addresses for routing packet.

There is a maximum size for the packet, say 1000 data bytes. If message is larger than this, it is broken up into separate packets, and each packet is sent through network separately.

Buffers provided inside nodes to hold packets before they are transferred onward to the next node. This form called store-and forward packet switching, e.g. the postal service, where letters are moved from the mailbox to the post office and handled at intermediate sites before being delivered to the destination.

Enables links to be used by other packets once the current packet has been forwarded. Incurs a significant latency since packets must first be stored in buffers within each node, whether or not an outgoing link is available.

Shazwan

Lecture 2: Communication methods and deadlocks

Communication Methods: Wormhole Routing

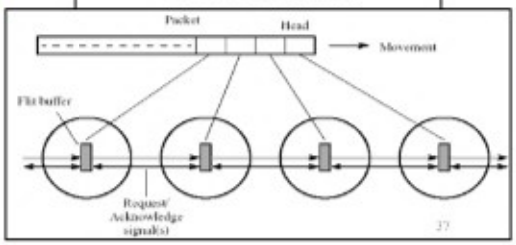
An alternative to normal store-and-forward routing to reduce the size of the buffers and decrease the latency.

Only head of message initially transmitted from source node to next node when connecting link available.

Message is divided into smaller units: flits (flow control digits).

Involves establishing path and maintaining all links in path for a message to pass, uninterrupted. All links are reserved for the transfer until the message transfer is complete.

Communication Methods: Wormhole Routing



Wormhole Routing: Request/acknowledge System

A way to "pull" flits along.

R/A reset to 0 by receiving node when ready to receive flit (its flit buffer empty).

Only requires a single wire between the sending node and receiving node, called R/A (request/acknowledge).

R/A set to 1 by sending node when sending node is about to send flit.

Sending node must wait for R/A = 0 before setting it to a 1 and sending the flit.

Sending node knows data has been received when receiving node resets R/A to a 0

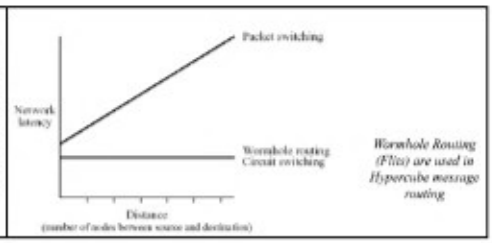
Deadlocks

Occurs when packets cannot be forwarded to next node because they are blocked by other packets waiting to be forwarded and these packets are blocked in a similar way such that none of the packets can move.

Example:

- Node 1 wishes to send a message through node 2 to node 3.
- Node 2 wishes to send a message through node 3 to node 4.
- Node 3 wishes to send a message through node 4 to node 1.
- Node 4 wishes to send a message through node 1 to node 2.

Network Delay Characteristics



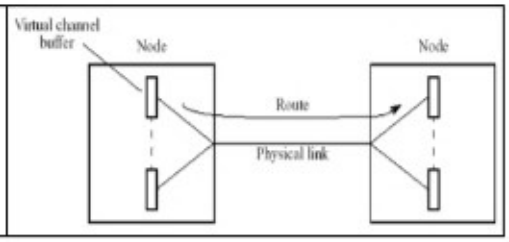
Virtual Channel

A general solution to deadlock.

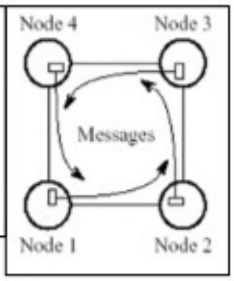
The physical links or channels are the actual hardware links between nodes.

Multiple virtual channels are associated with a physical channel and time-multiplexed onto the physical channel.

Virtual Channel



Deadlocks Example



Shazwan

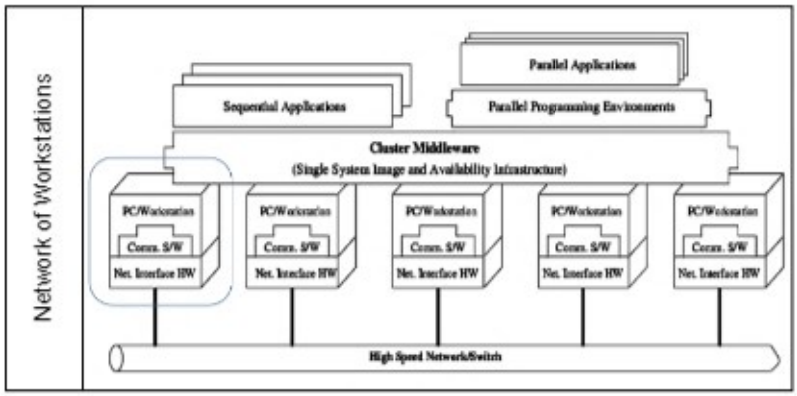
Lecture 2: Cluster Computing

A cluster of workstations (COWs) or network of workstations (NOWs) offers a very attractive alternative to expensive supercomputers and parallel computer systems for high-performance computing.

Simplest Cluster Computing

Key Advantages

+ Very high performance workstations and PCs are readily available at low cost.
+ The latest processors can easily be incorporated into the system as they become available.
+ Existing software can be used or modified.



Parallel Programming Software Tools for Clusters (More Later)

Parallel Virtual Machine (PVM)

+ Developed in the late 1980's.
+ Became very popular.
+ Now almost "gone".

Message-Passing Interface (MPI)

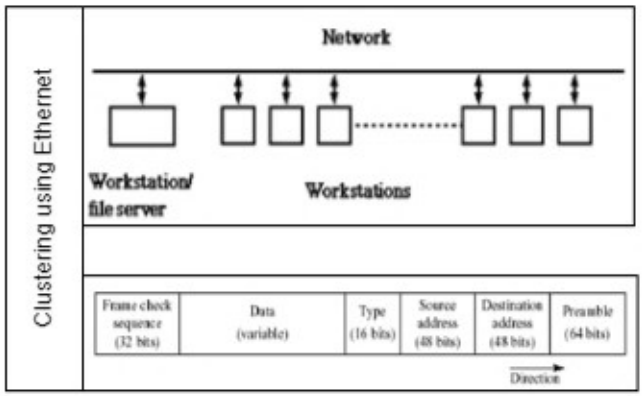
+ Standard was defined in 1990s.

Clustering using Ethernet

- Common communication network for workstations.
- Consisting of a single wire to which all the computers attach.

All transfers between a source and a destination are carried in packets (or frames) along the single wire, one bit at a time. The packet carries the source address, the destination address, and the data.

All transfers between a source and a destination are carried in packets (or frames) along the single wire, one bit at a time. The packet carries the source address, the destination address, and the data.



Ring Structures

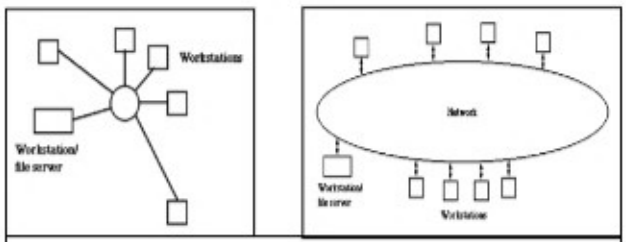
- Other forms of networks are also in wide use for work stations.

Examples

+ Token rings/FDDI networks
+ Star-connected networks

Provides the highest interconnection bandwidth.

Point-to-Point Communication



Example: Ring Structures

Each cluster could be Ethernet, ring or star-connected.

Individual workstations make request to the fileserver (or to each other) on the network.

Usually workstation networks are organised into groups of interconnected networks (workstation clusters).

Various point-to-point configurations can be created using hubs and switches. Examples:

High Performance Parallel Interface (HIPPI), Fast (100 MHz) and Gigabit Ethernet, and fiber optics.

Shazwan

Lecture 2: Cluster Computing

Overlapping Connectivity Networks

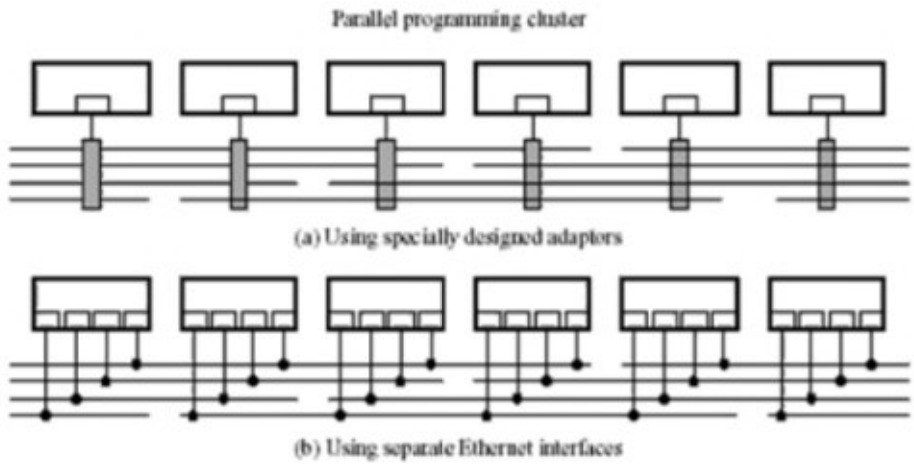
For parallel programming, workstations could be connected together into a dedicated parallel programming cluster.

A single Ethernet is the simplest method.

An interconnection structure that has higher performance than Ethernet and responds to many simultaneous local communications well is more desirable.

The concept of using multiple Ethernet lines configured as shown in the figures a and b in the next slide can be classified under the general classification of overlapping connectivity networks.

Overlapping Connectivity Networks



Overlapping Connectivity Networks

Overlapping connectivity networks have the characteristic that regions of connectivity are provided and the regions overlap.

There are several ways that overlapping connectivity can be achieved. An example using multi-tiered Ethernet switches.

Thoughts: Parallel Computing v.s. Distributed Processing

Parallel Computing	Distributed Processing
PEs are clustered together (tightly coupled)	Processing nodes are geographically distributed
Requires a fast communication network	Can do without a fast communication network
Nodes are homogeneous in general	Heterogeneous nodes are very common
Motivation: Fast processing	Information/Resource sharing and fault tolerance
May use shared memory	Memory is always distributed

Shazwan

Lecture 3: Computational Speed

In all forms of MIMD multiprocessor / multicomputer systems, it is necessary to divide the computations into tasks or processes that can be executed simultaneously.

Process

Potential for Increased Computational Speed

"size of the computation between communication or synchronization points"

$$\frac{Computation}{Communication} = \frac{Computation\ Time}{Communication\ Time} = \frac{t_{comp}}{t_{comm}}$$

Can be used as a granularity metric

Sometimes granularity is defined as the **size of the computation between communication or synchronization points.**

Generally we want to increase the granularity to reduce the cost of process creation and inter-process communication, but of course this will likely reduce the number of concurrent processes and amount of parallelism. A suitable compromise has to be made.

The size of a process can be described by its granularity, which is related to the number of processors being used.

In coarse granularity, each process contains a large number of sequential instructions and takes substantial time to execute.

In fine granularity, a process might consists of few instructions or perhaps even one instruction.

Medium granularity describes the middle ground.

Speed-Up Factor

The factor S(n) gives the increase in speed due to using a multiprocessor system.

For comparing a parallel solution with a sequential solution, we will use the fastest known sequential algorithm for running on a single processor. The underlying algorithm for a parallel implementation might be (and is usually) different.

A measure of relative performance between a multiprocessor system and a single processor system in the speed- up factor, S(n).

$$\frac{Execution\ Time\ using\ a\ Single\ Processor}{Execution\ Time\ using\ n\ Processors\ in\ a\ Multiprocessor\ System} = \frac{t_s}{t_p}$$

The speed up factor can also be cast in terms of computational steps:

$$S_{(n)} = \frac{Number\ of\ computational\ steps\ using\ one\ processor}{Number\ of\ parallel\ computational\ steps\ with\ processors}$$

The maximum speedup is n with n processors which is also called a linear speed up (or embarrassingly parallel)

Linear Speedup

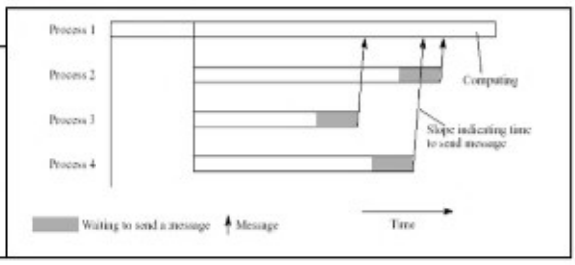
Superlinear speedup can occur in search algorithms.

It is reasonable to expect that some part of a computation cannot be divided at all into concurrent processes and must be performed serially.

Space-Time Diagram of a Message Passing Program

Superlinear Speedup

Space-Time Diagram of a Message Passing Program



Where S(n) > n, may be seen on occasion, but usually this is due to using a suboptimal sequential algorithm or some unique feature of the architecture that favours the parallel formation.

One common reason for superlinear speedup is the extra memory in the multiprocessor system which can hold more of the problem data at any instant, it leads to less relatively slow disk-memory traffic.

During the initialization period or period before concurrent processes are being setup, only one processor is doing useful work, but for the rest of the computation, additional processors are operating in parallel.

The figure on the next page illustrates a space-time diagram of a message-passing program that has one process on each of four processors. It starts with one process and later others begin to execute and send messages back. In general we would expect periods when processors are idle and are waiting to send their

Shazwan

Lecture 3: Computational Speed

Illustrated is the case with a single serial part at the beginning of the computation which cannot be parallelized, but the remaining parts could be distributed throughout the computation.

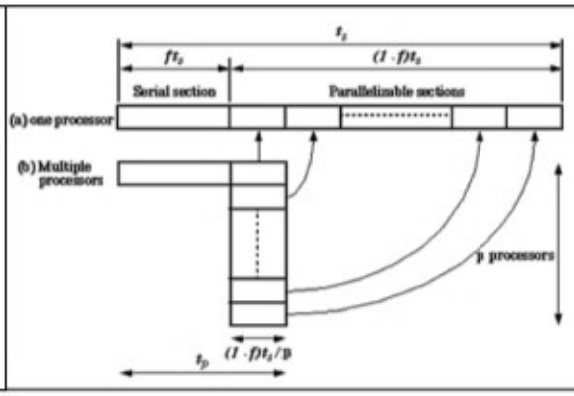
Maximum Speedup

The speed up factor S(p) is therefore given by

$$S(p) = \frac{t_S}{ft_S + (1-f)t_S/p} = \frac{p}{1+(p-1)f}$$

If the fraction of the computation that cannot be divided into concurrent tasks is f, the time to perform the computation with p processors is given by $f_{ts} + (1-f)t_s/p$ as shown in the next slide.

Parallelizing Sequential Problem



$$S(p) = \frac{t_S}{ft_S + (1-f)t_S/p} = \frac{p}{1+(p-1)f}$$

Amdahl's Law

$$S(p) = \frac{p}{1+(p-1)f}$$

Amdahl's Law

It shows S(n) plotted against number of processors and against f.

If a significant increase in speed of computation is to be achieved, the fraction of computation that is executed by concurrent processes needs to be substantial fraction of the overall computation.

Even with infinite number of processors, maximum speedup limited to 1/f.

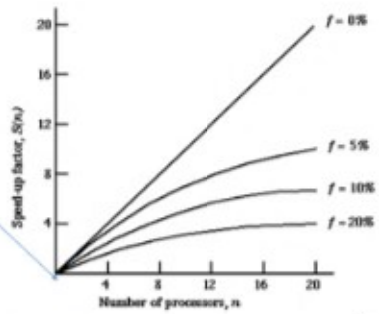
I.e., $S(n) = 1/f$ as $n \to \infty$

Amdahl's Law

Amdahl's Law

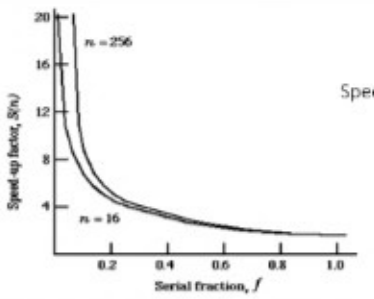
f = sequential portion

Speed-up against number of processors



Amdahl's Law

Speed- up against serial fraction, f



$$E = \frac{\text{Execution time using one processor}}{\text{Execution time using a multiprocessor} \times \text{number of processors}}$$

$$= \frac{t_s}{t_p \times n}$$

which leads to

$$S(n) = \frac{t_s}{t_p}$$

$$E = \frac{S(n)}{n} \times 100\%$$

when E is given as a percentage.

Efficiency gives fraction of time that processors are being used on computation.

Efficiency

The *processor-time* product or *cost* (or *work*) of a computation defined as

Cost = (execution time) × (total number of processors used)

The cost of a sequential computation is simply its execution time, t_s. The cost of a parallel computation is $t_p \times n$. The parallel execution time, t_p, is given by $t_s/S(n)$. Hence, the cost of a parallel computation is given by

$$\text{Cost} = \frac{t_s n}{S(n)} = \frac{t_s}{E}$$

Cost

Cost-Optimal Parallel Algorithm

One in which the cost to solve a problem on a multiprocessor is proportional to the cost (i.e., execution time) on a single processor system.

Shazwan

Lecture 3: Computational Speed

Amdahl's Law

Scalability

A measure used to indicate a hardware design that allows the system to be increased in size and in doing so, to obtain increased performance.

Could be described as architecture or hardware scalability.

Scalability is also used to indicate that a parallel algorithm can accommodate increased data items with low and bounded increase in computational steps.

Could be described as algorithmic scalability.

Problem Size

Intuitively, we would think of the number of data elements being processed in the algorithm as a measure of size. However,

Doubling the problem size would not necessarily double the number of computational steps. It does depend on the type of problem we are solving, for example:

Hence scaling different problems would imply different computational requirements. Alternative definition of problem size is to equate the problem size with the number of basic steps in the best sequential algorithm.

Adding two matrices has this effect of doubling the computational steps, but multiplying two matrices does not.

(What is the number of computational steps needed?)

Amdahl's law assumes that the problem size is fixed.

Gustafson's Law

Gustafson's law assumes that the parallel execution time is fixed. In increasing the problem size, Gustafson also makes the case that the serial section of the code does not increase with the problem size.

Gustafson's Law Scaled Speedup Factor

If we fix the parallel execution time t_p and the fraction of the process which cannot be parallelized f, we can use Gustafson's Law to figure out the speed-up factor given the number of processors, p.

This is called the scaled speedup factor because the t_p is scaled to 1.

The denominator t_p which we scaled to 1 was

+ $ft_s + (1-f)t_s/p = 1$
+ $pft_s+(1-f)t_s = p$
+ $t_s = p + (1-p)\, ft_s$

$$S_s(p) = \frac{ft_s + (1-f)t_s}{ft_s + (1-f)t_s/p} = \frac{p + (1-p)ft_s}{1} = p + (1-p)ft_s$$

$$S_s(p) = \frac{ft_s + (1-f)t_s}{ft_s + (1-f)t_s/p} = \frac{p + (1-p)ft_s}{1} = p + (1-p)ft_s$$

Shazwan

Lecture 3: Analytical Models for Parallel Processing

Programming Options

Parallel programming can be achieved by:

+ Designing a special parallel programming language. (e.g. OCCAM)

+ Extending the syntax/reserved words of an existing sequential high-level language to handle message-passing. (e.g. Intel SPMD Program Compiler mainly for Intel Xeon Phi

+ Using an existing sequential high-level language and providing a library of external procedures for message-passing (e.g. MPI)

Process Creation

Static

All the processes are specified before execution and the system will execute a fixed number of processes. E.g. MPI.

Dynamic

Processes can be created and their execution initiated during the execution of other processes. They can also be destroyed. Obviously, the code for each process must be written beforehand.

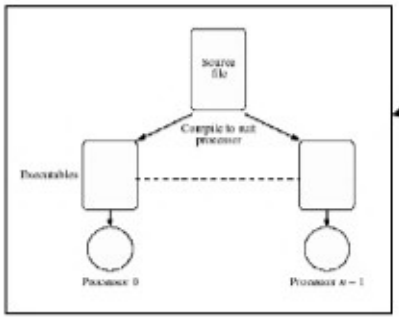
For example, spawn(name_of_process).

PVM has dynamic process creation capabilities, MPI doesn't have it. We will look at pthreads later for shared memory programming which also has this capability.

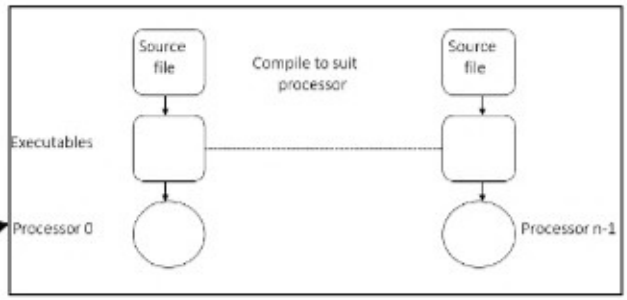
Single Program Multiple Data (SPMD) Model

Different processes are merged into one program.

Within the program are control statements that will customize the code, i.e. select different parts for each process.



It is the main approach for one of the most common message-passing systems, MPI.



Multiple Program Multiple Data (MPMD) Model

Separate and different programs are written for each processor.

Master-slave approach usually taken whereby a single processor executes a master process and other processes are started by the master process.

Recall, basic send() and receive() routines.

Message-Passing Routines

Send and receive message-passing library calls often have the form

send(parameter_list)

recv(parameter_list)

The very basic requirements would be

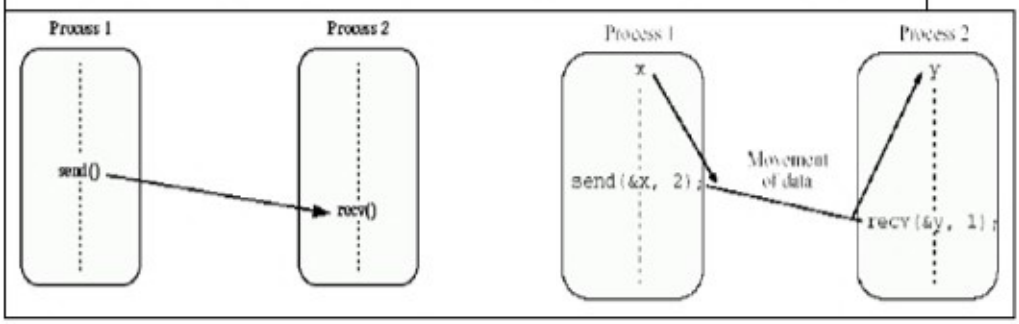
send()is placed in the source process originating the message. e.g. send(&x, destination_id);

recv()is placed in the destination process to collect the messages being sent. e.g. recv (&y, source_id);

Shazwan

Lecture 3: Analytical Models for Parallel Processing

Passing a Message between Processes using send() and recv() Library Calls



Routines that actually return when the message transfer has been completed.

Synchronous Message Passing

Do not need message buffer storage. A synchronous send routine could wait until the complete message can be accepted by the receiving process before sending the message.

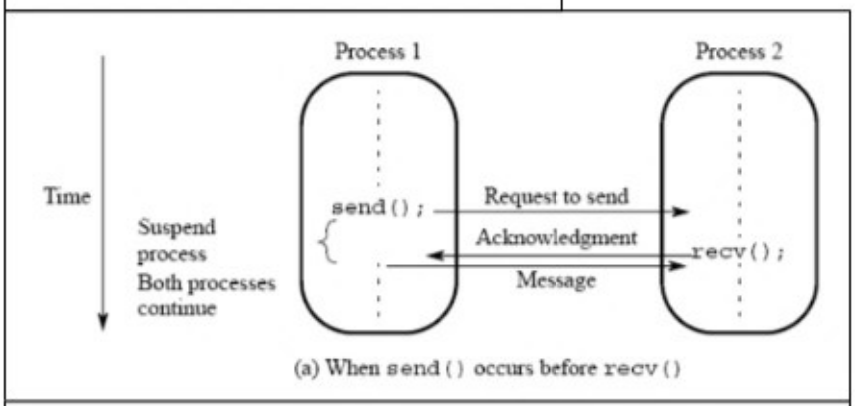
A synchronous receive routine will wait until the message it is expecting arrives.

Synchronous routines intrinsically perform two actions:

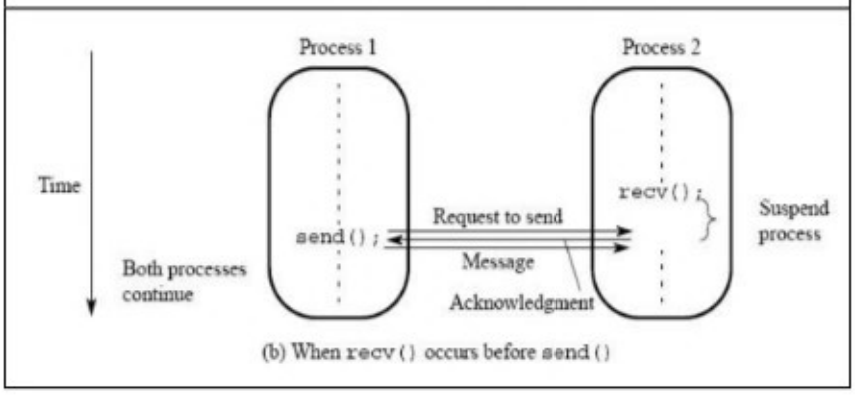
+ They transfer data and they synchronize processes.

+ Suggests some form of signalling, such as a three-way protocol as shown in next slide.

Synchronous send() and recv() Library Calls using a Three-Way Protocol



(a) When `send()` occurs before `recv()`



(b) When `recv()` occurs before `send()`

Blocking & Nonblocking Message-Passing

Blocking

has been used to describe routines that do not return until the transfer is completed.

The routines are "blocked" from continuing.

Non-blocking

has been used to describe routines that return whether or not the message had been received.

The terms blocking and nonblocking have been redefined in systems such as MPI.

In that sense, the terms synchronous and blocking were synonymous.

MPI Definitions of Blocking & Nonblocking

Blocking → return after their local actions complete, though the message transfer may not have been completed.

Non-blocking → return immediately. Assumed that the data storage being used for the transfer is not modified by the subsequent statements prior to the data storage being used for the transfer, and it is left to the programmer to ensure this.

Shazwan

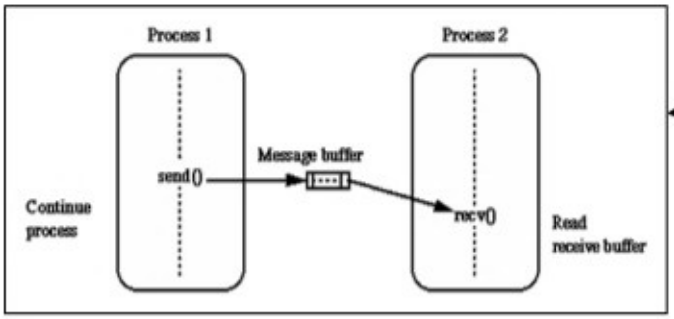
Lecture 3: Analytical Models for Parallel Processing

MPI Routine for Blocking & Non-Blocking Message (Receive)

- Blocking → recv(); /* Blocking receive */
- Non-blocking → nrecv(); /* Non-blocking receive */

Implementation of Non-Blocking Routines

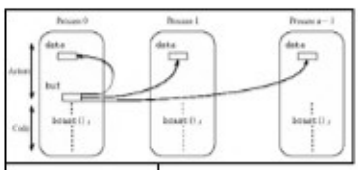
- Generally, a message buffer is needed between source and destination to hold message, as shown in the next slide.
- For the send routine, once the local actions have been completed and the message is safely on its way, the process can continue with subsequent work.
- Buffers can only be of finite length and a point could be reached when the send routine is held up because all the available buffer space has been exhausted.



Message Tag

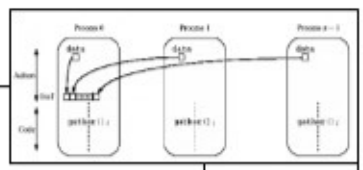
- Used to differentiate between different types of messages sent.
- Example
  - To send a message, x, with message tag 5 from a source process, 1, to a destination process, 2, and assign to y, we might have
    - send (&x, 2, 5);
    - in the source process and
    - recv (&y, 1, 5);
    - in the destination process. The message tag is carried within the message.
- If special type matching is not required, a wild card message tag is used, so that the recv() will match with any send().

More Basic Message Passing Routines	
+ Broadcast	+ Gather
+ Scatter	+ Reduce



Broadcast

- Sending the same message to all the processes concerned with the problem.
- **Multicast** - sending the same message to a defined group of processes.

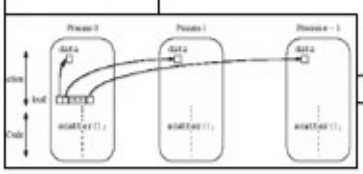


Having one process collect individual values from a set of processes.

Gather

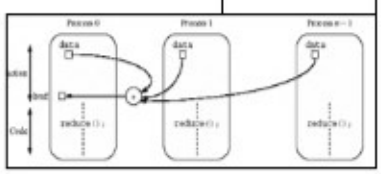
- Gather operation combined with a specified arithmetic or logical operation.
- Example: the values could be gathered and then added together by the root.

Scatter



- Sending each element of an array of data in the root to a separate process.
- The contents of the i^{th} location of the array is sent to the i^{th} process.

Reduce



Shazwan

Lecture 4: PVM

Cluster Computing

PVM (Parallel Virtual Machine)

The first widely adopted attempt at using workstation clusters as a multicomputer platform was done by Oak Ridge National Laboratories in the late 1980s and used widely in the 1990s.

Available at no charge.

Provides a software environment for message-passing between homogeneous or heterogeneous computers.

MPI (Message-Passing Interface)

Standard developed by group of academics and industrial partners to foster more widespread use and portability. Several free implementations exist.

Parallel Virtual Machines (PVM)

The problem is decomposed into separate programs by the programmer. Each program is written in C or Fortran) and compiled to run on specific types of computers in the network.

If the network contains a homogeneous collection of computers the programs only have to be compiled once.

In a network of workstations, each workstation has access to the file system containing the compiled programs. It is only necessary to ensure that the executables of programs exist for the specific workstations that will execute the programs.

It will automatically allocate computers if the user does not specify the specific computers that are to be used for executing the programs.

The set computers used on a problem first must be defined prior to running the programs, and form the "virtual parallel machine".

The most convenient way of doing this is by forming a list of the names of the computers that form the virtual machine in a file (a hostfile). The hostfile is then read by pvm.

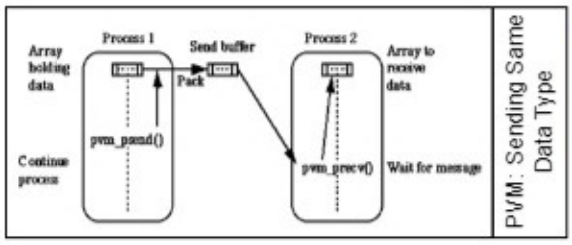
An alternative way is to start with one machine and add others manually by using pvm control commands.

The routing of messages between computers is done by PVM daemon processes installed by PVM on the computers that form the virtual machine.

PVM: Sending Same Data Type

All pvm send routines are non-blocking (or asynchronous in pvm terminology) while pvm receive routines can be either blocking (synchronous) or non-blocking.

If the data being sent is a list of items of a same data type, the pvm call pvm_psend()and pvm_precv() can be used where a parameter in pvm_psend() points to an array of data in the source process to be sent.



PVM: Sending Different Data Type

All pvm send routines are non-blocking (or asynchronous in pvm terminology) while pvm receive routines can be either blocking (synchronous) or non-blocking.

The receiving process must unpack its receive message buffer according to the format in which it was packed.

hellom.c

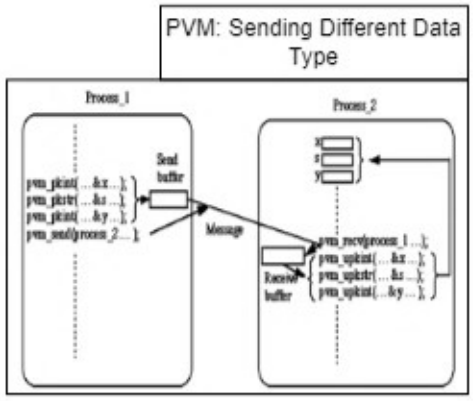
```
#include "pvm3.h"
main(){
int cc, tid, msgtag;  char buf[100];
printf("i'm t%x\n", pvm_mytid());
cc = pvm_spawn("hellow", (char**)0, 0, "", 1, &tid);
if (cc == 1) {
        msgtag = 1;
        pvm_recv(tid, msgtag);
        pvm_upkstr(buf);
        printf("from t%x: %s\n", tid, buf);
} else
        printf("can't start hellow\n");
pvm_exit();
}
```

hellow.c

```
#include "pvm3.h"
main()
int ptid, msgtag;  char buf[100];
ptid = pvm_parent();  strcpy(buf, "hello, world");
msgtag = 1;
//Initialize the send - Clear the buffer 4
//prepare for packing new message using the default
encoding
pvm_initsend(PvmDataDefault);
pvm_pkstr(buf);                 // Pack the string
pvm_send(ptid, msgtag);         //Send the packed string
to parent process
pvm_exit();
}
```

http://www.netlib.org/pvm3/book/node17.html

PVM Example Program



Shazwan

Lecture 4: MPI

History of MPI

+ It was developed in 1993-94 by a group, called the MPI Forum, composed of major IT corporations and research centres to standardize the way that parallel processing is done.

+ Still actively being enhanced, version 4.0 is expected to be released soon.

MPI is an "industrial strength" library with literally hundreds of different routines.

Goals of MPI

+ Provide source code portability,

+ Allow an efficient range of implementations across many architectures.

Features

+ Provide a great deal of functionality, and

+ Support heterogeneous parallelism, that is, the ability to have different types of processors communicate with each other.

Routine

```
MPI_Init(&argc, &argv);
MPI_Finalize();
MPI_Comm_size(MPI_COMM_WORLD, &p);
MPI_Comm_rank(MPI_COMM_WORLD, &myRank);
MPI_Send(&message, count, type, dest, tag, MPI_COMM_WORLD);
MPI_Recv(&message, count, type, source, tag, MPI_COMM_WORLD, &status);

MPI_Isend()
MPI_IRecv()
```

Collective communication is any type of communication other than point to point, that is, any type of communications that does not use SEND and RECEIVE. There are three fundamental kinds:

Collective Communications

Lecture 4: Further Refence

Three fundamental kinds

+ one → many

+ many → one

+ many → many

These routines allow you to implement all of the collective communications you will need for virtually any parallel algorithm.

MPI_Bcast()

- `MPI_Bcast(&message, count, datatype, root, communicator);`
  - One process, called the root, sends the same message to every other process in the specified communicator. So, a broadcast is a 1→many transmission.
  - Every process must execute the same command, but the root is the sender, and everyone else is the receiver.
  - The indicated message, containing count values each of type datatype is then sent from the root to every other process in the communicator.
  - If one of the processes in the communicator does not execute the broadcast, it will result in a deadlock condition.
  - For example, to broadcast the floating point value x from process 0 to every other process, you would execute the following command:

    ```
    MPI_Bcast(&x, 1, MPI_FLOAT,0,MPI_COMM_WORLD);
    ```

  - When every process has executed this command the broadcast will take place and every process will have the value of the variable X stored in its local memory.

MPI_Reduce()

Operation Name	Meaning
MPI_MAX	Maximum
MPI_MIN	Minimum
MPI_SUM	Sum
MPI_PROD	Product
MPI_LAND	Logical And
MPI_BAND	Bitwise And
MPI_LOR	Logical Or
MPI_BOR	Bitwise Or
MPI_LXOR	Logical Exclusive Or
MPI_BXOR	Bitwise Exclusive Or
MPI_MAXLOC	Maximum and Location of Maximum
MPI_MINLOC	Minimum and Location of Minimum

MPI_Reduce()

- `MPI_Reduce(&data_value, &global_value, count, datatype, reduction operator, root, communicator);`
  - Reduction is the reduction of many values to a single value using a single binary operand. So, reduction is a many→one operation.
  - Now the data value, containing count values of type datatype, is sent by every node in the communicator to the indicated root processor and combined there into a single global value using the specified reduction operator.
  - For example, if every processor had a floating point value called x, and we wanted to sum them and store them in the variable called xSum on processor number 4, we would do the following:

    ```
    MPI_Reduce(&x, &xSum, 1, MPI_FLOAT, MPI_SUM, 4, MPI_COMM_WORLD);
    ```

MPI_Allreduce()

- There is a variant of Reduce that will iterate on the reduction operation, using each node as the root.
- That is, we will reduce X to Xsum at node 0.Then we will reduce X to Xsum at node 1. Then we will reduce X to Xsum at node 2. This variant is called All_reduce.
- The calling sequence is the same except that we eliminate the root field, since every node will function as the root.
- All reduce is identical to doing P simultaneous reductions.
- It is an example of a many → many collective communication.

MPI_Gather()

- `MPI_Gather(&send_data, send_count, send_type, &receive_data, receive_count, receive_type, root, communicator)`
  - This is an operation that takes an N element data structure evenly distributed among P processors, and gathers them together on a single processor, called the root.
  - This operation assumes that every node contains send count (= N/P) data elements, each of type send type.
  - These elements then are sent to the processor called root where they are all stored in a data structure, called receive data, of the same type.
  - Receive_count is identical to send count and represents the number of elements received from each individual node.

MPI_Gather()

- So, lets say that a 100 element integer array x[0]…, x[99] was stored on 20 processors, with 5 elements of x located on each one. If we needed to gather all elements of x together on processor 0, we could do the following:

  ```
  MPI_Gather(&x, 5, MPI_INT, &allX, 5, MPI_INT, 0, MPI_COMM_WORLD);
  ```

- Following the execution of this operation allX which must be declared to be an array of size 100 or greater, would contain 100 elements, with the first 5 slots containing the 5 x values from processor 0, the second five slots containing the 5 x values from processor 1, the third group of five slots containing the 5 x values from processor 2, etc.
- Values that are spread out over a number of processors are gathered together in one place.

Shazwan

Lecture 4: Further Refence

MPI_Allgather()

- There is also a version of the Gather operation in which the elements are repeatedly gathered at every node.
- That is, we do a gather with root = 0, then another gather with root = 1, etc.
- This is called the AllGather operation.
- Its parameter sequence is the same as in Gather, except that the root field is not present.

MPI_Scatter()

- `MPI_Scatter(&send_data, send_count, send_type, &receive_data, receive_count, receive_type, root, communicator)`
  - This function is the exact opposite of Gather.
  - Now we take a data structure called send data containing N data elements of type send type and we distribute it evenly to P processors, giving each processor send count (= N/P) data elements.
  - The node that does the distribution is called the root.
  - So, lets assume that a floating point array Y located at processor 2 contained 64 data elements, and we wanted to distribute it evenly to four processors, each one getting 16 elements and each processor storing it is a local variable called myY.

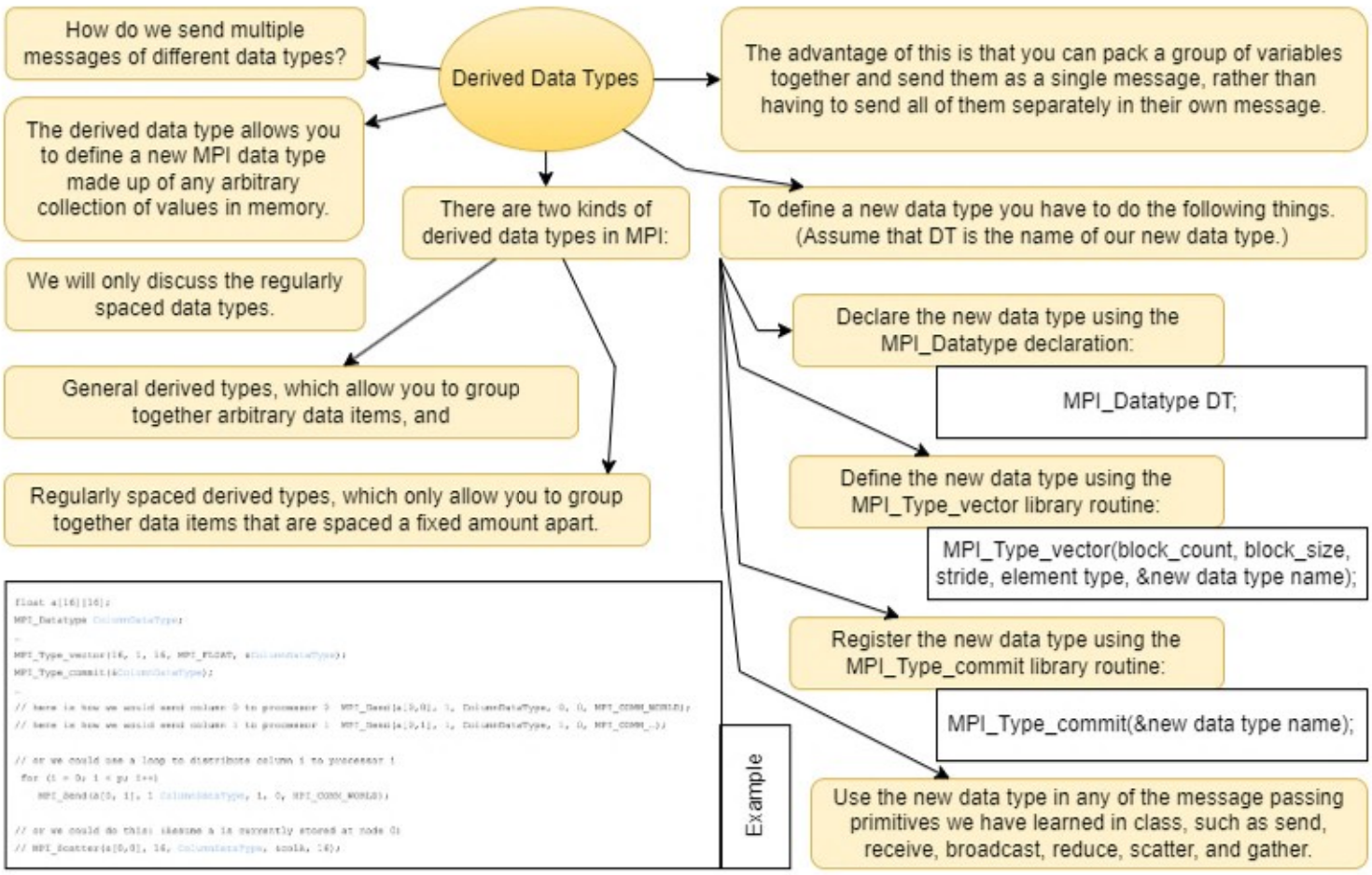
MPI_Scatter()

- Here is how we could do it:

```
MPI_Scatter(&y, 16, MPI_FLOAT, &myY, 16, MPI_FLOAT, 2, MPI_COMM_WORLD);
```

- When we have finished executing this operation, the variable myY on processor 0 would contain the values of y[0] to y[15], the variable myY on processor 1 would contain y[16] to y[31], the variable myY on processor 2 would contain y[32] to y[47], etc.
- Values on a single processor are distributed to all of the other processors in an equal fashion.

Lecture 4: Derived Data Types




```
float a[16][16];
MPI_Datatype ColumnDataType;
...
MPI_Type_vector(16, 1, 16, MPI_FLOAT, &ColumnDataType);
MPI_Type_commit(&ColumnDataType);
...
// here is how we would send column 0 to processor 0: MPI_Send(a[0,0], 1, ColumnDataType, 0, 0, MPI_COMM_WORLD);
// here is how we would send column 1 to processor 1: MPI_Send(a[0,1], 1, ColumnDataType, 1, 0, MPI_COMM_..);

// or we could use a loop to distribute column 1 to processor 1
 for (i = 0; i < p; i++)
    MPI_Send(a[0, 1], 1 ColumnDataType, 1, 0, MPI_COMM_WORLD);

// or we could do this: (assume a is currently stored at node 0)
// MPI_Scatter(a[0,0], 16, ColumnDataType, local, 16);
```

Shazwan

Lecture 4: Timing

Timing in MPI

Since the goal of parallel processing is speedup, timing how long your program takes to produce the correct answer is a very important part of any MPI program.

Three commands that are useful in helping you to get information on the running time of your program are:

MPI_Wtime()

This function returns a double value that is the number of time units since some previous standardized point in time.

You can use this function to measure elapsed time in the following way:

Example

```
double a, b, elapsed;
…
a = MPI_Wtime( ); // get the time before you start your measurements
// here is the code you want to time
b = MPI_Wtime( );
elapsed = (b-a);
```

Now different systems will have different clock tick rates, so you also have to be able to find out what that unit of time is. (i.e., does elapsed = 100 mean 100 nano seconds, 100 micro seconds, 100 seconds?)

To find out the unit of time use the following function:

MPI_Wtick()

This function returns a double value which is the resolution of the clock. That is, the time unit being used by MPI_Wtime.

To ensure accurate timings, you should use the barrier synchronization command.

MPI_Barrier(communicator)

This routine must be executed by every process in the specified communicator.

When a process executes this command, it will be suspended until everyone has executed it.

When everyone executes it, then every process will be released and allowed to continue.

It is sort of line a "starting line" that everyone reaches and waits for everyone to reach.

Then they are allowed to continue.

Example

```
double unit, a, b, elapsed;
…                    // there may be some  initialization operations
unit = MPI_Wtick( );
MPI_Barrier(MPI_COMM_WORLD);
// when everyone has finished initialization, then we will continue
a = MPI_Wtime( )
…                    // here is the code you want to time
b = MPI_Wtime( );  elapsed = (b-a) * unit;
// this will give you the time in seconds
```

Shazwan

Lecture 5: Parallel Execution Time

Two parts: a computation part, say t_{comp}, and a communication part, say t_{comm};

$$t_p = t_{comp} + t_{comm}$$

Computation time estimated in a similar way to that of a sequential algorithm

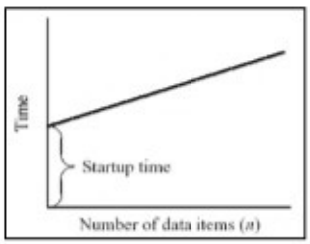
Parallel Execution Time

As a first approximation, we will use

$$t_{comm} = t_{startup} + n\, t_{data}$$

Only intended to give a starting point to how an algorithm might perform in practice.

where $t_{startup}$ is the startup time, sometimes called the message latency – essentially time to send a message with no data. Startup time is assumed constant.



The term t_{data} is the transmission time to send one data word, also assumed a constant, and there are n data words.

Important Notes on Interpretation of Equations

Parallel execution time, t_p normalized in units of an arithmetic operation, which will depend upon computer system.

We might find that the computation requires m computational steps so that $t_{comp} = m$

Since we are measuring time in units of computational steps, the communication time has to be measured in the same way.

All data types are assumed to require the same time.

Suppose q messages are sent, each containing n data items. We have $t_{comm} = q(t_{startup} + nt_{data})$

The startup and data transmission times would make the execution time of applications on most systems worse, especially networked workstations.

Latency Hiding

Nonblocking send routines are provided particularly to enable latency hiding.

A way to help deal with the problem of significant message communication times is to overlap communication with subsequent computations, i.e. by keeping the processor busy with useful work while waiting for the communication to be completed. This is known as latency hiding.

Latency hiding can also be achieved by mapping multiple processes on a processor and use a time-sharing facility that switches for one process to another when the first process is stalled because of incomplete message passing or otherwise.

Lecture 5: Time Complexity

Time Complexity

As with sequential computations, a parallel algorithm can be evaluated through the use of time complexity (notably the O notation — "order of magnitude," big-oh).

An expression of the number of computational steps is derived, often in terms of the number of data items being handled by the algorithm.

Start with an estimate of the number of the **computational steps**, considering all arithmetic and logical operations to be equal and ignoring other aspects of the computation such as computational tests.

Shazwan

Lecture 5: Time Complexity

The big-O notation

- Formal Definition
  - $f(x) = O(g(x))$ if and only if there exists positive constants, c and x_0, such that $0 \le f(x) \le cg(x)$ for all $x \ge x_0$
    - where f(x) and g(x) are functions of x.
    - For example, if $f(x) = 4x^2 + 2x + 12$, the constant **c = 6** would work with the formal definition to establish that $f(x) = O(x^2)$, since $0 < 4x^2 + 2x + 12 \le 6x^2$ for $x \ge 3$.

- Unfortunately, the formal definition also leads to alternative functions for g(x) that will satisfy the definition. For example,
  - g(x)=x3 also satisfies the definition 4x2+2x+12 ≤ 2x3 for x ≥ 3.
- Normally, we would use the function that grows the least for g(x).

Theta notation - upper bound

- Many cases we have a "tight bound" in that function *f(x)* equals *g(x)* to within a constant factor. This can be captured in a Θ notation.
- $f(x) = \Theta(g(x))$
  - if and only if there exists positive constants c_1, c_2, and x_0 such that $0 \le c_1 g(x) \le f(x) \le c_2 g(x)$ for all $x \ge x_0$.

- If $f(x) = \Theta(g(x))$, it is clear that $f(x) = O(g(x))$ is also true.
- We can say that $f(x) = 4x^2 + 2x + 12 = \Theta(x^2)$, which is more precise than using $O(x^2)$.
- We should really only use the big O notation if and only if the upper bound on growth can be satisfied. However it is common practice to use big O in any event.

Omega notation - upper bound

- The lower bound on growth can be described by the Ω notation which is defied as
  - $f(x) = \Omega(g(x))$ if and only if there exists positive constants c and x_0 such that $0 \le cg(x) \le f(x)$ for all $x \ge x_0$.
  - It follows from this definition that $f(x) = 4x^2 + 2x + 12 = \Omega(x^2)$.
  - We can read **O( ) as "grows at most as fast as"** and **Ω( ) as "grows at least as fast as."**
  - The function $f(x) = \Theta(g(x))$ is true if and only if $f(x) = \Omega(g(x))$ and $f(x) = O(g(x))$.
  - **The Ω notation can be used to indicate the best case situation.**

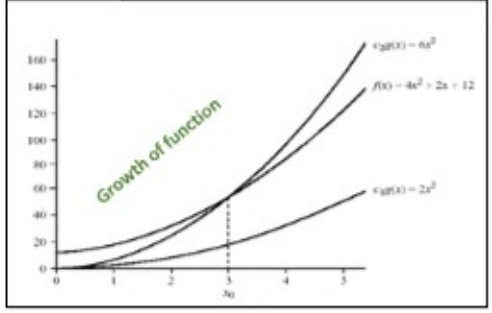
Example

The execution time of a sorting algorithm often depends upon the original order of the numbers to be sorted.

It may be that it requires at least **n log n** steps, but could require n^2 steps for n numbers depending upon the order of the numbers.

This would be indicated by a time complexity of Theta(n log n) and Omega(n2).

$f(x) = 4x^2 + 2x + 12$



Shazwan

Lecture 5: Time Complexity

Time Complexity of Parallel Algorithms

Typically, parallel time complexity analysis hides lower terms, i.e. t_{comm} has time complexity of O(n).

Time complexity of t_p will be sum of complexity of computation and communication.

Example

To add **n** numbers on two computers, where each computer adds **n/2** numbers together, and the numbers are initially all held by first computer.

Second computer submits its result to first computer for adding the two partial sums together. This problem has several phases:

1. Computer 1 sends **n/2** numbers to computer 2.
2. Both computers add **n/2** numbers simultaneously.
3. Computer 2 sends its partial result back to computer 1.
4. Computer 1 adds the partial sums to produce the final result.

Computation (for steps 2 and 4):

$t_{comp} = n/2 + 1$

Communication (for steps 1 and 3):

$t_{comm} = (t_{startup} + n/2\ t_{data}) + (t_{startup} + t_{data})$

$= 2t_{startup} + (n/2 + 1)t_{data}$

- The computational complexity is O(n).
- The communication complexity is O(n).
- The overall time complexity is O(n).

Lecture 5: Cost-Optimal Algorithms

A cost-optimal (or work-efficient or processor-time optimality) algorithm is one in which the cost to solve a problem is proportional to the execution time on a single processor system (using the fastest known sequential algorithm). i.e.

$Cost = t_p * n = k * t_s$

where k is a constant.

Given time complexity analysis, we can say that a parallel algorithm is a cost-optimal algorithm if

(Parallel time complexity) x (number of processors) = sequential time complexity

Example

Suppose the best known sequential algorithm for a problem has time complexity of O(n log n).

A parallel algorithm for the same problem that uses n processes and has a time complexity of O(log n) is cost optimal, whereas a parallel algorithm that uses n^2 processors and has time complexity of O(1) is not cost optimal.

Lecture 5: Communication routines & time complexity

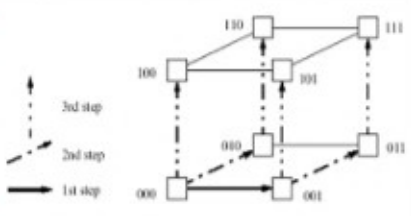
Time Complexity of Broadcast/Gather

Broadcast on a Hypercube Network

Consider a three-dimensional hypercube. To broadcast from node 000 to every other node, 001, 010, 011, 100, 101, 110 and 111, an efficient algorithm is

Time Complexity of Broadcast/Gather

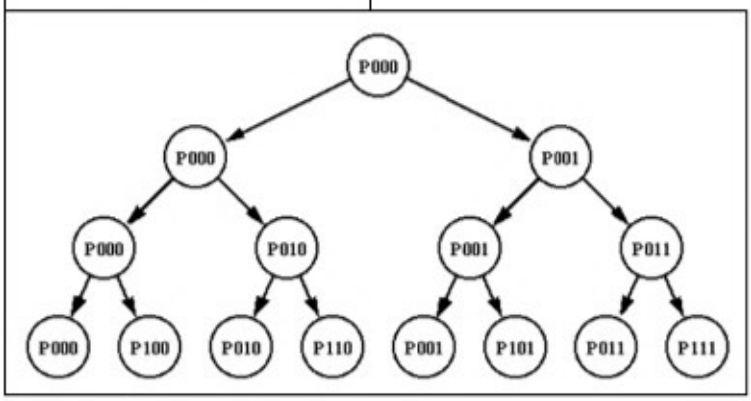
	Node	Node
1st step	000	001
2nd step	000	010
	001	011
3rd step	000	100
	001	101
	010	110
	011	111



Shazwan

Lecture 5: Communication routines & time complexity

Broadcast on a Tree Network

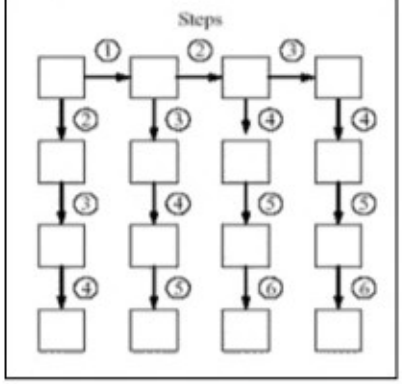


Broadcast on a Mesh Network

Broadcast can be achieved on a mesh network (without wraparound) - for instance, from the upper left corner-by sending the message across the top row and down each column as the message reaches the top node of that column as shown in the figure on the next slide.

This algorithm requires 2(n-1) steps or O(n) on the n*n mesh, again an optimal algorithm in terms of the number of steps because the diameter of a mesh without wraparound is given by 2(n-1).

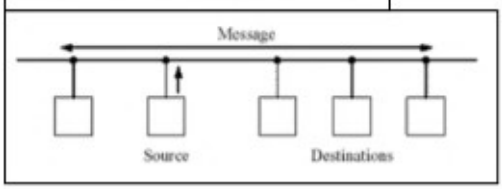
Broadcast on a Mesh Network



Broadcast on a single Ethernet connection can be done using a single message that is read by all the destinations on the network simultaneously. Hence broadcast could be very efficient with an O(1) time complexity for one data item; for n data item it is O(n).

Most networked computers use a variety of network structures, and the convenient broadcast medium available in a single Ethernet will not generally be applicable.

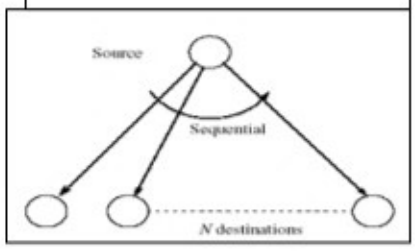
Broadcast on a Workstation Cluster



Broadcast on a Workstation Cluster

The PVM broadcast routine, pvm_bcast( ), broadcast messages through its daemons to the destinations. Once a message arrives at a daemon, it is converted in to 1-to-N fan-out broadcast call, where the same message is sent to each of the destinations. In turn, the communication time complexity for one source connecting to n destinations is O(n).

Broadcast on a Workstation Cluster



The 1-to-N fan-out broadcast applied to a tree structure has time complexity of O(M+N/M) for one data item broadcast to N destinations, where there are M daemons.

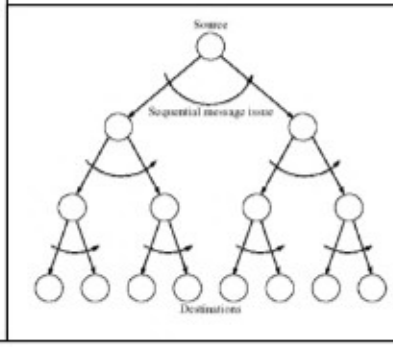
1-to-N fan-out Broadcast on a Tree Structure

Each of these daemons must receive the data in turn, and N/M is the number of destinations under each daemon.

One disadvantage of binary tree implementation of broadcastis that if a node fails in the tree, all nodes below it will not receive the message.

In addition a binary tree would be difficult to implement in a library call.

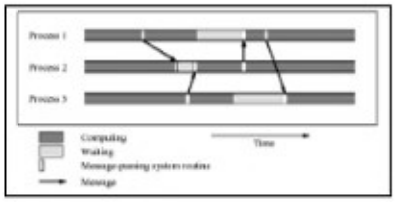
1-to-N fan-out Broadcast on a Tree Structure



Shazwan

Lecture 5: Parallel program debugging and evaluation

Programs can be watched as they are executed in a space-time diagram (or process- time diagram).



Visualization Tools

- A space-time diagram is useful to spot erroneous actions.
- The events that created the space-time diagram can be captured and saved so that the presentation can be replayed without having to re-execute the program.
- Also shows the amount of time spent by each process on communication, waiting, and on message passing library routines.
- An example is the Intel VTune program visualization system.

Debugging Strategies

Geist et al. (1994a) suggest a three-step approach to debugging message-passing programs:

- If possible, run the program as a single process and **debug as a normal sequential program.**
- **Execute the program using two to four multitasked processes on a single computer.** Now examine actions such as checking that messages are indeed being sent to the correct places. It is very common to make mistakes with message tags and have messages sent to the wrong places.
- Execute the program using the same two to four processes but now across several computers. This step helps find problems that are caused by network delays related to synchronization and timing.

Timing

- MPI provides the routine MPI_Wtime ( ) for returning time (in seconds).
- Measuring Execution Time
  - To measure the execution time between point L1 and point L2 in the code, we might have a construction such as

```
L1: time(&t1); /* start timer */
…..
L2: time(&t2); /* stop timer */
elapsed_time = difftime(t2, t1); /* elapsed_time = t2 - t1 */
printf("Elapsed time = %5.2f seconds", elapsed_time);
```

- Elapsed time include waiting time for messages and it is assumed that the processor is not executing other programs at the same time.

Communication Time by the Ping-Pong Method

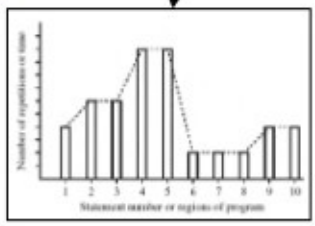
- One process, say P_0, is made to send a message to another process, say P_1
- Immediately upon receiving the message, P_1 sends the message back to P_0

```
P0
 L1: time(&t1);
 send(&x, P1);
 recv(&x, P1);
 L2: time(&t2);
 elapsed_time = 0.5 * difftime(t2, t1);
 printf("Elapsed time = %5.2f seconds", elapsed_time);
```

```
P1
 recv(&x, P0);
 send(&x, P0);
```

Profiling

- A profile of a program is a histogram or graph showing the time spent on different parts of the program.
- A profile can show the number of times particular source statements are executed.



Shazwan

Lecture 6: Partitioning

Partitioning → Partitioning Strategies

Partitioning simply divides the problem into parts.

It is the basis of all parallel programming, in one form or other.

Partitioning can be applied to the program data (that is, to dividing the data and operating upon the divided data concurrently) – known as data partitioning or domain decomposition.

Domain and Functional Decomposition

Partitioning can also be applied to the functions of the program (that is dividing the program into independent functions and executing the functions concurrently) - known as **functional decomposition.**

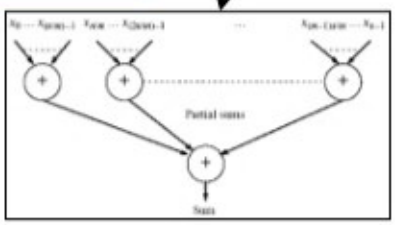
The idea of performing a task by dividing it into a number of smaller tasks that when completed will complete the overall task is well known and can be applied to many situations.

Partitioning / Divide & Conquer Examples → Many possibilities:

- **Operations on sequences of number** such as simply adding them together.
- Several **sorting** algorithms can often be partitioned or constructed in a recursive fashion.
- **N-body** problem.
- Numerical integration.

Partitioning / Divide & Conquer Examples → Adding a sequence of numbers

Dividing sequence into m parts, added independently to create partial sums.



Separate the data at the master node, send them directly to the slaves and receive the answer and sum them all up.

Summation Algorithm

Broadcast all numbers to the slaves, slaves decides which numbers to add and send it back to the master, who will sum them all up.

Using the scatter routine (which automatically splits and sends them) and the final summation is done as part of the reduction routine.

Example – Adding a Sequence of Numbers

Using separate `send()` and `recv()`

Master

```
s = n/p                          // no. of numbers 4 slaves
for (i = 1, x = 1; i < p; i++, x+=s)
  send(&numbers[x], s, P[i]);    // send s numbers to slave
sum = 0;
```

Slave

```
recv(numbers, s, P_master);      // receive s numbers
part_sum = 0;
for (i = 0; i < s; i++)          // add numbers
  part_sum = part_sum + numbers[i];
send(&part_sum, P_master);       // send sum to master
```

```
for (i = 1; i < p; i++) {        // wait for results
  recv(&part_sum, P[i]);
  sum = sum + part_sum;          // accumulate the sum
}
```

Using broadcast / multicast routine

Master

```
s = n/p                          // no. of numbers 4 slaves
bcast(numbers, n, P_broadcast);  // send all numbers to slaves
sum = 0;
```

Slave

```
bcast(numbers, n, P_master);
// compute the parts the slave is to sum
start = slave_number * s;        // slave_number obtained earlier
end = start + s;
part_sum = 0;
for (i = start; i < end; i++) // add numbers
  part_sum = part_sum + numbers[i];
send(&part_sum, P_master);       // send sum to master
```

```
for (i = 1; i < p; i++) {        // wait for results
  recv(&part_sum, P[i]);
  sum = sum + part_sum;          // accumulate the sum
}
```

Using scatter and reduce routines

Master

```
s = n/p          //number of numbers

// send numbers to slaves
scatter(numbers,&s,P_master,root=master);
```

```
reduce_add(&sum,&s,P_master,root=master); // results
```

Slave

```
scatter(&numbers, &s, P_master,root=master); // receive a numbers
part_sum = 0;
for (i = 0; i < s; i++)                  // add numbers
  part_sum = part_sum + numbers[i];

reduce_add(&part_sum, &s, P_master,root=master); // send part sum
```

Shazwan

Lecture 6: Partitioning

Analysis: How Good is the Parallel Algorithm

- Sequential
  - Requires (n – 1) additions with a time complexity of O(n).
- Parallel: Using individual send and receive routines
  - Phase 1: Communication
    - $t_{comm1} = p(t_{startup} + (n/p)t_{data})$
  - Phase 2: Computation
    - $t_{comp1} = n/p - 1$
  - Phase 3: Communication: Returning partial results using send/recv routines
    - $t_{comm2} = p(t_{startup} + t_{data})$
  - Phase 4: Computation: Final accumulation
    - $t_{comp2} = p - 1$

Number of data sent to individual slaves

The sequential summation parts

One data item sent back

Analysis

Overall

$$t_p = (t_{comm1} + t_{comm2}) + (t_{comp1} + t_{comp2})$$
$$= p(t_{startup} + (n/p)t_{data}) + p(t_{startup} + t_{data}) + n/p - 1 + p - 1$$
$$= 2pt_{startup} + (n + p)t_{data} + p + n/p - 2$$
$$= O(n+p)$$

If the number of processors, p, is fixed, then the parallel time complexity is the same as sequential time complexity.

Analysis

Parallel time complexity is worse than sequential time complexity.

O(n+p) > O(n)

Lecture 6: Divide and Conquer

Divide and Conquer

Characterized by dividing a problem into **sub-problems that are of the same form as the larger problem.**

Further divisions into smaller sub-problems are usually done by **recursion**. The recursive method will continually divide the problem until the tasks cannot be broken down into smaller parts.

Then the very simple tasks are performed and results are combined, with the combining continued with larger and larger tasks.

Example

- Adding a sequence of numbers
  - A sequential recursive definition for adding a list of numbers

```
/* This is a pseudo code, actual implementation has been omitted */

int add(int *s)       /* add list of numbers, s */
{
  if (number(s) =< 2)
    return (n1 + n2);
  else {
    /* divide s into two parts, s1 and s2 */  Divide (s, s1, s2);
    /* recursive calls to add sub lists */
    part_sum1 = add(s1);
    part_sum2 = add(s2);
    return (part_sum1 + part_sum2);
  }
}
```

Explanation for Code

As in all recursive definitions, a method must be present to terminate the recursion when the division can go no further.

number(s) returns the number of numbers in the list pointed by s. If there are two numbers in the list, they are called n_1 and n_2. If there is one number in the list, it is called n_1 and n_2 is zero. If there are no numbers, both n_1 and n_2 are zero.

Separate if statements could be used for each of the cases: 0, 1, or 2 numbers in the list. Each could cause termination of the recursive call.

This method can also be used for other global operations on a list. E.g.:

- Finding the maximum number in a list
- Sorting a list by dividing the list in to smaller and smaller lists to sort. Mergesort and quicksort-sorting algorithms are usually described by such recursive definitions.

Shazwan

Lecture 6: Divide and Conquer

Divide & Conquer / Binary Tree

When each division creates two parts, a recursive divide-and-conquer formulation forms a binary tree.

The tree is traversed downward as calls are made and upward when the calls return.

The root process first divides the problem into two parts.

These two parts are each divided into two parts, and so on until the leaves are reached. There the basic operation of the problem are performed.

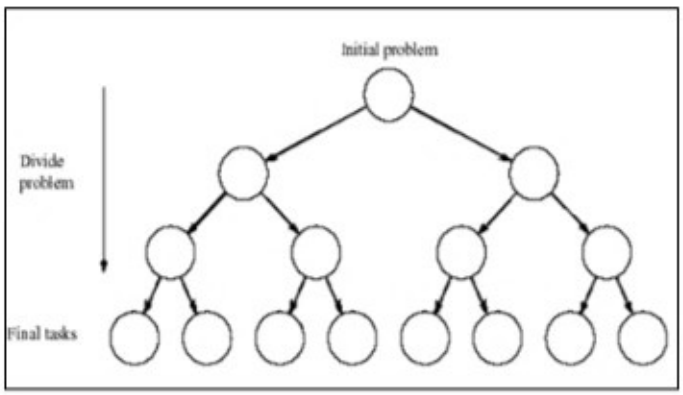
When this construction is applied to the above problem, first the list of numbers will be divided into two parts, then into four parts, and so on until each process has one equal part of the whole. After adding pairs at the bottom of the tree, the accumulation occurs in a reverse tree construction.

Binary Tree Structure: Recursion

The image shows a complete binary tree, i.e., a perfectly balanced tree with all bottom nodes at the same level.

This occurs if the tasks can be divided in to a number of parts that is a power of two. If not a power of two, one or more of bottom nodes will be at one level lower than the others.

Here we assume a balanced binary tree, for convenience.



In a sequential implementation, only one node of the tree can be visited at a time.

A parallel solution offers the prospect of traversing several parts of the tree simultaneously.

Implementation

Each of the list at the final stage will have n/8 numbers , or n/p in the general case of p processors. There are log p levels in the tree.

Once a division is made into two parts, both parts can be processed simultaneously.

The division stops when the total number of processors is committed. Until then, at each stage each processor keeps half of the list and passes on the other half.

But requiring $2^{m+1} - 1$ and having most of the processors idle would be inefficient.

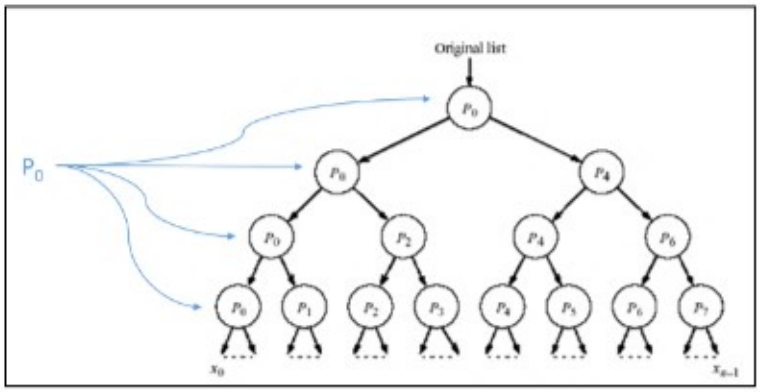
$2^{m+1} - 1$ processors are required to divide the tasks into 2^m parts and each processor would only be active at one level in the tree, leading to a very inefficient solution.

A more efficient solution is to reuse processors at each level of the tree as shown in the previous slide, which uses 8 processors.

First p_0 communicates with p_4, passing half of the list to p_4

Then p_0 and p_4 pass half of the list to they hold to p_2 and p_6 respectively.

Finally p_0, p_2, p_4 and p_6 pass half of the list they hold to p_1, p_3, p_5 and p_7 respectively.



Shazwan

Lecture 6: Divide and Conquer

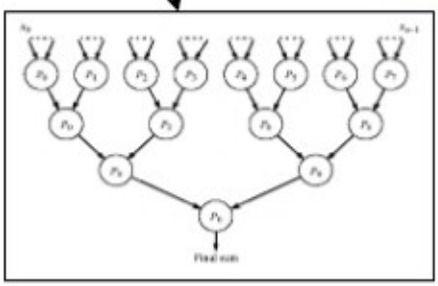
Traversing Up The Tree: Summation

When each division creates two parts, a recursive divide-and-conquer formulation forms a binary tree.

The even numbered processors then add the partial sum with its own partial sum and pass the result onward. This continues until p_0 has the final result.

This construction is same as the binary hypercube broadcast and gather algorithms. The construction would map on to a hypercube perfectly, but are also applicable to other systems. (Dilation of 1)

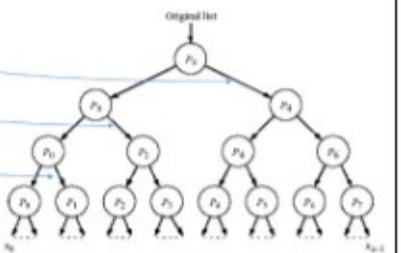
Processors communicate with processors whose addresses differ in one bit, starting with the most significant bit for the division phase and with the least.



Parallel Code

- Suppose we statically create eight processors (or processes) to add a list of numbers.

```
divide(s1, s1, s2);
send(s2, P4);
divide(s1, s1, s2);
send(s2, P2);
divide(s1, s1, s2);
send(s2, P1);
part_sum = *s1;
recv(&part_sum1, P1);
part_sum = part_sum + part_sum1;
recv(&part_sum1, P2);
part_sum = part_sum + part_sum1;
recv(&part_sum1, P4);
```



Analysis

Assume n is a power of 2. Communication time $t_{start-up}$ is not included in this analysis.

Communication

Division Phase

$$t_{comm1} = \frac{n}{2}t_{data} + \frac{n}{4}t_{data} + \frac{n}{8}t_{data} + \cdots + \frac{n}{p}t_{data} = \frac{n(p-1)}{p}t_{data}$$

Combining Phase

$$t_{comm2} = t_{data}\log p$$

Total Communication Time

$$t_{comm} = t_{comm1} + t_{comm2} = \frac{n(p-1)}{p}t_{data} + t_{data}\log p$$

Computation

$$t_{comp} = \frac{n}{p} + \log p$$

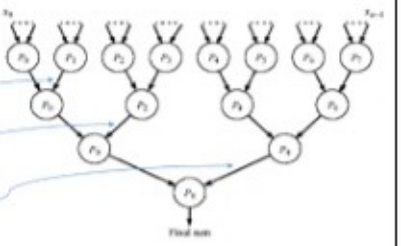
Total Parallel Execution Time (Complexity)

$$t_p = \frac{n(p-1)}{p}t_{data} + t_{data}\log p + \frac{n}{p} + \log p$$

Parallel Code

- Suppose we statically create eight processors (or processes) to add a list of numbers.

```
divide(s1, s1, s2);
send(s2, P4);
divide(s1, s1, s2);
send(s2, P2);
divide(s1, s1, s2);
send(s2, P1);
part_sum = *s1;
recv(&part_sum1, P1);
part_sum = part_sum + part_sum1;
recv(&part_sum1, P2);
part_sum = part_sum + part_sum1;
recv(&part_sum1, P4);
```



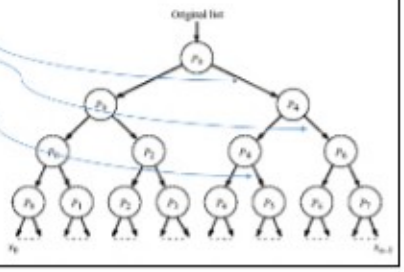
m-ary Tree

- Divide and conquer can also be applied where a Task is divided into more than two parts at each stage.
- Example: Task broken into four parts. The sequential recursive definition would be

```
int addlist *s)        {          /* add list of numbers, s */
  if (numbers(s) <= 4)
    return(s1 + s2 + s3 + s4);
  else {
    divide (s,s1,s2,s3,s4);      /* divide s into s1,s2,s3,s4*/
    part_sum1 = add(s1);         /*recursive calls to add sublists */
    part_sum2 = add(s2);
    part_sum3 = add(s3);
    part_sum4 = add(s4);
    return (part_sum1 + part_sum2 + part_sum3 + part_sum4);
  }
}
```

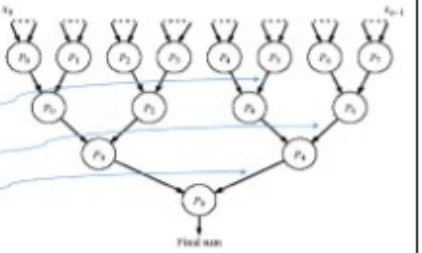
- A tree in which each node has four children is called a quadtree.

```
recv(s1, P0);
divide(s1, s1, s2);
send(s2, P4);
divide(s1, s1, s2);
send(s2, P2);
part_sum = *s1;
recv(&part_sum1, P1);
part_sum = part_sum + part_sum1;
recv(&part_sum1, P2);
part_sum = part_sum + part_sum1;
send(&part_sum, P1);
```



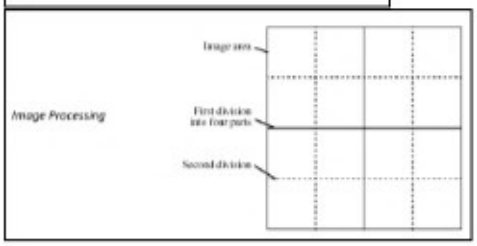
Parallel Code (P_4)

```
recv(s1, P4);
divide(s1, s1, s2);
send(s2, P6);
divide(s1, s1, s2);
send(s2, P5);
part_sum = *s1;
recv(&part_sum1, P5);
part_sum = part_sum + part_sum1;
recv(&part_sum1, P6);
part_sum = part_sum + part_sum1;
send(&part_sum, P4);
```



- Similar sequences are required for the other processes

QuadTree



Shazwan

Lecture 6: Partitioning & Divide and Conquer – Example I

Sorting using Bucket Sort

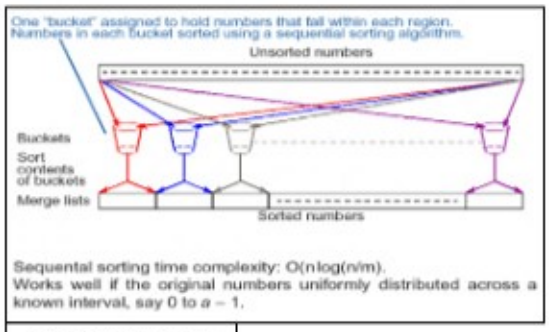
Range of numbers is divided into m equal regions, 0 to a/m - 1, a/m to 2a/m - 1, 2a/m to 3a/m - 1, ...

One "bucket" is assigned to hold numbers that fall within each region.

The numbers are placed into the appropriate buckets.

The numbers in each bucket will be sorted using a sequential sorting algorithm.

Works well if the original numbers are uniformly distributed across a known interval, say 0 to a - 1.



Sequential sorting time complexity: O(n log(n/m).
Works well if the original numbers uniformly distributed across a known interval, say 0 to $a - 1$.

Sequential Bucket Sort

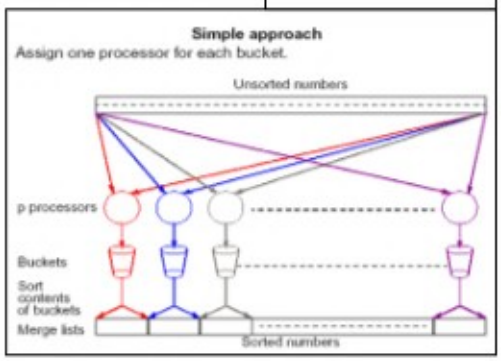
Parallel Bucket Sort

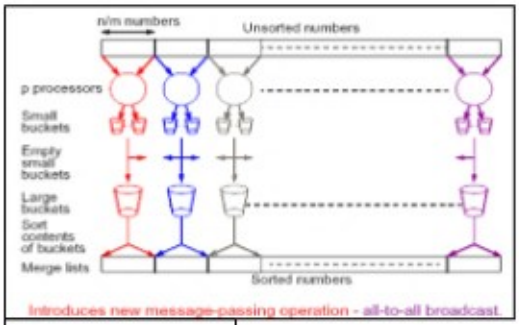
Further Parallelism

By partitioning the sequence into m regions, one region for each processor.

Each processor maintains p "small" buckets and separates the numbers in its region into its own small buckets.

These small buckets are then "emptied" into the p final buckets for sorting, which requires each processor to send one small bucket to each of the other processors (bucket i to processor i).





More Parallelism

Sequential Time

ts = n + m((n/m)log(n/m)) = n + n log(n/m) = O(n log(n/m))

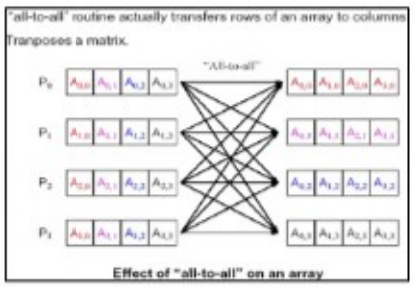
Parallel Time

Bucket sort can be parallelized by assigning one processor for each bucket – reduces second term in the preceding equation to (n/p)log(n/p) for p processors (where p = m).

In essence, each processor must communicate with every processor and an "all-to-all" mechanism would be appropriate.

An all-to-all routine sends data from each process to every other process and is illustrated on the next slide. This type of routine is available in MPI (MPI_Alltoall()), which is more efficient than using individual send()s and recv()s.

The all-to-all routine will actually transfer the rows of an array to column (and hence transpose a matrix).



- Phase 1 — Computation and Communication (Partition numbers)

  $t_{comm1} = t_{startup} + t_{data}n$

- Phase 2 — Computation (Sort into Small Buckets)

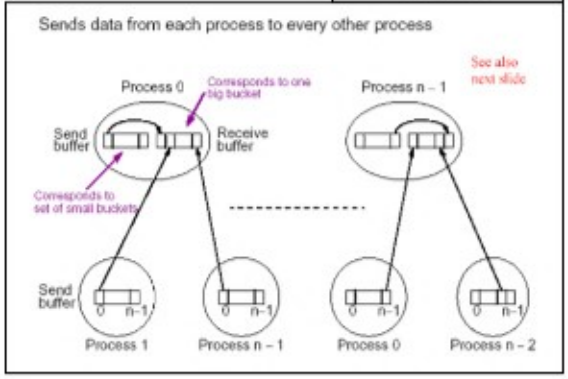
  $t_{comp2} = n/p$

- Phase 3 — Communication (Send to Large Buckets)
  - If all the communications could overlap:

  $t_{comm3} = (p - 1)(t_{startup} + (n/p^2)t_{data})$

Analysis

All-to-All Broadcast Routine



- Phase 4 — Computation (Sort Large Buckets)

  $t_{comp4} = (n/p)\log(n/p)$

- Overall

  $$t_p = t_{startup} + t_{data}n + n/p + (p - 1)(t_{startup} + (n/p^2)t_{data}) + (n/p)\log(n/p)$$

- It is assumed that the numbers are uniformly distributed to obtain these formula.
- Worst-case scenario would occur when all the numbers fall into one bucket!

Shazwan

Lecture 7: Numerical Integration

Previous example: Divide the problem and solve each sub-problem. The problem was assumed to be divided into equal parts and simple partitioning was employed.

Numerical Integration

Some times simple partitioning will not give the optimum solution, especially if the amount of work in each part is difficult to estimate.

Bucket sort, for example, is only effective when each region has approximately the same number of numbers.

A general divide-and-conquer technique divides the region continually into parts and lets some optimization function decide when certain regions are sufficiently divided. Let us take a different example, numerical integration

$$I = \int_a^b f(x)dx$$

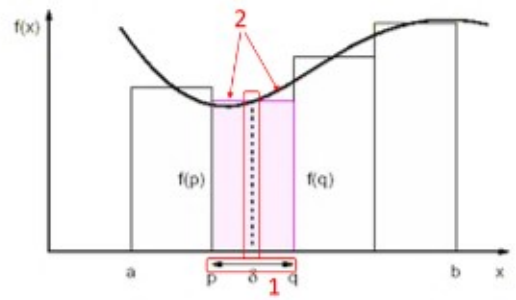
To integrate the function (i.e. to compute the 'area under the curve'), we can divide the area into separate parts, each of which can be calculated by a separate process.

Each region could be calculated using an approximation given by rectangle, where f(p) and f(q) are the heights of the two edges of the rectangular region and sigma is the width (interval).

The complete integration can be approximated by the summation of rectangular regions from a to b by aligning the rectangles so that the upper midpoint each rectangle intersect with the function. This has the advantage that errors on each side of midpoint end tend to cancel.

Each region calculated using an approximation given by rectangles:

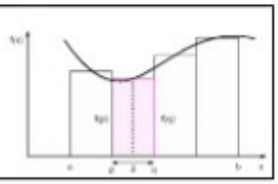
Aligning the rectangles:



Here we take the actual intersections of the lines with the function to create rectangles as shown in the fig.

- Each region is now calculated as

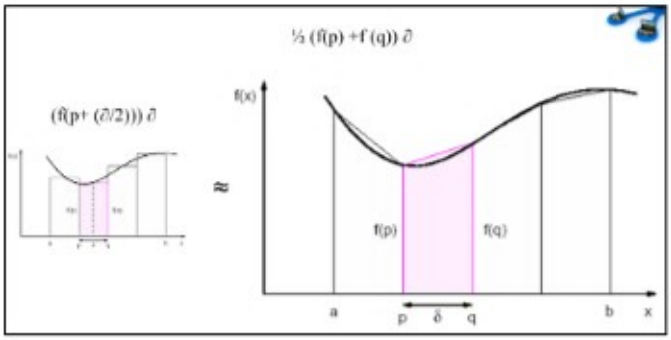
$$\{f(p+(\delta/2))\}\,\delta$$



Such approximate numerical methods for computing a definite integral using linear combination of values are called **quadrature methods**

$$\tfrac{1}{2}\,(f(p)+f(q))\,\delta$$

$$(f(p+(\delta/2)))\,\delta$$



Prior to start of the computation, one process is statically assigned to be responsible for computing each region. By making the interval smaller, we come closer to attaining the exact solution.

Let us consider a trapezoidal method.

Since each calculation is of the same form, the SPMD model is appropriate.

Suppose we were to sum the area from x=a to x=b using p processes, numbered 0 to p-1. The size of the region for each process is (b-a)/p. To calculate the area in the described manner, a section of SPMD pseudo-code could be as follows for Process Pi

```
if (i == master) {
        printf("Enter number of intervals ");
        scanf(%d",&n);                          // read number of intervals required
}
bcast(&n, P_group);                             // broadcast interval to all processes
region = (b - a)/p;                             // length of region for each process
start = a + region * i;                         // starting x coordinate for process f(p)
end = start + region;                           // ending x coordinate for process f(q)
d = (b - a)/n;                                  // size of interval / width δ
area = 0.0;
for (x = start; x < end; x = x + d)
        area = area + f(x) + f(x+d);
area = 0.5 * area * d;
reduce_add(&integral, &area, P_group);          // form sum of areas
```

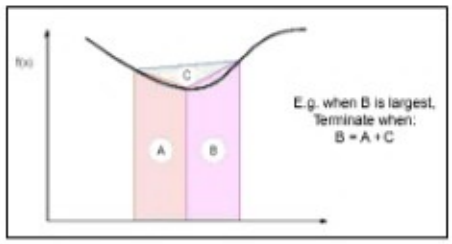
Shazwan

Lecture 7: Adaptive Quadrature

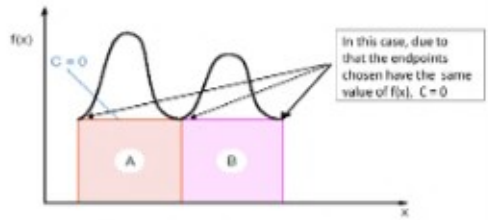
Method whereby the solution adapts to the shape of the curve. Example

Adaptive Quadrature

Use three areas, A, B, and C. The computation is terminated when the area computed for the largest of the A and B regions is sufficiently close to the sum of the areas computed for the other two regions.



Some care might be needed in choosing when to terminate.



Might cause us to terminate early, as two large regions are the same (i.e., C = 0).

Adaptive Quadrature with False Termination

Computations of areas under slowly varying parts of the curve stop earlier than computations of areas under more rapidly varying parts.

The spacing sigma will vary across the interval. The consequent of this is that a fixed process assignment will not lead to the most efficient use of processors, for which some care might be needed in choosing when to terminate.

For example, the function such as shown below in the fig might cause us to terminate early, as the two large regions are the same (i.e. C=0).

Gravitational N-Body Problem

Objective is to find the positions and movements of bodies in space (say planets) that are subject to gravitational forces from other bodies using Newtonian laws of physics.

The gravitational force between two bodies of masses m_a and m_b is given by

$$F = \frac{Gm_am_b}{r^2}$$

where G is the gravitational constant and r is the distance between the bodies.

Subject to forces, a body will accelerate according to Newton's second law:

$$F = ma$$

where m is the mass of the body, F is the force it experiences, and a is the resultant acceleration.

Let the time interval be Δt. Hence, for a body with mass m, the resulting F applied will be

$$F = \frac{m(v^{t+1} - v^t)}{\Delta t}$$

Note that once the bodies move into a new position, the force (F) changes. The computation has to be repeated.

Over the time Δt interval, the position changes

$$x^{t+1} = x^t + v\,\Delta t$$

where x^t is its position at time t

The new velocity will be

$$v^{t+1} = v^t + \frac{F\Delta t}{m}$$

where v^{t+1} will be the velocity at time t + 1, and v^t is the velocity at time t.

In a 3 dimensional space having a coordinate system (x, y, z), the distance between the bodies at (x_a, y_a, z_a) and (x_b, y_b, z_b) is given by

$$r = \sqrt{(x_b - x_a)^2 + (y_b - y_a)^2 + (z_b - z_a)^2}$$

Most of the problems that we want to solve, such as weather forecasting, involves such computation and they are in a 3 dimensional space.

Three Dimensional Space

Sequential Code

- The overall gravitational N-body computation can be described by the algorithm.

```
for (t = 0; t < tmax; t++) {                  // for each time period
  for (i = 0; i < N; i++) {                   // for each body (particle)
    F = Force_routine(i);                     // compute the force for body i
    v_new[i] = v[i] + F * dt / m;             // compute new velocity and
    x_new[i] = x[i] + v_new[i] * dt;          // new position
  }
  for (i = 0; i < nmax; i++) {                // for each body (particle)
    x[i] = x_new[i];                          // update the position
    v[i] = v_new[i];                          // and the velocity
  }
}                                             // repeat for next time period
```

$$F = \frac{Gm_am_b}{r^2}$$

The forces are resolve in the 3 directions, using (for example)

$$F_x = \frac{Gm_am_b}{r^2}\left(\frac{x_b - x_a}{r}\right)$$
$$F_y = \frac{Gm_am_b}{r^2}\left(\frac{y_b - y_a}{r}\right)$$
$$F_z = \frac{Gm_am_b}{r^2}\left(\frac{z_b - z_a}{r}\right)$$

where the particles (bodies) of mass m_a and m_b and have the coordinates (x_a, y_a, z_a) and (x_b, y_b, z_b)

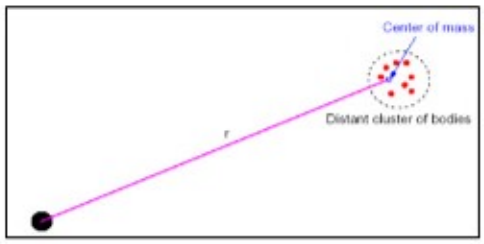
Shazwan

Lecture 7: Adaptive Quadrature

The algorithm is an O N^2 algorithm (for one iteration) as each of the N bodies is influenced by each of the other N - 1 bodies.

Not feasible to use this direct algorithm for most interesting N-body problems where N is very large.

Time complexity can be reduced using the observation that a cluster of distant bodies can be approximated as a single distant body of the total mass of the cluster situated at the centre of mass of the cluster.



Barnes-Hut Algorithm

Start with whole space (3-Dimensional) in which one cube contains the bodies (or particles).

First, this cube is divided into eight sub-cubes.

If a sub-cube contains no particles, the sub-cube is deleted from further consideration.

If a sub-cube contains more than one body, it is recursively divided until every sub-cube contains one body. (If a sub-cube contains one body, stop recursion).

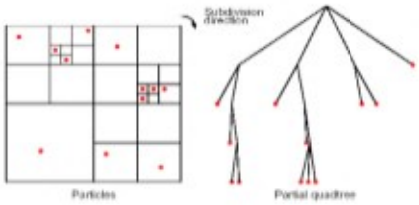
This process creates an oct-tree; that is, a tree with up to eight edges from each node.

The leaves represent cells each containing one body.

After the tree has been constructed, the total mass and centre of mass of the sub-cube is stored at each node.

Barnes-Hut Algorithm (2)

- For a 2 dimensional problem, each recursive sub division will create four sub-areas and a quad-tree.
- In general, the tree will be very unbalanced.
- Figure below illustrates the decomposition for a two dimensional space and resultant quadtree.
- The three dimensional case follows the same construction except with up to eight edges from each node



Barnes-Hut Algorithm (3)

If a sub-cube contains more than one body, it is recursively divided until every sub-cube contains one body. (If a sub-cube contains one body, stop recursion).

$$r > \frac{d}{\theta}$$

where θ is a constant typically 1.0 or less (θ is called the opening angle).

Constructing the tree requires a time of O n log n , and so does computing all the forces, so that the overall time complexity of the method is O n log n . (Remember earlier, the complexity of a tree structure is O log n and at each level we need to "decompose" it into n bodies.

The Barnes-Hut Algorithm creates a very unbalanced tree.

Orthogonal Recursive Bisection

A better way of dividing the groups would be **Orthogonal Recursive Bisection**

Consider a two-dimensional square area

First, a vertical line is found that divides the area into two areas each with an equal number of bodies.

For each area, a horizontal line is found that divides it into two areas each with an equal number of bodies.

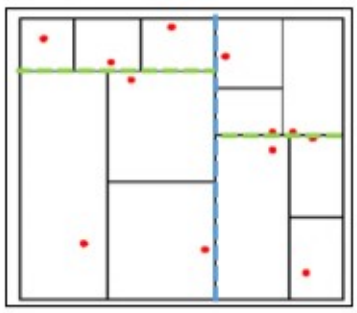
Repeated until there are as many areas as processors, and then one processor is assigned to each area.

For 12 processors, i.e. n = 12

a **vertical line** is found that divides the area into two areas each with an equal number of bodies.

Repeated until there are as many areas as processors, and then one processor is assigned to each area.

For each area, a **horizontal line** is found that divides it into two areas each with an equal number of bodies.



Shazwan

Lecture 7: Embarrassingly Parallel

A computation that can be divided into a number of completely independent parts, each of which can be executed by a separate processor.

Each process requires different or the same data and produces results from its input data without any need for results from other processes.

Embarrassingly Parallel

Master-slave approach for data transmission and processing in embarrassingly parallel computation is shown below:

Can achieve Maximum possible speedup