

# Day 3:

# High Performance Computing

# CMT106

David W. Walker

Professor of High Performance Computing

Cardiff University

<http://www.cardiff.ac.uk/people/view/118172-walker-david>

# Day 3

- 9:30 – 10:30am: **Lecture** on networks, efficiency, speed-up; performance analysis; scalable algorithms; Amdahl's Law.
- 10:30 – 10:50am: **Break**.
- 10:50am – 12:00pm: **Lecture** on point-to-point message-passing with MPI with simple examples.
- 12:00 – 1:30pm: Lunch break.
- 1:30 – 3:15pm: **Self-study**, read “The design of a standard message passing interface for distributed memory concurrent computers” (includes 15min break).
- 3:15 – 3:45pm: **Discussion** about the paper.
- 3:45pm – 5:00pm: **Lecture** and wrap-up. Collective communication; the integration example; application topologies.

# Topics Covered on Days 1-4

- *Day 1:* Introduction to parallelism; motivation; types of parallelism; Top500 list; classification of machines; SPMD programs; memory models; shared and distributed memory; OpenMP; example of summing numbers.
- *Day 2:* Interconnection networks; network metrics; classification of parallel algorithms; speedup and efficiency.
- *Day 3:* Scalable algorithms; Amdahl's law; sending and receiving messages; programming with MPI; collective communication; integration example.

# Topics Covered on Days 5-7

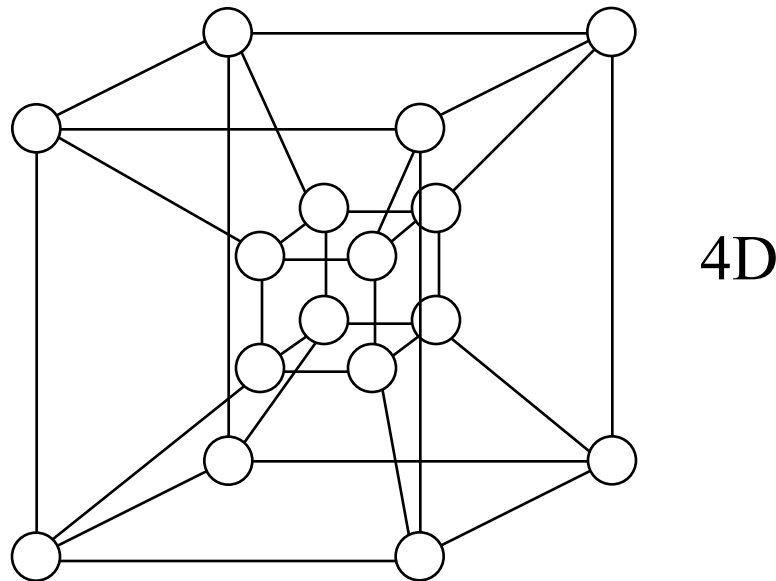
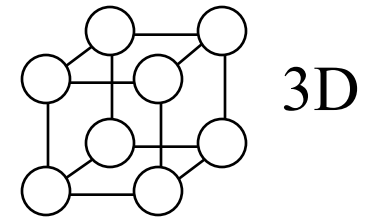
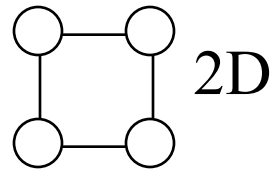
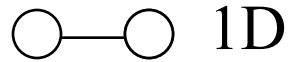
- *Day 4:* Regular computations and simple example 1D and 2D problems – the wave equation and Laplace equation.
- *Day 5:* High performance computing on GPUs. Parallel programming with CUDA on Nvidia GPUs.
- *Day 6:* Dynamic communication and the molecular dynamics example; irregular computations; the WaTor simulation .
- *Day 7:* Load balancing strategies; message passing libraries; block-cyclic data distribution.

# Hypercube Networks

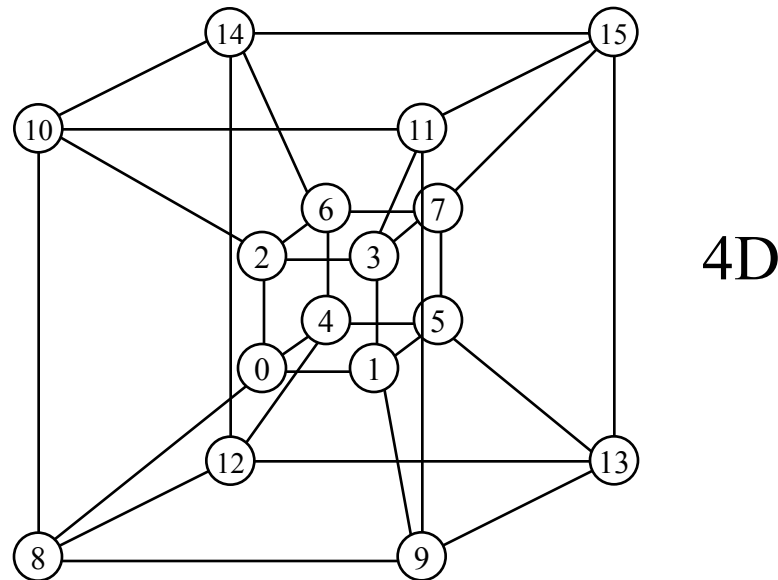
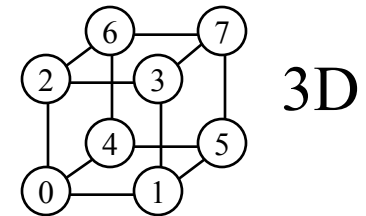
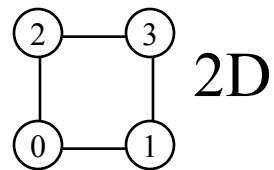
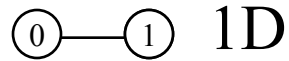
- A hypercube network consists of  $n=2^k$  nodes arranged as a  $k$ -dimensional hypercube. Sometimes called a *binary  $n$ -cube*.
- Nodes are numbered  $0, 1, \dots, n-1$ , and two nodes are connected if their node numbers differ in exactly one bit.
  - Network connectivity =  $k$
  - Network diameter =  $k$
  - Network narrowness =  $1$
  - Bisection width =  $2^{k-1}$
  - Expansion increment =  $2^k$
  - Edges per node =  $k$

# Examples of Hypercubes

See <http://en.wikipedia.org/wiki/Hypercube>



# Numbering Hypercube nodes



# Mapping Grids to Hypercubes

- In the example in which we summed a set of numbers over a square mesh of processors each processor needs to know where it is in the mesh.
- We need to be able to map node numbers to locations in the process mesh
  - Given node number  $k$  what is its location  $(i,j)$  in the processor mesh?
  - Given a location  $(i,j)$  in the processor mesh what is the node number,  $k$ , of the processor at that location?
  - We want to choose a mapping such that neighbouring processes in the mesh are also neighbours in the hypercube. This ensures that when neighbouring processes in the mesh communicate, this entails communication between neighbouring processes in the hypercube.



# Binary Gray Codes

- Consider just one dimension – a periodic processor mesh in this case is just a ring.
- Let  $G(i)$  be the node number of the processor at position  $i$  in the ring, where  $0 \leq i < n$ . The mapping  $G$  must satisfy the following,
  - It must be unique, i.e.,  $G(i) = G(j) \iff i = j$ .
  - $G(i)$  and  $G(i-1)$  must differ in exactly one bit for all  $i$ ,  $0 \leq i < n-1$ .
  - $G(n-1)$  and  $G(0)$  must differ in exactly one bit.

# Binary Gray Codes 2

- A class of mappings known as *binary Gray codes* satisfy these requirements. There are several  $n$ -bit Gray codes. Binary Gray codes can be defined recursively as follows:

Given a  $d$ -bit Gray code, a  $(d+1)$ -bit Gray code can be constructed by listing the  $d$ -bit Gray code with the prefix 0, followed by the  $d$ -bit Gray code in reverse order with prefix 1.

# Example of a Gray Code

- Start with the Gray code  $G(0)=0$ ,  $G(1)=1$ .
- Then the 2-bit Gray code is given in Table 1, and the 3-bit Gray code is given in Table 2.

i	$[G(i)]_2$	G(i)
0	00	0
1	01	1
2	11	3
3	10	2

Table 1: A 2-bit Gray code

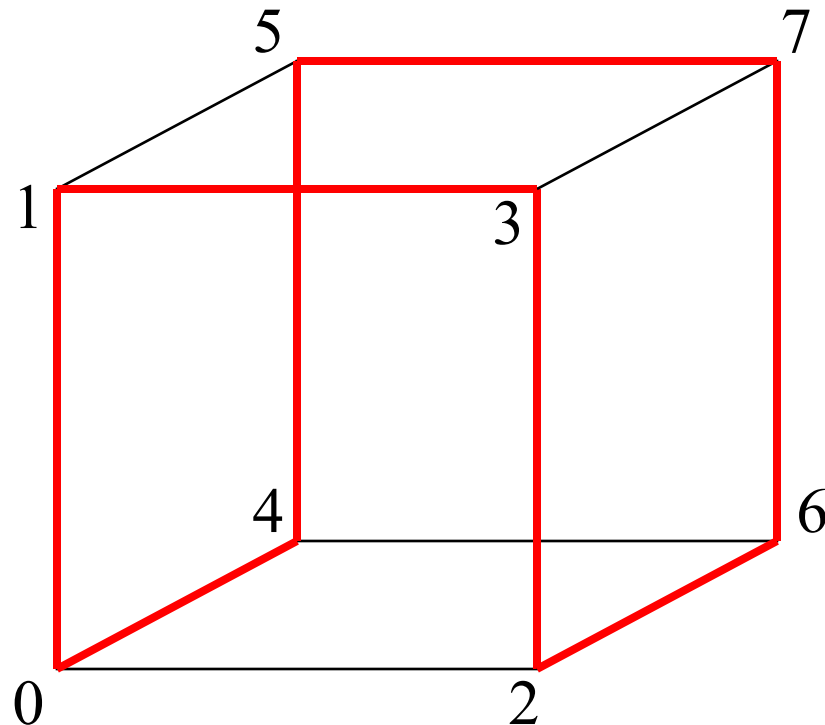
# Example of a Gray Code 2

i	$[G(i)]_2$	G(i)
0	000	0
1	001	1
2	011	3
3	010	2
4	110	6
5	111	7
6	101	5
7	100	4

Table 2: A 3-bit Gray code

# Example of a Gray Code 3

- A ring can be embedded in a hypercube as follows:



# Multi-Dimensional Gray Codes

- To map a multidimensional mesh of processors to a hypercube we require that the number of processors in each direction of the mesh be a power of 2. So

$$2^{d_{r-1}} \times 2^{d_{r-2}} \times \dots \times 2^{d_0}$$

is an  $r$ -dimensional mesh and if  $d$  is the hypercube dimension then:

$$d_0 + d_1 + \dots + d_{r-1} = d$$

# Multi-Dimensional Gray Codes 2

- We partition the bits of the node number and assign them to each dimension of the mesh. The first  $d_0$  go to dimension 0, the next  $d_1$  bits go to dimension 1, and so on. Then we apply separate inverse Gray code mappings to each group of bits.

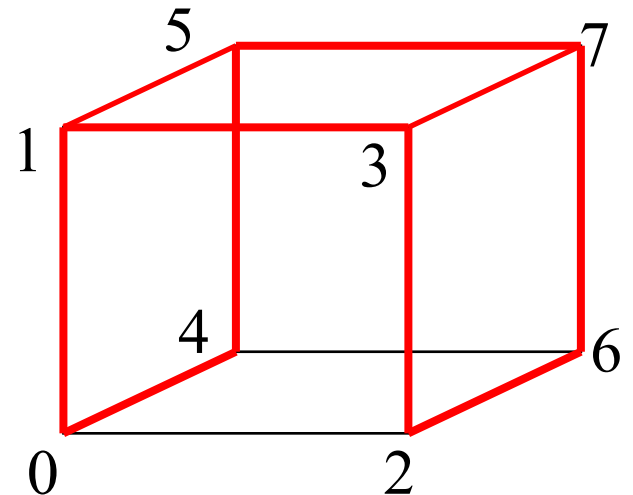
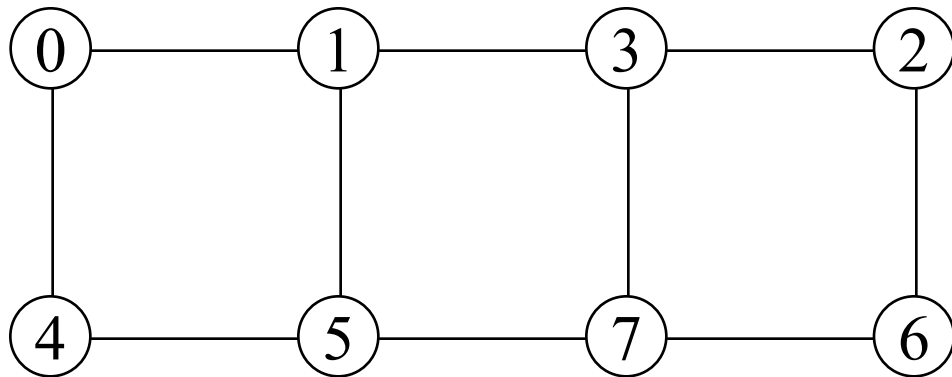
# Mapping a $2 \times 4$ Mesh to a Hypercube

K	$[k_1]_2, [k_0]_2$	$[G^{-1}(k_1)]_2, [G^{-1}(k_0)]_2$	(i,j)
0	0, 00	0, 00	(0,0)
1	0, 01	0, 01	(0,1)
2	0, 10	0, 11	(0,3)
3	0, 11	0, 10	(0,2)
4	1, 00	1, 00	(1,0)
5	1, 01	1, 01	(1,1)
6	1, 10	1, 11	(1,3)
7	1, 11	1, 10	(1,2)



# Mapping a $2 \times 4$ Mesh to a Hypercube 2

- A  $2 \times 4$  mesh is embedded into a 3D hypercube as follows:

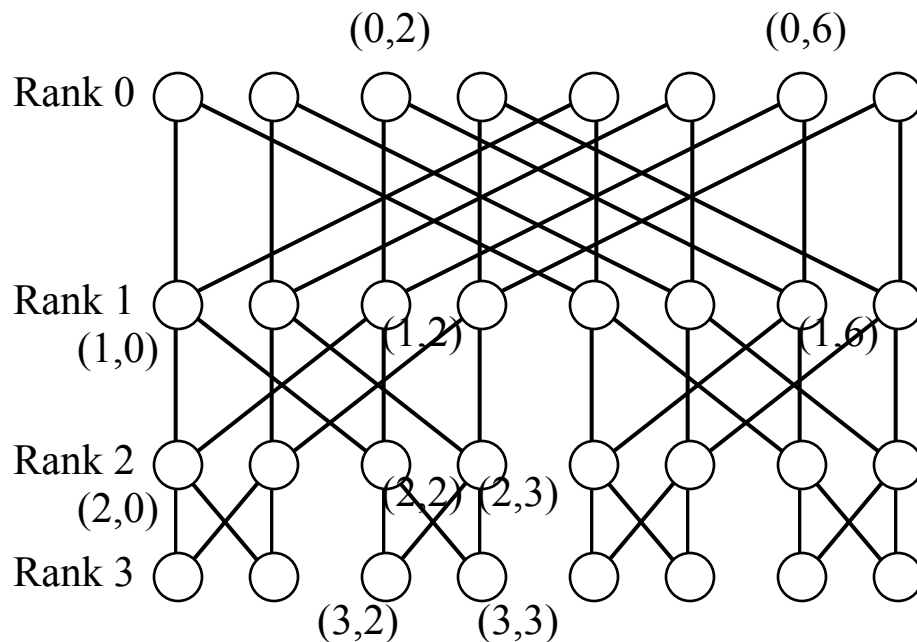


# Butterfly Network

- A butterfly network consists of  $(k+1)2^k$  nodes divided into  $k+1$  rows, or *ranks*.
- Let node  $(i,j)$  refer to the  $j$ th node in the  $i$ th rank. Then for  $i > 0$  node  $(i,j)$  is connected to 2 nodes in rank  $i-1$ , node  $(i-1,j)$  and node  $(i-1,m)$ , where  $m$  is the integer found by inverting the  $i$ th most significant bit of  $j$ .
- Note that if node  $(i,j)$  is connected to node  $(i-1,m)$ , then node  $(i,m)$  is connected to node  $(i-1,j)$ . This forms a butterfly pattern.
  - Network diameter =  $2k$
  - Bisection width =  $2^k$

# Example of a Butterfly Network

Here is a butterfly network for  $k = 3$ .



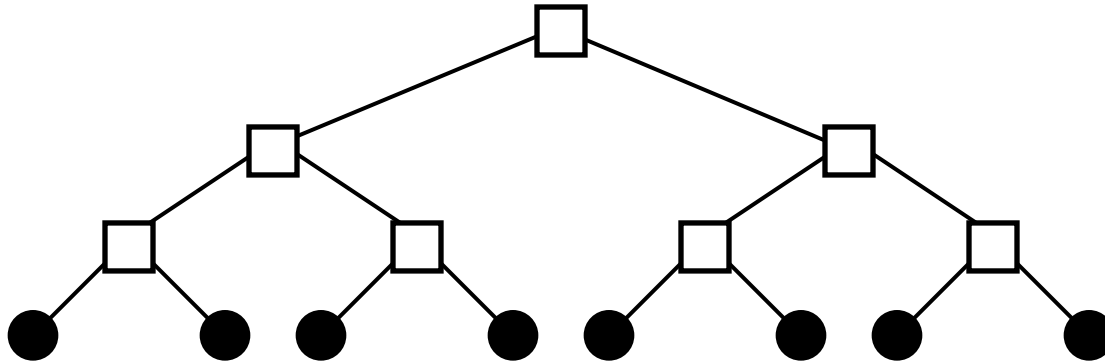
$$i = 1, j = 2 = (010)_2, j^{\text{q}} = (110)_2 = 6$$

$$i = 2, j = 2 = (010)_2, j^{\text{q}} = (000)_2 = 0$$

$$i = 3, j = 2 = (010)_2, j^{\text{q}} = (011)_2 = 3$$

# Complete Binary Tree Network

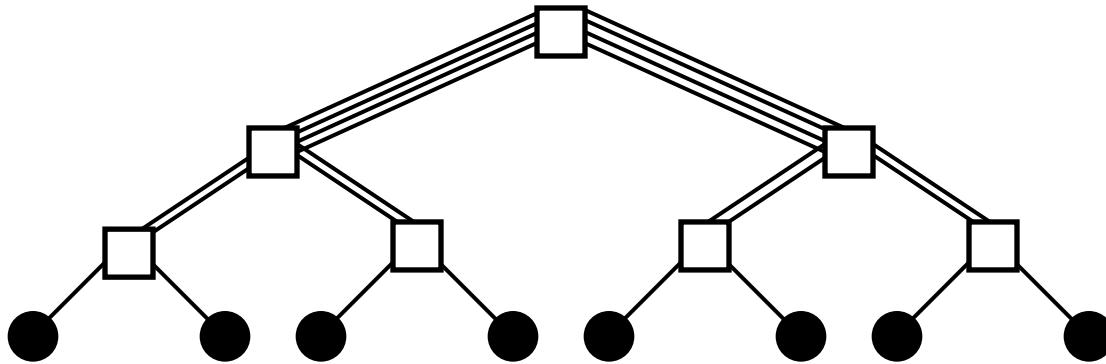
- Tree-based networks use switches to connect processors. An example is the binary tree network.



- This has a bisection width of 1, and a connectivity of 1. The low bisection width can result in congestion in the upper levels of the network.

# Fat Tree Network

- The fat tree network seeks to reduce the congestion in the upper levels of the network by adding extra links.



- The connectivity is still 1, but if there are  $2^d$  processing nodes the bisection width is  $2^{d-1}$ .
- This type of network was used in the CM-5.

# Classifying Parallel Algorithms

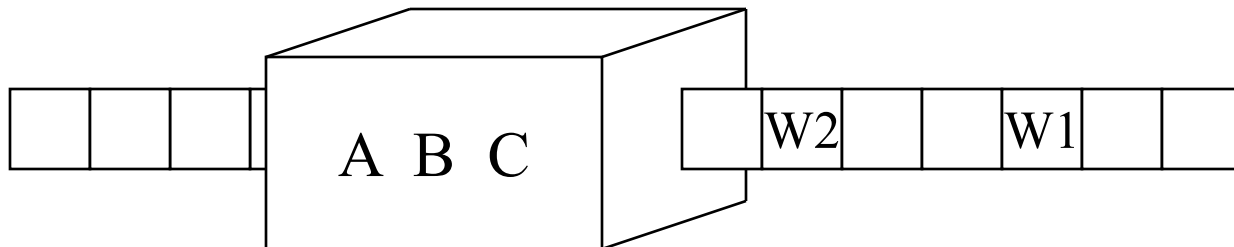
- Parallel algorithms for MIMD machines can be divided into 3 categories
  - Pipelined algorithms
  - Data parallel, or partitioned, algorithms
  - Asynchronous, or relaxed, algorithms

# Pipelined Algorithms

- A pipelined algorithm involves an ordered set of processes in which the output from one process is the input for the next.
- The input for the first process is the input for the algorithm.
- The output from the last process is the output of the algorithm.
- Data flows through the pipeline, being operated on by each process in turn.

# Pipelines Algorithms 2

- **Example:** Suppose it takes 3 steps, A, B, and C, to assemble an item, and each step takes one unit of time.
- In the sequential case it takes 3 time units to assemble each item.
- Thus it takes  $3n$  time units to produce  $n$  items.



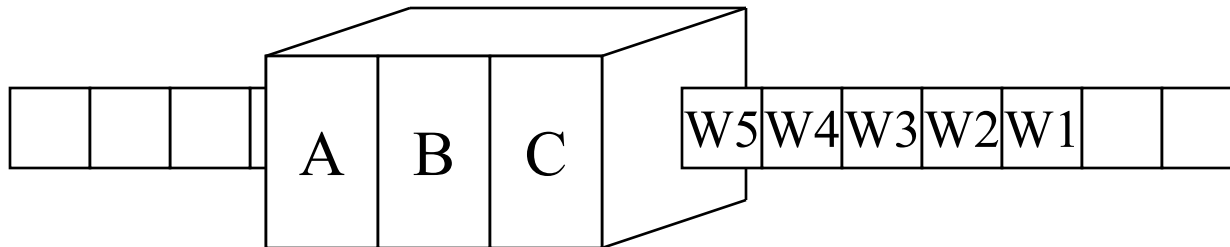


# Example of Pipelined Algorithm

- In the pipelined case the following happens
  - Time step 1: A operates on W1
  - Time step 2: A operates on W2, B operates on W1
  - Time step 3: A operates on W3, B operates on W2, C completes W1
  - Time step 4: A operates on W4, B operates on W3, C completes W2
- After 3 time units, a new item is produced every time step.

# Pipelined Algorithm

- If the pipeline is  $n$  processes long, a new item is produced every time step from the  $n$ th time step onwards. We then say the pipeline is *full*.
- The pipeline *start-up time* is  $n-1$ .
- This sort of parallelism is sometimes called *algorithmic parallelism*.



# Performance of Pipelining

If

- N is the number of steps to be performed
- T is the time for each step
- M is the number of items, then

Sequential time =  $NTM$

Pipelined time =  $(N+M-1)T$

# Pipeline Performance Example

If  $T = 1$ ,  $N = 100$ , and  $M = 10^6$ , then

- Sequential time =  $10^8$
- Pipelined time = 1000099

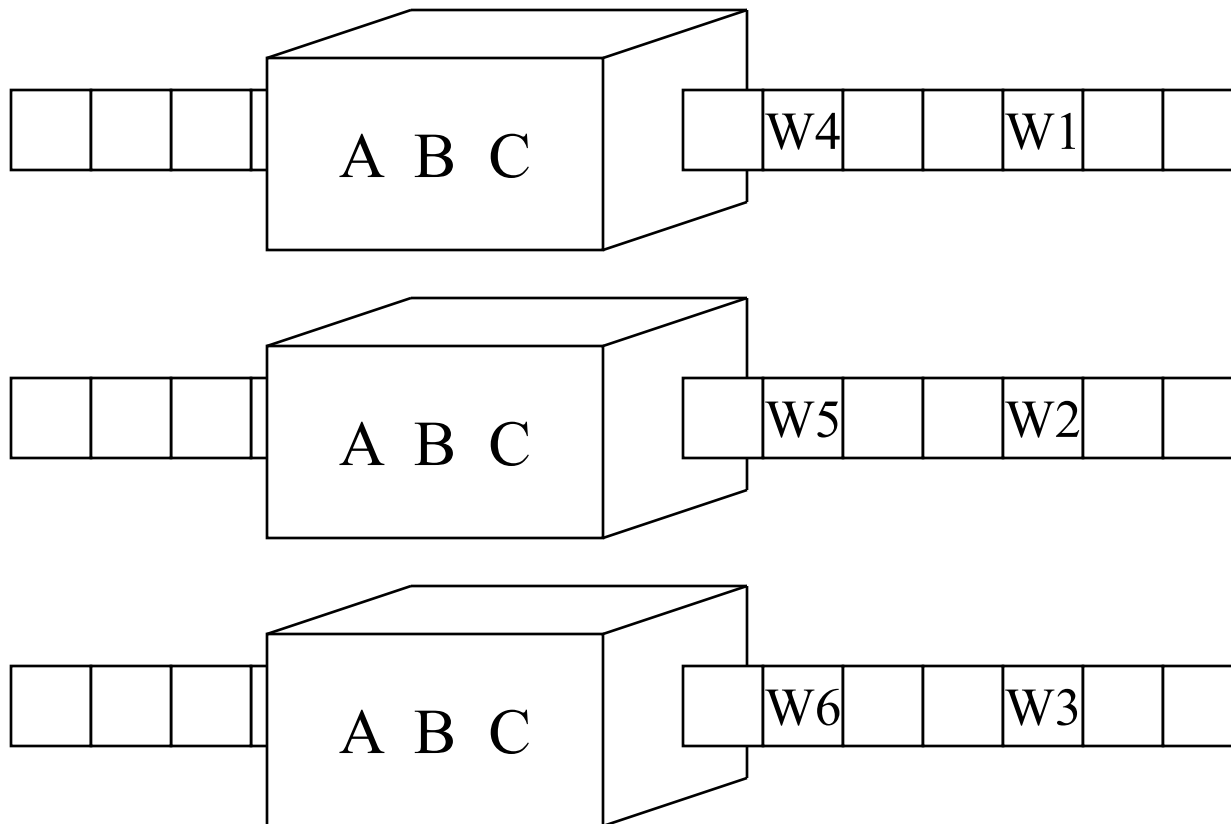
The speed-up  $T_{\text{seq}}/T_{\text{pipe}} \approx 100$ .

# Data Parallelism

- Often there is a natural way of decomposing the data into smaller parts, which are then allocated to different processors.
- This way of exploiting parallelism is called *data parallelism* or *geometric parallelism*.
- In general the processors can do different things to their data, but often they do the same thing.
- Processors combine the solutions to their sub-problems to form the complete solution. This may involve communication between processors.

# Data Parallelism Example

Data parallelism can be exploited in the example. For 3-way data parallelism we have:



# Relaxed Parallelism

- Relaxed parallelism arises when there is no explicit dependency between processes.
- Relaxed algorithms never wait for input – they use the most recently available data.

# Relaxed Parallelism Example

Suppose we have 2 processors, A and B.

- A produces a sequence of numbers,  $a_i$ ,  $i=1,2,\dots$
- B inputs  $a_i$  and performs some calculation on it to produce  $F_i$ .
- Say B runs much faster than A.



# Synchronous Operation

- A produces  $a_1$ , passes it to B, which calculates  $F_1$
- A produces  $a_2$ , passes it to B, which calculates  $F_2$
- and so on.....

# Asynchronous Operation

1. A produces  $a_1$ , passes it to B, which calculates  $F_1$
  2. A is in the process of computing  $a_2$ , but B does not wait – it uses  $a_1$  to calculate  $F_2$ , i.e.,  $F_1 = F_2$ .
- Asynchronous algorithms keep processors busy.  
Drawbacks of asynchronous algorithms are
    - they are difficult to analyse
    - an algorithm that is known to converge in synchronous mode may not converge in asynchronous mode.

# Example of Asynchronous Algorithm

- The Newton-Raphson method is an iterative algorithm for solving non-linear equations  $f(x)=0$ .

$$x_{n+1} = x_n - f(x_n) / f'(x_n)$$

- generates a sequence of approximations to the root, starting with some initial value  $x_0$ .

# Example

- Suppose  $f(x) = x^2 - 2$  so we have:

$$x_{n+1} = x_n - (x_n^2 - 2)/(2x_n)$$

- Choose  $x_0 = 1$ .
- $f(x) = 0$  has solution  $x = \sqrt{2}$

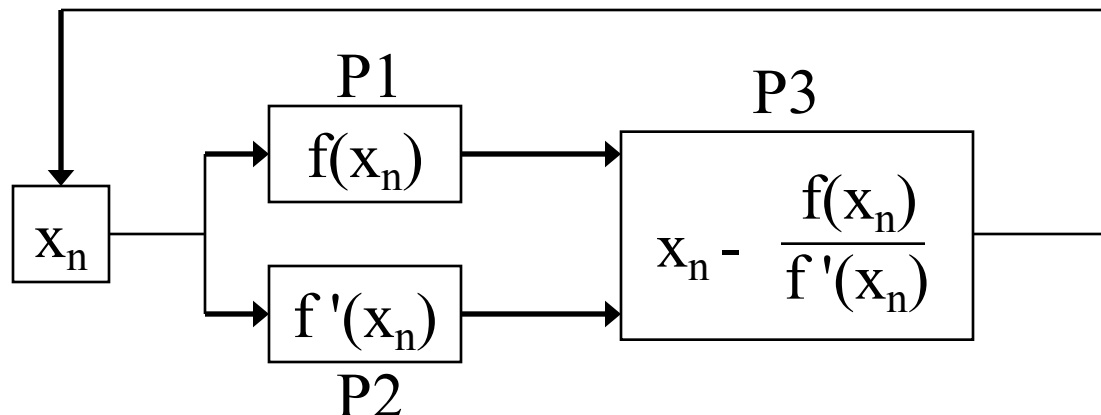
$x_0$	1
$x_1$	1.5
$x_2$	1.41666667
$x_3$	1.41421569
$x_4$	1.41421356

# Example of Asynchronous Algorithm 2

Suppose we have 3 processors

- P1: given  $x$ , P1 calculates  $f(x)$  in  $t_1$ , and sends it to P3.
- P2: given  $y$ , P2 calculates  $f'(y)$  in  $t_2$ , and sends it to P3.
- P3: given  $a$ ,  $b$ , and  $c$ , P3 calculates  $d = a - b/c$ .

If  $|d - a| > \epsilon$  then  $d$  is sent to P1 and P2; otherwise it is output.



# Serial Mode Time Complexity

## Serial mode

- P1 computes  $f(x_n)$ , then P2 computes  $f'(x_n)$ , then P3 computes  $x_{n+1}$ .
- Serial time is  $t_1 + t_2 + t_3$  per iteration.
- If  $k$  iterations are needed, total time is  $k(t_1 + t_2 + t_3)$

# Synchronous Parallel Mode

- P1 and P2 compute  $f(x_n)$  and  $f'(x_n)$  simultaneously, and when *both* have finished the values of  $f(x_n)$  and  $f'(x_n)$  are used by P3 to find  $x_{n+1}$ .
- Time per iteration is  $\max(t_1, t_2) + t_3$ .
- $k$  iterations are necessary so the total time is,  $k(\max(t_1, t_2) + t_3)$ .

# Asynchronous Parallel Mode

- P1 and P2 begin computing as soon as they receive a new input value from P3.
- P3 computes a new value as soon as it receives a new input value from *either* P1 *or* P2.



# Asynchronous Parallel Mode

## Example

- For example, if  $t_1=2$ ,  $t_2=3$  and  $t_3=1$ .
- $C_i$  indicates processor is using  $x_i$  in its calculation.
- Cannot predict number of iterations.

Time	P1	P2	P3
1	C0	C0	—
2	$f(x_0)$	C0	—
3	—	$f'(x_0)$	
4	—	—	$x_1 = x_0 - f(x_0) / f'(x_0)$
5	C1	C1	—
6	$f(x_1)$	C1	—
7	—	$f'(x_1)$	$x_2 = x_1 - f(x_1) / f'(x_1)$
8	C2	C2	$x_3 = x_2 - f(x_1) / f'(x_1)$
9	$f(x_2)$	C2	—
10	C3	$f'(x_2)$	$x_4 = x_3 - f(x_2) / f'(x_1)$
11	$f(x_3)$	C4	$x_5 = x_4 - f(x_2) / f'(x_2)$
12	C5	C4	$x_6 = x_5 - f(x_3) / f'(x_2)$

# Speed-up and Efficiency

- We now define some metrics which measure how effectively an algorithm exploits parallelism.
- **Speed-up** is the ratio of the time taken to run the best sequential algorithm on one processor of the parallel machine divided by the time to run on N processors of the parallel machine.

$$S(N) = T_{\text{seq}}/T_{\text{par}}(N)$$

- **Efficiency** is the speed-up per processor.

$$\epsilon(N) = S(N)/N = (1/N)(T_{\text{seq}}/T_{\text{par}}(N))$$

- **Overhead** is defined as

$$f(N) = 1/\epsilon(N) - 1$$

# Example

- Suppose the best known sequential algorithm takes 8 seconds, and a parallel algorithm takes 2 seconds on 5 processors. Then

$$\text{Speed-up} = 8/2 = 4$$

$$\text{Efficiency} = 4/5 = 0.8$$

$$\text{Overhead} = 1/0.8 - 1 = 0.25$$

# Self Speed-up and Linear Speed-up

- *Self speed-up* is defined using the parallel algorithm running on one processor.
- If the speed-up using  $N$  processors is  $N$  then the algorithm is said to exhibit *linear speed-up*.

# Factors That Limit Speed-up

## 1. Software Overhead

Even when the sequential and parallel algorithms perform the same computations, software overhead may be present in the parallel algorithm. This includes additional index calculations necessitated by how the data were decomposed and assigned to processors, and other sorts of “bookkeeping” required by the parallel algorithm but not the sequential algorithm.

# Factors That Limit Speed-up

## 2. Load Imbalance

Each processor should be assigned the same amount of work to do between synchronisation points. Otherwise some processors may be idle while waiting for others to catch up. This is known as load imbalance. The speedup is limited by the slowest processor.

# Factors That Limit Speed-up

## 3. Communication Overhead

Assuming that communication and calculation cannot be overlapped, then any time spent communicating data between processors reduces the speed-up.

# Grain Size

The *grain size* or *granularity* is the amount of work done between communication phases of an algorithm. We want the grain size to be large so the relative impact of communication is less.



# Definition of Load Imbalance

- Suppose the work done by processor  $i$  between two successive synchronisation points is  $W_i$
- If the number of processors is  $N$ , then the average workload is:

$$\overline{W} = \left( \frac{1}{N} \right) \sum_{i=0}^{N-1} W_i$$

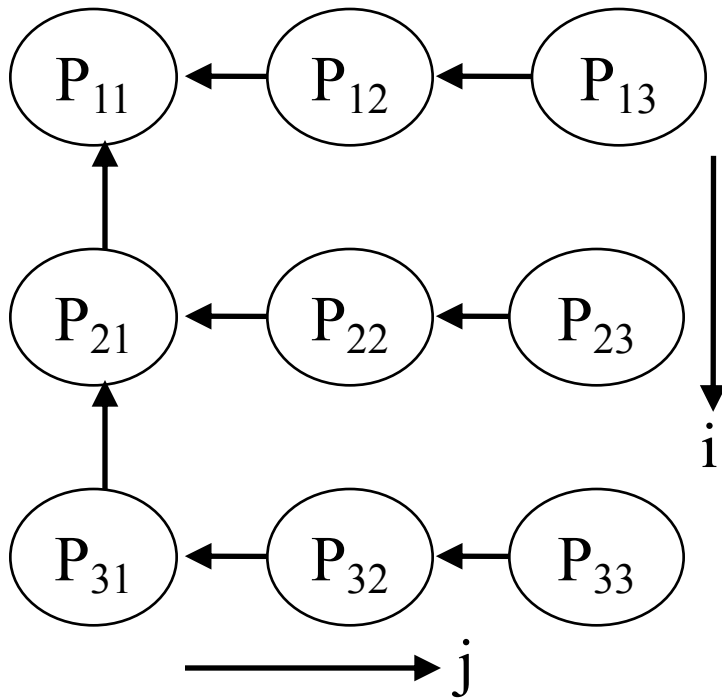
- The amount of load imbalance is then given by:

$$L = \max \left( \frac{W_i - \overline{W}}{\overline{W}} \right)$$

where the maximum is taken over all processors.

# Analysis of Summing Example

Recall the example of summing  $m$  numbers on a square mesh of  $N$  processors.



The algorithm proceeds as follows

1. Each processor finds the local sum of its  $m/N$  numbers
2. Each processor passes its local sum to another processor in a coordinated way
3. The global sum is finally in processor  $P_{11}$ .

# Analysis of Summing Example 2

- Time for best sequential algorithm is

$$T_{\text{seq}} = (m-1)t_{\text{calc}}$$

where  $t_{\text{calc}}$  is time to perform one floating-point operation.

- Time for each phase of parallel algorithm
  - Form local sums  $T_1 = (m/N-1) t_{\text{calc}}$
  - Sum along processor rows  $T_2 = (\sqrt{N} - 1)(t_{\text{calc}} + t_{\text{comm}})$
  - where  $t_{\text{comm}}$  is time to communicate one floating-point number between neighbouring processors.
  - Sum up first column of processors  $T_3 = (\sqrt{N} - 1)(t_{\text{calc}} + t_{\text{comm}})$

# Analysis of Summing Example 3

- Total time for the parallel algorithm is:

$$T_{\text{par}} = (m/N + 2\sqrt{N} - 3)t_{\text{calc}} + 2(\sqrt{N} - 1)t_{\text{comm}}$$

- So the speed-up for the summing example is:

$$\begin{aligned} S(N) &= \frac{(m-1)t_{\text{calc}}}{(m/N + 2\sqrt{N} - 3)t_{\text{calc}} + 2(\sqrt{N} - 1)t_{\text{comm}}} \\ &= \frac{N(1-1/m)}{1 + (N/m)(2\sqrt{N} - 3) + 2(N/m)(\sqrt{N} - 1)\tau} \end{aligned}$$

where  $\tau = t_{\text{comm}} / t_{\text{calc}}$

# Analysis of Summing Example 4

- In this algorithm a good measure of the grain size,  $g$ , is the number of elements per processor,  $m/N$ . We can write  $S$  as:

$$S(g,N) = \frac{N(1-1/m)}{1 + (N/m)(2\sqrt{N} - 3) + 2(N/m)(\sqrt{N} - 1)\tau}$$

- As  $g \rightarrow \infty$  with  $N$  constant,  $S \rightarrow N$ .
- As  $N \rightarrow \infty$  with  $g$  constant,  $S \approx g\sqrt{N}/(2(1+\tau))$ .
- As  $N \rightarrow \infty$  with  $m$  constant,  $S \rightarrow 0$ .

# Analysis of Summing Example 5

- If  $m \gg 1$  and  $N \gg 1$ ,

$$S(g,N) = \frac{N}{1 + 2 \sqrt{N(1+\tau)}/g}$$

$$\varepsilon(g,N) = \frac{1}{1 + 2 \sqrt{N(1+\tau)}/g}$$

$$f(g,N) = 2 \sqrt{N(1+\tau)}/g$$

# Scalable Algorithms

- Scalability is a measure of how effectively an algorithm makes use of additional processors.
- An algorithm is said to be *scalable* if it is possible to keep the efficiency constant by increasing the problem size as the number of processors increases.
- An algorithm is said to be *perfectly scalable* if the efficiency remains constant when the problem size and the number of processors increase by the same factor.
- An algorithm is said to be *highly scalable* if the efficiency depends only weakly on the number of processors when the problem size and the number of processors increase by the same factor.

# Scalability of the Summing Example

- The summing algorithm is scalable since we can take  $g$  proportional to  $\sqrt{N}$ .
- The summing algorithm is not perfectly scalable, but it is highly scalable.
- “Problem size” may be either:
  - the work performed, or
  - the size of the data.



# Amdahl's Law

- Amdahl's Law states that the maximum speedup of an algorithm is limited by the relative number of operations that must be performed sequentially, i.e., by its *serial fraction*.
- If  $\alpha$  is the serial fraction,  $n$  is the number of operations in the sequential algorithm, and  $N$  the number of processors, then the time for the parallel algorithm is:

$$T_{\text{par}}(N) = (\alpha n + (1 - \alpha)n/N)t + C(n, N)$$

where  $C(n, N)$  is the time for overhead due to communication, load balancing, etc., and  $t$  is the time for one operation.

# Derivation of Amdahl's Law

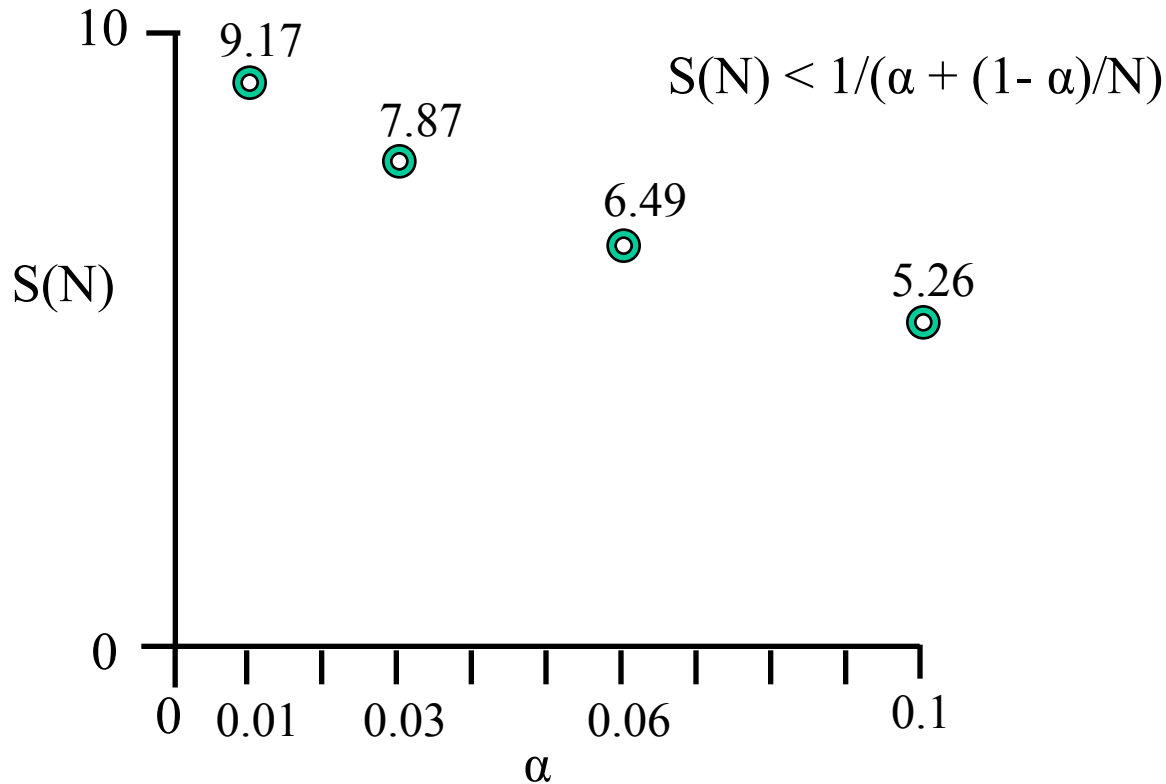
- The speed-up satisfies:

$$\begin{aligned} S(N) &= T_{\text{seq}}/T_{\text{par}}(N) = nt/[(\alpha n + (1-\alpha)n/N)t + C(n,N)] \\ &= 1/[(\alpha + (1-\alpha)/N) + C(n,N)/(nt)] \\ &< 1/(\alpha + (1-\alpha)/N) \end{aligned}$$

- Note that as  $N \rightarrow \infty$ , then  $S(N) \rightarrow 1/\alpha$ , so the speed-up is always limited to a maximum of  $1/\alpha$  no matter how many processors are used.

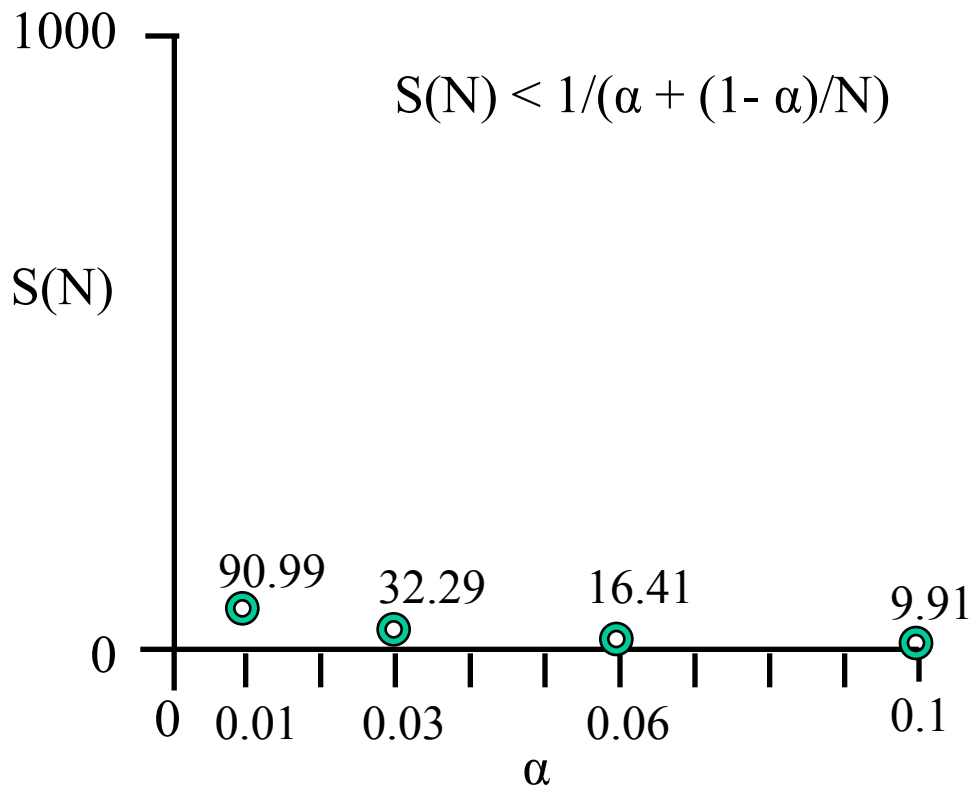
# Examples of Amdahl's Law

Consider the effect of Amdahl's Law on speed-up as a function of serial fraction,  $\alpha$ , for  $N=10$  processors.



# Examples of Amdahl's Law 2

Consider the effect of Amdahl's Law on speed-up as a function of serial fraction,  $\alpha$ , for  $N=1000$  processors.



If 1% of a parallel program involves serial code, the maximum speed-up is 9 on a 10-processor machine, but only 91 on a 1000-processor machine.

# Implications of Amdahl's Law

- Amdahl's Law says that the serial fraction puts a severe constraint on the speed-up that can be achieved as the number of processors increases.
- Amdahl's Law suggests that it is not cost effective to build systems with large numbers of processors because sufficient speed-up will not be achieved.
- It turns out that most important applications that need to be parallelised contain very small serial fractions, so large machines are justified.

# Speed-Up for Large Problems

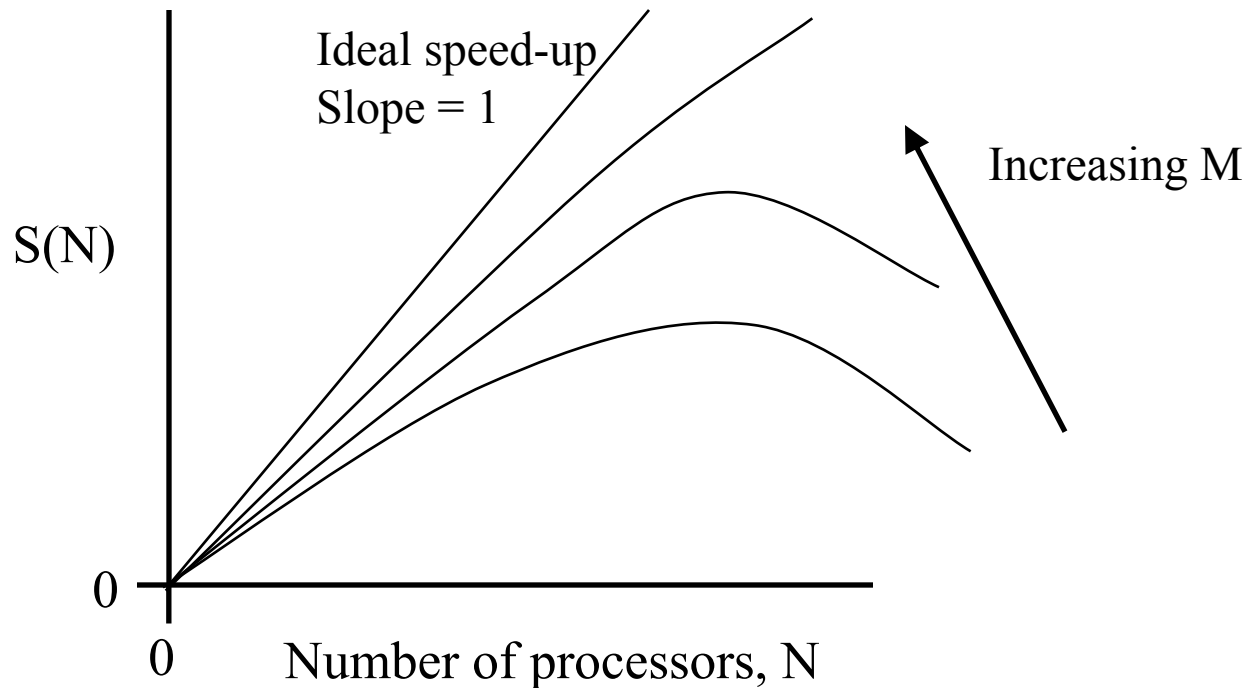
- *Speed-up* is the ratio between how long the best sequential algorithm takes on a single processor and how long it takes to run on multiple processors.
- To measure the speed-up the problem must be small enough to fit into the memory of one processor.
- This limits us to measuring the speed-up of only small problems.

# Speed-Up for Large Problems 2

- In finding the speedup we can *estimate* the time to run on one processor, so much larger problems can be considered.
- In general overhead costs increase with problem size, but at a slower rate than the amount of computational work (measured by the grain size). Thus, speed-up is an increasing function of problem size, and so this approach to speed-up allows us to measure larger speed-ups.

# Speed-Up and Problem Size

For a given number of processors, speed-up usually increases with problem size,  $M$ .





# Speed-up and Scaling

- Investigate the *strong scaling* of an application or algorithm by measuring the speed-up as the number of processors increases, for a fixed problem size.
- Investigate the *weak scaling* by measuring the speed-up as the number of processors increases, for a fixed problem size per processor.

# Semantics of Message Sends

- Suppose one node sends a message to another node:

*send (data, count, datatype, destination)*

- There are two possible behaviours:
  - Blocking send
  - Non-blocking send

# Semantics of Blocking Send

- The send does not return until the data to be sent has “left” the application.
- This usually means that the message has been copied by the message passing system, or it has been delivered to the destination process.
- On return from the send() routine the *data* buffer can be reused without corrupting the message.

# Semantics of Non-Blocking Send

- Upon return from the `send()` routine the *data* buffer is volatile.
- This means that the data to be sent is not guaranteed to have left the application, and if the *data* buffer is changed the message may be corrupted. The idea here is for the `send()` routine to return as quickly as possible so the sending process can get on with other useful work.
- A subsequent call is used to check for completion of the send.

# Blocking Send Semantics

- On return from a blocking send you can change the message buffer, x, without changing the message



```
bsend(x,count, data, destination);  
x[0] = 2;
```

This is OK because on return from the bsend x has left the application.

- On return from a nonblocking send you should not change the message buffer, x, because this might change the message.




```
isend(x,count, data, destination);  
x[0] = 2;
```

This is not OK because on return from the isend you don't know if x has left the application.

# Blocking Send Semantics


- On return from a blocking send you can change the message buffer, x, without changing the message



```
bsend(x,count, data, destination);  
x[0] = 2;
```

This is OK because on return from the bsend x has left the application.

- On return from a nonblocking send you should not change the message buffer, x, because this might change the message.



```
isend(x,count, data, destination);  
:  
wait();  
x[0] = 2;
```

This is OK because after return from the wait x has left the application.

# Semantics of Message Receives

- Suppose one node receives a message from another node:

*receive (data, count, datatype, source)*

- There are two possible behaviours:
  - Blocking receive
  - Non-blocking receive

# Semantics of Blocking Receive

- The receive does not return until the data to be received has “entered” the application.
- This means that the message has been copied into the *data* buffer and can be used by the application on the receiving processor.



# Semantics of Non-Blocking Receive

- Upon return from the `receive()` routine the status of the *data* buffer is undetermined.
- This means that it is not guaranteed that the message has yet been received into the *data* buffer.
- We say that a receive has been *posted* for the message.
- The idea here is for the `receive()` routine to return as quickly as possible so the receiving process can get on with other useful work. A subsequent call is used to check for completion of the receive.

# Blocking Receive Semantics

- On return from a blocking receive you can use the message buffer, x, to access the data received.



```
brecv(x, maxcount, data, source);  
y = x[0];
```

This is OK because on return from the brecv x has entered the application.

- On return from a nonblocking receive you should not use the message buffer, x, because the data might not have arrived yet.




```
irecv(x, maxcount, data, source);  
y = x[0];
```

This is not OK because on return from the irecv you don't know if x has entered the application.

# Blocking Receive Semantics


- On return from a blocking receive you can use the message buffer, x, to access the data received.



```
brecv(x, maxcount, data, source);  
y = x[0];
```

This is OK because on return from the brecv x has entered the application.

- On return from a nonblocking receive you should not use the message buffer, x, because the data might not have arrived yet.



```
irecv(x, maxcount, data, source);  
:  
wait();  
y = x[0];
```

This is OK because after return from the wait x has entered the application.

# Message Passing Protocols

- Suppose one node sends a message and another receives it:

SOURCE: send (data, count, datatype, destination)

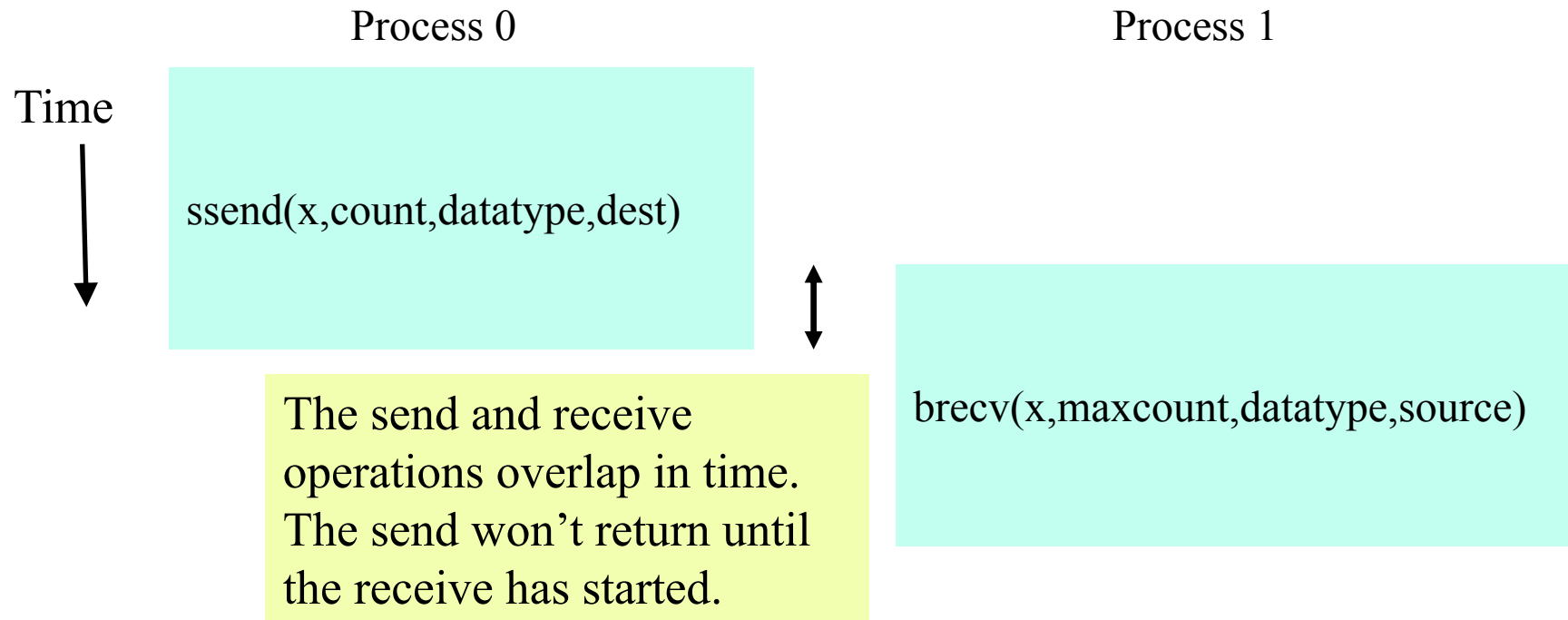
DEST: receive (data, count, datatype, source)

- Two important message passing protocols are
  - Synchronous send protocol
  - Asynchronous send protocol

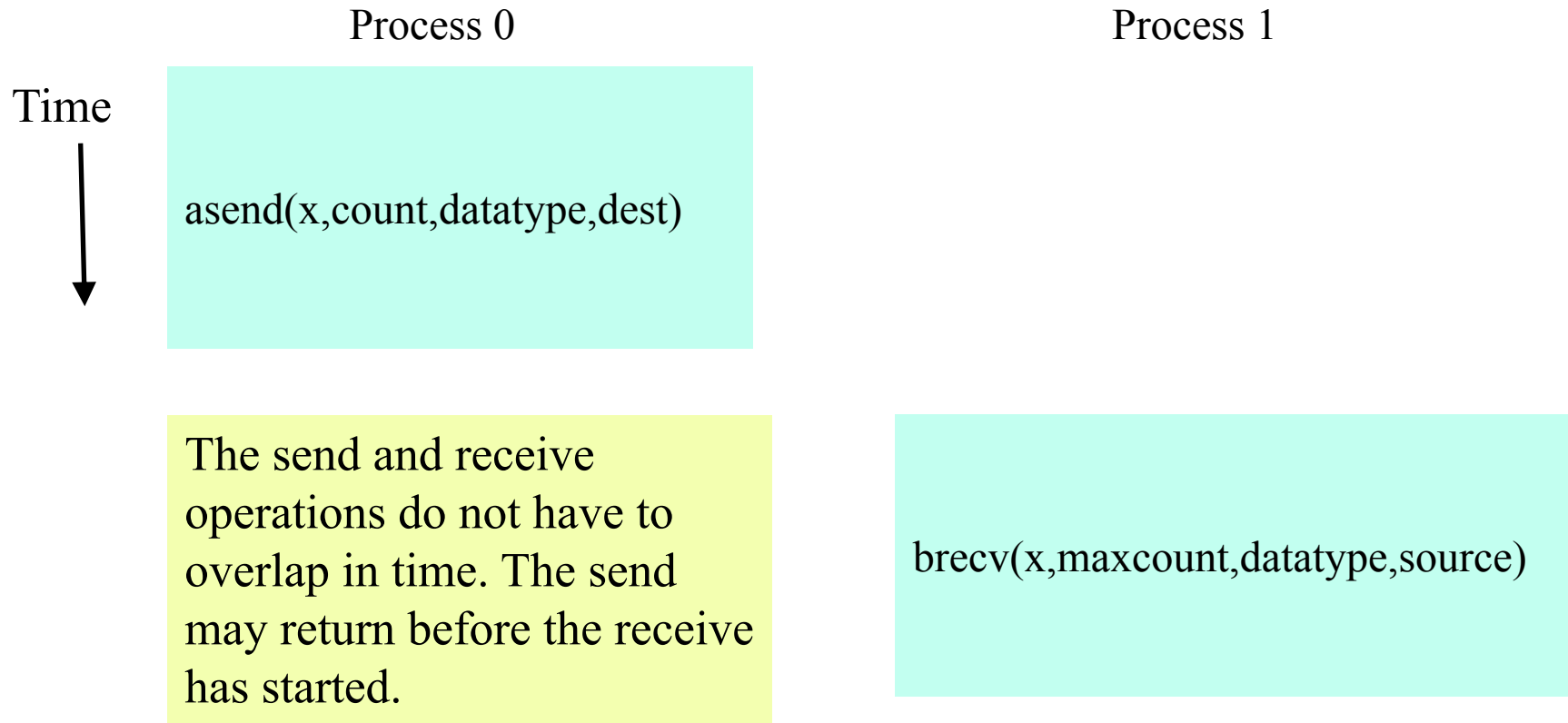
# Message Passing Protocols 2

- *Synchronous*: The send and receive routines overlap in time. The send does not return until the receive has started. This is also known as a *rendezvous* protocol.
- *Asynchronous*: The send and receive routines do not necessarily overlap in time. The send can return regardless of whether the receive has been initiated.

# Synchronous Send



# Asynchronous Send



# MPI Point-to-Point Communication

- MPI is a widely-used standard for message passing on distributed memory concurrent computers.
- Communication between pairs of processes is called point-to-point communication.
- Sender and receiver must specify the type of data communicated.
- Sender specifies the number of data items sent.
- Receiver specifies the maximum number that can be received.



# MPI API for C

- Must include the file **mpi.h**.
- This header file contains the function prototypes for the MPI library, and global constants, such as the default communicator **MPI\_COMM\_WORLD**.
- In addition to the predefined datatypes, such as **MPI\_INT** and **MPI\_FLOAT**, user-defined *derived datatypes* can also be used.

# Common Basic Datatypes

MPI Datatype	C Datatype
MPI_CHAR	signed char
MPI_SHORT	signed short
MPI_INT	signed int
MPI_LONG	signed long
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double

# MPI\_Send() and MPI\_Recv()

```
int MPI_Send(void *buf, int count,  
             MPI_Datatype datatype, int dest,  
             int tag, MPI_Comm comm);
```

```
int MPI_Recv(void *buf, int count,  
             MPI_Datatype datatype, int source,  
             int tag, MPI_Comm comm, MPI_Status  
             *status);
```

# Communicators

- A communicator defines which processes may be involved in the communication. In most elementary applications the MPI-supplied communicator **MPI\_COMM\_WORLD** is used.
- Two processes can communicate only if they use the same communicator.
- User-defined datatypes can be used, but mostly the standard MPI-supplied datatypes are used, such as **MPI\_INT** and **MPI\_FLOAT**.

# Process ranks

- When an MPI program is started the number of processes,  $N$ , is supplied to the program from the invoking environment. The number of processes in use can be determined from within the MPI program with the **MPI\_Comm\_size()** function.

```
int MPI_Comm_size(MPI_Comm comm, int *size)
```

- Each of the  $N$  processes is identified by a unique integer in the range 0 to  $N-1$ . This is called the process *rank*. A process can determine its rank with the **MPI\_Comm\_rank()** method.

```
int MPI_Comm_rank(MPI_Comm comm, int *rank)
```

# Message Tags

- The message tag can be used to distinguish between different types of message. The tag specified by the receiver must match that of the sender.
- In a **MPI\_Recv()** routine the message source and tag arguments can have the values **MPI\_ANY\_SOURCE** and **MPI\_ANY\_TAG**. These are called *wildcards* and indicate that the requirement for an exact match does not apply.

# Return Status Objects

- If the message source and/or tag are/is wildcarded, then the actual source and tag can be found from the **source** and **tag** fields of the status structure returned by **MPI\_Recv()**.
- The number of items received can be found using:

```
int MPI_Get_count(MPI_Status *status,  
                  MPI_Datatype datatype, int *count)
```

# Summary of Point-to-Point Communication

- Message selectivity on the receiver is by rank and message tag.
- Rank and tag are interpreted relative to the scope of the communication.
- The scope is specified by the communicator.
- Source rank and tag may be wildcarded on the receiver.
- Communicators must match on sender and receiver.



# Minimal MPI Program

```
#include <stdio.h>
#include <mpi.h>

int main (int argc, char *argv[])
{
    int rank, n, i, message, tag=111;
    MPI_Status status;

    MPI_Init (&argc, &argv);
    MPI_Comm_size (MPI_COMM_WORLD, &n);
    MPI_Comm_rank (MPI_COMM_WORLD, &rank);

    if (rank==0) { /* Process 0 will output data */
        printf ("Hello from process %3d\n", rank);
        for (i=1;i<n;i++) {
            MPI_Recv (&message,1,MPI_INT,i,MPI_ANY_TAG,MPI_COMM_WORLD,&status);
            printf ("Hello from process %3d\n", message);
        }
    }
    else MPI_Send (&rank, 1, MPI_INT, 0, tag, MPI_COMM_WORLD);

    MPI_Finalize ();
    return 0;
}
```

```
mpiexec -n 8 -f machines.txt hello
```

```
Hello from process 0
```

```
Hello from process 1
```

```
Hello from process 2
```

```
Hello from process 3
```

```
Hello from process 4
```

```
Hello from process 5
```

```
Hello from process 6
```

```
Hello from process 7
```

# Compiling and Running

```
mpicc -o exe exe.c
```

```
mpiexec -n 8 -f machines.txt exe
```

The file machines.txt contain the names of the machines to run the MPI processes on, e.g.,

```
helium.cs.cf.ac.uk  
lithium.cs.cf.ac.uk  
beryllium.cs.cf.ac.uk  
boron.cs.cf.ac.uk  
carbon.cs.cf.ac.uk  
nitrogen.cs.cf.ac.uk  
oxygen.cs.cf.ac.uk  
fluorine.cs.cf.ac.uk  
.....
```

# Another MPI Example

```
#include <stdio.h>
#include <mpi.h>

int main (int argc, char *argv[])
{
    int rank, n, i, message, tag=111;
    MPI_Status status;

    MPI_Init (&argc, &argv);
    MPI_Comm_size (MPI_COMM_WORLD, &n);
    MPI_Comm_rank (MPI_COMM_WORLD, &rank);

    if (rank==0) { /* Process 0 will output data */
        printf ("Hello from process %3d\n", rank);
        for (i=1;i<n;i++) {
            MPI_Recv (&message,1,MPI_INT,MPI_ANY_SOURCE,MPI_ANY_TAG,
                      MPI_COMM_WORLD,&status);
            printf ("Hello from process %3d\n", message);
        }
    }
    else MPI_Send (&rank, 1, MPI_INT, 0, tag, MPI_COMM_WORLD);

    MPI_Finalize ();
    return 0;
}
```

```
mpiexec -n 8 -f machines.txt hello2
```

```
Hello from process 0
```

```
Hello from process 1
```

```
Hello from process 2
```

```
Hello from process 3
```

```
Hello from process 5
```

```
Hello from process 7
```

```
Hello from process 4
```

```
Hello from process 6
```

# Notes on Examples

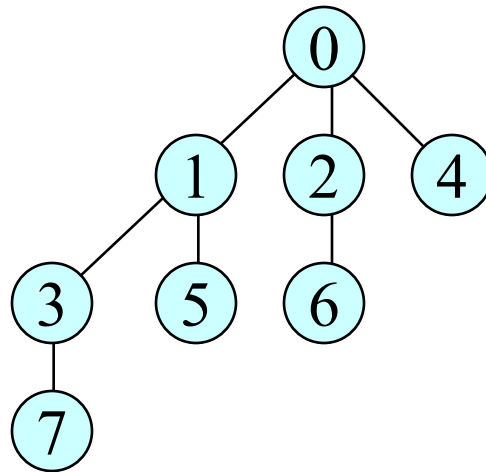
- All MPI calls must come between the calls to `MPI_Init()` and `MPI_Finalize()`.
- In the first example, the information from the processes is output in ascending order of processor number
- In the second example because the program does not specify the order in which process 0 receives messages, so output can be in any order.

# Collective Communication

- The send and receive style of communication between pairs of processors is known as *point-to-point communication*. This is distinct from *collective communication* in which several processors are involved in a coordinated communication task.
- Examples include:
  - Broadcasting data. One processor, known as the *root*, sends the same data to all processors.
  - Data reduction. Data from all processors is combined using a *reduction function* to produce a single result. The result may reside on a single processor or on all processors.

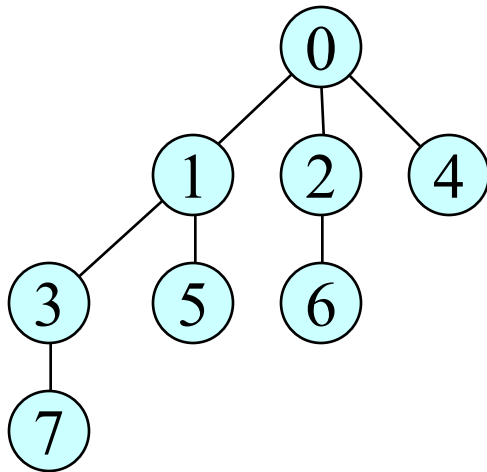
# Broadcast

- A common form of broadcast algorithm is based upon a *broadcast tree*.
- Suppose node 0 is the root of the broadcast. Consider the following tree.



# Broadcast Algorithm 1

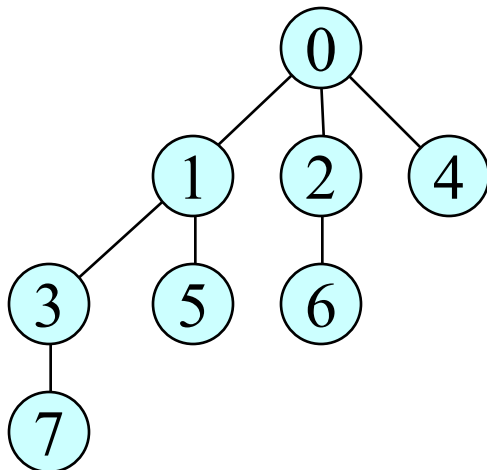
- Send on all links simultaneously
  - 1) Node 0 sends to nodes 1, 2, and 4
  - 2) Node 1 sends to nodes 3 and 5; node 2 sends to node 6
  - 3) Node 3 sends to node 7



On a hypercube this broadcast algorithm uses only physical links in the interconnect that directly connect nodes.

# Broadcast Algorithm 2

- Send on one link at a time:
  - 1) Node 0 sends to node 1.
  - 2) Node 0 sends to node 2, and node 1 sends to node 3.
  - 3) Node 0 sends to node 4, node 1 sends to node 5, node 2 sends to node 6, and node 3 sends to node 7

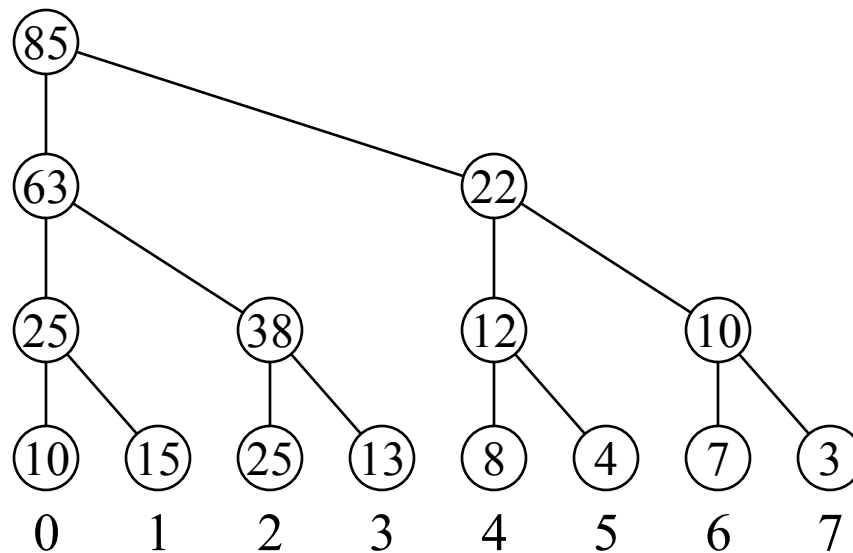


On a hypercube this broadcast algorithm uses only physical links in the interconnect that directly connect nodes.



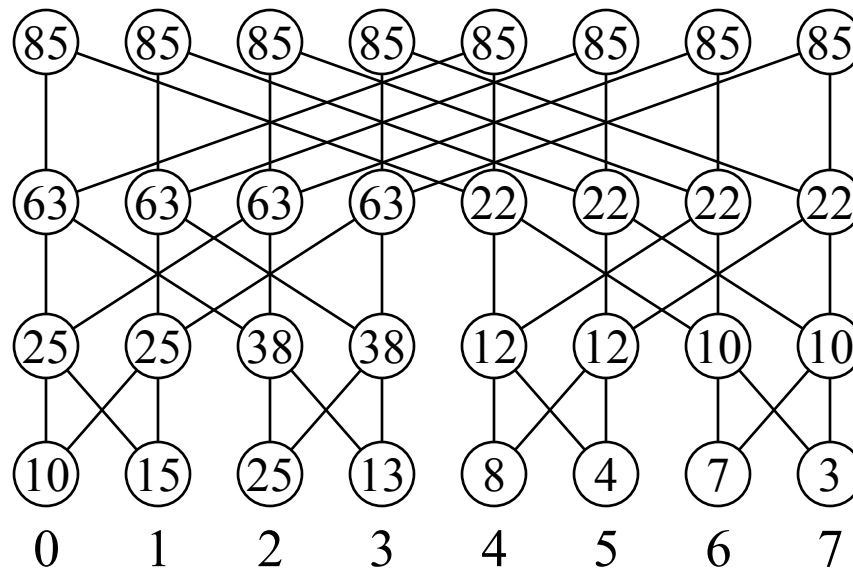
# Reduction

- Reduction can also be represented by a tree algorithm. For example, if we want to sum numbers on all nodes to one node:



# Reduction To All Nodes

- If we want to perform the sum so that all nodes end up with the result:



# Collective Routines

- Other forms of reduction include finding the maximum or minimum of a set of numbers over all processes.
- These reduction and broadcast algorithms are logarithmic in number of nodes, i.e., number of steps is approximately proportional to  $\log_2(n)$ .
- On hypercubes the logarithmic algorithms involve communication between only neighbouring processes.
- Other algorithms may be better for other network topologies.
- MPI provides routines for broadcasting and reduction.

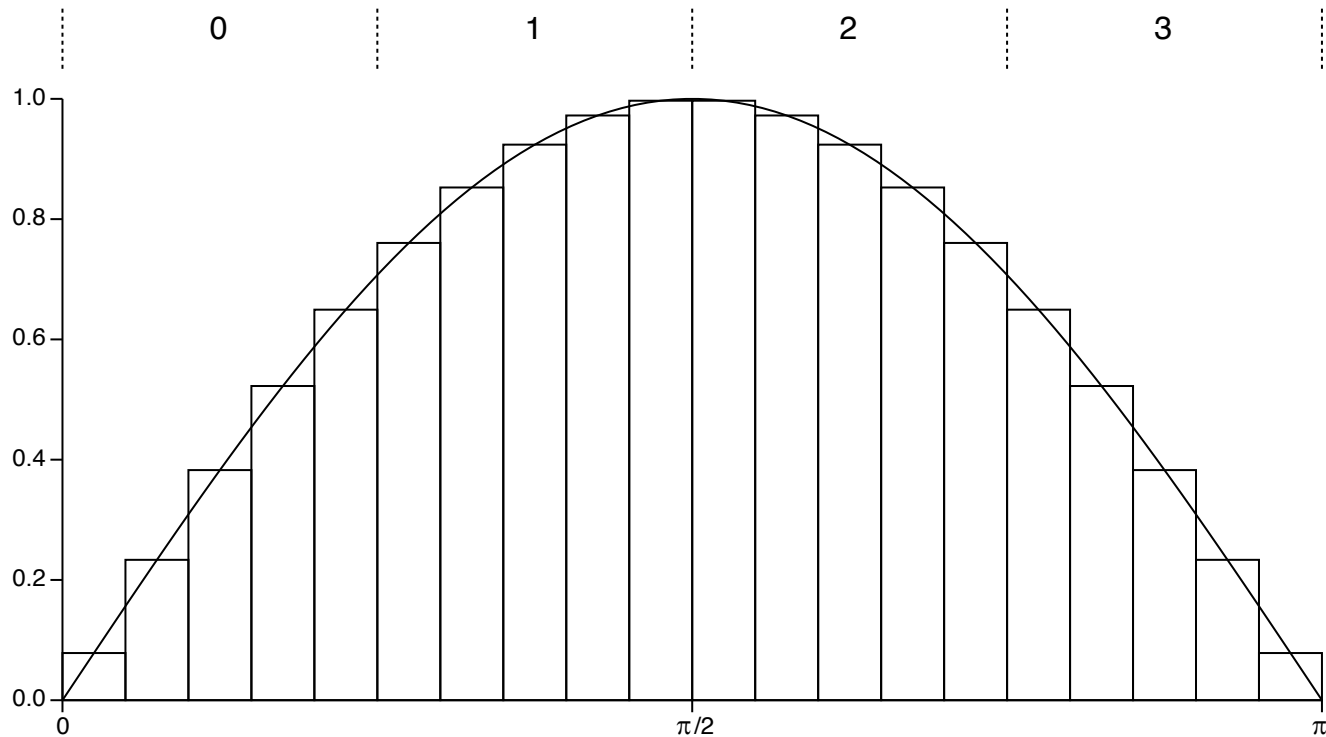
# MPI Integration Example

Want to find:

$$\int_0^{\pi} \sin(x) dx$$

- Initialisation
  - initialise MPI
  - communicate problem parameters
- Compute
  - each process computes its contribution
  - reduction operation sums process contributions
- Output
  - Process with rank 0 outputs the result
- Tidy Up
  - All processes call MPI.Finalize()

Approximate area under the curve by  
sum of the areas of thin rectangles



# Data Decomposition

- Let  $m$  be the number of rectangles and  $N$  be the number of processes.
- Write  $m = \alpha * N + \beta$ , where  $0 \leq \beta < N$ .
- Then  $\alpha = m/N$  and  $\beta = m \pmod{N}$
- The first  $\beta$  processes have  $\alpha+1$  rectangles, and the rest of the processes have  $\alpha$  rectangles.

# MPI Integration Code: Outline

```
#include <stdio.h>
#include <math.h>
#include <mpi.h>

#define PI 3.141592654
#define min(A,B) ((A)<(B) ? (A) : (B))

int main (int argc, char *argv[])
{
    int rank, nprocs, alpha, beta;
    int ick, i, nlocal, nbeg, nend, m;
    double deltax, psum ,sum, x;

    MPI_Init (&argc, &argv);
    MPI_Comm_size (MPI_COMM_WORLD, &nprocs);
    MPI_Comm_rank (MPI_COMM_WORLD, &rank);

    if (rank==0) {
        printf("\nGive the number of rectangles => \n");
        ick = scanf ("%d", &m);
    }
    MPI_Bcast (&m, 1, MPI_INT, 0, MPI_COMM_WORLD);
    :
    :
    MPI_Finalize ();
    return 0;
}
```



See next slide for what  
goes here

# MPI Integration Code: Computation

```
alpha = m/nprocs;
beta = m%nprocs;
nlocal = (rank<beta) ? alpha+1 : alpha;
nbeg = (rank<beta) ? nlocal*rank : nlocal*rank + beta;
nend = nbeg + nlocal - 1;

deltax = PI/m;
psum = 0.0;
for(i=nbeg;i<=nend;i++){
    x = (i+0.5)*deltax;
    psum += sin(x);
}

MPI_Reduce (&psum, &sum, 1, MPI_DOUBLE, MPI_SUM, 0,
            MPI_COMM_WORLD);

if (rank==0) printf ("\nThe integral is %f\n\n",sum*deltax);
```



# Application Topologies

- In many applications, processes are arranged with a particular topology, e.g., a regular grid.
- MPI supports general application topologies by a graph in which communicating processes are connected by an arc.
- MPI also provides explicit support for Cartesian grid topologies. Mostly this involves mapping between a process rank and a position in the topology.

# Cartesian Application Topologies

```
int MPI_Cart_create (MPI_Comm comm_old, int  
ndims, int *dims, int *period, int reorder,  
MPI_Comm *comm_cart)
```

- Periodicity in each grid direction may be specified.
- Inquiry routines transform between rank in group and location in topology
- For Cartesian topologies, row-major ordering is used for processes, i.e., (i,j) means row i, column j.

# Topological Inquiries

- Can get information about a Cartesian topology:

```
int MPI_Cart_get(MPI_Comm comm, int maxdims,  
int *dims, int *periods, int *coords)
```

This gives information about a Cartesian topology:

```
int *dims;    // number of processes in each dimension  
int *periods; // periodicity of each dimension  
int *coords;  // coordinates of calling process
```

# Mapping Between Rank and Position

- The rank of a process at a given location:

```
int MPI_Cart_rank(MPI_Comm comm, int  
*coords, int *rank)
```

- The location of a process of a given rank:

```
int MPI_Cart_coords(MPI_Comm comm,  
int rank, int maxdims, int *coords)
```

# Uses of Topologies

- Knowledge of application topology can be used to efficiently assign processes to processors.
- Cartesian grids can be divided into hyperplanes by removing specified dimensions.
- MPI provides support for shifting data along a specified dimension of a Cartesian grid.
- MPI provides support for performing collective communication operations along a specified grid direction.

# Topologies and Data Shifts

Consider the following two types of shift for a group of  $N$  processes:

- Circular shift by  $J$ . Data in process  $K$  is sent to process  $\text{mod}((J+K), N)$
- End-off shift by  $J$ . Data in process  $K$  is sent to process  $J+K$  if this is between 0 and  $N-1$ . Otherwise, no data are sent.

# Topologies and Data Shifts 2

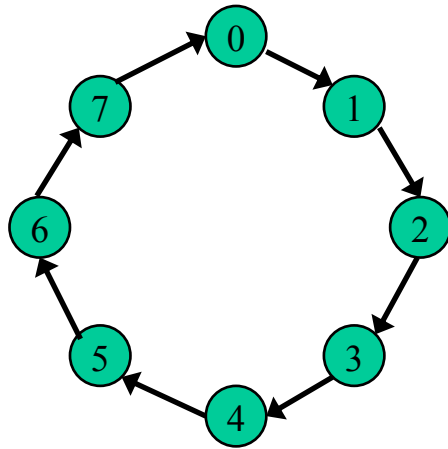
- Topological shifts are performed using  
`int MPI_Sendrecv(...)`
- The ranks of the processes that a process must send to and receive from when performing a shift on a topological group are returned by:

```
int MPI_Cart_shift(MPI_Comm comm, int direction,  
int disp, int *rank_source, int *rank_dest)
```

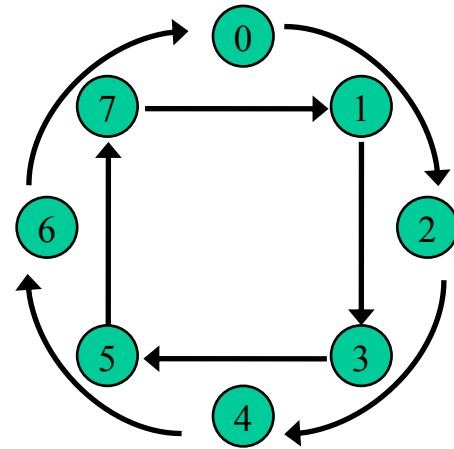
# Shifts

Circular:

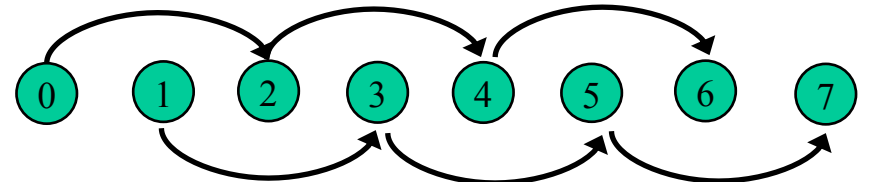
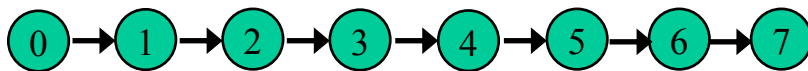
$N = 8, J = 1$



$N = 8, J = 2$



End-off:





# Send/Receive Operations

- In many applications, processes send to one process while receiving from another.
- Deadlock may arise if care is not taken.
- MPI provides routines for such send/receive operations.
- For distinct send/receive buffers:

`int MPI_Sendrecv(...)`

- For identical send/receive buffers:

`int MPI_Sendrecv_replace(...)`