Syllabus for CM0224

Parallel Processing I

W9W10, 207T

1. *Lecture 1.* Introduction; motivation; types of parallelism; data decomposition; uses of parallel computers; classification of machines.
2. *Lecture 2.* SPMD programs; memory models; shared and distributed memory.
3. *Lecture 3.* Example of summing numbers; interconnection networks; network metrics; Gray code mappings.
4. *Lecture 4.* Classification of parallel algorithms; Speedup and efficiency.
5. *Lecture 5.* Scalable algorithms; Amdahl's law; sending and receiving messages; programming with MPI.
6. *Lecture 6.* Collective communication; integration example; regular computations and the vibrating string example.
7. *Lecture 7.* Two-dimensional problems and the Laplace equation example.
8. *Lecture 8.* Dynamic communication and the molecular dynamics example.
9. *Lecture 9.* Irregular computations; the WaTor simulation. Load balancing strategies.
10. *Lecture 10.* Message passing libraries; introduction to PVM.
11. *Lecture 11.* Spare lecture.

- Starting from week 6 there will be a weekly lab at which students will gain practical experience in using MPI on a network of workstations.
- Starting from week 6 there will be a weekly tutorial session.

Books and References

- "Solving Problems on Concurrent Processors, Volume 1," Fox, Johnson, Lyzenga, Otto, Salmon, and Walker, published by Prentice-Hall, 1988. ISBN 0-13-823022-6.
- "Parallel Computing: Theory and Practice," M. Quinn, published by McGraw-Hill, 1994.
- "Using MPI," Gropp, Lusk, and Skjellum, published by MIT Press, 1994.
- "MPI: the Complete Reference," Snir, Otto, Huss-Lederman, Walker and Dongarra, MIT Press, 1996.
- World-wide web site for course

  `http://www.cs.cf.ac.uk/User/David.W.Walker/PPI/`
- World-wide web site for course given in 1995

  `http://www.cs.cf.ac.uk/Parallel/Year2/`

What is Parallelism?

Parallelism refers to the simultaneous occurence of events on a computer.

An event typically means one of the following

- An arithmetical operation
- A logical operation
- Accessing memory
- Performing input or output (I/O)

Types of Parallelism

Parallelism can be examined at several levels.

- *Job level*: several independent jobs simultaneously run on the same computer system.
- *Program level*: several tasks are performed simultaneously to solve a single common problem.
- *Instruction level*: the processing of an instruction, such as adding two numbers, can be divided into sub-instructions. If several similar instructions are to be performed their sub-instructions may be overlapped using a technique called *pipelining*.
- *Bit level*: when the bits in a word are handled one after the other this is called a *bit-serial* operation. If the bits are acted on in parallel the operation is *bit-parallel*.

In this parallel processing course we shall be mostly concerned with parallelism at the program level.

Concurrent processing is the same as parallel processing.

Job Level Parallelism

This is parallelism between different independent jobs, or phases of a jobs, on a computer.

Example

Consider a computer with 4 processors. It runs jobs that are classified as small (S), medium (M), and large (L). Small jobs require 1 processor, medium jobs 2 processors, and large jobs 4 processors. Each job takes one unit of time. Initially there is a queue of jobs:

S M L S S M L L S M M

Show how these jobs would be scheduled to run if the queue is treated in order. Also give a better schedule.

Scheduling Example

Time	Jobs running	Utilisation
1	S,M	75%
2	L	100%
3	S,S,M	100%
4	L	100%
5	L	100%
6	S,M	75%
7	M	50%

- Average utilisation is 83.3%.
- Time to complete all jobs is 7 time units.

A Better Schedule

A better schedule would allow jobs to be taken out of order to give higher utilisation.

S M L S S M L L S M M

Allow jobs to "float" to the front to the queue to maintain high utilisation.

Time	Jobs running	Utilisation
1	S,M,S	100%
2	L	100%
3	S,M,S	100%
4	L	100%
5	L	100%
6	M,M	100%

- Average utilisation is 100%.
- Time to complete all jobs is 6 time units.

Actual situation is more complex as jobs may run for differing lengths of time. Real job scheduler must balance high utilisation with fairness (otherwise large jobs may never run).

Parallelism Between Job Phases

Parallelism also arises when different independent jobs running on a machine have several phases, e.g., computation, writing to a graphics buffer, I/O to disk or tape, and system calls.

- Suppose a job is executing and needs to perform I/O before it can progress further. I/O is usually expensive compared with computation, so the job currently running is suspended, and another is started. The original job resumes after the I/O operation has completed.
- This requires special hardware: I/O channels or special I/O processor.
- The *operating system* controls how different jobs are scheduled and share resources.

Program Level Parallelism

This is parallelism between different parts of the same job.

Example

A robot has been programmed to look for electrical sockets when it runs low on power. When it finds one it goes over to it and plugs itself in to recharge. Three subsystems are involved in this — the vision, manipulation, and motion subsystems. Each subsystem is controlled by a different processor, and they act in parallel as the robot does different things.

Task	Vision	Manipulation	Motion
1. Looking for electrical socket	×		×
2. Going to electrical socket	×		×
3. Plugging into electrical socket	×	×	

- The subsystems are fairly independent, with the vision subsystem guiding the others.
- There may also be a central "brain" processor.

This is an example of *task parallelism* in which different tasks are performed concurrently to achieve a common goal.

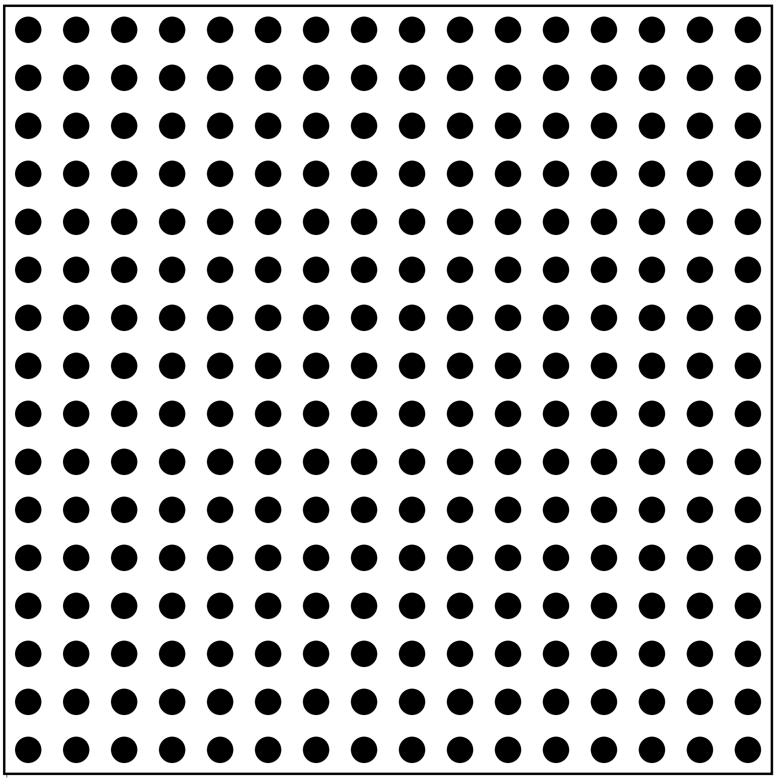
Domain Decomposition

- A common form of program-level parallelism arises from the division of the data to be programmed into subsets.
- This division is called *domain decomposition*.
- Parallelism that arises through domain decomposition is called *data parallelism*.
- The data subsets are assigned to different computational processes. This is called *data distribution*.
- Processes may be assigned to hardware processors by the program or by the runtime system. There may be more than one process on each processor.

Data Parallelism

Example

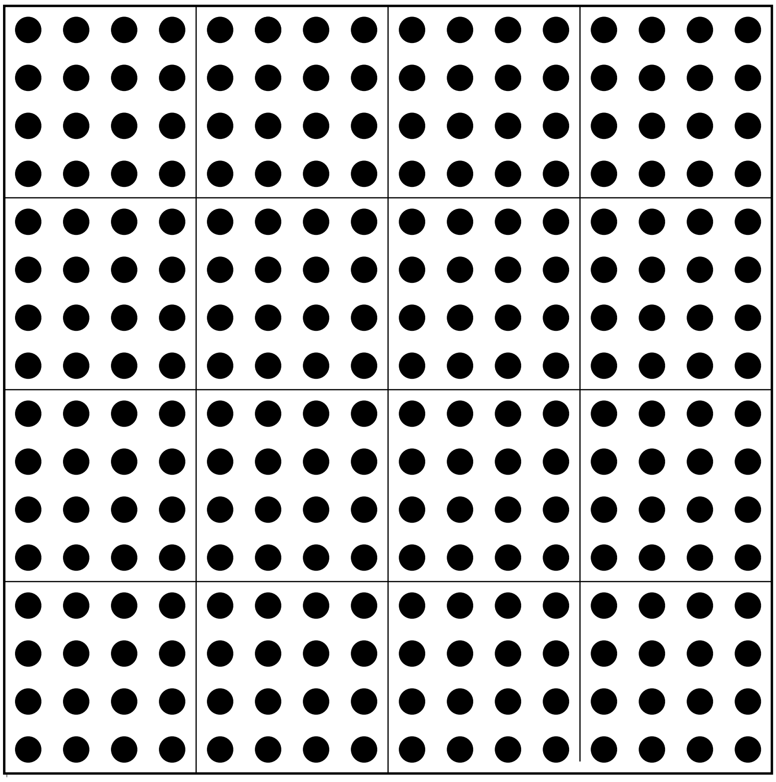
Consider an image digitised as a square array of pixels which we want to process by replacing each pixel value by the average of its neighbours.



The *domain* of the problem is the two-dimensional pixel array.

Domain Decomposition

Suppose we decompose the problem into 16 subdomains



We then distribute the data by assigning each subdomain to a process.

Points to note:

- The pixel array is a *regular* domain because the geometry is simple.
- This is a homogeneous problem because each pixel requires the same amount of computation (almost — which pixels are different?).

Why Use Parallelism?

1. Better utilisation of resources. Want to keep hardware busy.
2. Want to run programs faster by spreading work over several processors. Ways to measure speed:
   - Floating point operations per second. 1 Mflop/s is one million floating point operations per second.
     - High performance workstation $\approx$ few 100 Mflop/s
     - Current best supercomputer $\approx$ 1 Tflop/s
   - Transactions per second. 1 Tps is one transaction per second.
   - Instructions per second. 1 Mips is one million instructions per second.
3. Want more memory to solve bigger or more complex problems.
   - Typical workstations have 64 Mbytes of RAM, expandible to 512 Mbytes. Can fit an 8192×8192 array into 512 Mbytes of memory.
   - An Intel Paragon parallel computer has 1024 nodes with 64 Mbytes each for a total of 64 Gbytes. Can fit a $92,000 \times 92,000$ array into memory.

Parallelism and Supercomputing

- Parallelism is exploited on a variety of high performance computers, in particular *massively parallel computers* (MPPs).
- MPPs and high-performance vector computers are termed *supercomputers*
- Currently supercomputers have peak performance in the range of a few 10's to a few 100's of Gflop/s, and memory of 10 to 100 Gbytes. They cost about 20 million pounds.
- Supercomputers are leading to a new methodology in science called *computational science* joinng theoretical and experimental approaches.

Uses of Parallelism

Here are some important uses of supercomputers.

- **Weather forecasting**. Currently forecasts are usually accurate up to about 5 days. This should be extended to 8 to 10 days over the next few years. Researchers would like to better model local nonlinear phenomena such as thunderstorms and tornadoes.
- **Climate modeling**. Studies of long-range behaviour of global climate. This is relevant to investigating global warming.
- **Engineering**. Simulation of car crashes to aid in design of cars. Design of aircraft in "numerical wind tunnels."
- **Material science**. Understanding high temperature superconductors. Simulation of semiconductor devices. Design of lightweight, strong materials for construction.
- **Drug design**. Prediction of effectiveness of drug by simulation. Need to know configuration and properties of large molecules.

More Uses of Parallelism

- **Plasma physics**. Investigation of plasma fusion devices such as tokamaks as future source of cheap energy.
- **Economics**. Economic projections used to guide decision-making. Prediction of stock market behaviour.
- **Defense**. Tracking of multiple missiles. Event-driven battlefield simulations. Code cracking.
- **Astophysics** Modeling internal structure of stars. Simulating supernova. Modeling the structure of the universe.

Classification of Parallel Machines

To classify parallel machines we must first develop a model of computation. The approach we follow is due to Flynn (1966).

Any computer, whether sequential or parallel, operates by executing instructions on data.

> a stream of **instructions** (the algorithm) tells the computer what to do.
>
> a stream of **data** (the input) is affected by these instructions.

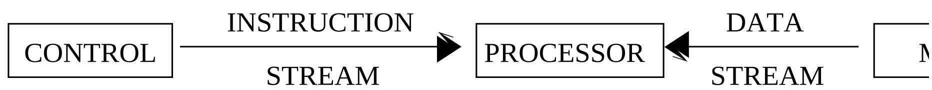
Depending on whether there is one or several of these streams we have 4 classes of computers.

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

SISD Computers

This is the standard sequential computer.

A single processing unit receives a single stream of instructions that operate on a single stream of data.

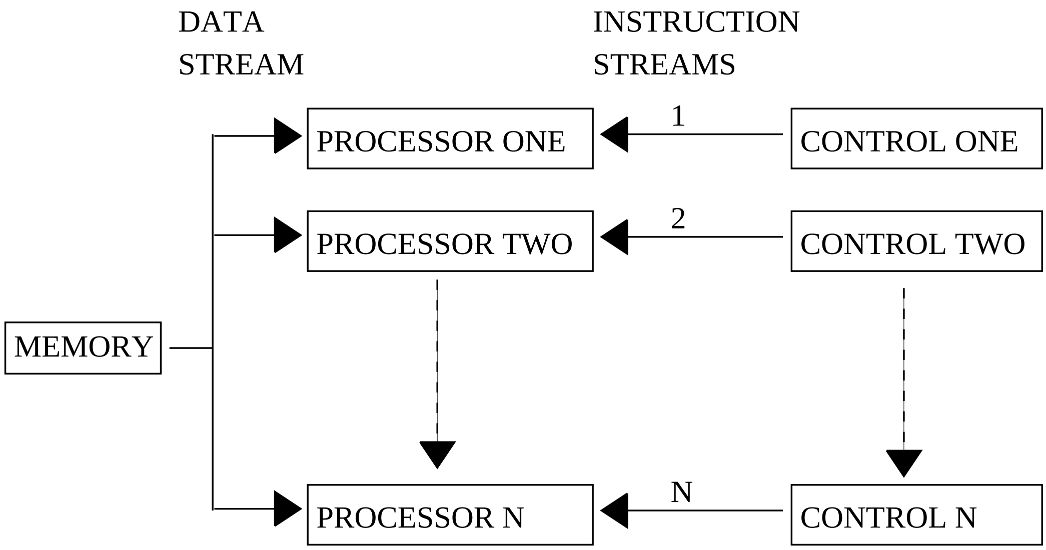


Example
To compute the sum of N numbers $a_1, a_2, \ldots, a_N$ the processor needs to gain access to memory N consecutive times. Also $N - 1$ additions are executed in sequence. Therefore the computation takes O(N) operations

Algorithms for SISD computers do not contain any parallelism since there is only one processor.

MISD Computers

N processors, each with its own control unit, share a common memory.



- There are N streams of instructions (algorithms/programs) and one stream of data. Parallelism is achieved by letting the processors do different things at the same time to different data.
- MISD machines are useful in computations where the same input is to be subjected to several different operations.

MISD Example

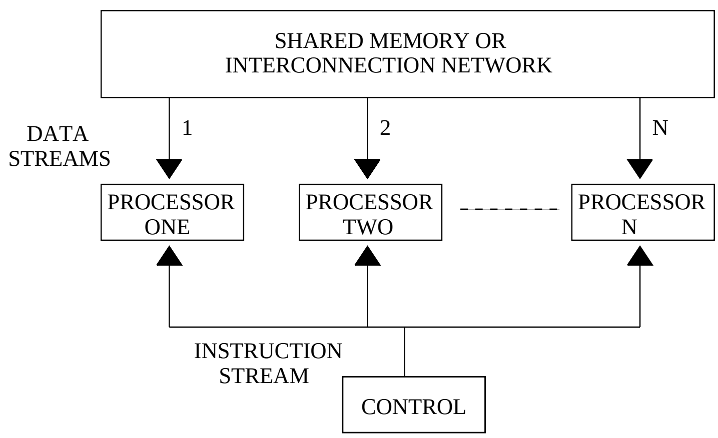
Example

Checking whether a number Z is prime. A simple solution is to try all possible divisions of Z. Assume the number of processors is $N = Z - 2$. All processors take Z as input and each tries to divide it by its associated divisor. So it is possible in one step to check if Z is prime. More realistically, if $N < Z - 2$ then a subset of divisors is assigned to each processor.

For most applications MISD computers are very awkward to use and no commercial machines exist with this design.

SIMD Computers

- All N identical processors operate under the control of a single instruction stream issued by a central control unit.
- There are N data streams, one per processor, so different data can be used in each processor.



- The processors operate *synchronously* and a global clock is used to ensure lockstep operation, i.e., at each step (global clock tick) all processors execute the same instruction, each on a different datum.
- Array processors such as the ICL DAP, Connection Machine CM-200, and MasPar are SIMD computers.
- SIMD machines are particularly useful at exploiting data parallelism to solve problems having a regular structure in which the same instructions are applied to subsets of data.

SIMD Example

Problem: Add two 2×2 matrices on 4 processors

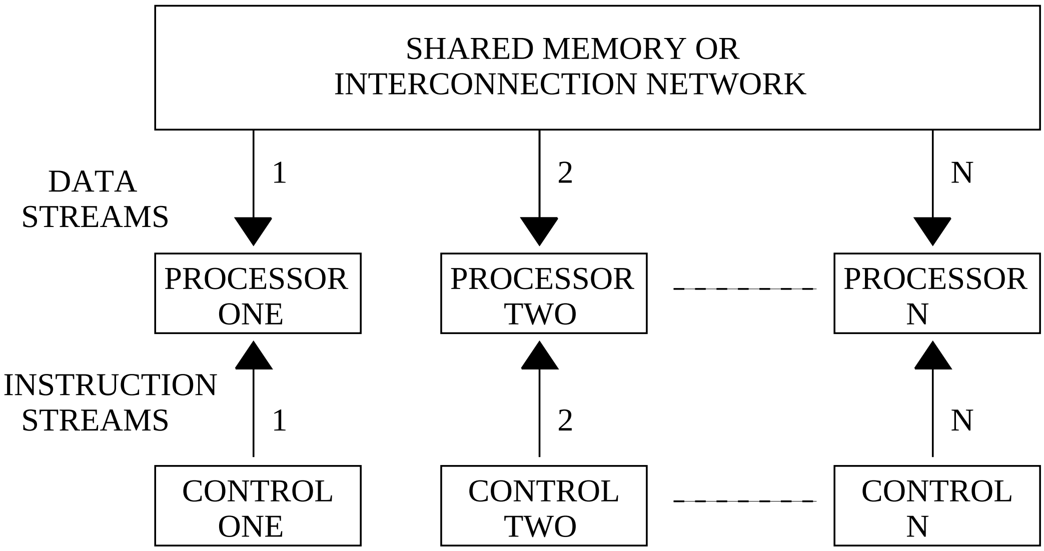
$$\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} + \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix} = \begin{pmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{pmatrix}$$

The same instruction is issued to all 4 processors (add two numbers), and all processors execute the instructions simultaneously. It takes one step to add the matrices, compared with 4 steps on a SISD machine.

- In this example the instruction is simple, but in general it could be more complex such as merging two lists of numbers.
- The datum may be simple (one number) or complex (several numbers).
- Sometimes it may be necessary to have only a subset of the processors execute an instruction, i.e., only some data needs to be operated on for that instruction. This information can be encoded in the instruction itself indicating whether
  1. the processor is active (execute the instruction)
  2. the processor is inactive (wait for the next instruction)

MIMD Computers

This is the most general and most powerful of our classification. We have N processors, N streams of instructions, and N streams of data.



- The processors can operate *asynchronously*, i.e., they can do different things on different data at the same time.
- As with SIMD computers, communication of data or results between processors can be via shared memory or an interconnection network.

Notes on SIMD and MIMD

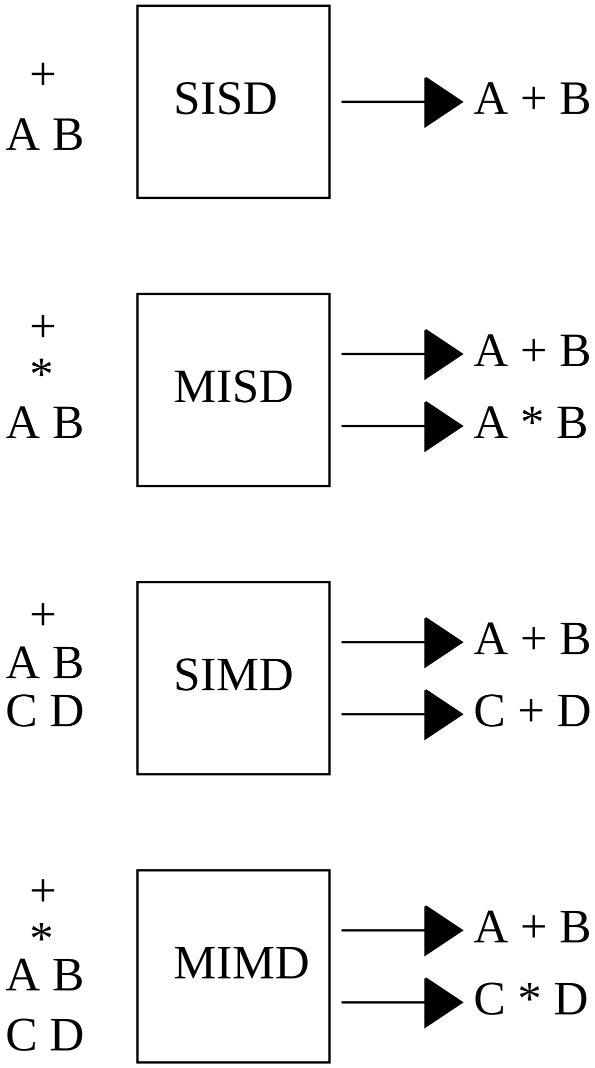
- In most problems to be solved on SIMD and MIMD computers it is useful for the processors to be able to communicate with each other to exchange data or results. This can be done in two ways

  1. by using a shared memory and shared variables, or
  2. using an interconnection network and message passing (distributed memory)

- MIMD computers with shared memory are known as *multiprocessors*, Examples are produced by ENCORE and Silicon Graphics.

- MIMD computers with an interconnection network are known as *multicomputers*. Examples are the Intel Paragon, the nCUBE-2, and transputer networks.

Potential of the 4 Classes



Single Program Multiple Data

- An MIMD computer is said to be running in SPMD mode if the same program is executing on each process
- SPMD is not a hardware paradigm, so it is not included in our 4 classifications. It is a software paradigm for MIMD machines.
- Each processor executes an SPMD program on different data so it is possible that different branches are taken, leading to *asynchronous parallelism.* the processors no longer do the same thing (or nothing) in lockstep as they do on an SIMD machine. They execute different instructions within the same program.
- Suppose `x` is 0 on processor 1, and 1 on processor 2. Consider

```
IF X = 0
THEN S1
ELSE S2
```

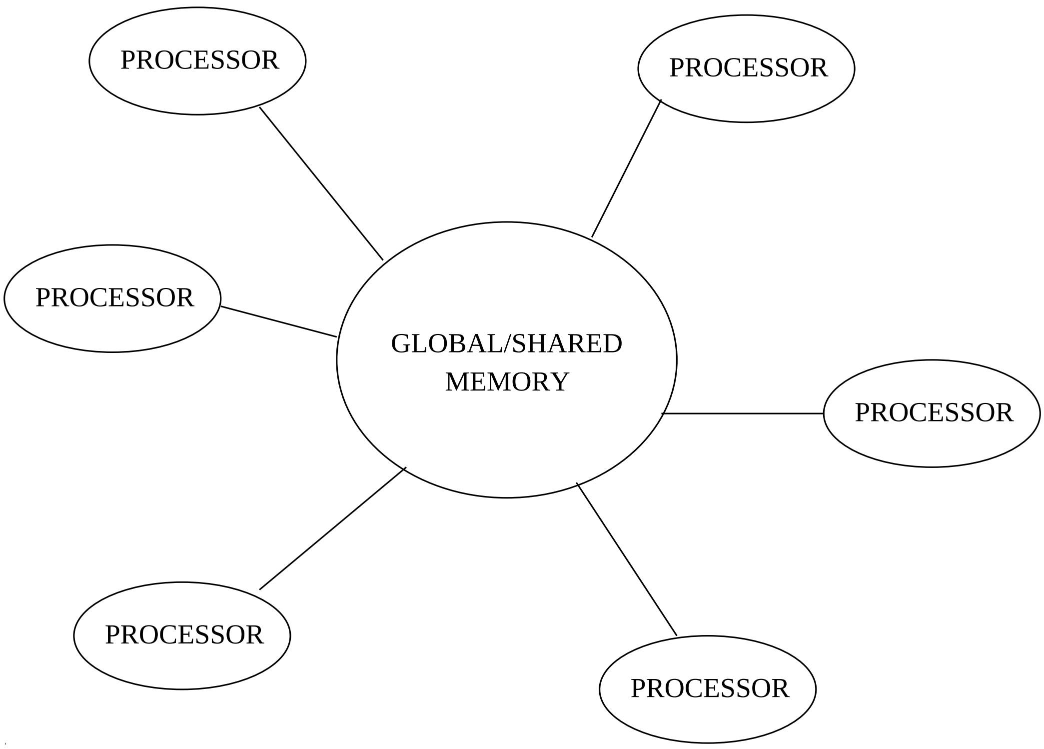
  Then processor 1 executes S1 at the same time that processor 2 executes S2. This could not happen on an SIMD machine.

Interprocessor Communication

Usually a parallel program needs to have some means of sharing data and results processed by different processors. There are two main ways of doing this

1. Shared Memory
2. Message passing

Shared memory consists of a global address space. All processors can read from and write into this global address space.

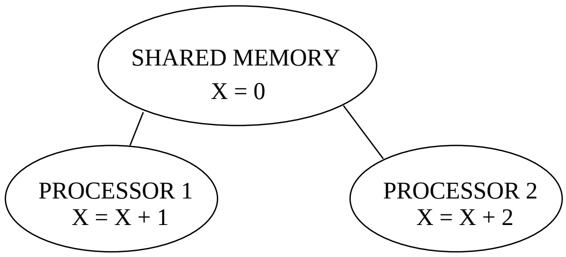


Shared Memory Conflicts

The shared memory approach is simple but can lead to problems when processors simultaneously access the same location in memory.

Example

Suppose the shared memory initially holds a variable `x` with value 0. Processor 1 adds 1 to `x` and processor 2 adds 2 to `x`. What is the final value of `x`?



The following outcomes are possible

1. If P1 executes and completes `x=x+1` before P2 reads the value of `x` from memory then `x` is 3. Similarly, if P2 executes and completes `x=x+2` before P1 reads the value of `x` from memory then `x` is 3.

2. If P1 or P2 reads `x` from memory before the other has written back its result, then the final value of `x` depends on which finishes last.

   - if P1 finishes last the value of `x` is 1
   - if P2 finishes last the value of `x` is 2

This is an example of *non-determinism* or *non-determinancy*.

Non-Determinancy

Non-determinancy is caused by *race conditions*.

- A race condition occurs when two statements in concurrent tasks access the same memory location, at least one of which is a write, and there is no guaranteed execution ordering between accesses.
- The problem of non-determinancy can be solved by synchronising the use of shared data. That is if `x=x+1` and `x=x+2` were mutually exclusive then the final value of `x` would always be 3.
- Portions of a parallel program that require synchronisation to avoid non-determinancy are called *critical sections*.
- In shared memory programs *locks* can be used to give mutually exclusive access.

```
Processor 1                  Processor 2
   LOCK (x)                     LOCK (x)
     x = x + 1                    x = x + 2
   UNLOCK (x)                   UNLOCK (x)
```

Classifying Shared Memory Computers

Shared memory computers can be classified as follows depending on whether two or more processors can gain access to the same memory simultaneously.

1. **Exclusive Read, Exclusive Write (EREW)**

   Access to memory locations is exclusive, i.e., no 2 processors are allowed to simultaneously read from or write into the same location.

2. **Concurrent Read, Exclusive Write (CREW)**

   Multiple processors are allowed to read from the same location, but write is still exclusive, i.e., no 2 processors are allowed to write into the same location simultaneously.

3. **Exclusive Read, Concurrent Write (ERCW)**

   Multiple processors are allowed to write into the same location, but read access remains exclusive.

4. **Concurrent Read, Concurrent Write (CRCW)**

   Both multiple read and write privileges are allowed.

Notes on Shared Memory

- Allowing concurrent read access to the same address should pose no problems in the sense that such an operation is well-defined. Conceptually each processor makes a copy of the contents of the memory location and stores it in its own register.
- Problems arise with concurrent write access, because if several processors write simultaneously to the same address, which should "succeed?" There are several ways of deterministically specifying the contents of a memory location after a concurrent write
  - assign proirities to processors and store value from processor with highest priority.
  - all the processors are allowed to write, provided all the values they are attempting to store are the same.
  - the max, min, sum, or average of the values is stored (for numeric data).

More Notes on Shared Memory

- SIMD machines usually have 1000's of very simple processors. Shared memory SIMD machines are unrealistic because of the cost and difficulty in arranging for efficient access to shared memory for so many processors. There are no commercial shared memory SIMD machines.
- MIMD machines use more powerful processors and shared memory machines exist for small numbers of processors (up to about 64).

Example for Shared Memory

To show how the 4 subclasses of shared memory machines behave, consider the following example.

Problem: We have N processors to search a list `S =` {`L1, L2, Lm`} for the index of a given element `x`. Assume `x` may appear several times, and any index will do. $1 < N \leq m$.

Algorithm

```
procedure SM_search (S, x, k)
  STEP 1: for i=1 to N do in parallel
             read x
          end for
  STEP 2: for i=1 to N do in parallel
             Si = {L((i-1)m/N+1), ...,L(im/N)}
             perform sequential search on sublist Si
                 (return Ki=-1 if not in list,
                  otherwise index)
          end for
  STEP 3: for i=1 to N do in parallel
             if Ki > 0 then k=Ki end if
          end for
end procedure
```

Time Complexity of Algorithm

If the sequential search step takes $O(m/N)$ time, what is the time complexity for each of the 4 subclasses of shared memory computer?

- **EREW**

  Step 1 takes $O(N)$ (N reads, one at a time).

  Step 2 takes $O(m/N)$ time.

  Step 3 takes $O(N)$ time.

  Total time is $O(N) + O(m/N)$.

- **ERCW**

  Step 1 takes $O(N)$ time.

  Step 2 takes $O(m/N)$ time.

  Step 3 takes constant time.

  Total time is $O(N) + O(m/N)$.

Time Complexity of Algorithm

- **CREW**

  Step 1 takes constant time.

  Step 2 takes $O(m/N)$ time.

  Step 3 takes $O(N)$ time

  Total time is $O(N) + O(m/N)$.

  CRCW

  Step 1 takes constant time.

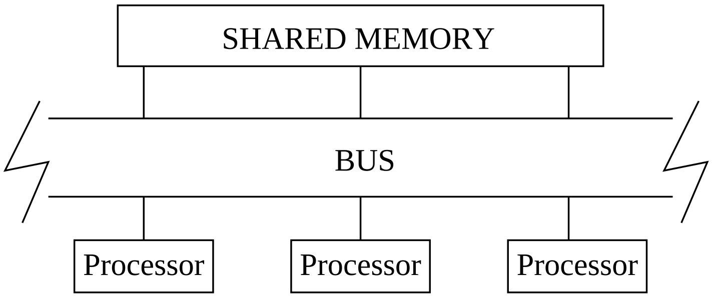
  Step 2 takes $O(m/N)$ time.

  Step 3 takes constant time.

  Total time is $O(m/N)$.

Limitations on Shared Memory

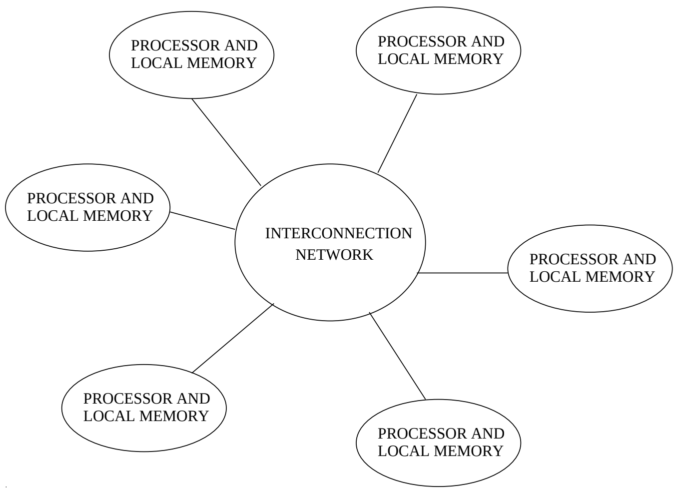
Shared memory computers such as the Sequent Symmetry are often implemented by incorporating a fast bus to connect processors to memory.



However, because the bus has a finite bandwidth, i.e., it can carry only a certain maximum amount of data at any one time, then as the number of processors increase the *contention* for the bus becomes a problem. So it is feasible to build shared memory machines with up to only about 30 processors.

Interconnection Network and Message Passing

In this case each processor has its own private (local) memory and there is no global, shared memory. The processors need to be connected in some way to allow them to communicate data.



Example

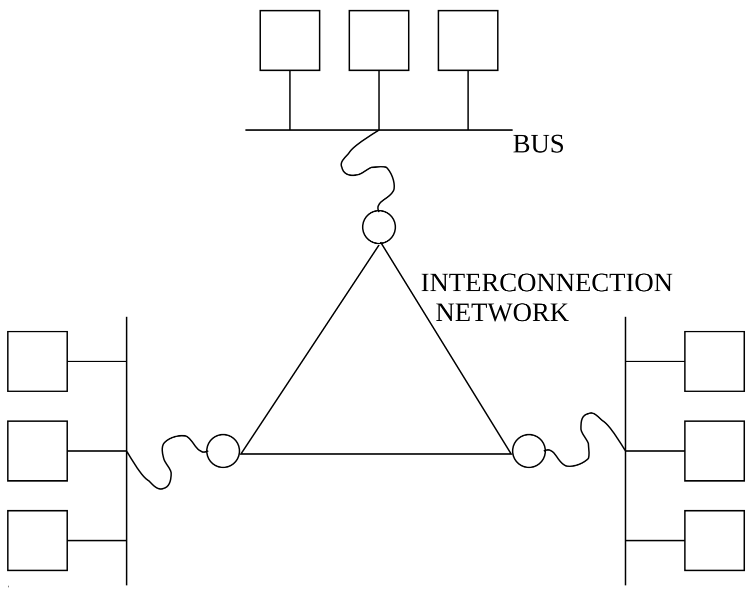
If a processor requires data contained on a different processor then it must be explicitly passed by using communication instructions, e.g., send and receive

```
P1                  P2
receive (x, P2)     send (x, P1)
```

The value `x` is explicitly passed from P2 to P1. This is known as *message passing*.

Hybrid Computers

- In addition to the cases of shared memory and distributed memory there are possibilities for hybrid designs that incorporate features of both.
- Clusters of processors are connected via a high speed bus for communication within a cluster, and communicate between clusters via an interconnection network.



Comparison of Shared and Distributed Memory

Distributed memory	Shared Memory
Large number of processors (100s to 1000s)	Modest number of processors (10's to 100)
High peak performance	Modest performance
Unlimited expansion	Limited expansion
Difficult to fully utilise	Relatively easy to fully utilise
Revolutionary parallel programming	Evolutionary parallel programming

Summing m Numbers

Example: summing m numbers
On a sequential computer we have,

```
sum = a[0]
for (i=1;i<m;i++)
  sum = sum + a[i]
end for
```

Would expect running time be be roughly proportional to m. We say that the running time is $\Theta(m)$.

What if we have N processors, with each calculating the m/N numbers assigned to it. We must add these partial sums together to get the total sum.

Summing Using Shared Memory

The m numbers, and the global sum, are held in global shared memory

```
global_sum = 0
for (each processor)
   local_sum = 0
   calculate local sum of m/N numbers
   LOCK
     global_sum = global_sum + local_sum
   UNLOCK
end for
```

Since `global_sum` is a shared variable each processor must have mutually exclusive access to it – otherwise the final answer may be incorrect.

The running time (or *algorithm time complexity*) is

$$\Theta(m/N) + \Theta(N)$$

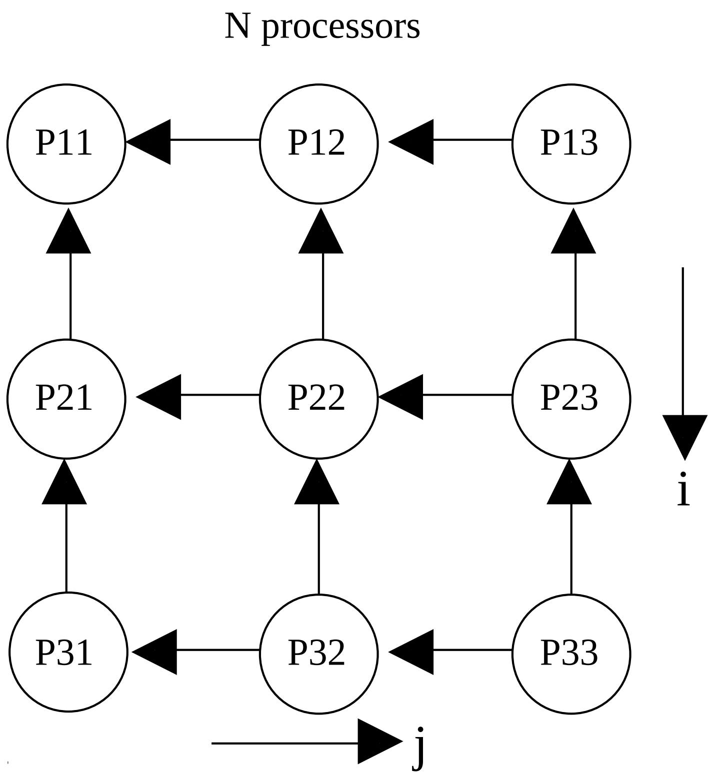
where

m/N comes from finding the local sums in parallel

N comes from adding N numbers in sequence

Summing Using Distributed Memory

Suppose we have a square mesh of 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 P11.

Distributed Memory Pseudocode

```
FORALL processors
  find local_sum of m/N numbers
  for (j=sqrt(N);j>1;j--)
    IFPROC in column j
      send local_sum to neighbor in column j-1
    END IFPROC
    IFPROC in column j-1
      receive new_local_sum from neighbor in column j
      local_sum = local_sum + new_local_sum
    END IFPROC
  end for
  IFPROC in first column
    for (i=sqrt(N);i>1;i--)
      IFPROC in row i
        send local_sum to neighbor above
      END IFPROC
      IFPROC in row i-1
        receive new_local_sum from neighbor below
        local_sum = local_sum + new_local_sum
      END IFPROC
    end for
    IFPROC in first row
      global_sum = local_sum
    END IFPROC
  END IFPROC
END FORALL
```

Summing Example

There are $\sqrt{N}-1$ additions and $\sqrt{N}-1$ communications in each direction, so the total time complexity is

$$\Theta(m/N) + \Theta(\sqrt{N}) + C$$

where C is the time spent communicating.

Initially

10	10	10
10	10	10
10	10	10

j = 3

10	20	–
10	20	–
10	20	–

j = 2

30	–	–
30	–	–
30	–	–

i = 3

30	–	–
60	–	–
–	–	–

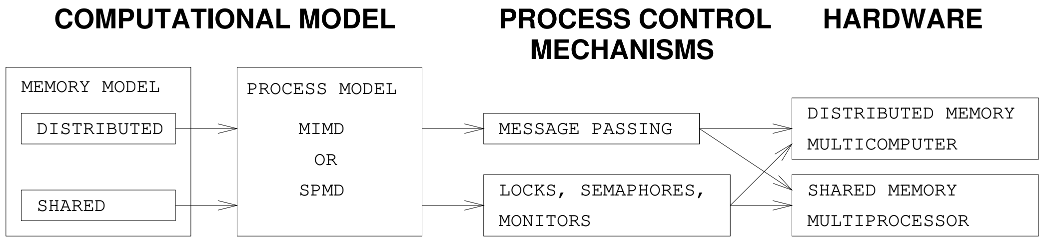
i = 2

90	–	–
–	–	–
–	–	–

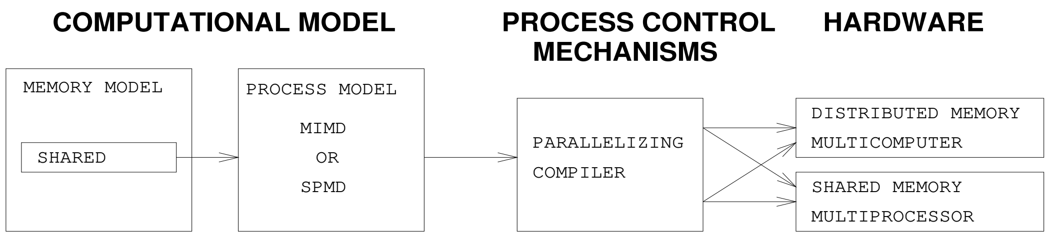
Control Mechanisms

Mechanisms are needed to control how processes interact.

Past and Current:



Future:



Interconnection Networks

- Parallel computers with many processors do not use shared memory hardware.
- Instead each processor has its own local memory and data communication takes place via message passing over an *interconnection network*.
- The characteristics of the interconnection network are important in determining the performance of a multicomputer. If network is too slow for an application, processors may have to wait for data to arrive.
- Important networks include
  - fully connected or all-to-all
  - mesh
  - ring
  - hypercube
  - shuffle-exchange
  - butterfly
  - cube-connected cycles

Network Metrics

A number of metrics can be used to evaluate and compare interconnection networks.

- **Network connectivity** is the minimum number of nodes or links that must fail to partition the network into two or more disjoint networks.

  Network connectivity measures the resiliency of a network, and its ability to continue operation despite disabled components. When components fail we would like the network to continue operation with reduced capacity.

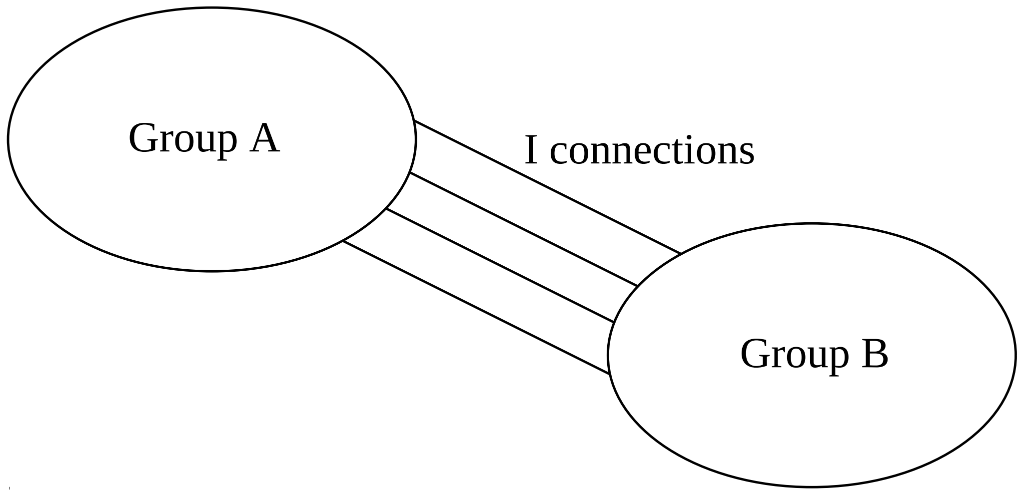
- **Bisection width** is the minimum number of links that must be cut to partition the network into two equal halves (to within one). The *bisection bandwidth* is the bisection width multiplied by the data transfer rate of each link.

- **Network diameter** is the maximum internode distance, i.e., the maximum number of links that must be traversed to send a message to any node along the shortest path.

  The lower the network diameter the shorter the time to send messages to distant nodes.

More on Network Metrics

- **Network narrowness** measures congestion in a network.



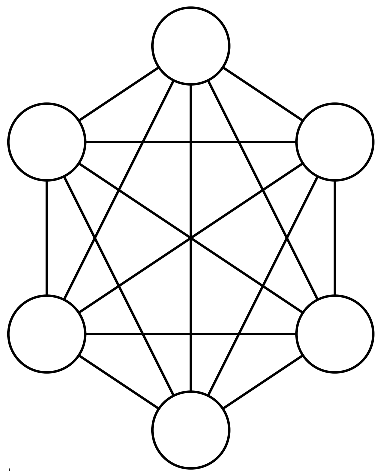
  - Partition the network into two groups A and B, containing NA and NB nodes, respectively, with NB $\leq$ NA. Let I be the number on connections between nodes in A and nodes in B. The narrowness of the network is the maximum value of NB/I for all partitionings of the network.
  - If the narrowness is high (NB>I) then if the group B nodes want to communicate with the group A nodes congestion in the network will be high.

More on Network Metrics

- **Network Expansion Increments** is the minimum number of nodes by which the network can be expanded.
  - A network should be expandable to create larger and more powerful parallel systems by simply adding more nodes to the network.
  - For reasons of cost it is better to have the option of small increments since this allows you to upgrade your machine to the required size.
- **Number of edges per node**. If this is independent of the size of the network then it is easier to expand the system.

Fully Connected Network

- In the fully connected, or all-to-all, network each node is connected directly to all other nodes.
- This is the most general and powerful interconnection network, but it can be implemented for only a small number of nodes.

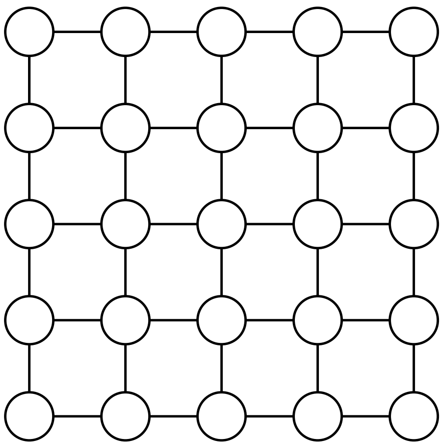


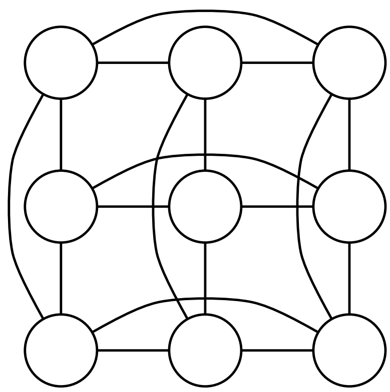
For n even,

$$\begin{aligned}
\text{Network connectivity} &= n-1 \\
\text{Network diameter} &= 1 \\
\text{Network narrowness} &= 1/(n-\lfloor n/2 \rfloor) \\
\text{Bisection width} &= n^2/4 \\
\text{Expansion Increment} &= 1 \\
\text{Edges per node} &= n-1
\end{aligned}$$

Mesh Networks

- In a mesh network nodes are arranged as a q-dimensional lattice, and communication is allowed only between neighboring nodes.
- In a *periodic mesh*, nodes on the edge of the mesh have wrap-around connections to nodes on the other side. This is sometimes called a *toroidal mesh*.





For a q-dimensional nonperiodic lattice with k^q nodes

$$
\begin{aligned}
\text{Network connectivity} &= q \\
\text{Network diameter} &= q(k-1) \\
\text{Network narrowness} &= k/2 \\
\text{Bisection width} &= k^{q-1} \\
\text{Expansion Increment} &= k^{q-1} \\
\text{Edges per node} &= 2q
\end{aligned}
$$

Ring Networks

- A simple ring network is just a 1D periodic mesh.

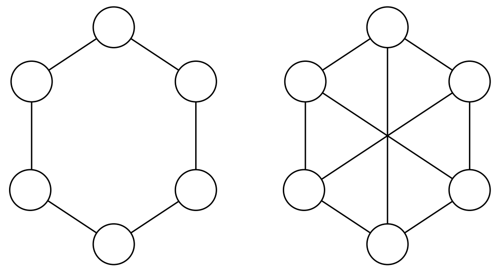
$$\begin{aligned}
\text{Network connectivity} &= 2 \\
\text{Network diameter} &= n/2 \\
\text{Network narrowness} &= n/4 \\
\text{Bisection width} &= 2 \\
\text{Expansion Increment} &= 1 \\
\text{Edges per node} &= 2
\end{aligned}$$

- Problem for a simple ring is large diameter.
- A *chordal ring* uses extra chordal links to reduce the diameter. For a ring with extra diametric links we have (for n even)

$$\begin{aligned}
\text{Network connectivity} &= 3 \\
\text{Network diameter} &= \lceil n/4 \rceil \\
\text{Network narrowness} &= n/(n+4) \\
\text{Bisection width} &= 2 + n/2 \\
\text{Expansion Increment} &= 2 \\
\text{Edges per node} &= 3
\end{aligned}$$

Examples of Ring Networks

Here are a simple ring and a chordal ring with diametric links, each of size 6 nodes.



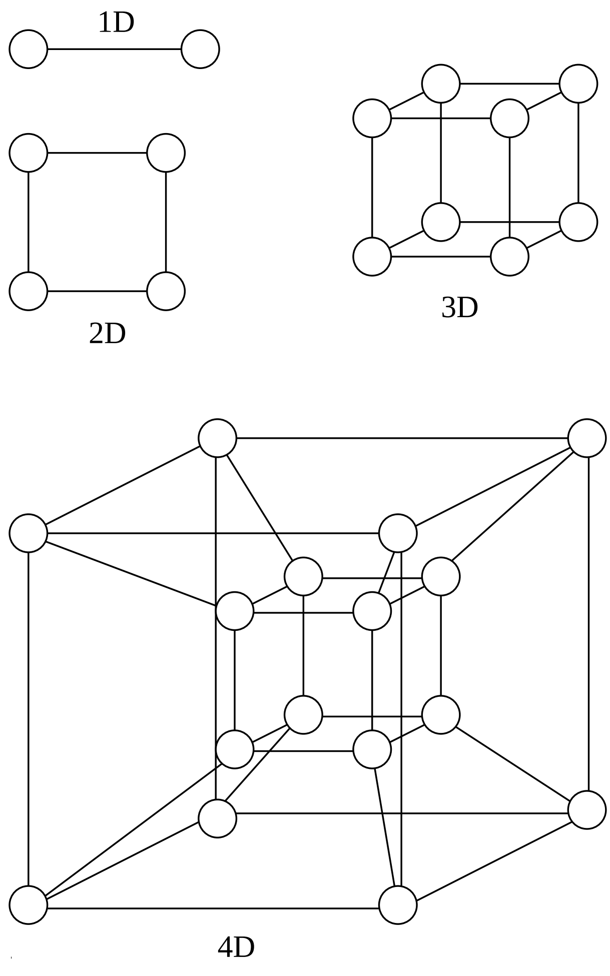
Simple ring Chordal ring

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 $0, 1, \ldots, n-1$, and two nodes are connected if their node numbers differ in exactly one bit.

$$
\begin{aligned}
\text{Network connectivity} &= k \\
\text{Network diameter} &= k \\
\text{Network narrowness} &= 1 \\
\text{Bisection width} &= 2^{k-1} \\
\text{Expansion Increment} &= 2^{k} \\
\text{Edges per node} &= k
\end{aligned}
$$

Examples of Hypercubes



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., $G(i) = G(j) \Rightarrow i = j$.
- $G(i)$ 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.

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

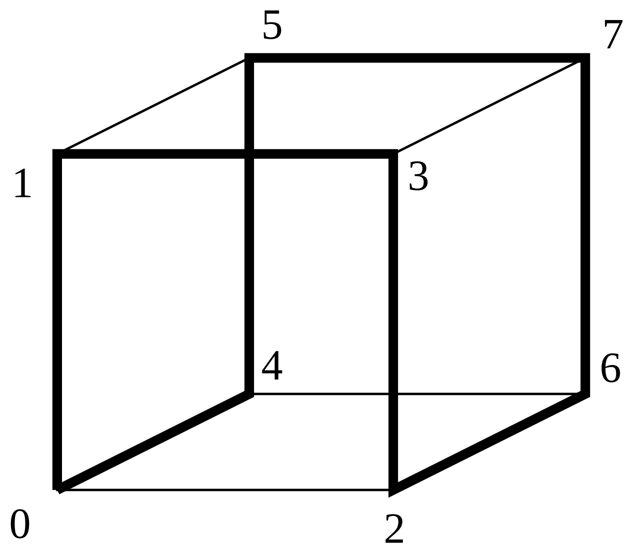
Table 1: 2-bit Gray code

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: 3-bit Gray code

Example of Gray Code

A ring is embedded into a 3D hypercube as follows



Multidimensional 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

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

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

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

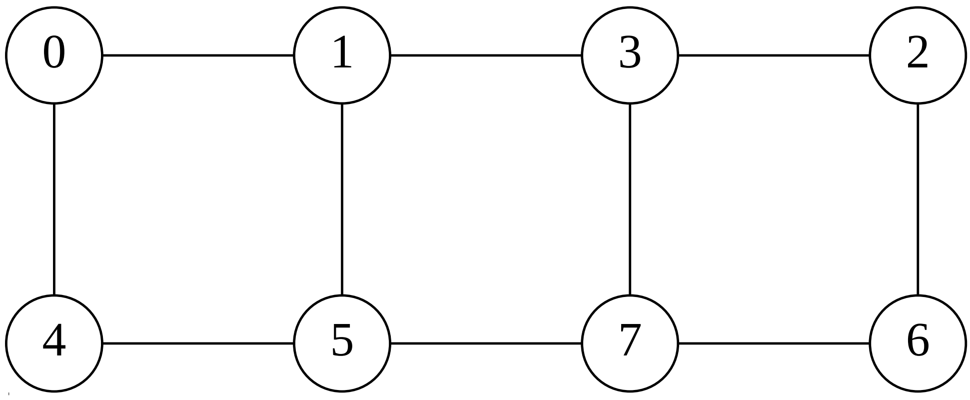
We partition the bits of node number and assign them to each dimension of the mesh. The first d_0 go to dimension 0, the next d_1 bits go to dimension 1, and so on. Then we apply separate inverse Gray code mappings to each group of bits.

To map a 2×4 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)

Example of Gray Code

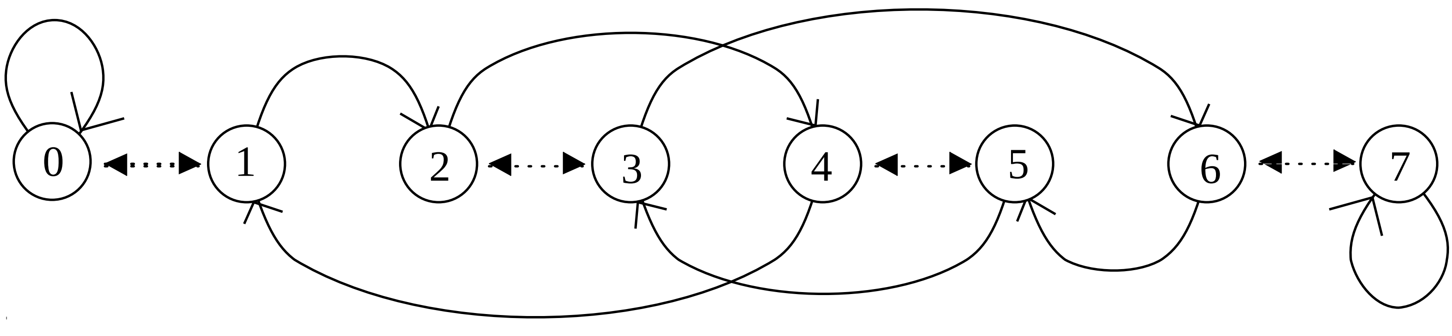
A 2×4 mesh is embedded into a 3D hypercube as follows



On the nCUBE-2, routines are provided for mapping meshes to hypercubes, and *vice versa*, using Gray codes.

Shuffle-Exchange Networks

- A shuffle-exchange network consists of $n = 2^k$ nodes, and two kinds of connections.
  1. *Exchange* connections links nodes whose numbers differ in their lowest bit.
  2. *Perfect shuffle* connections link node i with node $2i \bmod (n-1)$, except for node $n-1$ which is connected to itself.
- Below is an 8-node shuffle-exchange network, in which shuffle links are shown with solid lines, and exchange links with dashed lines



Shuffle-Exchange Networks

What is the origin of the name "shuffle-exchange"

Consider a deck of 8 cards numbered $0, 1, 2, \ldots, 7$. The deck is divided into two halves and shuffled perfectly, giving the order

$$0, 4, 1, 5, 2, 6, 3, 7$$

The final position of a card i can be found by following the shuffle link of node i in a shuffle-exchange network.

- Let $a_{k-1}, a_{k-2} \ldots a_1 a_0$ be the address of a node in a shuffle-exchange network in binary.
- A datum at this node will be at node number

  $$a_{k-2} \ldots a_1 a_0 a_{k-1}$$

  after a shuffle operation.
- This corresponds to a cyclic leftward shift in the binary address.
- After k shuffle operations we get back to the node we started with, and nodes through which we pass are called a *necklace*.

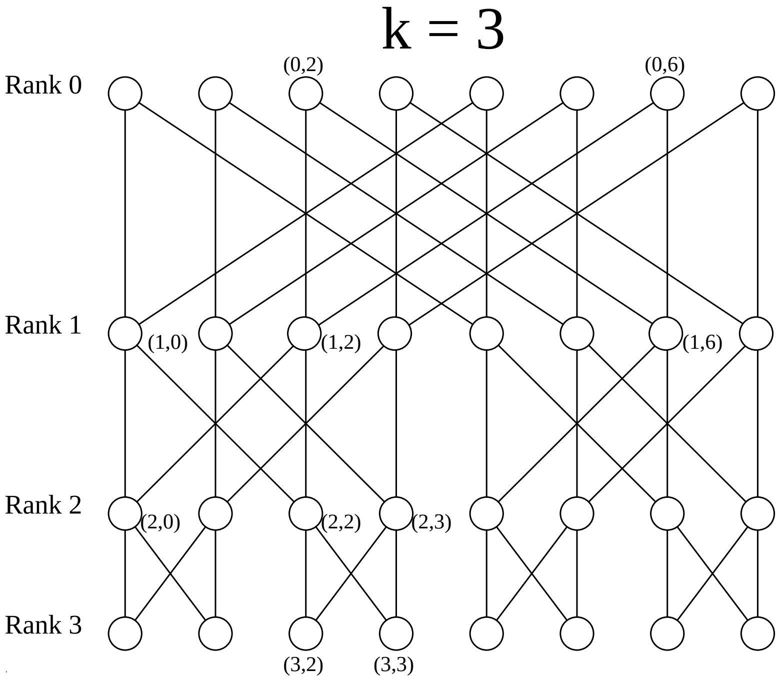
Butterfly Network

- A butterfly network consists of $(k+1)2^k$ nodes divided into $k+1$ rows, or *ranks*.
- Let node(i,j) refer to the jth node on the ith rank. Then for $i > 0$ node(i,j) is connected to 2 nodes on rank $i-1$, node$(i-1,j)$ and node$(i-1,m)$, where m is the integer found by inverting the ith 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.

$$\begin{aligned}\text{Network diameter} &= 2k \\ \text{Bisection width} &= 2^k\end{aligned}$$

Example of a Butterfly Network

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



$$i = 1, j = 2 = (010)_2, j' = (110)_2 = 6$$
$$i = 2, j = 2 = (010)_2, j' = (000)_2 = 0$$
$$i = 3, j = 2 = (010)_2, j' = (011)_2 = 3$$

Cube-Connected Cycles Network

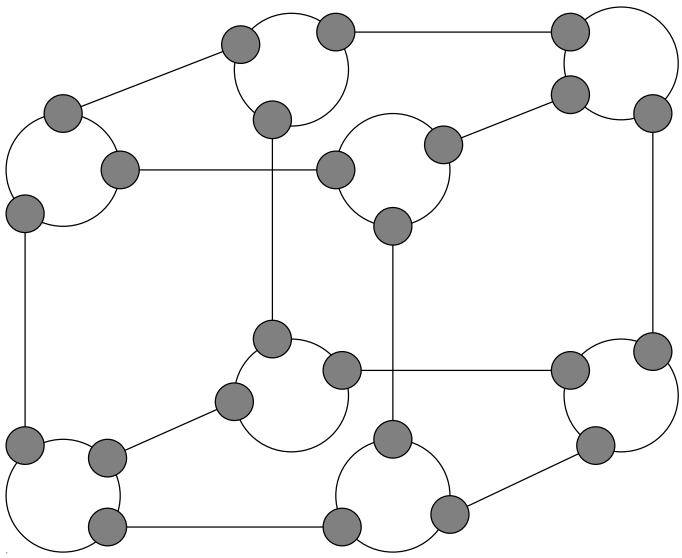
- A cube-connected cycles network is a k-dimensional hypercube whose 2^k vertices are actually cycles of k nodes.
- The advantage compared with the hypercube is that the number of edges per node is a constant, 3. Disadvantages are that network diameter is twice that of a hypercube, and the bisection width is lower.

For a cube-connected cycle network of size $k2^k$,

$$\begin{aligned} \text{Network diameter} &= 2k \\ \text{Bisection width} &= 2^{k-1} \\ \text{Edges per node} &= 3 \end{aligned}$$

Example of a Cube-Connected Cycles Network

A cube-connected cycles network with $k = 3$ looks like this:



Classifying Parallel Algorithms

Parallel algorithms for MIMD machines can be divided into 3 categories

1. Pipelined algorithms
2. Data parallel, or partitioned, algorithms
3. Asynchronous, or relaxed, algorithms

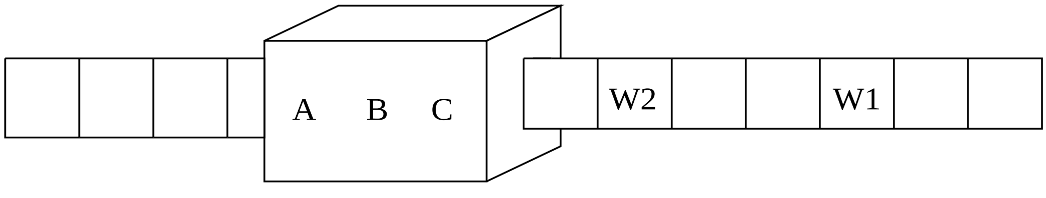
Pipelined Algorithms

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.

Example: Suppose it takes 3 steps, A, B, and C, to assemble a widget, and each step takes one unit of time.

- In the sequential case it takes 3 time units to assemble each widget. Thus it takes $3n$ time units to produce n widgets.



Example of Pipelined Algorithm

- In the pipelined case the following happens

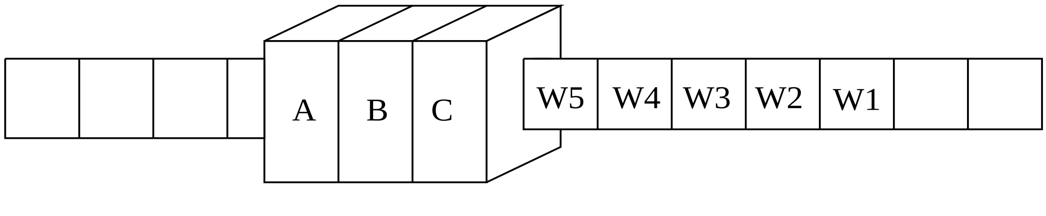
  Time step 1: A operates on W1

  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 widget is produced every time step.
- If the pipeline is n processes long, a new widget is produced every time step from the nth 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 (widgets)

then

$$\begin{aligned} \text{Sequential time} &= NTM \\ \text{Pipelined time} &= (N+M-1)T \end{aligned}$$

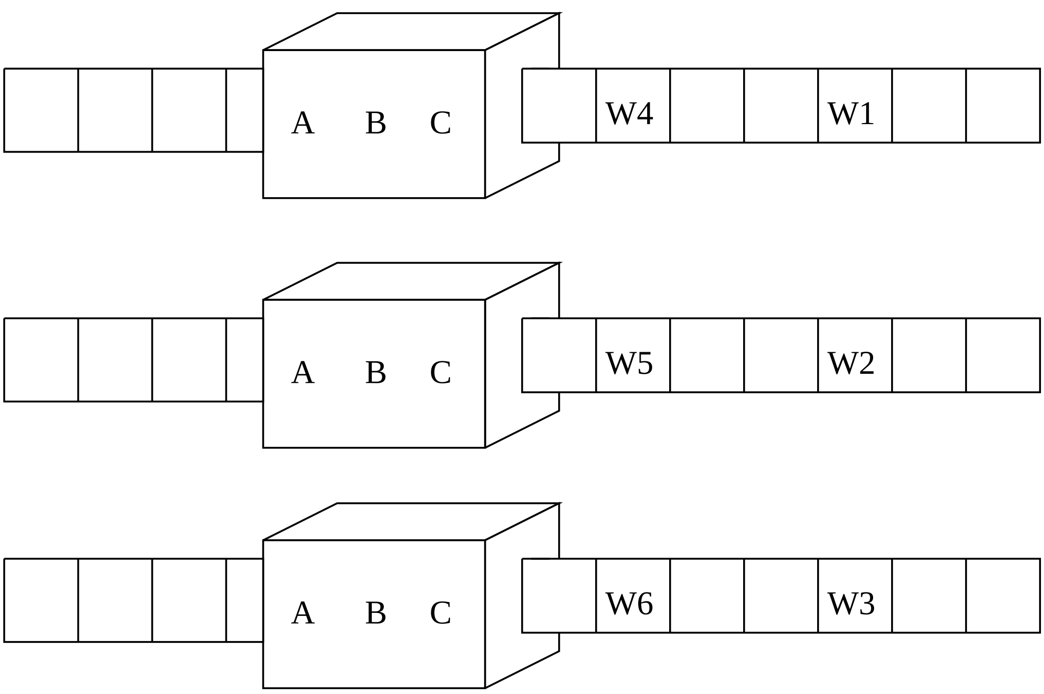
Example: If $T = 1$, $N = 100$, and $M = 10^6$, then

$$\begin{aligned} \text{Sequential time} &= 10^8 \\ \text{Pipelined time} &= 1000099 \end{aligned}$$

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 subproblems to form the complete solution. This may involve communication between processors.
- Data parallelism can be exploited in the widget 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.

For example, suppose we have 2 processors, A and B.

- A produces a sequence of numbers, a_i, $i = 1, 2, \ldots$.
- 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

etc.

Asynchronous Mode

Asynchronous mode

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
  1. they are difficult to analyse
  2. 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 $f(x) = 0$.

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

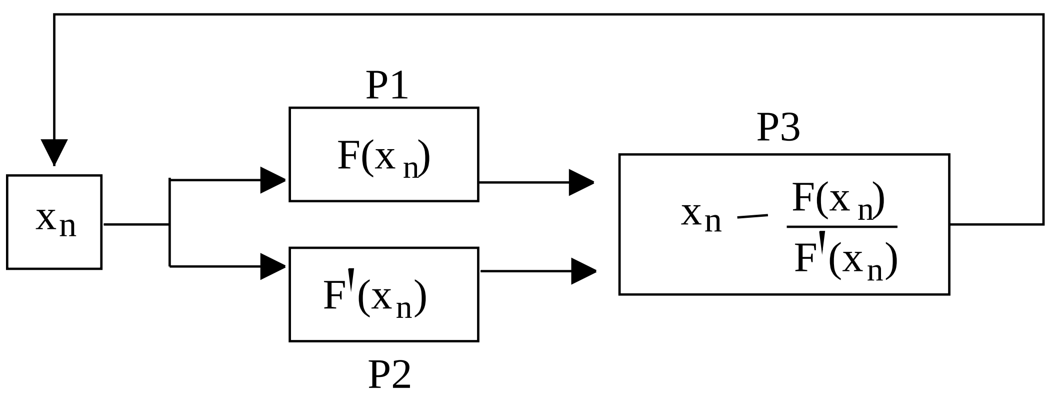
generates a sequence of approximations to the root, starting with some initial value x_0.

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.



Example of Asynchronous Algorithm

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,

$$k(\max(t_1, t_2) + t_3)$$

Example Continued

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.

For example, if $t_1 = 2$, $t_2 = 3$ and $t_3 = 1$.

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	–	$f'(x_1)$	$x_2 = x_1 - f(x_1)/f'(x_0)$
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	$x_6 = x_5 - f(x_3)/f'(x_2)$

Ci indicates processor is using x_i in its calculation.

- Time per iteration is at most $\min(t_1, t_2) + t_3$, but may be as low as t_3.
- Cannot predict number of iterations.

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.

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

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

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

- **Overhead** is defined as

$$f(N) = \frac{1}{\epsilon(N)} - 1$$

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

$$\begin{aligned} \text{Speed-up} &= 8/2 = 4 \\ \text{Efficiency} &= 4/5 = 0.8 \\ \text{Overhead} &= 1/0.8 - 1 = 0.25 \end{aligned}$$

- *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.

2. **Load imbalance overhead**

   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.

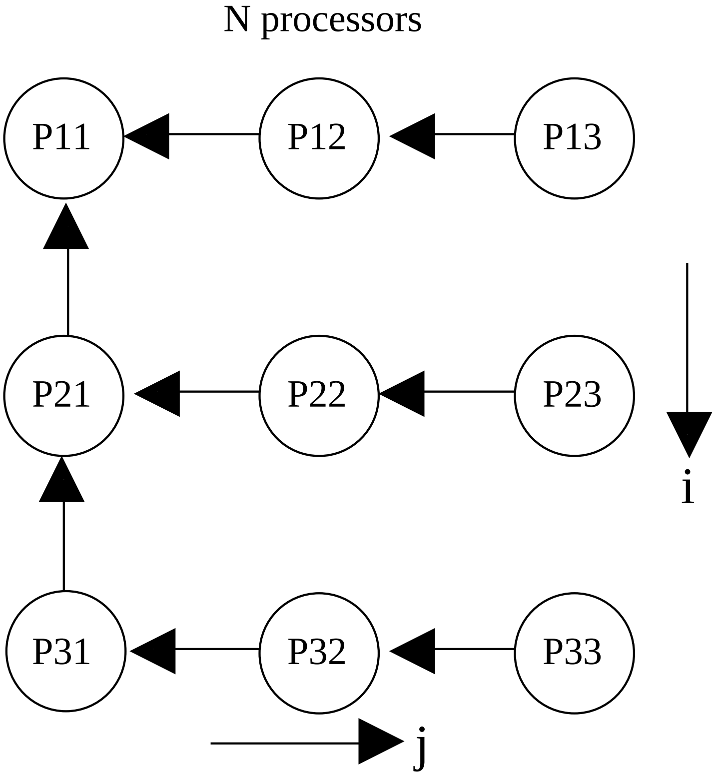
3. **Communication overhead**

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

The *grain size* 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.

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 P11.

Analysis of Summing Example

Time for best sequential algorithm is

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

where t_{calc} is time to perform one floating-point operation.
Time for each phase of parallel algorithm

1. Form local sums

$$T_1 = (m/N - 1)t_{\text{calc}}$$

2. Sum along processor rows

$$T_2 = (\sqrt{N} - 1)(t_{\text{calc}} + t_{\text{comm}})$$

   where t_{comm} is time to communicate one floating-point number between neighbouring processors.

3. Sum up first column of processors

$$T_3 = (\sqrt{N} - 1)(t_{\text{calc}} + t_{\text{comm}})$$

Total time for parallel algorithm is

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

Analysis of Summing Example

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

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

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

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

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

$$
\begin{aligned}
S &= \frac{N}{1 + 2\sqrt{N}(1+\tau)/g} \\
\epsilon &= \frac{1}{1 + 2\sqrt{N}(1+\tau)/g} \\
f &= 2\sqrt{N}(1+\tau)/g
\end{aligned}
$$

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.

  The summing algorithm is scalable since we can take $g \propto \sqrt{N}$.
- 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.

  The summing algorithm is not perfectly scalable.
- 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.

  The summing algorithm 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 α 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

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

where C is the time for overhead due to communication, load balancing, etc. Then the speed-up satisfies

$$\begin{aligned} S(N) &\leq \frac{n}{(1-\alpha)n/N + \alpha n} \\ \Rightarrow S(N) &\leq \frac{1}{(1-\alpha)/N + \alpha} \end{aligned}$$

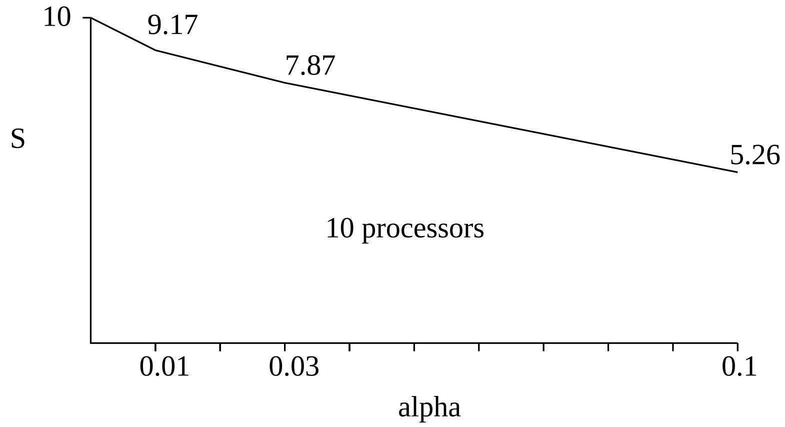
Note that

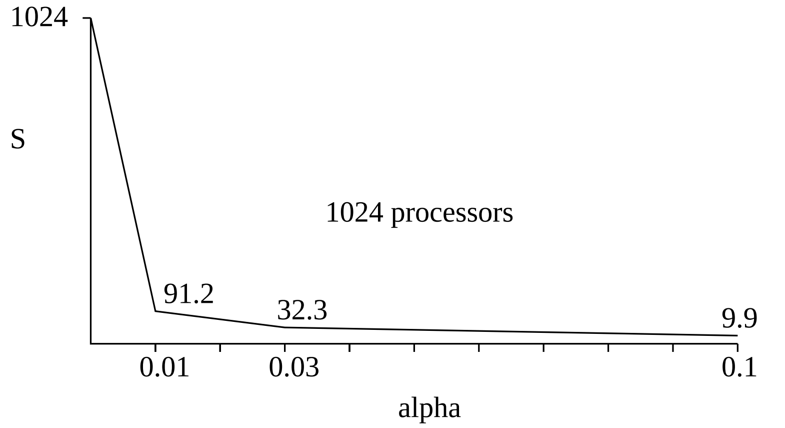
$$S(N) \to \frac{1}{\alpha} \text{ as } N \to \infty$$

so the speed-up is always limited to a maximum of $1/\alpha$, no matter how many processors are used.

Example of Amdahl's Law

Consider the effect of Amdahl's Law on speed-up as a function of serial fraction, α, for $N = 10$ and $N = 1024$ 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 1024-processor machine.

Implications of Amdahl's Law

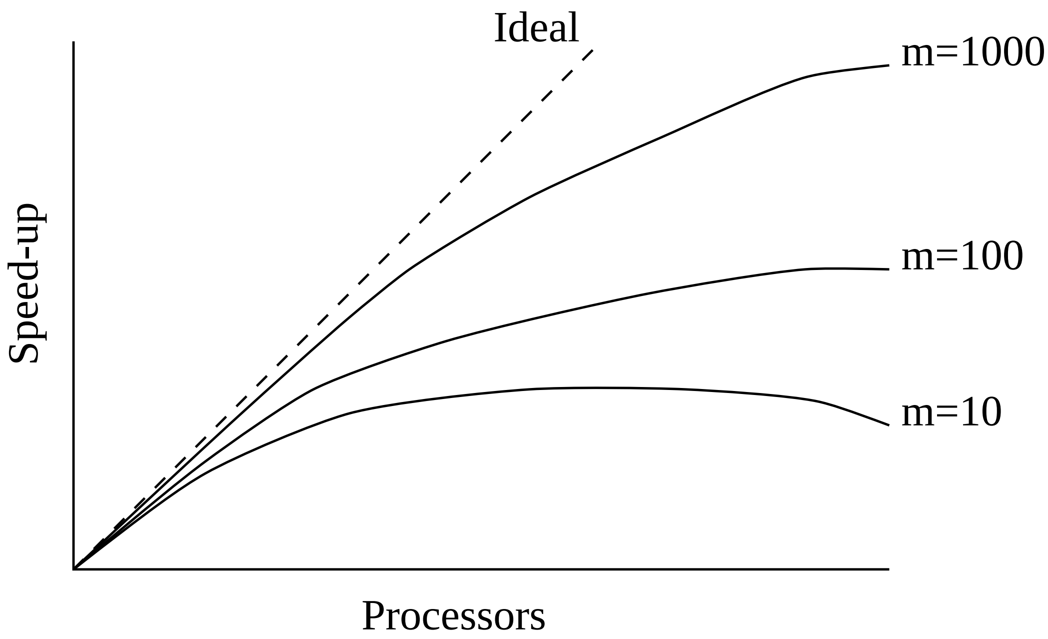
- Amdahl's Law says that the serial fraction puts a severe constraint on the speed-up that can be achieved as the number of processors increases.
- Amdahl's Law suggests that it is not cost effective to build systems with large numbers of processors because sufficient speed-up will not be achieved.
-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

Scaled Speed-Up

- *Scaled speed-up* is the ratio between how long the best sequential algorithm would take on a single processor and how long it actually took to run on multiple processors.
- Speed-up and scaled speed-up differ because to measure 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.
- In finding the scaled speedup we are allowed to 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 grain size. Thus, speed-up is an increasing function of problem size, and so use of scaled speed-up allows us to measure larger speed-ups.

Speed-Up and Problem Size

For a given number of processors, speedup increases with problem size, m.



Semantics of Message Sends

Suppose one node sends a message to another node

```
send (data, length, destination)
```

There are two possible behaviours

- **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.

- **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.

MPI Communication Modes

- In MPI the *mode* of a point-to-point communication operation governs when a send operation is initiated, or when it completes.
- **Standard mode:**
  - A send may be initiated even if a matching receive has not been initiated.
- **Ready mode:**
  - A send may be initiated only if a matching receive has been initiated.
- **Synchronous mode:**
  - The same as standard mode, except the send will not complete until message delivery is guaranteed.
- **Buffered mode:**
  - Similar to standard mode, but completion is always independent of matching receive, and message may be buffered to ensure this.

Semantics of Message Receives

Suppose one node receives a message from another node

```
receive (data, max_length, source)
```

There are two possible behaviours

- **Blocking receive**

  The receive does not return until the data to be sent 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 process.

- **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.

Message Passing Protocols

Suppose one node sends a message and another receives it.

```
SOURCE:   send (data, length, destination)
DEST:     receive (data, max_length, source)
```

Two important message passing protocols are

- **Synchronous send protocol**

  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 send protocol**

  The send and receive routines do not necessarily overlap in time. The send can return regardless of whether the receive has been initiated.

MPI Blocking Send

- Consider the standard blocking send routine in C:

```
int MPI_Send (
          void          *start_of_buffer,
          int            number_of_items,
          MPI_Datatype   datatype,
          int            destination_rank,
          int            tag,
          MPI_Comm       communicator)
```

MPI Blocking Send

- *start_of_buffer*: Address of the start of the data to be sent.
- *number_of_items*: Number of items of the specified datatype to be sent.
- *datatype*: The datatype of the items to be sent. User-defined datatypes can be used, but mostly the standard MPI-supplied datatypes are used, such as `MPI_INT` and `MPI_FLOAT`.
- *destination_rank*: The process number of the destination process.
- *tag*: The message tag. This is can be used to distinguish between different types of message.
- *communicator*: The communicator defines which processes may be involved in the communication. In most elementary applications the MPI-supplied communicator `MPI_COMM_WORLD` is used.

MPI Blocking Receive

- Consider the standard blocking receive routine in C:

```
int MPI_Recv (
          void          *start_of_buffer,
          int            number_of_items,
          MPI_Datatype   datatype,
          int            source_rank,
          int            tag,
          MPI_Comm       communicator,
          MPI_Status     status)
```

- In a receive routine, `source_rank` and `tag` can have the values `MPI_ANY_SOURCE` and `MPI_ANY_TAG`.
- The tag and communicator values specified by the receiver must match those of the sender for a communication to take place.

Return Status Objects

- The return status object may be used after completion of a receive to find the actual length, source, and tag of a message.
- Return status object is MPI-defined type.
- In C `status` is a structure.

  status.source gives source process

  status.tag gives the message tag
- Number of elements in message is given by

```
MPI_Get_count (status, datatype, count)
```

Point-to-point Communication

- MPI provides for point-to-point communication between pairs of processes.
- Message selectivity 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.
- Rank and tag may be wildcarded.
- The components of a communicator may not be wildcarded.

Communication Completion

- A communication operation is **locally complete** on a process if the process has completed its part in the operation.
- A communication operation is **globally complete** if all processes involved have completed their part in the operation.

A communication operation is globally complete
if and only if
it is locally complete for all processes.

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 routine `MPI_Comm_size`.

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

In most applications, the communicator is `MPI_COMM_WORLD`.

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 routine `MPI_Comm_rank`.

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

Example Program 1

In this simple program every process sends its rank to process 0 where it is output.

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

main (argc, argv)
int argc;
char *argv[];
{
   int rank, n, i, message;
   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 ("\n Hello from process %3d", rank);
      for (i=1;i<n;i++) {
         MPI_Recv (&message, 1, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG,
                   MPI_COMM_WORLD, &status);
         printf ("\n Hello from process %3d", message);
      }
   }
   else MPI_Send (&rank, 1, MPI_INT, 0, 111, MPI_COMM_WORLD);

   MPI_Finalize ();
}
```

Notes on Program 1

Note the following:

- All MPI calls must come between the calls to `MPI_Init` and `MPI_Finalize`.
- The information from the processes is not necessarily output in ascending order because the program does not specify the order in which process 0 receives messages.
- We could have each process output its rank instead of sending it to process 0.

Compiling and Running

MPI programs may be run on a network of workstations such as those in the COMSC Unix lab. Just log on to one of the unix workstations and use the following command to compile:

```
mpicc -O -o file file.c -I/mnt51/mpich/include
      -L/mnt51/mpich/lib/solaris/ch_p4 -lmpi
```

where `file.c` is the C source code, and `file` is the name of the executable. The above should be input all on one line.

To run a program on the network of workstations do the following
mpirun -np `n` *-machinefile* `machines file`
where `n` is the number of processes you want to use, `file` is the name of the executable, and `machines` is a file you have previously set up containing the names of the machines you wanr to use, one on each line. For example, to run the program `sum` with 4 processes

```
mpirun -np 4 -machinefile machines sum
```

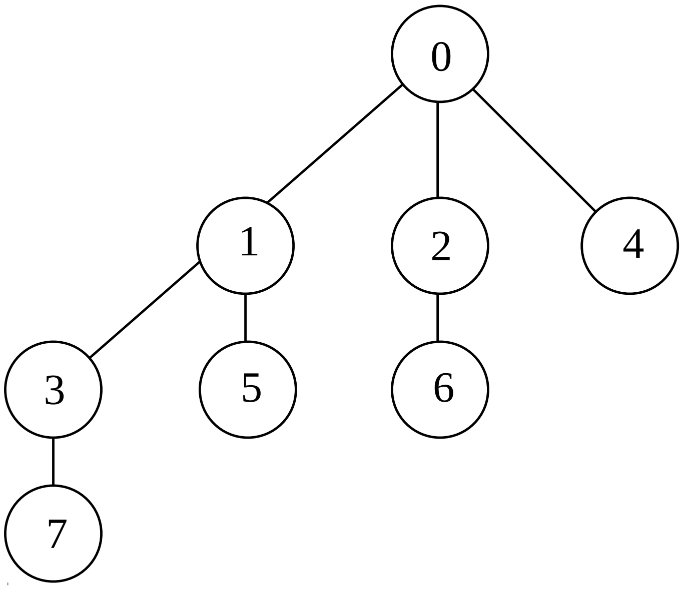
Collective Communication

The send and receive style of communication between pairs of processes is known as *point-to-point communication*. This is distinct from *collective communication* in which several processes are involved in a coordinated communication task.
Examples include:

- Broadcasting data. One process, known as the *root*, sends the same data to all processes.
- Data reduction. Data from all processes is combined using a *reduction function* to produce a single result. The result may reside on a single process or on all processes.

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.



For a hypercube a tree broadcast uses direct links and if hardware can send on all link simultaneously then

- **Step 1**. 0 sends to 1, 2, and 4
- **Step 2**. 1 sends to 3 and 5, 2 sends to 6
- **Step 3**. 3 sends to 7

If hardware can send on only one link at a time

- **Step 1**. 0 sends to 1
- **Step 2**. 0 sends to 2, 1 send to 3
- **Step 3**. 0 sends to 4, 1 sends to 5, 2 sends to 6, 3 sends to 7

Reduction

Reduction can also be represented by a tree algorithm. For example, if we want to sum numbers on all nodes to one node see Fig. 1.

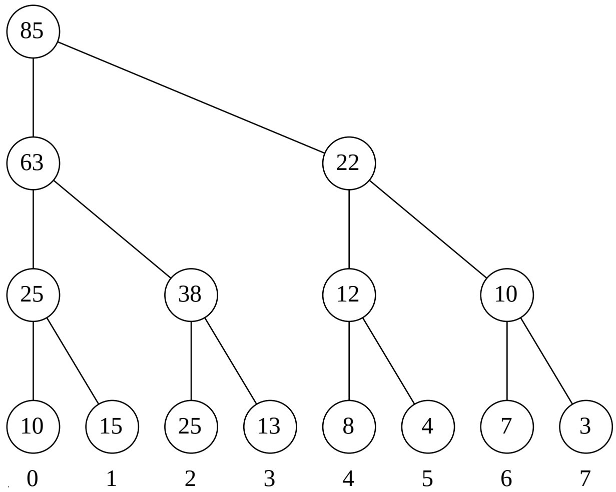


Figure 1: Reduction to node 0.

If we want to perform the sum so that all nodes end up with the result see Fig. 2.

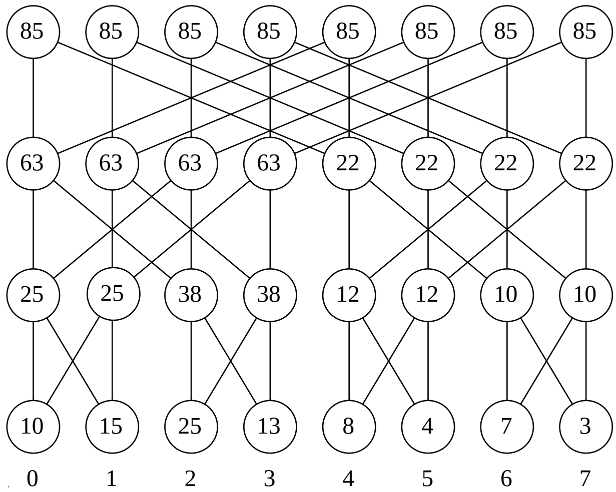


Figure 2: Reduction to all nodes.

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., time to run is approximately proportional to $\lg n$.
- On hypercubes the logarithmic algorithms involve communication only between neighbouring processes.
- Other algorithms may be better for other network topologies.
- MPI provides routines for broadcasting and reduction.

MPI Integration Example

Want to find

$$\int_0^\pi \sin x \, dx$$

1. **Initialisation**
   (a) initialise MPI
   (b) communicate problem parameters
2. **Compute**
   (a) each process computes its contribution
   (b) reduction operation sums process contributions
3. **Output**

   Process with rank 0 outputs the result
4. **Tidy Up**

   All processes call `MPI_Finalize`

MPI Code for Integration

Initialisation...

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

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

main (argc, argv)
int argc;
char *argv[];
{
   int rank, nprocs;
   int i, nlocal, nbeg, nend, npts;
   FILE *fp;
   float 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) {
      fp = fopen ("input.dat", "r");
      fscanf (fp, "%d", &npts);
   }

   MPI_Bcast (&npts, 1, MPI_INT, 0, MPI_COMM_WORLD);

   nlocal = (npts-1)/nprocs + 1;
   nbeg   = rank*nlocal + 1;
   nend   = min (nbeg+nlocal-1, npts);
```

MPI Code for Integration

. . .computation, output, and exit

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

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

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

   MPI_Finalize ();
}
```

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.

```
MPI_Cart_create (comm_old, ndims, *dims, *period,
                 reorder, *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.

Topological Inquiry Routines

- mpi_topo_test returns the type of topology associated with a communicator.
- Can find number of dimensions in a Cartesian topology:

```
MPI_Cartdim_get (comm, *ndims)
```

- More information on a Cartesian topology can be obtained with:

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

- Mapping of coordinate position in Cartesian topology to rank:

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

- Mapping of rank to coordinate position:

```
MPI_Cart_coords (comm, rank, maxdims,
                 *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 group of size N:
  - Circular shift by J. Data in process K is sent to process
  $$(\bmod\ (J+K), N)$$
  - 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.
- Topological shifts are performed using `MPI_Sendrecv`.
- `MPI_Cart_shift` returns the ranks of the processes that a process must send to and receive from when performing a shift on a topological group.

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:

  `MPI_Sendrecv`

- For identical send/receive buffers:

  `MPI_Sendrecv_replace`

Vibrating String Problem

We shall now study the vibration of waves on a string, and design a parallel MPI program to solve the partial differential equation that describes the problem mathematically.

Problem

A string of length L and fixed at each end is initially given a known displacement. What is the displacement at later times?

- Introduce coordinate x so that one end of the string is at $x = 0$ and the other end is at $x = L$.
- Denote the displacement of the string at position x and time t by $\psi(x, t)$.
- We want to know $\psi(x, t)$.

The Wave Equation

Mathematically the vibrating string problem is described by the *wave equation*. The mathematical description of the problem is to solve

$$\frac{1}{c^2}\frac{\partial^2\psi}{\partial t^2} - \frac{\partial^2\psi}{\partial x^2} = 0$$

subject to the boundary conditions

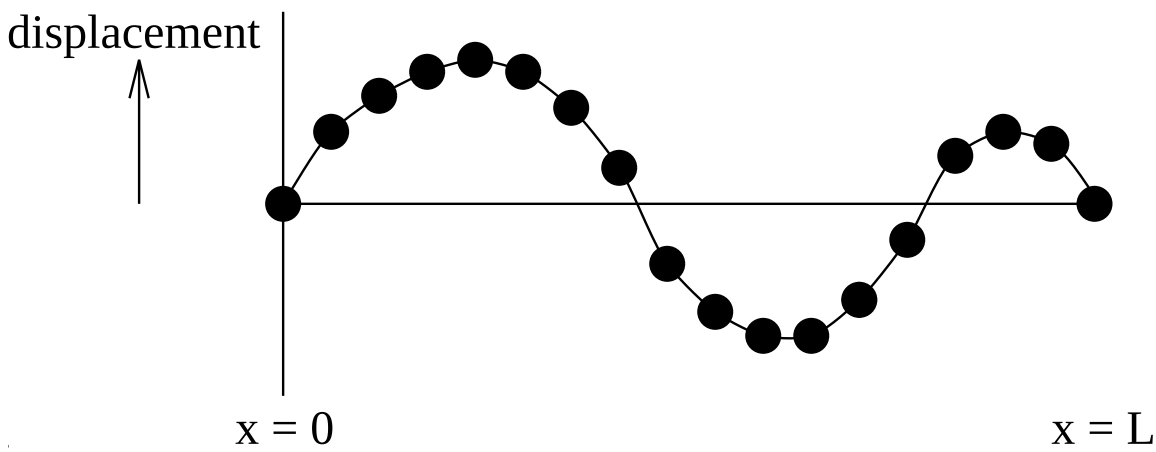
$$\psi(0,t) = \psi(L,t) = 0$$

and the initial condition

$$\psi(x,0) = u(x)$$

where $u(x)$ is some known function described the initial displacement of the string.

We shall solve this problem numerically by approximating the solution at a number of equally-spaced values of x.



Numerical Approximation

- Let the n values of x at which we wish to approximate the solution be denoted by $x_0, x_1, \ldots, x_{n-1}$, where $x_i = i\Delta x$ for $i = 0, 1, \ldots, n-1$, and $\Delta x = L/(n-1)$.
- We shall denote the displacement, $\psi(x_i, t)$, at position x_i and time t by $\psi_i(t)$.
- We shall find the solution at a series of time steps $t_j = j\Delta t$ for $j = 0, 1, \ldots, J-1$.
- Given estimates of the solution at times t and $t - \Delta t$ it can be shown that the solution at time $t + \Delta t$ can be approximated by

  $$\psi_i(t+\Delta t) = 2\psi_i(t) - \psi_i(t-\Delta t) + \tau^2(\psi_{i-1}(t) - 2\psi_i(t) + \psi_{i+1}(t))$$

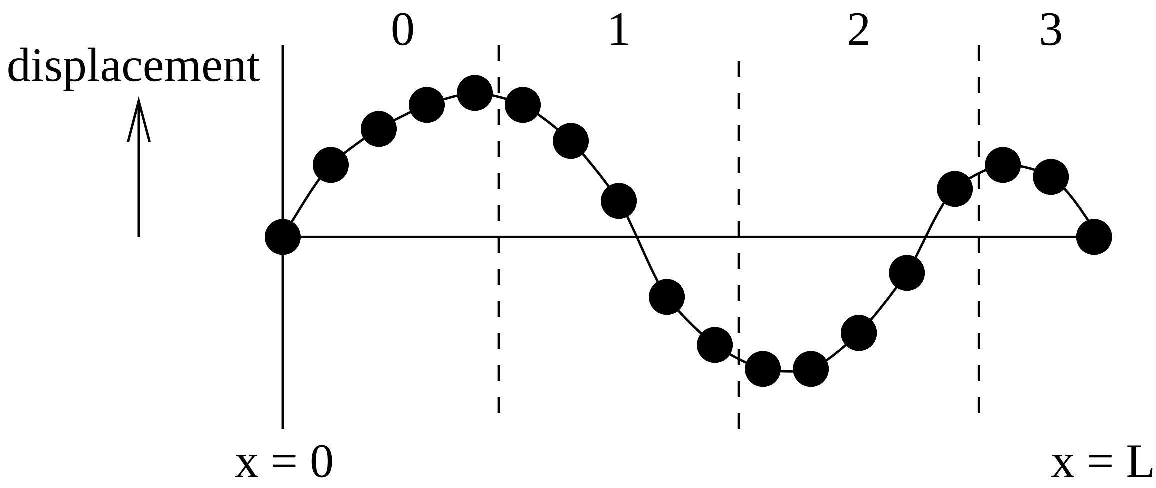
  where $\tau = c\Delta t/\Delta x$.
- Thus, ψ at a new time step depends on the value at the same position for the previous two time steps, and the values at neighbouring positions for the most recent time step.
- We shall specify the initial displacement to be a sine wave, i.e., $u(x) = \sin x$.

Sequential Code

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#define MAX_POINTS 128
main()
{
    double psi[MAX_POINTS];
    double new_psi[MAX_POINTS];
    double old_psi[MAX_POINTS];
    double pi = 3.141592654;
    double tau = 0.05;
    double x;
    int    i, j;
    int    npoints = 128;
    int    nsteps  = 100;
    for(i=0;i<npoints;i++){
        x = 2.0*pi*(double)i/(double)(npoints-1);
        x = sin(x);
        psi[i] = old_psi[i] = x;
    }
    for(j=0;j<nsteps;j++){
        for(i=1;i<npoints-1;i++){
            new_psi[i] = 2.0*psi[i] - old_psi[i] +
                tau*tau*(psi[i-1]-2.0*psi[i]+psi[i+1]);
        }
        for(i=0;i<npoints;i++){
            old_psi[i] = psi[i];
            psi[i]     = new_psi[i];
        }
    }
    for(i=0;i<npoints;i++)
        printf("%9.6f%c",psi[i],((i+1)%8==0) ? '\n' : ' ');
    exit(0);
}
```

Data Distribution

- Give each process a block of points on the string.
- Each process should have approximately the same number of points to ensure good load balance.

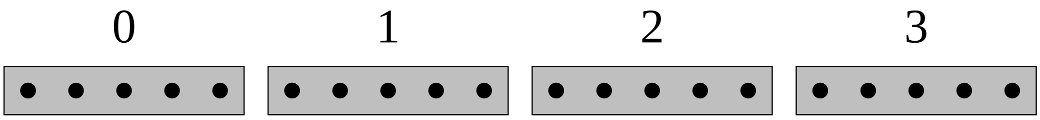


Communication Requirements

The update formula is

$$\psi_i(t+\Delta t) = 2\psi_i(t) - \psi_i(t-\Delta t) + \tau^2(\psi_{i-1}(t) - 2\psi_i(t) + \psi_{i+1}(t))$$

so to update a point we need to know the value of ψ at the neighbouring points. This entails communication.



Each process needs to communicate the ψ values for its first and last points before performing an update.

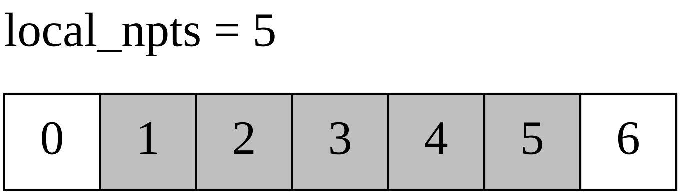
Outline of Parallel Code

The main phases of the parallel version of the code are as follows.

- **Initialise data distribution**
  - Find position of each process to determine which block of points it handles.
  - Find out the node numbers of processes to left and right
- **Initialise arrays**
  - Determine how many points each process handles and the index of the first point in each.
  - Set the `psi` and `old_psi` arrays
- **Perform Update**
  - Communicate end points
  - Do update locally
- **Output results**

Array Declarations

Each process needs to store the endpoint values received from the neighbouring processes. These are stored at the 0 and `local_npts+1` positions in the `psi array`



Thus, the `psi` array need two "extra" entries, and is declared as

```
double psi[MAX_POINTS+2];
double new_psi[MAX_POINTS+2];
double old_psi[MAX_POINTS+2];
```

Initialising the Data Distribution

```
int periods=0, reorder=1;
int mypos, nprocs, rank;
MPI_Comm new_comm;

MPI_Comm_size (MPI_COMM_WORLD, &nprocs);
MPI_Comm_rank (MPI_COMM_WORLD, &rank);
MPI_Cart_create (MPI_COMM_WORLD, 1, &nprocs, &periods,
                 reorder, &new_comm);
MPI_Cart_coords (new_comm, rank, 1, &mypos);
MPI_Cart_shift  (new_comm, 0, -1, &right, &left);
```

- `MPI_Comm_size` gives us the number of processes in use.
- `MPI_Comm_rank` gives the rank of each process.
- `MPI_Cart_create` sets up a new communicator, `new_comm`, with a 1D topology associated with it.
- `MPI_Cart_coords` gives the position in the 1D topology of each process.
- `MPI_Cart_shift` gives `left` and `right`, the neighbouring processes to the left and right respectively.

Initialisation of Arrays

The arrays are initialised as follows.

```
local_npts = npoints/nprocs;
lbound     = mypos*local_npts;
for(i=0;i<local_npts;i++){
    x = 2.0*pi*(double)(lbound+i)/(double)(npoints-1);
    x = sin(x);
    psi[i+1] = old_psi[i+1] = x;
}
```

- `local_npts` is the number of points updated by each process.
- `lbound` is the index of the first point in each process, i.e., it is the global index corresponding to local index 1.
- We initialise the arrays for indices 1 up to `local_npts`. Indices 0 and `local_npts+1` will be used later to store values received from neighbouring processes.

Update Phase

The update phase has 3 main parts.

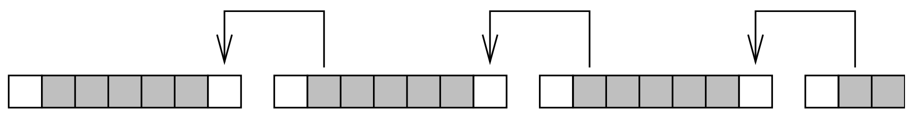
1. Communicate endpoints between neighbours
2. Update points locally
3. Copy arrays ready for next update step

```
istart = (mypos==0) ? 2 : 1;
iend   = (mypos==nprocs-1) ? local_npts-1 : local_npts;
for(j=0;j<NSTEPS;j++){
   MPI_Sendrecv (&psi[1],            1, MPI_DOUBLE, left,  111,
                 &psi[local_npts+1], 1, MPI_DOUBLE, right, 111,
                 new_comm, &status);
   MPI_Sendrecv (&psi[local_npts], 1, MPI_DOUBLE, right,  111,
                 &psi[0],          1, MPI_DOUBLE, left,   111,
                 new_comm, &status);
   for(i=istart;i<=iend;i++){
      new_psi[i] = 2.0*psi[i] - old_psi[i] +
                   tau*tau*(psi[i-1]-2.0*psi[i]+psi[i+1]);
   }
   for(i=1;i<=local_npts;i++){
      old_psi[i] = psi[i];
      psi[i]     = new_psi[i];
   }
}
```

Communication Code

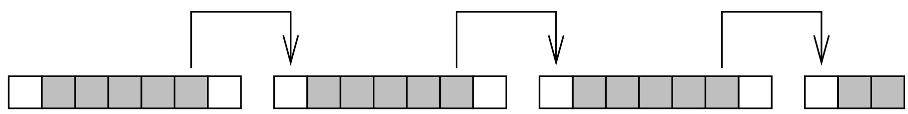
First all processes send `psi[1]` to the process to the left, and receive data from the process to the right, storing it in `psi[local_npts+1]`.

```
MPI_Sendrecv (&psi[1],             1, MPI_DOUBLE, left,  111,
              &psi[local_npts+1], 1, MPI_DOUBLE, right, 111,
              new_comm, &status);
```



Next all processes send `psi[local_npts]` to the process to the right, and receive data from the process to the left, storing it in `psi[0]`.

```
MPI_Sendrecv (&psi[local_npts], 1, MPI_DOUBLE, right,  111,
              &psi[0],          1, MPI_DOUBLE, left,   111,
              new_comm, &status);
```



Output

- We would like the format of the output to be independent of the number of processes used, and the same as for the sequential program.
- We can ensure this happens by passing the data to be output to process 0 and outputting it from there.

```
for(j=0;j<nprocs;j++){
   if (mypos==0){
      MPI_Cart_rank (new_comm, &j, &proc);
      if (j>0) MPI_Recv (&psi[1], local_npts, MPI_DOUBLE,
                          proc, a_tag, new_comm, &status);
      for(i=1;i<=local_npts;i++)
         printf("%9.6f%c",psi[i],(i%8==0) ? '\n' : ' ');
    }
    else if (mypos==j)
       MPI_Send (&psi[1], local_npts, MPI_DOUBLE, 0,
                 a_tag, new_comm);
}
```

Performance Analysis

To analyse the performance of the parallel wave equation code we just look at the update phase.

- To update each point requires 6 floating-point operations in the parallel and sequential codes.
- In the parallel code each process sends and receives two floating-point numbers in each update step.
- Ignoring the time to copy to the arrays `old_psi` and `psi`, the speed-up is

$$\begin{aligned} S(N) &= \frac{6nt_{\mathrm{calc}}}{(6n/N)t_{\mathrm{calc}} + 2t_{\mathrm{comm}}} \\ &= \frac{N}{1+\tau/(3g)} \end{aligned}$$

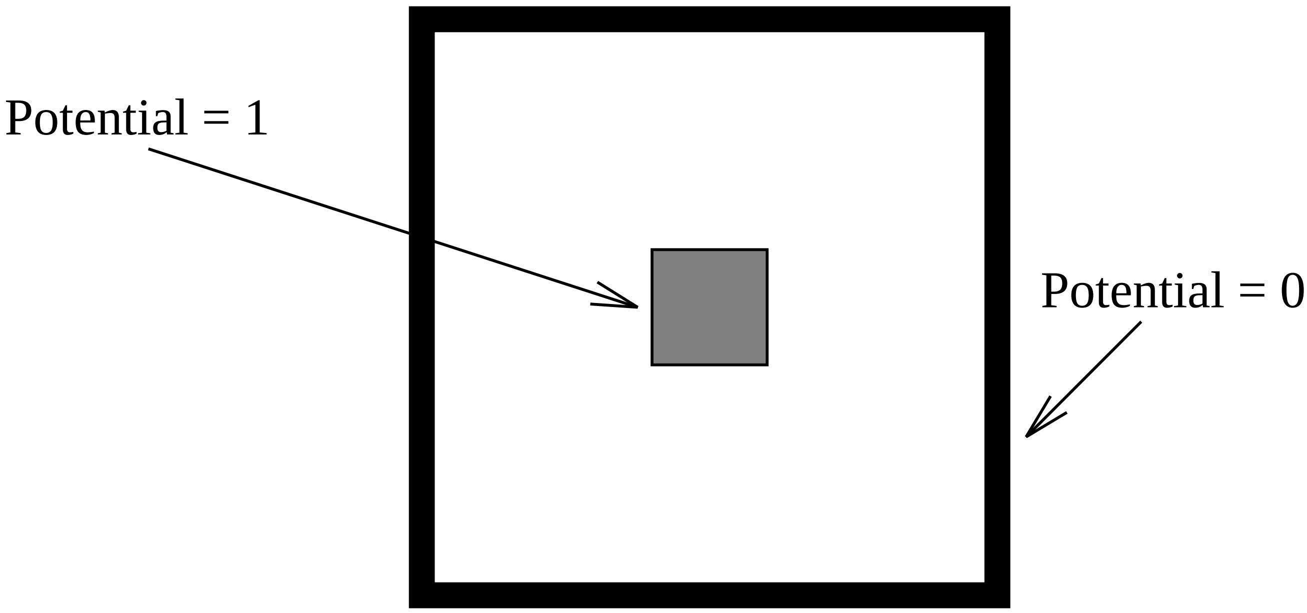
  where N is the number of process , n is the number of points, $g = n/N$ is the grain size, and $\tau = t_{\mathrm{comm}}/t_{\mathrm{calc}}$.
- The efficiency is

$$\epsilon(N) = \frac{1}{1+\tau/(3g)}$$

  and the overhead is $f = \tau/(3g)$.
- Since the efficiency depends on g but not independently on N the parallel algorithm is perfectly scalable.

Laplace Equation Problem

- The next problem we shall look at may be used to determine the electric field around a conducting object held at a fixed electrical potential inside a box also at a fixed electrical potential.
- As with the vibrating string problem, this problem can also be expressed mathematically as a partial differential equation known as the Laplace equation.
- We shall design a parallel MPI program to solve the partial differential equation.



Laplace Equation

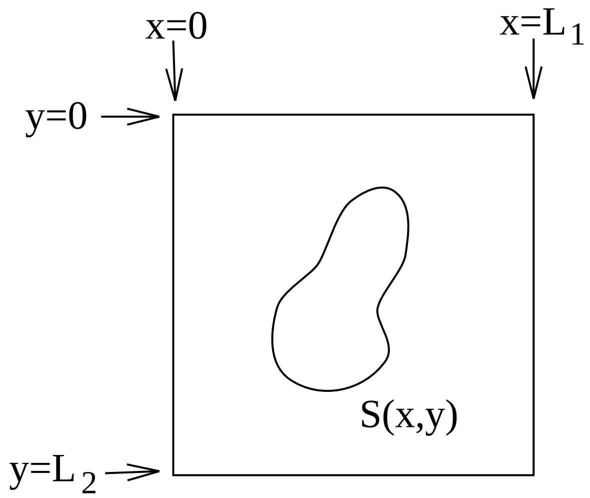
Mathematically the problem of determining the electric field in the box is described by the two-dimensional Laplace equation.

$$\frac{\partial^2 \phi}{\partial x^2} + \frac{\partial^2 \phi}{\partial y^2} = 0$$

subject to the boundary conditions

$$\begin{aligned} \phi(0, y) = \phi(L_1, y) &= 0 \\ \phi(x, 0) = \phi(x, L_2) &= 0 \end{aligned}$$

and $\phi(x, y) = 1$ on $S(x, y)$, where $S(x, y)$ gives the shape of the object inside the box.

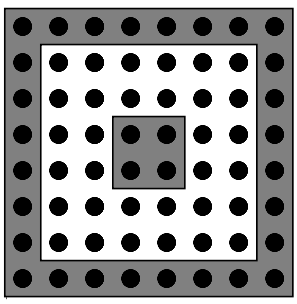


Numerical Approximation

- We solve the problem numerically by approximating the solution at equally-spaced points on the rectangle.
- For a square problem $L_1 = L_2 = L$ we find the approximate solution at n^2 points lying in a regular $n \times n$ grid. The x values are denoted by $x_0, x_1, \ldots, x_{n-1}$, and the y values by $y_0, y_1, \ldots, y_{n-1}$.
- The solution is found using an algorithm called *Jacobi iteration*. Starting with an initial approximation we refine the approximation in a series of steps as follows

$$\phi_{i,j}^{k} = \frac{1}{4}(\phi_{i-1,j}^{k-1} + \phi_{i+1,j}^{k-1} + \phi_{i,j-1}^{k-1} + \phi_{i,j+1}^{k-1})$$

  where $\phi(x_i, y_j)$ at iteration step k is denoted by $\phi_{i,j}^{k}$.
- We keep iterating until the solution has converged.



Sequential Code

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#define NXMAX 32
#define NYMAX 32
main()
{
    double phi[NYMAX][NXMAX];
    double oldphi[NYMAX][NXMAX];
    int    mask[NYMAX][NXMAX];
    int    i, j, k;
    int    nptsx = 32, nptsy = 32;
    int    nsteps  = 20;

    setup_grid (phi, nptsx, nptsy, mask);

    for(k=1;k<=nsteps;k++){
       for(j=0;j<nptsy;j++)
          for(i=0;i<nptsx;i++)
             oldphi[j][i] = phi[j][i];
       for(j=0;j<nptsy;j++)
          for(i=0;i<nptsx;i++)
             if (mask[j][i]) phi[j][i] = 0.25*(oldphi[j][i-1] +
                oldphi[j][i+1] + oldphi[j-1][i] + oldphi[j+1][i]);
    }

    for(j=0,k=0;j<nptsy;j++)
       for(i=0;i<nptsx;i++,k++)
          printf("%9.6f%c",phi[j][i],((k+1)%8==0) ? '\n' : ' ');

    exit(0);
}
```

Initialising the Arrays

```
setup_grid (phi, nptsx, nptsy, mask)
double  phi[NYMAX][NXMAX];
int     nptsx, nptsy;
int     mask[NYMAX][NYMAX];
{
    int i, j;

    for(j=0;j<nptsy;j++)
       for(i=0;i<nptsx;i++){
          phi[j][i]  = 0.0;
          mask[j][i] = 1;
       }

    for(i=0;i<nptsx;i++) {
       mask[0][i]       = 0;
       mask[nptsy-1][i] = 0;
    }

    for(j=0;j<nptsy;j++) {
       mask[j][0]       = 0;
       mask[j][nptsx-1] = 0;
    }

    mask[nptsy/2][nptsx/2]      = 0;
    mask[nptsy/2][nptsx/2-1]    = 0;
    mask[nptsy/2-1][nptsx/2]    = 0;
    mask[nptsy/2-1][nptsx/2-1] = 0;
    phi[nptsy/2][nptsx/2]       = 1.0;
    phi[nptsy/2][nptsx/2-1]     = 1.0;
    phi[nptsy/2-1][nptsx/2]     = 1.0;
    phi[nptsy/2-1][nptsx/2-1]   = 1.0;
}
```

Initial Values of Arrays

The routine `setup_grid` initialises the arrays `phi` and `mask` as follows

0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	1	1	0	0	0
0	0	0	1	1	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

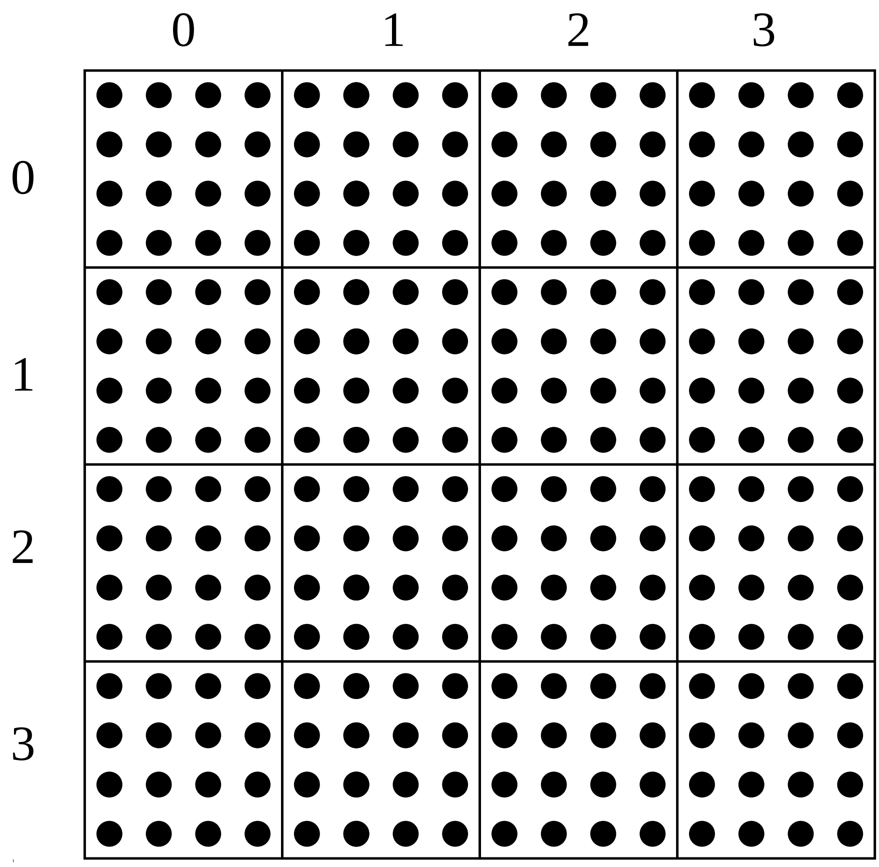
(a) phi

0	0	0	0	0	0	0	0
0	1	1	1	1	1	1	0
0	1	1	1	1	1	1	0
0	1	1	0	0	1	1	0
0	1	1	0	0	1	1	0
0	1	1	1	1	1	1	0
0	1	1	1	1	1	1	0
0	0	0	0	0	0	0	0

(b) mask

Data Distribution

- Give each process a 2D block of points.
- Each process should have approximately the same number of points to ensure good load balance.
- Use a MPI's topology routines to map each block of points to a process.

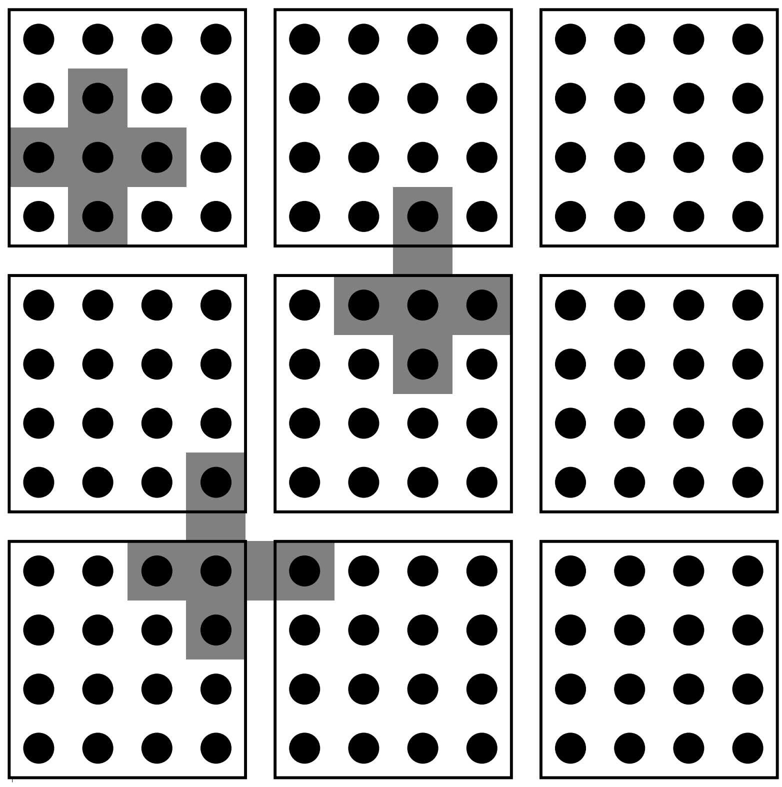


Communication Requirements

The update formula is

$$\phi_{i,j}^{k} = \frac{1}{4}(\phi_{i-1,j}^{k-1} + \phi_{i+1,j}^{k-1} + \phi_{i,j-1}^{k-1} + \phi_{i,j+1}^{k-1})$$

so to update a point we take the average of the 4 neighbouring points from the previous iteration. Points lying along the boundary of a process need data from neighbouring processes.



Each process needs to communicate the points lying along its boundary before performing an update.

Outline of Parallel Code

The main phases of the parallel version of the code are as follows.

- **Initialise data distribution**
  - Find position of each process to determine which block of points it handles.
  - Find out the node numbers of processes in the left, right, up, and down directions.
- **Initialise arrays**
  - Determine how many points each process handles.
  - Set the `phi` and `mask` arrays
- **Perform Update**
  - Copy `phi` array to `oldphi` array.
  - Communicate boundary points.
  - Do update locally
- **Output results**

Array Declarations

Each process needs to be able to store the boundary values received from its neighbours. These are stored in rows 0 and **`local_nptsy+1`** and in columns 0 and **`local_nptsx+1`** of the **`phi`** array.

0,0	0,1	0,2	0,3	0,4	0,5	0,6
1,0	1,1	1,2	1,3	1,4	1,5	1,6
2,0	2,1	2,2	2,3	2,4	2,5	2,6
3,0	3,1	3,2	3,3	3,4	3,5	3,6
4,0	4,1	4,2	4,3	4,4	4,5	4,6
5,0	5,1	5,2	5,3	5,4	5,5	5,6
6,0	6,1	6,2	6,3	6,4	6,5	6,6

local_nptsx = 5
local_nptsy = 5

Thus, the arrays are declared as

```
double phi[NYMAX+2][NXMAX+2]
double oldphi[NYMAX+2][NXMAX+2]
int    mask[NYMAX+2][NXMAX+2]
```

Initialising the Data Distribution

```
/* Find rank and number of processes */
MPI_Comm_size (MPI_COMM_WORLD, &nprocs);
MPI_Comm_rank (MPI_COMM_WORLD, &rank);

/* Work out number of processes in each direction */
dims[0] = dims[1] = 0;
MPI_Dims_create (nprocs, 2, dims);
npy = dims[0];
npx = dims[1];

/* Set up 2D topology */
periods[0] = periods[1] = 0;
MPI_Cart_create (MPI_COMM_WORLD, 2, dims, periods,
                 1, &new_comm);
MPI_Cart_coords (new_comm, rank, 2, coords);
myposy = coords[0];
myposx = coords[1];

/* Determine neighbouring processes for communication */
MPI_Cart_shift  (new_comm, 0, -1,  &down,   &up);
MPI_Cart_shift  (new_comm, 1, -1, &right, &left);
```

Initialising the Data Distribution

- `npx` and `npy` are the number of processes in the process grid in each direction. We make the grid as square as possible.
- This time we set up a 2D communicator, `new_comm`.
- `MPI_Cart_coords` gives the position in the topology of each process.
- Calls to `MPI_Cart_shift` give the ranks of the neighbouring processes in the four directions.

Initialisation of Arrays

Initialisation of the `phi` and `mask` arrays involves the following.

1. Set all of `phi` to 0, and all of `mask` to 1.
2. For processes in row 0 of the process mesh we must set row 1 of the `mask` array to 0.
3. For processes in the last row of the process mesh we must set row `local_nptsy` of the `mask` array to 0.
4. For processes in column 0 of the process mesh we must set column 1 of the `mask` array to 0.
5. For processes in the last column of the process mesh we must set row `local_nptsx` of the `mask` array to 0.
6. For the 4 points in the centre we must set the `phi` and `mask` entries to 1 and 0, respectively, in the processes containing them.

Initialisation of Arrays

```
local_nptsx = nptsx/npx;
local_nptsy = nptsy/npy;

for(j=0;j<=local_nptsy+1;j++)
   for(i=0;i<=local_nptsx+1;i++){
      phi[j][i]  = 0.0;
      mask[j][i] = 1;
   }

if (myposy == 0)
   for(i=0;i<=local_nptsx+1;i++) mask[1][i] = 0;

if (myposy == npy-1)
   for(i=0;i<=local_nptsx+1;i++) mask[local_nptsy][i] = 0;

if (myposx == 0)
   for(j=0;j<=local_nptsy+1;j++) mask[j][1] = 0;

if (myposx == npx-1)
   for(j=0;j<=local_nptsy+1;j++) mask[j][local_nptsx] = 0;

for(j=1;j<=local_nptsy;j++){
   global_y = local_nptsy*myposy + j - 1;
   if (global_y == nptsy/2 || global_y == nptsy/2-1){
      for(i=1;i<=local_nptsx;i++){
         global_x = local_nptsx*myposx + i - 1;
         if (global_x == nptsx/2 || global_x == nptsx/2-1){
            mask[j][i] = 0;
            phi[j][i]  = 1.0;
         }
      }
   }
}
```

Update Phase

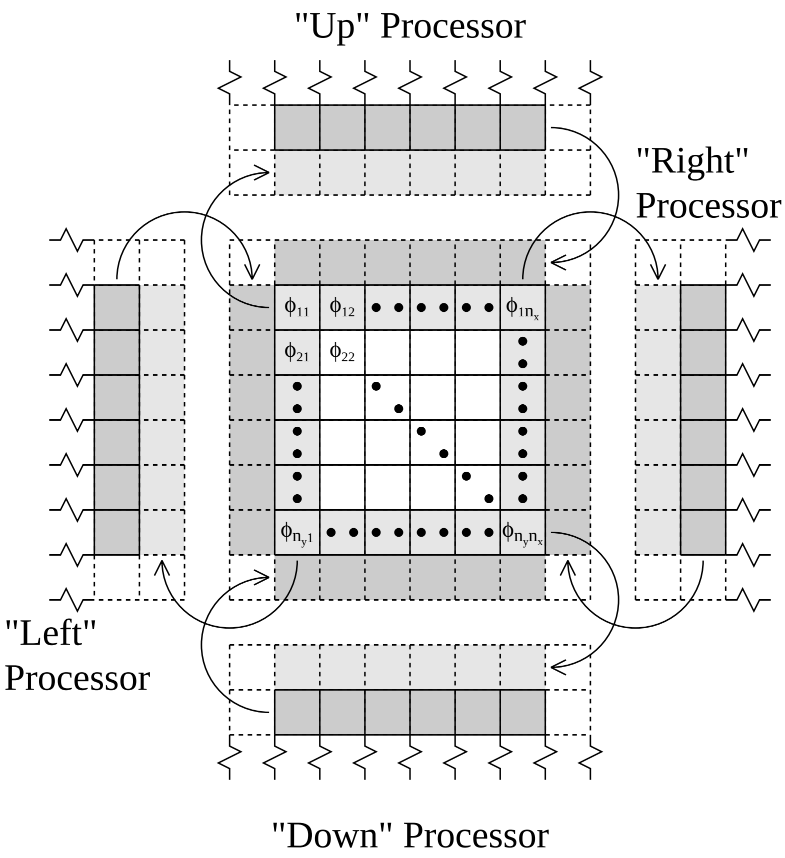
The update phase has three main parts.

1. Copy **`phi`** to **`oldphi`** array.
2. Communicate boundary data.
3. Update points locally.

```
for(k=1;k<=nsteps;k++){
   for(j=1;j<=local_nptsy;j++)
      for(i=1;i<=local_nptsx;i++)
         oldphi[j][i] = phi[j][i];
   MPI_Sendrecv (&oldphi[1][1],  local_nptsx, MPI_DOUBLE,  up, 111,
      &oldphi[local_nptsy+1][1], local_nptsx, MPI_DOUBLE,down, 111,
        new_comm, &status);
   MPI_Sendrecv (&oldphi[local_nptsy][1],local_nptsx,MPI_DOUBLE,down,112,
      &oldphi[0][1], local_nptsx, MPI_DOUBLE, up, 112,
        new_comm, &status);
   for(i=1;i<=local_nptsy;i++) sbuf[i-1] = oldphi[i][1];
   MPI_Sendrecv (sbuf, local_nptsy, MPI_DOUBLE, left, 113,
                 rbuf, local_nptsy, MPI_DOUBLE, right, 113,
                 new_comm, &status);
   for(i=1;i<=local_nptsy;i++) oldphi[i][local_nptsx+1] = rbuf[i-1];
   for(i=1;i<=local_nptsy;i++) sbuf[i-1] = oldphi[i][local_nptsx];
   MPI_Sendrecv (sbuf, local_nptsy, MPI_DOUBLE, right, 114,
                 rbuf, local_nptsy, MPI_DOUBLE, left, 114,
                 new_comm, &status);
   for(i=1;i<=local_nptsy;i++) oldphi[i][0] = rbuf[i-1];
   for(j=1;j<=local_nptsy;j++)
      for(i=1;i<=local_nptsx;i++)
         if (mask[j][i]) phi[j][i] = 0.25*(oldphi[j][i-1] +
            oldphi[j][i+1] + oldphi[j-1][i] + oldphi[j+1][i]);
}
```

Communication

- Communication takes place by shifting data in each of the four directions (left, right, up, and down).
- Before communicating in the left and right direction we must explicitly buffer the data to be sent, and unpack it when it is received.



Output

- As with the vibrating string example, we want the output to be formatted the same in the sequential and parallel programs. We do this by having process 0 do all the output.
- We output the **phi** array one row at a time. The outer two for loops below pick out the rows of **phi**.

```
for(m=0;m<npy;m++)
   for(j=1;j<=local_nptsy;j++)
      for(n=0;n<npx;n++){
         if (myposx == n && myposy == m)
            if (rank!=0) MPI_Send (&phi[j][1], local_nptsx,
               MPI_DOUBLE, 0, 115,MPI_COMM_WORLD);
            else
               for(i=1;i<=local_nptsx;i++,k++)
                  printf("%9.6f%c",phi[j][i],
                               ((k+1)%8==0) ? '\n' : ' ');
         else if (rank==0){
            coords[0] = m;
            coords[1] = n;
            MPI_Cart_rank (new_comm, coords, &source);
            MPI_Recv (&phi[1][1], local_nptsx, MPI_DOUBLE,
                      source, 115, MPI_COMM_WORLD, &status);
            for(i=1;i<=local_nptsx;i++,k++)
               printf("%9.6f%c",phi[1][i],
                               ((k+1)%8==0) ? '\n' : ' ');
         }
      }
```

Performance Analysis

- The update formula is
$$\phi^k_{i,j} = \frac{1}{4}(\phi^{k-1}_{i-1,j} + \phi^{k-1}_{i+1,j} + \phi^{k-1}_{i,j-1} + \phi^{k-1}_{i,j+1})$$
which requires 4 floating-point operations per grid point.
- The number of grid points per processor shifted in the left/right direction is n/P, where $n \times n$ is the size of the grid and P is the number of processors in the y direction of the processor mesh.
- The number of grid points per processor shifted in the up/down direction is n/Q, where Q is the number of processors in the x direction of the processor mesh.

Performance Analysis

- Speed-up is

$$
\begin{aligned}
S(N) &= \frac{4n^2 t_{\rm calc}}{(4n^2/N)t_{\rm calc} + (2n/Q)t_{\rm shift} + (2n/P)t_{\rm shift}} \\
&= \frac{N}{1 + (P+Q)\tau/(2n)} \\
&= \frac{N}{1 + (Q/n)(1+\alpha)\tau/2}
\end{aligned}
$$

  where $M = n \times n$ is the size of the grid, $P \times Q$ is the processor mesh, $P = \alpha Q$, and $\tau = t_{\rm shift}/t_{\rm calc}$.

- Since $N = PQ = \alpha Q^2$ and $M = n^2$ the efficiency is

$$
\begin{aligned}
\epsilon(N) &= \frac{1}{1 + (1+\alpha)/(2\sqrt{\alpha})(\sqrt{N/M})\tau} \\
&= \frac{1}{1 + (1+\alpha)/(2\sqrt{\alpha})(\tau/\sqrt{g})}
\end{aligned}
$$

  where $g = M/N$ is the grain size, and the overhead is $f = (1+\alpha)/(2\sqrt{\alpha})(\tau/\sqrt{g})$.

- Since the efficiency depends only on g, and not independently on n and N, the parallel algorithm is perfectly scalable.

Irregular Communication

- In the wave equation and Laplace equation problems the communication is very *regular*. Once we set the number of processes and the size of the problem the communication requirements of the algorithm are fully determined.
- We shall now consider a parallel molecular dynamics simulation. In this simulation we know that data may need to be communicated between processes at a particular point in the program, but we do not know which data it will be. In this example the communication is slightly *irregular*.

Molecular Dynamics Simulations

- We have n particles in a periodic square domain.
- The particles interact in a known pairwise way. Each particle exerts a force on the other particles so that
  - if the particles are close enough they repel each other
  - if the particles are far enough apart they are attracted to each other
  - if the particles are more than some distance, r_0, apart they do not influence each other.
- This sort of interaction is typical of many molecules.
- Given initial positions and velocities for the particles, we follow the movement of the particles at a series of discrete time steps.

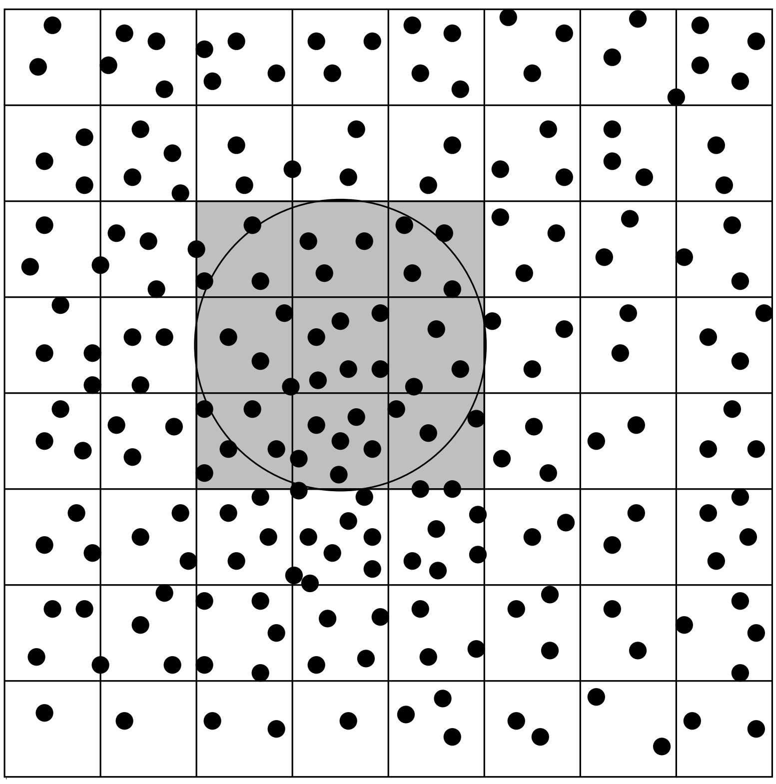
Cutoff Distance

- We could find the force on particle i by summing over all the other particles,

$$\mathbf{F}_i = \sum_{j=0}^{n-1} \mathbf{f}_{ij}$$

  where $\mathbf{f}_{ij}$ is the force exerted by particle j on particle i. This results in an $O(n^2)$ algorithm.

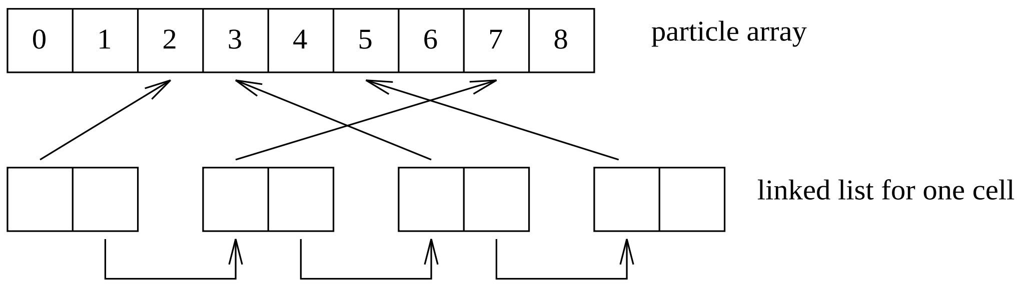
- We can improve the running time if we make use of the fact that the force $\mathbf{f}_{ij}$ is zero for particles more than distance r_0 apart.



- If we divide the domain of the problem into cells of size $r_0 \times r_0$ each particle only interacts with the particles in its own cell and the 8 neighbouring cells.

Linked Lists

- The linked list for a particular cell tells us what particles are in that cell.



- Each cell has its own list. The starts of the lists are stored in a 2D array.
- A particle can find out which cell it is in from its position.

Data Structures

- Each particle has a position and velocity, so the particle data can be stored in an array of structures.

```
struct particle {
   double position[2];
   double velocity[2];
} Particles[MAX_PARTICLES];
```

- We need to be able to access the particles in a particular cell. We could do this by associating a vector with each cell to store the indices of the particles in that cell. But different cells contain different numbers of particles so memory may be wasted.
- Instead we store pointers to the particles in a particular cell in a linked list.
- We need a separate linked list for each cell.

```
struct list {
   struct particle *part;
   struct list     *list;
} *Lists[MAX_CELLY][MAX_CELLX];
```

- This is called *geometric hashing*.

Outline of Sequential Code

1. **Initialise arrays**.
   (a) initialise particle positions and velocities
   (b) initialise the 2D array of cell lists by inserting particles into the linked lists
2. **Update loop**.

   For each time step

   (a) find force on each particle
   (b) move particle under influence of the force exerted on it and update particle positions and velocities
3. **Output**

   Perform whatever output is required.

The Update Loop

At each time step we do two main things

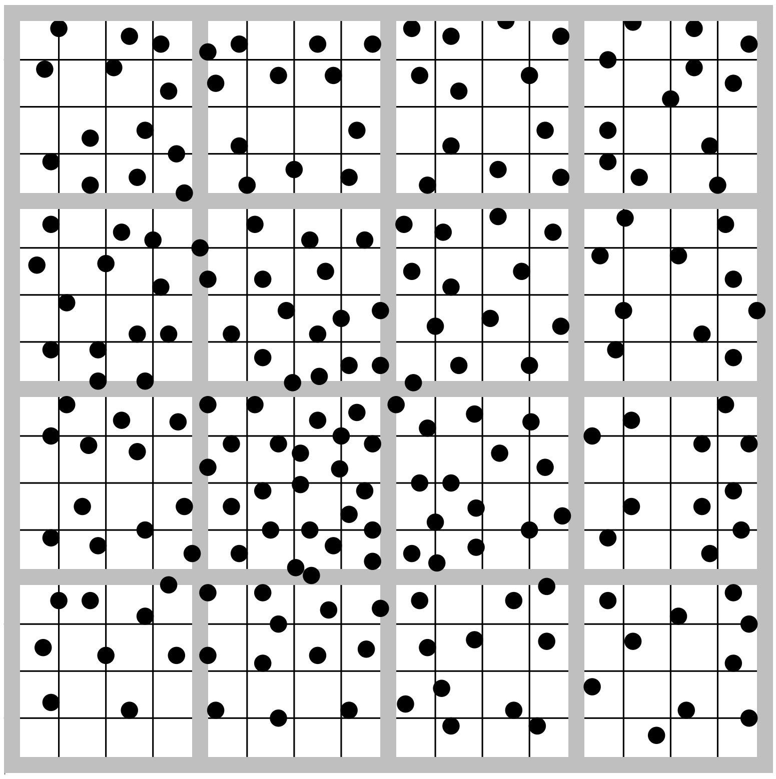
1. Find the force on each particle
2. Move each particle according to the force on it

The pseudocode for the update loop is as follows.

```
for (each time step)
   for (each particle, p)
      find out the location (i,j) of cell p is in
      set force[p] = 0
      for (cell (i,j) and the 8 neighbouring cells)
         for (each particle q in cell)
             add force of q on p to force[p]
         end for
      end for
   end for
   for (each particle, p)
      update velocity and position of p using force[p]
   end for
end for
```

Data Distribution

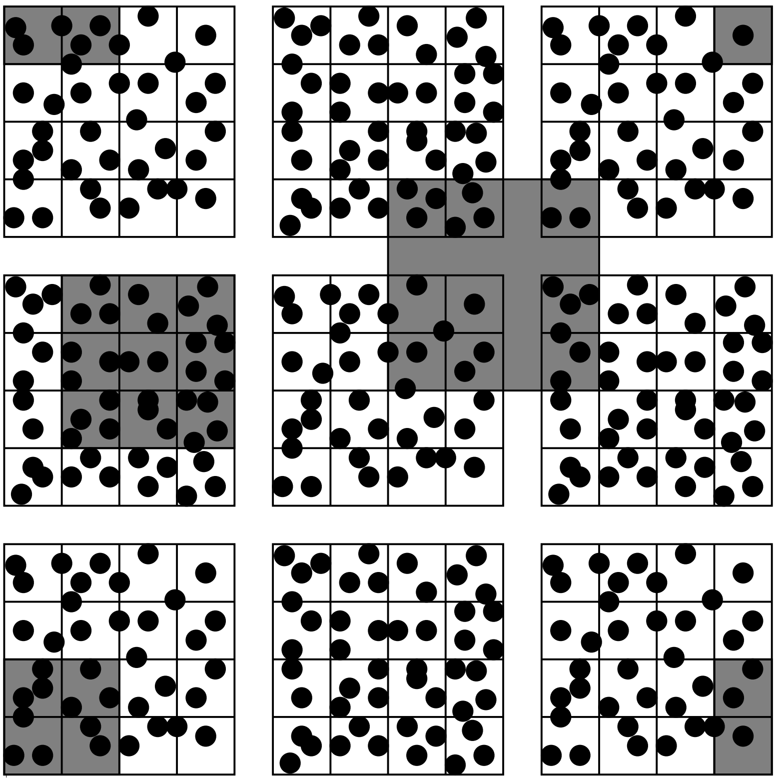
- The particles are distributed to processes by assigning a rectangular block of cells to each process.
- We find out the node number, location in the process mesh, and the node numbers of the neighbouring processes as in the Laplace equation problem.



Data Dependencies

Particles in cells along the boundary of a process need to know about particles in other processes in order to evaluate the force on them.

Each process needs particle data from 8 neighbouring processes.



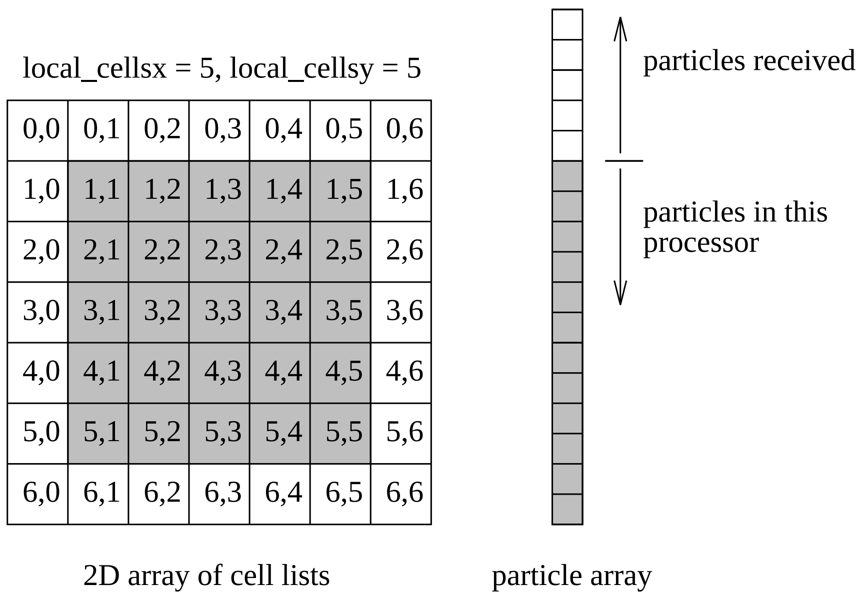
Communication Requirements

- Each particle needs information about the particles in the neighbouring cells in order to determine the force on it. So we need to communicate particles lying in cells along the boundary of each process
- When particles move they may travel from the set of cells owned by one process to those of another process. This is called *particle migration* and requires communication.

Array Declarations

- We maintain a 2D array of linked lists – one for each cell in a process.
- When we receive particle information from cells lying along the boundary of adjacent processes we store the data at the end of the particle array.
- Pointers to the particles received are placed into cell lists.

```
struct list {
   struct particle *part;
   struct list     *list;
} *Lists[MAX_CELLY+2][MAX_CELLX+2];
```



Parallel Update Loop

In the parallel algorithm there are two communication phases — one for boundary cell data and one for particle migration.

```
for (each time step)
   communicate particle data for boundary cells
   for (each particle, p)
      find out the location (i,j) of cell p is in
      set force[p] = 0
      for (cell (i,j) and the neighbouring cells)
         for (each particle q in cell)
             add force of q on p to force[p]
         end for
      end for
   end for
   for (each particle, p)
      update velocity and position of p using force[p]
   end for
   migrate particles
end for
```

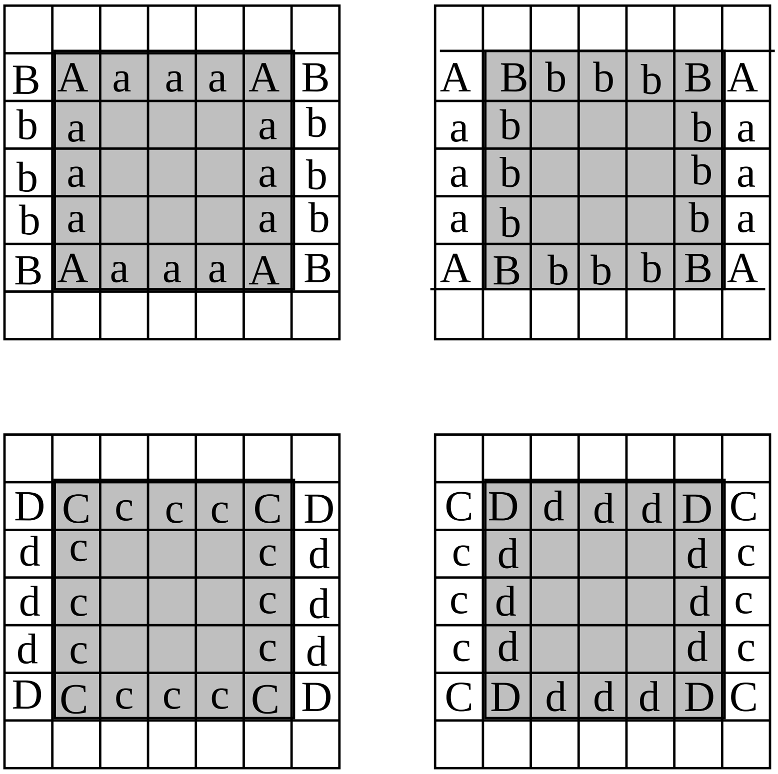
Communication of Boundary Cell Data

The communication is similar to that for the Laplace equation problem except

1. We have to look in the cell lists to see which particles have to be sent and pack this information into a send buffer.
2. The receiving process does not know beforehand how many particles it is going to receive.
3. We have to communicate between diagonally adjacent processes.

Left and Right Shifts

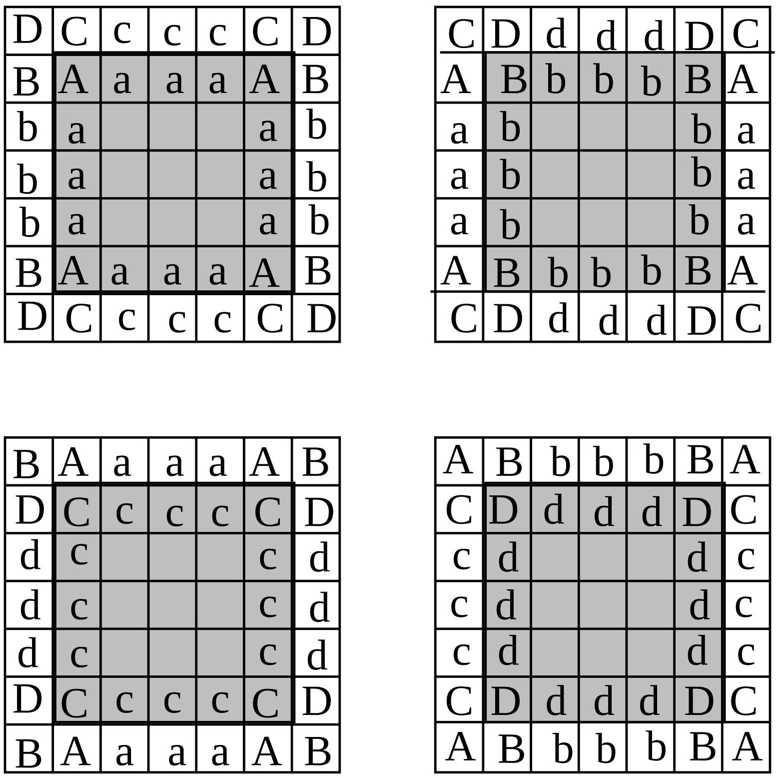
We need to communicate particles in corner cells to diagonally adjacent processes. This can be done be done in 4 shift operations.



Shift left, then shift right

Up and Down Shifts

To ensure the corner cell data are communicated correctly we shift complete rows of cells indexed from 0 to `local_ncellsx`.



Shift up, then down

Pseudocode for Left Shift

```
nsend = 0
for (i=1 to local_ncellsy)
   for (each particle, p, in cell (i,1))
      pack position of p into sbuf
      nsend = nsend + 2
   end for
end for
MPI_Sendrecv (sbuf, nsend, MPI_DOUBLE, left, a_tag,
              rbuf, max_size, MPI_DOUBLE, right, a_tag,
              new_comm, &status);
MPI_Get_count (&status, MPI_DOUBLE, &nrecv);
for (i=1 to nrecv)
   take next 2 numbers from rbuf, store in x and y
   set position of particle npart+i-1 to (x,y)
   find out which cell (x,y) is in
   add particle npart+i-1 to list for that cell
end for
```

Pseudocode for Up Shift

```
nsend = 0
for (i=0 to local_ncellsx+1)
   for (each particle, p, in cell (1,i))
      pack position of p into sbuf
      nsend = nsend + 2
   end for
end for
MPI_Sendrecv (sbuf, nsend, MPI_DOUBLE, up, a_tag,
              rbuf, max_size, MPI_DOUBLE, down, a_tag,
              new_comm, &status);
MPI_Get_count (&status, MPI_DOUBLE, &nrecv);
for (i=1 to nrecv)
   take next 2 numbers from rbuf, store in x and y
   set position of particle npart+i-1 to (x,y)
   find out which cell (x,y) is in
   add particle npart+i-1 to list for that cell
end for
```

Particle Migration

We have to be able to handle the case where particles move to diagonally adjacent processes. The pseudocode to update particle positions and velocities and to migrate them if needed to new processes is as follows.

```
for (each particle, p)
   update position and velocity
   determine which cell p is in
   if (p has moved to new cell)
      delete p from list of old cell
      if (p has moved to different process)
         put p into appropriate communication buffer
         remove p from particle array
      else
         add p to list of new cell
      end if
   end if
end for
shift left
shift right
shift up
shift down
```

Shifting in Particle Migration

After receiving particle data from another process, a process must determine if the particle belongs to it, or if it has to be passed on to another process

```
MPI_Sendrecv (leftbuf, nsend_left, MPI_DOUBLE, left, a_tag,
              rbuf, max_size, MPI_DOUBLE, right, a_tag,
              new_comm, &status);
MPI_Get_count (&status, MPI_DOUBLE, &nrecv);
for (i=1 to nrecv)
   get next 4 numbers from rbuf, store in x, y, vx, vy
   if (particle belongs in this process)
      add particle to end of particle array
      find out what cell particle is in
      add particle to list for that cell
   else
      put particle in appropriate communication buffer
   end if
end for
```

Notes on Parallel Molecular Dynamics Simulations

- As in previous examples we need to exchange "boundary" data between processes.
- Need to communicate with diagonally adjacent processes. This can be done with four shift operations.
- We do not know beforehand how many particles will be communicated between processas in the boundary data exchange or migration steps. The receiving process determines this from the number of bytes received.
- In general each process holds a different number of particles and this changes over time. However, we don't expect load imbalance to be too bad because particles tend to be evenly distributed in space.

WaTor — A Dynamical System

- We shall now look at a very dynamic simulation called WaTor. WaTor is a periodic 2D dimensional ocean (hence, *Wa*tery *Tor*us) in which sharks and fish compete and survive.
- The parallel implementation of WaTor has a number of interesting features.
  1. A very inhomogeneous and dynamic load distribution
  2. The need for irregular communication
  3. The possibility of conflicts between updates performed by different processors.
- Other advanced parallel applications share some of these features.

The Rules of WaTor

- WaTor takes place on a periodic grid. Each grid cell either contains a fish, a shark, or is empty.
- The grid is initially populated by a specified number of fish and sharks. The populations then evolve in a series of discrete time steps according to certain rules.

Fish Rules

- **Moving**

  In each time step, each fish notes which of the 4 neighbouring sites are empty. One of these empty sites is chosen at random and the fish moves there. If there are no empty neighbouring sites the fish stays where it is.

- **Breeding**

  If a fish is past the fish breeding age, then when it moves it breeds, leaving a fish of age zero at its previous location. A fish cannot breed if it doesn't move.

- **Eating**

  Fish eat plankton available throughout the ocean. Fish never starve.

Rules for Sharks

- **Moving**

  In each time step, each shark notes which of the 4 neighbouring sites are occupied by fish. One of these sites is chosen at random and the shark moves there, eating the fish. If there are no neighbouring sites containing fish the shark notes which of the 4 neighbouring sites are empty and moves to one of these sites at random. If all 4 neighbouring sites are already occupied by sharks, the shark stays where it is.

- **Breeding**

  If a shark is past the shark breeding age, then when it moves it breeds, leaving a shark of age zero at its previous location. A shark cannot breed if it doesn't move.

- **Eating**

  Sharks eat only fish. If a shark does not eat for more than a certain number of time steps (known as the shark starvation age) then it dies.

Input for WaTor

The input to the simulation is

1. The size of the grid
2. The initial number of sharks and fish (these are placed at random)
3. The shark and fish breeding ages
4. The shark starvation age

Data Structures

The two fundamental data structures are the 2D grid and a list of sharks and fish.

The ocean data structure is:

```
struct ocean {
  int    occupier_type;     // Fish, shark, or empty
  struct swimmer *occupier; // pointer to shark or fish
} Ocean[MAXY][MAXX];
```

Fishes and sharks use the same data structure

```
struct swimmer {
  int type;        // Fish or shark
  int x, y;        // Position in ocean
  int age;
  int last_ate;
  int iteration;   // Time step number
  swimmer *prev;   // Pointer to previous
  swimmer *next;   // Pointer to next
} *List;
```

Outline of Sequential Code

- **Initialise**

  Initialise `ocean` array by placing fish and sharks at random grid locations.

  Initialise fish/shark list

- **Update**

  In each time step, we process the fish and sharks in the order in which they appear in the list.

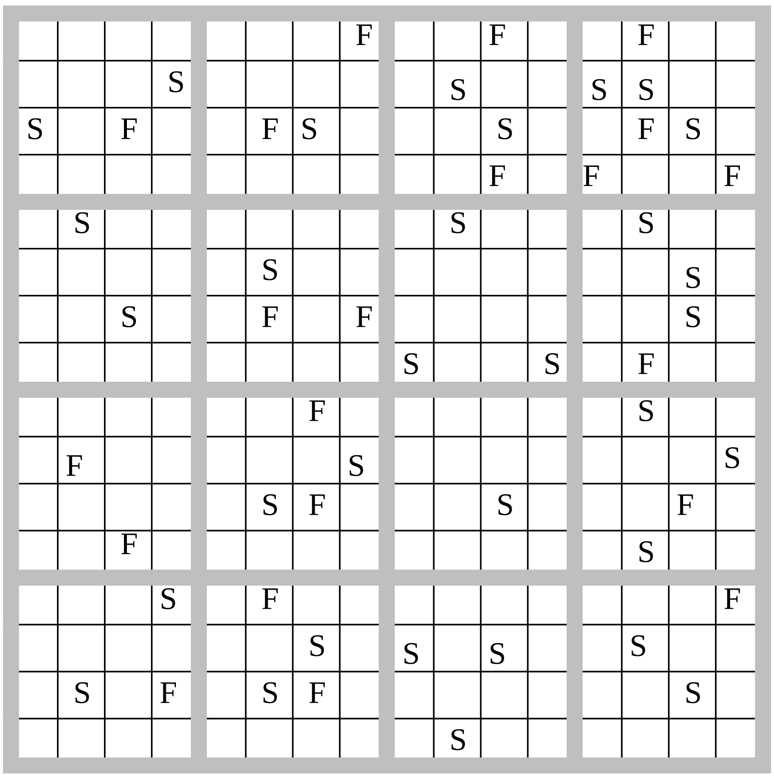
  We may also update the display, or perform some other output, in each time step.

- **Finalise**

  Output final state, statistics, etc.

Data Distribution

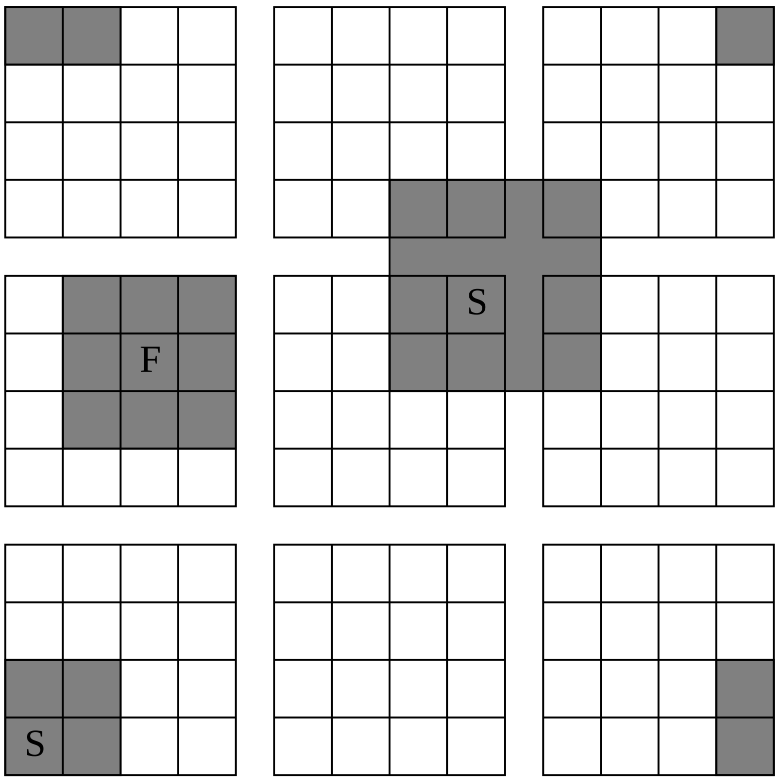
- Initially we shall use a 2D block data distribution, just the same as in the Laplace equation problem and the molecular dynamics simulation.
- Each processor looks after a block of the ocean and all the fish and sharks in it.
- We need to know the node number, location in the processor mesh, and the node numbers of the neighbouring processors.



Data Dependencies

Fish and sharks lying along the boundary of a processor need to know about the grid cells lying along the boundary of other processors in order to follow the WaTor rules

Each processor needs data from 8 neighbouring processors.



Array Declarations

- When we receive fish and shark data from cells lying along the boundary of adjacent processors we insert the data into the linked list and point to it from the ocean array.

```
struct ocean {
   int occupier_type;
   struct swimmer *occupier;
} Ocean[MAXY+2][MAXX+2];
```

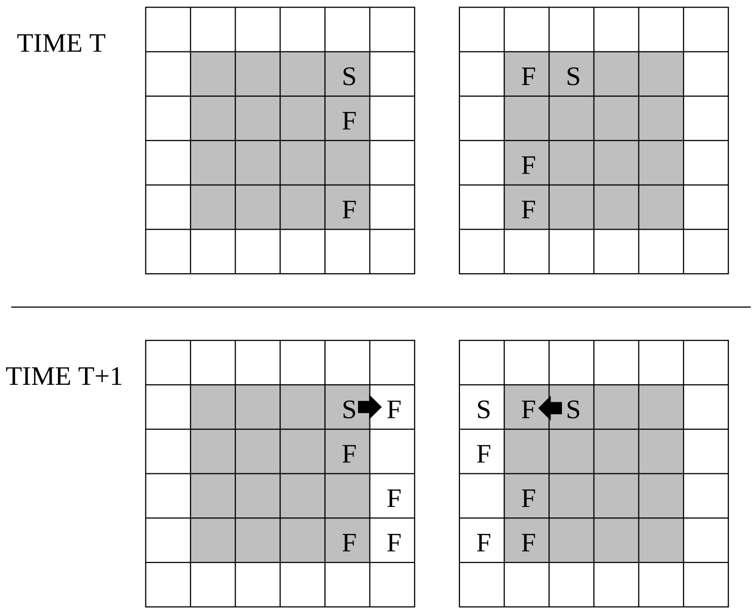
local_nptsx = 5, local_nptsy = 5

0,0	0,1	0,2	0,3	0,4	0,5	0,6
1,0	1,1	1,2	1,3	1,4	1,5	1,6
2,0	2,1	2,2	2,3	2,4	2,5	2,6
3,0	3,1	3,2	3,3	3,4	3,5	3,6
4,0	4,1	4,2	4,3	4,4	4,5	4,6
5,0	5,1	5,2	5,3	5,4	5,5	5,6
6,0	6,1	6,2	6,3	6,4	6,5	6,6

2D ocean grid array

Potential for Conflict

- In the course of an update a fish or shark may move into a "border" grid cell. Thus, unlike in the Laplace and molecular dynamics problems, the border cells in a processor may change in an update.
- These changes must be communicated back to the processor that originally send that border strip, to be re-integrated into its data structures.
- However, the processor owning those grid cells may also have updated them, and these updates may be in conflict.



Rollback

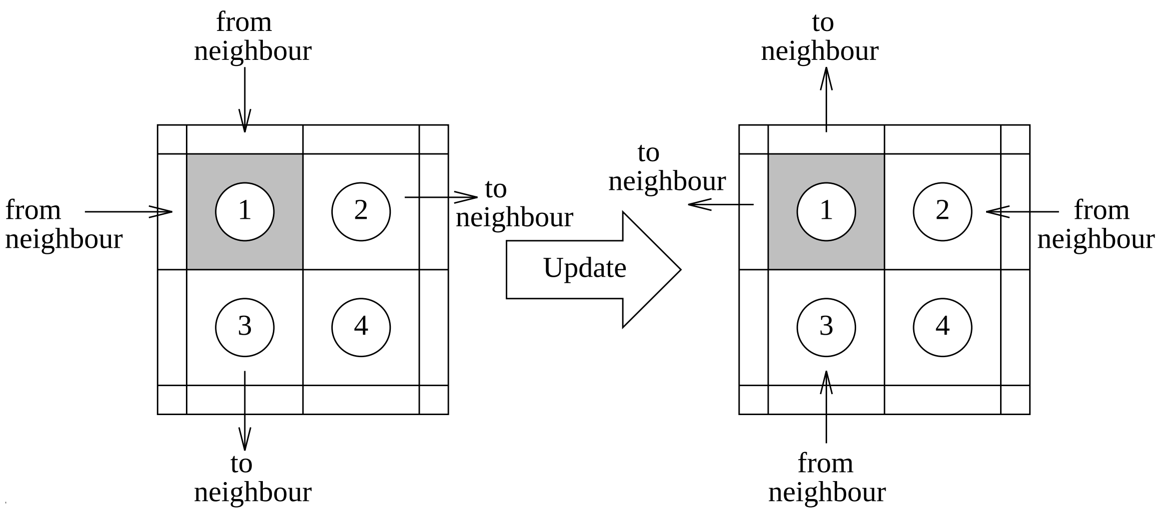
One way to resolve conflicts is called *rollback*, and works as follows.

1. Return the fish or shark that crossed the processor boundary back to its original processor, and place it back in its original position.
2. If that position has been occupied by another fish or shark, then that fish or shark must be rolled back.
3. This rollback process continues until until every fish and shark has a place to go.

Sub-Partitioning

An alternative solution to rollback in resolving conflicts is to avoid them in the first place. This is can be done as follows,

1. Divide each processor's grid into 4 smaller sub-grids, labelled 1, 2, 3, and 4.
2. Exchange the parts of sub-grids 2 and 4 that have data along their boundaries needed to update sub-grid 1 in adjacent processors.
3. Update sub-grid 1 in each processor.
4. Return boundary information to original owner and update data structures.
5. Repeat steps 2, 3, and 4 for each of the other sub-grids in turn.



Outline of Parallel Code

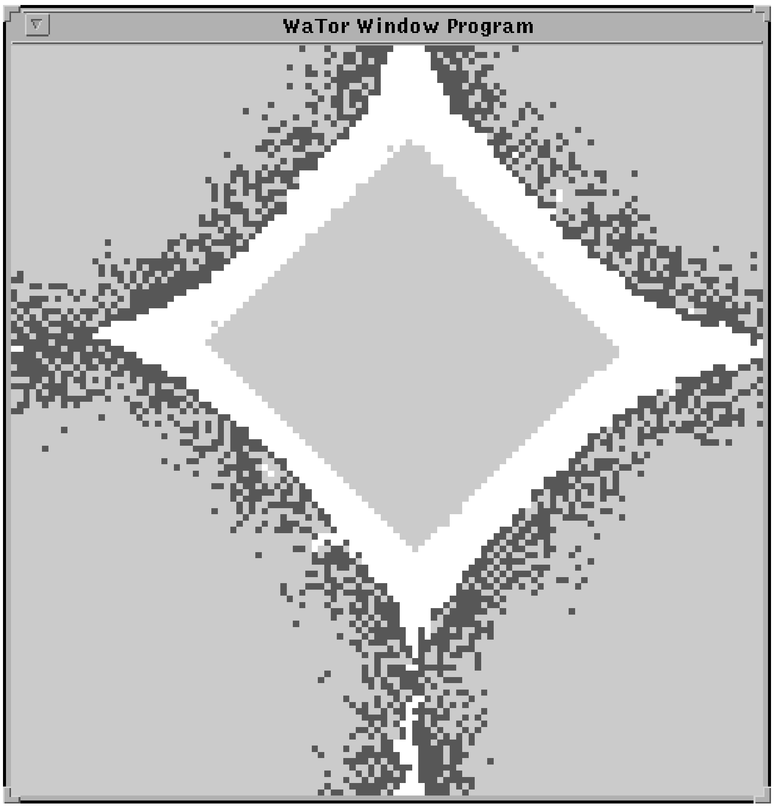
The update loop of the parallel version of WaTor using sub-grids is as follows.

```
for (each time step)
   for (each sub-grid, GI, I=1,2,3,4)
      exchange boundary data along 2 edges of GI
      store data received in border of ocean and in
              fish/shark list
      update fish and sharks in GI
      exchange boundary data, overwriting original
              data with updated data
   end for
end for
```

- Sub-grid 1 communicates with the left and up processors
- Sub-grid 2 communicates with the right and up processors
- Sub-grid 3 communicates with the left and down processors
- Sub-grid 4 communicates with the right and down processors

Load Imbalance in WaTor

- Load balance is an important consideration in WaTor and many other applications.
- In WaTor the workload if generally not evenly distributed over the ocean, so a block data distribution means that some processors have less work than others at certain times.
- Load balance in WaTor changes with time as the fish and sharks move.



Dynamic Load Imbalance

In dealing with dynamic load imbalance the following two approaches are important,

1. Use of a dynamic load balancer so that the distribution of the ocean among the processors changes as the fish and shark system evolves. When dealing with grids some form of *recursive bisection* is often used.

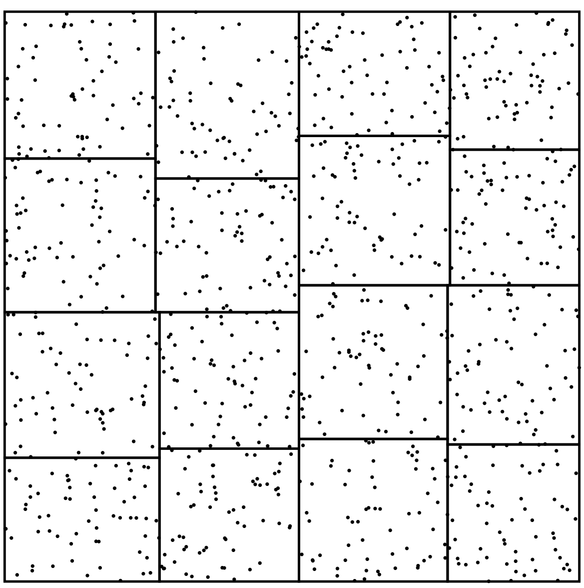
2. Use of a *cyclic*, or *scattered*, data distribution. The parts of the grid assigned to one processor do not form a contiguous block but are scattered in a regular way over the whole domain. The aim in this case is to achieve statistical load balance.

.

Orthogonal Recursive Bisection

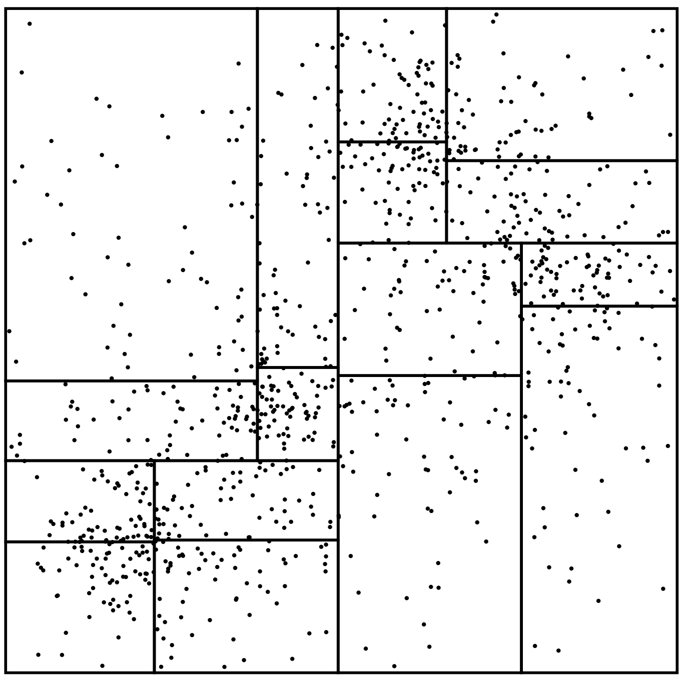
- Orthogonal Recursive Bisection (ORB) first divides the domain orthogonal to the x-direction so there are equal numbers of items in each of the two subdomains.
- Then each of these 2 subdomains is independently divided orthogonal to the y-direction, to give 4 subdomains each with approximately the same number of items in. each
- This process of bisection continues, alternating between the x and y directions, until there is one subdomain for each processor.

ORB is not used when the items are distributed uniformly over the domain – in this case the subdomains would come out about the same size and shape.



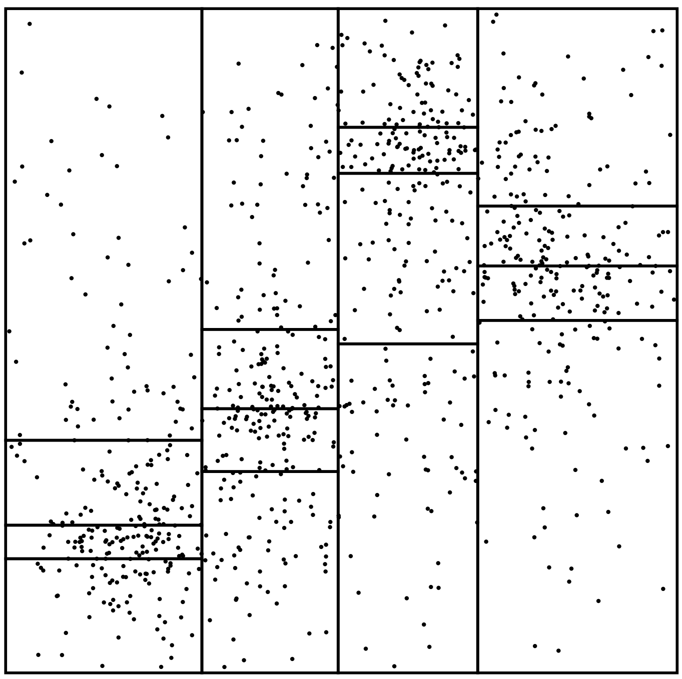
Orthogonal Recursive Bisection

- If the items are distributed unevenly over the domain, ORB can give rise to a variety of different shaped processor subdomains.
- Using a dynamic load balance scheme such as ORB adds to the complexity of the software, particularly in deciding which boundary data must be communicated with which processors.



Hierarchical Recursive Bisection

- HRB is a variation of ORB in which we first make all the cuts in one direction, and then all the cuts in the second direction, rather than alternating directions.
- HRB allows the data distribution to be adjusted over just one direction, rather than both.
- ORB and HRB can easily be extended to 3 or more dimensions.



Cyclic Data Distributions

- In a cyclic data distribution the data assigned to each processor is scattered in a regular way over the domain of the problem.
- The figure below shows how a grid might be cyclicly distributed over a 4×4 mesh of processors.
- The cyclic distribution is a simple way to improve load balance but can result in more communication as it increases the amount of boundary data in a processor.

0,0	0,1	0,2	0,3	0,0	0,1	0,2	0,3	0,0	0,1
1,0	1,1	1,2	1,3	1,0	1,1	1,2	1,3	1,0	1,1
2,0	2,1	2,2	2,3	2,0	2,1	2,2	2,3	2,0	2,1
3,0	3,1	3,2	3,3	3,0	3,1	3,2	3,3	3,0	3,1
0,0	0,1	0,2	0,3	0,0	0,1	0,2	0,3	0,0	0,1
1,0	1,1	1,2	1,3	1,0	1,1	1,2	1,3	1,0	1,1
2,0	2,1	2,2	2,3	2,0	2,1	2,2	2,3	2,0	2,1
3,0	3,1	3,2	3,3	3,0	3,1	3,2	3,3	3,0	3,1
0,0	0,1	0,2	0,3	0,0	0,1	0,2	0,3	0,0	0,1
1,0	1,1	1,2	1,3	1,0	1,1	1,2	1,3	1,0	1,1

Cyclic Data Distributions

Consider a one-dimensional cyclic data distribution of an array, such as,

0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3

- This is known as a `cyclic[1]` data distribution, and can be regarded as mapping a global index, m, to a processor location, p, and a local index, i.

  $$m \mapsto (p, i)$$

  where p and i are given by

  $$\begin{aligned} p &= m \bmod N \\ i &= \lfloor m/N \rfloor \end{aligned}$$

  and N is the number of processors.

- The inverse mapping is

  $$m = i * N + p$$

- If we arrange array entries in groups of size k and cyclicly distribute these we get a `cyclic[k]` data distribution. For example, the following shows a `cyclic[2]` data distribution.

0	0	1	1	2	2	3	3	0	0	1	1	2	2	3	3	0	0	1	1

Block Data Distributions

- For a one-dimensional block data distribution the mapping of global index, m, to a processor location, p, and a local index, i, is

$$\begin{aligned} p &= \lfloor m/T \rfloor \\ i &= m \bmod T \end{aligned}$$

where $T = \lceil M/N \rceil$, M is the number of items in the array, and N is the number of processors.

0	0	0	0	0	1	1	1	1	1	2	2	2	2	2	3	3	3	3	3

- The inverse mapping is

$$m = p * T + i$$

Block Data Distributions

- Multi-dimensional arrays are distributed by applying the desired data distribution separately to each array index. Thus, for a two-dimensional data distribution the global index (m, n) is mapped so that $m \mapsto (p, i)$ and $n \mapsto (q, j)$, where (p, q) is location on a $P \times Q$ processor mesh and (i, j) is the index into the local 2D array.
- Different data distributions can be applied over each array dimension.
- For a 2D (`cyclic[1], cyclic[1]`) data distribution of a $M \times N$ array over a $P \times Q$ processor mesh we would have

$$\begin{aligned} m \mapsto (p, i) &= (m \bmod P, \lfloor m/P \rfloor) \\ n \mapsto (q, j) &= (n \bmod Q, \lfloor n/Q \rfloor) \end{aligned}$$

0,0	0,1	0,2	0,3	0,0	0,1	0,2	0,3	0,0	0,1
1,0	1,1	1,2	1,3	1,0	1,1	1,2	1,3	1,0	1,1
2,0	2,1	2,2	2,3	2,0	2,1	2,2	2,3	2,0	2,1
3,0	3,1	3,2	3,3	3,0	3,1	3,2	3,3	3,0	3,1
0,0	0,1	0,2	0,3	0,0	0,1	0,2	0,3	0,0	0,1
1,0	1,1	1,2	1,3	1,0	1,1	1,2	1,3	1,0	1,1
2,0	2,1	2,2	2,3	2,0	2,1	2,2	2,3	2,0	2,1
3,0	3,1	3,2	3,3	3,0	3,1	3,2	3,3	3,0	3,1
0,0	0,1	0,2	0,3	0,0	0,1	0,2	0,3	0,0	0,1
1,0	1,1	1,2	1,3	1,0	1,1	1,2	1,3	1,0	1,1

Message Passing Libraries

Message passing libraries fall into two categories:

1. Proprietary libraries designed for just one family of machines. The message passing routines on the nCUBE-2 are of this type. Another example is Intel's NX message passing library.
2. Portable message passing libraries that are intended to be implemented on a wide range of different parallel computers. Recently two have emerged as the most widely used message passing libraries
   (a) PVM (Parallel Virtual Machine). Originally designed to allow clusters of workstations to work together in parallel, but now available on most commercial parallel computers.
   (b) MPI (Message Passing Interface). Designed as a message passing standard by a large group of people.

Parallel Virtual Machine

- PVM enables a collection of different computing systems to be viewed as a single parallel virtual machine
- MIMD process model
- Distributed memory model
- PVM transparently handles message passing, data conversion, and task scheduling
- Tasks (or processes) interact by message passing
- Message passing
  - is done by calls to a message passing library
  - uses system controlled buffers
  - is typed
- Point-to-point and collective message passing operations are provided
- PVM has support for process groups

PVM Components

The main components of PVM are:

- **The PVM daemon**. This runs on every host. It controls the PVM resources on each host and mediates interactions with other hosts.
- **The PVM console**. This is a user interface that allows the user to interactively configure the parallel virtual machine.
- **The PVM group server**. This manages process groups within PVM
- **The PVM library**. A set of routines for managing tasks and process groups, and for performing message passing.

The Parallel Virtual Machine

- PVM runs on a large number of machines ranging from networks of workstations up to massively parallel processors.
- The PVM hosts can be specified by
  - adding hosts via the PVM console
  - a host file command line argument to the PVM console

PVM Processes

- Usually a PVM program starts as a single process that spawns other processes to do useful work

  ```
  numt = pvm_spawn (char *task, char **argv, int flag,
                    char *where, int ntask, int *tids)
  ```

  will start `ntask` processes, each running the excutable designated by the character string contained in `task`. `argv` is a pointer to an array of arguments passed to each executable. `flag` and `where` control exactly how and where the processes are spawned. On return `tids` is an array of task IDs that labels the processes spawned. `numt` is the actual number of processes created.

- This is called the "host-worker" model
- Each PVM process has an integer task ID that uniquely identifies it.
- Spawning process knows all task IDs (in array `tids`).
- Process can find its own task ID thus:

  ```
  mytid = pvm_mytid ()
  ```

- Task IDs are used to identify processes in all communication.

PVM Message Passing

- Task IDs are used to specify the source and destination processes.
- In general, messages are buffered by the PVM system
- On receipt messages are selected by source and message tag. These may be wildcarded
- Data to be sent are packed into the send buffer on the source process, and are unpacked from the receive buffer on the destination process
- Datatypes of message components are specified when packing and unpacking messages permitting representation conversion.

Message Exchange

In this example 2 processes exchange an integer and two real numbers. `i` is a scalar int, and `vec` is an array of floats.

```
tid = pvm_mytid ();
bufid = pvm_initsend (PvmDataDefault);
info  = pvm_pkint (&i, 1, 1);
info  = pvm_pkfloat (vec, 2, 1);
if (tid == task1) {
   info  = pvm_send (task2, tag);
   bufid = pvm_recv (task2, tag);
}
else if (tid == task2) {
   info  = pvm_send (task1, tag);
   bufid = pvm_recv (task1, tag);
}
info = pvm_upkint (&i, 1, 1);
info = pvm_upkfloat (vec, 2, 1);
```

Homogeneous Messages

- PVM provides routines for communicating data of a single data type.

```
tid = pvm_mytid ();
if (tid == task1) {
   info = pvm_psend (task2, tag, vec, 10, PVM_FLOAT);
   info = pvm_precv (task2, tag, vec, 10, PVM_FLOAT,
                     &atid, &atag, &alen);
}
else if (tid == task2) {
   info = pvm_psend (task1, tag, vec, 10, PVM_FLOAT);
   info = pvm_precv (task1, tag, vec, 10, PVM_FLOAT,
                     &atid, &atag, &alen);
}
```

- Avoids need for pack and unpack in simple case.

PVM Integration Example

The master program

```
#include "pvm3.h"
#include "math.h"
#define PI 3.141592654
#define NPMAX 100

main()
{
   int tids[NPMAX], bufid, atid, atag, alen;
   int nprocs, npts, info, nlocal, nbeg, nend;
   int i, info, bufid, alen, atid, atid;
   float deltax, sum, psum, x;

   printf ("\nGive number of worker processes");
   scanf ("%d",&nprocs);
   printf ("\nGive number of points");
   scanf ("%d",&npts);

   deltax = PI/(float)npts;
   if (nprocs > 0) info = pvm_spawn ("int-worker2", NULL,
               PvmTaskDefault, NULL, nprocs, tids);

   nlocal = (npts-1)/(nprocs+1) + 1;
   sum = 0.0;
   nend = (nlocal < npts) ? nlocal : npts;

   for(i=1;i<=nend;++i){
      x = (i-0.5)*deltax;
      sum += sin(x);
   }
```

Continued on next slide. . .

PVM Integration Example

...continued from previous slide

```
   nbeg = nlocal + 1;
   for(i=0;i<nprocs;++i){
      nend  = (nbeg+nlocal)<npts ? (nbeg+nlocal) : npts;
      bufid = pvm_initsend (PvmDataDefault);
      info  = pvm_pkint (&nbeg, 1, 1);
      info  = pvm_pkint (&nend, 1, 1);
      info  = pvm_pkfloat (&deltax, 1, 1);
      info  = pvm_send (tids[i], 10);
      nbeg = nend + 1;
   }
   for(i=0;i<nprocs;++i){
      info = pvm_precv (-1, 10, &psum, 1, PVM_FLOAT,
                        &atid, &atag, &alen);
      sum += psum;
   }
   printf("\nThe integral is %f",sum*deltax);
   info = pvm_exit ();
}
```

PVM Integration Example

The worker program

```
#include "pvm3.h"
#include "math.h"

main()
{
   int parent, bufid, info;
   int i, nbeg, nend;
   float x, psum, deltax;

   parent = pvm_parent ();
   bufid = pvm_recv (parent, 10);
   info = pvm_upkint (&nbeg, 1, 1);
   info = pvm_upkint (&nend, 1, 1);
   info = pvm_upkfloat (&deltax, 1, 1);
   psum = 0.0;
   for(i=nbeg;i<=nend;++i){
      x = (i-0.5)*deltax;
      psum = psum + sin(x);
   }
   info = pvm_psend (parent, 10, &psum, 1, PVM_FLOAT);
   info = pvm_exit ();
}
```