Parallel Processing
CM0323

David W. Walker

http://users.cs.cf.ac.uk/David.W.Walker/

Syllabus: Part 1

- *Week 1*: Introduction; motivation; types of parallelism; data decomposition; uses of parallel computers; classification of machines.
- *Week 2*: SPMD programs; memory models; shared and distributed memory; OpenMP.
- *Week 3*: Example of summing numbers; interconnection networks; network metrics; Gray code mappings.
- *Week 4*: Classification of parallel algorithms; Speedup and efficiency.
- *Week 5*: Scalable algorithms; Amdahl's law; sending and receiving messages; programming with MPI.

Syllabus: Part2

- *Week 6*: Collective communication; integration example; regular computations and a simple example.
- *Week 7*: Regular two-dimensional problems and an example.
- *Week 8*: Dynamic communication and the molecular dynamics example.
- *Week 9*: Irregular computations; the WaTor simulation. Load balancing strategies.
- *Week 10*: Message passing libraries; introduction to PVM.
- *Week 11*: Review lectures.

Books

- “Parallel Programming,” B. Wilkinson and M. Allen, published by Prentice Hall, 1999. ISBN 0-13-671710-1.
- “Parallel Computing: Theory and Practice,” M. Quinn, published by McGraw-Hill, 1994.
- “Solving Problems on Concurrent Processors, Volume 1,” Fox, Johnson, Lyzenga, Otto, Salmon, and Walker, published by Prentice-Hall, 1988.
- “Using MPI,” Gropp, Lusk, and Skjellum, published by MIT Press, 1994.
- “Parallel Programming with MPI,” Peter Pacheco, published by Morgan Kaufmann, 1996.

Web Sites

- For the module: http://users.cs.cf.ac.uk/David.W.Walker/CM0323/.
- For MPI: http://www.mcs.anl.gov/mpi/
- For OpenMP: https://computing.llnl.gov/tutorials/openMP/
- For information on the World's fastest supercomputers: http://www.top500.org/

What is Parallelism?

- *Parallelism* refers to the simultaneous occurrence 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 1

- 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.

Types of Parallelism 2

- 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.

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%

Notes on Scheduling Example

- In the last example:
  - 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, andthey act in parallel as the robot does different things.

Robot Example

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

Notes on Robot Example

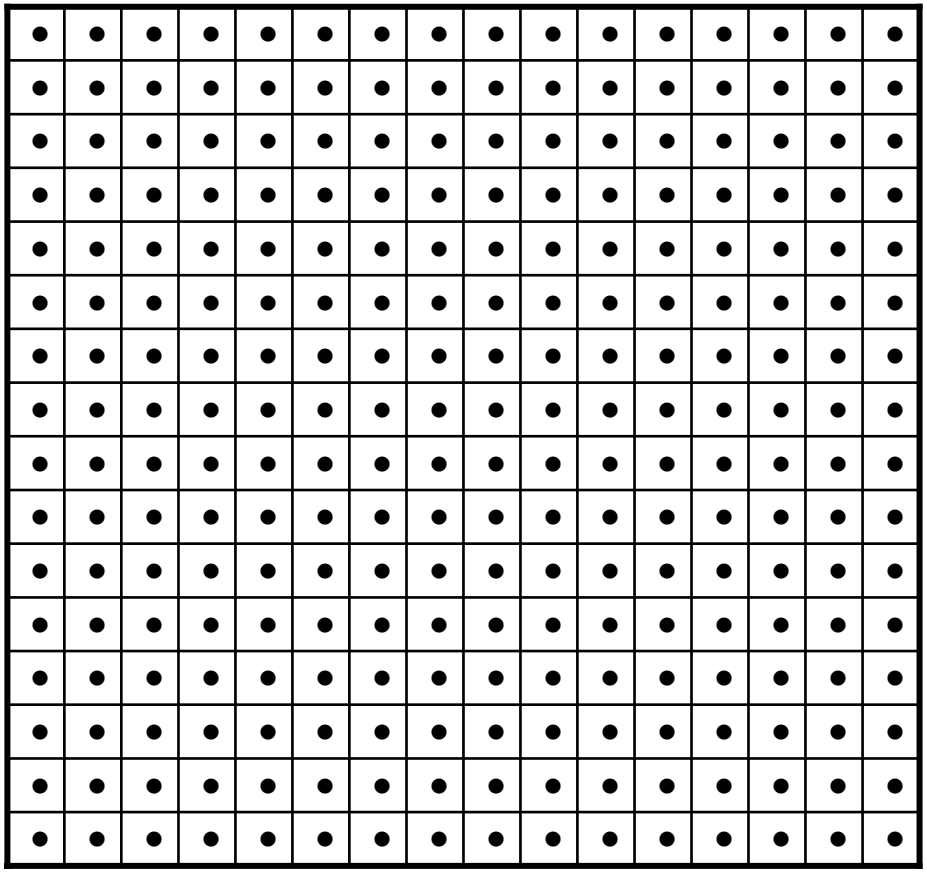
- The subsystems are fairly independent, with the vision subsystem guiding the others.
- There may also be a central “brain” processor.
- This is an example of *task parallelism* in which different tasks are performed concurrently to achieve a common goal.

Domain Decomposition

A common form of program-level parallelism arises from the division of the data to be programmed into subsets.

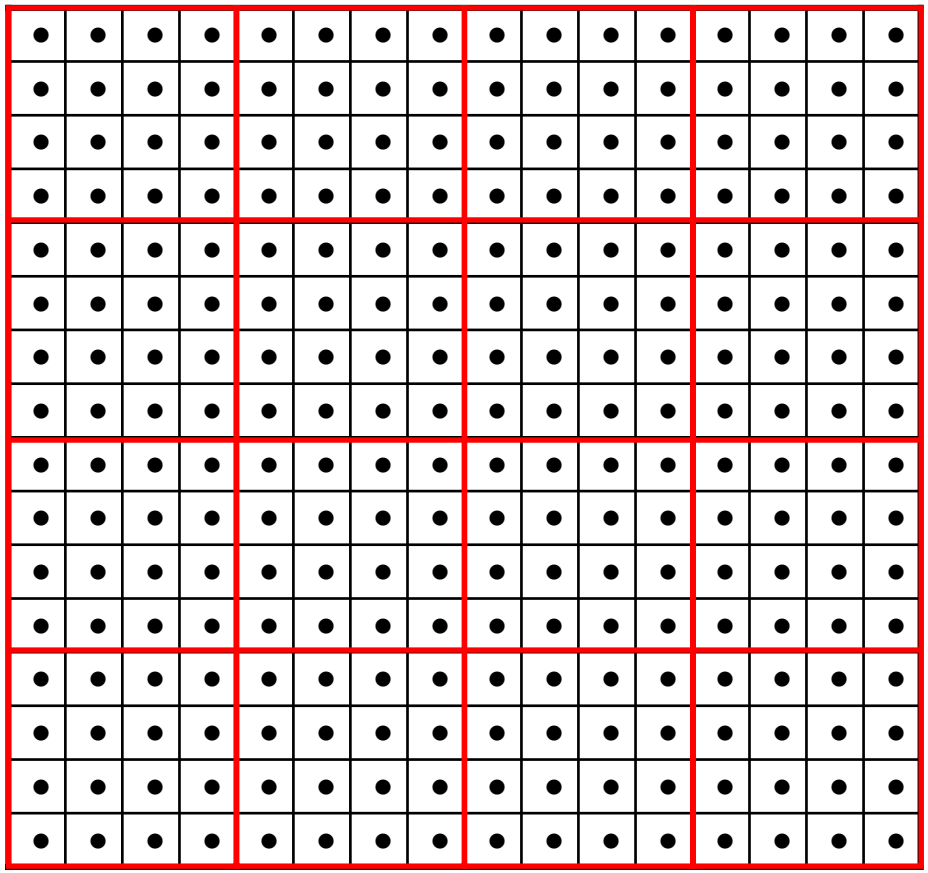
- This division is called *domain decomposition*.
- Parallelism that arises through domain decomposition is called *data parallelism*.
- The data subsets are assigned to different computational processes. This is called *data distribution*.
- Processes may be assigned to hardware processors by the program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

Data Parallelism



- 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.
- 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?).

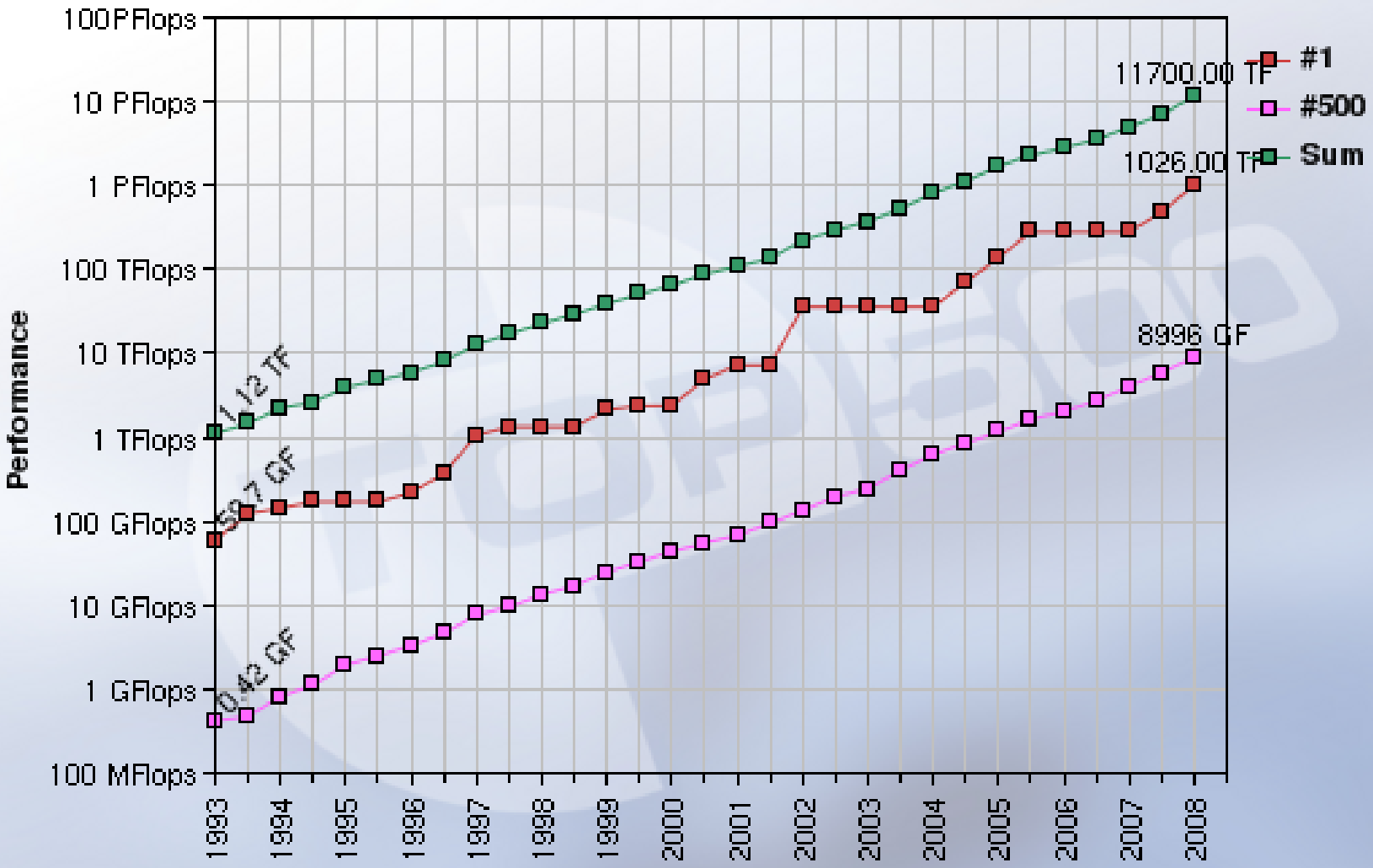
Why Use Parallelism?

- Better utilisation of resources. Want to keep hardware busy.
- 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 ≈ 10-20 Gflop/s
    - Current best supercomputer ≈ 1 Pflop/s
  - Transactions per second. 1 Tps is one transaction per second.
  - Instructions per second. 1 Mips is one million instructions per second.

Parallelism and Memory

- Want more memory to solve bigger or more complex problems.
- Typical workstations have 1 Gbytes of RAM, expandable to 32 Gbytes. Can fit an $65{,}536 \times 65{,}536$ array into 32 Gbytes of memory.
- The IBM "Roadrunner" parallel computer at Lawrence Livermore National Lab has 122,400 cores with a total of 98 Tbytes of RAM. Can fit a $3{,}670{,}000 \times 3{,}670{,}000$ array into memory. See http://www.top500.org/system/9485.

Performance Development



Parallelism and Supercomputing

- Parallelism is exploited on a variety of high performance computers, in particular *massively parallel computers* (MPPs) and *clusters*.
- MPPs, clusters, and high-performance vector computers are termed *supercomputers*.
- Currently supercomputers have peak performance in the range of 100-1000 Tflop/s, and memory of 10 to 100 Tbytes. They cost about 20-50 million pounds.
- Supercomputers are leading to a new methodology in science called *computational science* joining theoretical and experimental approaches.

Uses of Parallel 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 modelling**. 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.
- **Astrophysics**. 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.

Classification of Parallel Machines

- 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

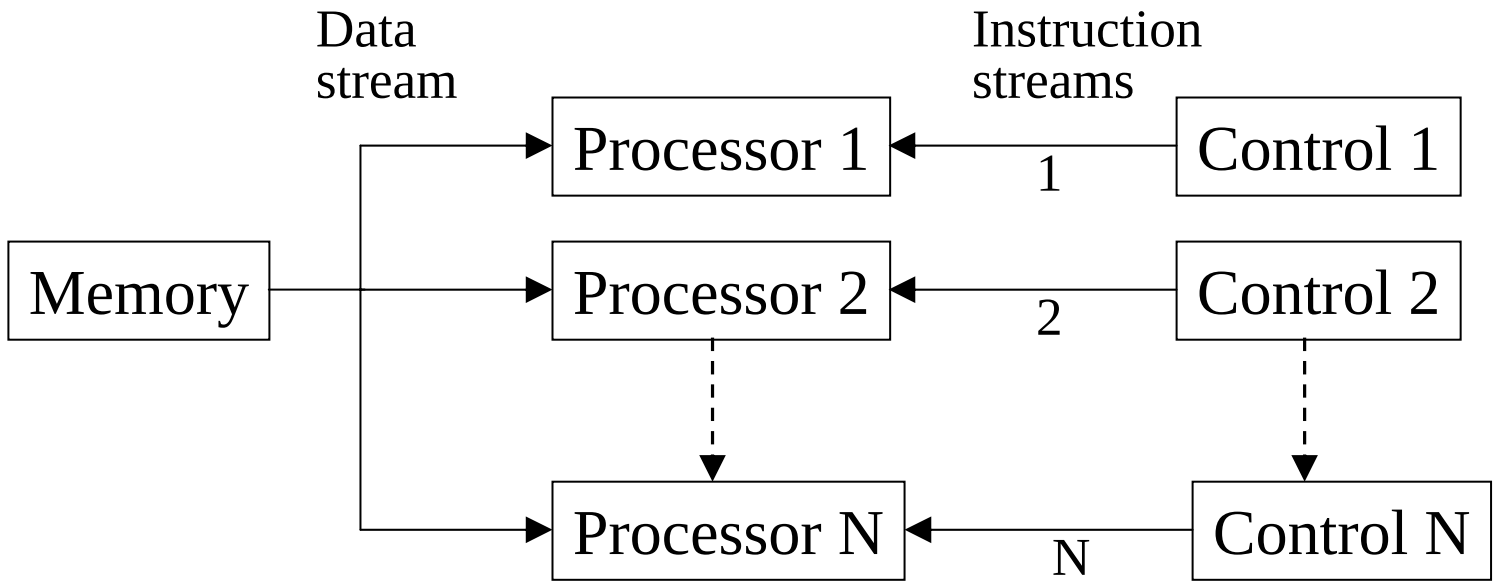
Example:

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 process parallelism since there is only one processor.

MISD Computers

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



MISD Computers (continued)

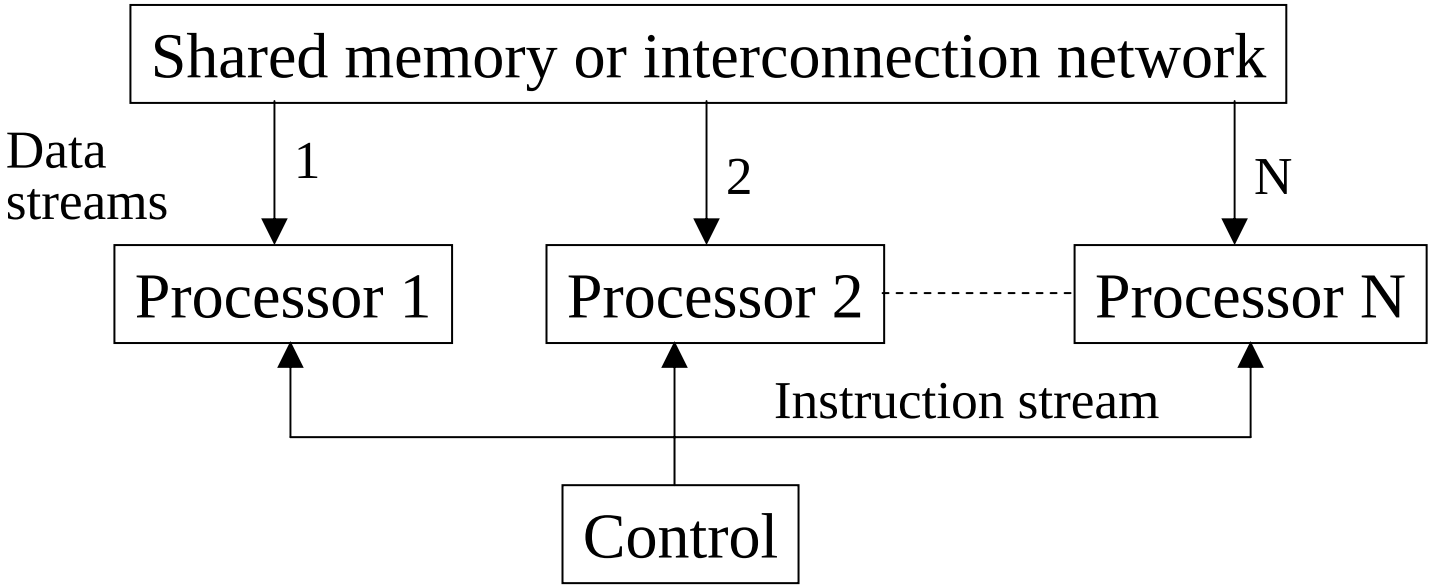
- There are N streams of instructions (algorithms/programs) and one stream of data. Parallelism is achieved by letting the processors do different things at the same time to the same data.
- MISD machines are useful in computations where the same input is to be subjected to several different operations.

MISD 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.



Notes on SIMD Computers

- 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.

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

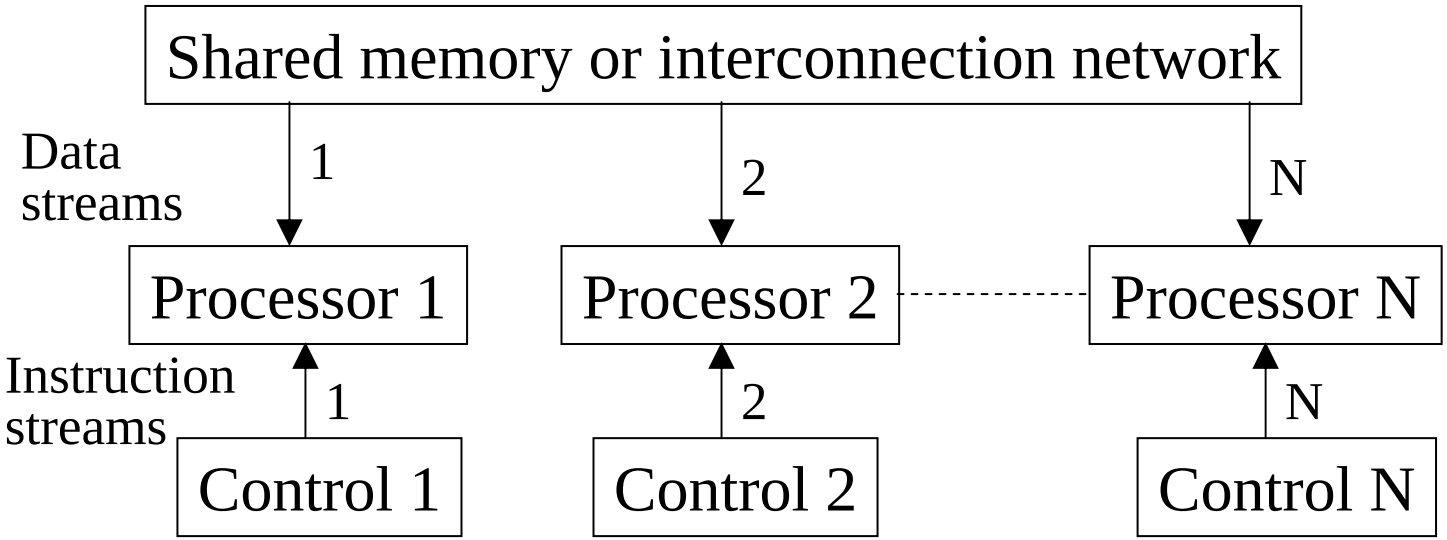
The same instruction is issued to all 4 processors (add two numbers), and all processors execute the instructions simultaneously. It takes one step to add the matrices, compared with 4 steps on a SISD machine.

Notes on SIMD Example

- In this example the instruction is simple, but in general it could be more complex such as merging two lists of numbers.
- The data 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
  - the processor is active (execute the instruction)
  - 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.



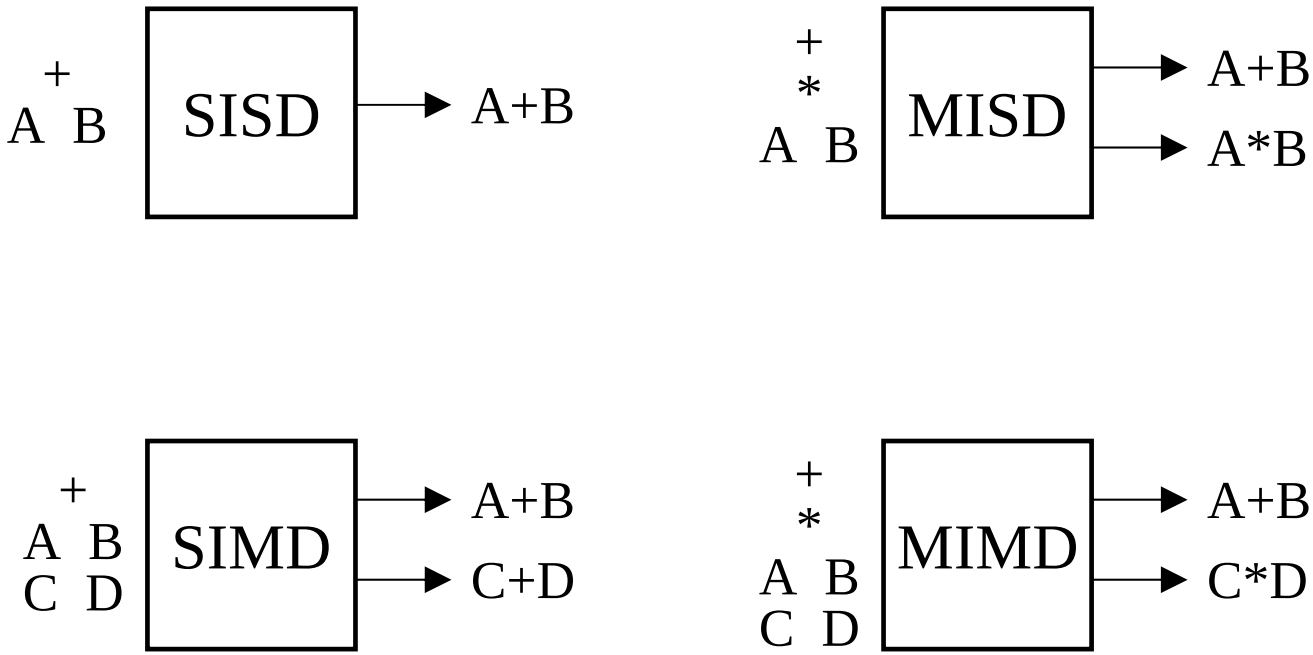
Notes on MIMD Computers

- 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
  - by using a shared memory and shared variables, or
  - using an interconnection network and message passing (distributed memory)
- MIMD computers with shared memory are known as *multiprocessors.* An example is the Onyx 300 produced by Silicon Graphics Inc.
- MIMD computers with an interconnection network are known as *multicomputers*. An example is the E6500 produced by Sun Microsystems.
- *Clusters* are multicomputers composed of off-the-shelf components

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.

SPMD Example

- Suppose X is 0 on processor 1, and 1 on processor 2. Consider

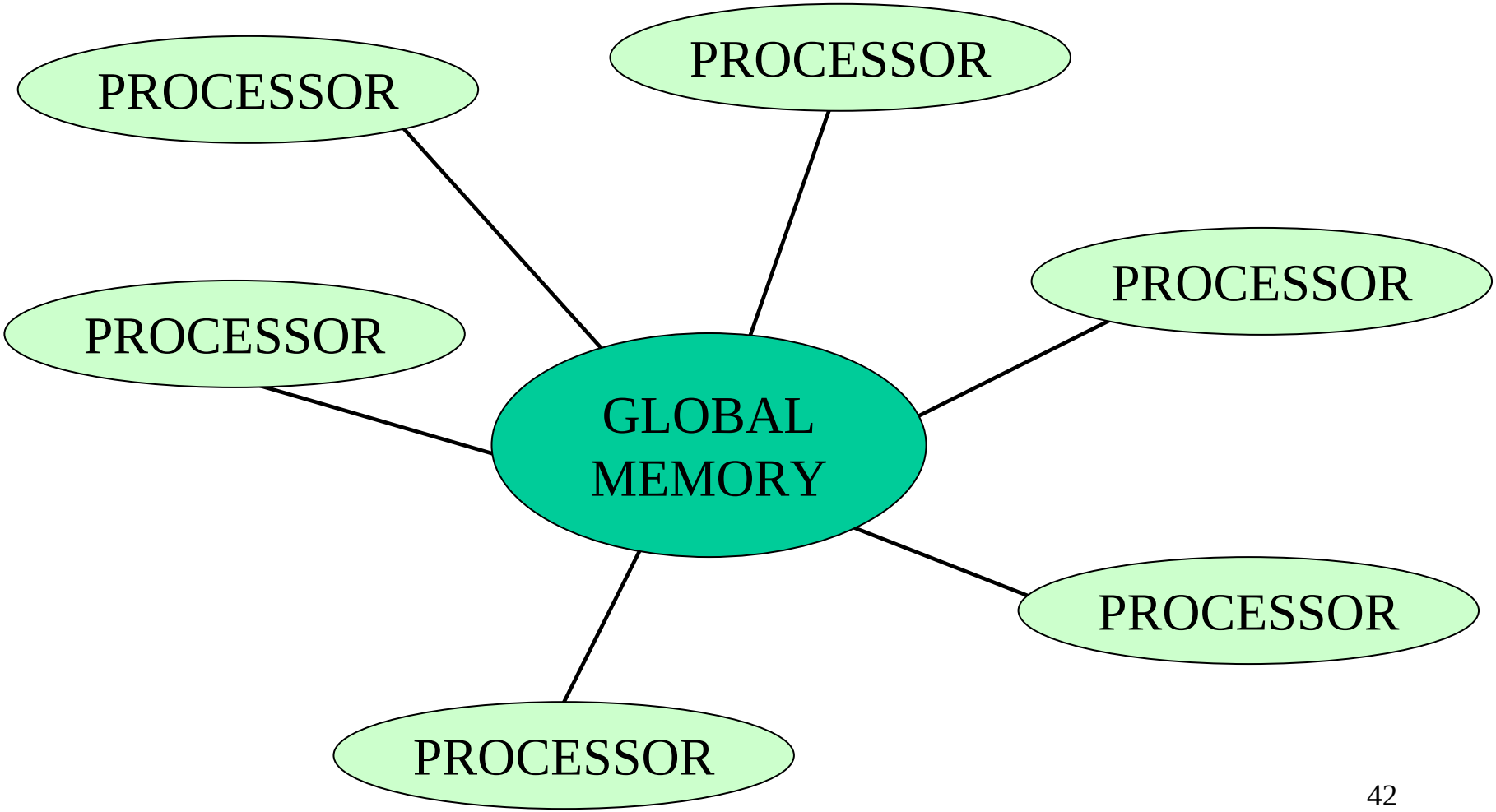
```
IF X = 0
   THEN S1
   ELSE S2
```

- 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.

Global Shared Memory



Shared Memory Conflicts

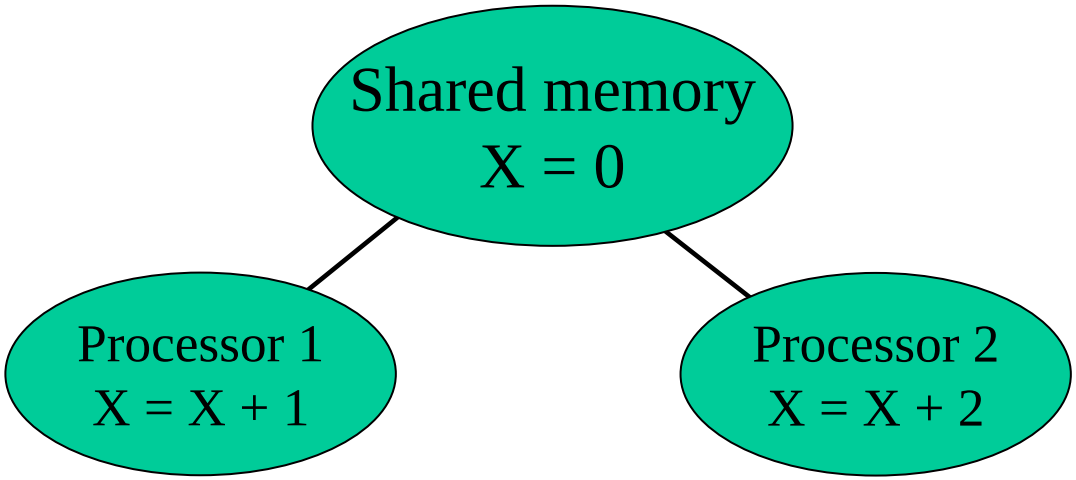
The shared memory approach is simple but can lead to problems when processors simultaneously access the same location in memory.

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?

You should have met this problem before when studying locks and critical sections in the operating systems module.

Shared Memory Conflicts 2



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

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

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*.

Locks and Mutual Exclusion

In shared memory programs *locks* can be used to give mutually exclusive access.

```
Processor 1:
        LOCK (X)
                X = X + 1
        UNLOCK (X)
```

```
Processor 2:
        LOCK (X)
                X = X + 2
        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 1

- 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?”

Notes on Shared Memory 2

- There are several ways of deterministically specifying the contents of a memory location after a concurrent write
  1. Assign priorities to processors and store value from processor with highest priority.
  2. All the processors are allowed to write, provided all the values they are attempting to store are the same.
  3. The max, min, sum, or average of the values is stored (for numeric data).

Notes on Shared Memory 3

- 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 100).

Examples of 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 = {L_1, L_2, …,L_m} for the index of a given element x. Assume x may appear several times, and any index will do. $1 < N \leq m$.

The 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

$S_i = \{L_{((i-1)m/N+1)}, ...,L_{(im/N)}\}$

perform sequential search on sublist S_i

(return K_i= -1 if not in list, otherwise index)

end for

STEP 3: for i=1 to N do in parallel

if $K_i > 0$ then $k=K_i$ end if

end for

end procedure

Time Complexity for EREW

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).

Time Complexity for ERCW

- 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 for CREW

- 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).

Time Complexity for CRCW

- CRCW

  Step 1 takes constant time.

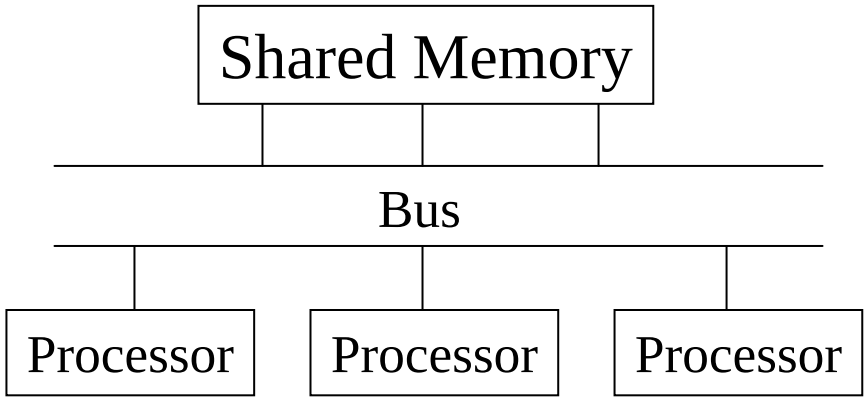
  Step 2 takes O(m/N) time.

  Step 3 takes constant time.

  Total time is O(m/N).

Limits on Shared Memory

- Shared memory computers are often implemented by incorporating a fast bus to connect processors to memory.

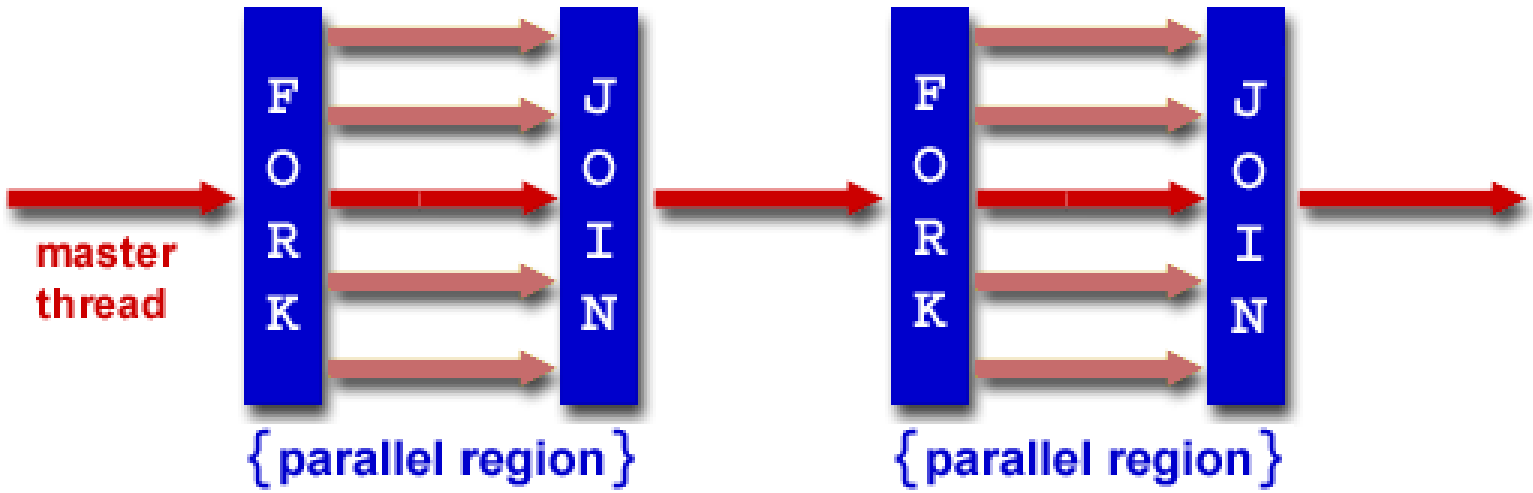


- However, because the bus has a finite bandwidth, i.e., it can carry only a certain maximum amount of data at any one time, then as the number of processors increase the *contention* for the bus becomes a problem. So it is feasible to build shared memory machines with up to only about 100 processors.

Quick Overview of OpenMP

- OpenMP can be used to represent task and data parallelism.
- In case of data parallelism, OpenMP is used to split loop iterations over multiple threads.
- Threads can execute different code but share the same address space.
- OpenMP is most often used on machines with support for a global address space.

OpenMP Fork/Join Model



OpenMP and Loops

```
#include <omp.h>
#define CHUNKSIZE 100
#define N 1000
main () {
   int i, chunk;
   float a[N], b[N], c[N];
/* Some initializations */
   for (i=0; i < N; i++) a[i] = b[i] = i * 1.0;
   chunk = CHUNKSIZE;
   #pragma omp parallel shared(a,b,c,chunk) private(i)
   {
      #pragma omp for schedule(dynamic,chunk) nowait
      for (i=0; i < N; i++) c[i] = a[i] + b[i];
   }
/* end of parallel section */ }
```

Number of Threads

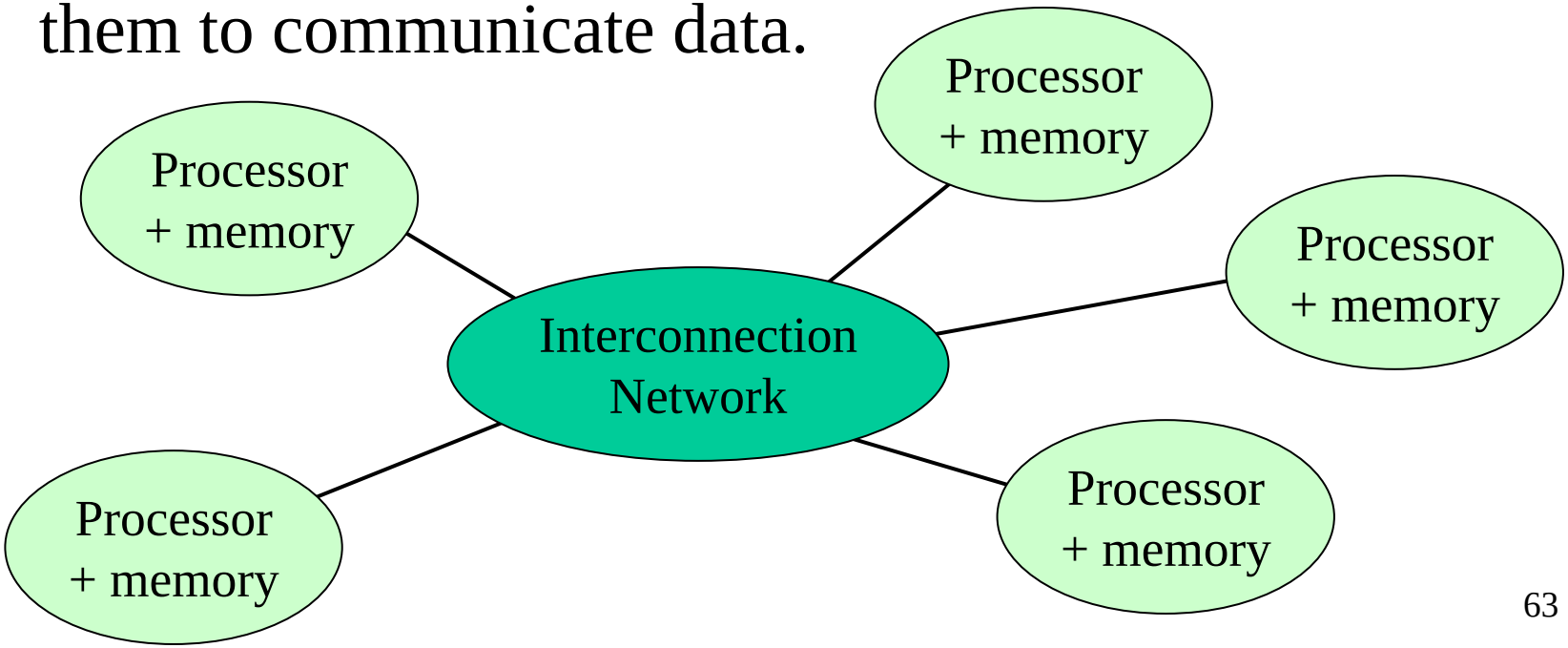
```
#include <omp.h>
#define CHUNKSIZE 100
#define N 1000
main () {
   int i, chunk;
   float a[N], b[N], c[N];
/* Some initializations */
   for (i=0; i < N; i++) a[i] = b[i] = i * 1.0;
   chunk = CHUNKSIZE;
   #pragma omp parallel shared(a,b,c,chunk) private(i) num_threads(4)
   {
      #pragma omp for schedule(dynamic,chunk) nowait
      for (i=0; i < N; i++) c[i] = a[i] + b[i];
   }
/* end of parallel section */ }
```

Reduction Operations

```
#include <omp.h>
main () {
   int i, n, chunk;
   float a[100], b[100], result;
/* Some initializations */
   n = 100;
   chunk = 10;
   result = 0.0;
   for (i=0; i < n; i++) {
      a[i] = i * 1.0;
      b[i] = i * 2.0;
   }
   #pragma omp parallel for default(shared) private(i) schedule(static,chunk) reduction(+:result)
   for (i=0; i < n; i++)
      result = result + (a[i] * b[i]);
   printf("Final result= %f\n",result);
}
```

Interconnection Networks 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.



Message Passing

- 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.

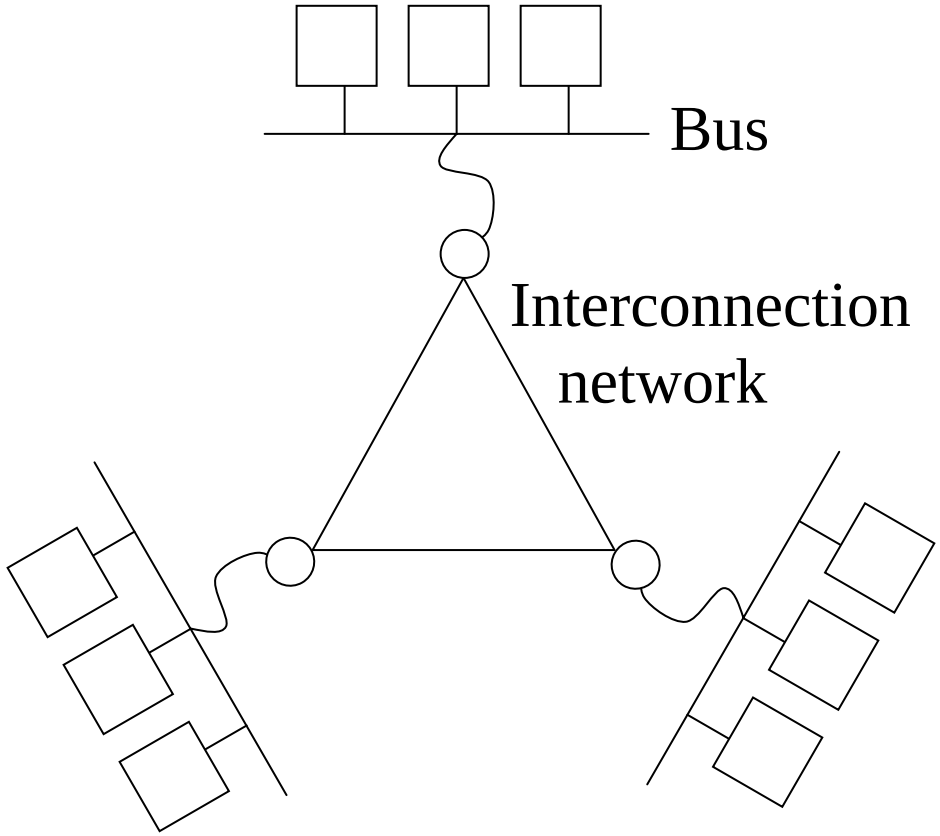
```
P1                          P2

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

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

Hybrid Computers

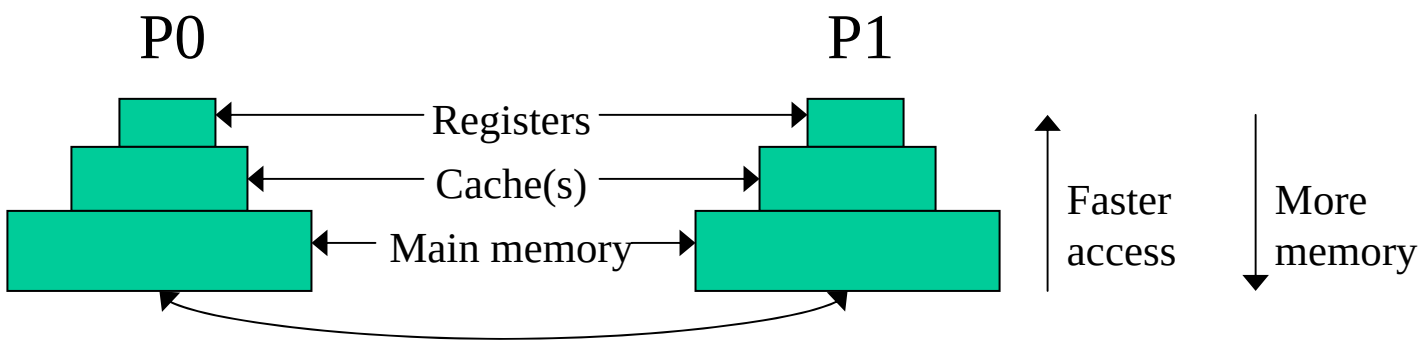
- In addition to the cases of shared memory and distributed memory there are possibilities for hybrid designs that incorporate features of both.
- Clusters of processors are connected via a high speed bus for communication within a cluster, and communicate between clusters via an interconnection network.



Comparison of Shared and Distributed Memory

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

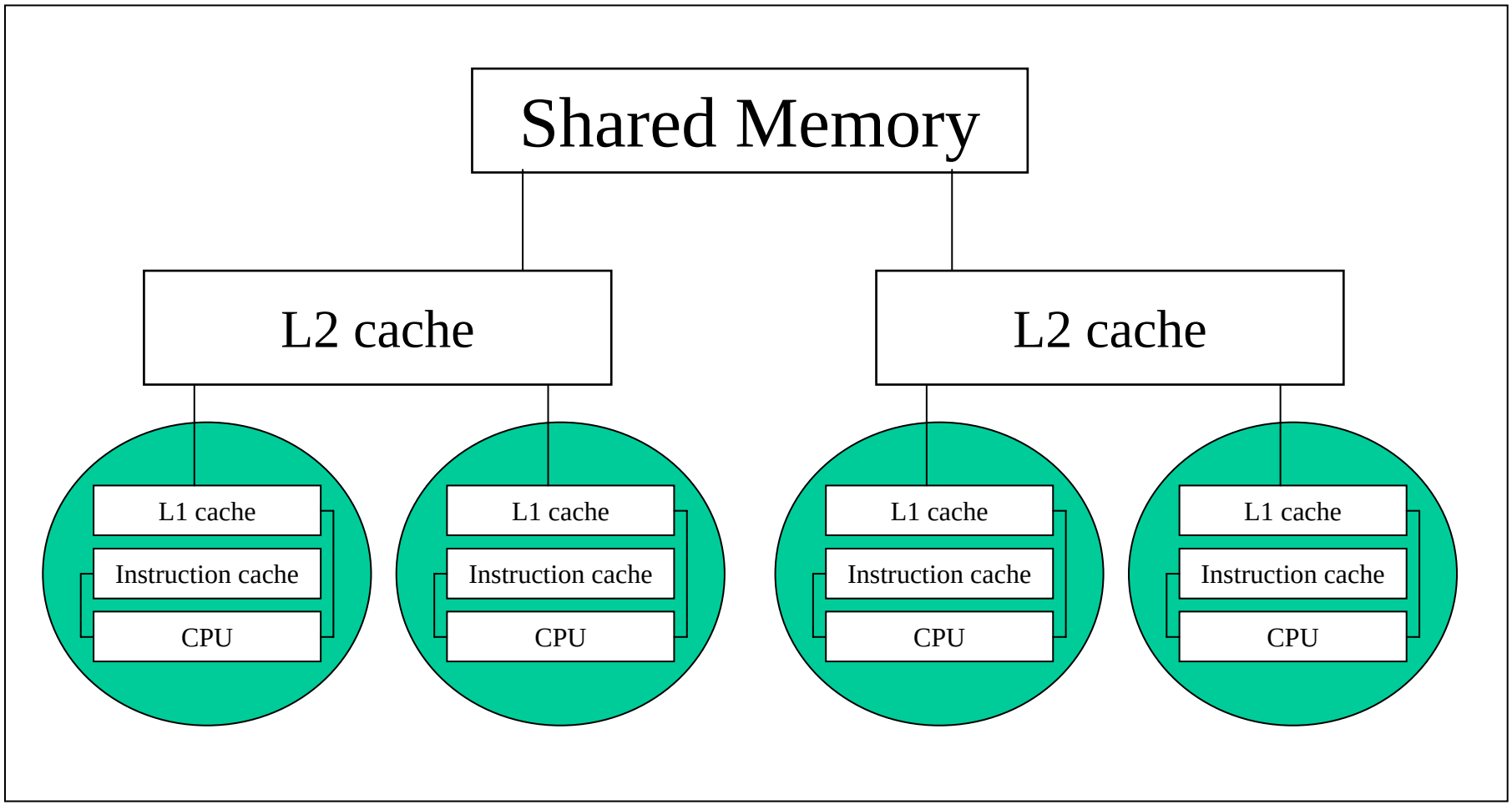
Memory Hierarchy 1



Data sharing between processes

- In general, certain memory locations have a greater affinity for certain processes.
- Parallel programming language may make distinction between “near” and “far” memory, but will not fully represent the memory hierarchy.

Typical Quad-Core Chip



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];
}
```

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

Summing m Numbers in Parallel

- 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
}
```

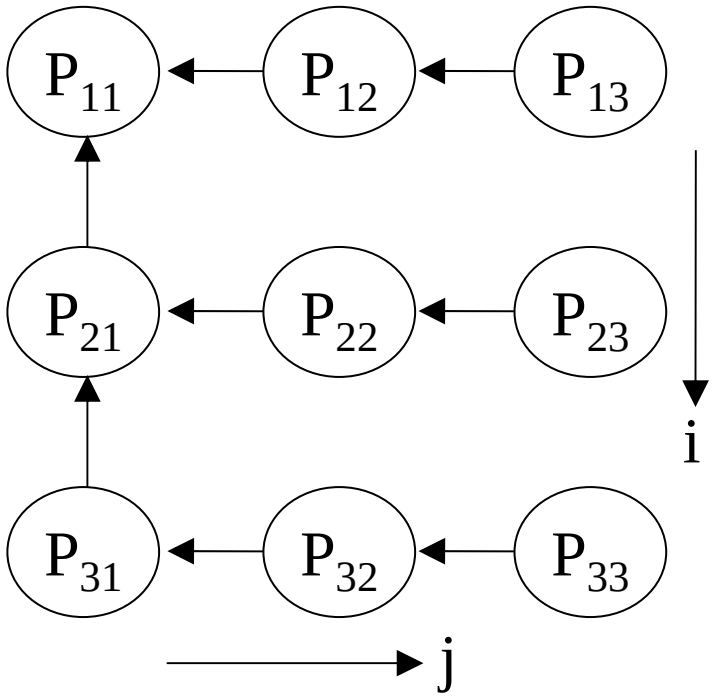
Notes on Shared Memory Algorithm

- 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 N processors.



The algorithm is as follows:

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

Distributed Memory Algorithm

The algorithm proceeds as follows:

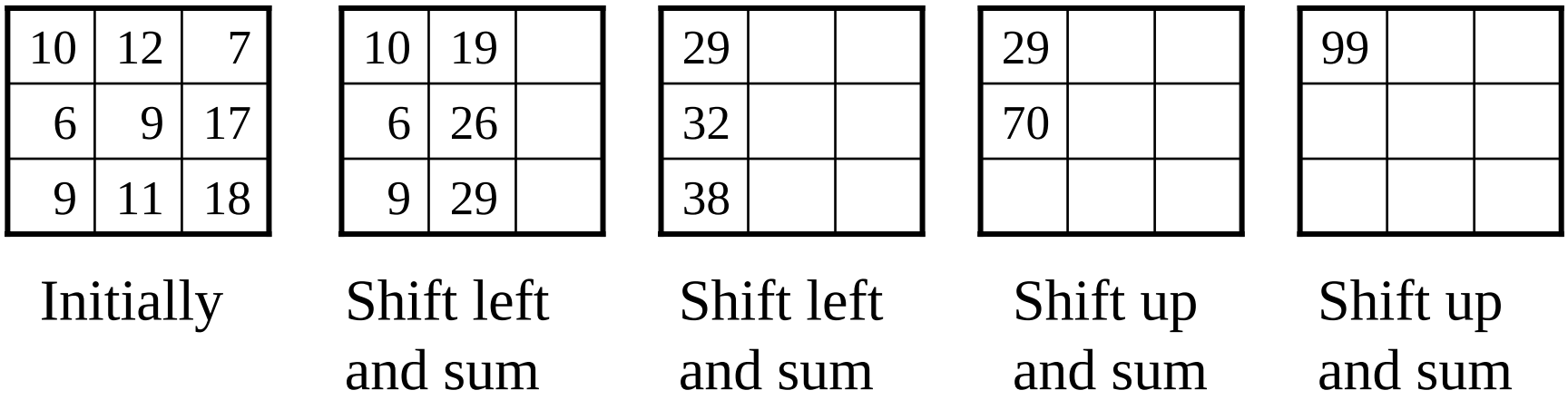
1. Each processor finds its local sum.
2. Sum along rows:
   a) If the processor is in the rightmost column it sends its local sum to the left.
   b) If the processor is not in the rightmost or leftmost column it receives the number form the processor on its right, adds it to its local, and send the result to the processor to the left.
   c) If the processor is in the leftmost column it receives the number from the processor on its right and adds it to it local sum to give the row sum.
3. Leftmost column only – sum up the leftmost column:
   a) If the processor is in the last row send the row sum to the processor above
   b) If the processor is not in the last or first row receive the number from the processor below, add it to the row sum, and send result to processor above
   c) If the processor is in the first row receive the number from the processor below. This is the global sum.

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.



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.

Examples of Networks

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.

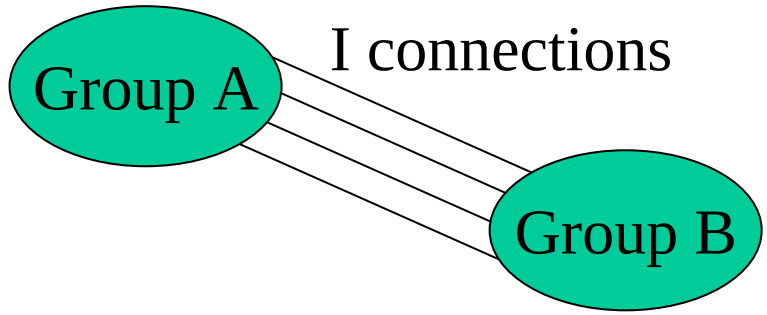
Network Metrics 2

- **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.

Network Metrics 3

Network narrowness measures congestion in a network.

- Partition the network into two groups A and B, containing NA and NB nodes, respectively, with NB < 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.



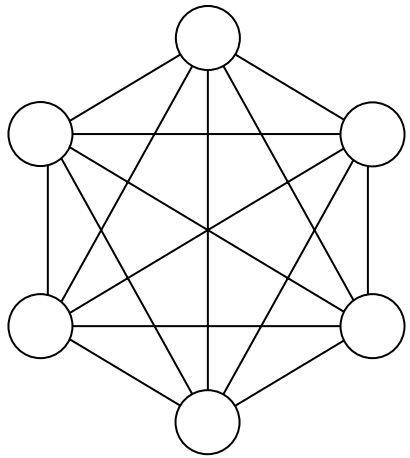
- If the narrowness is high (NB>I) then if the group B nodes want to communicate with the group A nodes congestion in the network will be high.

Network Metrics 4

- **Network Expansion Increment** 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

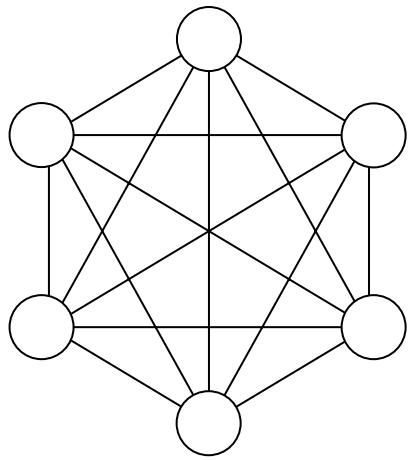
- In the fully connected, or all-to-all, network each node is connected directly to all other nodes.
- This is the most general and powerful interconnection network, but it can be implemented for only a small number of nodes.



Fully Connected Network 2

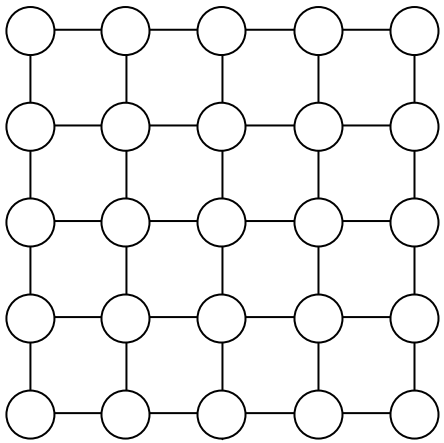
For n even:

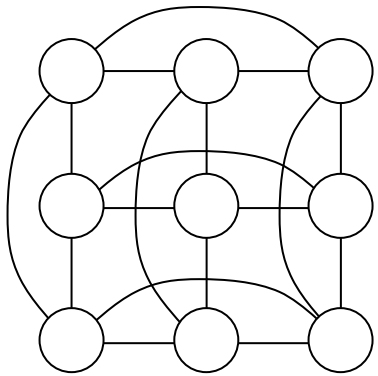
- Network connectivity = n - 1
- Network diameter = 1
- Network narrowness = 2/n
- Bisection width = $n^2/4$
- Expansion Increment = 1
- Edges per node = n - 1



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*.

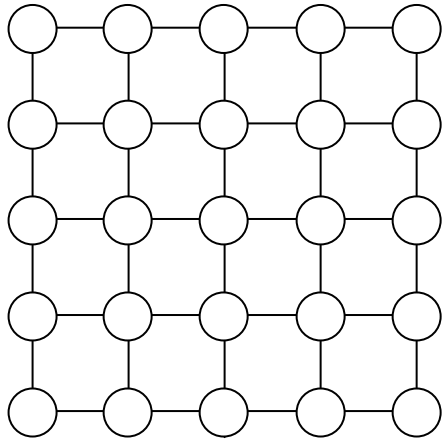




Mesh Metrics

For a q-dimensional non-periodic lattice with k^q nodes:

- Network connectivity = q
- Network diameter = q(k-1)
- Network narrowness = k/2
- Bisection width = k^{q-1}
- Expansion Increment = k^{q-1}
- Edges per node = 2q



Ring Networks

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

- Network connectivity = 2
- Network diameter = n/2
- Network narrowness = n/4
- Bisection width = 2
- Expansion Increment = 1
- Edges per node = 2

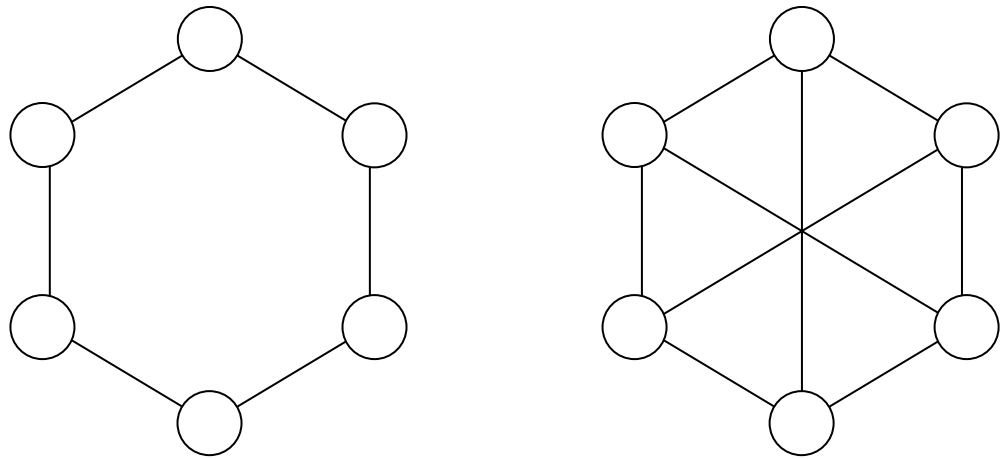
The problem for a simple ring is its large diameter.

Chordal Ring Networks

- A *chordal ring* uses extra chordal links to reduce the diameter.
- For a ring with extra diametric links we have (for n even)
  - Network connectivity = 3
  - Network diameter = ceiling(n/4)
  - Network narrowness = n/(n+4)
  - Bisection width = 2+n/2
  - Expansion Increment = 2
  - Edges per node = 3

Examples of Ring Networks

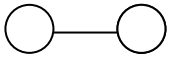
- Here are a simple ring and a chordal ring with diametric links, each of size 6 nodes.



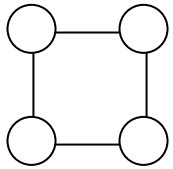
Hypercube Networks

- 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,…,n-1, and two nodes are connected if their node numbers differ in exactly one bit.
  - Network connectivity = k
  - Network diameter = k
  - Network narrowness = 1
  - Bisection width = 2^{k-1}
  - Expansion increment = 2^k
  - Edges per node = k

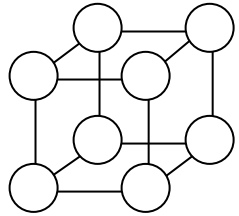
Examples of Hypercubes



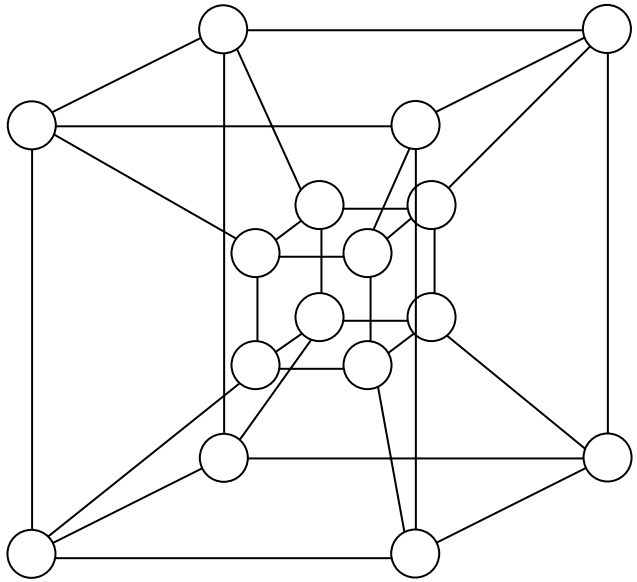
1D



2D



3D



4D

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.

Binary Gray Codes 2

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

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

Example of a Gray Code

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

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

Table 1: A 2-bit Gray code

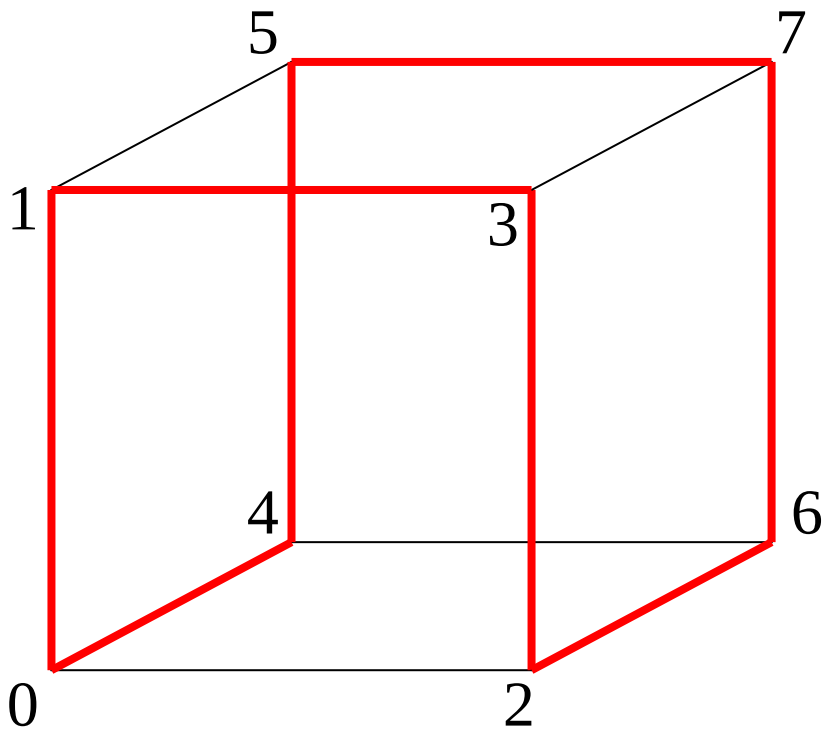
Example of a Gray Code 2

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

Table 2: A 3-bit Gray code

Example of a Gray Code 3

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



Multi-Dimensional Gray Codes

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

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

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

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

Multi-Dimensional Gray Codes 2

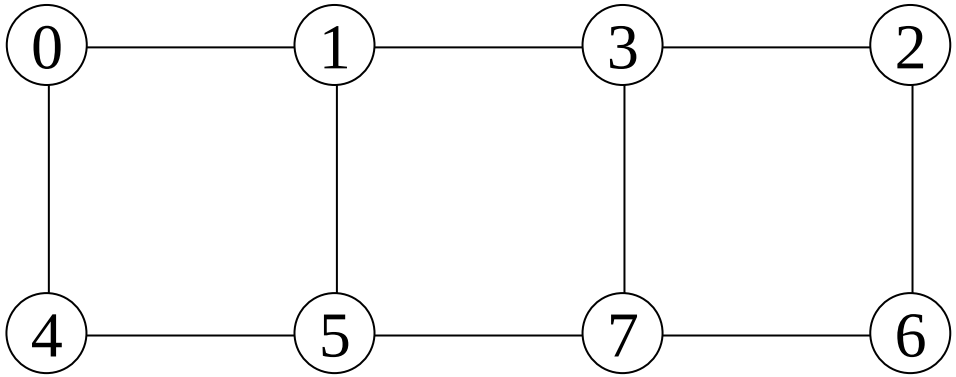
- We partition the bits of node number and assign them to each dimension of the mesh. The first d_0 go to dimension 0, the next d_1 bits go to dimension 1, and so on. Then we apply separate inverse Gray code mappings to each group of bits.

Mapping 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)

Mapping a 2×4 Mesh to a Hypercube 2

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

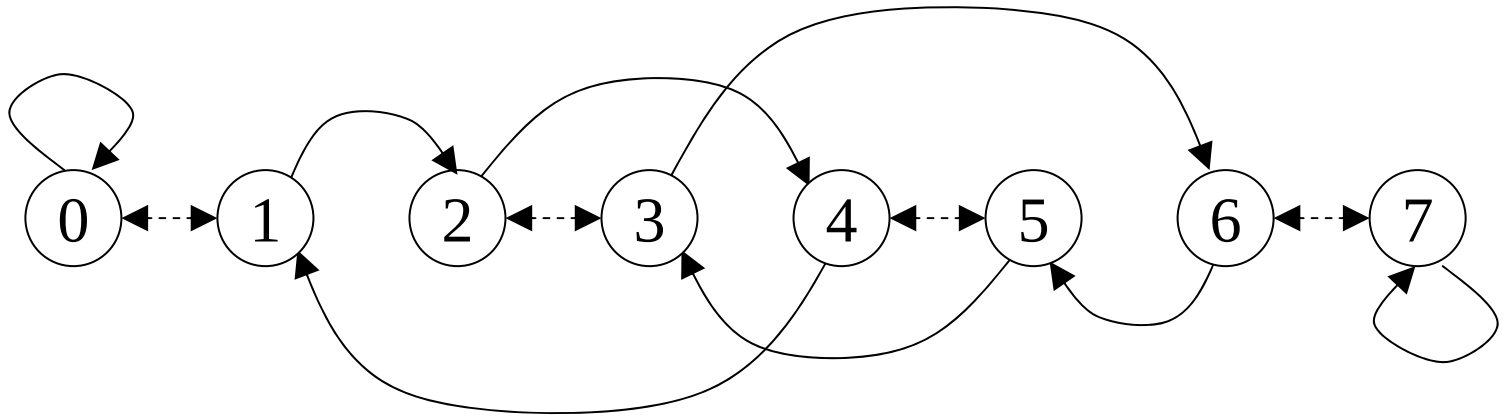


Shuffle-Exchange Networks

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

8-Node Shuffle-Exchange Network

- 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,…,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.

Shuffle-Exchange Networks

- 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}

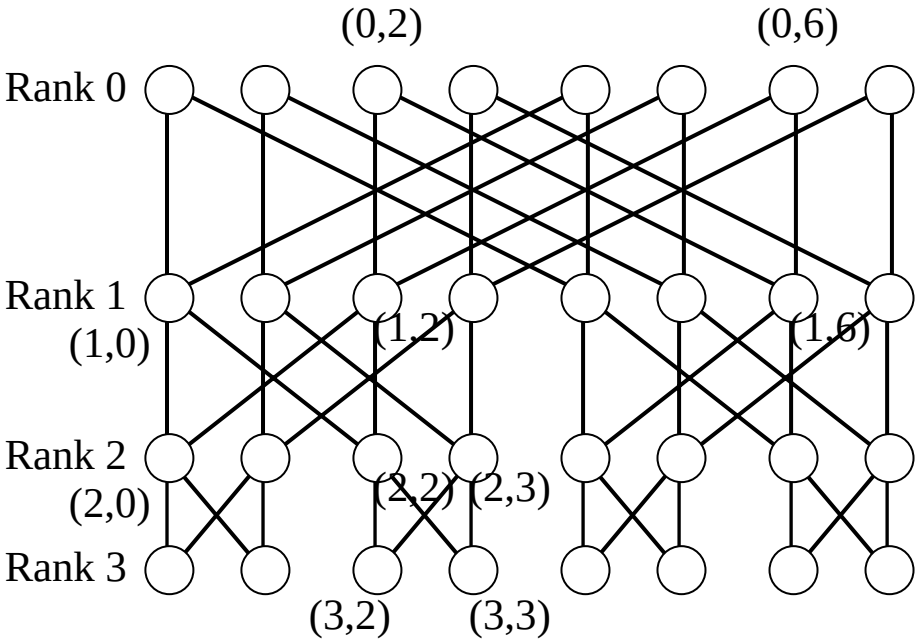
  after a shuffle operation.
- This corresponds to a cyclic leftward shift in the binary address.
- After k shuffle operations we get back to the node we started with, and nodes through which we pass are called a *necklace*.

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 in the ith rank. Then for i > 0 node (i,j) is connected to 2 nodes in rank i-1, node (i-1,j) and node (i-1,m), where m is the integer found by inverting the 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.
  - Network diameter = 2k
  - Bisection width = 2^k

Example of a Butterfly Network

Here is a butterfly network for k = 3.



$i = 1, j = 2 = (010)_2, j' = (110)_2 = 6$

$i = 2, j = 2 = (010)_2, j' = (000)_2 = 0$

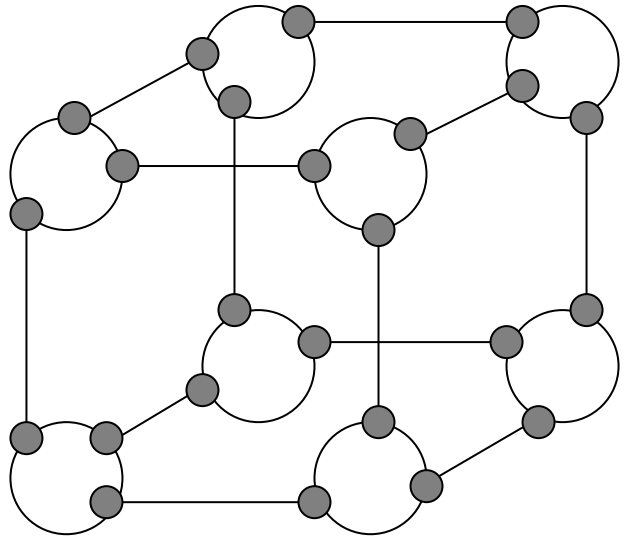
$i = 3, j = 2 = (010)_2, j' = (011)_2 = 3$

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 $k2^k$,
  - Network diameter = 2k
  - Bisection width = 2^{k-1}
  - Edges per node = 3

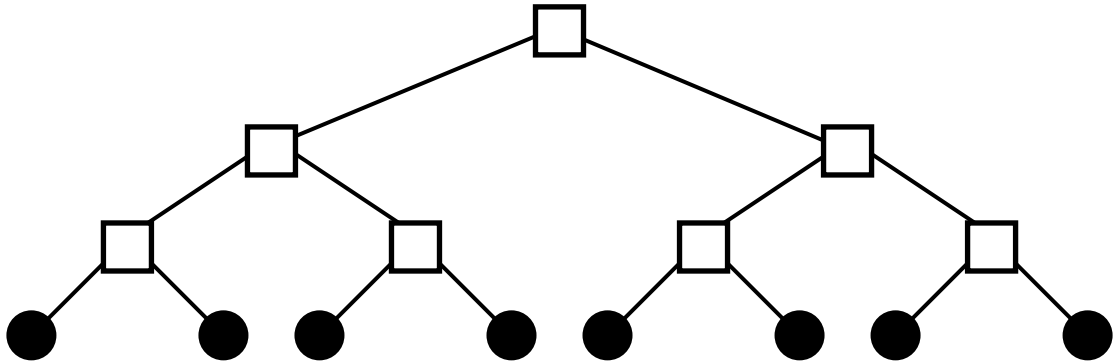
Example of a Cube-Connected Cycles Network

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



Complete Binary Tree Network

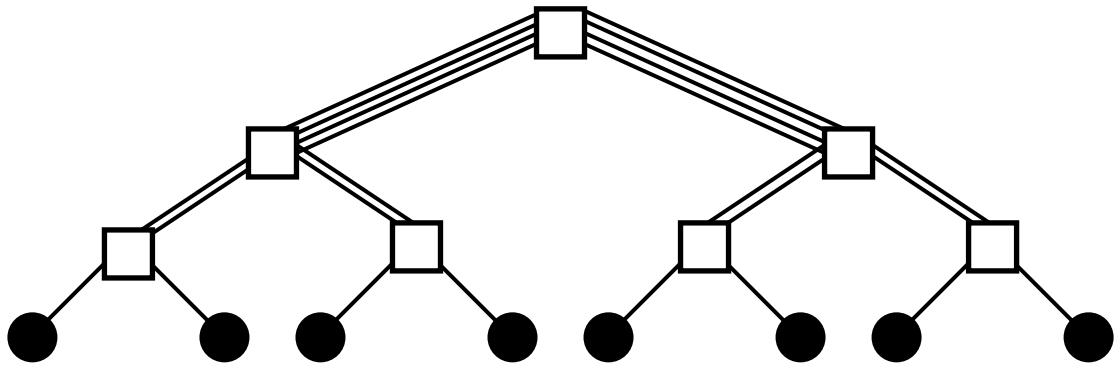
- Tree-based networks use switches to connect processors. An example is the binary tree network.



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

Fat Tree Network

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



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

Classifying Parallel Algorithms

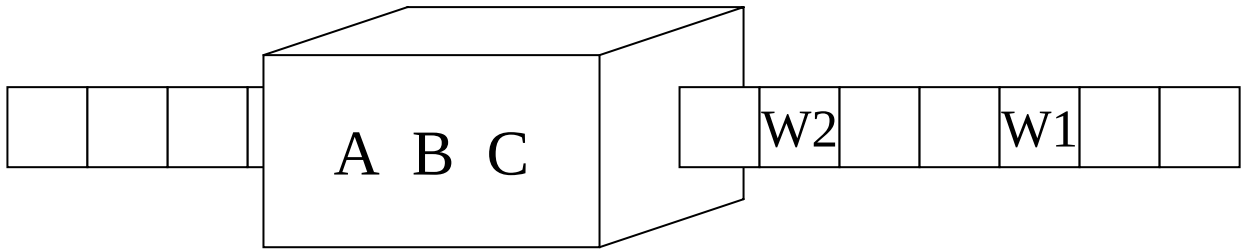
- Parallel algorithms for MIMD machines can be divided into 3 categories
  - Pipelined algorithms
  - Data parallel, or partitioned, algorithms
  - Asynchronous, or relaxed, algorithms

Pipelined Algorithms

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

Pipelines Algorithms 2

- **Example**: Suppose it takes 3 steps, A, B, and C, to assemble 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.

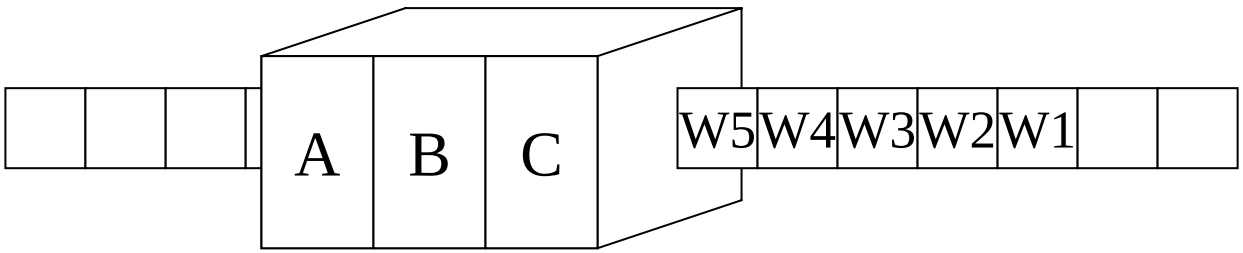


Example of Pipelined Algorithm

- 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.

Pipelined Algorithm

- 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

Sequential time = NTM

Pipelined time = (N+M-1)T

Pipeline Performance Example

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

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

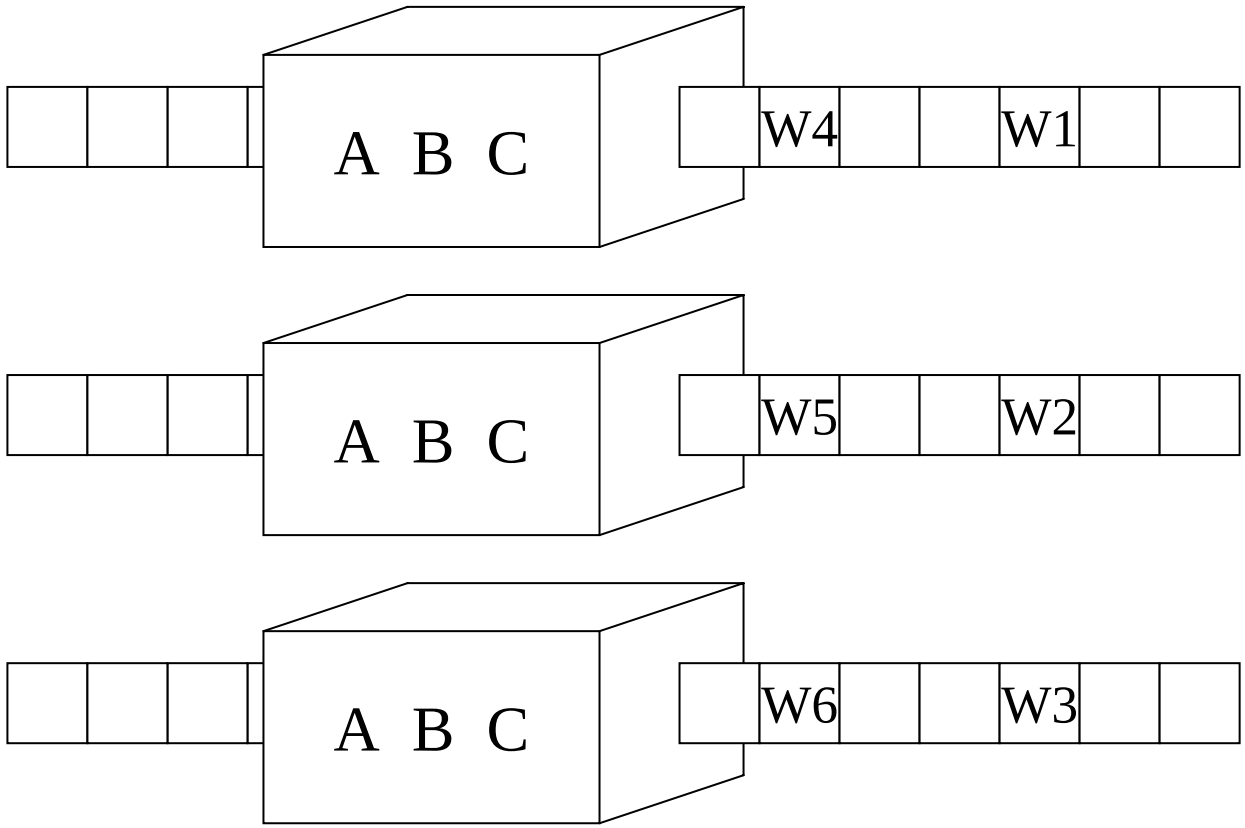
The speed-up $T_{seq}/T_{pipe} \approx 100$.

Data Parallelism

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

Data Parallelism Example

Data parallelism can be exploited in the 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.

Relaxed Parallelism Example

Suppose we have 2 processors, A and B.

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

Synchronous Operation

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

Asynchronous Operation

1. A produces a_1, passes it to B, which calculates F_1
2. A is in the process of computing a_2, but B does not wait – it uses a_1 to calculate F_2, i.e., $F_1=F_2$.

- Asynchronous algorithms keep processors busy. Drawbacks of asynchronous algorithms are
  - they are difficult to analyse
  - an algorithm that is known to converge in synchronous mode
  - may not converge in asynchronous mode.

Example of Asynchronous Algorithm

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

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

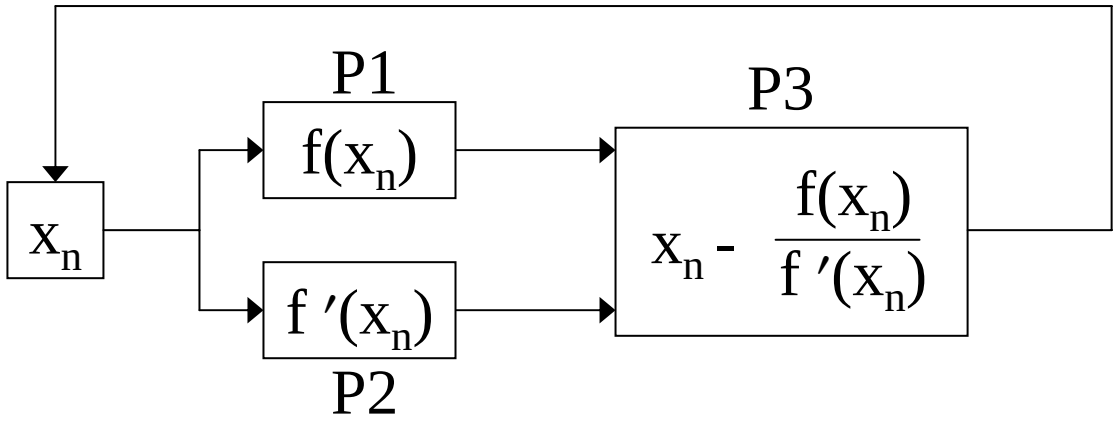
- generates a sequence of approximations to the root, starting with some initial value x_0.

Example of Asynchronous Algorithm 2

Suppose we have 3 processors

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

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



Serial Mode Time Complexity

Serial mode

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

Synchronous Parallel Mode

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

Asynchronous Parallel Mode

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

Asynchronous Parallel Mode Example

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

Time	P1	P2	P3
1	C0	C0	–
2	$f(x_0)$	C0	–
3	–	$f'(x_0)$	
4	–	–	$x_1= x_0- f(x_0)/ f'(x_0)$
5	C1	C1	–
6	$f(x_1)$	C1	–
7	–	$f'(x_1)$	$x_2= x_1- f(x_1)/ f'(x_0)$
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	$x_6= x_5- f(x_3)/ f'(x_2)$

Speed-up and Efficiency

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

  $S(N) = T_{seq}/T_{par}(N)$
- **Efficiency** is the speed-up per processor.

  $\epsilon(N) = S(N)/N=(1/N)(T_{seq}/T_{par}(N))$
- **Overhead** is defined as

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

Example

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

  Speed-up = 8/2 = 4

  Efficiency = 4/5 = 0.8

  Overhead = 1/0.8 – 1 = 0.25

Self Speed-up and Linear Speed-up

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

Factors That Limit Speed-up
1. Software Overhead

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

Factors That Limit Speed-up
2. Load Imbalance

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

Factors That Limit Speed-up
3. Communication Overhead

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

Grain Size

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

Definition of Load Imbalance

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

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

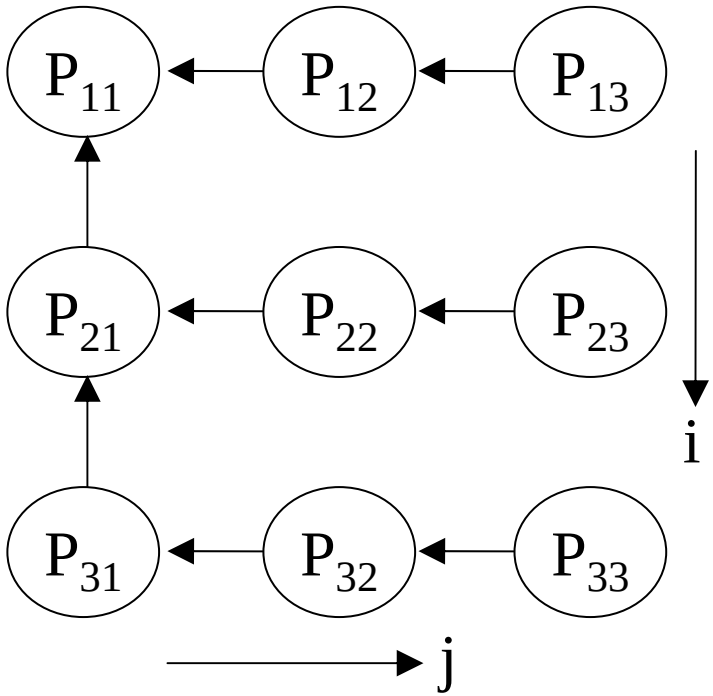
- The amount of load imbalance is then given by:

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

where the maximum is taken over all processors.

Analysis of Summing Example

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



The algorithm proceeds as follows

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

Analysis of Summing Example 2

- Time for best sequential algorithm is

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

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

Analysis of Summing Example 3

- Total time for the parallel algorithm is:

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

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

$$S(N) = \frac{(m-1)t_{calc}}{(m/N + 2\sqrt{N} - 3)t_{calc} + 2(\sqrt{N} - 1)\, t_{comm}}$$

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

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

Analysis of Summing Example 4

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

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

- As $g \to \infty$ 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, $S \to 0$.

Analysis of Summing Example 5

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

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

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

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

Scalable Algorithms

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

Scalability of the Summing Example

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

Amdahl’s Law

- Amdahl's Law states that the maximum speedup of an algorithm is limited by the relative number of operations that must be performed sequentially, i.e., by its *serial fraction*.
- If 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:

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

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

Derivation of Amdahl's Law

- The speed-up satisfies:

$$S(N) = Tseq/Tpar(N) = nt/[(\alpha n + (1-\alpha)n/N)t + C(n,N)]$$

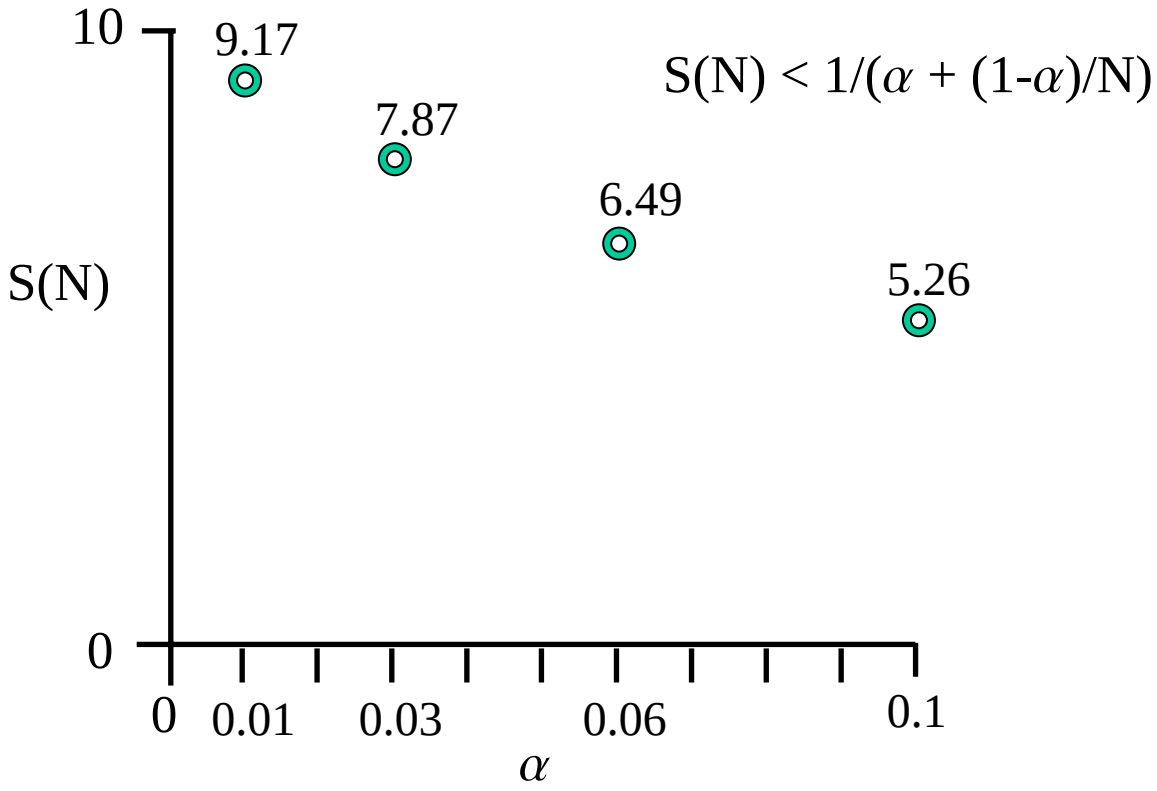
$$= 1/[(\alpha + (1-\alpha)/N) + C(n,N)/(nt)]$$

$$< 1/(\alpha + (1-\alpha)/N)$$

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

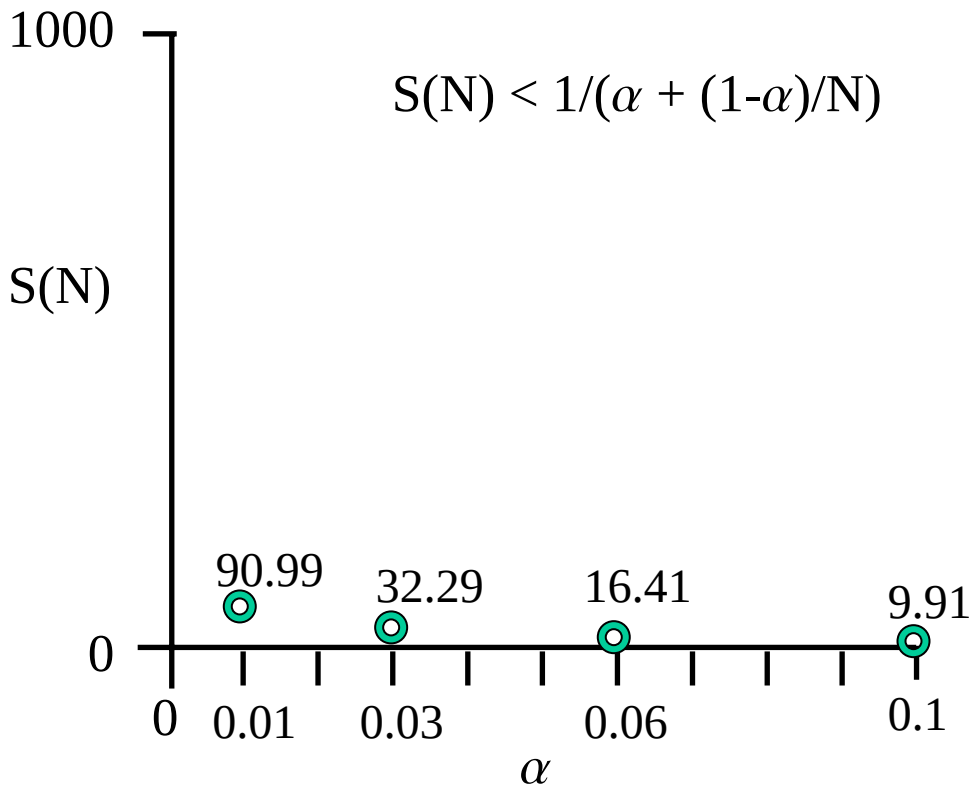
Examples of Amdahl's Law

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



Examples of Amdahl's Law 2

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



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

Implications of Amdahl’s Law

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

Speed-Up for Large Problems

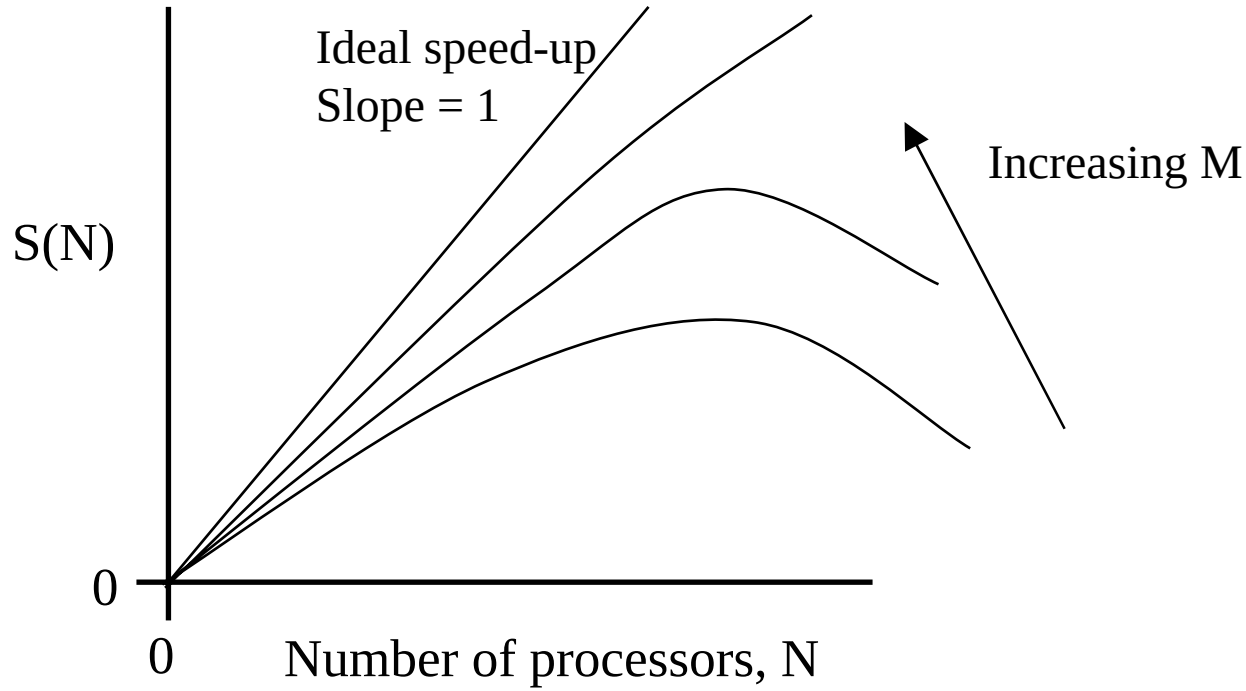
- *Speed-up* is the ratio between how long the best sequential algorithm takes on a single processor and how long it takes to run on multiple processors.
- To measure the speed-up the problem must be small enough to fit into the memory of one processor.
- This limits us to measuring the speed-up of only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

Speed-Up for Large Problems 2

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

Speed-Up and Problem Size

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



Semantics of Message Sends

- Suppose one node sends a message to another node:

  send (data, count, datatype, destination)

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

Semantics of Blocking Send

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

Semantics of Non-Blocking Send

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

Semantics of Message Receives

- Suppose one node receives a message from another node:

  receive (data, count, datatype, source)

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

Semantics of Blocking Receive

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

Semantics of Non-Blocking Receive

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

Message Passing Protocols

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

  SOURCE: send (data, count, datatype, destination)
  DEST: receive (data, count, datatype, source)

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

Message Passing Protocols 2

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

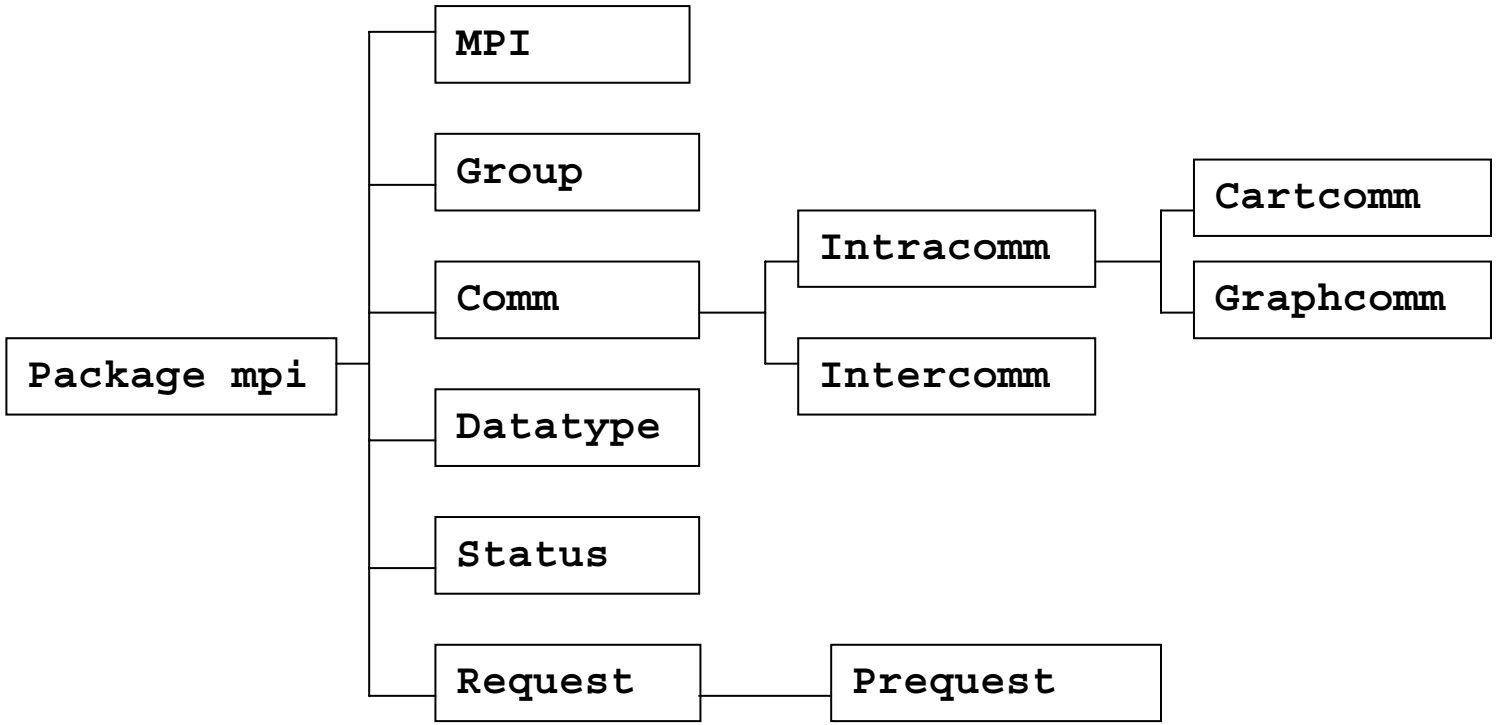
MPI Point-to-Point Communication

- MPI is a widely-used standard for message passing on distributed memory concurrent computers.
- Communication between pairs of processes is called point-to-point communication.
- There are several Java versions of MPI, but we shall use **`mpiJava`**.
- In **`mpiJava`** point-to-point communication is provided through the methods of the **`Comm`** class.

mpiJava API

- The class **`MPI`** only has static members.
- It acts as a module containing global services, such as initialisation, and many global constants including the default communicator **`COMM_WORLD`**.
- The most important class in the package is the communicator class **`Comm`**.
- All communication functions in **`mpiJava`** are members of **`Comm`** or its subclasses.
- Another very important class is the **`Datatype`** class.
- The **`Datatype`** class describes the type of elements in the message buffers to send and receive.

Class hierarchy



Basic Datatypes

MPI Datatype	**Java Datatype**
MPI.BYTE	byte
MPI.CHAR	char
MPI.SHORT	short
MPI.BOOLEAN	boolean
MPI.INT	int
MPI.LONG	long
MPI.FLOAT	float
MPI.DOUBLE	double
MPI.OBJECT	object

mpiJava send()/recv()

- Send and receive members of **Comm**:

```
void Send(Object buf, int offset, int
count, Datatype type, int dst, int
tag);

Status Recv(Object buf, int offset,
int count, Datatype type, int src, int
tag);
```

- **buf** must be an array.
- **offset** is the element where message starts.
- **Datatype** class describes type of elements.

Communicators

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

Process ranks

- When an MPI program is started the number of processes ,N, is supplied to the program from the invoking environment. The number of processes in use can be determined from within the MPI program with the **`Size()`** method.

```
int Comm.Size()
```

- Each of the N processes is identified by a unique integer in the range 0 to N-1. This is called the process *rank*. A processcan determine its rank with the **`Rank()`** method.

```
int Comm.Rank()
```

Message Tags

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

Return Status Objects

- If the message source and/or tag are/is wildcarded, then the actual source and tag can be found from the publicly accessible **`source`** and **`tag`** fields of the status object returned by **`recv()`**.
- The number of items received can be found using:

```
int Status.Get_count(Datatype datatype)
```

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.

Summary of Point-to-Point Communication

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

Minimal **mpiJava** Program

```
import mpi.*

class Hello {
  static public void main(String[] args) {
    MPI.Init(args) ;
    int myrank = MPI.COMM_WORLD.Rank();
    if(myrank == 0) {
      char[] data = new char [20];
      MPI.COMM_WORLD.Recv(data, 0, 20, MPI.CHAR, 1, 99);
      System.out.println("received:" + new String(data) + ":");
    } else if (myrank==1){
      char[] data = "Hello, there".toCharArray();
      MPI.COMM_WORLD.Send(data, 0, data.length, MPI.CHAR, 0, 99);
    }
    MPI.Finalize();
  }
}
```

Another MPI Example

```
import mpi.*

class HelloAll {
  static public void main(String[] args) {
    MPI.Init(args) ;
    int myrank = MPI.COMM_WORLD.Rank();
    int nprocs = MPI.COMM_WORLD.Size();
    int[] irecv = new int[1];
    if(myrank == 0) {
       for(int i=1;i<nprocs;i++){
          MPI.COMM_WORLD.Recv(irecv, 0, irecv.length, MPI.INT,
                              MPI.ANY_SOURCE, 99);
           System.out.println("Hello from process:" + irecv[0]);
      }
    } else {
      irecv[0] = myrank;
      MPI.COMM_WORLD.Send(irecv, 0, 1, MPI.INT, 0, 99);
    }
    MPI.Finalize();
  }
}
```

Notes on Example

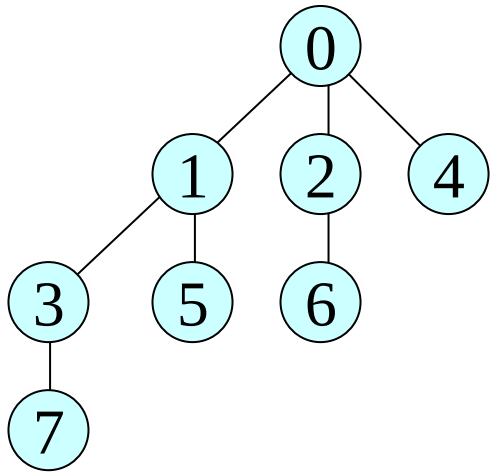
- All MPI calls must come between the calls to MPI.Init() and MPI.Finalize().
- The information from the processes is not necessarily output in ascending order because the program does not specify the order in which process 0 receives messages.
- We could have each process output its rank instead of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

Collective Communication

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

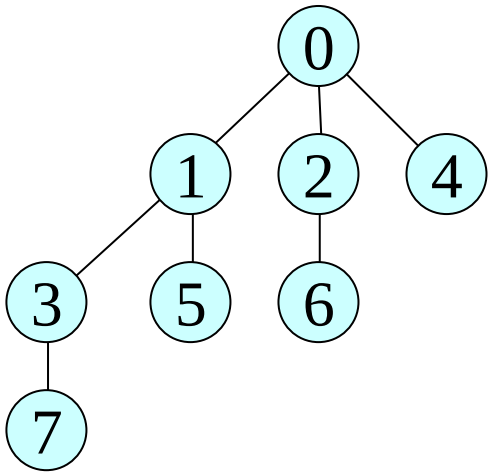
Broadcast

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



Broadcast Algorithm 1

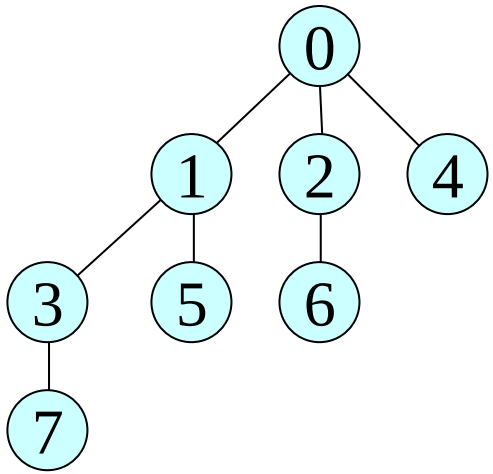
- Send on all links simultaneously
  1) Node 0 sends to nodes 1, 2, and 4
  2) Node 1 sends to nodes 3 and 5; node 2 sends to node 6
  3) Node 3 sends to node 7



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

Broadcast Algorithm 2

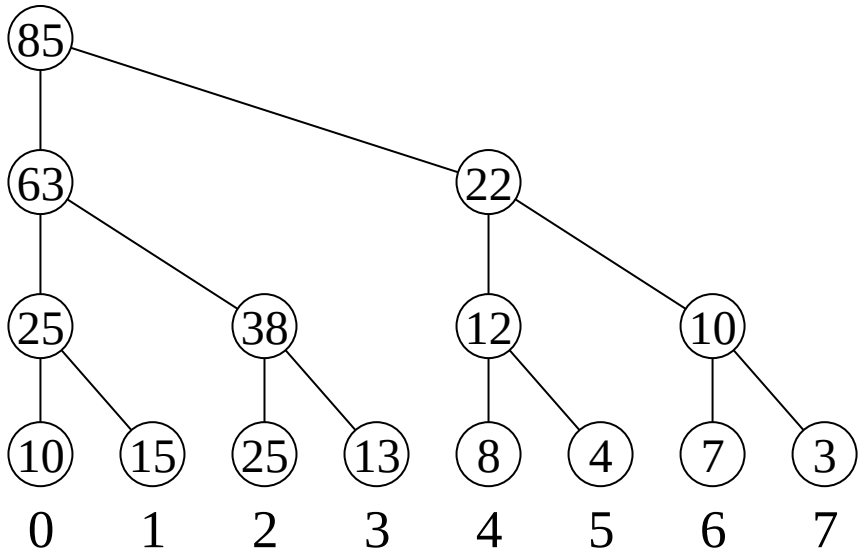
- Send on one link at a time:
  1) Node 0 sends to node 1.
  2) Node 0 sends to node 2, and node 1 sends to node 3.
  3) Node 0 sends to node 4, node 1 sends to node 5, node 2 sends to node 6, and node 3 sends to node 7



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

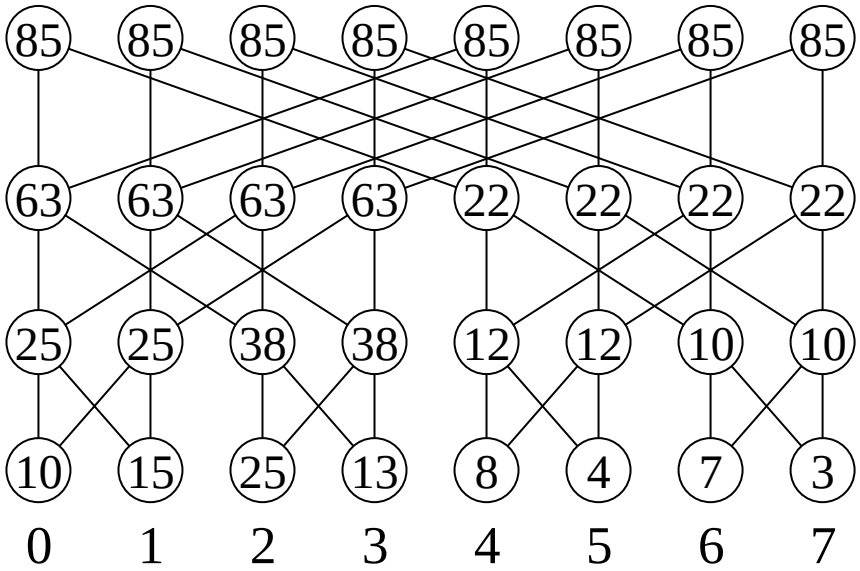
Reduction

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



Reduction To All Nodes

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



Collective Routines

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

MPI Integration Example

Want to find:

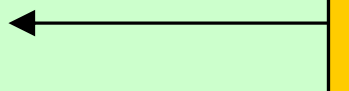
$$\int_0^{\pi} \sin(x)\, dx$$

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

MPI Integration Code: Outline

```
import mpi.*;

class Integrate {
  static public void main(String[] args) {
    int npts;
    MPI.Init(args) ;
    int myrank = MPI.COMM_WORLD.Rank();
    int nprocs = MPI.COMM_WORLD.Size();
    int[] irecv = new int[1];
    if(myrank == 0) {
       npts = 100; irecv[0] = npts;
    }
    MPI.COMM_WORLD.Bcast(irecv, 0, 1, MPI.INT, 0);
    npts = irecv[0];
         ⋮
    MPI.Finalize();
  }
}
```



See next slide for what goes here

MPI Integration Code: Computation

```
npts = irecv[0];
int nlocal = (npts-1)/nprocs + 1;
int nbeg = myrank*nlocal +1;
int nend = Math.min(nbeg+nlocal-1,npts);
double delta = Math.PI/npts;
double psum = 0.0;
for(int i=nbeg;i<=nend;i++){
   psum += (Math.sin((i-0.5)*delta))*delta;
}
double [] dval = new double[2];
dval[0] = psum;
MPI.COMM_WORLD.Reduce(dval,0,dval,1,1,MPI.DOUBLE,MPI.SUM,0);
if (myrank==0){
   System.out.println("The integral = " + dval[1]);
}
```

Application Topologies

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

Cartesian Application Topologies

Cartcomm Intracomm.Create_cart (int [] dims, boolean [] period, boolean reorder)

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

Topological Inquiries

- Can get information about a Cartesian topology:

  CartParms Cartcomm.Get()

  This returns an object containing information about a Cartesian topology:

```
public class CartParms{
    int [] dims;     // number of processes in each dimension
    boolean [] period; // periodicity of each dimension
    int [] coords;  // coordinates of calling process
}
```

Mapping Between Rank and Position

- The rank of a process at a given location:

  int Cartcomm.Rank(int [] coords)

- The location of a process of a given rank:

  int [] Cartcomm.Coords(int rank)

Uses of Topologies

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

Topologies and Data Shifts

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

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

Topologies and Data Shifts 2

- Topological shifts are performed using

  Status Comm.Sendrecv(…)

- The ranks of the processes that a process must send to and receive from when performing a shift on a topological group are returned by:

  ShiftParms Cartcomm.Shift(int direction, int disp)

  where the ShiftParms class is:

```
public ShiftParms{
    public int rankSource;
    public int rankDest;
}
```

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:

  Status Comm.Sendrecv(…)

- For identical send/receive buffers:

  Status Comm.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.

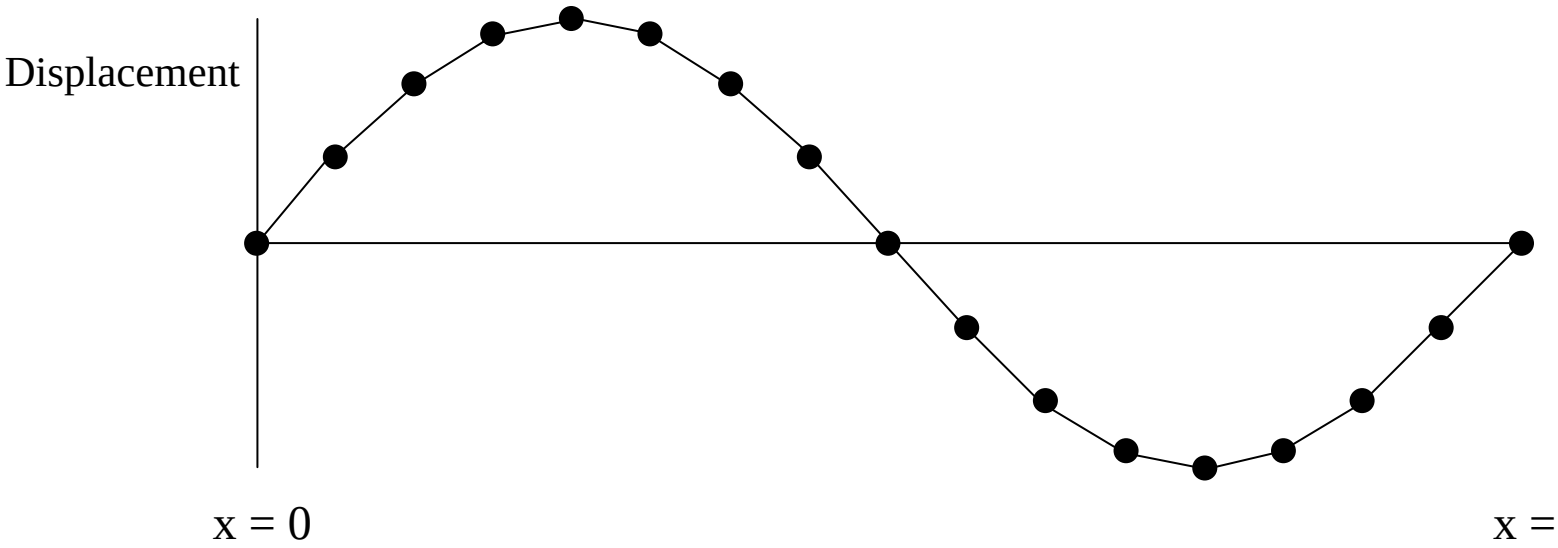
Problem

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

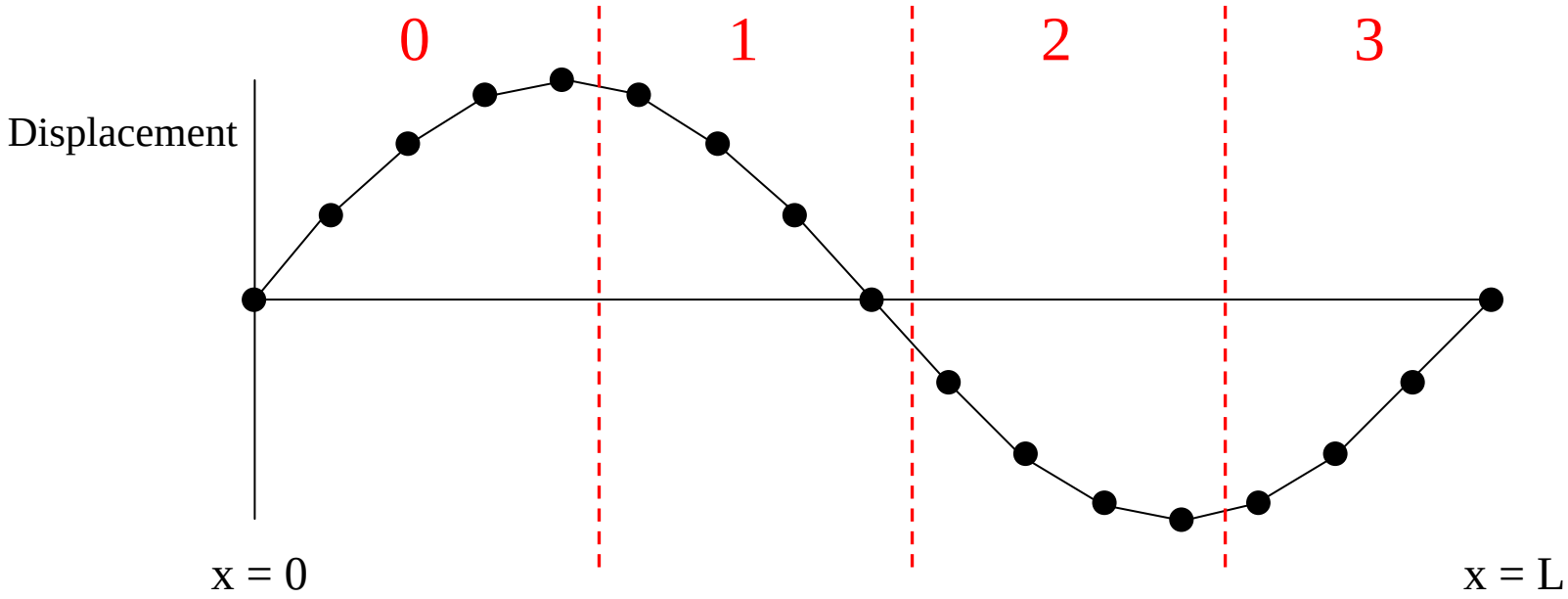


Method of Solution

- We find the solution at a series of time steps, t_0, t_1, t_2, etc.
- At each time step we find the displacement at the points x_0, x_1,…, x_{n-1}, where $x_0=0$ and $x_{n-1}=L$
- At $t_0 = 0$ we assume the string has a known shape, i.e., we know $\Psi(x,0)$.
- Given the solution at position x_i at time t_j, the value there at the next time step depends on the current and previous values at that point, and on current values at the neighbouring points.

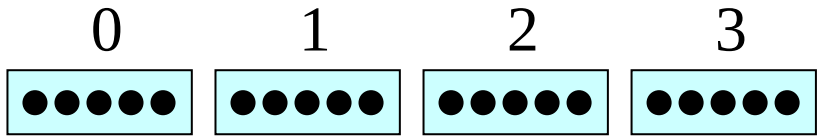
Data Distribution

- 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

- Given the solution at position x_i at time t_j, the value there at the next time step depends on the current and previous values at that point, and on current values at the neighbouring points.
- So to update a point we need to know the displacement at neighbouring points. This entails communication.
- Each process needs to communicate the displacement values for its first and last points before updating its points



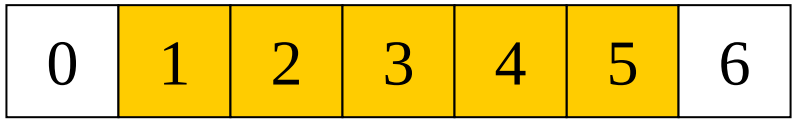
Outline of Parallel Code

- *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 oldpsi arrays.
- *Perform Update*
  - Communicate end points.
  - Do update locally.
- *Output results*

Displacement Arrays

- Each process needs to store the endpoint values received from the neighbouring processes. These are stored at the 0 and nlocal+1 positions in the displacement arrays.
- Thus, the displacement arrays need two “extra” entries at each end.

nlocal = 5



Outline MPI Code

```
import mpi.*

class VibratingString {
  static public void main(String[] args) {
    MPI.Init(args) ;



    MPI.Finalize();
  }
}
```

Code for initialising data distribution goes here

Code for initialising arrays goes here

Code for update loop goes here

Code for output phase goes here

Initialising the Data Distribution

```
boolean [] periods = new boolean[1];
boolean reorder = false;
int myrank = MPI.COMM_WORLD.Rank();
int nprocs = MPI.COMM_WORLD.Size();
int [] dims = new int[1];
dims[0] = nprocs;
periods[0] = false;
Cartcomm comm1d = MPI.COMM_WORLD.Create_cart(dims, periods,
                                             reorder);
int [] coords = new int[1];
coords = comm1d.Coords(myrank);
ShiftParms shift1d = comm1d.Shift(0, 1);
int left  = shift1d.rankSource;
int right = shift1d.rankDest;
```

- comm1d is a new communicator with a 1D Cartesian topology.
- coords array gives the position of a process in the topology.
- Shift() allows us to find the left and right neighbours of a process.

Initialising the Arrays

```
int npoints = 100;
double psi = new double[102];
double oldpsi = new double[102];
double newpsi = new double[102];
int nlocal = npoints/nprocs;
int nstart = coords[0]*nlocal;
double x;
for(int i=0;i<nlocal;i++){
    x = 2.0*Math.PI*(double)(nstart+i)/(double)(npoints-1);
    x = Math.sin(x);
    psi[i+1] = oldpsi[i+1] = x;
}
```

- nlocal is the number of points updated by each process.
- nstart 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 nlocal. Indices 0 and nlocal+1 will be used later to store values received from neighbouring processes.

Update Loop

```
double tau = 0.05;
int start = 1;
if (coords[0]==0) start = 2;
int end = nlocal;
if (coords[0]==nprocs-1) end = nlocal-1;
Status s;
for(int j=0;j<500;j++){
  s = MPI.COMM_WORLD.Sendrecv(psi,1,1,MPI.DOUBLE,left,99,
                              psi,nlocal+1,1,MPI.DOUBLE,right,99);
  s = MPI.COMM_WORLD.Sendrecv(psi,nlocal,1,MPI.DOUBLE,right,99,
                                        psi,0,1,MPI.DOUBLE,left,99);
  for(int i=start;i<=end;i++){
    newpsi[i] = 2.0*psi[i]-oldpsi[i]+
                  tau*tau*(psi[i-1]-2.0*psi[i]+psi[i+1]);
  }
  for(int i=1;i<=nlocal;i++){
    oldpsi[i] = psi[i];
    psi[i] = newpsi[i];
  }
}
```

Notes on Update Loop

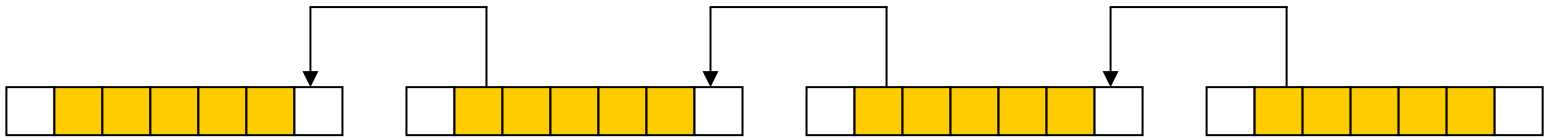
The update phase has 3 main parts:

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

Communication Code: Left Shift

- All processes send psi[1] to the process to the left, and receive data from the process to the right, storing it in psi[nlocal+1].

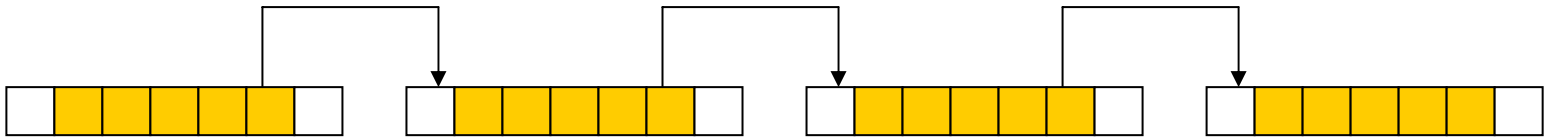
```
s = MPI.COMM_WORLD.Sendrecv(psi,1,1,MPI.DOUBLE,left,99,
                            psi,nlocal+1,1,MPI.DOUBLE,right,99);
```



Communication Code: Right Shift

- All processes send psi[nlocal] to the process to the right, and receive data from the process to the left, storing it in psi[0].

```
s = MPI.COMM_WORLD.Sendrecv(psi,nlocal,1,MPI.DOUBLE,right,99,
                            psi,0,1,MPI.DOUBLE,left,99);
```



Output Phase

- We assume the results are output to a file and/or a visualisation device.
- We won't look at this as its mostly a Java coding issue.
- One parallel computing issue that arises is whether all processes have access to the file system. Usually they do, but this is not required by MPI.

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.
- We ignore the time to copy to the arrays old_psi and psi.

Performance Analysis 2

The speed-up is:

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

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

Performance Analysis 3

The efficiency is

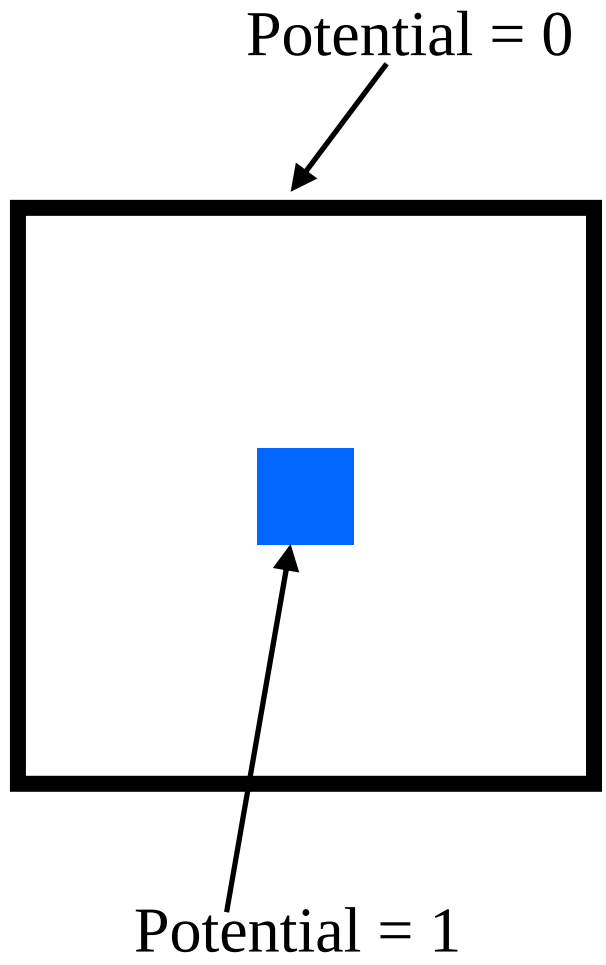
$$\epsilon(N) = \frac{1}{1 + \tau/(3g)}$$

so the overhead is $f(N) = \tau/(3g)$.

Since the efficiency depends on g but not independently on N the parallel algorithm is perfectly scalable.

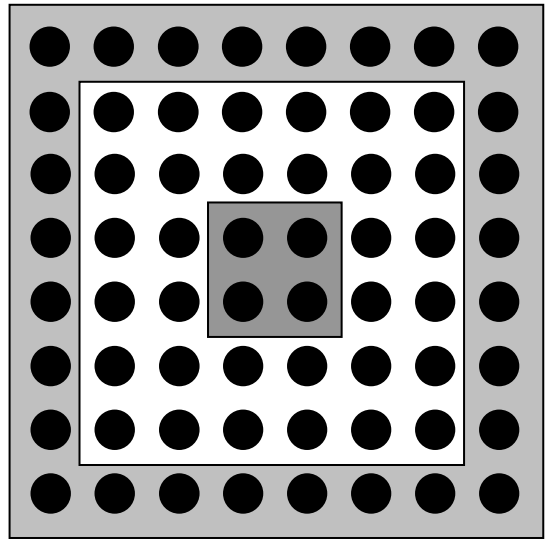
Laplace Equation Problem

- 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 2

- This is a 2-D problem whereas the vibrating string was a 1-D problem.
- We divide the problem domain into a regular grid of points, and find an approximation to the solution at each of these points.

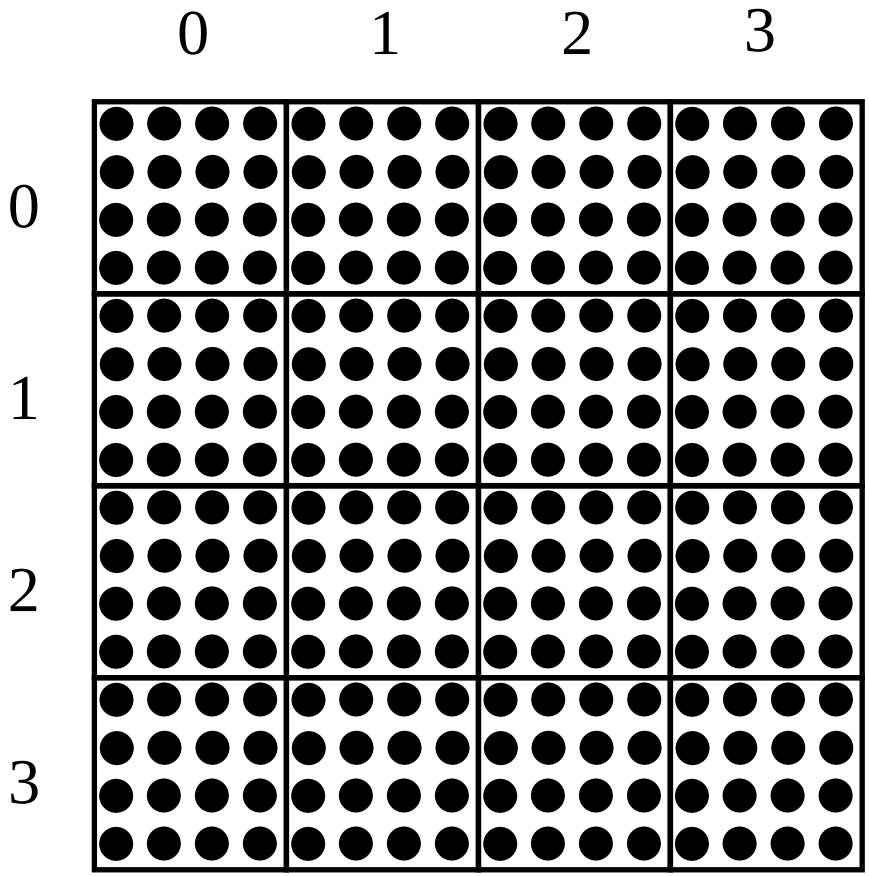


- We start with an initial guess at the solution, and perform a series of iterations that get progressively closer to the solution.

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.

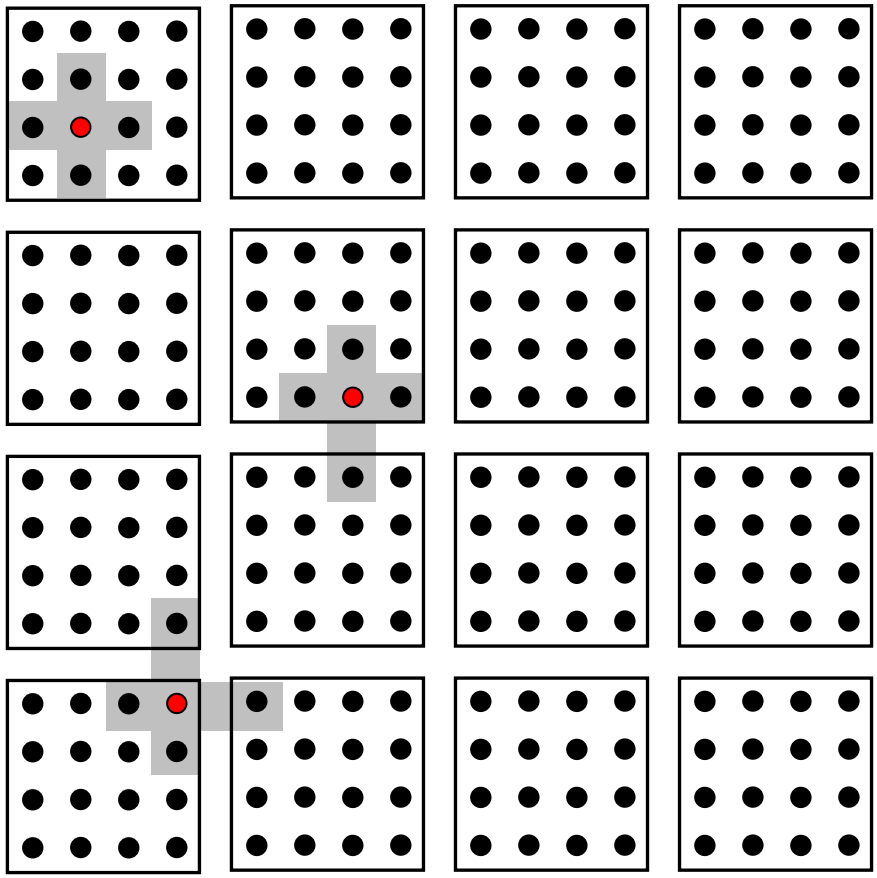
Data Distribution 2



Communication Requirements

- The update formula replaces the solution at a point by 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.

Communication Requirements 2



- To update a red point we need to know the values of the points in the shaded region.
- For points on the edge this requires communication

Outline of Parallel Code

- **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 nlocaly+1 and in columns 0 and nlocalx+1 of the phi array.

(0,0)	(0,1)	(0,2)	(0,3)	(0,4)	(0,5)
(1,0)	(1,1)	(1,2)	(1,3)	(1,4)	(1,5)
(2,0)	(2,1)	(2,2)	(2,3)	(2,4)	(2,5)
(3,0)	(3,1)	(3,2)	(3,3)	(3,4)	(3,5)
(4,0)	(4,1)	(4,2)	(4,3)	(4,4)	(4,5)
(5,0)	(5,1)	(5,2)	(5,3)	(5,4)	(5,5)

nlocalx = 4
nlocaly = 4

Array Initialisation

There are 3 arrays:

- phi : the current values of the solution
- oldphi : the values of the solution for the previous iteration.
- mask : equals false on boundaries and true elsewhere.

```
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
```

phi

```
F F F F F F F F
F T T T T T T F
F T T T T T T F
F T T F F T T F
F T T F F T T F
F T T T T T T F
F T T T T T T F
F F F F F F F F
```

mask

Outline MPI Code

```
import mpi.*;

class LaplaceEquation {
  static public void main(String[] args) {
    MPI.Init(args) ;
                  <-- Code for initialising data distribution goes here
                  <-- Code for initialising arrays goes here
                  <-- Code for update loop goes here
                  <-- Code for output phase goes here
    MPI.Finalize();
  }
}
```

Initialising the Data Distribution

```
boolean [] periods = new boolean[2];
boolean reorder = false;
int myrank = MPI.COMM_WORLD.Rank();
int nprocs = MPI.COMM_WORLD.Size();
int [] dims = new int[2];
MPI.COMM_WORLD.Dims_create(nprocs, dims);
periods[0] = periods[1] = true;
Cartcomm comm2d = MPI.COMM_WORLD.Create_cart(dims, periods,
                                             reorder);
int [] coords = new int[2];
coords = comm2d.Coords(myrank);
ShiftParms vshift = comm2d.Shift(0, 1);
int up    = vshift.rankSource;
int down  = vshift.rankDest;
ShiftParms hshift = comm2d.Shift(1, 1);
int left  = hshift.rankSource;
int right = hshift.rankDest;
```

Initialising the Data Distribution 2

- dims[0] and dims[1] are the number of processes in the process grid in each direction. We make the grid as square as possible using Create_dims().
- This time we set up a 2D communicator, comm2d.
- The Coords() method gives the position in the topology of each process.
- Calls to Shift() give the ranks of the neighbouring processes in the four directions.

Initialising phi and mask Arrays

- Set all of phi to 0, and all of mask to true.
- For processes in row 0 of the process mesh we must set row 1 of the mask array to false.
- For processes in the last row of the process mesh we must set row nlocaly of the mask array to false.
- For processes in column 0 of the process mesh we must set column 1 of the mask array to false.
- For processes in the last column of the process mesh we must set column nlocalx of the mask array to false.
- For the 4 points in the centre we must set the phi and mask entries to 1 and false, respectively, in the processes containing them.

Initialisation of Arrays

```
int nlocalx = 100;
int nlocaly = 100;
double [] phi = new double[102][102];
double [] oldphi = new double[102][102];
boolean [] mask = new boolean[102][102];
double [] sendbuf = new double[102];
double [] recvbuf = new double[102];

for (int j=0;j<=nlocaly+1;j++){
   for (int i=0;i<=nlocalx+1;i++){
      phi[j][i]  = 0.0;
      mask[j][i] = true;
   }
}
```

Initialisation of Arrays 2

```
if (coords[0]==0){
   for(int i=0;i<=nlocalx+1;i++){
      mask[1][i] = false;
   }
}
if (coords[0]==dims[0]-1){
   for(int i=0;i<=nlocalx+1;i++){
      mask[nlocaly][i] = false;
   }
}
if (coords[1]==0){
   for(int j=0;j<=nlocaly+1;j++){
      mask[j][1] = false;
   }
}
if (coords[1]==dims[1]-1){
   for(int j=0;j<=nlocaly+1;j++){
      mask[j][nlocalx] = false;
   }
}
```

Initialisation of Arrays 3

```
if (coords[0]==dims[0]/2-1 && coords[1]==dims[1]/2-1){
   phi[nlocaly][nlocalx] = 1.0;
   mask[nlocaly][nlocalx] = false;
}
if (coords[0]==dims[0]/2-1 && coords[1]==dims[1]/2){
   phi[nlocaly][1] = 1.0;
   mask[nlocaly][1] = false;
}
if (coords[0]==dims[0]/2 && coords[1]==dims[1]/2-1){
   phi[1][nlocalx] = 1.0;
   mask[1][nlocalx] = false;
}
if (coords[0]==dims[0]/2 && coords[1]==dims[1]/2){
   phi[1][1] = 1.0;
   mask[1][1] = false;
}
```

Update Phase

The update phase has three main parts.

- Copy phi to oldphi array.
- Communicate boundary data.
- Update points locally.

Update Phase 2

```
for(int k=0;k<500;k++){
   for(int j=1;j<=nlocaly;j++){
      for(int i=1;i<=nlocalx;i++){
         oldphi[j][i]=phi[j][i];
      }
   }
```

Shift up

Shift down

Shift right

Shift left

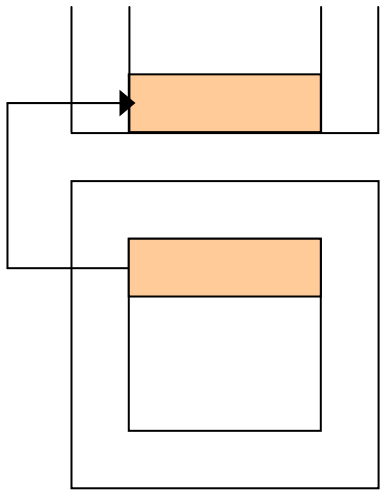
```
   for(int j=1;j<=nlocaly;j++){
      for(int i=1;i<=nlocalx;i++){
         if (mask[j][i]){
           phi[j][i]=0.25*(oldphi[j-1][i]+oldphi[j+1][i]+
                           oldphi[j][i-1]+oldphi[j][i+1]);
         }
      }
   }
```

Communication

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

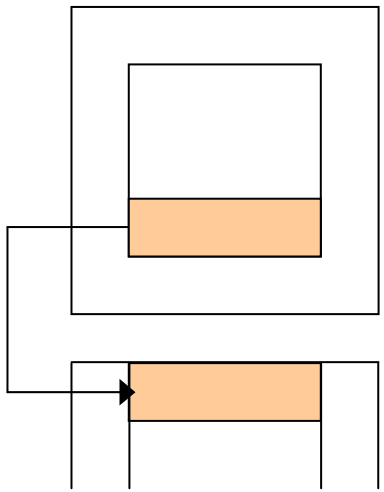
Shift Up

```
Status s;
for(int i=1;i<=nlocalx;i++){
   sendbuf[i] = phi[1][i];
}
s =  MPI.COMM_WORLD.Sendrecv(sendbuf,1,nlocalx,MPI.DOUBLE,up,99,
                             recvbuf,1,102,MPI.DOUBLE,down,99);
for(int i=1;i<=nlocalx;i++){
   oldphi[nlocaly+1][i] = recvbuf[i];
}
```



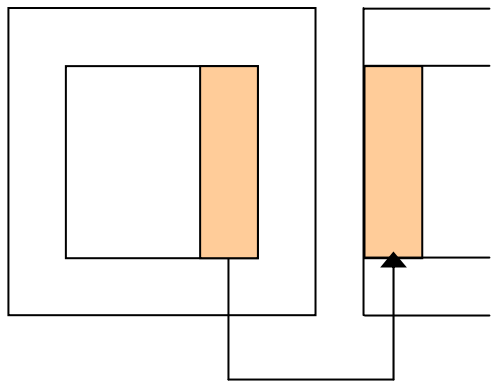
Shift Down

```
Status s;
for(int i=1;i<=nlocalx;i++){
   sendbuf[i] = phi[nlocaly][i];
}
s =  MPI.COMM_WORLD.Sendrecv(sendbuf,1,nlocalx,MPI.DOUBLE,down,
                             99,recvbuf,1,102,MPI.DOUBLE,up,99);
for(int i=1;i<=nlocalx;i++){
   oldphi[0][i] = recvbuf[i];
}
```



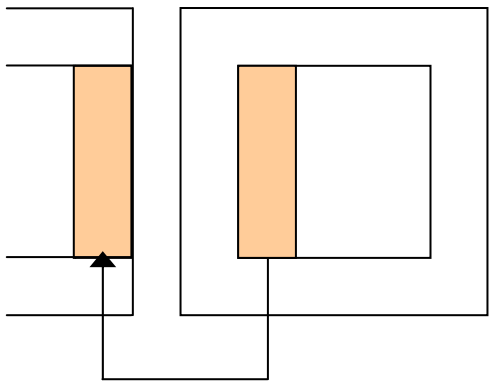
Shift Right

```
Status s;
for(int j=1;j<=nlocaly;j++){
   sendbuf[j] = phi[j][nlocalx];
}
s =  MPI.COMM_WORLD.Sendrecv(sendbuf,1,nlocaly,MPI.DOUBLE,right,
                             99,recvbuf,1,102,MPI.DOUBLE,left,99);
for(int j=1;j<=nlocaly;j++){
   oldphi[j][0] = recvbuf[j];
}
```



Shift Left

```
Status s;
for(int j=1;j<=nlocaly;j++){
   sendbuf[j] = phi[j][1];
}
s =  MPI.COMM_WORLD.Sendrecv(sendbuf,1,nlocaly,MPI.DOUBLE,left,
                             99,recvbuf,1,102,MPI.DOUBLE,right,99);
for(int j=1;j<=nlocaly;j++){
   oldphi[j][nlocalx+1] = recvbuf[j];
}
```



Performance Analysis

- The update formula 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×n is the size of the grid and P is the number of processors in one column 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 one row of the processor mesh.

Speed Up

The speed-up is:

$$S(N) = \frac{4n^2 t_{calc}}{(4n^2/N)t_{calc} + (2n/Q)t_{shift} + (2n/P)t_{shift}}$$

$$= \frac{N}{1 + (P+Q)\,\tau/(2n)}$$

$$= \frac{N}{1 + (Q/n)(1+\alpha)\tau/2}$$

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

Efficiency and Overhead

- Since $N=PQ=\alpha Q^2$ and $M=n^2$ is the number of points, the efficiency is given by:

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

where g = M/N is the grain size.

- Since the efficiency depends only on g, and not independently on n and N, the 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.

Molecular Dynamics Simulations 2

- 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.
- Usually we are interested in the macroscopic properties of such particle systems, such as temperature, energy, etc.

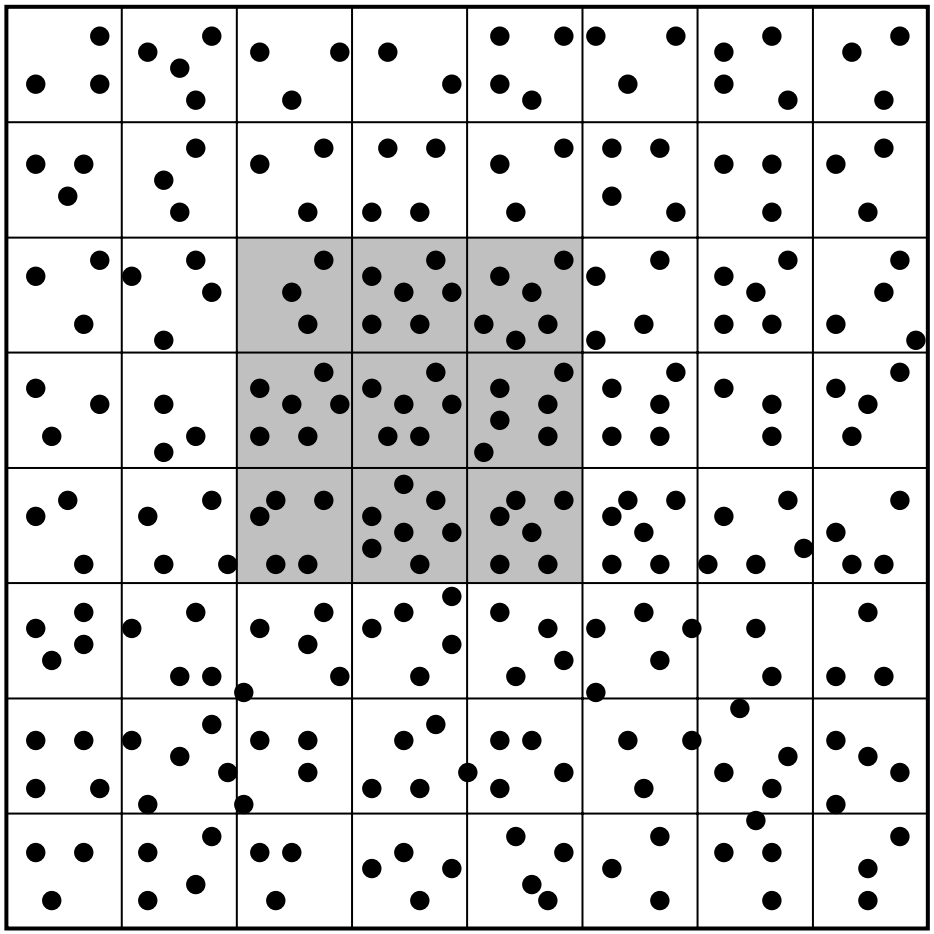
Cut-Off Distance

- 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.

Cut-Off Distance 2



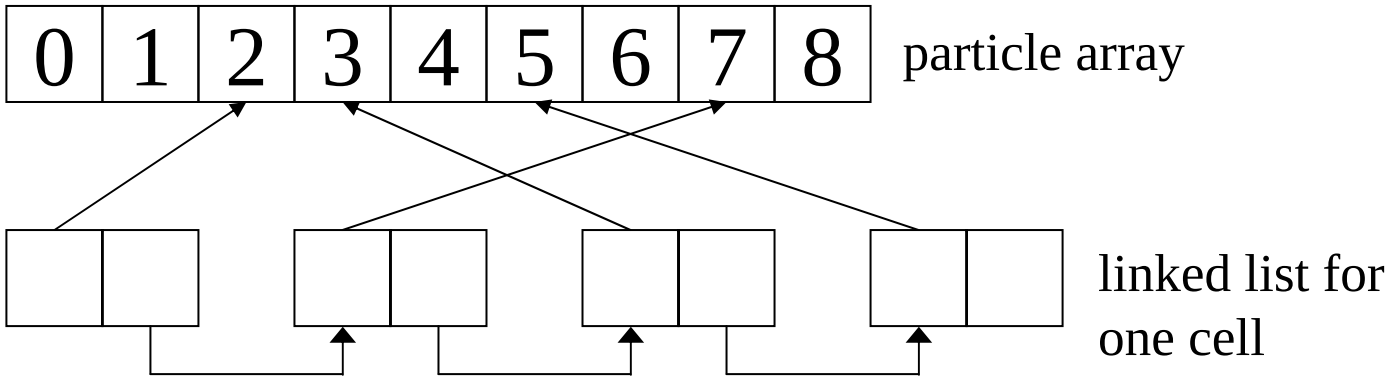
If we divide the domain of the problem into cells of size $r_0 \times r_0$ each particle only interacts with the particles in its own cell and the 8 neighbouring cells.

Data Structures

- The particle data structure contains a particle’s position , velocity, and other data such as particle type, etc.
- Particles can be stored in an array.
- The cell data structure contains a list of the particles it contains, i.e., a list of the array index for each of its particles.
- The cell data can be stored as a linked list.
- There is a 2D array of cells.

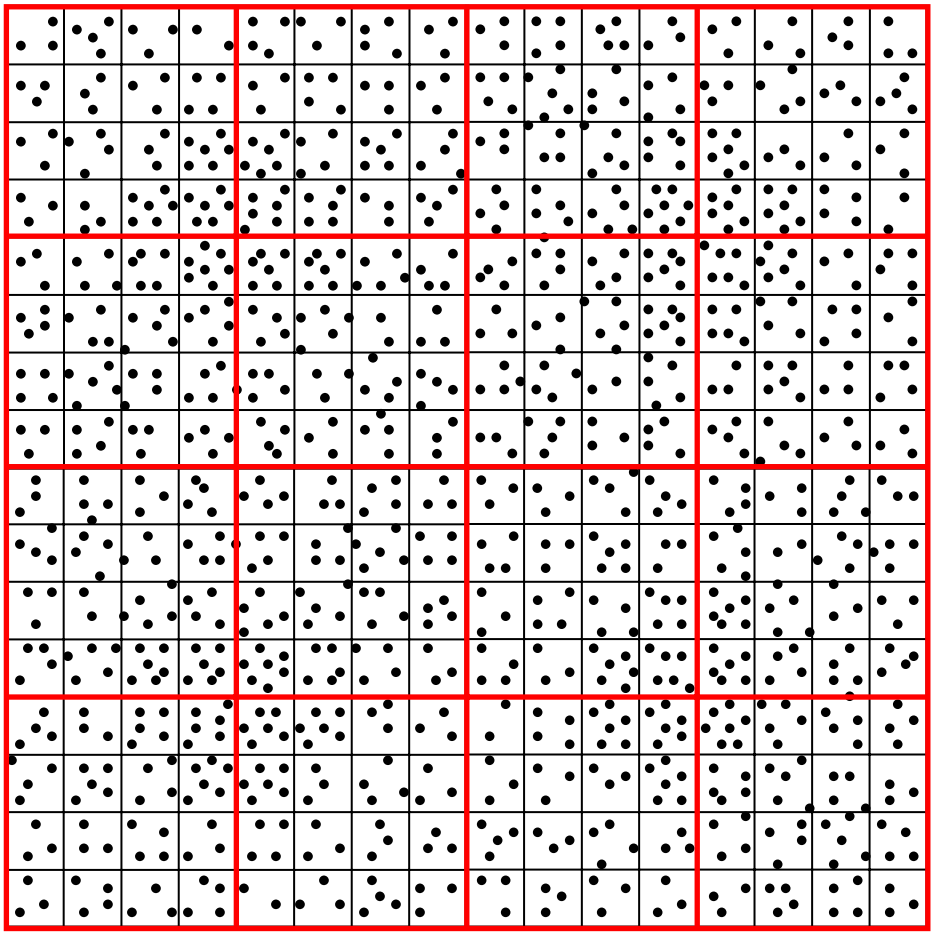
Cell Data Structure

- We can use a linked list to keep track of the particles in each cell.
- We can use Java's LinkedList class.



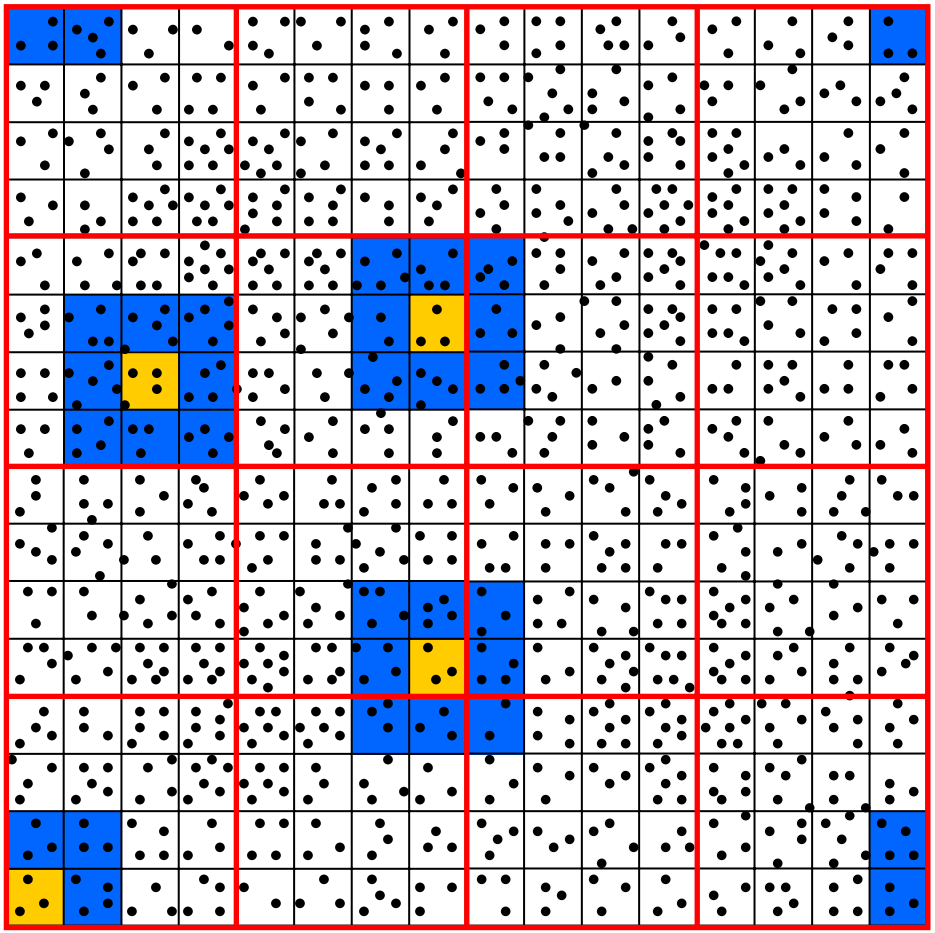
- Each cell has its own list. The starts of the lists are stored in a 2D array.
- A particle can find out which cell it is in from its position.

Data 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



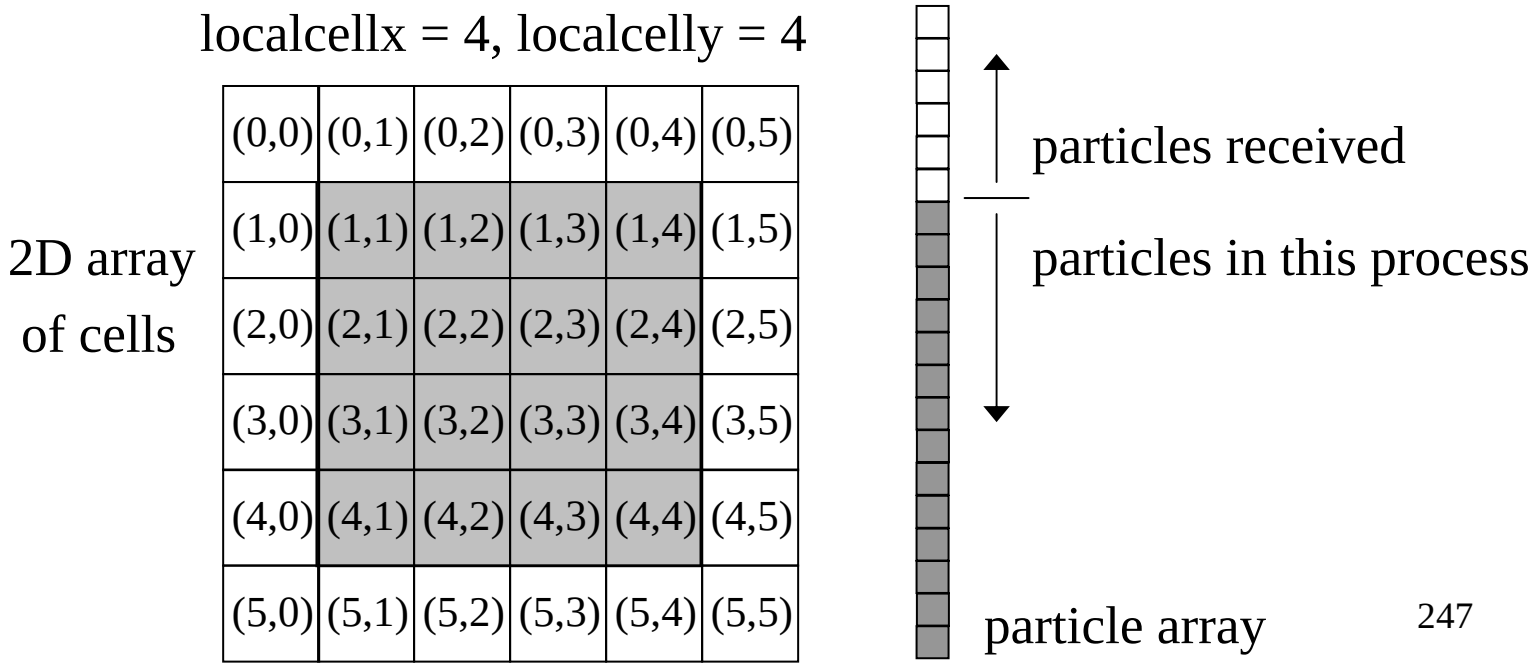
- Particles in cells along the boundary of a process need toknow about particles in other processes in order to evaluate the force on them.
- Each process needs particle data from 8 neighbouring processes

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.



Outline of Parallel Code

- **Initialise data distribution**
  - Find position of each process to determine which block of cells it handles.
  - Find out the node numbers of processes in the left, right, up, and down directions.
- **Initialise particle arrays and cell lists**
  - Generate or input initial particle positions and velocities .
  - Insert particles into cell lists.
- **Perform update**
  - Communicate boundary cell data.
  - Do update locally.
  - Communicate particles that have migrated
- **Output results**

Outline MPI Code

```
import mpi.*

class MolecularDynamics {
  static public void main(String[] args) {
    MPI.Init(args) ;
```

Code for initialising data distribution goes here

Code for initialising arrays and lists goes here

Code for update loop goes here

Code for output phase goes here

```
    MPI.Finalize();
  }
}
```

Initialising the Data Distribution

```
boolean [] periods = new boolean[2];
boolean reorder = false;
int myrank = MPI.COMM_WORLD.Rank();
int nprocs = MPI.COMM_WORLD.Size();
int [] dims = new int[2];
MPI.COMM_WORLD.Dims_create(nprocs, dims);
periods[0] = periods[1] = true;
Cartcomm comm2d = MPI.COMM_WORLD.Create_cart(dims, periods,
                                             reorder);
int [] coords = new int[2];
coords = comm2d.Coords(myrank);
ShiftParms vshift = comm2d.Shift(0, 1);
int up    = vshift.rankSource;
int down  = vshift.rankDest;
ShiftParms hshift = comm2d.Shift(1, 1);
int left  = hshift.rankSource;
int right = hshift.rankDest;
```

Parallel Update Loop

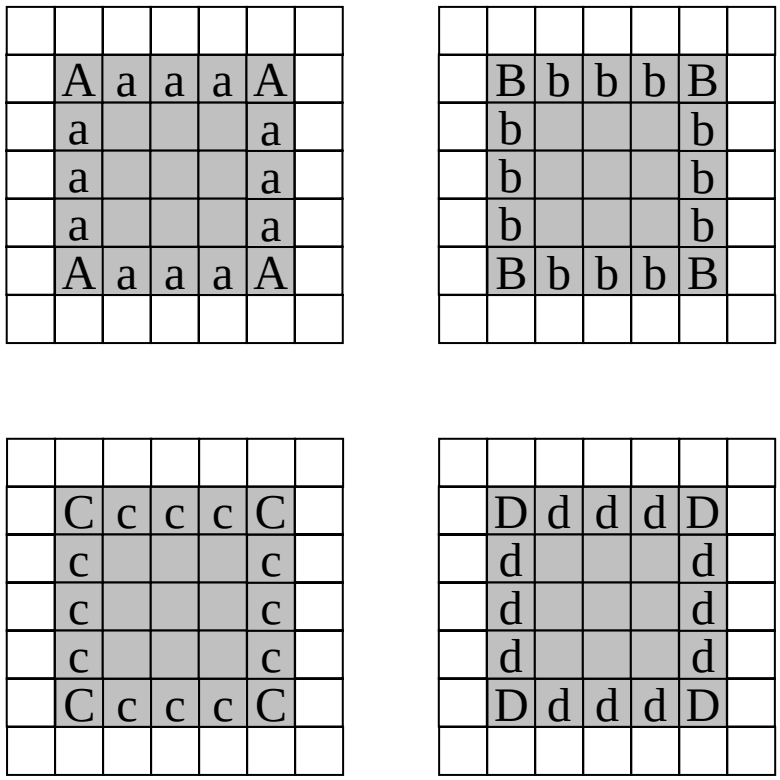
```
for (each time step){
    communicate particle data for boundary cells
    for (each particle, p, in this process){
        find out the location (i,j) of cell p is in
        force[p] = 0;
        for (cell (i,j) and the 8 neighbouring cells){
            for (each particle q in cell){
                add force of q on p to force[p]
            }
         }
    }
    for (each particle, p)
        update velocity and position of p using force[p]
    }
    migrate particles
}
```

Communication of Boundary Cell Data

- The communication is similar to that for the Laplace equation problem except
  - We have to look in the cell lists to see which particles have to be sent and pack this information into a send buffer.
  - The receiving process does not know beforehand how many particles it is going to receive.
  - We have to communicate between diagonally adjacent processes.

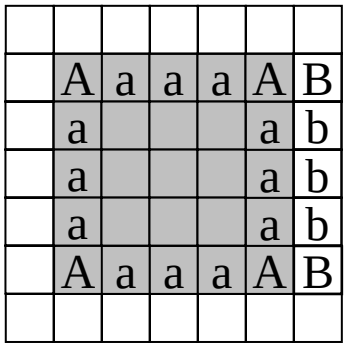
Communicating Corner Data

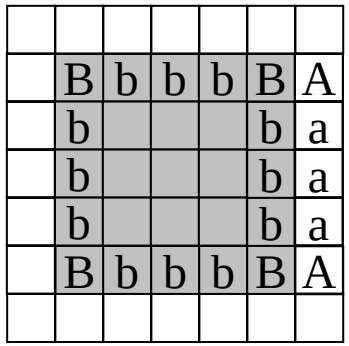
We need to communicate particles in corner cells to diagonally adjacent processes. This can be done be done in 4 shift operations.

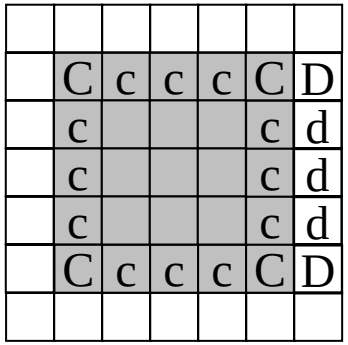


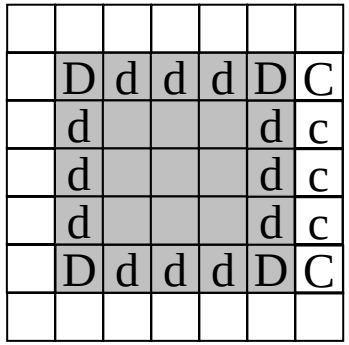
Initial state for 2×2 mesh of processes

Left Shift

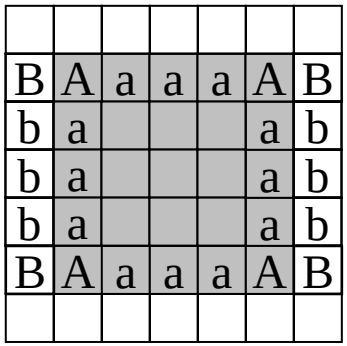

A a a a A B
a a
a a
a a
a a
A a a a A B

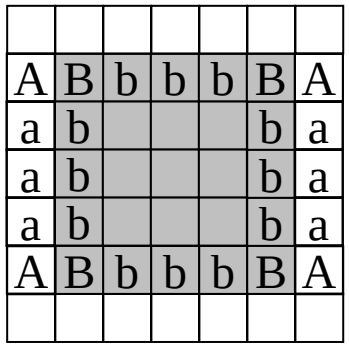

B b b b B A
b b
b b
b b
b b
B b b b B A

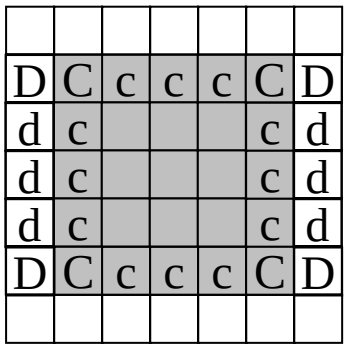

C c c c C D
c c
c c
c c
c c
C c c c C D

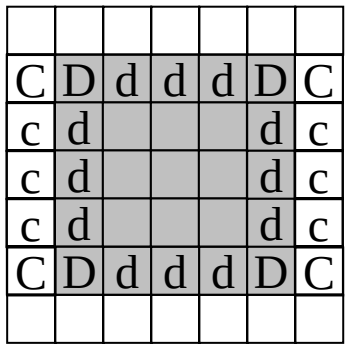

D d d d D C
d d
d d
d d
d d
D d d d D C

Right Shift

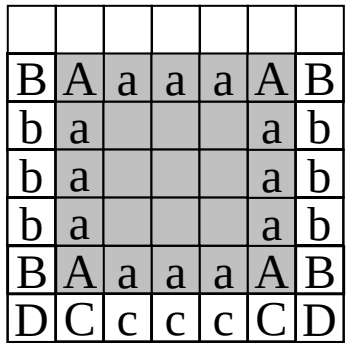

B A a a a A B
b a a
b a a
b a a
B A a a a A B


A B b b b B A
a b b
a b b
a b b
A B b b b B A

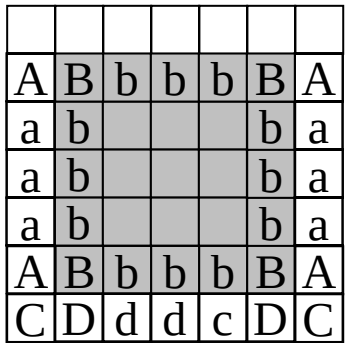

D C c c c C D
d c c
d c c
d c c
D C c c c C D


C D d d d D C
c d d
c d d
c d d
C D d d d D C

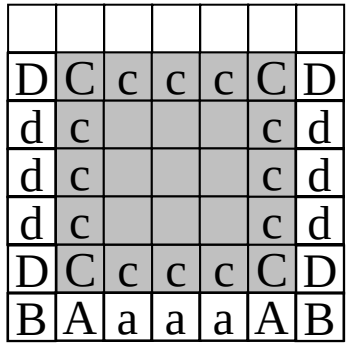
Up Shift



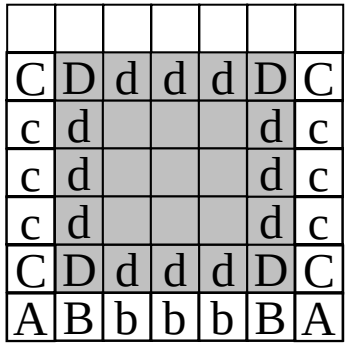
B	A	a	a	a	A	B
b	a				a	b
b	a				a	b
b	a				a	b
B	A	a	a	a	A	B
D	C	c	c	c	C	D



A	B	b	b	b	B	A
a	b				b	a
a	b				b	a
a	b				b	a
A	B	b	b	b	B	A
C	D	d	d	c	D	C



D	C	c	c	c	C	D
d	c				c	d
d	c				c	d
d	c				c	d
D	C	c	c	c	C	D
B	A	a	a	a	A	B



C	D	d	d	d	D	C
c	d				d	c
c	d				d	c
c	d				d	c
C	D	d	d	d	D	C
A	B	b	b	b	B	A

Down Shift

D	C	c	c	c	C	D
B	A	a	a	a	A	B
b	a				a	b
b	a				a	b
b	a				a	b
B	A	a	a	a	A	B
D	C	c	c	c	C	D

C	D	d	d	c	D	C
A	B	b	b	b	B	A
a	b				b	a
a	b				b	a
a	b				b	a
A	B	b	b	b	B	A
C	D	d	d	c	D	C

B	A	a	a	a	A	B
D	C	c	c	c	C	D
d	c				c	d
d	c				c	d
d	c				c	d
D	C	c	c	c	C	D
B	A	a	a	a	A	B

A	B	b	b	b	B	A
C	D	d	d	d	D	C
c	d				d	c
c	d				d	c
c	d				d	c
C	D	d	d	d	D	C
A	B	b	b	b	B	A

Pseudocode for Left Shift

```
nsend = 0
for (i=1 to localcelly){
    for (each particle, p, in cell (i,1))
        pack position of p into sendbuf
        nsend = nsend + 2;
    }
}
Status s = MPI.COMM_WORLD.Sendrecv (
          sendbuf, 1, nsend, MPI_DOUBLE, left, 99,
          recvbuf, 1, recvbuf.length(), MPI_DOUBLE, right, 99);
int nrecv = s.Get_count (MPI_DOUBLE);
for (i=1 to nrecv in steps of 2){
    take next 2 numbers from recvbuf, 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
}
```

Pseudocode for Up Shift

```
nsend = 0
for (i=0 to localcellx+1){
    for (each particle, p, in cell (1,i))
        pack position of p into sendbuf
        nsend = nsend + 2;
    }
}
Status s = MPI.COMM_WORLD.Sendrecv (
          sendbuf, 1, nsend, MPI_DOUBLE, up, 99,
          recvbuf, 1, recvbuf.length(), MPI_DOUBLE, down, 99);
int nrecv = s.Get_count (MPI_DOUBLE);
for (i=1 to nrecv in steps of 2){
    take next 2 numbers from recvbuf, 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
}
```

Particle Migration

- We assume that a particle stays in the same cell or moves to one of the 8 adjacent cells.
- This may involve moving to a cell in another process.
- We have to be able to handle the case where particles move to diagonally adjacent processes.

Pseudocode for Particle Migration 1

```
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
       }
    }
 }
 shift left
 shift right
 shift up
 shift down
```

Pseudocode for Particle Migration 2

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. For left shift:

```
Status s = MPI.COMM_WORLD.Sendrecv(
          leftbuf, 1, nsendleft, MPI_DOUBLE, left, 99,
          recvbuf, 1, recvbuf.length(), MPI_DOUBLE, right, 99);

int nrecv = s.Get_count(MPI_DOUBLE);
for (i=1 to nrecv in steps of 4){
    get next 4 numbers from recvbuf, 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
    }
}
```

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 processes in the boundary data exchange or migration steps. The receiving process determines this with the Get_count() method.
- 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 ocean (hence, *Wa*tery *Tor*us) in which predators (sharks) and prey (fish) compete and survive.
- The parallel implementation of WaTor has a number of interesting features:
  - A very inhomogeneous and dynamic load distribution.
  - The need for irregular communication.
  - The possibility of conflicts between updates performed by different processes (data inconsistency).
- 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, placed at random.
- The populations then evolve in a series of discrete time steps according to certain rules that govern how the fish and sharks move, breed, eat, and die.

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.

Shark Rules

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.

WaTor Inputs

The inputs to the simulation are:

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

WaTor Data Structures

- The two fundamental data structures are the 2D grid of cells and a list of sharks and fish.
- Each cell in the grid is an object with an occupation status (empty, occupied by fish, or occupied by shark), and a reference to the fish or shark object it contains (if not empty).
- Each shark/fish object contains its type, age, cell location, and time since it last ate (if shark).

Update Order

- The order of updating fish and sharks is not specified in the WaTor rules.
- Could update by looping over ocean cell locations. This is inefficient if a significant fraction of the ocean is empty.
- Alternative approach is to process the fish/shark list. In this case only active objects are processed.
- Must ensure that newly-born fish/sharks or not processed until the next time step.

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.

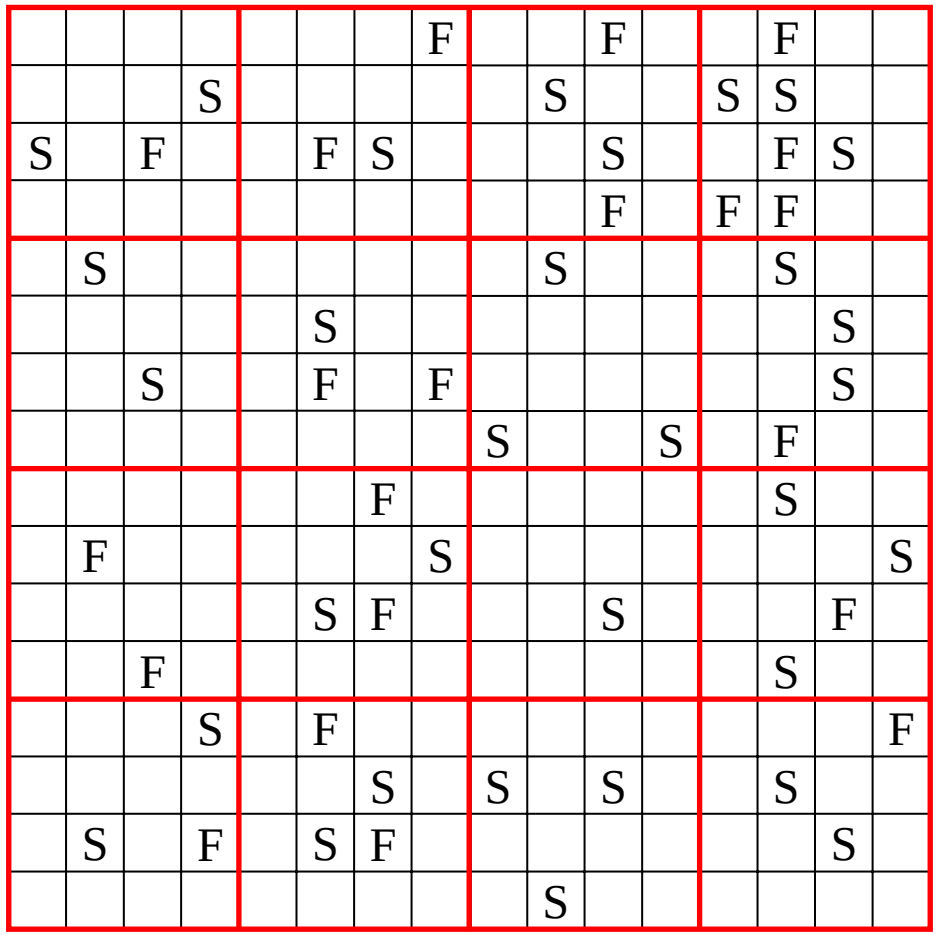
Finalise

- Output final state, statistics, etc.

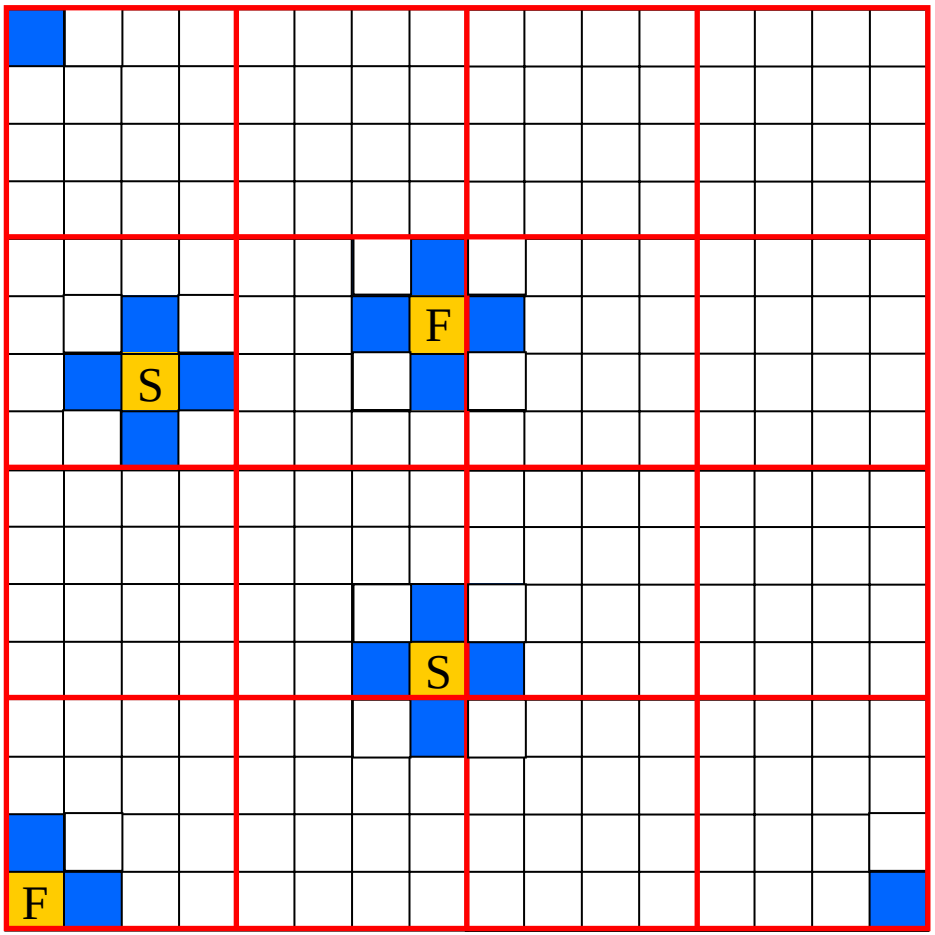
Data Distribution

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

Data Distribution 2



Data Dependencies



- Fish and sharks lying along the boundary of a process need to know about the grid cells lying along the boundary of other processes in order to follow the WaTor rules
- Each process needs data from 4 neighbouring processes.

Array Declarations

- When we receive fishes and sharks from ocean cells lying along the boundary of adjacent processes we store them in the fish/shark list and update the local ocean cell to indicate that is is occupied.

localcellx = 4, localcelly = 4

2D array of cells

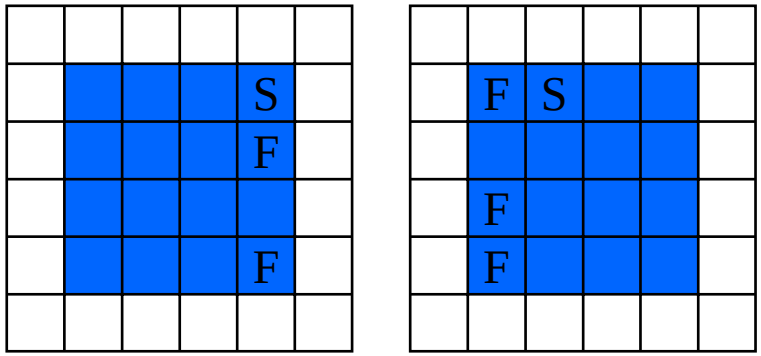
(0,0)	(0,1)	(0,2)	(0,3)	(0,4)	(0,5)
(1,0)	(1,1)	(1,2)	(1,3)	(1,4)	(1,5)
(2,0)	(2,1)	(2,2)	(2,3)	(2,4)	(2,5)
(3,0)	(3,1)	(3,2)	(3,3)	(3,4)	(3,5)
(4,0)	(4,1)	(4,2)	(4,3)	(4,4)	(4,5)
(5,0)	(5,1)	(5,2)	(5,3)	(5,4)	(5,5)

Potential for Data Inconsistency

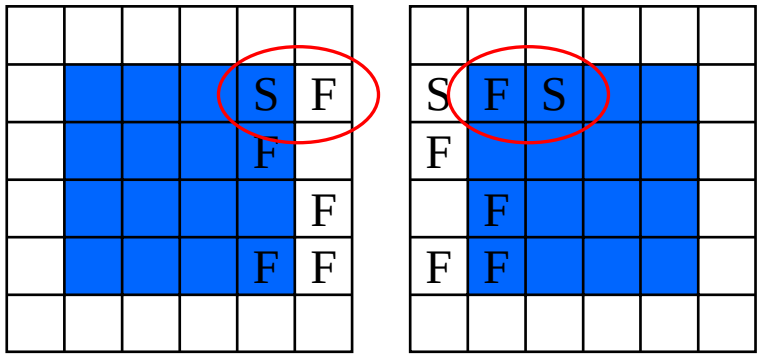
- In the course of an update a fish or shark may move into a “border” grid cell. Thus, unlike in the Laplace and molecular dynamics problems, the border cells in a process may change in an update.
- These changes must be communicated back to the process that originally send that border strip, to be re-integrated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

Example of Data Inconsistency

Time T



Time T+1



- Here we show two adjacent processes at successive time steps.
- Two sharks try to eat the same fish!

Inconsistency and Non-Determinism

- On a shared memory parallel computer this type of data conflict would indicate a non-deterministic algorithm.
- On a distributed memory machine using message passing the communication operations determine the update order for a memory location, so the algorithm is still deterministic – but we would still like to avoid data conflicts.

Rollback

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

1. Return the fish or shark that crossed the process boundary back to its original process, 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.

Rollback 2

- Rollback requires us to remember the previous position of each fish and shark.
- Rollback can result in complicated communication requirements if a sequence of rollbacks traverses multiple processes.
- Rollback has been used in certain event-driven simulations, such as battlefields simulations.

Isolating Boundary Updates

- We could perform updates by looping over the ocean cells instead of by processing the fish/shark list.
- Then we can update all the interior (i.e., non-boundary) cells.
- Next we do a left shift of the lefthand boundary fish/sharks, and update the righthand boundary.
- This is repeated to update the lefthand, upper, and lower boundaries.

Sub-Partitioning

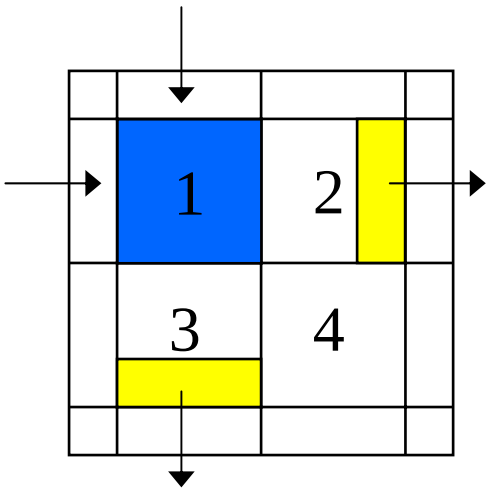
- A third way to avoid data inconsistencies is to partition the part of the ocean assigned to each process into 4 parts.
- A process has a separate fish/shark list for each of these 4 sub-partitions.
- Each of the 4 sub-partitions is updated in turn.
- This avoids adjacent ocean cells being updated concurrently.

Sub-Partitioning Algorithm

1. Divide the ocean array of each process into 4 smaller sub-grids, labelled 1, 2, 3, and 4.
2. Exchange the parts of sub-grids 2 and 3 that have data along their boundaries needed to update sub-grid 1 in adjacent processes.
3. Update sub-grid 1 in each process.
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.

Update Cycle for Sub-Partition 1

Step 1

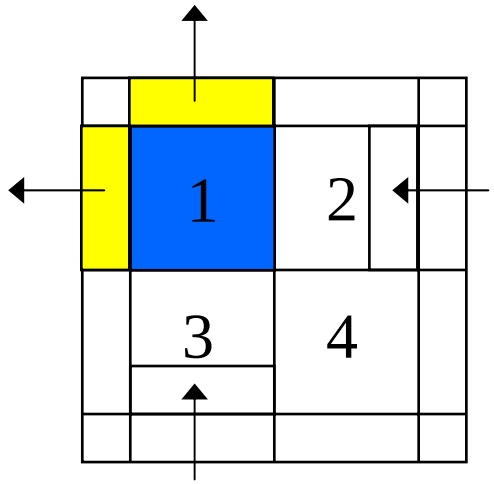


Right shift, down shift

Step 2

Update fish and sharks in list for sub-partition 1

Step 3



Left shift, up shift

Communication Phases

- To update sub-partition 1: shift right and down before update, then shift left and up after update.
- To update sub-partition 2: shift left and down before update, then shift right and up after update.
- To update sub-partition 3: shift right and up before update, then shift left and down after update.
- To update sub-partition 4: shift left and up before update, then shift right and down after update.

Outline of Parallel Code

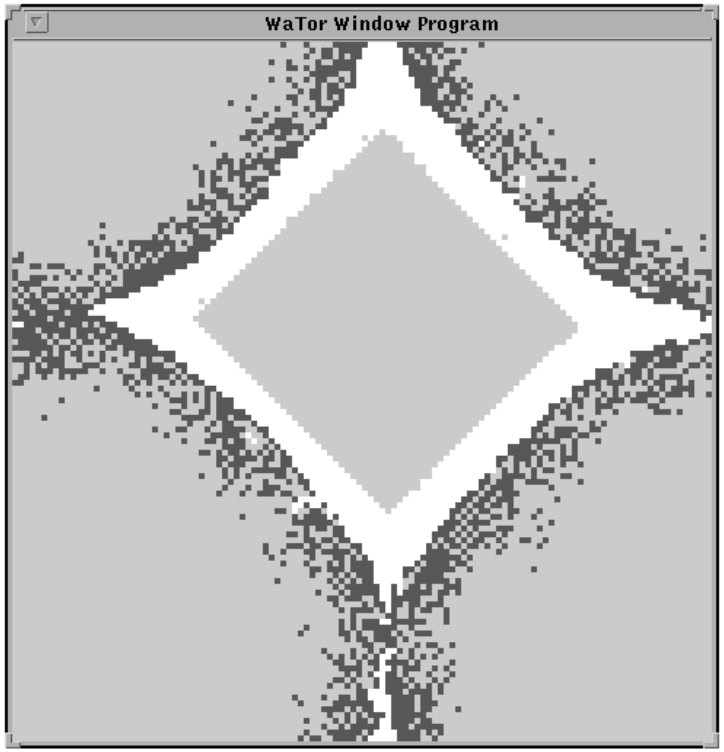
The update loop of the parallel version of WaTor using sub-grids is as follows:

```
for (each time step){
    for (each sub-partition, i=1,2,3,4){
        shift boundary data across 2 edges of sub-partition i
        store data received in border of ocean and in
            fish/shark list
        update fish and sharks in sub-partition i
        shift boundary data back across the 2 edges,
            overwriting original data with updated data
    }
}
```

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 distributing the data in contiguous blocks means that some processes have less work than others at certain times.
- Load balance in WaTor changes with time as the fish and sharks move.

Example of WaTor Output



- White = empty
- Light grey = fish
- Dark grey = sharks

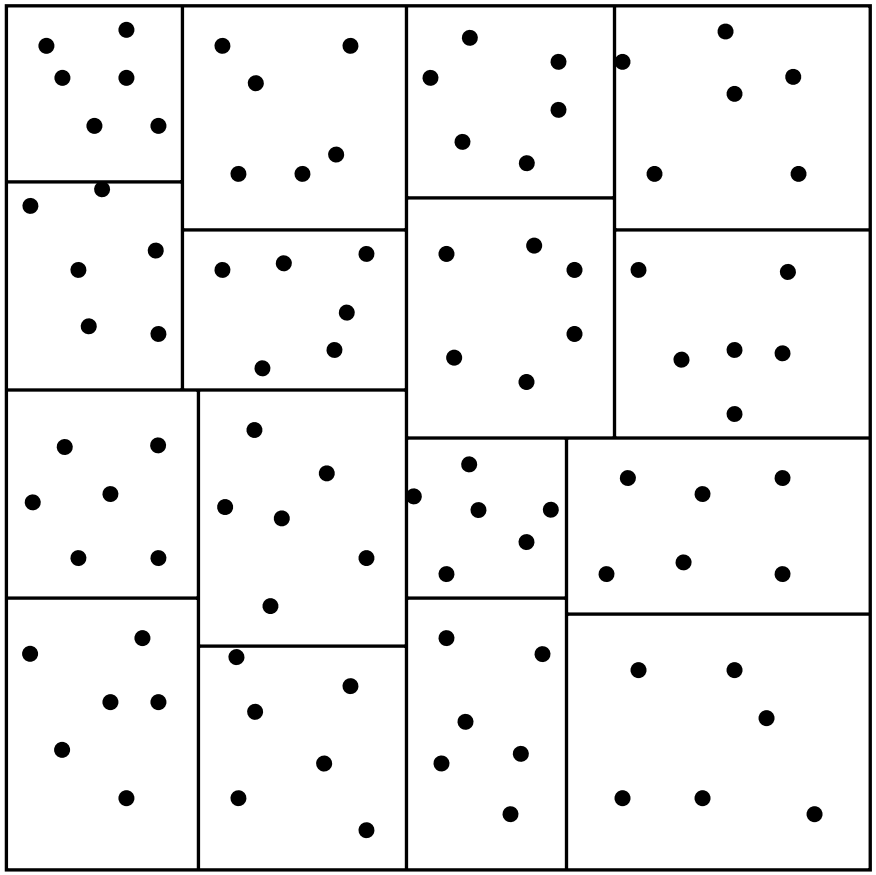
Dynamic Load Imbalance

- In dealing with dynamic load imbalance the following two approaches are important:
- Use of a dynamic load balancer so that the distribution of the ocean among the processes changes as the fish and shark system evolves. When dealing with grids some form of *recursive bisection* is often used.
- Use of a *cyclic*, or *scattered*, data distribution. The parts of the grid assigned to one process 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

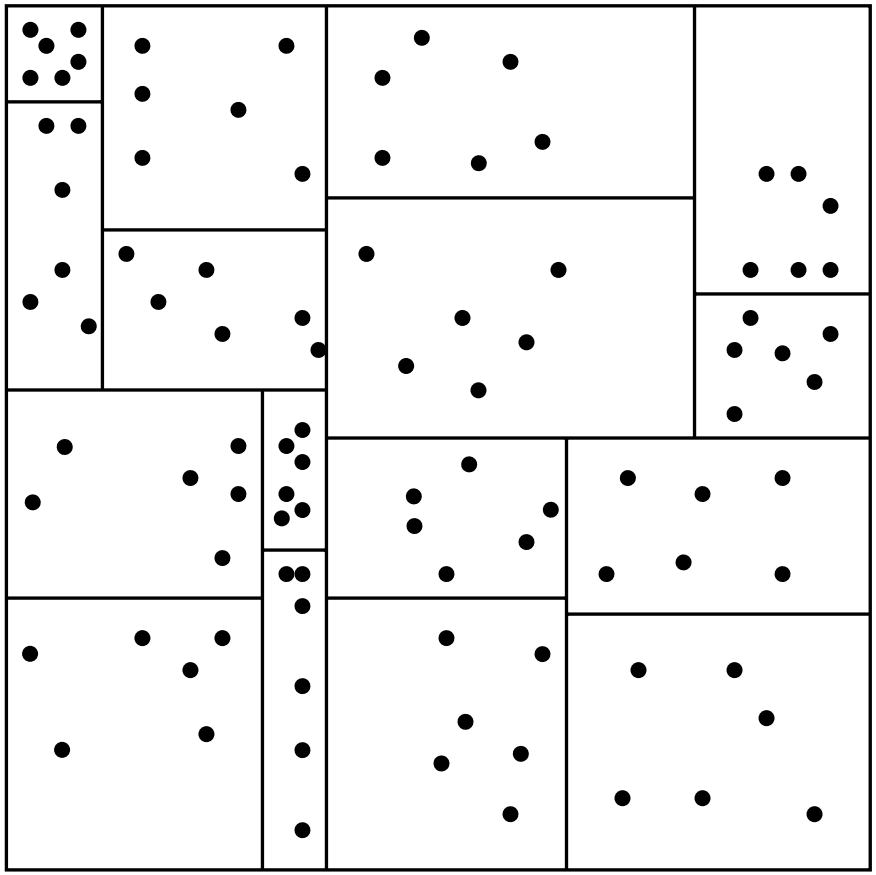
- Orthogonal Recursive Bisection (ORB) first divides the domain orthogonal to the x-direction so there are equal numbers of items in each of the two subdomains.
- Then each of these 2 subdomains is independently divided orthogonal to the y-direction, to give 4 subdomains each with approximately the same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

Example of ORB 1



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.

Example of ORB 2



If the items are distributed unevenly over the domain, ORB can give rise to a variety of different shaped process subdomains.

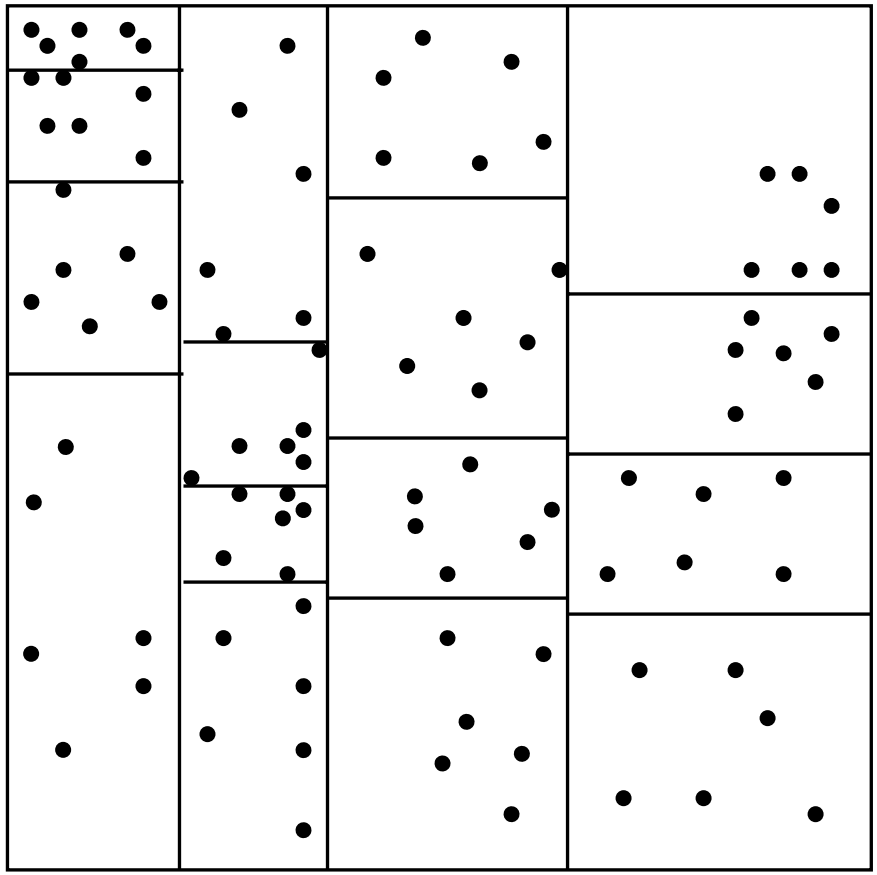
Notes on ORB

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 processes.

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.

Example of HRB



Cyclic Data Distributions

- In a cyclic data distribution the data assigned to each process is scattered in a regular way over the domain of the problem.
- The figure on the next slide shows how a grid might be cyclically distributed over a 4×4 mesh of processes.
- 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 process.

(0,0)	(0,1)	(0,2)	(0,3)	(0,0)	(0,1)	(0,2)	(0,3)	(0,0)	(0,1)	(0,2)	(0,3)
(1,0)	(1,1)	(1,2)	(1,3)	(1,0)	(1,1)	(1,2)	(1,3)	(1,0)	(1,1)	(1,2)	(1,3)
(2,0)	(2,1)	(2,2)	(2,3)	(2,0)	(2,1)	(2,2)	(2,3)	(2,0)	(2,1)	(2,2)	(2,3)
(3,0)	(3,1)	(3,2)	(3,3)	(3,0)	(3,1)	(3,2)	(3,3)	(3,0)	(3,1)	(3,2)	(3,3)
(0,0)	(0,1)	(0,2)	(0,3)	(0,0)	(0,1)	(0,2)	(0,3)	(0,0)	(0,1)	(0,2)	(0,3)
(1,0)	(1,1)	(1,2)	(1,3)	(1,0)	(1,1)	(1,2)	(1,3)	(1,0)	(1,1)	(1,2)	(1,3)
(2,0)	(2,1)	(2,2)	(2,3)	(2,0)	(2,1)	(2,2)	(2,3)	(2,0)	(2,1)	(2,2)	(2,3)
(3,0)	(3,1)	(3,2)	(3,3)	(3,0)	(3,1)	(3,2)	(3,3)	(3,0)	(3,1)	(3,2)	(3,3)
(0,0)	(0,1)	(0,2)	(0,3)	(0,0)	(0,1)	(0,2)	(0,3)	(0,0)	(0,1)	(0,2)	(0,3)
(1,0)	(1,1)	(1,2)	(1,3)	(1,0)	(1,1)	(1,2)	(1,3)	(1,0)	(1,1)	(1,2)	(1,3)
(2,0)	(2,1)	(2,2)	(2,3)	(2,0)	(2,1)	(2,2)	(2,3)	(2,0)	(2,1)	(2,2)	(2,3)
(3,0)	(3,1)	(3,2)	(3,3)	(3,0)	(3,1)	(3,2)	(3,3)	(3,0)	(3,1)	(3,2)	(3,3)

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

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

Cyclic[1] Data Mappings

- The global index, m, maps to a process location, p, and a local index, i.

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

where p and i are given by:

$$p = m \pmod N$$

$$i = \text{floor}(m/N)$$

and N is the number of processes. The inverse mapping is:

$$m = iN + p$$

Cyclic[k] Data Mappings

- If we arrange array entries in groups of size k and cyclically 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

Cyclic[k] Data Mappings 2

Global index m is mapped to process location p, local block index b, and local index i within the block, as follows:

$$p = B \pmod N$$

$$b = \text{floor}(B/N)$$

$$i = m \pmod k$$

where B=floor(m/k) is the global block index. The inverse mapping is:

$$m=(bN+p)k + i$$

Block Data Distributions

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

$$p = \text{floor}(m/T)$$

$$i = m \pmod T$$

where T=ceil(M/N), M is the number of items, and N the number of processes.

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

The inverse mapping is:

$$m = pT + i$$

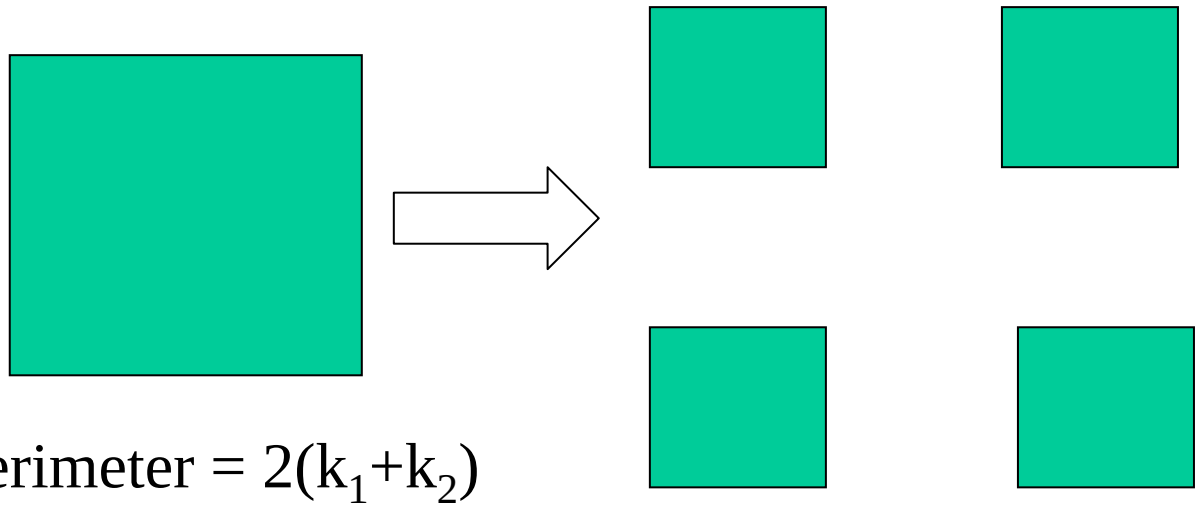
Communication and Load Imbalance Tradeoff

- A block cyclic data distribution can be used to improve load balance when data is distributed inhomogeneously across the problem domain.
- However, a smaller block size results in more boundary data and hence gives rise to increased communication.
- There is, therefore, a tradeoff between load imbalance and communication cost.
- It is important to choose the correct block size so that the total overhead is minimised.

Example

- Assume that the amount of communication associated with a block is proportional to its perimeter.
- Suppose we have a 2-D block cyclic distribution with block size k_1 by k_2.
- Now we reduce the block size by a factor of 2 in each direction, so each block in the original data distribution is split into 4 blocks, each of size $k_1/2$ by $k_2/2$.

Example (continued)



Perimeter = $2(k_1+k_2)$

Perimeter = $2(k_1/2+k_2/2) = k_1+k_2$

- The perimeter of the original block is $2(k_1+k_2)$.
- After it is split into 4 smaller blocks the total perimeter of these blocks is $4(k_1+k_2)$.
- So, for a 2D problem, we expect the communication cost to double when the block size is halved in each direction

Multi-Dimensional 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 $P \times Q$ process mesh, and (i,j) is the index into the local 2D array.
- Different data distributions can be applied over each array dimension.

Multi-Dimensional Data Distributions 2

- For a 2D (cyclic[1],cyclic[1]) data distribution we would have:

$$m \mapsto (p,i) = (m(\text{mod } P), \text{floor}(m/P))$$
$$n \mapsto (q,j) = (n(\text{mod } Q), \text{floor}(n/Q))$$

(0,0)	(0,1)	(0,2)	(0,3)	(0,0)	(0,1)	(0,2)	(0,3)	(0,0)	(0,1)	(0,2)	(0,3)
(1,0)	(1,1)	(1,2)	(1,3)	(1,0)	(1,1)	(1,2)	(1,3)	(1,0)	(1,1)	(1,2)	(1,3)
(2,0)	(2,1)	(2,2)	(2,3)	(2,0)	(2,1)	(2,2)	(2,3)	(2,0)	(2,1)	(2,2)	(2,3)
(3,0)	(3,1)	(3,2)	(3,3)	(3,0)	(3,1)	(3,2)	(3,3)	(3,0)	(3,1)	(3,2)	(3,3)
(0,0)	(0,1)	(0,2)	(0,3)	(0,0)	(0,1)	(0,2)	(0,3)	(0,0)	(0,1)	(0,2)	(0,3)
(1,0)	(1,1)	(1,2)	(1,3)	(1,0)	(1,1)	(1,2)	(1,3)	(1,0)	(1,1)	(1,2)	(1,3)
(2,0)	(2,1)	(2,2)	(2,3)	(2,0)	(2,1)	(2,2)	(2,3)	(2,0)	(2,1)	(2,2)	(2,3)
(3,0)	(3,1)	(3,2)	(3,3)	(3,0)	(3,1)	(3,2)	(3,3)	(3,0)	(3,1)	(3,2)	(3,3)
(0,0)	(0,1)	(0,2)	(0,3)	(0,0)	(0,1)	(0,2)	(0,3)	(0,0)	(0,1)	(0,2)	(0,3)
(1,0)	(1,1)	(1,2)	(1,3)	(1,0)	(1,1)	(1,2)	(1,3)	(1,0)	(1,1)	(1,2)	(1,3)
(2,0)	(2,1)	(2,2)	(2,3)	(2,0)	(2,1)	(2,2)	(2,3)	(2,0)	(2,1)	(2,2)	(2,3)
(3,0)	(3,1)	(3,2)	(3,3)	(3,0)	(3,1)	(3,2)	(3,3)	(3,0)	(3,1)	(3,2)	(3,3)

Load Balancing Issues in a Parallel Cellular Automata Application

- This looks at an application that uses a cyclic data distribution to achieve static load balance.
- As in WaTor, data inconsistency in how updates are performed is an issue

CA for Surface Reactions

- A cellular automaton is used to model the reaction of carbon monoxide and oxygen to form carbon dioxide

$$CO + O \leftrightharpoons CO_2$$

- Reactions take place on surface of a crystal which serves as a catalyst.

The Problem Domain

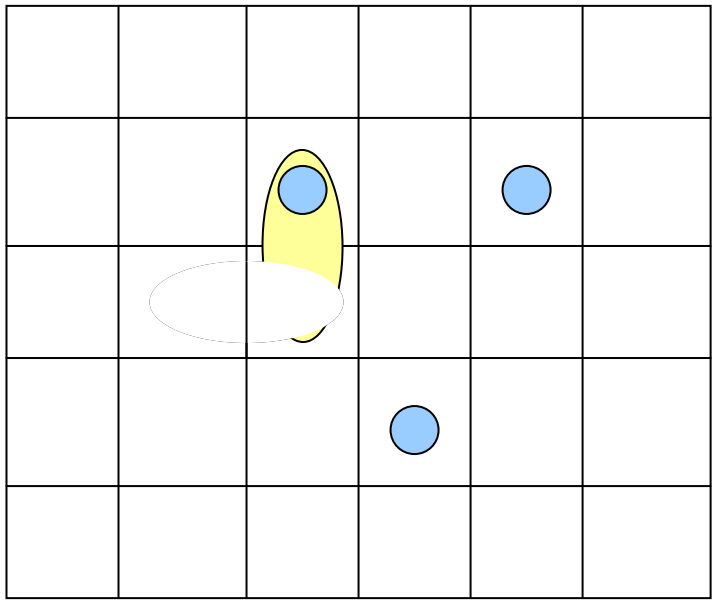
- The problem domain is a periodic square lattice representing the crystal surface.
- CO and O_2 are adsorbed onto the crystal surface from the gas phase.
- Parameter y is the fraction of CO and 1-y is the fraction of O_2.

Interaction Rules

- Choose a lattice site at random and attempt to place a CO or an O_2 there with probabilities y and 1-y, respectively.
- If site is occupied then the CO or O_2 bounces off, and a new trial begins.
- O_2 disassociates so we have to find 2 adjacent sites for these.
- The following rules determine what happens next.

Interaction Rules for CO

1. CO adsorbed
2. Check 4 neighbors for O
3. CO and O react
4. CO_2 desorbs

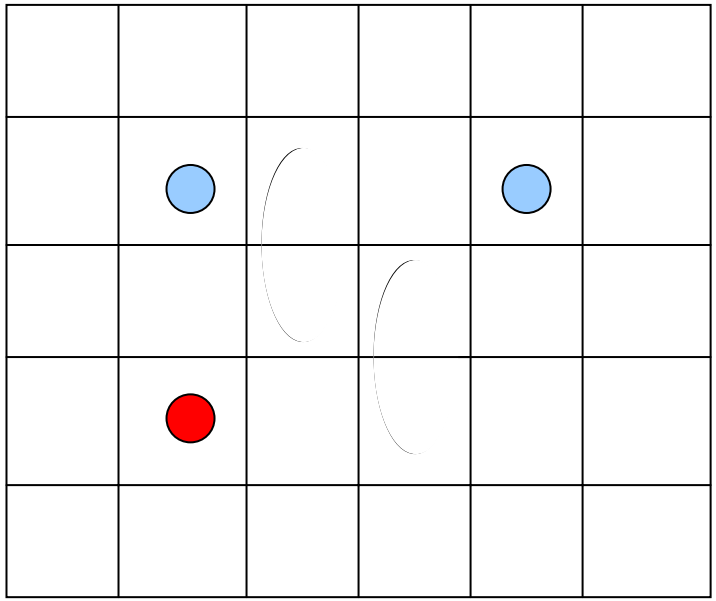


oxygen

CO

Interaction Rules for O

1. O_2 adsorbed
2. O_2 disassociates
3. Check 6 neighbors for CO
4. O and CO react
5. CO_2 desorbs



oxygen
CO
O_2

Parallel Version of Code

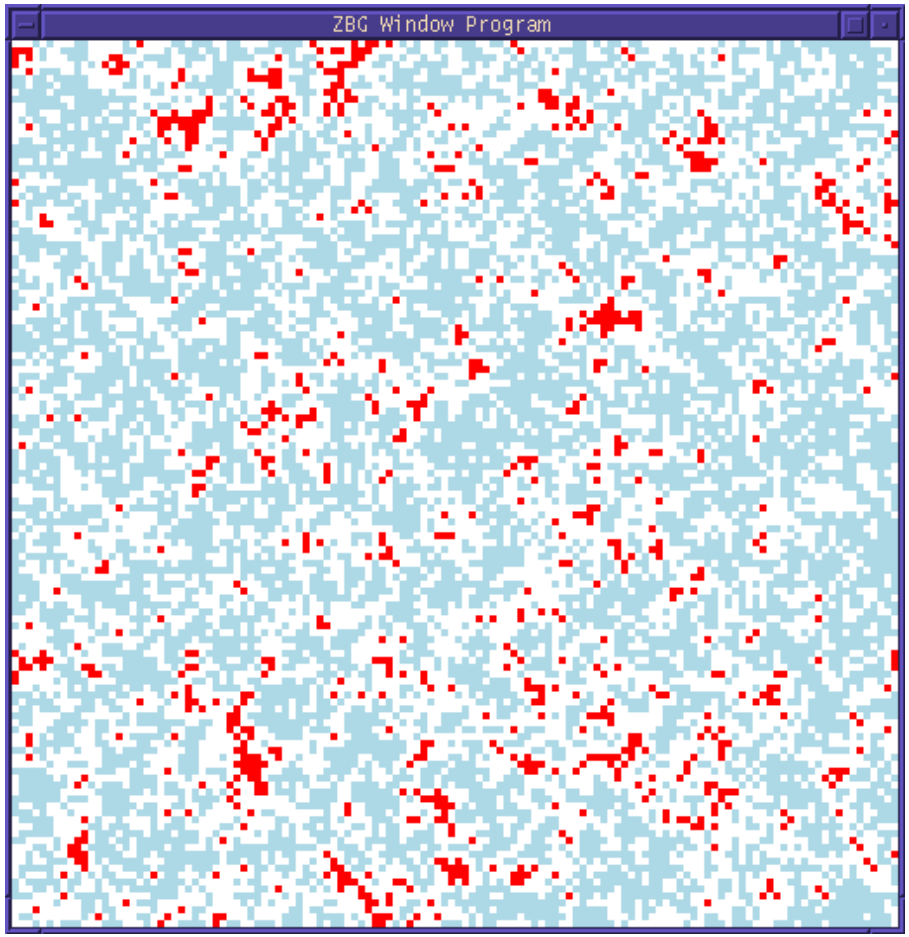
- As simulation evolves the distribution of molecules may become very uneven.
- This results in load imbalance.
- Use a 2-D block cyclic data distribution for the lattice.
- This will give statistical load balance, but smaller block sizes will result in more communication.

Steady State Reaction

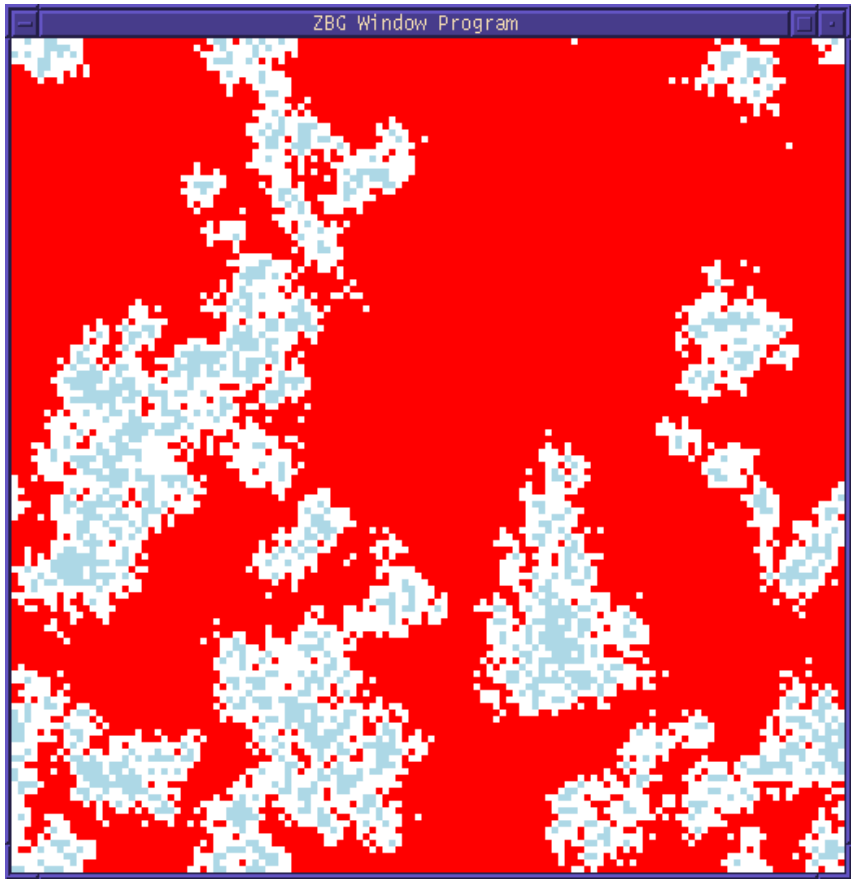
For $y_1<y<y_2$ we get a steady state.

$y_1 \cong 0.39$

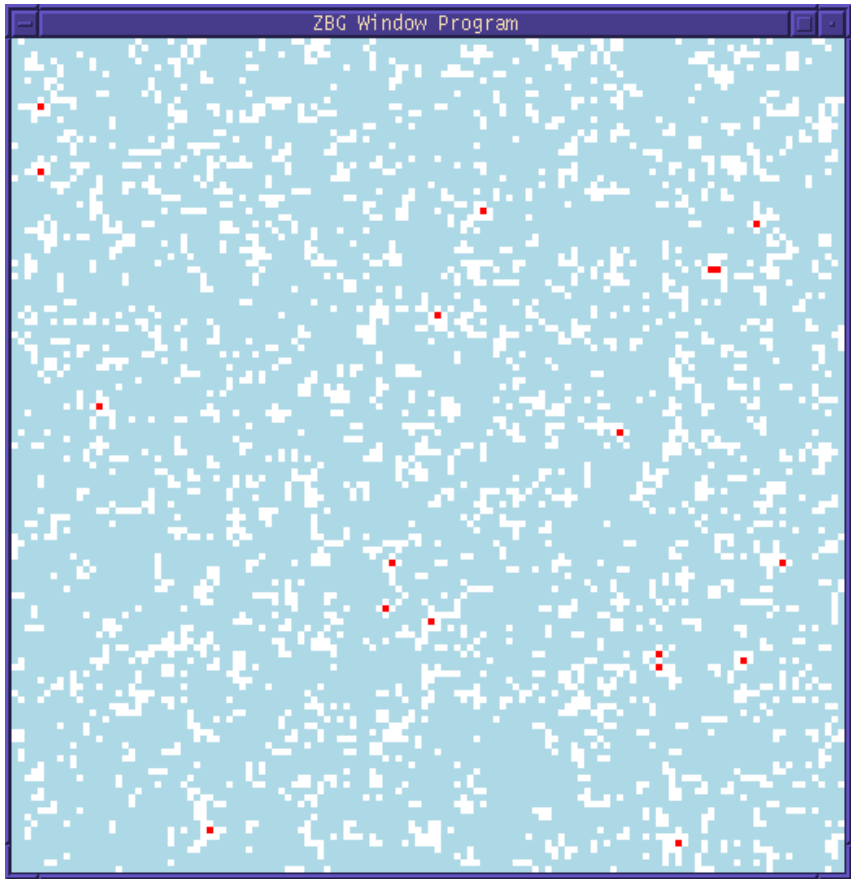
$y_2 \cong 0.53$



CO Poisoning: $y > y_2$



Oxygen Poisoning: $y < y_1$



Main Issues

- MPI used – user-defined datatypes were important in performing communication.
- There is a trade-off between load imbalance and communication.
- A block-cyclic data distribution is used.
- Performance can be modelled.

Block-Cyclic Data Distribution

Block-cyclic data distribution improves load balance by scattering processes over the lattice in a regular way.

Block size is k_r x k_c

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

Parallel Implementation

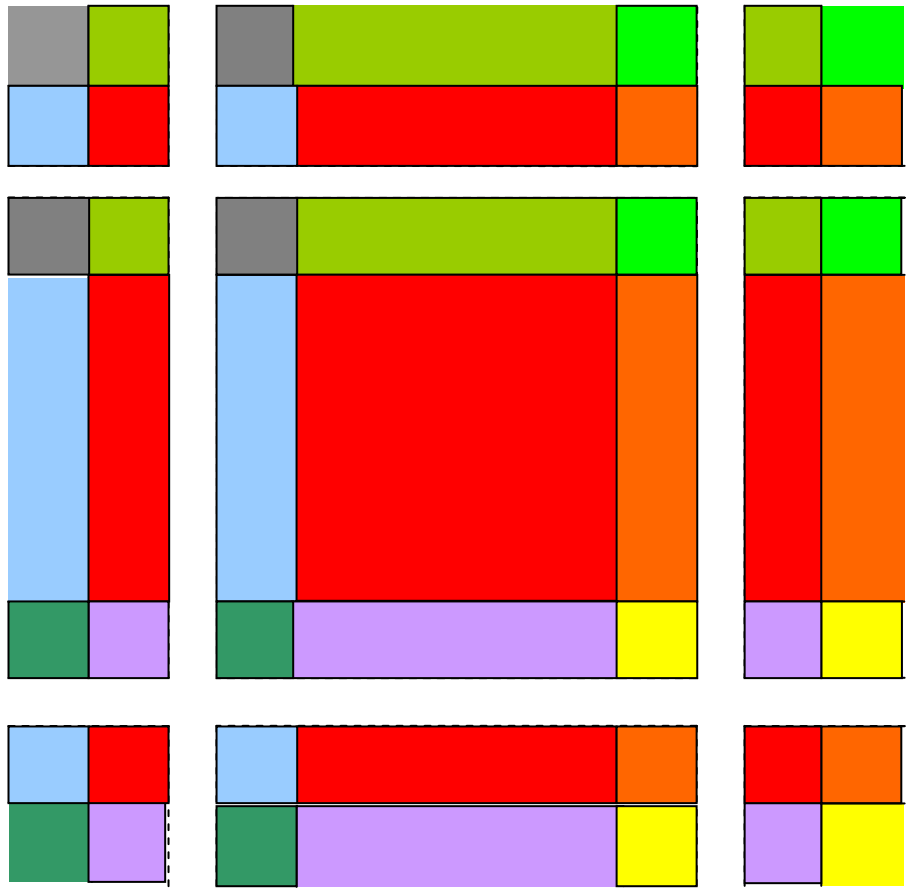
- Processes need to communicate their boundary data to neighboring processes.
- Sites within two sites from the boundary must be communicated.
- Each process can generate random numbers independently.

A Communication Strategy

- Do a left shift: send leftmost 2 columns left while receiving from the right.
- Do a right shift: send rightmost 2 columns right while receiving from the left.
- Similarly for up shifts and down shifts.
- After these 4 shifts have been done each process can update all its lattice sites.

Communication Shifts

1. Left shift.
2. Right shift.
3. Up shift.
4. Down shift.

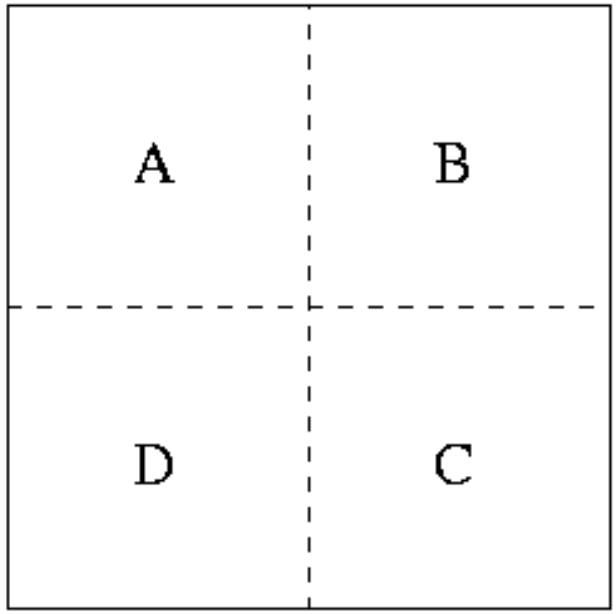


Update Conflicts

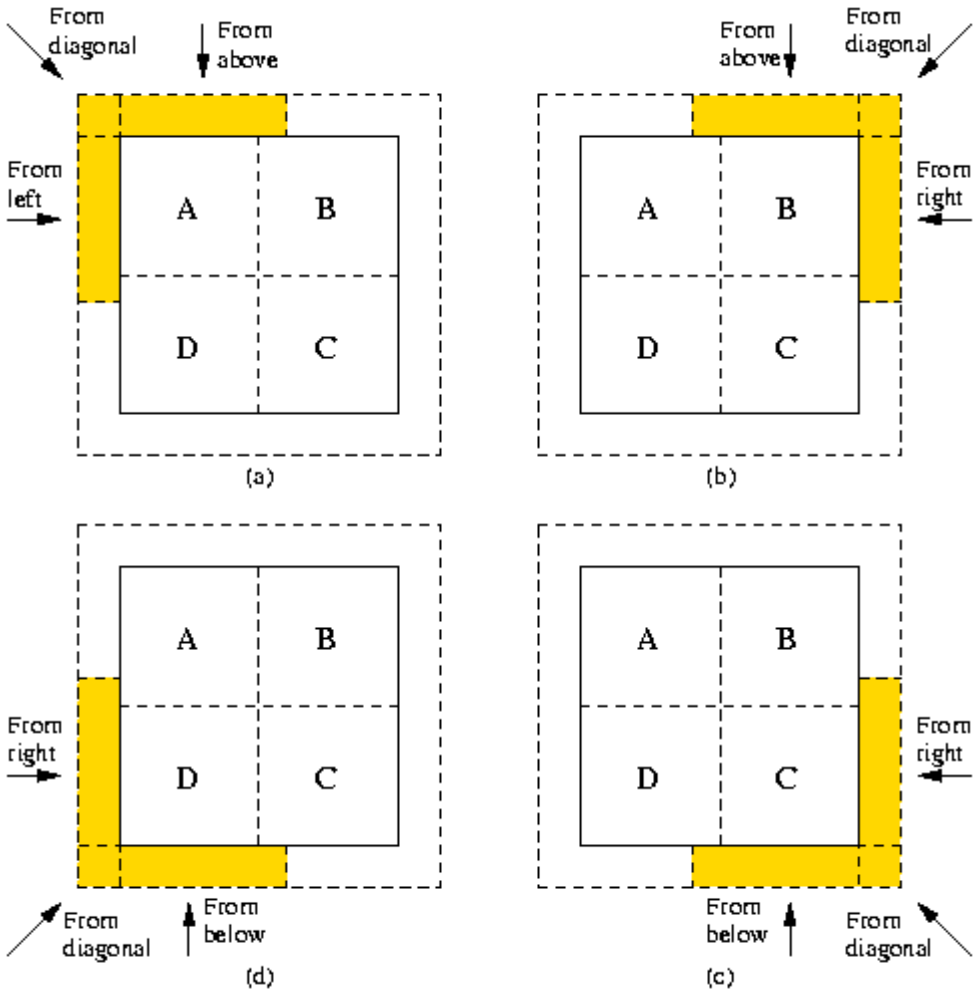
- Two adjacent processes can concurrently update the same lattice site close to their common boundary.
- This is an *update conflict*.
- Avoid conflicts by never updating adjacent areas in processes concurrently.
- Use sub-partitioning to do this.

Sub-partitioning

- First each process updates A, then B, C, and D.
- Before updating a sub-partition communication is needed to ensure each process has all the data to update its points.
- After updating a sub-partition the data is sent back to the process it came from.



Communication Before Update

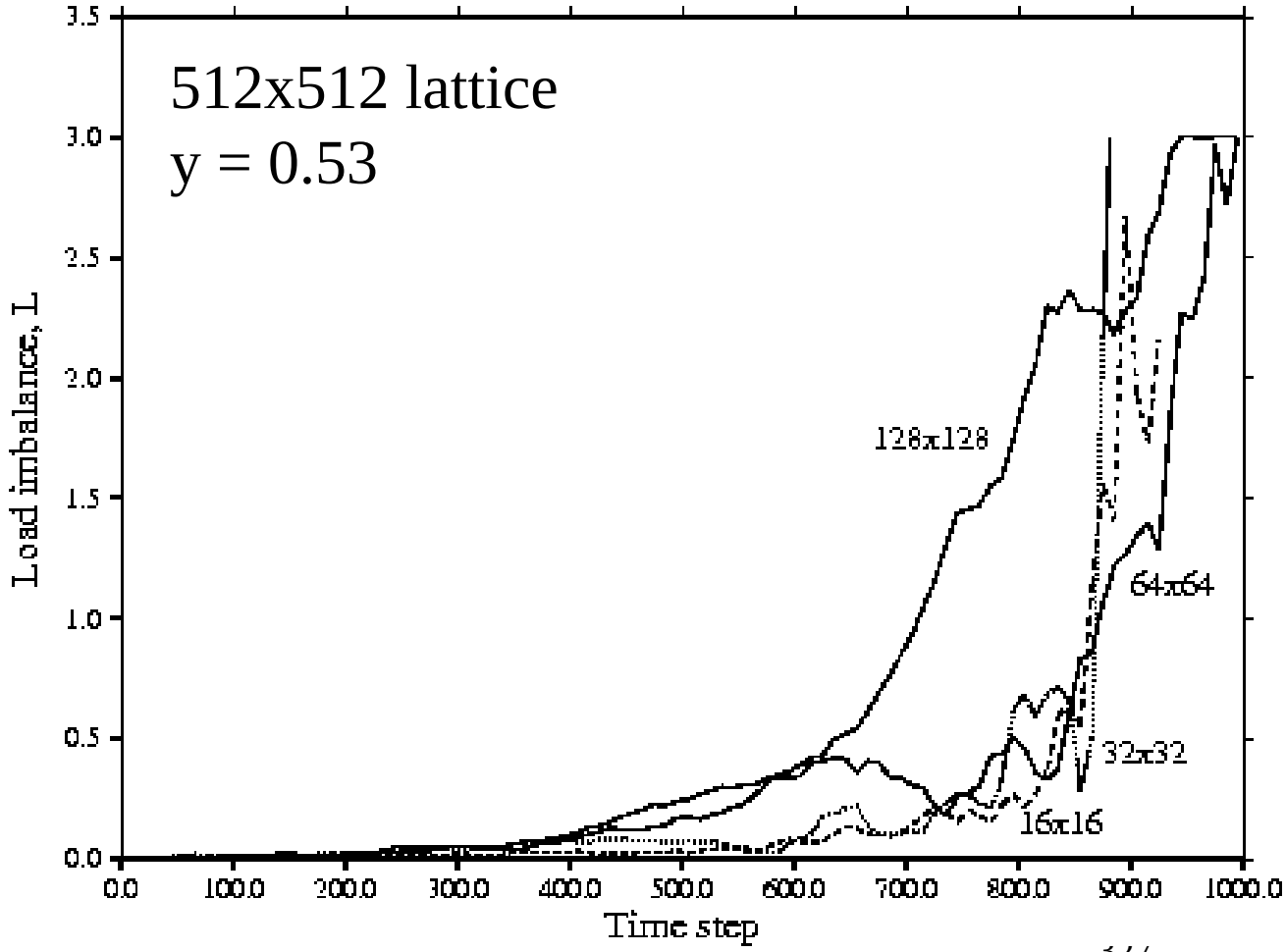


Load Imbalance

2×2 process mesh used.

Load imbalance is smaller for smaller block sizes.

Load imbalance is large as CO poisoning occurs.

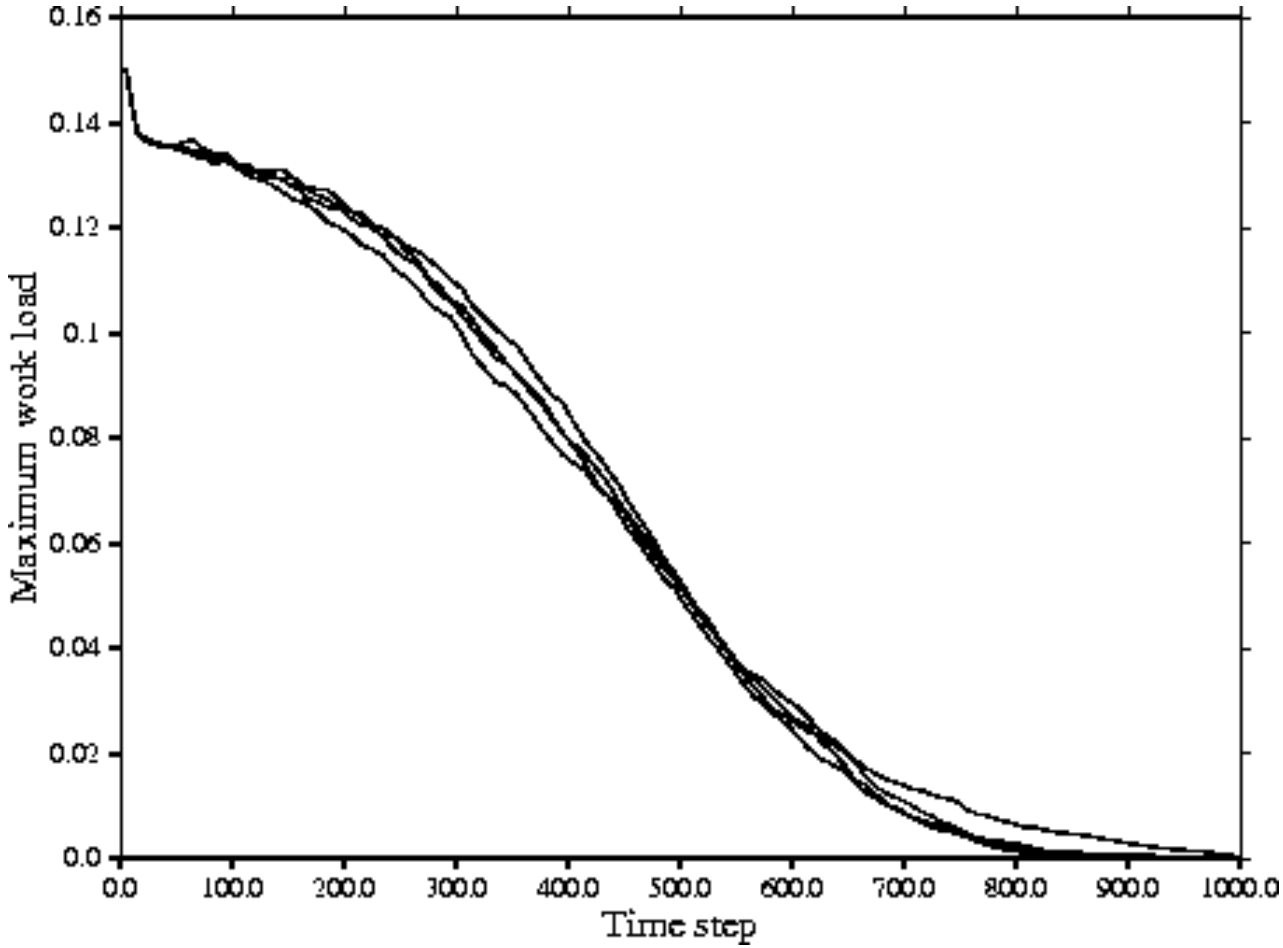


Maximum Work Load

Maximum work load is similar for different block size, except after step 700.

Not much work available at this time.

Load imbalance not very important!

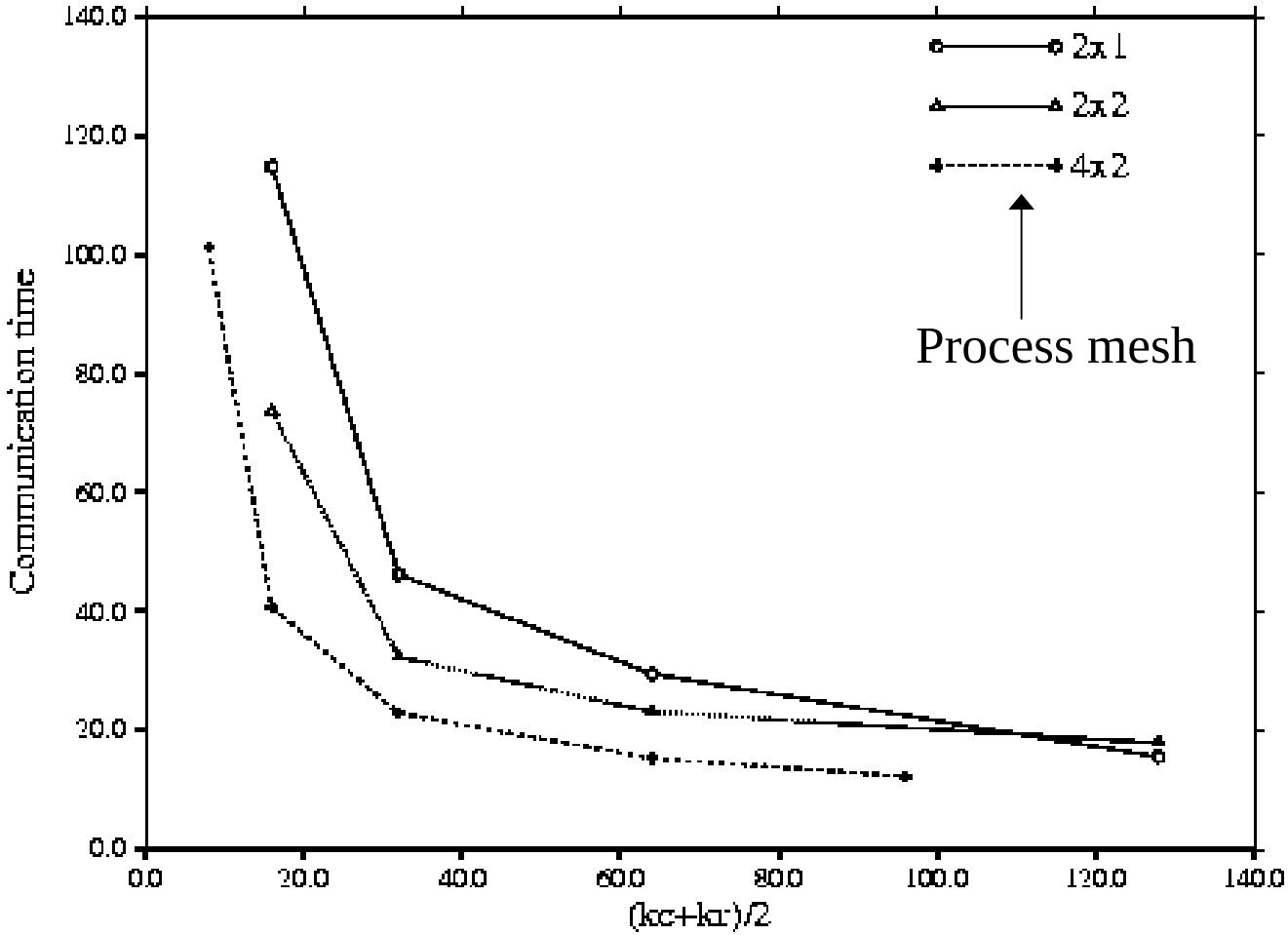


Communication Time

512x512 lattice,
y = 0.53

For given problem communication is smaller for more processes - less data per process.

Smaller blocks require more communication.



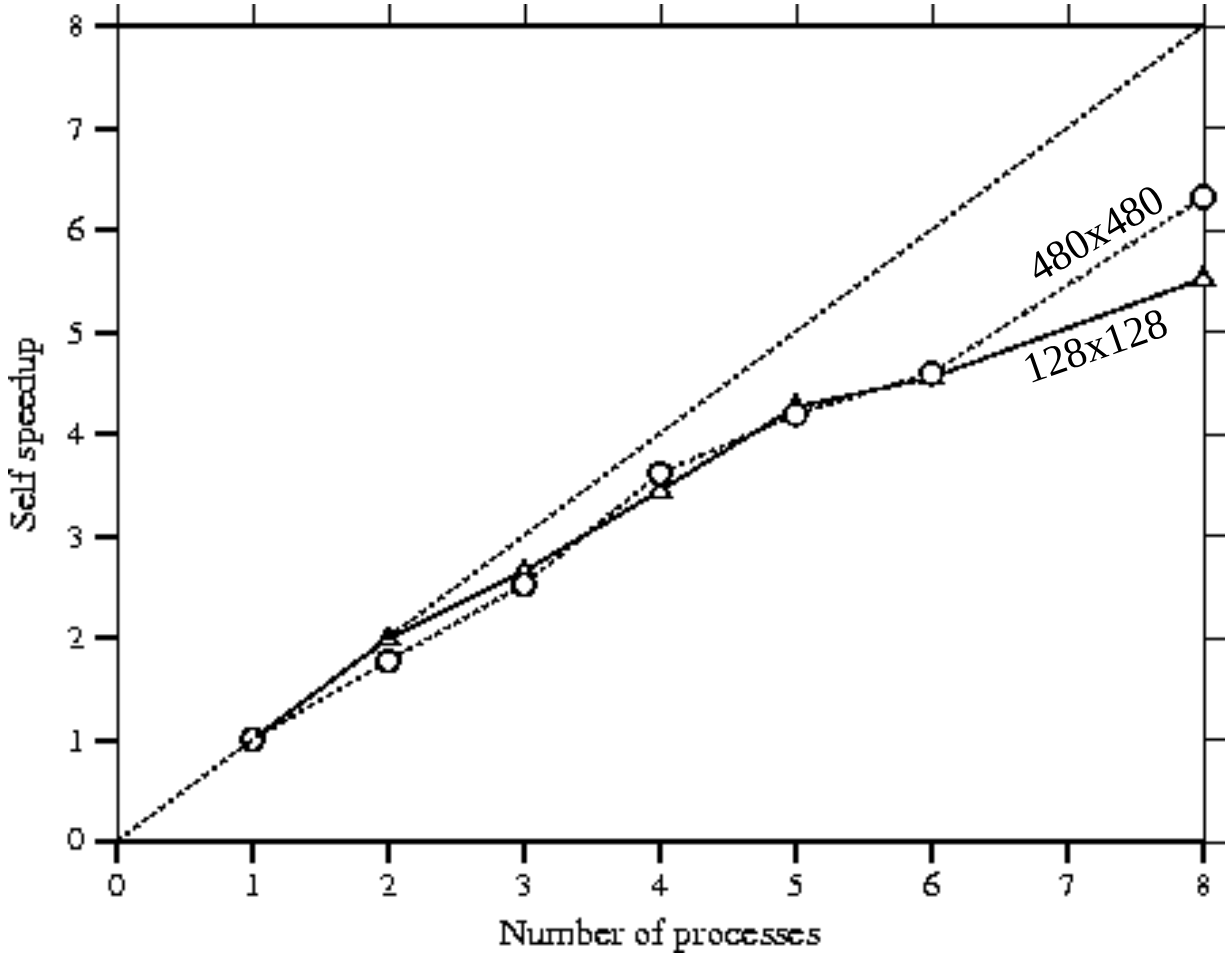
Performance Model

- Amount of communication and computation both depend linearly on problem size.
- Speed-up is independent of problem size and is given by:

$$S = \frac{N}{1 + A\,\tau\,(k_r + k_c)/(k_r\,k_c)}$$

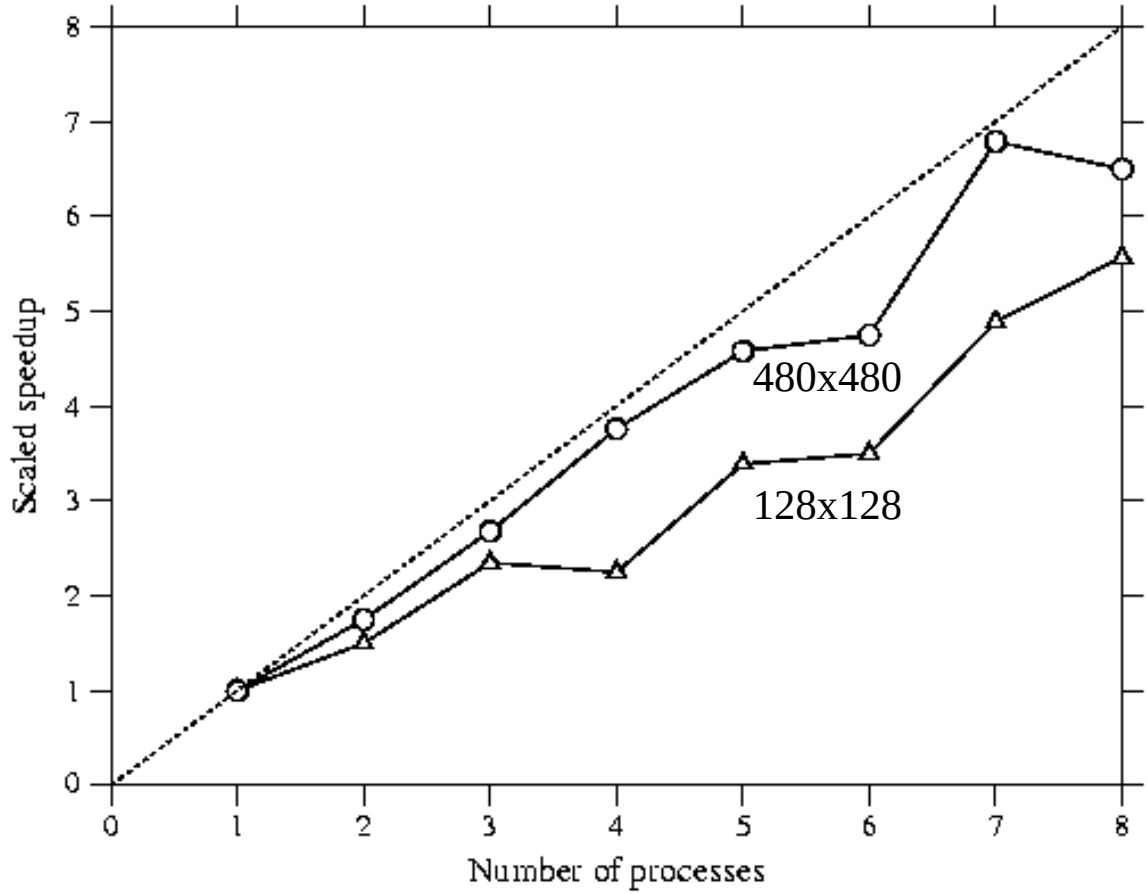
Self Speed-Up

As expected, speed-up is independent of problem size (except at 8!)



Scaled Speed-Up

Fixed problem size per process



Summary

- It turns out that load imbalance is not very important in this problem.
- Load imbalance will be important in cellular automata with more complex geometries.
- Easy to modify code for other CA problems.
- Speed-up independent of problem size.