

CASE4

Concurrent Programming

Dr. Martin Crane

Recommended Text:

“Foundations of Multithreaded, Parallel and Distributed Programming”, G.R. Andrews, Addison-Wesley, 2000. ISBN 0-201-35752-6

Additional Texts:

“Principles of Concurrent and Distributed Programming”, M. Ben-Ari, Prentice Hall, 1990.

“The SR Programming Language, Concurrency in Practice”, G.R. Andrews and R.A. Olsson, Benjamin/Cummings, 1993.

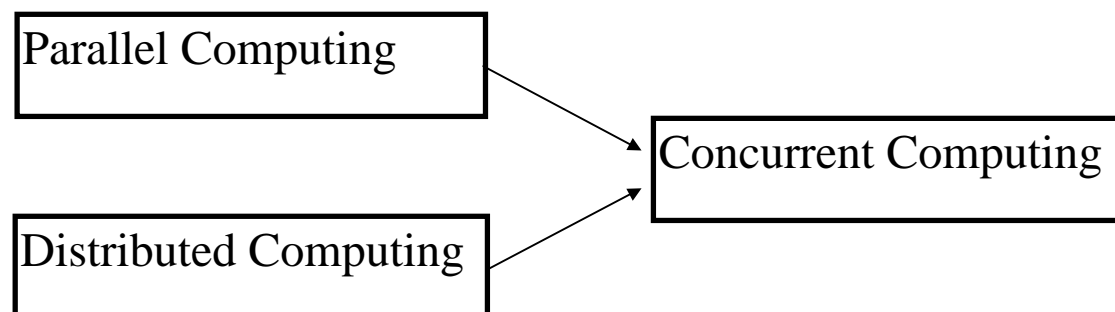
“Using MPI: Portable Parallel Programming with the Message Passing Interface”, W. Gropp, E. Lusk, A. Skjellum, 2nd Edition, MIT Press 1999.

Course Outline

- Introduction to Concurrent Processing
- Critical Sections and Mutual Exclusion
- Semaphores
- Monitors
- Message Passing (Java and MPI)
- Remote Procedure Calls (RPC)
- Rendezvous
- Languages for Concurrent Processing (SR, Java, Linda)
- Load Balancing and Resource Allocation
- Fault Tolerance

Parallel vs. Distributed Computing

The individual processors of a distributed system are physically distributed (loosely coupled), whereas parallel systems are “in the same box” (tightly coupled). Both are examples of multiprocessing, but distributed computing introduces extra issues.



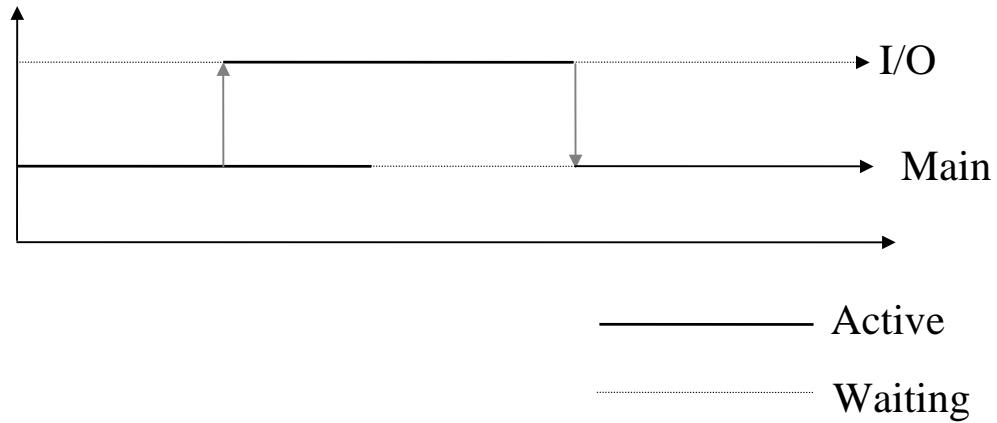
Why bother?

The advantages of concurrent processing are:

- 1) Faster processing
- 2) Better resource usage
- 3) Fault tolerance

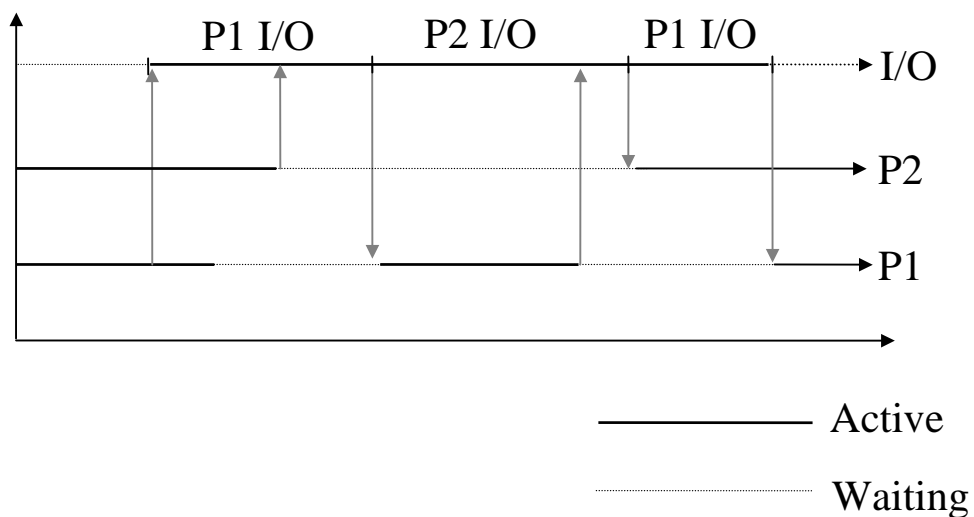
Types of Concurrent Processing

Overlapping I/O and Processing



Multi-Programming

One processor *time-slices* between several programs.



Multi-tasking

This is a generalisation of the multi-programming concept. Each program is decomposed into a set of concurrent tasks. The processor time-slices between all the tasks.

Multi-Processing

The set of concurrent tasks are processed by a set of interconnected processors. Each processor may have more than one task allocated to it, and may time-slice between its allocated tasks.

Applications of Concurrent Processing

Predictive Modelling and Simulation

Weather forecasting, Oceanography and astrophysics, Socioeconomics and governmental use, etc.

Engineering Design and Automation

Finite-element analysis, Computational Aerodynamics, Artificial Intelligence (Image processing, pattern recognition, speech understanding, expert systems, etc.), Remote Sensing, etc.

Energy Resource Exploration

Seismic Exploration, Reservoir Modelling, Plasma Fusion Power, Nuclear Reactor Safety, etc.

Medical

Computer-assisted Tomography.

Military

Weapons Research, Intelligence gathering surveillance, etc.

Research

Computational Chemistry and Physics, Genome Research, VLSI analysis and design, etc.

Parallel Speed-up

Minsky's Conjecture

Minsky argued that algorithms such as binary search, branch and bound, etc. would give the best speed-up. All processors would be used in the 1st iteration, half in the 2nd, and so, giving a speed-up of $\log_2 P$.

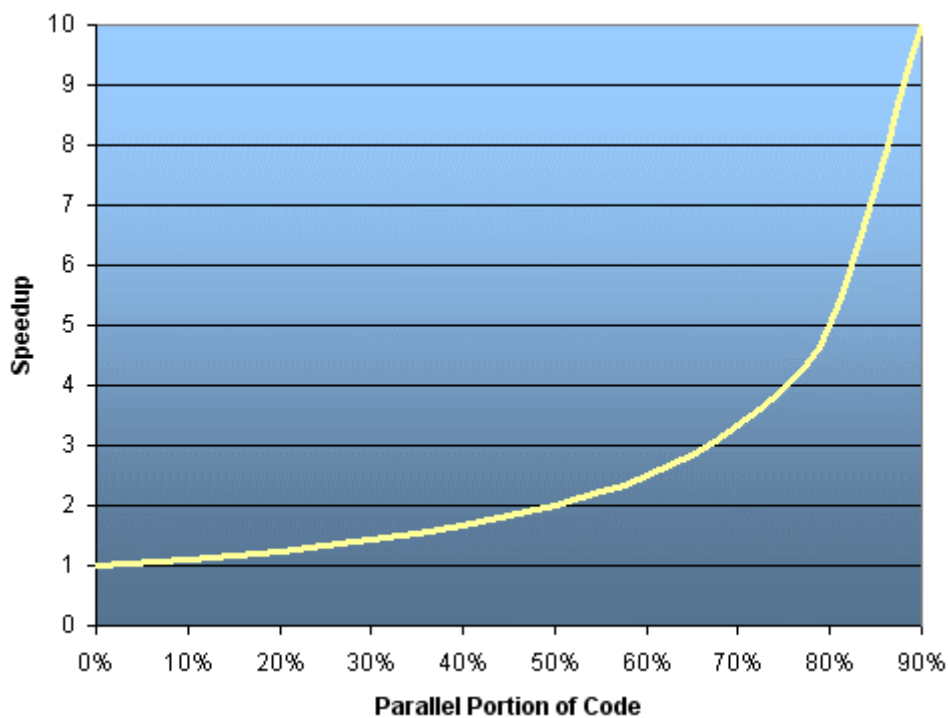
Amdahl's Law

Gene Amdahl divided a program into 2 sections, one that is inherently serial and the other which can be parallel. Let α be the fraction of the program which is inherently serial. Then,

$$\text{Speed-up, } S = \frac{T(\alpha + (1 - \alpha))}{T\left(\alpha + \frac{(1 - \alpha)}{P}\right)} = \frac{P}{1 + (P - 1)\alpha} \leq \frac{1}{\alpha}, \forall P > 1$$

If $\alpha = 5\%$, then $S \leq 20$.

Graph of S against $1 - \alpha$ for different α



Taking Amdahl's law again:

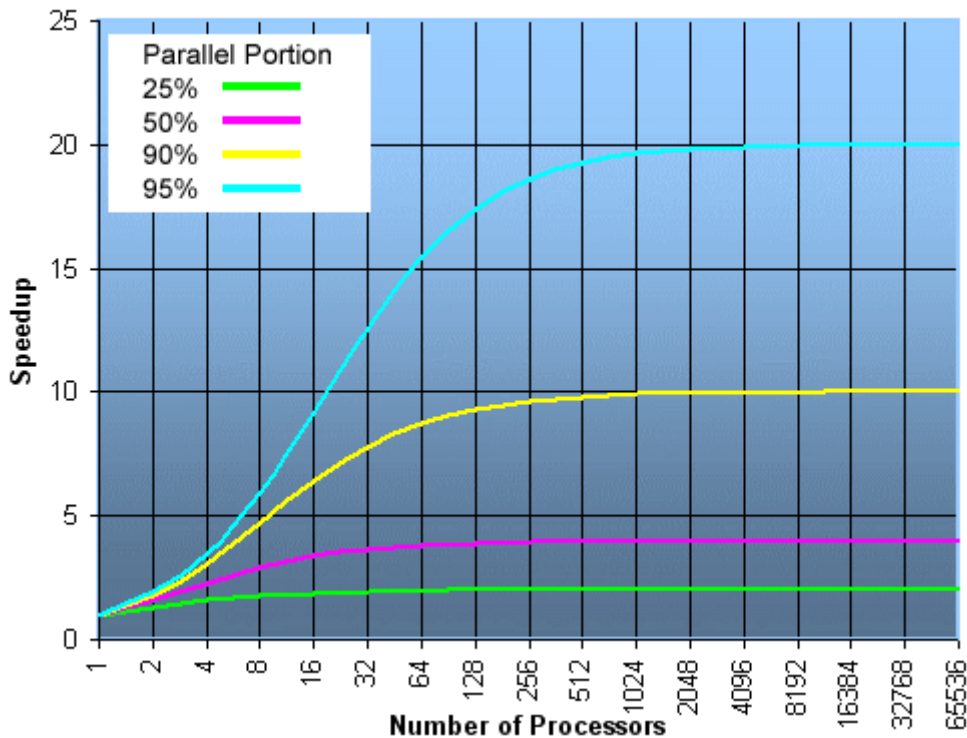
$$\text{Speed-up}, S = \frac{T(\alpha + (1-\alpha))}{T\left(\alpha + \frac{(1-\alpha)}{P}\right)} = \frac{1}{\alpha + \frac{(1-\alpha)}{P}}, \forall P > 1$$

for $\alpha=1$ (wholly serial problems) $S=1/\alpha=1$

for $\alpha=0$ (wholly parallel problems) $S=P$

for $\alpha=0.5$, $S=\frac{2}{1+\frac{1}{P}}$, as P becomes large S approaches 2

Graph of S against 1- α for different P



The Sandia Experiments

The Karp prize was established for the first program to achieve a speed-up of 200 or better. In 1988, a team from Sandia laboratories reported a speed-up of over 1,000 on a 1,024 processor system on three different problems.

How?

Moler's Law

An implicit assumption of Amdahl's Law is that the fraction of the program which is inherently serial, α , is independent of the size of the program. The Sandia experiments showed this to be false. As the problem size increased the inherently serial parts of a program remained the same or increased at a slower rate than the problem size. So Amdahl's law should be

$$S \leq \frac{1}{\alpha(n)}$$

so as the problem size, n , increases, $\alpha(n)$ decreases and S increases.

For example, certain problems demonstrate increased performance by increasing the problem size. For example:

2D Grid Calculations:	85 seconds	85%
Serial fraction:	15 seconds	15%

We can increase the problem size by doubling the grid dimensions and halving the time step. This results in four times the number of grid points and twice the number of time steps. The timings then look like:

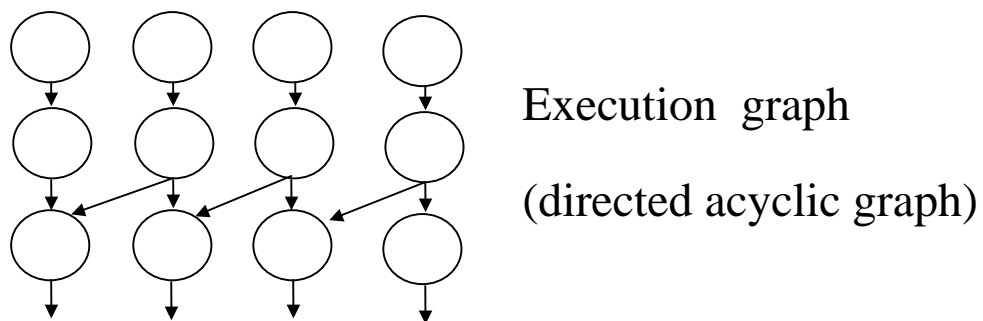
2D Grid Calculations	680 seconds	97.84%
Serial fraction	15 seconds	2.16%

Problems that increase the percentage of parallel time with their size are more scalable than problems with a fixed percentage of parallel time

Moler defined an efficient parallel algorithm as one where $\alpha(n) \rightarrow 0$ as $n \rightarrow \infty$ (Moler's law).

Sullivan's First Theorem

The speed-up of a program is $\min(P, C)$, where P is the number of processors and C is the concurrency of the program.



If N is the number of operations in the execution graph, and D is the longest path through the graph then the concurrency $C = N/D$.

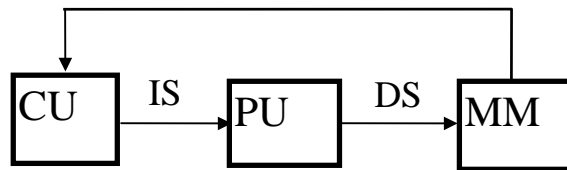
The maximum speed-up is a property of the structure of the parallel program.

Architectural Classification Schemes

Flynn's Classification

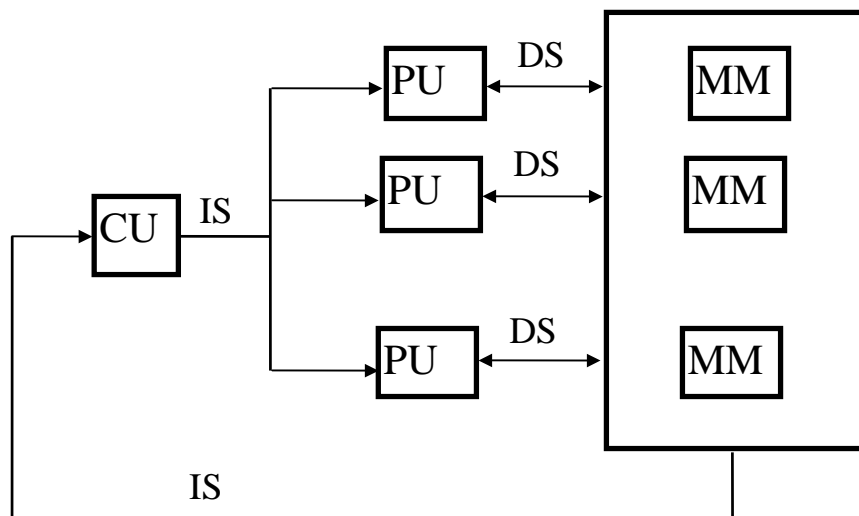
Bases on multiplicity of *instruction streams* and *data streams*.

SISD Single Instruction Single Data



(old but still common)

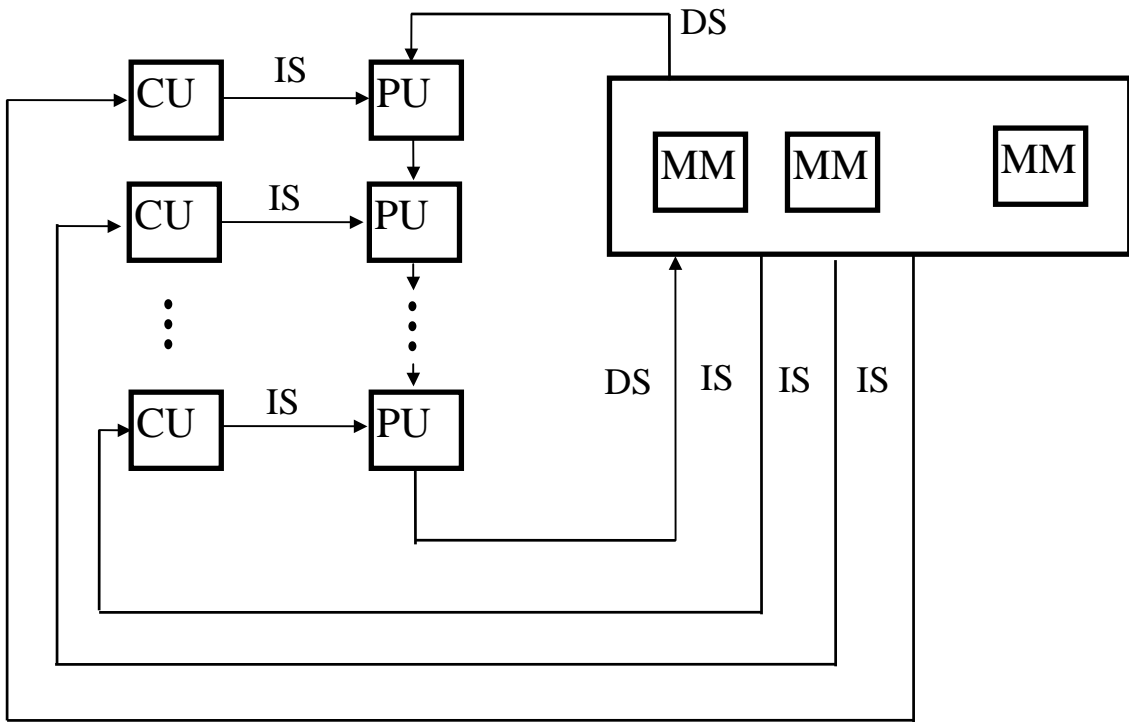
SIMD Single Instruction Multiple Data Array processors



(most modern especially those with Graphics)

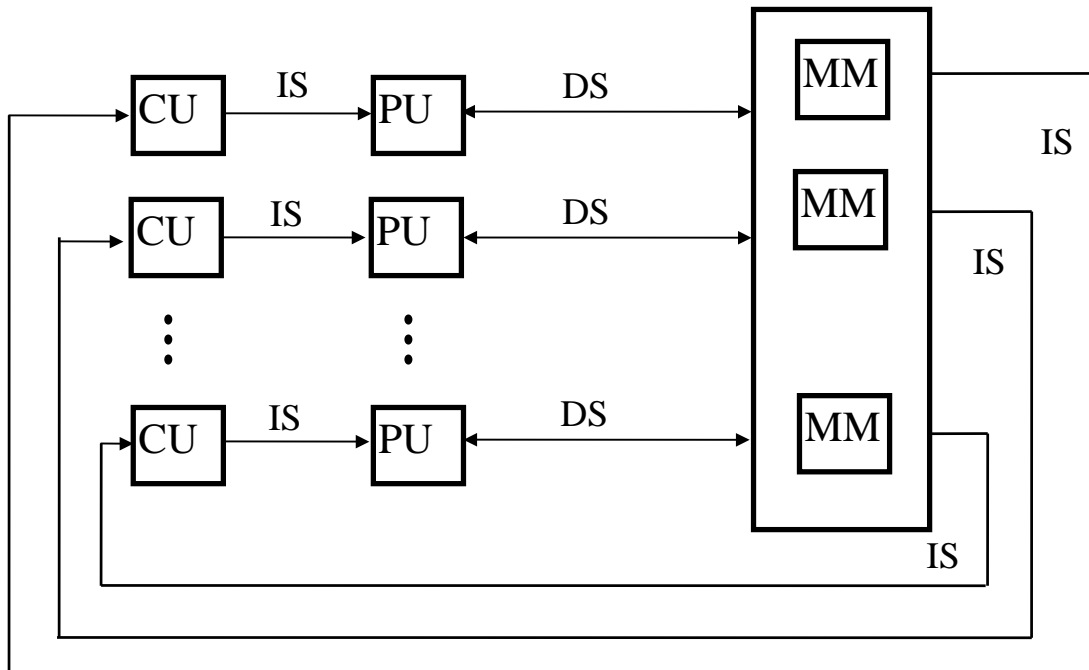
MISD Multiple Instruction Single Data

Some say this is impractical - no MISD machine exists. Maybe multiple Crypto Algorithms on a single message?



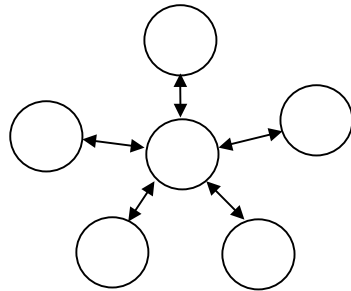
MIMD Multiple Instruction Multiple Data

Most modern supercomputers are MIMD with SIMD subcomponents for specialised tasks.



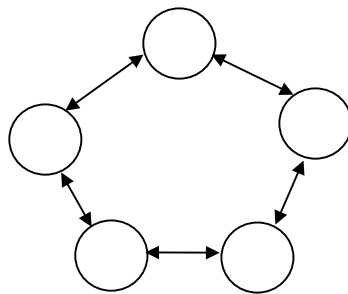
Processor Topologies

Farm (Star)



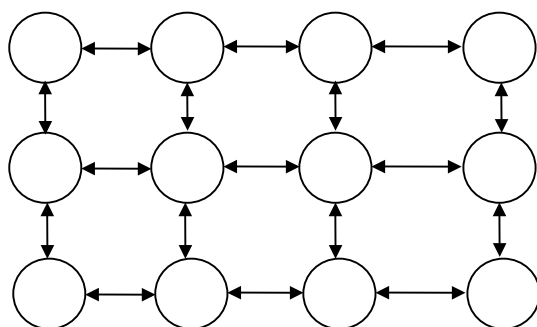
Ring

Used in image processing applications, or anything which uses a nearest-neighbour operator.



Mesh

Image processing, finite-element analysis, nearest-neighbour operations.



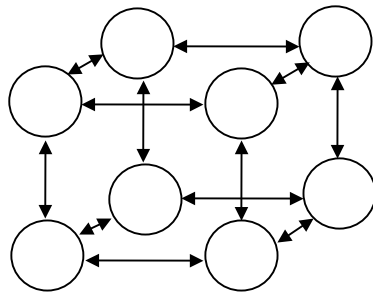
Torus

A variant of the mesh.

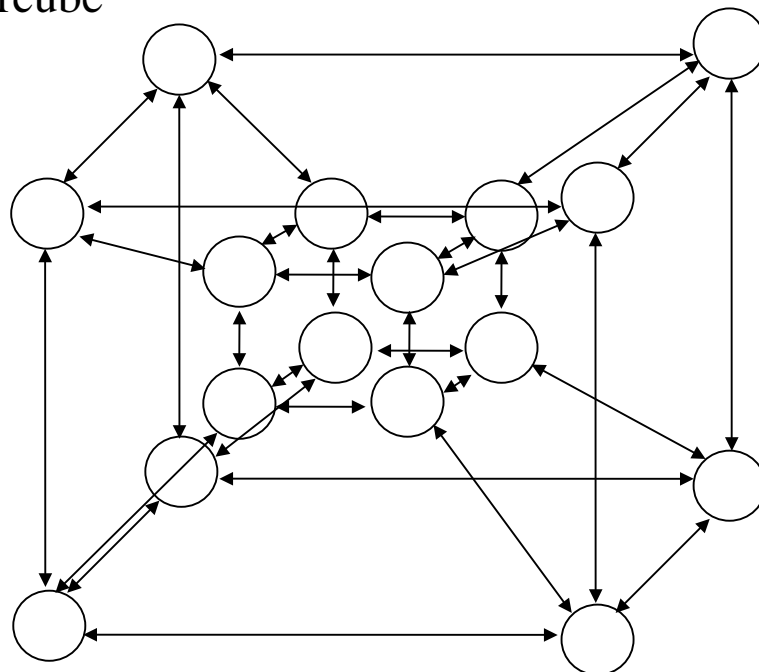
Hypercube

A structure with the minimum diameter property.

3-d hypercube



4-d hypercube



A Model of Concurrent Programming

A *concurrent program* is the *interleaving* of sets of sequential *atomic* instructions.

A concurrent program can be considered as a set of interacting sequential processes. These sequential processes execute at the same time, on the same or different processors. The processes are said to be *interleaved*, that is at any given time each processor is executing one of the instructions of the sequential processes. The relative rate at which the instructions of each process are executed is not important.

Each sequential process consists of a series of *atomic instructions*. An *atomic instruction* is an instruction that once it starts, proceeds to completion without interruption. Different processors have different atomic instructions, and this can have a big effect.

N : Integer := 0;

Task body P1 is
begin
 N := N + 1;
end P1;

Task body P2 is
begin
 N := N + 1;
end P2;

If the processor includes instructions like INC then this program will be correct no matter which instruction

is executed first. But if all arithmetic must be performed in registers then following interleaving does not produce the desired results.

P1: load reg, N
P2: load reg, N
P1: add reg, #1
P2: add reg, #1
P1: store reg, N
P2: store reg, N

A concurrent program must be correct under **all** possible interleavings.

Correctness

If $P(\vec{a})$ is a property of the input (pre-condition), and $Q(\vec{a}, \vec{b})$ is a property of the input and output (post-condition), then correctness is defined as:

Partial correctness

$$\left(P(\vec{a}) \wedge \text{terminates}(\text{Prog}(\vec{a}, \vec{b})) \right) \Rightarrow Q(\vec{a}, \vec{b})$$

Total correctness

$$P(\vec{a}) \Rightarrow \left(\text{terminates}(\text{Prog}(\vec{a}, \vec{b})) \wedge Q(\vec{a}, \vec{b}) \right)$$

Totally correct programs terminate. A totally correct specification of the incrementing tasks is:

$$a \in N \Rightarrow (\text{terminates}(\text{INC}(a, a) \wedge a = a + 1))$$

There are 2 types of correctness properties:

Safety properties	These must <i>always</i> be true.
<u>Mutual exclusion</u>	Two processes must not interleave certain sequences of instructions.
<u>Absence of deadlock</u>	Deadlock is when a non-terminating system cannot respond to any signal.
Liveness properties	These must <i>eventually</i> be true.
<u>Absence of starvation</u>	Information sent is delivered.
<u>Fairness</u>	That any contention must be resolved.

There 4 different way to specify *fairness*.

Weak Fairness	If a process continuously makes a request, eventually it will be granted.
Strong fairness	If a process makes a request infinitely often, eventually it will be granted.
Linear waiting	If a process makes a request, it will be granted before any other process

is granted the request more than once.

FIFO

If a process makes a request, it will be granted before any other process makes a later request.

Mutual Exclusion

A concurrent program must be correct in all allowable interleavings. Therefore there must be some sections of the different processes which cannot be allowed to be interleaved. These are called *critical sections*.

```
do true ->  
  Non_Critical_Section  
  Pre_protocol  
  Critical_Section  
  Post_protocol  
od
```

First proposed solution

```
var Turn: int := 1;

process P1
  do true ->
    Non_Critical_Section
    do Turn != 1 ->
      od
    Critical_Section
    Turn := 2
  od
end

process P2
  do true ->
    Non_Critical_Section
    do Turn != 2 ->
      od
    Critical_Section
    Turn := 1
  od
end
```

This solution satisfies mutual exclusion.

This solution cannot deadlock, since both processes would have to loop on the test on Turn infinitely and fail. This would imply $\text{Turn} = 1$ and $\text{Turn} = 2$ at the same time.

There is no starvation. This would require one task to execute its critical section infinitely often and the other task to be stuck in its pre-protocol.

However this solution can fail in the absence of contention. If one process halts in its critical section the other process will always fail in its pre-protocol. Even if the processes are guaranteed not to halt, both processes are forced to execute at the same rate. This, in general, is not acceptable.

Second proposed solution

The first attempt failed because both processes shared the same variable.

```
var C1:int := 1
var C2:int := 1

process P1
  do true ->
    Non_Critical_Section
    do C2 != 1 ->
      od
    C1 := 0
    Critical_Section
    C1 := 1
  od
end
```

```
process P2
  do true ->
    Non_Critical_Section
    do C1 != 1 ->
      od
      C2 := 0
      Critical_Section
      C2 := 1
    od
  end
```

This unfortunately violates the mutual exclusion requirement. To prove this we need to find only one interleaving which allows P1 and P2 into their critical section simultaneously. Starting from the initial state, we have:

P1 checks C2 and finds $C2 = 1$.

P2 checks C1 and finds $C1 = 1$.

P1 sets $C1 = 0$.

P2 sets $C2 = 0$.

P1 enters its critical section.

P2 enters its critical section.

QED

Third proposed solution

The problem with the last attempt is that once the pre-protocol loop is completed you cannot stop a process from entering its critical section. So the pre-protocol loop should be considered as part of the critical section.

```
var C1:int := 1
```

```
var C2:int := 1
```

```
process P1
```

```
  do true ->
```

```
    Non_Critical_Section    # a1
```

```
    C1 := 0                  # b1
```

```
    do C2 != 1 ->          # c1
```

```
    od
```

```
    Critical_Section       # d1
```

```
    C1 := 1                  # e1
```

```
  od
```

```
end
```

```
process P2
```

```
  do true ->
```

```
    Non_Critical_Section    # a2
```

```
    C2 := 0                  # b2
```

```
    do C1 != 1 ->          # c2
```

```
    od
```

```
    Critical_Section       # d2
```

```
    C2 := 1                  # e2
```

```
  od
```

```
end
```

We can prove that the mutual exclusion property is valid. To do this we need to prove that the following equations are *invariants*.

$$C1 = 0 \equiv at(c1) \vee at(d1) \vee at(e1) \quad - \text{eqn 1}$$

$$C2 = 0 \equiv at(c2) \vee at(d2) \vee at(e2) \quad - \text{eqn 2}$$

$$\neg(at(d1) \wedge at(d2)) \quad - \text{eqn 3}$$

$at(x) \Rightarrow$ x is the next instruction to be executed in that process.

Eqn 1 is initially true. Only the $b1 \rightarrow c1$ and $e1 \rightarrow a1$ transitions can affect its truth. But each of these transitions also changes the value of C1.

A similar proof is true for eqn 2.

Eqn 3 is initially true, and can only be made false by a $c2 \rightarrow d2$ transition while $at(d1)$ is true. But by eqn 1, $at(d1) \Rightarrow C1=0$, so $c2 \rightarrow d2$ cannot occur since this requires $C1=1$. Similar proof for process P2.

Fourth proposed solution

The problem with the last proposed solution was that once a process indicated its intention to enter its critical section, it also **insisted** on entering its critical section. What we need is some way for a process to relinquish its attempt if it fails to gain immediate access to its critical section, and try again.

```

var C1:int := 1
var C2:int := 1

process P1
  do true ->
    Non_Critical_Section
    C1 := 0
    do true ->
      if C2 = 1 -> exit fi
      C1 :=1
      C1 := 0
    od
    Critical_Section
    C1 := 1
  od
end

```

```

process P2
  do true ->
    Non_Critical_Section
    C2 := 0
    do true ->
      if C1 = 1 -> exit fi
      C2 :=1
      C2 := 0
    od
    Critical_Section
    C2 := 1
  od
end

```

This proposal has two drawbacks.

- 1) *A process can be starved.* You can find an interleaving in which a process never gets to enter its critical section.
- 2) *The program can livelock.* This is a form of deadlock. In *deadlock* there is no possible interleaving which allows the processes to enter their critical sections. In *livelock*, some interleaving succeed, but there sequences which do not succeed.

Dekker's Algorithm

This is a combinations of the first and fourth proposals. The first proposal explicitly passed the right to enter the critical sections between the processes, whereas the fourth proposal had its own variable to prevent problems in the absence of contention. In Dekker's algorithm the right to insist on entering a critical section is explicitly passed between processes.

```
var C1:int := 1
var C2:int := 1
var Turn:int := 1
```

```

process P1
  do true ->
    Non_Critical_Section
    C1 := 0
    do true ->
      if C2 = 1 -> exit fi
      if Turn = 2 ->
        C1 := 1
        do Turn != 1 -> od
        C1 := 0
      fi
    od
    Critical_Section
    C1 := 1
    Turn := 2
  od
end

```

```

process P2
  do true ->
    Non_Critical_Section
    C2 := 0
    do true ->
      if C1 = 1 -> exit fi
      if Turn = 1 ->
        C2 := 1
        do Turn != 2 -> od
        C2 := 0
      fi
    od
    Critical_Section
    C2 := 1
    Turn := 1
  od
end

```

```
od
end
```

This is a solution for mutual exclusion for 2 processes.

Mutual Exclusion for N Processes

There are many N process mutual exclusion algorithms; all complicated and relatively slow to other methods. One such algorithm is the *Bakery Algorithm*. The idea is that each process takes a numbered ticket (whose value constantly increases) when it want to enter its critical section. The process with the lowest current ticket gets to enter its critical section.

```
var Choosing: [N] int
var Number: [N] int
```

```
# Choosing and Number arrays initialised to zero
```

```
process P(i:int)
do true ->
  Non_Critical_Section
  Choosing [i] := 1
  Number [i] := 1 + max (Number)
  Choosing := 0
  fa j := 1 to N ->
    if j != i ->
```

```

do Choosing [ j] != 0 -> od
do true ->
  if (Number [ j] = 0) or
    (Number [i] < Number [ j]) or
    ((Number [i] = Number [ j]) and
     (i < j)) ->
    exit
  fi
od
fi
af
Critical_Section
Number [i] := 0
od
end

```

The bakery algorithm is not practical because:

- a) the ticket numbers will be unbounded if some process is always in its critical section, and
- b) even in the absence of contention it is very inefficient as each process must query the other processes for their ticket number.

Hardware-Assisted Mutual Exclusion

If the atomic instructions available allow a load and store in a single atomic instruction all out problems disappear. For example if there is an atomic *test and set* instruction equivalent to `li := C; C := 1` in one instruction, then we could have mutual exclusion as follows.

```
var C:int := 0
```

```
process P
  var li:int
  do true ->
    Non_Critical_Section
    do li != 0 -> test_and_set (li) od
    Critical_Section
    C := 0
  od
end
```

A similar solution exists with atomic *exchange* instructions.