# Introduction

## Objectives

To understand the problems involved in writing distributed code.

Multi-processor; multi-box; …. Multi-site.

To become familiar with some of the techniques and concepts which allow you to write distributed code.

Problems and an introduction to their solutions.

---

Single processor programming is a **solved** problem.

If you write to solve the problem, the processor; operating system; compiler (interpreter) will produce an efficient solution without you knowing what is happening *under the hood.*

Distributed programming is more complex **unsolved**

Techniques available, but no black box solution: description of the problem to a system leads to a solution

## Problems

### Transferring of information

*Embarrassingly parallel*

Problems where information is transferred to the sub task at the start and no further interaction is required until the end.
Information supplied at task creation

At the end some sort of collecting of the information usually occurs, but that collecting has no influence on any other sub tasks.
Information is collated by one task, with which the sub task needs to communicate.

Trade off.
Sequential operation is clear and easy, but removes the advantages of parallel execution.

The creation of the task and collection of the data will constitute a small part of the problem and can be carried out quasi-sequentially without a great loss of efficiency.
(Gustaffson rather than Amdahl scaling)

## Embarrassingly parallel

If the data can be be broken up into disjoint segments and the same code can be run on each segment. Running that code is the major part of the problem.

Particle physics … each collision is a separate problem and there are millions of collisions.

Creating a film in CGI, each frame is a separate problem.

Many search problems, including database searches.

For these sort of problems there is an existing partial solution and it is the **MAP-Reduce** paradigm.

This is supported by Hadoop, where the data can be spread over many sites and it appears to the user as one data set.

“Query” includes ‘what is this particle physics event?’

The Map phase is then running the “query” and the processing is done by a processor which is local to the location of the data.
Reduce phase is collecting the data together.

## MAP-Reduce-Hadoop

Specific implementation of the paradigm

Useful and many tutorials exist on the web

Will not cover

Concentrate on general techniques – which provide a broader set of skills to use distributed computing.

## Distributed v Multi-core

At one time distributed would have meant "multi-boxes"

Will use **multi-processor** to indicate both

Now much is concerned with multi-core on the same chip

Intel and AMD make multi-core chips almost exclusively

The two are related
multi-core is the technique which has more general application
multi-processor can be applied to bigger problems

Will cover topics from both, which I consider are most educational

## Road-Map

The topics will be …

1. Computer Arithmetic and Instructions
2. Execution of instructions – the data path
3. Pipelining – achieving single core performance
4. Caches – tools to achieve single core performance
5. Caches – problems they produce for multi-core computation
6. Networks – joining computers. Performance
7. Problems  with multi-processor computing
8. Tools for multi-processor programming

Not multi-issue processors.

An interesting and important topic, but the problems of multi-processor computing can be discussed with them.

You can’t understand the i7 without looking at it’s evolution, any more than you can understand a whale without understanding its evolution

Test Pit

Do you need to know all the levels from the properties of silicon to the configuration of the Motherboard.

Probably – but not enough time.

At times I will dig down to the basic device level, in order to understand the design decisions at the upper level.

You want to understand the architecture of computers – but at times architectural decisions are driven by deeper level constraints.

Catalhoyuk

Start with ISA – distributed architecture relies on understanding single processor.

A large amount of the modern ISA is about distributed processing – and about communicating between different parts of the processors.

So when we cover cache coherence
a problem with multi-processor cpus
The techniques are applicable to running jobs on distributed systems.

When we cover network topologies we can be talking about on chip caches or machines on different continents. The problems are the same, but the balance is different and so the best solution may be different.

This is real engineering, there is no optimum solution only a balance.

Concentrate on generalisable

Intel and AMD make different choices and at times each has made a bad decision.

So much of this module is applicable in many places.

## Performance

Everyone wants faster computers …

Users want their Desktop/Laptop machines to respond more rapidly.

Engineers/Meteorologists want their models to be more accurate and return better results.

Resource Providers want to put more work through their systems (and make more money)

Wait for Intel

## How do we improve performance.

Simple answer : perform operations faster.
Faster clocking of processors and their components.

This is actually much older than the Intel/AMD retreat from clock speed

Improvements

Improve the "efficiency" of computers at given clock speed.

Increase the speed of a computer by increasing the speed at which instructions execute

Increase the speed at which instructions complete.

## Performance improvements

We will look at techniques to increase performance.

1. Take Advantage of Parallelism
2. Principle of Locality – spatial and temporal
3. Caching

How we can measure these improvements?

What does it mean to say a computer has a better performance?

How to quantify the improvements.

Distinguish between **latency** and **bandwidth**

For a production line **Latency** is the time for the first car to come off the line**.**
**Bandwidth** is the number per hour in steady state

Multiple Sites (Grid/Cloud), Multiple machines, multiple cores, multiple disks, Multiple components in a single core.

Only the last of these covered this weekend

Such as
Carry look-a-head adders sums from linear to log

Multiple memory banks searched in parallel in set-associative caches

**11** Common Cases

## Make your effort count

**Optimise** the frequent situation not the infrequent one

Instruction fetch and decode unit used more frequently than multiplier, so where if there are conflicting requirements satisfy. Fetch and decode.

*The infrequent case is often harder to deal with and people put in much more effort to optimise*

True for your code

Database server has 50 disks / processor
Storage dependability dominates system dependability.

overflow is rare when adding two numbers optimise no overflow performance *at the expense of overflow*

**Amdahl's Law** (not just parallel computing)
Best you could ever hope to do:

What is the maximum speed up for 1% non-par?

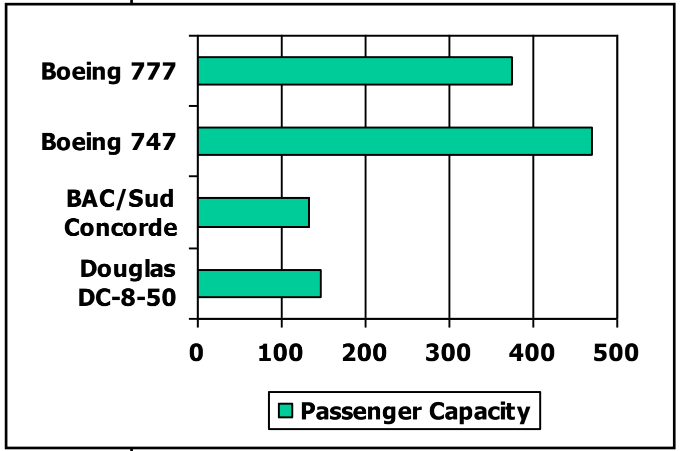
$$\text{ExTime}_{\text{new}} = \text{ExTime}_{\text{old}} \times \left[ \left(1 - \text{Fraction}_{\text{enhanced}}\right) + \frac{\text{Fraction}_{\text{enhanced}}}{\text{Speedup}_{\text{enhanced}}} \right]$$

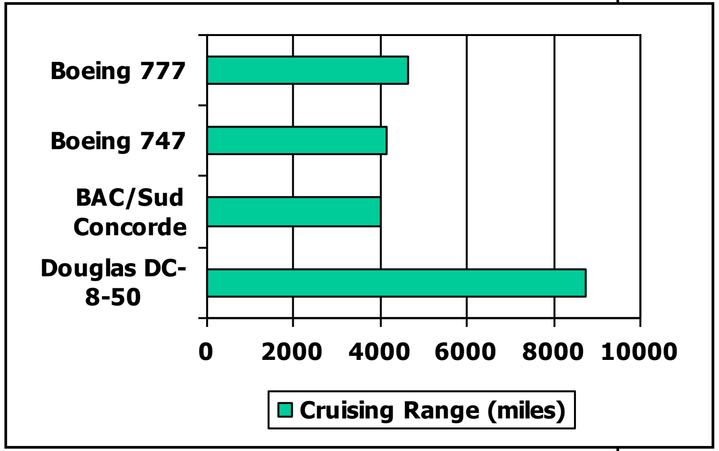
$$\text{Speedup}_{\text{maximum}} = \frac{1}{\left(1 - \text{Fraction}_{\text{enhanced}}\right)}$$

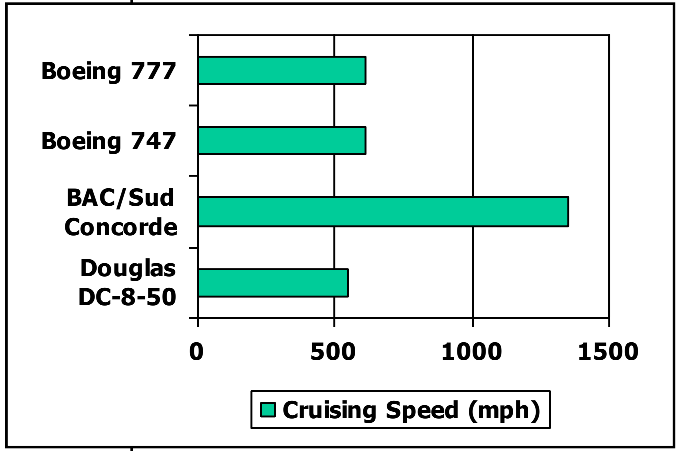
## Your answer depends on what you measure

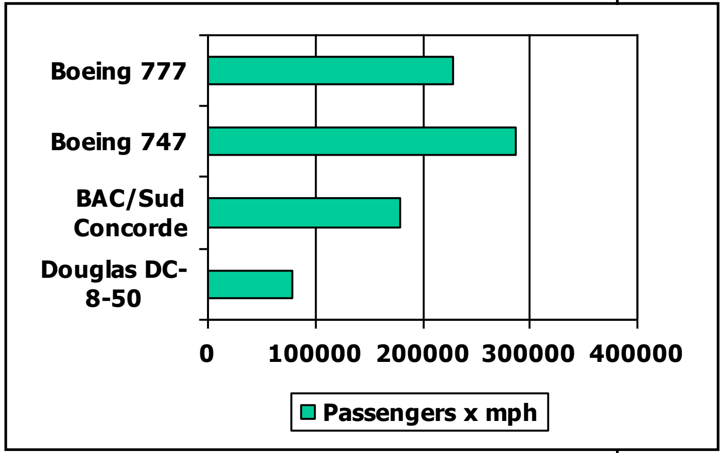
Which aircraft has the best performance?

What do you mean by best?









Fastest?

Profit per passenger mile?

How fast is your company internet? What is the best way to get 20 TB of data to Berlin?

**13** Components

**Performance Components.**

**Performance Equation**

$$\text{CPU time} = \frac{\text{Seconds}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Cycle}}$$

Program and complier give number of instruction

Instruction set architecture gives number of instructions
ISA also gives Cycles per instruction.

Cycles per instruction **CPI**

Technology determines seconds per cycle.

**Clock**

Free running devices are possible.

Numerous components of a computer depend on a number of different inputs all being together, in order to give the correct output.

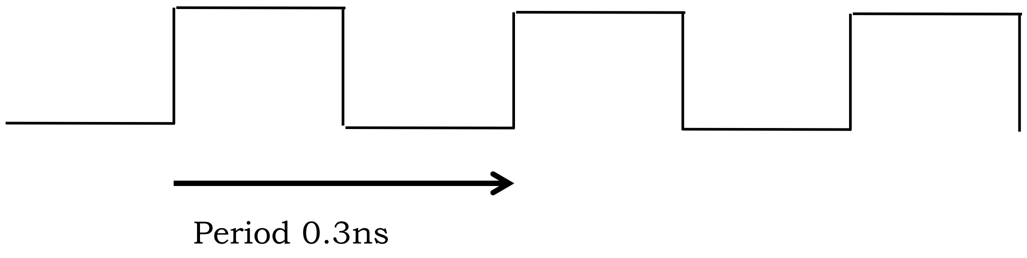
Further the output may not reach a stable state until some time after it is established.

The easiest way to ensure all inputs are present and that all outputs are stable is to synchronise operations on a clock.

A repetitive square wave of constant frequency and where transitions occur at specific times

3 GHz

Light travels 100mm
Electrical signals~ 10mm

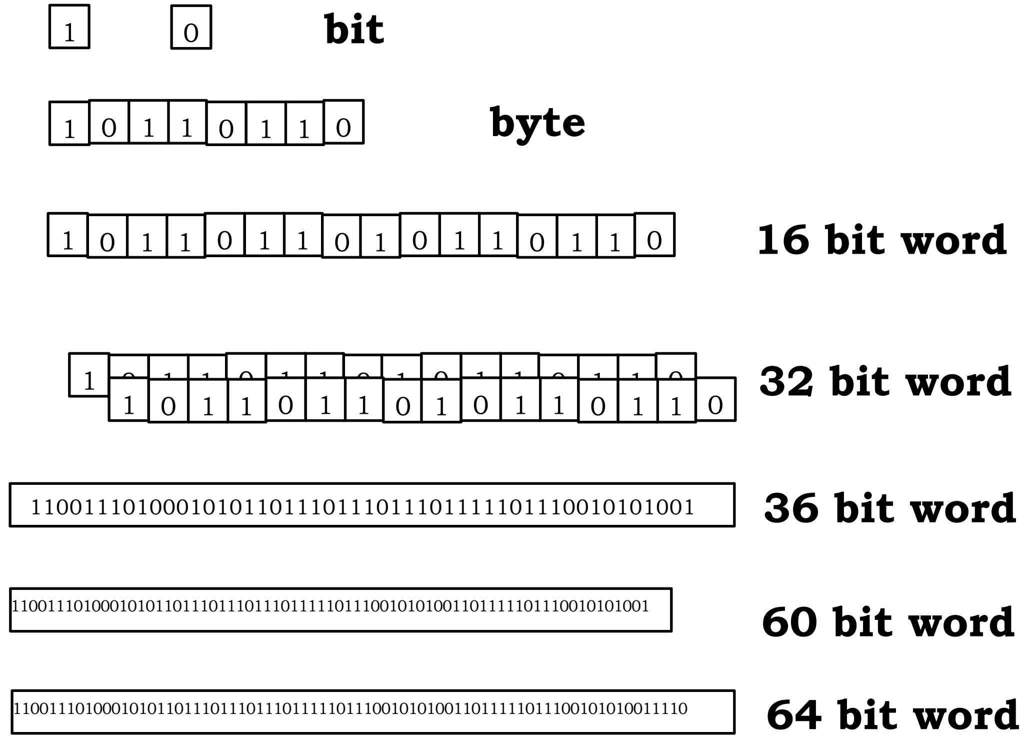


Transitions <0.1t
Worry about propagation delays

## Components

Von Neumann architecture

A computer has a **memory** to store **data** and **instructions**.
A CPU which loads data from memory to **registers** and operates on the data with the **Arithmetic-Logic Unit (ALU)**
All data and instructions are stored in memory as words.



PDP 11
Intel 286

Word length architecture specific

VAX
Pentium

PDP 10

Cray 7600

1974

Alpha

Intel

**Memory** contains data or instructions
memory locations can be read or written.

**Registers** the values in the registers can in addition can be modified.

David Patterson

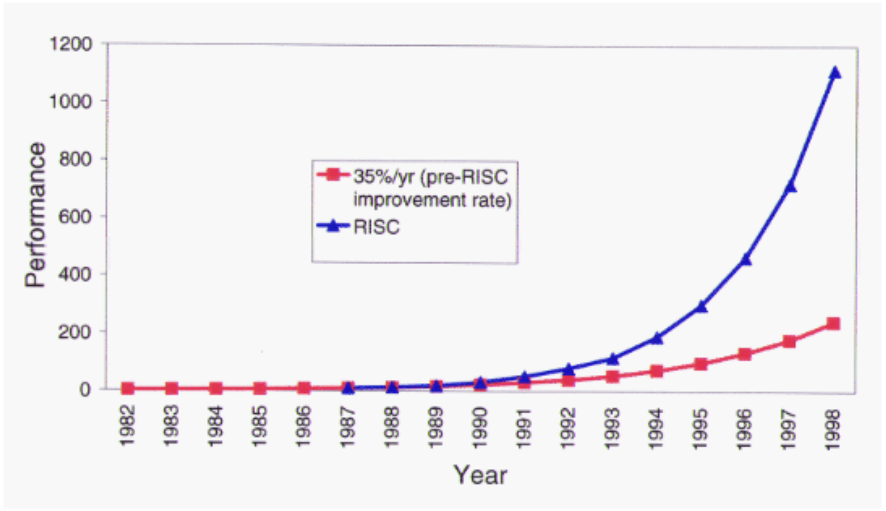
Berkeley

## Von Neumann Architecture

Lectures are based round the RISC architectures which were developed by Hennessey and Patterson.

Why?

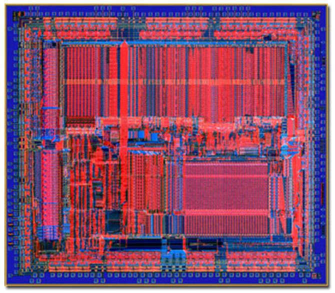
RISC machines have driven the improvements in computing power over the last 25 years.



All RISC machines based on their work.
Textbooks based on their work.

Intel chips are an inelegant combination of RISC/CISC made fast by brute force.
Understanding how they work requires
Understanding the underpining.

John Hennessey


MIPS R2000

Stanford

Von Neumann architecture

Computer Architecture

Computer Organisation and Design

## Instructions

Use the MIPS as an example

Used in Patterson & Henessy; Hennessy & Patterson; other texts

Clean (simple) architecture
Intel architecture is complicated by need to remain backward compatible to the 8088 chip. Introduced 1979. Cut down version of the 8086 1978.
If computer history goes back to 1948 ....
This represents something ½ the age of the programmable computer!

High level language instructions

$$\mathbf{C = A + B}$$

are translated to a number of "low level" instructions which are understood by the chip.

MIPS architecture Arithmetico-Logical operations only on registers.

Instructions are divided into a number of subtypes.

Memory is divided into words. Each word is four bytes.
Addresses are in bytes.
Words are **aligned** byte address is a multiple of 4 (bottom of memory is word 0)

## Von Neumann Architecture

An instruction is enabled (executed) if it is pointed to by the program counter (PC).
The PC is incremented by 1 word (4 bytes) after every instruction, except when a jump is executed or a branch is taken.

## Data flow Architecture

Different architecture:
An instruction is enabled if all the operands are available to it.
An instruction is fired when it is enabled and resources are available for its execution.

Data flow architectures are easiest to implement for single assignment languages.

At low level modern processors have elements of both.

**Single assignment language.**
A language where a variable is only allowed to be on the left hand side of an assignment once.
eg Erlang
No anti-dependencies or output dependencies.

Agner Krarup
Erlang

Single assignment language.
Good for parallel

Erlang, developed by Ericsson for telecoms apps.

Written 1986
Open source 1998

2009 gaining in popularity