COMP2221 Networks

David Head

University of Leeds

Lecture 9

Reminder of the Last Lectures

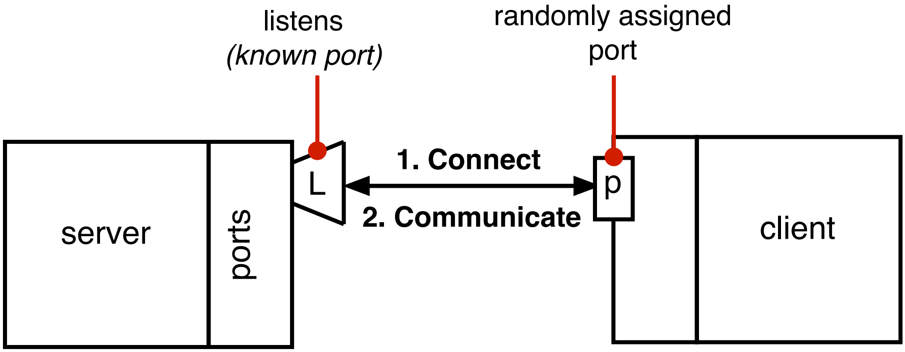
In the last two lectures we have seen how to implement a client and a simple server in Java.

The **client** uses the `Socket` class to:

- Contact the server on a prescribed host **and port**.
- Communicate according to an agreed **protocol**.

The **server** uses the `ServerSocket` class to:

- **Listen** to the prescribed port by calling `accept()`.
- Once a connection is made, `accept()` returns a `Socket` object that the server can use to implement its part of the protocol.


listens
(known port)
randomly assigned
port
server
ports
L
1. Connect
2. Communicate
p
client

Problems with the simple server

We also mentioned some **limitations** of the simpler server:

`accept()` is **blocking**:

- It only returns once a client makes contact.
- In the meantime, the **entire server application is doing nothing**.

Only communicates with **one client at a time**:

- Other clients will get **queued**, leading to delays.
- Server may be frequently **idle** during communication with the client (*i.e.* while waiting for a response).
- Only one **protocol** handler per server, rather than one per client (*e.g.* `KnockKnockProtocol`).

Today's lecture

Whereas creating one protocol handler per client is conceptually simple, the other problems are more serious.

There are two solutions:

1. Use **non-blocking communication** as provided by `java.nio`.
2. Use **concurrent threads**, with one or more clients per thread.

We will consider (2) over the next 3 lectures, and (1) in lecture 12.

Why parallel computation?

Today is the first of 3 lectures when we will consider **parallel computation**.

- *i.e.* writing software that makes use of hardware capable of performing calculations 'in parallel,' *i.e.* **simultaneously**.
- An increasingly important skill, as almost all modern hardware **is** parallel.

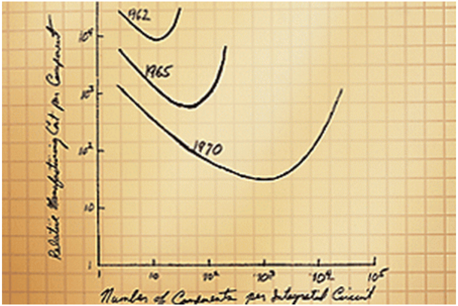
For the BCS-accredited programmes, you are required to cover some parallel programming.

- For some of you, this is the **only** time you will encounter parallel programming in your programme.
- Some optional Level 3 modules (*esp.* COMP3221) cover this much more extensively.

The rise of parallel architectures

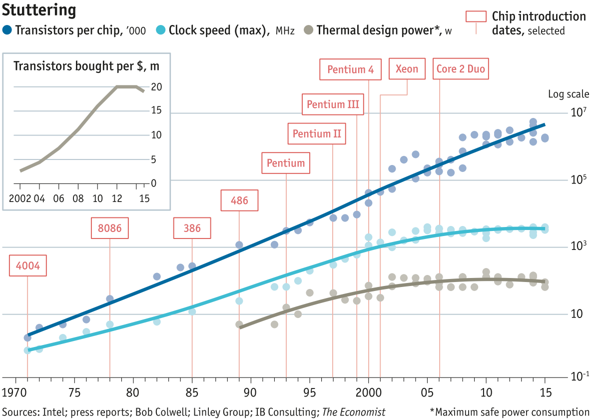
In 1965 Gordon Moore made the empirical observation that the density of transistors on a chip doubles every 18-24 months.

- Plotted component cost *vs.* no. of components.
- Included projected 1970 data.
- Identified **exponential increase** of the most cost-effective number of components.



https://image.slidesharecdn.com/mooreslaw-090930204517-phpapp01/95/moores-law-25-728.jpg?cb=1254343796

Although the law still holds, processor speeds no longer track it:

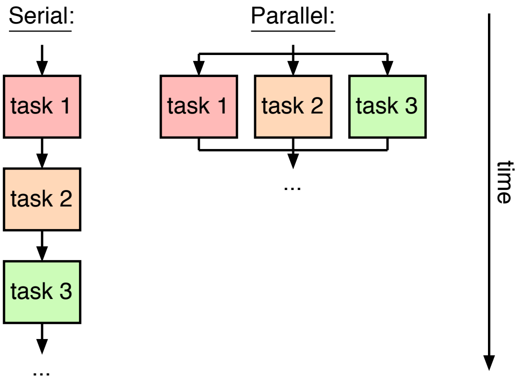


`http://www.economist.com/technology-quarterly/2016-03-12/after-moores-law`

The problem is heat generation increases rapidly with clock speed.

The benefits of parallelism

One way to improve performance is to allow calculations to be performed **simultaneously**, *i.e.* in **parallel**:



The limits of automated parallelism

Chip designers have already tried to automate some parallelism.

Instruction-level parallelism (ILP) processors use a *pipelining* architecture in which each instruction is processed in multiple stages, somewhat like an assembly line at a factory.

- Different stages for subsequent instructions can be performed simultaneously.

CPUs also have **multiple functional units** such as FPUs, ALUs *etc.* that can work independently.

- **Superscalar** architectures can merge nearby calculations into a single one.

However, each of these have limits - they do not **scale**.

Current parallel architectures

Almost all modern architectures contain multiple **processing units**, *e.g.* multiple cores, CPUs *etc.*

Pro: **Scales** better than previous automated parallelism.

Con: Software must be **specifically developed** to take into account these processing units.

Current parallel hardware can be broadly classified into:

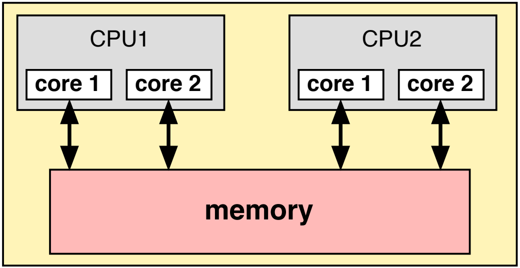
1. Shared memory architectures.
2. Distributed memory architectures.
3. Graphics Processing Units or **GPU**s.

1. Shared memory architectures

Definition

All **processing units** *(e.g. cores)* access the same memory.

This includes anything with one or more **multi-core CPUs**, so almost all modern desktops, laptops, tablets, smart phones *etc.*

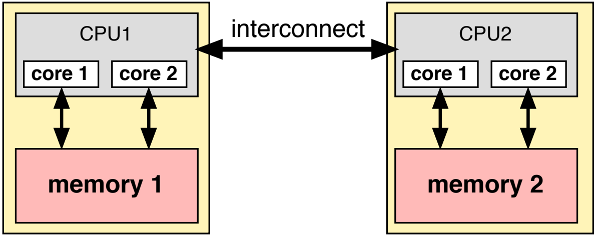


2. Distributed memory architectures

Definition

Processing units only access a **fraction** of total memory available

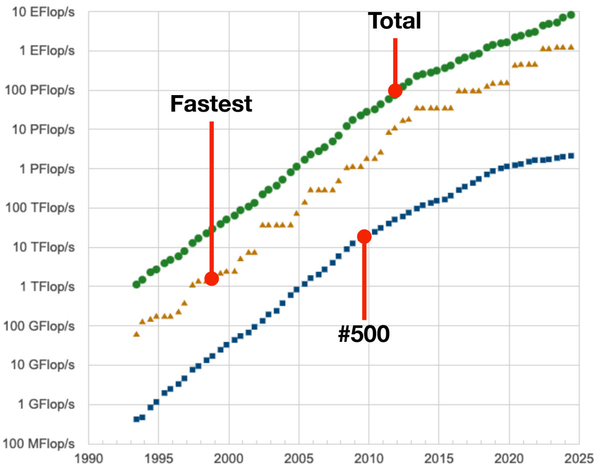
Includes **High-Performance Computing** (HPC) clusters (*e.g.* supercomputers), and **distributed systems** *('cloud computing')*.



Benchmark speeds of **all** supercomputers in the world [Nov. 2024].

1 EFlop/s
= 1 **exa-FLOP**
= 10^{18} floating point operations per second.

[From top500.org]



3. Graphics Processing Units (GPUs)

GPUs

Hardware specifically designed to rapidly perform calculations that arise in the graphical rendering of scenes

Historically driven by graphics applications, especially video games, but increasing being used for **non-graphical** applications such as **machine learning**, **cryptocurrencies** *etc.*

- Some devices design for general-purpose calculations.
- Known as **GPGPU**s for General-Pupose GPUs.

GPU hardware has multiple memory stores, some of which can be viewed as distributed, some shared.

The most powerful ‘computer’ known?

Arguably the most complex system known is the **human brain**.

If regarded as a computer, it would be **massively parallel**:

- Synapse speeds are about 5 ms, so the ‘clock speed’ would be less than 1kHz.
- We have about 10^{11} neurons, each connected to 10^4 others.
- The current fastest supercomputer has 10^7 cores.

http://scitechconnect.elsevier.com

How to program parallel architectures?

Parallel architectures allow multiple calculations to be performed **simultaneously**.

- Can improve performance without increasing the clock speed.

However, what if one calculation requires the result of *another* calculation as input? How can they be performed *in parallel*?[1]

Algorithms typically need to be **redesigned** to make good use of parallel hardware. These new algorithms need to be designed and implemented by **programmers**, *i.e.* **us**!

[1] *Answer:* They can't ...

Concurrency

Concurrency is when two or more tasks are in progress in the same **time frame**[1].

For instance, for an event-driven GUI, a user event (*e.g.* a mouse click) might result in a **callback function** being called.

- Would normally be called by a separate task *(thread)*, so that the main loop can continue (and detect other user events).

Can be achieved using **interrupts**:

- OS 'slices' CPU time amongst all running tasks.
- Most commonly **pre-emptive** multi-tasking.

[1]McCool *et. al.*, *Structured Parallel Programming* (Morgan-Kaufman, 2012).

Parallelism

An increasingly important concept related to concurrency is **parallelism**.

Parallelism refers to the ability to perform **multiple calculations simultaneously** by using more than one **processing unit**.

Possible to be concurrent but *not* parallel:

- Multi-tasking is possible on a single-core CPU, which only has one compute unit (*i.e.* the core).

Parallel $\implies$ Concurrent **but** Concurrent $\not\Longrightarrow$ Parallel

Processes *versus* threads

There are two basic units of execution, **processes** and **threads**.

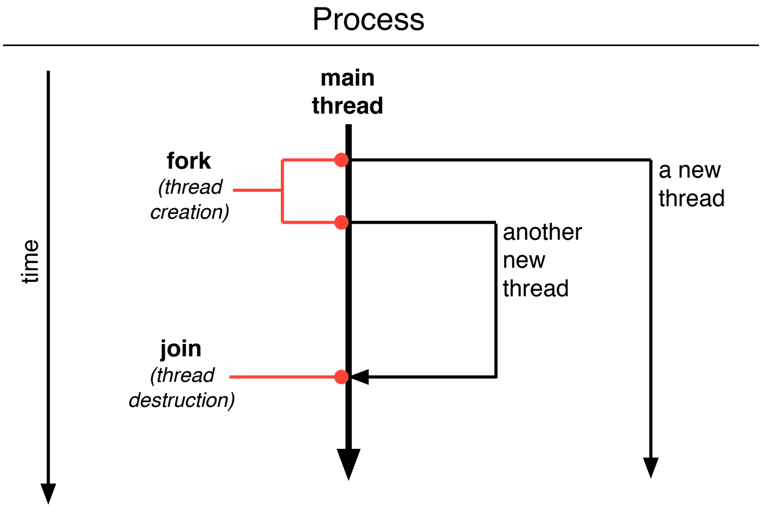
Process:

- A **self-contained** execution environment.
- Private set of run-time resources.
- Has its own **heap** memory.
    - All objects created with `new` (inc. arrays), global variables *etc.*
- Also has its own **stack** memory, which has local variables, function arguments *etc.*
- **Expensive** to create and destroy.
- Normally 1 application = 1 process, but not always:
    - For example, each **tab** in an application may correspond to a different process

Threads:

- Are **launched by**, and **exist within**, a process.
- Every process has at least one (the **main** thread in Java).
- Have their own stack memory.
- **Shares** the heap memory with the launching process.
    - Potential problems if multiple concurrent threads can read and write the same data.
- Is **lightweight**; small cost for creating and destroying threads.
- How threads are assigned to cores is up to the OS, more specifically the **scheduler**.

One process, multiple threads



Parallel languages and frameworks

Most languages support parallel programming at some level:

- C, C++ has libraries/standards, *e.g.* OpenMP, Cilk, the `pthread` library, *etc.*
- C++11 has native multi-threading support.
- Python has a `threading` library[1].

Even early versions of Java supported multi-threading at the **language level**.

- At least partly motivated by event-driven GUI design.
- *i.e.* `java.lang.Thread`, `java.util.concurrent`.

[1] Although need to work around its Global Interpreter Lock (GIL) to exploit multi-cores.

Summary and next time

Today we have looked at parallel architectures and software.

- Why parallel computation is becoming increasingly important.
- Shared *versus* distributed memory.
- Concurrency *versus* parallelism.
- Processes *versus* threads.

Over the next two lectures we will various ways of implementing a multi-threaded server in Java.