Computers are always too slow!

The Free Lunch is over!

- Don’t expect your sequential program to run faster on new processors (Moore’s law: CPU/memory speed doubles every 18 months)
- Still, processor technology advances
- BUT the focus now is on multiple cores per chip
- Today’s desktops typically have 8 cores.
- Today’s servers have up to 64 cores.
- Expect 100s of cores in the near future.
- Additionally, there is specialised hardware such as multi-byte vector processors (e.g. Intel MMX - 128 bit) or high-end graphics cards (GPGPUs)
- Together, this is a heterogeneous, high-performance architecture.

Clock Rates

![CPU clock frequency, 1993–2005](image)

- Intel
- AMD

CPU power consumption, 1993–2005

![CPU power consumption, 1993–2005](image)
Supercomputers

The Hector supercomputer at the Edinburgh Parallel Computing Center (2011):
- total of 464 compute blades;
- each blade contains four compute nodes,
- each with two 12-core AMD Opteron 2.1GHz Magny Cours processors.
- Total: **44,544 cores**
- Upgraded in 2011 to 24-core chips with a total of **90,112 cores**

Hector is *out-dated* and will be turned off in March 2014. The new supercomputer at EPCC is Archer:
- Cray XC30 architecture
- uses Intel Xeon Ivy Bridge processors
- total of 3008 compute nodes
- each node comprises two 12-core 2.7 GHz Ivy Bridge multi-core processors,
- Total: **72,192 cores**
- Peak performance: **1.56 Petaflops**
- each node has at least 64 GB of DDR3-1833 MHz main memory,
- scratch disk storage: Cray Sonexion Scalable Storage Units *(4.4PB at 100GB/s)*
- all compute nodes are interconnected via an Aries Network Interface Card.

For details on Archer see: [https://www.epcc.ed.ac.uk/media/publications/newsletters/epcc-news-74](https://www.epcc.ed.ac.uk/media/publications/newsletters/epcc-news-74) and [www.archer.ac.uk](http://www.archer.ac.uk)

Application Areas of High-performance Computing

Ever increasing demand for compute power:
- **data mining**: seeking statistically significant connections in large, disparate data sets
- **weather forecasting**: calculating interactions between temperature/pressure readings through the atmosphere across the planet
- **genome analysis**: matching long protein sequences against each other
- **3D graphics**: rendering rapidly changing, rich object models in real time
- **engineering simulation**: calculating interactions under stress of all components of a complex device
- **VLSI development**: fully testing VLSI designs in software before committing to production
The major distinction is between:
- Single Instruction Multiple Data (SIMD);
- Multiple Instruction Multiple Data (MIMD)

SIMD typically involves specialised CPU & communications
- Control CPU + multiple ALUs e.g. CDC 6600
- Today’s graphics processors (GPGPUs)

MIMD typically involves specialised communications
- Point to point on channels e.g. Meiko Computing Surface
- Communication hierarchy e.g. nCube, BBN Butterfly

Few firms now still make specialised parallel systems in hardware (IBM, NEC, Sun, Cray)

Specialised parallel systems are very expensive

Standard von Neumann CPU price/performance always outstrips specialised systems (e.g. Beowulf clusters)

In late 20th century, military was most significant customer e.g. nuclear weapon simulation; early warning systems; ICBM guidance; Star Wars

End of Cold War led to reduced defence spending

Until recently, most machines in today’s TOP500 list of fastest supercomputers use standard hardware:

http://www.top500.org/list/2013/11/

Today, the fastest supercomputers use graphics processors (Tesla) or many integrated core (MIC) architectures (Xeon Phi).

Parallel hardware is increasingly heterogeneous:
- Often SIMD components complement von Neumann CPUs in standard microprocessors
  - digital signal processing (DSP) on vectors of bits
  - mainly for graphics and animation
    - e.g. NVidias Tesla cards or Intel MMX instructions
- Poor support in compilers: the programmer must drop into assembly language
- No generic libraries: compiler specific

**Classes of Architectures**

- **Shared memory**: CPUs access common memory across high-speed bus

- **Symmetric Multi-Processing (SMP)**, e.g. Sun SMP
  - Advantage: very fast communication between processors
  - Disadvantage: bus contention limits number of CPUs

- **Hierarchical SMP**, e.g. IBM ASCI White, 1,512 * 16 PowerPC SMP
**Classes of Architectures**

- **Distributed memory**: CPUs communicate by message passing on dedicated high-speed network (e.g. IBM SP2, Cray T3E)

  - advantage: highly scalable
  - disadvantage: explicit data communication is relatively slow

**Today’s Main High-performance Architectures**

- Most contemporary MIMD systems are based on clusters of commodity workstations linked via high-speed switch & dedicated fast LAN, e.g. **Beowulf**
  - first developed at NASA in 1994
  - usually built from PCs but also Sun workstations, Apple etc
  - no graphics, sound, display, keyboard, mouse
  - fast network card
  - fast hard disk
  - lots of memory
  - Linux + C + MPI
    - advantage: very cheap, good scalability
    - disadvantage: slower than specialised architectures

**Parallel Programming: implicit parallelism**

*Implicit parallelism:*

- compiler/run time system exploits parallelism latent in program
  - e.g. High Performance Fortran
- avoids need for programmer expertise in architecture/communication
- identifying parallelism in programs is hard and undecidable in general
- requires complex usage analysis to ensure independence of potentially parallel components
- typically look for parallelism in loops
- check that each iteration is independent of next

**Advantages**

- no programmer effort

**Disadvantages**

- parallelising compilers very complex
- beyond common patterns of parallelism, often human can do better

**Example Code**

A **good** example:

```c
for (i=0; i<n; i++)
a[i] = b[i]*c[i]
```

- no dependency between stages
- could execute `a[i] = b[i]*c[i]` on separate processors

A **bad** example:

```c
for (i=1; i<n; i++)
a[i] = a[i-1]*b[i]
```

- each stage depends on previous stage so no parallelism
Parallel Programming: explicit parallelism

**Explicit parallelism:**
- programmer nominates program components for parallel execution
- three approaches
  - extend existing language
  - design new language
  - libraries

Parallel Programming: extend existing languages

**Extend existing languages**
- add primitives for parallelism to an existing language
  - advantage:
    - can build on current suite of language support e.g. compilers, IDEs etc
    - user doesn’t have to learn whole new language
    - can migrate existing code
  - disadvantage
    - no principled way to add new constructs
    - tends to be ad-hoc,
      - i.e. the parallelism is language dependent
  - e.g. many parallel Cs in 1990s
  - none have become standard, yet
  - an emerging standard is Unified Parallel C (UPC)

Parallel Programming: language independent extensions

**Use language independent extensions**
- Example: **OpenMP**
- for shared memory programming, e.g. on multi-core
- Host language can be Fortran, C or C++
- programmer marks code as
  - `parallel:` execute code block as multiple parallel threads
  - `for:` for loop with independent iterations
  - `critical:` critical section with single access

Parallel Programming: compiler directives

**Compiler directives**
- advantage
  - directives are transparent so can run program in normal sequential environment
  - concepts cross languages
- disadvantage
  - up to implementor to decide how to realise constructs
  - no guarantee of cross-language consistency
  - i.e. the parallelism is platform dependent
Parallel Programming: Develop new languages

Develop new languages

- advantage:
  - clean slate

- disadvantage
  - huge start up costs (define/implement language)
  - hard to persuade people to change from mature language

- Case study: sad tale of INMOS occam (late 1980's)
  - developed for transputer RISC CPU with CSP formal model
  - explicit channels + wiring for point to point CPU communication
  - multi-process and multi-processor
  - great British design: unified CPU, language & formal methodology
  - great British failure
  - INMOS never licensed/developed occam for CPUs other than transputer
  - T9000 transputers delivered late & expensive compared with other CPUs libraries

Parallel Programming: Language independent libraries

Language independent libraries:

- most successful approach so far
  - language independent
  - platform independent

- Examples:
  - Posix thread library for multi-core
  - Parallel Virtual Machines (PVM) for multi-processor
  - Message Passing Interface (MPI) for multi-processor

- widely available for different languages under different operating systems on different CPUs
  - e.g. MPICH-G: MPI for GRID enabled systems with Globus

- we will use \textit{C with MPI} on Beowulf

Reading List

Parallel Programming (introduction and low-level programming):

  (a good but dated introductory on-line textbook on parallel programming; focus on Part I, covering concepts and parallel program design)

  (standard textbook on parallel programming; mainly numerical algorithms and message passing but also a little bit OpenMP (shared-memory))

  (another recommended textbook)

Parallel Patterns (high-level programming):

  (a book on parallel patterns, aimed at practitioners in parallel programming, using Cilk and TBB as frameworks)


See the on-line reading list for the course: http://www.macs.hw.ac.uk/~hwloidl/Courses/F21DP/index.html#reading