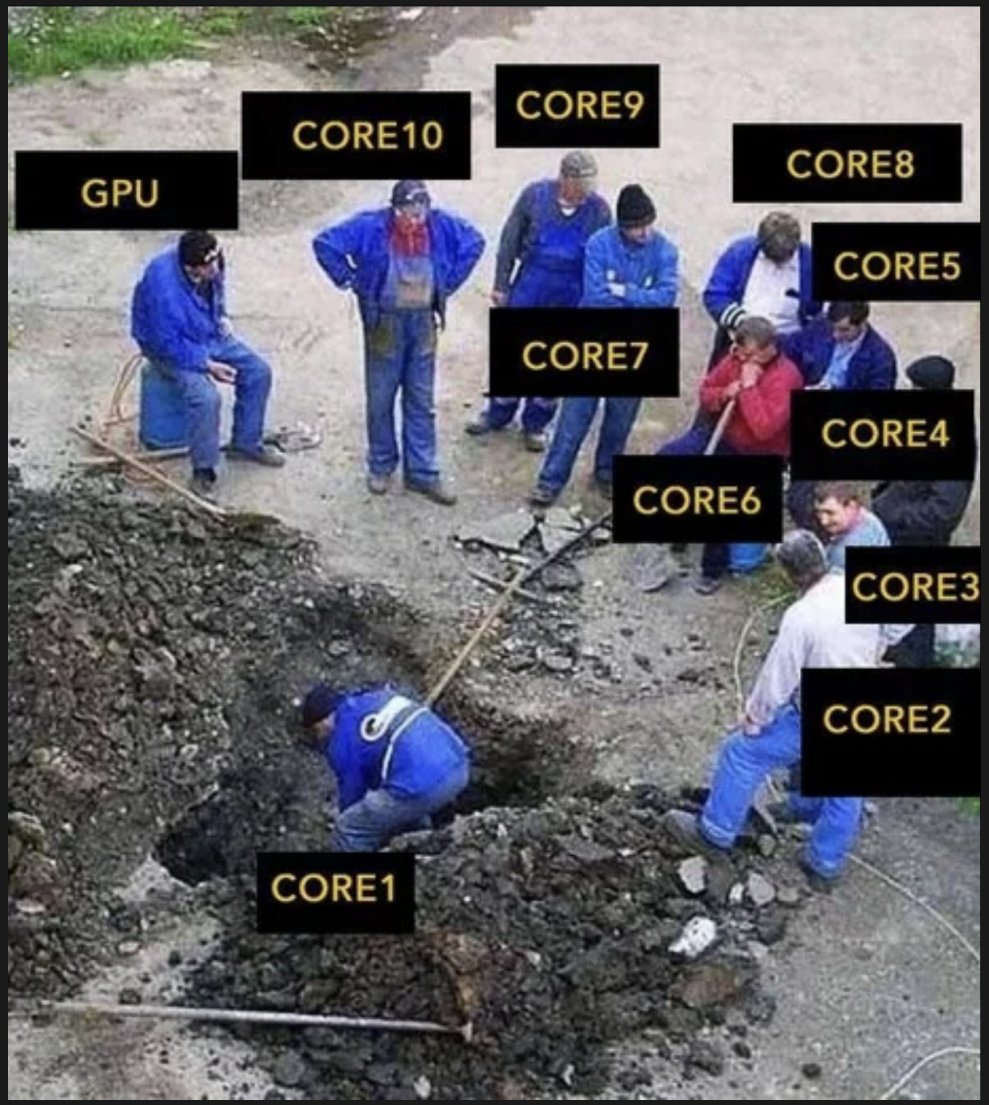
Introduction to Parallelism

Daniel Lawson — University of Bristol

Lecture 09.1 (v4.0.0)

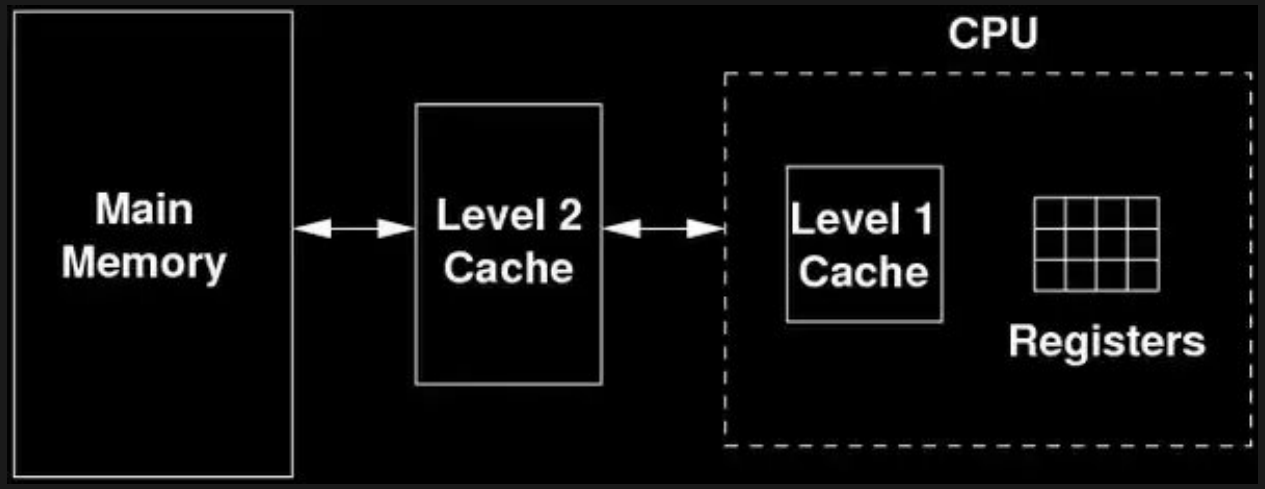
Working in Parallel



Questions

- When can an **algorithm** be run in parallel, i.e. concurrently?
- Which **parts** of an algorithm can be sped up?
- What **scale** of parallelism are possible?
  - ... within a processor?
  - ... across components on a single computer?
  - ... across machines within an institution?
  - ... distributed across time and space?

CPUs are parallel processing units



- Each CPU (central processing unit) is a sophisticated architecture.
- Parallelism exists in:
  - how the CPU accesses memory,
  - how memory is structured (L1 cache, general memory),
  - how the CPU processes registers...
- You only need to write **vectorized code** in order to access this.

Computers are parallel processing units

- General purpose parallelism can be:
- A single machine containing a **multi-core CPU** (central processing unit):
  - Most commonly coded with OpenMP,
  - The cores share memory and are multipurpose, and hence coding is easy,
  - Accessed via simple libraries.
- A **GPU** (graphical processing unit):
  - Most commonly accessed via **libraries**, e.g. OpenCL, PyTorch, etc.
  - Contain a large number of relatively limited cores that can perform simple computations (e.g. matrix operations; linear computations) efficiently,
  - Dedicated GPU hardware is multipurpose (e.g. TPU), i.e. has an increased feature set.

Clusters of computers are parallel processing units

- **Multiple machines** act as a processing unit, either:
- A set of (identical) machines on a high-bandwidth network connection, able to perform **computation as a coherent unit**.
  - Extremely flexible and the most popular setup; a "supercomputer".
  - Coded with Hadoop, Spark, OpenMPI, etc depending on goal.
- **Massively distributed computing**, able to communicate but not rely on one another.
  - Non-realtime computations can be distributed and returned when ready.
  - For example, **SETI@Home**; **Folding@Home**, internet routing; low priority access to Amazon AWS/Azure.
  - Biological decision making (the brain),
  - Societal decision making (social insects, humans).

Formal classes of parallel computer

- Computer scientists may think in terms of **control** (instruction sets) and **processing** (data streams):
- Single Instruction stream, Single Data stream (**SISD**):
  - Single control, single processor. A sequential processor, as we conceptualise a computer.
- Single Instruction stream, Multiple Data stream (**SIMD**):
  - Single control, multiple processors. dedicated to *vector calculations*.
- Multiple Instruction stream, Single Data stream (**MISD**):
  - Used for streaming computations (e.g. *splitting pipes*) for fast response, e.g. the space shuttle. . .
- Multiple Instruction stream, Multiple Data stream (**MIMD**):
  - Multiple control, multiple processors.
- They also think in terms of shared vs distributed (interconnected) memory.
  - e.g. MIMD **may** have distributed memory.

Parallel algorithms for data science

- Most parallel coding is about thinking about your problem:
  - What **dependencies** (on the output of some other computation) really exist?
  - How can you write code avoiding **unnecessary dependencies**?
- There are hardcore parallel algorithms and paradigms. We just need to know:
  - Should we try to parallelise to solve a particular problem?
  - Will simple tricks work for you?
- This involves describing your problem in a well-supported paradigm

Embarrassingly parallel algorithms

- The meaning of the word is as in:

`"an embarrassment of riches..."`

- **embarrassingly parallel** algorithms are the most important class.
  - In these, there is **no dependency** between threads.
  - You can run them in an arbitrary order, in series if needed.
- Most parallel coding is about turning a problem into a series of embarrassingly parallel algorithms.
  - Vectorisation is the first

Embarrassingly parallel examples

- **Monte-Carlo sampling** (for integration or search):
  - Run a large number of independent, randomised processes.
- **Grid search** or **Latin hypercube sampling**:
  - Run a large and (pre-defined or algorithmically defined) set of independent processes.
- Independent **database queries** (assuming database storage is distributed with compute)
- **Rendering** of graphics in games/video editing
- **Note**:
  - Still not trivial be to implement if **memory** or **communication** bandwidth becomes limiting.

Vectorisation

- Vectorised code is parallelised code.
  - Each operation for vectorised code is computable independently
  - The **same** operation is applied to each element (with different data)
  - **CPU optimisation** is possible and may be straightforward
  - GPU acceleration is possible
  - Vectorisations are always **one dimensional** representations
- A set of standardized elementwise computations is possible:
  - addition, subtraction, multiplication, division
  - other operations are possible, this becomes architecture dependent

Vectorisation of K-dimensional objects

- Matrices can be represented by standardized vectorisation procedures
  - $A = \begin{bmatrix} a & b & c \\ d & e & f \end{bmatrix}$
  - **Row major** order: $\mathrm{vec}(A) = (a, b, c, d, e, f)$
  - **Column major** order: $\mathrm{vec}(A) = (a, d, b, e, c, f)$
- Matrix multiplication:
  - Is just sums of the correct components of the vectorised matrices
  - Choice of row vs column major order affects efficiency!
- Parallelization:
  - On a **shared memory** machine, the computations are distributed
  - Otherwise a **memory distribution** problem
- Efficient implementations for many common computations

Vectorisation and time complexity

- Assuming no parallelization:
- A **for loop** with N iterations is $O(N)$
- A **vectorisation** with N elements is $O(N)$
- But the vectorised code may still be orders of magnitude faster:
  - It often can be pushed into low-level code (C backend)
  - It can exploit CPU memory architecture: **caching** the correct content to avoid overhead
  - It can exploit CPU compute architecture: **multiple registers** in parallel
- Vectorisation also leads directly into parallel implementations:
  - It emphasises dependencies,
  - It encourages reordering of loops which can reduce time complexity.

Portfolio 09.1.1 - Vectorisation in pytorch

- Neural networks are usually expressed almost entirely as tensor operations (matrix products, convolutions, element-wise functions). Why has modern machine learning converged on this representation?
  - Explain how automatic differentiation makes this possible.
  - Can non-neural network models be learned the same way? What are the limits?

References

- A Brief Overview of Parallel Algorithms
- Parallel computing concepts e.g. Amdahl's Law for the overall speedup
- MISD/MIMD/SIMD/SISD
- Parallel time complexity