Introduction to Parallelism (Part 1, Parallel computers)

Daniel Lawson — University of Bristol

Lecture 10.1.1 (v1.0.2)

Signposting

- Block 10 on **parallel algorithms** is paired with Block 11 on **parallel infrastructure**.
  - Block 08 on **Algorithms** is the also highly relevant.
  - Specific content includes **complexity**.
- The block is split into Lecture 10.1 (Introduction) and a Workshop 10.2.
- The lecture is split into two parts
- This is 10.1.1, covering:
  - What is a parallel computer?
  - How to design code that parallelises,
  - Parallelism and complexity,
  - Computation graphs.

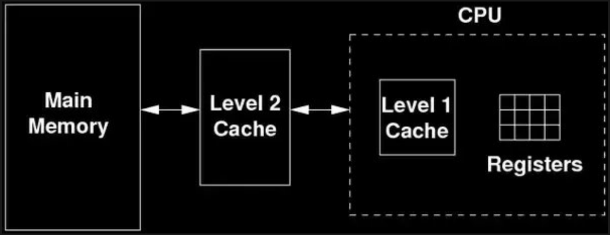
ILOs

- ILO4 Be able to use high throughput computing infrastructure and understand appropriate algorithms

Parallelism

- Parallelism is a concept that exists at many levels
- Can an **algorithm** be run in parallel, i.e. concurrently?
  - Compared with: must some computations be performed sequentially?
- 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 either 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 coded with OpenCL or libraries that enable this,
  - Contain a large number of relatively limited cores that can perform simple computations (e.g. matrix operations; linear computations) efficiently,
  - Dedicated scientific GPU hardware is increasingly multipurpose, 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,
  - Societal decision making.

Formal classes of parallel computer

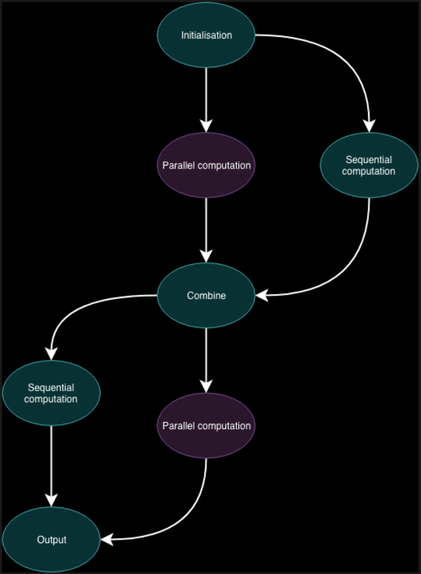
- Computer scientists may think in terms of **control** (instruction sets) and **processing** (data streams):
- Single Instruction stream, Single Data stream (**SISD**):
  - Single control, single processor. A sequential processor, as we conceptualise a computer.
- Single Instruction stream, Multiple 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

Computation Graph

- How is the computation structured?
- Which parts are parallisable?
- Where is the output of one computation required?
- In this illustration,
    - Gather Input
    - Do something in parallel
    - Collect the answer
    - Do something else in parallel
    - Collect the answer
    - Return
- There are always sequential limits in e.g. memory allocation, variable construction, etc.



Computation dependence

- **Real example**: Dimensionality reduced similarity

```
procedure Example(x[], n, m)
    while i ≤ n do
        y[i] ← f(x[i]; m)
    end while
    while i ≤ n, j ≤ n do
        z[i, j] ← g(y[i], y[j]; k)
    end while
    return z
end procedure
```

- Consider a matrix x of dimension n items with m features **and p CPUs**.
  - First: compute $y_i = f(x_i)$, a vector of length $k \ll m$
  - Second: compute a similarity

$$z_{ij} = g(y_i, y_j) \approx g'(x_i, x_j)$$

- Raw cost:

$$\Theta(nm + n^2 k)$$

- Parallel cost:

$$\Theta(\lceil n/p \rceil m + \lceil n^2/p \rceil k)$$

Parallel speedup

- There are two key concepts:
- Total **Speedup** $S_t := \frac{\text{Sequential algorithm runtime}}{\text{Parallel algorithm runtime}} = T_s/T_p$.
  - The speed benefit of running compute in parallel
  - $S_t = t/p$ in the best case (for total time t and p processors)
- **Work efficiency** $E := \frac{\text{Total Sequential compute}}{\text{Total Parallel compute}}$.
  - The efficiency penalty for running in parallel
  - $E = 1$ in the best case.
- For example:
  - If the runtime t decreased as $t = \Theta(\log(n)/p)$,
  - and we used $p = \sqrt{n}$ processors,
  - then the speedup is $\sqrt{n}/\log(n)$ whilst the efficiency is $1/\log(n)$.
  - These can be defined both for actual times, and rates.

Maximum speedup

- **Amdahl's Law**: $\text{Max speedup} = \frac{1}{(1-P)+P/S_p}$
  - where P is the **parallelisable proportion** of the algorithm
  - and S_p is the **Speedup** for the parallelisable proportion
  - This follows directly from writing the compute time of the parallel algorithm:
$$T_t = (1-P)T_s + PT_s/S_p$$
  - it asymptotes to $1/(1-P)$
- It **doesn't matter how much compute** resource you throw at a problem, you can't reduce it further than this!

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.

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**:
  - May not trivial be to implement if **memory** or **communication** bandwidth becomes limiting.

Reflection

- What are the differences between a compute node with a GPU vs many CPUs?
- What concepts are needed to describe computational complexity of a parallel algorithm?
- Can you think of any non-trivial embarrassingly parallel algorithms?
- By the end of the course you should:
  - Know the types of parallel computer
  - Be able to construct and understand very simple computational graphs
  - Understand parallel speedup in the context of computational complexity

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