High Performance and Parallel Computing

Semester 2, 2024

Associate Professor Wayne Kelly

Tarang Janawalkar

Contents

1 Processor Design

High performance programs must be designed to take advantage of the hardware they run on. This section discusses how Central Processing Unit (CPU) design influences how programs are executed, and why parallelisation is necessary for high performance computing.

1.1 Von Neumann Architecture

Most general purpose computers are based on the von Neumann architecture where instructions and data are kept in the same memory. In this model, the CPU performs a fetch-decode-execute cycle to execute instructions, as described below:

1. **Fetch**: The CPU fetches the next instruction from memory into the instruction register and increments the program counter.
2. **Decode**: The CPU decodes the instruction to determine what operation to perform. This may involve one of the following:
    - Load data from memory into a register.
    - Store data from a register into memory.
    - Perform an arithmetic or logical operation on data in registers.
    - Transfer control to another part of the program by changing the program counter.
3. **Execute**: The CPU performs the operation specified by the instruction. This may involve the Arithmetic Logic Unit (ALU) for arithmetic and logical operations, or the bus/memory module for memory operations.

1.2 CPU Clock

Each instruction in the fetch-decode-execute cycle takes a certain amount of time to complete. This time is determined by the speed of the CPU which is controlled by a clock that generates a series of pulses at a fixed frequency. These pulses are called **clock cycles** and the time taken for one cycle to complete is called the **clock period** of the CPU. This period must be long enough to:

- allow electrical signals to propagate through the CPU, and
- allow transistors to reach a steady state when switching between on and off.

An increase in clock frequency therefore results in a faster CPU, but due to the physical limitations of the speed of light and the size of transistors inside the CPU, single-core CPU performance has plateaued over the past decade.

Note in some cases, the CPU may be “over-clocked” to run at a higher frequency than it was designed for, but this can result in incorrect operation if the physical components are unable to switch states fast enough, and lead to overheating.

1.3 Von Neumann Bottleneck

Main memory resides outside the CPU, and is connected to the CPU via a shared bus. As the memory controller has a much lower clock frequency than the CPU, the CPU often stalls while a data operation is being performed between the CPU and memory. This is known as the **von Neumann bottleneck** and is a major limitation of the von Neumann architecture. To minimise performance loss, memory accesses should have:

- maximal throughout (i.e., transfer as much data as possible in one operation), and
- minimal latency (i.e., minimise the time taken to start and end an operation).

1.3.1 Cache Memory

A common way CPUs achieve this is by using **cache memory** to store frequently accessed data and instructions in small specialised blocks of memory that are close to each core. These caches provide higher throughput and lower latency than main memory, but are typically more expensive and therefore have limited capacity. A CPU often has multiple levels of cache, starting with the Level 1 (L1) cache which is the smallest and fastest, increasing in size and latency as the level increases. On modern CPUs, the L1 cache is typically split into two parts: one for *instructions* and one for *data*. Multi-core CPUs may also have unified L2 and L3 caches that are shared between cores.

Cache Operation The cache works by loading data in blocks of a fixed size called **cache lines**. When the CPU requests less than a cache line of data, it loads surrounding data into the cache to take advantage of locality of reference. This is done in the hope that the CPU will eventually request the surrounding data, which will already be in the cache and is therefore faster to access. When memory is requested, the CPU first checks if it exists in the cache, requesting it from main memory only if it is not found.

Cache Replacement When the cache is full, the CPU must decide what data to replace in the cache, through a *cache replacement policy*. Typically this is done using the Least Recently Used (LRU) policy, where the data that has not been accessed for the longest time is replaced. The *cache placement policy* determines where new data is placed, and can be:

- **Fully associative**, where any cache line can be replaced.
- **N-way set associative**, where the cache is divided into N sets, and the replacement policy is applied within each set.
- **Direct mapped**, where new data must be mapped to a specific cache line.

1.4 Instruction-Level Parallelism

Scalar processors are designed to process and execute instructions one at a time. High-performance processors take advantage of disjoint operations that can be performed in parallel through **instruction level parallelism** (ILP). This can be achieved in several ways:

- **Superscalar Execution**: Dispatching multiple scalar operations to different functional units (ALU, integer multiplication, FPU, store/load, etc.) during a single clock cycle. This is facilitated by hardware that determines dependencies between instructions and schedules them for parallel execution.

- **Out-of-Order Execution**: Executing other instructions while waiting for data required by the current instruction. As this can result in instructions being executed out of order, the processor must ensure that the final result is correct.
- **Instruction Pipelining**: Dividing instruction processing into multiple stages to keep the processor busy. For example the fetch-decode-execute cycle can be divided into three stages, with each stage being executed in parallel, allowing the processor to begin executing the fetch stage of the next instruction while the decode stage of the current instruction is being executed.

To keep the processor busy, processors may also employ **branch prediction** to predict the outcome of branch instructions and **speculative execution** to execute instructions that may not be required, but are likely to be executed.

Each of these techniques result more complex processor cores that occupy more space on the CPU chip, consume more power, and have longer cycle times.

1.5 Processor Architectures

There are two main types of processor architectures:

- **Complex Instruction Set Computing (CISC)**: Designed to execute complex instructions that can perform multiple operations in a single instruction. These instructions may be decoded into micro-operations that are executed by the CPU. These processors have large instruction sets and typically have slower clock speeds.
- **Reduced Instruction Set Computing (RISC)**: Designed to execute simple instructions that perform a single operation. These processors have smaller instruction sets and typically have faster clock speeds.

1.6 Multi-Core Processors

Moore's Law

Moore's Law states that the number of transistors on a CPU chip doubles approximately every two years. This has led to:

- Smaller circuits
- Faster clock speeds
- Additional complex hardware level optimisations

While this has held true for the past few decades, it is now increasingly difficult to reduce the size of transistors due to physical limitations, such as the size of a wire compared to the width of an atom.

Rather than allocating this additional space to a single core, CPUs now contain multiple cores that can execute an independent instruction stream[1]. Each core is itself a complete processor, with its own functional units, program counter, instruction register, general-purpose registers, and cache.

[1] Note that each core may execute the same stream if each stream operates on different data, as in Graphical Processing Units.

These cores can communicate with each other via shared memory, but this leads to the same bottleneck as before. To overcome this, cores may have their own private caches, and communicate with each other via a shared L2 or L3 cache.

1.6.1 Cache Coherence

When multiple cores share the same memory, it is important to ensure that each core has the most up-to-date copy of the data. Cache coherence is the consistency of data stored in multiple private caches that reference shared memory. Cache mechanisms are used to ensure that modifications to data in private caches are propagated to other cores to ensure other cores do not read stale data. This is typically done using *snooping* or *directory-based* mechanisms:

- **Snooping**: Each cache monitors the bus for access to memory locations that have been cached. If a write is detected, the cache either updates its data or invalidates it.
- **Directory-based**: A centralised directory keeps track of data shared between cores. Each processor must request permission to load an entry from a memory location. When an entry is updated, the directory either updates or invalidates other caches with that entry.

Caches also have inclusion/exclusion policies that determine whether data in a lower-level cache may be present in a higher-level cache.

1.7 Software Parallelism

Parallelism can be achieved at the software level through processes and threads. A **process** is an instance of a program being executed by the operating system that has its own memory space and threads of execution. Upon initialisation, a process creates a **thread** that executes its main function. This main thread can then spawn additional threads that execute independent streams of instructions.

A **thread** is a lightweight process that has its own program counter and local memory called the **stack**. The lifetime of the data in the stack is determined by the function that created it.

Every process also has a shared memory space called the **heap**, where data with indeterminate lifetimes is stored (i.e., dynamically allocated objects). This space can be accessed by threads within the same process, and by threads in other processes through inter-process communication APIs.

1.7.1 Mapping Threads to Cores

Operating systems achieve concurrency in threads through time slicing, where each thread is allocated a core for a fixed amount of time. When a time slice ends, the operating system performs a context switch to another thread by saving the current thread's state and loading the next thread's state. This allows multiple threads to make progress on a single core, and allows threads to be executed in parallel on systems with multiple cores. Note that for a CPU with N cores, a process with N threads may not always be granted all N cores, as the operating system may allocate some cores to other processes.

1.8 Simultaneous Multi-Threading

To reduce the overhead incurred during a context switch, some processors allow more than one thread to be executed on a single core at the same time by interleaving instructions from multiple

threads. This is known as **Simultaneous Multi-Threading (SMT)** or **Hyper-Threading**. In these processors, each core has multiple sets of registers and program counters (typically two) for each thread, with a shared set of functional units, that allow multiple threads to be executed in parallel.

2 Parallel Computing

2.1 Forms of Parallelism

Parallelism can be achieved at multiple levels:

- **Instruction Level Parallelism (ILP)**: Parallelism achieved by executing multiple instructions in parallel.
- **Vector Parallelism**: Parallelism achieved by executing the same instruction on multiple data elements simultaneously. This is typically done using **SIMD** (Single Instruction, Multiple Data) instructions.
- **Thread Level Parallelism (TLP)**: Parallelism achieved by executing multiple threads in parallel. This can be done using multiple cores, or by interleaving instructions from multiple threads on a single core.
- **Process Level Parallelism**: Parallelism achieved by executing multiple processes in parallel. Processes can communicate with each other through inter-process communication. Threads within the same process communicate via shared memory, while processes on different machines communicate via message passing. Distributed shared memory systems provide a shared memory programming model for distributed memory systems.

2.2 Types of Parallel Computers

2.2.1 Flynn's Taxonomy

Flynn's Taxonomy classifies parallel computers based on the number of instruction streams and data streams that can be processed at the same time. It has four categories:

- **Single Instruction, Single Data (SISD)**: A single instruction stream is executed on a single data stream. This is the traditional von Neumann architecture.
- **Single Instruction, Multiple Data (SIMD)**: A single instruction stream is executed on multiple data streams (includes superscalar processors). This is typically done using vector processors or GPUs.
- **Multiple Instruction, Single Data (MISD)**: Multiple instruction streams are executed on a single data stream. Commonly used in fault-tolerant redundant systems.
- **Multiple Instruction, Multiple Data (MIMD)**: Multiple instruction streams are executed on multiple data streams. This is the most common form of parallelism, and is used in multi-core CPUs and distributed systems, including both shared memory and distributed memory systems.

2.3 Types of Supercomputers

- **Vector Processors**: Processors that can execute vector instructions on multiple data elements in parallel. This includes early supercomputers like the Cray-1.
- **Symmetric Multi-Processor (SMP)**: Multi-core processors that share memory and have equal access to all resources.
- **Massively Parallel Processors (MPP)**: Tightly coupled systems with multiple processors that communicate with each other via proprietary high-speed interconnect.
- **Clusters**: Loosely coupled distributed memory systems that consist of commodity nodes.
- **Asymmetric Multi-Processor (AMP)**: Specialised co-processors that offload specific tasks from the main CPU (i.e., GPUs).
- **Hybrid Systems**: Systems that combine multiple architectures to take advantage of the strengths of each architecture.
- **Cycle Stealing Systems**: Systems that use idle resources on a network of computers to perform computations.

2.4 Parallel Computing Models

2.4.1 Concurrent Computing

A concurrent computing model is one where multiple operation streams progress independently. This does not require multiple processors nor does it imply simultaneous execution. Such models may be prone to problems such as deadlocks.

2.4.2 Parallel Computing

A parallel computing model is one where multiple operation streams progress simultaneously. This requires multiple processors and simultaneous execution.

2.4.3 Distributed Computing

A distributed computing model is one where multiple operation streams progress simultaneously on different machines. This includes parallel computing clusters and distributed concurrent systems.

2.5 Parallelisation

Parallelisation is the process of converting a sequential program into a parallel program, that can run on parallel hardware. Doing this effectively requires programmers to use a fundamentally different algorithm than the one used on sequential hardware, that is expressed in a manner that makes parallelisation more explicit or easier.

In some cases, we can exploit parallelism without changing the algorithm, by analysing which computational steps performed in the algorithm can be executed in parallel. This is known as exploiting **inherent parallelism**.

2.5.1 Parallelisation in Functional and Imperative Programming

- **Functional Programming**: Pure functional programming languages express computation via function evaluation. As function evaluation does not produce side effects, functions that do not depend on the result of another can be executed in parallel.
- **Imperative Programming**: Imperative programming languages express computation via a sequence of statements. Parallelisation in imperative programming is more difficult as statements may have side effects that depend on the order of execution.

2.6 Safe Parallelisation

The process of parallelisation must be done in a manner that ensures the same result is produced as the sequential program. Two dependencies must be preserved to ensure safe parallelisation:

- Control dependencies
- Data dependencies

2.6.1 Control Dependences

A control dependency is a dependency where one statement depends on whether another statement is executed. Some examples of control dependencies are shown below:

```
int gcd(int a, int b) {
    // if statement
    if (b == 0) {
        return a;              // Control dependency
    } else {
        return gcd(b, a % b); // Control dependency
    }
}

int gcd(int a, int b) {
    // while loop
    while (b != 0) {
        int temp = b; // Control dependency
        b = a % b;    // Control dependency
        a = temp;     // Control dependency
    }
    return a;
}

int gcd(int a, int b) {
    // error handling
    if (a < 0 || b < 0) {
        return -1;
    }
```

```
    ... // Control dependency
}
```

Control dependencies can be difficult to determine when the program contains many conditional statements as the control flow depends on input data.

2.6.2 Data Dependencies

A data dependency is a dependency where one statement depends on the same data as another statement. Data dependencies can be further classified into:

- **True Dependencies (W to R)**: Where a statement depends on the result of a previous instruction.

  ```
  a = 1;
  b = a; // True dependency on statement 1
  c = b; // True dependency on statement 1 and 2
  ```

- **Anti-Dependence (R to W)**: Where a statement requires a value that is later written to.

  ```
  a = 1;
  b = a; // Anti-dependency on statement 3
  a = 2;
  ```

- **Output Dependence (W to W)**: Where a variable depends on the ordering of another write statement.

  ```
  // Output dependency between statements 1 and 3
  a = 1;
  b = a;
  a = 2;
  ```

- **Input Dependence (R to R)**: Where a statement reads

  ```
  a = 1;
  b = a; // Input dependence on statement 3
  c = a; // Input dependence on statement 2
  ```

  Note that the order of input dependencies do not need to be preserved, as they do not affect the result of the program.

Data dependencies can be more challenging to determine when a program introduces pointers or references to objects, as such aliases may inadvertently introduce dependencies between statements. For example:

```
T a = T::new();
T *b = &a;       // b is a reference to a

a.mutator_method();  // changes a
b->mutator_method(); // also changes a
```

In this example, some programming languages may not explicitly show the true dependency of the final statement.

2.6.3 Dependency Analysis

Dependency analysis allows us to determine whether it is safe to reorder or parallelise statements in a program. For example, given the following code:

```
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        a[i][j + 1] = a[n][j];
    }
}
```

we may wish to know whether there are any data dependencies between loop iterations. That is, does there exist an iteration (i_r, j_r) that reads the same array element that is written to by iteration (i_w, j_w)? Mathematically,

$$\begin{aligned}\exists i_r, j_r, i_w, j_w : 0 \leqslant i_r < n \land 0 \leqslant j_r < n \land \\ 0 \leqslant i_w < n \land 0 \leqslant j_w < n \land \\ i_w = n \land j_w + 1 = j_r.\end{aligned}$$

Another way to analyse a loop would be to draw an iteration space diagram, with one axis representing the iteration number and the other representing the array index. This allows us to visualise the dependencies between iterations.

Unfortunately, any form of static dependency analysis is inexact in general.

- Alias analysis is undecidable
- Data dependence analysis are undecidable

When in doubt, we must assume that a dependency exists, and therefore cannot parallelise the code.

2.7 Automatic Parallelisation

Automatic parallelisation can be performed either

- Automatically by the compiler, or
- Manually by the programmer.

Current compilers are not smart enough to perform parallelisation in general as they are necessarily conservative in their analysis. On the other hand, manual parallelisation requires competence and can be very time-consuming and error-prone.

2.8 Parallelisation Process

Generally, we can break the process of parallelisation into three steps:

1. Determine what can be safely parallelised. This may involve:
    - Analysing control and data dependencies.

- Transforming the program to expose parallelism.
- Employing a more efficient algorithm that is better suited for parallelism.

2. Decide if parallelism will increase performance:
   - Is a significant amount of time spent in that code?
   - Is there a overhead associated with creating and managing threads?
   - Will the latency and/or throughput of communication between processors be a bottleneck?
   - Does parallelisation actually result in a speedup?
3. Transform the program into an explicitly parallel form:
   - Use programming language constructs to map computation (and possibly data) to processors and appropriate synchronisation.

2.9 Parallelism Granularity

While a potential to parallelise code may exist, executing this code has an associated overhead due to the costs of creating and managing threads, the synchronisation of parallel computations, and the latency of messages sent between processors. Therefore the amount of computation in each "work unit" must be sufficiently large to offset this overhead.

Here we can introduce the concept of **parallelism granularity**, which is the size of the work unit that is executed in parallel. Generally,

- **Coarse-grained parallelism** has large work units that are executed in parallel. Distributed systems and clusters are suited for coarse-grained parallelism.
- **Fine-grained parallelism** has small work units that are executed in parallel. CPUs with vector computation units are suited for fine-grained parallelism.

2.10 Speedup

The speedup of a parallel program is the ratio of the time taken to execute the best sequential program to the time taken to execute the parallel program:

$$S = \frac{\text{execution time of best sequential program}}{\text{execution time of parallel program}} \tag{1}$$

Note that the best sequential program is not the same as the parallel program restricted to a single processor, due to the aforementioned overheads. Rather, it is the best possible sequential program that can be written for the given problem, which may use different algorithms.

2.10.1 Speedup Curves

Commonly, we will plot the speedup of a parallel program against the number of processors used. The resulting curves can be classified into five categories:

- **Super-Linear Speedup**: The speedup increases faster than the number of processors used. This typically only occurs in extreme cases where the parallel program runs on different hardware that allows it to outperform the sequential program.

- **Linear Speedup**: The speedup increases linearly with the number of processors used. This is the ideal case, but is rarely achieved due to overheads associated with parallelism.
- **Sub-Linear Speedup**: The speedup increases slower than the number of processors used. This is the most common case, and often the speedup plateaus as the number of processors increases.
- **No Speedup**: The speedup remains constant as the number of processors used increases. This indicates that the program likely is not parallelisable.
- **Slowdown**: The speedup decreases below 1 as the number of processors used increases. This indicates that the overhead of parallelism is greater than the benefits of parallelism.

2.10.2 Scalable Parallelism

Typically an increase in problem size results in an increase in total execution time. In such cases, we want large problems to provide more potential for parallelism. A problem is said to be **scalable** if the speedup of the parallel program increases with the problem size, without plateauing.

2.10.3 Computational Complexity

While we may be able to achieve scalable parallelism, it is important to note that the computational complexity of the problem does not change. This is because the computational complexity of a problem is determined by the size of the problem, and distributing the problem across finitely many processors does not change the limiting behaviour of the problem.

$$\mathcal{O}\left(n \log\left(n\right)/c\right) = \mathcal{O}\left(n \log\left(n\right)\right)$$

(n is the size of the problem and c is the number of processors used).

2.10.4 Amdahl's Law

Amdahl's Law states that the speedup of a parallel program is limited by the fraction of the program that cannot be parallelised. If p is the fraction of the program that can be parallelised, then the theoretical speedup S of the execution of the program is given by:

$$S = \frac{1}{(1-p) + p/s} \tag{2}$$

where s is the speedup of the parallelisable portion of the program.

2.11 Parallelisation Methodology

When we want to parallelise a program, we can take the following steps:

1. Obtain representative and realistic data sets.
2. Time and profile the sequential version of the program.
3. View the source code and understand the high-level structure of the program.

4. Analyse dependencies.
5. Determine sections that can be parallelised.
6. Decide what parallelism might be worth exploiting.
7. Consider restructuring the program or replacing algorithms to expose more parallelism.
8. Transform the program into an explicitly parallel form.
9. Test and debug the parallel version of the program.
10. Time and profile the parallel version of the program.
11. Determine issues inhibiting greater performance.

2.11.1 Timing and Profiling

When timing an application, we must:

- Time a variety of data sets.
- Time multiple times to get an average.
- Be aware of other applications running in the background.
- Compare performance under identical conditions.
- Eliminate any unnecessary screen I/O by redirecting outputs to a file.
- Use high resolution timers.
- Distinguish between timing the entire program and timing specific sections.
- Distinguish between real time (wall clock time) and CPU time (time actually spent executing instructions in the program).
- Be aware of one-off costs (e.g., just-in-time compilation/dynamic library loading).

Profiling is the process of analysing the performance of a program by measuring the time spent in each section of the program. Profiling can capture time spent executing a section of code and the number of times that section is executed. Profiling can be done through sampling where the program counter is probed at regular intervals, or through instrumentation where the program is modified to record the time spent in each section. It is important to note that profiling can introduce overheads that may affect the performance of the program. Additionally, some compilers may allow the programmer to optimise a program aggressively so that the profiled program may not be representative of the actual program (i.e., sections may be optimised out or reordered).

2.12 Premature Optimisation

Roughly 80% of effects come from 20% of causes.

Pareto Principle

"There is no doubt that the grail of efficiency leads to abuse. Programmers waste enormous amounts of time thinking about, or worrying about, the speed of noncritical parts of their programs, and these attempts at efficiency actually have a strong negative impact when debugging and maintenance are considered. We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil.

Yet we should not pass up our opportunities in that critical 3%. A good programmer will not be lulled into complacency by such reasoning, he will be wise to look carefully at the critical code; but only after that code has been identified. It is often a mistake to make a priori judgments about what parts of a program are really critical, since the universal experience of programmers who have been using measurement tools has been that their intuitive guesses failed."

Donald Knuth