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

2.13 Explicit Parallelism

To transform a program into an explicitly parallel form, we must map computation and data to processors and synchronise processors at appropriate times. The exact mechanism for this depends on the programming model or parallelisation framework.

2.13.1 Parallel Programming Models

There are two ways of abstracting the underlying hardware that is used in parallel computing:

- **Communicating Sequential Processes (CSP)**
  - Processes operate independently and interact with each other solely through message-passing.
  - Both computation and data to processors.
  - Used by most distributed memory machines.
- **Parallel Random Access Machine (PRAM)**
  - Processes operate asynchronously and have constant-time access to shared memory.
  - Only computation needs to be mapped to processors.
  - Used for thread level parallelism.

2.14 Mapping Computation to Processors

When mapping threads to processors, they can either be created explicitly or implicitly:

- Explicit threads
  - Created by the programmer in code using a function call.
  - Usually associated with a function that is executed when the thread is first started.

  - Threads are explicitly started, stopped, or paused.

- Implicit threads
  - Created by a high-level library that builds on top of primitive threading libraries.
  - The programmer must only indicate what calculations can be performed in parallel.
  - Thread creation, management, and synchronisation is handled by the library.

2.14.1 Thread Scheduling

Threads are mapped to processors by the **scheduler** which is part of the operating system. This allows different threads to be mapped to different processors, and allows situations with more threads than processors to be handled through interleaving. Threads can be assigned a relative **priority** which the scheduler uses to determine which thread to execute next. Threads can also be assigned a **processor affinity** to ensure that they are always executed on the same processor. This can be useful when we want to reuse local caches.

There are several scheduling algorithms that the operating system can use to determine which thread to execute next:

- **Pre-emptive**: The scheduler can interrupt a thread that is currently executing to run another thread of a higher priority.
- **Cooperative**: The scheduler relies on threads to voluntarily yield control to other threads.
- **Round Robin**: Each thread is given a fixed time slice to execute before the next thread is executed.
- **First-Come, First-Served (FCFS)**: Threads are executed in the order they are created.
- **Shortest Job First (SJF)**: Threads are executed in order of the time they are expected to take to complete.

2.14.2 Thread Pools

As creating new threads is an expensive operation, we can instead create a pool of threads at the start of a program to avoid needing to create and destroy threads each time we want to parallelise a section of code. When threads are needed, they are taken from the pool, and when they are done, they are returned to the pool. New threads can also be added to the pool when necessary. Threaded applications can either use thread pools explicitly or use a high-level library that implements thread pools internally.

2.14.3 Load Balancing

When mapping work to processors, it is important to balance the work done by each processor. This can be done using

- **Dynamic work allocation**: where work is overdecomposed into many work units, typically far more than the number of processors. This is good if execution times are unpredictable or if the work is not evenly distributed, but can have larger overheads. Another approach can

involve creating work pools where we have a queue of tasks awaiting execution. Here work can be centralised or decentralised and work can be stolen from other processors if a processor runs out of work.

- **Static work allocation**: where work is divided into equal-sized chunks and assigned to each processor. This is good if execution times are predictable and work is evenly distributed, but can lead to processors idling if work is not evenly distributed. Work can be divided:
  - **By blocks**: which offer good locality but may suffer from poor load balance if work is not evenly distributed.
  - **Cyclically**: where work is assigned in a round-robin fashion. This can be good if work is evenly distributed, but can lead to poor locality.

3 Synchronisation

Some parallel programs require threads to be executed in a particular order to ensure access to shared memory is correct.

3.1 Race Conditions

Consider the problem of adding the elements in an array.

```
#define N 100

int array[N] = {0};
int sum = 0;

for (int i = 0; i < NUM_THREADS; i++) {
    thread_create(add, i);
}

void add(int thread_id)
{
    int block_size = ARRAY_SIZE / NUM_THREADS;
    int start = thread_id * block_size;

    if (start + block_size < N)
    {
        for (int i = start; i < start + block_size; i++)
        {
            sum += array[i];
        }
    }
    else
    {
        for (int i = start; i < N; i++)
        {
```

```
            sum += array[i];
        }
    }
}
```

Assume at some point in time, the value of `sum` is s, and let two threads be scheduled to execute the `add` function at indices $i_1 = 20$ and $i_2 = 30$, where `array` has the values 20 and 30. Now note that the compound operation being performed within the `add` function is not atomic, and will be expanded to the following operations for thread 1:

1. Read `sum`.
2. Read `array` at index i_1.
3. Add `sum` and `array[i1]`
4. Write result to `sum`.

and similarly for thread 2:

1. Read `sum`.
2. Read `array[i2]`
3. Add `sum` and `array[i2]`
4. Write result to `sum`.

Due to the non-deterministic nature of context switching, it is possible, while unlikely, for a context switch to occur from thread 1 to thread 2, after Step 2. This results in the final write to `sum` being scheduled after thread 2 has finished executing it's operations:

1. Read `sum` (s).

2. Read `array[i1]` (20).

1. Read `sum` (s).

2. Read `array[i2]` (30).

3. Add `sum` and `array[i2]` ($s + 30$).

4. Write result to `sum` ($s_2 \leftarrow s + 30$).

3. Add `sum` and `array[i1]` ($s + 20$).

4. Write result to `sum` ($s_1 \leftarrow s + 20$).

Observe how this fails to accumulate `array[i2]` into `sum` as the value of `sum` used by thread 1 is outdated. This problem of synchronisation is known as a **race condition**, where the program relies on the non-deterministic scheduling and execution of threads.

Note that non-deterministic behaviour is not always negative. In some cases, it is acceptable for there to be more than one equally correct result, or for there to be non-deterministic components that lead to the same final result. However, it should be noted that non-determinism can suggest the existence of a bug in a program, as such behaviour is often very difficult to debug.

3.2 Mutual Exclusion

Consider a program that has a segment of code that must be executed by several threads, called a **critical section**, in which threads may access and update shared memory. To prevent inconsistent accesses to this shared memory, we can enforce that only one thread may be inside the critical section at any time. This splits the program in the following manner:

- **Entry protocol**: This protocol checks if any other thread is currently in the critical section. If not, the thread is allowed to enter the critical section.
- **Critical section**: This section contains code that accesses shared memory.
- **Exit protocol**: This protocol signals that the thread has finished executing the critical section.
- **Remainder section**: This section contains code that is executed after the critical section, and does not access shared memory.

In this model, the following properties must also be satisfied:

- **Mutual exclusion** (safety): Only one thread may be in the critical section at any time.
- **Progress** (liveness): If no thread is in the critical section, and a thread wishes to enter the critical section, then the thread will eventually be allowed to enter the critical section.
- **Bounded waiting** (fairness): If a thread wishes to enter the critical section, then there is a bound on the number of times other threads may enter the critical section before the thread is allowed to enter.

It is important to maintain a queue of waiting threads in a way that ensures fairness and prevents starvation, where a thread is never allowed to enter the critical section.

The following sections discuss various mechanisms that can be used to enforce mutual exclusion.

3.2.1 Mutex Lock

A **mutex** (short for mutual exclusion) is a binary lock that is either locked or unlocked. A thread that wishes to enter the critical section must first **acquire** the lock. If the lock is already held by another thread, the thread must wait until the lock is **released**. When the thread has finished executing the critical section, it must **release** the lock so that other threads may enter the critical section.

3.2.2 Semaphore

Semaphores are a generalisation of mutex locks that enforces mutual exclusion for shared memory that is accessed by multiple threads. A semaphore is an integer that represents the number of threads that may access a shared resource. A semaphore first calls the **wait** operation, which decrements the semaphore if the semaphore is greater than 0. Otherwise, the thread is blocked until the semaphore is equal to zero. This allows a thread to enter the critical section. The thread then calls the **signal** operation, which increments the semaphore, allowing one of the blocked threads to enter the critical section.

3.2.3 Monitors

A **monitor** is an abstract data type used in object-oriented programming to enforce mutual exclusion. A monitor encapsulates data with a set of methods that operate on that data, where each operation within the monitor is mutually exclusive. In addition to this, monitors provide **condition variables** that allow threads to wait for a certain condition to be met before proceeding. This allows a monitor to model more complex synchronisation patterns. Condition variables are equipped with a **wait** operation that blocks the thread until another thread signals the condition variable, and a **signal** operation that unblocks one of the waiting threads.

3.2.4 Dekker's Mutual Exclusion Algorithm

Dekker's algorithm is the first known correct solution to the mutual exclusion problem where threads communicate through shared memory. A C implementation of Dekker's algorithm is shown below:

```
// Only thread 0 can modify lock[0]
// Only thread 1 can modify lock[1]
bool lock[2] = {false, false}; // initially neither thread has the lock

// whose turn it is to enter the critical section
int turn = 0; // initial value is arbitrary

void T0()
{
    /* Entry protocol */
    lock[0] = true; // thread 0 wants to enter the critical section

    // thread 1 has already requested to enter the critical section
    while (lock[1] == true)
    {
        if (turn == 1) // thread 1 is already in the critical section
        {
            lock[0] = false;  // release the lock
            while (turn == 1) // wait until thread 1 has finished
                ;
            lock[0] = true;   // acquire the lock again
        }
    }

    /* Critical section */

    /* Exit protocol */
    turn = 1;        // give the turn to thread 1
    lock[0] = false; // release the lock

    /* Remainder section */
}
```

```
void T1()
{
    /* Entry protocol */
    lock[1] = true; // thread 1 wants to enter the critical section

    // thread 0 has already requested to enter the critical section
    while (lock[0] == true)
    {
        if (turn == 0) // thread 0 is already in the critical section
        {
            lock[1] = false;  // release the lock
            while (turn == 0) // wait until thread 0 has finished
                ;
            lock[1] = true;   // acquire the lock again
        }
    }

    /* Critical section */

    /* Exit protocol */
    turn = 0;         // give the turn to thread 0
    lock[1] = false; // release the lock

    /* Remainder section */
}
```

This algorithm guarantees mutual exclusion and prevents deadlocks and starvation.

3.3 Memory Consistency Models

A memory consistency model defines a set of rules that the hardware follows when operating on memory. These rules allow the programmer to make assumptions about the order in which memory operations are executed. A strong memory consistency model (**sequential consistency**) is one where:

- All memory operations are executed in the order they are specified in the program.
- Memory operations within a single thread are executed in the order they are specified in the program.

Here we cannot assume that operations from different threads will be executed in the relative order they are specified in the program. If we want the hardware to execute these operations efficiently, we may choose to relax some of these rules. An example of this may be to allow program execution to continue while a write operation is being requested. However, this can violate mutual exclusion if this write operation was to acquire a lock, and two competing threads try to enter their respective critical sections. Consider a simplified mutual exclusion model:

```
void T0()
{
    /* Entry protocol */
    lock[0] = true; // acquire a lock but don't wait for write

    while (lock[1] != true)
        ;

    /* Critical section */

    /* Exit protocol */
    lock[0] = false;
}

void T1()
{
    /* Entry protocol */
    lock[1] = true; // acquire a lock but don't wait for write

    while (lock[0] != true)
        ;

    /* Critical section */

    /* Exit protocol */
    lock[1] = false;
}
```

By allowing both threads to execute the condition in the while loop before the lock has been acquired, we allow both threads to enter the critical section, violating mutual exclusion.

3.3.1 Atomic Hardware Operations

To promote safety, hardware can support atomic operations that perform writes, comparisons, swaps, etc., to ensure that mutual exclusion is satisfiable on relaxed memory consistency models. An example of an atomic hardware operation is shown below:

```
while (!test_and_set(lock))
    ;

/* Critical section */
lock = false;

// requires hardware support to ensure atomic execution
bool test_and_set(bool lock)
{
    if (!lock)
```

```
        lock = true;

    return lock;
}
```

3.3.2 Active and Passive Waiting

When acquiring a lock, a thread could wait for another thread to release a lock in a number of different ways:

- **Busy wait**: The thread continuously checks if the lock has been released. This leads to wasted CPU cycles.

  ```
  while (!test_and_set(lock))
      ; // spinlock

  /* Critical section */
  lock = false;
  ```

- **Sleeping**: The thread sleeps for a certain amount of time before checking if the lock has been released. This can lead to a delay in the thread entering the critical section and also requires choosing an appropriate sleep duration.

  ```
  while (!test_and_set(lock))
      sleep(time);

  /* Critical section */
  lock = false;
  ```

- **Suspending**: The thread is suspended until the thread with the lock releases the lock. This can be done using condition variables and can allow multiple threads to wait for a lock to be released.

  ```
  while (!test_and_set(lock))
      event.wait(); // Suspend current thread

  /* Critical section */
  lock = false;
  event.signal(); // Signal other threads
  ```

It is crucial to ensure that a thread releases a lock after it has finished executing it's critical section to prevent threads from blocking other threads, especially if those threads can raise errors while executing their critical section.

3.4 Deadlocks

A **deadlock** is a situation where two or more threads are waiting for each other to release a lock, preventing any of the threads from progressing. Deadlocks can occur if four conditions hold simultaneously:

- **Mutual exclusion**: Only one thread may hold a lock at any time.
- **Hold and wait**: A thread may hold a lock while waiting for another lock.
- **No preemption**: A lock may only be released voluntarily.
- **Circular wait**: A cycle of threads waiting for locks exists. For example, thread 0 is waiting for a lock held by thread 1, thread 1 is waiting for a lock held by thread 2, and thread 2 is waiting for a lock held by thread 0.

It is important to be mindful of deadlocks when working with more than one lock, as they prevent the program from progressing. Note that it may be possible to prevent deadlocks by ensuring that all locks are available before a thread enters the critical section, however, this requires cooperation between threads and may not always be possible. Therefore, locks are said to be **non-composable** as they break abstraction by exposing the internal state of the program, preventing software components from being developed independently.

3.4.1 Livelocks and Starvation

Livelocks are similar situtations to deadlocks, where instead of being completely blocked, threads are able to perform work, but unable to make progress. **Starvation** is a situation where a thread is unable to enter the critical section because it is never scheduled to run.

3.5 Priority Inversion

Priority inversion is a situation where a low-priority thread prevents a high-priority thread from executing. Consider the following scenario:

1. A low priority thread acquires a lock for a resource.
2. A high priority thread waits for the lock to be released.
3. A medium priority thread that does not require the resource is scheduled to run before the low priority thread as it is a higher priority.

This situation results in the high priority thread being blocked by both the low priority thread and the medium priority thread, resulting in a priority inversion.

3.6 Thread Safe Libraries

When calling library functions from different threads, it is important to ensure that such functions are thread safe, that is, they can be called from different threads without the risk of incorrect behaviour. An example of this is a random number generator that relies on a global seed—it is important to ensure that this seed is correctly updated when a random number is generated concurrently. A function can be thread-safe if:

- it does not share state across concurrent invocations, or if
- it uses mutex locks to protect access to shared state

3.7 Barrier Synchronisation

A **barrier** is a synchronisation mechanism that allows threads to wait for each other to reach a certain point in the program before continuing. Most parallel programming frameworks provide automatic barrier synchronisation after a parallel section of code.

3.8 Ad hoc Synchronisation

Some operations may require ordering constraints between threads that must be satisfied for the program to execute correctly. This can be achieved using a condition variable that signals when a certain operation has been completed. An example of this is shown below:

```
// Thread 0
operation_A();
A.signal(); // Signal that operation A has been completed
operation_B();

// Thread 1
operation_C();
A.wait(); // Wait for operation A to complete
operation_D(); // Relies on operation A
```

In this example, threads 0 and 1 can be executed independently, but operation D must be executed after operation A has been completed.

4 Locality

4.1 Array Layout

We can define static arrays in C using the following syntax:

```
int array[N];   // array of N integers
int array[N][M]; // 2D array of N rows and M columns
```

In most languages, 2D arrays are stored in row-major order, where the elements of each row are stored contiguously in memory, one after the other. This allows for efficient access to elements in the same row as they are stored in adjacent memory locations. In C, we can access the element at row i and column j of a 2D array using the following syntax:

```
array[i][j] // *(((int *) array) + i * M + j)
```

where `M` is the number of columns in the array. If the length of an array is not known at compile time, we can create a dynamic array using a heap allocation:

```
int *array = (int *) malloc(sizeof(int) * N);
```

This lets us allocate an array of N integers on the heap. We can access the element at index i of the array using the same syntax as before. Furthermore, we can construct a multi-dimensional array by allocating memory for each row of the array separately:

```
int **array = (int **) malloc(sizeof(int *) * N);
for (int i = 0; i < N; i++)
{
    array[i] = (int *) malloc(sizeof(int) * M);
}
```

This allows us to access the element at row i and column j of the array using the same syntax as before, as the array is defined as a pointer to a pointer to an integer. If we wanted all elements of the 2d array to be stored contiguously in memory, we would need to flatten the array and allocate memory for all elements at once:

```
int *array = (int *) malloc(sizeof(int) * N * M);
```

Note that we can no longer access the element at row i and column j of the array using the same syntax as before, as the array is defined as a pointer to an integer. Instead, we must use:

```
array[i * M + j] // *(array + i * M + j)
```

One advantage of creating a flattened array is that it allows for better cache locality, as all elements are stored contiguously in memory. On the contrary, multi-dimensional arrays allow arrays to be jagged, where rows can have a different number of columns.

4.2 Cache Locality

Locality of reference is the principle that memory locations that have been recently accessed are likely to be accessed again in the near future. There are two types of locality:

- **Temporal locality**: If a memory location is accessed, it is likely to be accessed again in the near future.
- **Spatial locality**: If a memory location is accessed, memory locations near it are likely to be accessed in the near future.

When data is loaded into the cache, it is loaded in blocks of memory called **cache lines**, typically 128 bytes in size, allowing adjacent memory locations to be accessed quickly. If we consider a simple matrix multiplication algorithm:

```
for (int i = 0; i < N; i++)
{
    for (int j = 0; j < N; j++)
    {
        for (int k = 0; k < N; k++)
        {
            C[i][j] = C[i][j] + A[i][k] * B[k][j];
        }
    }
}
```

As arrays are stored in row-major order, we have good temporal locality when accessing elements of `C`, given that the indices `i` and `j` do not change between iterations of the innermost loop. Additionally, we have good spatial locality when accessing elements of `A`, as the innermost loop accesses elements of `A` in a contiguous manner. However, we have poor locality when accessing elements of `B`, as the innermost loop accesses elements of `B` in a non-contiguous manner.

4.3 False Sharing

False sharing is a situation where two threads access different data that belongs to the same cache line. If we consider the above matrix multiplication example with two threads modifying disjoint columns of `C`, the first thread may load a cache line corresponding to the first column of `C`, which may contain elements that are accessed by the second thread, even though it is not accessing those elements. Then, as the second thread modifies elements of those other columns, the cache line from the first thread is invalidated as it is shared with another thread. To avoid this problem, we can align these memory accesses so that they coincide with cache lines, ensuring that each thread accesses its own cache line.

4.4 Eliminating "False" Dependencies

In some cases, data dependencies may be eliminated by introducing new variables without changing the outcome of a program. Consider the following examples:

- **Flow Dependence**:

  ```
  f = 42;
  ...
  a = f + 19;
  ```

  The order of the statements in this example is important, and cannot be changed. This is a true dependence.

- **Anti Dependence**:

  ```
  b = f + 42;
  ...
  f = 42;

  // can be rewritten as
  b = f + 42;

  f2 = 42;
  ```

  Here any subsequent reads of `f` will read the value of `f2` instead of the original value of `f`, allowing the statements to be reordered.

- **Output Dependence**:

  ```
  f = 42;
  ...
  f = 19;

  // can be rewritten as
  f = 42;
  ...
  f2 = 19;
  ```

Here the value of `f` is not used after the second assignment, so the statements can be reordered.

We can also eliminate false dependencies in loops. For example, there is flow dependence in the following loop the first line to the second line:

```
int b;

for (int i = 0; i < N; i++)
{
    b = a[i] + d[i];
    c[i] = b * 2 + (b - 1);
}
```

Additionally, as `b` is shared between iterations of the loop, there is an anti-dependence of the second line between iterations of the loop. These dependencies can be eliminated by using the following techniques:

- **Thread Local Memory**: Each thread has its own copy of `b` that is not shared with other threads.

  ```
  for (int i = 0; i < N; i++)
  {
      int b = a[i] + d[i];
      c[i] = b * 2 + (b - 1);
  }
  ```

  We can similarly promote the scalar variable `b` to an array, however this may consume more memory.

- **Creating Copies of Read-Only Variables**: If a variable is read-only, we can create multiple copies of the data to be shared between threads.

4.5 Memory Access Optimisations

We can further optimise memory accesses in loops by:

- Introducing **padding** to ensure that data is aligned with cache lines. This may be done by increasing the size of the data to be a power of 2.

  ```
  int array[1000][1000];

  for (int i = 0; i < 1000; i++)
  {
      for (int j = 0; j < 1000; j++)
      {
          array[i][j] = 0;
      }
  }

  // can be transformed to
  ```

```
int array[1024][1024]; // extend the array dimensions to 1024

for (int i = 0; i < 1024; i++)
{
    for (int j = 0; j < 1024; j++)
    {
        array[i][j] = 0;
    }
}
```

- **Tiling** loops to improve spatial locality. This involves breaking the loop into smaller blocks that can be loaded into the cache.

```
for (int x = 0; x < N; x++)
{
    for (int y = 0; y < M; y++)
    {
        S1(x, y);
    }
}

// can be transformed to
for (int x = 0; x < N; x += 32)
{
    for (int y = 0; y < M; y += 32)
    {
        // loop over blocks of size 32x32
        for (int dx = x; dx < x + 32; dx++)
        {
            for (int dy = y; dy < y + 32; dy++)
            {
                S1(dx, dy);
            }
        }
    }
}
```

- **Interchanging** loops for coarse grain parallelism or better spatial locality.

```
for (int i = 0; i < N; i++) // spatially local loop
{
    for (int j = 0; j < M; j++) // parallelisable loop
    {
        S1(i, j);
    }
}

// can be transformed to
```

```
for (int j = 0; j < M; j++) // coarse grain parallelism
{
    for (int i = 0; i < N; i++) // better spatial locality
    {
        S1(i, j);
    }
}
```

Here the j loop may be parallelisable, in which case making it the outer loop can allow for coarse grain parallelism. Or, the i loop may be iterating over the rows of a matrix, allowing for better spatial locality.

- **Loop Fusion** to allow for better temporal locality.

```
for (int i = 0; i < N; i++)
{
    S1(i);
}

for (int i = 0; i < N; i++)
{
    S2(i); // may use data from S1
}

// can be transformed to
for (int i = 0; i < N; i++)
{
    S1(i);
    S2(i); // data access is localised from previous operation
}
```

- **Loop Distribution** to allow potential parallelism.

```
for (int i = 0; i < N; i++)
{
    S1(i); // can be parallelised
    S2(i); // cannot be parallelised
}

// can be transformed to
for (int i = 0; i < N; i++) // parallelisable loop
{
    S1(i);
}

for (int i = 0; i < N; i++) // non-parallelisable loop
{
    S2(i);
}
```

- **Loop Peeling** may be used to remove parts of a loop that are more expensive than others.

```
for (int i = 0; i < N; i++)
{
    S1(i); // S1(0) is more expensive than remaining iterations
           // or may be read by all subsequent iterations
}

// can be transformed to
S1(0); // extract the first iteration
for (int i = 1; i < N; i++) // may parallelise remaining iterations
{
    S1(i);
}
```

- **Loop Unrolling** may be used to remove loops entirely by manually unrolling them. This may be useful for small loops. Here we avoid the overhead of the loop counter and can more finely control which iterations are parallelised.

```
for (int i = 0; i < N; i++)
{
    S1(i);
}

// can be transformed to
S1(0);
S1(1);
...
S1(N - 1);
```

4.6 Barriers for Parallelisation Speedup

To overcome barries when parallelising code, we can consider the following:

- If the overhead of parallelisation is too high:
  - look for more coarse-grained parallelism
- If some threads are doing more work than others:
  - use dynamic work allocation to balance the work across cores
  - overdecompose the work into many work units by creating more threads than cores to ensure that all cores are kept busy
- If threads are waiting for each other:
  - lock at different granularities
- Apply loop transformations to expose parallelism
- Alter data structures to improve data locality

5 Graphics Processing Units

A graphics processing unit (GPU) is specialised hardware that is designed to perform parallel computations on extremely large amounts of data. GPUs were originally designed to accelerate the rendering of 3D graphics, but are increasingly used for general-purpose computing.

5.1 Graphics Rendering Pipeline

Consider a video game that renders a 4K image at 60 frames per second. This requires $3840 \times 2160 \times 60 = 497\,664\,000$ pixel updates per second. This massive workload is handled by the graphics rendering pipeline, a sequence of stages that transform abstract 3D data into a final 2D image displayed on the screen.

1. **Vertex Shader**: Vertices are individual points in 3D space that define the shapes of objects. The vertex shader processes each vertex independently, transforming them according to the camera's perspective and other transformations such as scaling or rotation.
2. **Shape Assembly**: After vertices are processed, they are assembled into geometric shapes (typically triangles), which represent the surface of 3D objects.
3. **Geometry Shader** (*optional*): Additional vertices may be added or modified based on the existing geometry to add details to the shape before it is rasterised.
4. **Rasterisation**: Geometric shapes are converted into pixels on a 2D grid. Each shape is broken into fragments that map onto the screen, allowing for pixel-level processing.
5. **Fragment Shader**: Each fragment is processed to determine the colour of each pixel. The fragment shader applies lighting, textures, and other effects to calculate the colour and intensity of each fragment.
6. **Tests and Blending**: Depth and stencil tests are performed to determine if a fragment should be visible or obscured. Blending operations can combine colours from multiple translucent layers to produce the final output rendered to the screen.

5.2 General Purpose GPUs

GPUs are massively parallel high-throughput processors that can perform simple operations on a large scale. Over the years, it has become easier to program GPUs for general-purpose computing, where we can create custom shaders to perform parallel computations on scientific data, machine learning models, and other applications. Due to this, we treat GPUs as co-processors to the CPU, where the CPU is the **host** and the GPU is the **device** which helps accelerate computations. This involves the following steps:

1. **Process Initialisation**: The host initialises the process and allocates memory for the data to be processed.
2. **Data Transfer**: The host transfers the data to the device for processing.
3. **Kernel Execution**: The host launches a kernel on the device, which is a function that is executed in parallel by multiple threads.

4. **Data Transfer**: The device transfers the processed data back to the host.

In this process, the host and device have separate memory spaces, and therefore any transfer of data between the host and device is a bottleneck in the process, due to the high latency and low bandwidth of the transfer bus. To mitigate this, we can limit the number of data transfers between the host and device by loading data on the device for as long as possible before transferring it back to the host. In some cases we can use middleware to manage the data transfer between the host and device, where we can assume the memory spaces are unified, however it is often more efficient to manage the data transfer manually.

5.3 GPU Architecture

GPUs are composed of multiple **streaming multiprocessors** (SMs), each of which contains many processing **cores**. A streaming multiprocessor is made up of multiple **warp schedulers** that schedule **warps** of threads to be executed on the cores. These warps are groups of 32 threads that all execute the same instruction in parallel. Each streaming multiprocessor also has its own **shared memory** that can be accessed by all threads in the warp, and a set of **registers** that are used to store temporary data. Each core is a simple functional unit that can perform floating-point or integer arithmetic operations which does not have it's own program counter. These cores are not as complex as those in a CPU as they are designed to execute the same instruction on many threads in parallel.

5.4 GPU Programming

When developing a GPU kernel, we must consider the following:

- **Threads**: As with CPUs, we can create threads on a GPU to perform computations in parallel. Due to the abundance of threads, we can create many more threads on a GPU than on a CPU without incurring a significant overhead.
- **Thread Blocks**: We can group threads into **thread blocks** which can be executed by a single streaming multiprocessor. These thread blocks typically contain 1024 threads.
- **Grids**: If we have multiple streaming multiprocessors, we can also group thread blocks into **grids** which can be executed in parallel.

When thread blocks are scheduled to run on the GPU, they are executed in groups of 32 threads[2] called **warps**. Each thread in a warp is executed in parallel on a streaming multiprocessor. An example of a simple GPU kernel to multiply two matrices is shown below:

```
void matrix_multiply(float* A, float* B, float* C, int N)
{
    for (int i = 0; i < N; i++)
    {
        for (int j = 0; j < N; j++)
        {
            float sum = 0.0f;
```

[2]This number may vary depending on the GPU architecture.

```
            for (int k = 0; k < N; k++)
            {
                sum += A[i * N + k] * B[k * N + j];
            }
            C[i * N + j] = sum;
        }
    }
}

__global__ void matrix_multiply_kernel(float* A, float* B, float* C, int N)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;

    if (i < N && j < N)
    {
        float sum = 0.0f;
        for (int k = 0; k < N; k++)
        {
            sum += A[i * N + k] * B[k * N + j];
        }
        C[i * N + j] = sum;
    }
}
```

In this example, the `blockIdx.x` and `blockDim.x` variables are used to determine the position of the thread block in the grid, and the size of the thread block, respectively. The `threadIdx.x` variable is used to determine the position of the thread within the thread block. The `__global__` keyword is used to indicate that the function is a GPU kernel that is designed to be run on the GPU, to be called from the host. The kernel function is then launched from the host using the following syntax:

```
dim2 grid_size(N / BLOCK_SIZE, N / BLOCK_SIZE);
dim2 block_size(BLOCK_SIZE, BLOCK_SIZE);
matrix_multiply_kernel<<<grid_size, block_size>>>(A, B, C, N);
```

where `grid_size` and `block_size` are `dim2` or `dim3` structures that define the size of the grid (number of blocks), and the size of the thread block, respectively.

Note that we loop over both the rows and columns of the matrix in the kernel, so that each thread only operates on the innermost loop of the algorithm. This lets us take advantage of the large number of cores available, and ensures each thread does very fine-grained work. It also allows for better memory access patterns and improves the performance of the kernel. The additional `if` statement ensures that if the number of threads is not a multiple of the number of elements in the matrix, the kernel will not access memory outside the bounds of the matrix.

In addition to the kernel, we must also allocate memory on the device to store the input and output matrices, and transfer the data back to the host after the kernel has finished executing.

5.4.1 Optimising Warp Scheduling

When writing GPU kernels, it is important to consider how warps are scheduled to run on the GPU. As all threads in a streaming multiprocessor are scheduled to run at the same time, it is important to ensure they can all execute. For example, in the following code, some threads will execute a different instruction to others, causing the warp to diverge:

```
if (threadIdx.x % 2 == 0)
{
    // do something
}
else
{
    // do something else
}
```

In this case, half of the threads in the warp will execute the first branch, and the other half will execute the second branch. Given the architecture of the GPU, the warp will execute each path sequentially, rather than in parallel, as it cannot tell half of the threads to execute branch 1 and the other half to execute branch 2. This can lead to a loss in performance, as the warp will take longer to execute the kernel. To avoid this, we should write kernel code to ensure that all threads perform SIMD operations instead.

5.5 Memory Access

As we saw in the above example, each thread in a warp has its own local memory that is stored in the register file, for example, the iterators `i`, `j`, and `k` in the matrix multiplication kernel. If we wish to share memory between threads in a warp, we can use the shared memory space available to each streaming multiprocessor. This acts as an L1 cache for the streaming multiprocessor, and is much faster than global memory. An example of using shared memory in a kernel is shown below:

```
__global__ void matrix_multiply_kernel(float* A, float* B, float* C, int N)
{
    __shared__ float shared_A[BLOCK_SIZE][BLOCK_SIZE];
    __shared__ float shared_B[BLOCK_SIZE][BLOCK_SIZE];

    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;

    float sum = 0.0f;
    for (int k = 0; k < N; k += BLOCK_SIZE)
    {
        shared_A[threadIdx.x][threadIdx.y] = A[i * N + k + threadIdx.y];
        shared_B[threadIdx.x][threadIdx.y] = B[(k + threadIdx.x) * N + j];

        __syncthreads();

        for (int kk = 0; kk < BLOCK_SIZE; kk++)
```

```
        {
            sum += shared_A[threadIdx.x][kk] * shared_B[kk][threadIdx.y];
        }

        __syncthreads();
    }

    C[i * N + j] = sum;
}
```

In this example, we use shared memory to store blocks of the input matrices `A` and `B` that are used by each thread in the thread block. We then use the `__syncthreads()` function to synchronise all threads in the thread block before proceeding to the next iteration of the loop. This ensures that all threads have finished writing to shared memory before any thread reads from it.

Additionally, if we want per-device shared memory access across multiple streaming multiprocessors, we can use the `__global__` keyword. This uses the GPU memory which is slower than accessing local registers and the shared memory cache. In the first example, this correspond to matrices `A`, `B`, and `C`. Finally, we can also access host memory from the device, however this is the slowest form of memory access and should be minimised where possible.

5.6 Synchronisation

Synchronisation is only allowed within blocks, and therefore blocks must be independent of each other. Any synchronisation between blocks is not possible, but there is however an implicit barrier between kernels, where the host waits for all blocks within a kernel to finish executing before proceeding to the next kernel. For example:

```
kernel1<<<grid_size, block_size>>>(...);
// implicit barrier
kernel2<<<grid_size, block_size>>>(...);
```

To synchronise threads within a block, we can use the `__syncthreads()` function, which ensures that all threads in the block have reached the same point in the kernel before proceeding. This is useful when using shared memory to ensure that all threads have finished writing to shared memory before any thread reads from it.

5.7 Thread and Block Heuristics

When writing GPU kernels, it is important to consider the following heuristics:

- When calling a kernel, we must consider the dimension and size of blocks per grid and the dimension and size of threads per block. The key consideration is to keep the entire GPU busy by choosing the optimal number of threads per block and blocks per grid.
- The multi-dimensional nature of these parameters allows easier mapping of multi-dimensional problems to CUDA and does not play any role in performance.
- Thread instructions are executed sequentially and as a result, the only way to hide latency is to execute multiple warps while one is busy.

- The number of threads per block should be a multiple of 32 so that all warps within that block use all cores within on that multiprocessor.
- the number of blocks in a grid should be larger than the number of multiprcessors so that all multiprocessors are kept busy.

6 Parallel Programming

6.1 Designing Parallel Algorithms

When designing parallel algorithms, we can take one of two approaches:

- **Option 1**:
  - Start with a sequential implementation
  - Largely preserve the structure of the sequential implementation
  - Incrementally introduce parallelism to parts of the code
- **Option 2**:
  - Analyse the high level parallel structure of the algorithm
  - Develop and test a skeleton of the parallel structure first
  - Use high-level parallel abstractions such as the consumer-producer pattern
  - Plug in the pre-tested sequential components into the parallel skeleton

The advantages of the second option is that we can test and debug the sequential components of the algorithm separately from the parallel structure.

6.2 Analysing Parallel Algorithms

We define **work** T_1 as the time taken to execute a program on a single processor, and **span** T_∞ as the time taken to execute a program on an infinite number of processors. If we draw our algorithm as a directed acyclic graph, the work is the sum of the weights of all nodes in the graph, and the span is the length of the longest path from the start node to the end node, also known as the **critical path**. We can also define the variable T_p as the time taken to execute a program on p processors.

Using these variables we can define the following relationships:

- Span law: $T_p \geqslant T_\infty$
- Work law: $T_p \geqslant T_1/p$
- Speedup: $T_1/T_p \leqslant p$
- Parallelism: T_1/T_∞

6.3 Parallel Design Patterns

6.3.1 Divide and Conquer

1. Divide the problem into two or more subproblems
2. Solve each subproblem recursively in parallel
3. Combine the results of the subproblems to solve the original problem

An example of this pattern is the quicksort algorithm, which has the following pseudocode:

```
// sort array A

partition A at position p
    such that A[i] <= A[j] for all i < p and j >= p

parallel quicksort(A[0:p])
parallel quicksort(A[p+1:n])

// A is now sorted
```

On average, if we assume the sequential paritioning of the array takes $\mathcal{O}(n)$ time, the work of the algorithm for a problem of size n is given by:

$$T_1(n) = 2T_1\left(\frac{n}{2}\right) + \mathcal{O}(n) \implies T_1(n) = \mathcal{O}(n \log n)$$

The span of the algorithm is given by:

$$T_\infty(n) = T_\infty\left(\frac{n}{2}\right) + \mathcal{O}(n) \implies T_\infty(n) = \mathcal{O}(n)$$

Therefore, the parallelism efficiency of the algorithm is given by:

$$T_1/T_\infty = \mathcal{O}(\log n)$$

6.3.2 Contraction

1. Reduce the problem into a smaller problem (reduction)
2. Solve the smaller problem (often recursively)
3. Use the results of the subproblems to solve the original problem (expansion)

An example of this pattern is the parallel prefix sum algorithm, which has the following pseudocode:

```
// compute the prefix sum of array A

parallel for i = 1 to n/2
    B[i] = B[2 * i] + A[2 * i + 1]

R = prefix_sum(B)
```

```
parallel for i = 1 to n/2
    sum[2 * i] = R[i] - A[2 * i + 1]
    sum[2 * i + 1] = R[i]

// sum is now the prefix sum of A
```

The work of the algorithm for a problem of size n is given by:

$$T_1(n) = T_1\left(\frac{n}{2}\right) + \mathcal{O}(n) \implies T_1(n) = \mathcal{O}(n)$$

The span of the algorithm is given by:

$$T_\infty(n) = T_\infty\left(\frac{n}{2}\right) + \mathcal{O}(1) \implies T_\infty(n) = \mathcal{O}(\log n)$$

Therefore, the parallelism efficiency of the algorithm is given by:

$$T_1/T_\infty = \mathcal{O}(n/\log n)$$

6.3.3 Partitioning

1. Partition the input data into k disjoint subsets
2. Solve each subset independently using a different algorithm
3. Combine the results of the subsets to solve the original problem

An example of this pattern is computing the sum of an array.

```
// compute the sum of array A

parallel for p = 0 to k
    partial_sum[p] = 0

    for i = p * n / k to min(n, (p + 1) * n / k - 1)
        partial_sum[p] += A[i]

for p = 0 to k
    sum += partial_sum[p]

// sum is now the sum of A
```

The work of the algorithm for a problem of size n is given by:

$$T_1(n) = n + k$$

The span of the algorithm is given by:

$$T_\infty(n) = n/k + k$$

Therefore, the parallelism efficiency of the algorithm is given by:

$$T_1/T_\infty = k$$

6.3.4 Pointer Jumping

1. Each node is replaced by its parent in a tree recursively

This pattern is used in very specific cases, such as when finding the root of a forest of trees.

```
for i = 0 to log(n)
    parallel foreach node p in tree
        if p.parent.parent != NULL
            p.parent = p.parent.parent
```

The work of the algorithm for a problem of size n is given by:

$$T_1(n) = \mathcal{O}(n \log n)$$

The span of the algorithm is given by:

$$T_\infty(n) = \mathcal{O}(\log n)$$

Therefore, the parallelism efficiency of the algorithm is given by:

$$T_1/T_\infty = \mathcal{O}(n)$$

6.4 Search Algorithms

Combinatorial search problems are a class of problems that involve searching through a large number of possible solutions to find one or more optimal (or suboptimal) solutions, defined in a problem space. These problems can be classified into two categories:

- **Decision Problems**: These problems involve finding a solution that satisfies a set of constraints. The answer to a decision problem is either "yes" or "no". For example, the travelling salesman problem involves finding the shortest path that visits a set of cities exactly once.
- **Optimisation Problems**: These problems involve finding the best solution that maximises or minimises an objective function. The answer to an optimisation problem is a value that represents the quality of the solution. For example, the knapsack problem involves finding the combination of items that maximises the total value while staying within a weight limit.

6.4.1 Search Trees

We can often depict search algorithms as a tree, where each node represents a state in the search space, and each edge represents a decision that leads to a new state. At each node, we can perform the following checks:

- Have we been in this state before?
  - try to avoid potential cycles
- Is this state a solution to the problem?
  - is the solution valid and better than the current best solution?
- If not, determine the next states to explore recursively

6.4.2 Exploring the Search Space

We can use several techniques to explore the search space:

- **Backtracking**: This is a depth-first search algorithm that explores the search space by recursively exploring all possible solutions. After searching all options at a node, it backtracks to the previous branching point and explores the next option.
- **Recursive Depth-First Search**: This is a depth-first search algorithm that explores the search space by recursively exploring all possible solutions. It is similar to backtracking, but does not backtrack to the previous branching point after exploring all options at a node, as they have already been explored. It is therefore one possible implementation of the backtracking algorithm.

6.4.3 Exploring the Search Space in Parallel

When exploring the search space in parallel, we must ensure that each thread explores a disjoint subset of the search space and that threads are load balanced. This can be achieved by using the following steps:

- Explore the first few levels of the search space sequentially.
- Assign a subset of the search space to each thread. Here we should consider over-decomposing the search space to ensure that all threads are kept busy.
- Use shared memory to inform threads of the best solution found so far. If the best solution has been found, all threads should terminate.

An alternative approach to depth-first search is to use a breadth-first search algorithm, which explores the search space level by level.

6.4.4 Pruning the Search Space

When exploring the search space, we can use pruning techniques to eliminate parts of the search space that are not worth exploring. If we already have a solution with quality q, and the best case quality of the current partial solution is p, where $p < q$, we do not need to explore the current partial solution further, as it will not lead to a better solution. Thus, we can prune the subtree beneath the current partial solution with quality p. Note that pruning is only possible is we can guarantee that the quality of the current partial solution will be worse than the best solution found so far. Pruning should not eliminate any potential solutions.

To be able to apply pruning, we need to determine the best solution early in the search process, so that we can apply pruning as soon as possible, that is, a solution with large quality. Therefore we want to use a heuristic that lets us explore the most promising parts of the search space first. This is known as **branch and bound**, where we prioritise the most promising parts of the search space to explore first in parallel. We do this by using a priority queue to store the partial solutions, where the priority is determined by the quality of the solution. We explore any low quality solutions after these high quality solutions, as we do not yet know if they are all worse than the best solution found so far.

On a distributed system, it is not worth using a shared global priority queue that is accessed by all systems, as this would lead to a bottleneck. Instead, it is m can use a local priority queue on each

system, and only communicate the best solution found so far to all systems. Each machine can apply periodic **work stealing** to ensure all systems are kept busy at all times and to find the best solution as quickly as possible, by sharing the best solution found so far with other systems.

Minimax is a strategy in decision-making and game theory that finds the optimal move by minimising the possible loss. In a two-player game, Player A (the maximiser) aims to achieve the best score while Player B (the minimiser) tries to reduce it. The minimax algorithm explores the game tree by alternating between moves that either maximize or minimise the score, depending on the player.

Alpha-Beta Pruning optimises minimax by skipping branches that won't affect the final decision, which speeds up the search. This pruning uses two values:

- **Alpha**: the minimum score the maximiser (Player A) can guarantee.
- **Beta**: the maximum score the minimiser (Player B) can secure.

If a move's potential outcome is worse than the current best, it stops further exploration of that branch. This strategy mirrors the principle of branch and bound, discarding paths that can't improve the solution, and allows a larger search space to be covered efficiently.

For the minimax algorithm to work effectively, the game tree must be sufficiently deep to allow for a meaningful exploration of the search space. In addition to this, the evaluation function used to determine the quality of a solution must be accurate and efficient. These algorithms are commonly used in games such as chess, where programs often outperform human players due to their ability to select the best move from a large number of possibilities.
