packt

1ST EDITION

Asynchronous Programming with C++

Build blazing-fast software with multithreading and asynchronous programming for ultimate efficiency

JAVIER REGUERA-SALGADO
JUAN ANTONIO RUFES

1
Parallel Programming Paradigms

Before we dive into **parallel programming** using C++, throughout the first two chapters, we will focus on acquiring some foundational knowledge about the different approaches to building parallel software and how the software interacts with the machine hardware.

In this chapter, we will introduce parallel programming and the different paradigms and models that we can use when developing efficient, responsive, and scalable concurrent and asynchronous software.

There are many ways to group concepts and methods when classifying the different approaches we can take to develop parallel software. As we are focusing on software built with C++ in this book, we can divide the different parallel programming paradigms as follows: concurrency, asynchronous programming, parallel programming, reactive programming, dataflows, multithreading programming, and event-driven programming.

Depending on the problem at hand, a specific paradigm could be more suitable than others to solve a given scenario. Understanding the different paradigms will help us to analyze the problem and narrow down the best solution possible.

In this chapter, we're going to cover the following main topics:

- What is parallel programming and why does it matter?
- What are the different parallel programming paradigms and why do we need to understand them?
- What will you learn in this book?

Technical requirements

No technical requirements apply for this chapter.

Throughout the book, we will develop different solutions using C++20 and, in some examples, C++23. Therefore, we will need to install GCC 14 and Clang 8.

All the code blocks shown in this book can be found in the following GitHub repository: `https://github.com/PacktPublishing/Asynchronous-Programming-with-CPP`.

Getting to know classifications, techniques, and models

Parallel computing occurs when tasks or computations are done simultaneously, with a task being a unit of execution or unit of work in a software application. As there are many ways to achieve parallelism, understanding the different approaches will be helpful to write efficient parallel algorithms. These approaches are described via paradigms and models.

But first, let us start by classifying the different parallel computing systems.

Systems classification and techniques

One of the earliest classifications of parallel computing systems was made by Michael J. Flynn in 1966. Flynn's taxonomy defines the following classification based on the **data streams** and number of instructions a parallel computing architecture can handle:

- **Single-instruction-single-data (SISD) systems**: Define a sequential program
- **Single-instruction-multiple-data (SIMD) systems**: Where operations are done over a large dataset, for example in signal processing of GPU computing
- **Multiple-instructions-single-data (MISD) systems**: Rarely used
- **Multiple-instructions-multiple-data (MIMD) systems**: The most common parallel architectures based in multicore and multi-processor computers

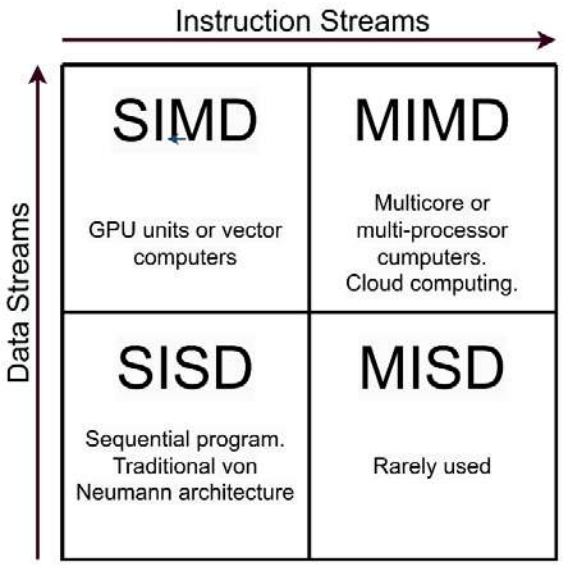

Figure 1.1: Flynn's taxonomy

This book is not only about building software with C++ but also about keeping an eye on how it interacts with the underlying hardware. A more interesting division or taxonomy can probably be done at the software level, where we can define the techniques. We will learn about these in the subsequent sections.

Data parallelism

Many different data units are processed in parallel by the same program or sequence of instructions running in different processing units such as CPU or GPU cores.

Data parallelism is achieved by how many disjoint datasets can be processed at the same time by the same operations. Large datasets can be divided into smaller and independent data chunks exploiting parallelism.

This technique is also highly scalable, as adding more processing units allows for the processing of a higher volume of data.

In this subset, we can include SIMD instruction sets such as SSE, AVX, VMX, or NEON, which are accessible via intrinsic functions in C++. Also, libraries such as OpenMP and CUDA for NVIDIA GPUs. Some examples of its usage can be found in machine learning training and image processing. This technique is related to the SIMD taxonomy defined by Flynn.

As usual, there are also some drawbacks. Data must be easily divisible into independent chunks. This data division and posterior merging also introduces some overhead that can reduce the benefits of parallelization.

Task parallelism

In computers where each CPU core runs different tasks using processes or threads, **task parallelism** can be achieved when these tasks simultaneously receive data, process it, and send back the results that they generate via message passing.

The advantage of task parallelism resides in the ability to design heterogeneous and granular tasks that can make better usage of processing resources, being more flexible when designing a solution with potentially higher speed-up.

Due to the possible dependencies between tasks that can be created by the data, as well as the different nature of each task, scheduling and coordination are more complex than with data parallelism. Also, task creation adds some processing overhead.

Here we can include Flynn's MISD and MIMD taxonomies. Some examples can be found in a web server request processing system or a user interface events handler.

Stream parallelism

A continuous sequence of data elements, also known as a **data stream**, can be processed concurrently by dividing the computation into various stages processing a subset of the data.

Stages can run concurrently. Some generate the input of other stages, building a **pipeline** from stage dependencies. A processing stage can send results to the next stage without waiting to receive the entire stream data.

Stream parallel techniques are effective when handling continuous data. They are also highly scalable, as they can be scaled by adding more processing units to handle the extra input data. Since the stream data is processed as it arrives, this means not needing to wait for the entire data stream to be sent, which means that memory usage is also reduced.

However, as usual, there are some drawbacks. These systems are more complex to implement due to their processing logic, error handling, and recovery. As we might also need to process the data stream in real time, the hardware could be a limitation as well.

Some examples of these systems include monitoring systems, sensor data processing, and audio and video streaming.

Implicit parallelism

In this case, the compiler, the runtime, or the hardware takes care of parallelizing the execution of the instructions transparently for the programmer.

This makes it easier to write parallel programs but limits the programmer's control over the strategies used, or even makes it more difficult to analyze performance or debugging.

Now that we have a better understanding of the different parallel systems and techniques, it's time to learn about the different models we can use when designing a parallel program.

Parallel programming models

A **parallel programming model** is a parallel computer's architecture used to express algorithms and build programs. The more generic the model the more valuable it becomes, as it can be used in a broader range of scenarios. In that sense, C++ implements a parallel model through a library within the **Standard Template Library** (**STL**), which can be used to achieve parallel execution of programs from sequential applications.

These models describe how the different tasks interact during the program's lifetime to achieve a result from input data. Their main differences are related to how the tasks interact with each other and how they process the incoming data.

Phase parallel

In **phase parallel**, also known as the agenda or loosely synchronous paradigm, multiple jobs or tasks perform independent computations in parallel. At some point, the program needs to perform a synchronous interaction operation using a barrier to synchronize the different processes. A barrier is a synchronization mechanism that ensures that a group of tasks reach a particular point in their

execution before any of them can proceed further. The next steps execute other asynchronous operations, and so on.

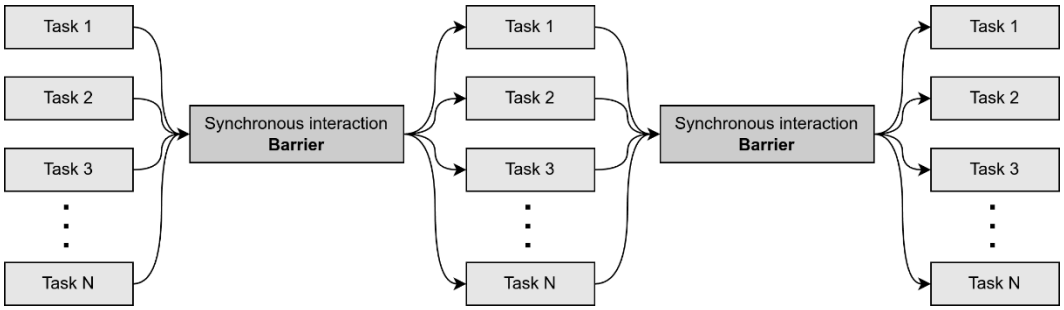

Figure 1.2: Phase parallel model

The advantage of this model is that the interaction between tasks does not overlap with computation. On the other hand, it is difficult to reach a balanced workload and throughput among all processing units.

Divide and conquer

The application using this model uses a main task or job that divides the workload among its children, assigning them to smaller tasks.

Child tasks compute the results in parallel and return them to the parent task, where the partial results are merged into the final one. Child tasks can also subdivide the assigned task into even smaller ones and create their own child tasks.

This model has the same disadvantage as the phase parallel model; it is difficult to achieve a good load balance.

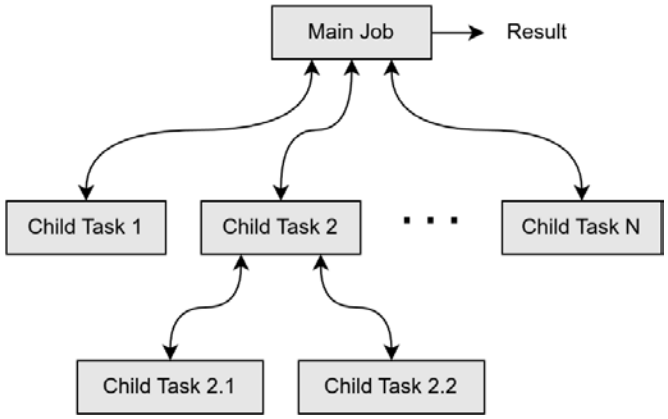

Figure 1.3: Divide and conquer model

In *Figure 1.3*, we can see how the main job divides the work among several child tasks, and how **Child Task 2**, in turn, subdivides its assigned work into two additional tasks.

Pipeline

Several tasks are interconnected, building a virtual pipeline. In this pipeline, the various stages can run simultaneously, overlapping their execution when fed with data.

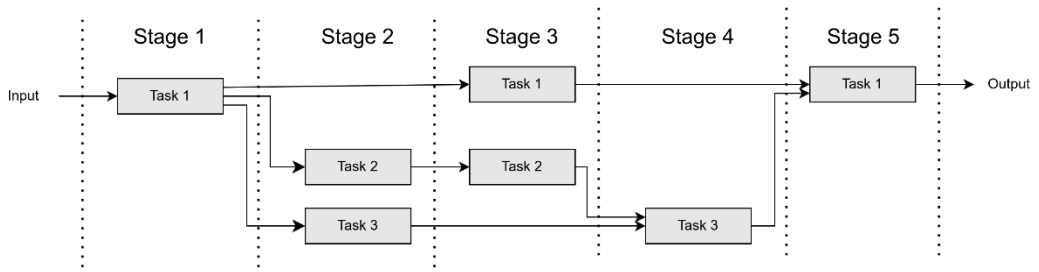

Figure 1.4: Pipeline model

In the preceding figure, three tasks interact in a pipeline composed of five stages. In each stage, some tasks are running, generating output results that are used by other tasks in the next stages.

Master-slave

Using the **master-slave model**, also known as **process farm**, a master job executes the sequential part of the algorithm and spawns and coordinates slave tasks that execute parallel operations in the workload. When a slave task finishes its computation, it informs the master job of the result, which might then send more data to the slave task to be processed.

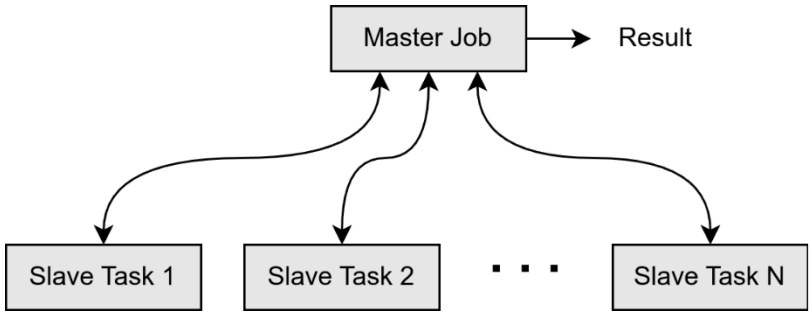

Figure 1.5: The master-slave model

The main disadvantage is that the master can become a bottleneck if it needs to deal with too many slaves or when tasks are too small. There is a tradeoff when selecting the amount of work to be performed by

each task, also known as **granularity**. When tasks are small, they are named fine-grained, and when they are large, they are coarse-grained.

Work pool

In the work pool model, a global structure holds a pool of work items to do. Then, the main program creates jobs that fetch pieces of work from the pool to execute them.

These jobs can generate more work units that are inserted into the work pool. The parallel program finishes its execution when all work units are completed and the pool is thus empty.

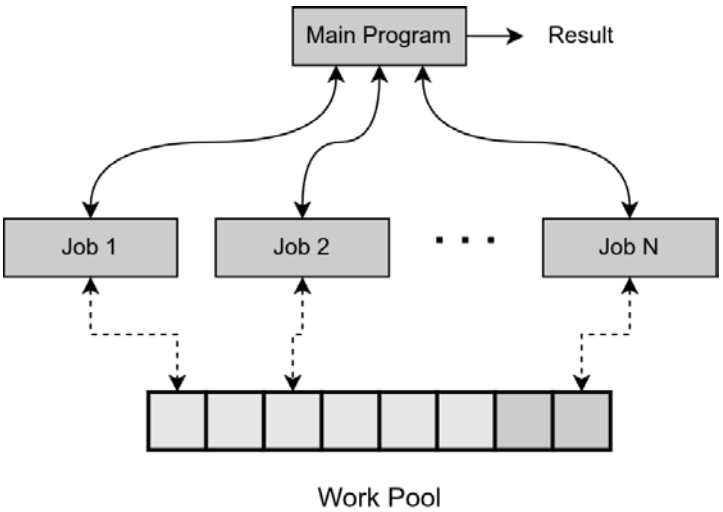

Figure 1.6: The work pool model

This mechanism facilitates load balancing among free processing units.

In C++, this pool is usually implemented by using an unordered set, a queue, or a priority queue. We will implement some examples in this book.

Now that we have learned about a variety of models that we can use to build a parallel system, let's explore the different parallel programming paradigms available to develop software that efficiently runs tasks in parallel.

Understanding various parallel programming paradigms

Now that we have explored some of the different models used for building parallel programs, it is time to move to a more abstract classification and learn about the fundamental styles or principles of how to code parallel programs by exploring the different parallel programming language paradigms.

Synchronous programming

A **synchronous programming** language is used to build programs where code is executed in a strict sequential order. While one instruction is being executed, the program remains blocked until the instruction finishes. In other words, there is no multitasking. This makes the code easier to understand and debug.

However, this behavior makes the program unresponsive to external events while it is blocked while running an instruction and difficult to scale.

This is the traditional paradigm used by most programming languages such as C, Python, or Java.

This paradigm is especially useful for reactive or embedded systems that need to respond in real time and in an ordered way to input events. The processing speed must match the one imposed by the environment with strict time bounds.

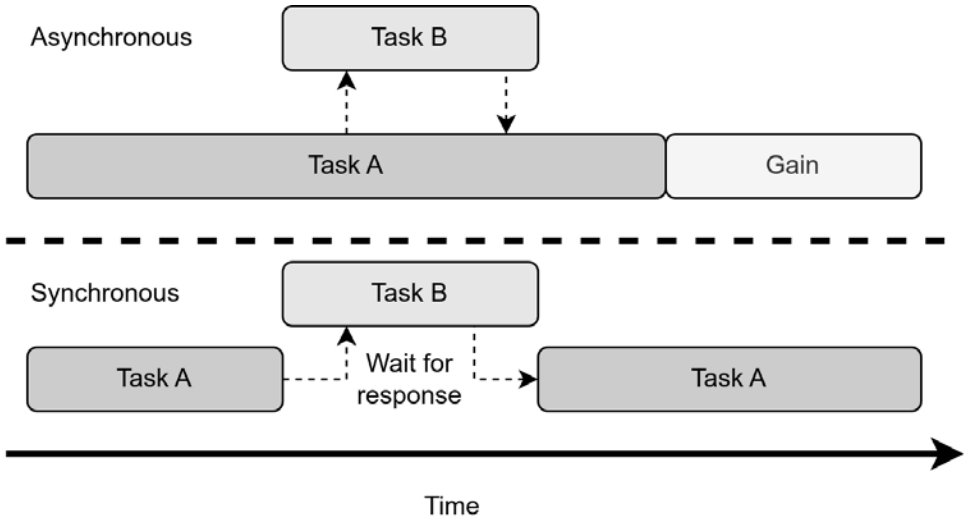

Figure 1.7: Asynchronous versus synchronous execution time

Figure 1.7 shows two tasks running in a system. In the synchronous system, task A is interrupted by task B and only resumes its execution once task B has finished its work. In the asynchronous system, tasks A and B can run simultaneously, thus completing both of their work in less time.

Concurrency programming

With **concurrency programming**, more than one task can run at the same time.

Tasks can run independently without waiting for other tasks' instructions to finish. They can also share resources and communicate with each other. Their instructions can run asynchronously, meaning

that they can be executed in any order without affecting the outcome, adding the potential for parallel processing. On the other hand, that makes this kind of program more difficult to understand and debug.

Concurrency increases the program throughput, as the number of tasks completed in a time interval increases with concurrency (see the formula for Gustafson's law in the section *Exploring the metrics to assess parallelism* at the end of this chapter). Also, it achieves better input and output responsiveness, as the program can perform other tasks during waiting periods.

The main problem in concurrent software is achieving correct concurrency control. Exceptional care must be taken when coordinating access to shared resources and ensuring that the correct sequence of interactions is taking place between the different computational executions. Incorrect decisions can lead to race conditions, deadlocks, or resource starvation, which are explained in depth in *Chapter 3* and *Chapter 4*. Most of these issues are solved by following a consistency or memory model, which defines rules on how and in which order operations should be performed when accessing the shared memory.

Designing efficient concurrent algorithms is done by finding techniques to coordinate tasks' execution, data exchange, memory allocations, and scheduling to minimize the response time and maximize throughput.

The first academic paper introducing concurrency, *Solution of a Problem in Concurrent Programming Control*, was published by Dijkstra in 1965. Mutual exclusion was also identified and solved there.

Concurrency can happen at the operating system level in a preemptive way, whereby the scheduler switches contexts (switching from one task to another) without interacting with the tasks. It can also happen in a non-preemptive or cooperative way, whereby the task yields control to the scheduler, which chooses another task to continue work.

The scheduler interrupts the running program by saving its state (memory and register contents), then loading the saved state of a resumed program and transferring control to it. This is called **context switching**. Depending on the priority of a task, the scheduler might allow a high-priority task to use more CPU time than a low-priority one.

Also, some special operating software such as memory protection might use special hardware to keep supervisory software undamaged by user-mode program errors.

This mechanism is not only used in single-core computers but also in multicore ones, allowing many more tasks to be executed than the number of available cores.

Preemptive multitasking also allows important tasks to be scheduled earlier to deal with important external events quickly. These tasks wake up and deal with the important work when the operating system sends them a signal that triggers an interruption.

Older versions of Mac and Windows operating systems used **non-preemptive multitasking**. This is still used today on the RISC operating system. Unix systems started to use preemptive multitasking in 1969, being a core feature of all Unix-like systems and modern Windows versions from Windows NT `3.1` and Windows 95 onward.

Early-days CPUs could only run one path of instructions at a given time. Parallelism was achieved by switching between instruction streams, giving the illusion of parallelism at the software level by seemingly overlapping in execution.

However, in 2005, Intel® introduced multicore processors, which allowed several instruction streams to execute at once at the hardware level. This imposed some challenges at the time of writing software, as hardware-level concurrency now needed to be addressed and exploited.

C++ has supported concurrent programming since C++11 with the `std::thread` library. Earlier versions did not include any specific functionality, so programmers relied on platform-specific libraries based on the POSIX threading model in Unix systems or on proprietary Microsoft libraries in Windows systems.

Now that we better understand what concurrency is, we need to distinguish between concurrency and parallelism. Concurrency happens when many execution paths can run in overlapping time periods with interleaved execution, while parallelism happens when these tasks are executed at the same time by different CPU units, exploiting available multicore resources.

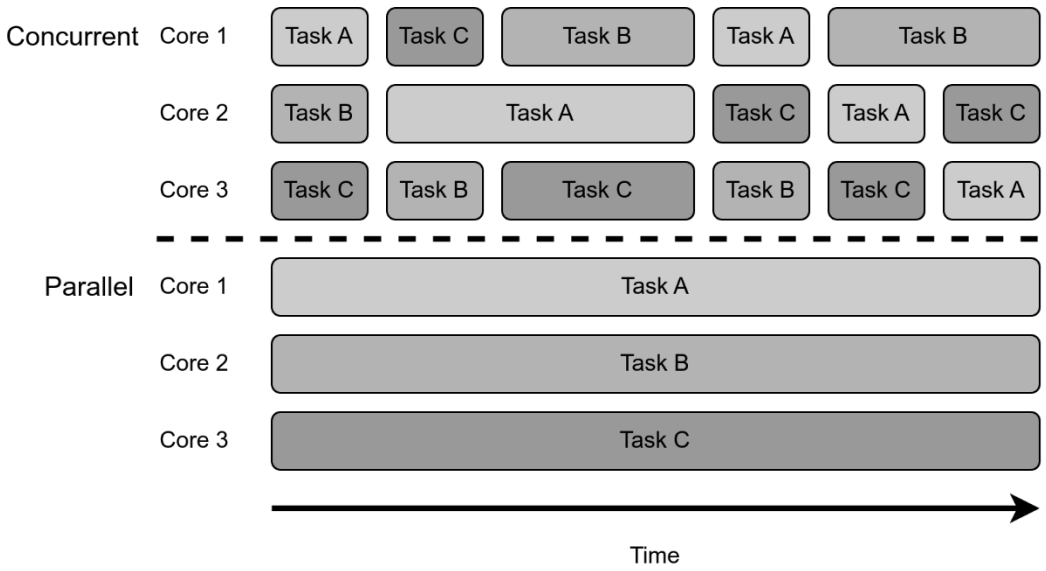

Figure 1.8: Concurrency versus parallelism

Concurrent programming is considered more general than parallel programming as the latter has a predefined communication pattern while the former can involve arbitrary and dynamic patterns of communication and interaction between tasks.

Parallelism can exist without concurrency (without interleaved time periods) and concurrency without parallelism (by multitasking by time-sharing on a single-core CPU).

Asynchronous programming

Asynchronous programming allows some tasks to be scheduled and run in the background while continuing to work on the current job without waiting for the scheduled tasks to finish. When these tasks are finished, they will report their results back to the main job or scheduler.

One of the key issues of synchronous applications is that a long operation can leave the program unresponsive to further input or processing. Asynchronous programs solve this issue by accepting new input while some operations are being executed with non-blocking tasks and the system can do more than one task at a time. This also allows for better resource utilization.

As the tasks are executed asynchronously and they report results back when they finish, this paradigm is especially suitable for event-driven programs. Also, it is a paradigm usually used for user interfaces, web servers, network communications, or long-running background processing.

As hardware has evolved toward multiple processing cores on a single processor chip, it has become mandatory to use asynchronous programming to take advantage of all the available compute power by running tasks in parallel across the different cores.

However, asynchronous programming has its challenges, as we will explore in this book. For example, it adds complexity, as the code is not interpreted in sequence. This can lead to race conditions. Also, error handling and testing are essential to ensure program stability and prevent issues.

As we will learn in this book, modern C++ also provides asynchronous mechanisms such as coroutines, which are programs that can be suspended and resumed later, or futures and promises as a proxy for unknown results in asynchronous programs for synchronizing the program execution.

Parallel programming

With parallel programming, multiple computation tasks can be done simultaneously on multiple processing units, either with all of them in the same computer (multicore) or on multiple computers (cluster).

There are two main approaches:

- **Shared-memory parallelism**: Tasks can communicate via shared memory, a memory space accessible by all processors
- **Message-passing parallelism**: Each task has its own memory space and uses message passing techniques to communicate with others

As with the previous paradigms, to achieve full potential and avoid bugs or issues, parallel computing needs synchronization mechanisms to avoid tasks interfering with each other. It also calls for load balancing the workload to reach its full potential, as well as reducing overhead when creating and managing tasks. These needs increase design, implementation, and debugging complexity.

Multithreading programming

Multithreading programming is a subset of parallel programming wherein a program is divided into multiple threads executing independent units within the same process. The process, memory space, and resources are shared between threads.

As we already mentioned, sharing memory needs synchronization mechanisms. On the other hand, as there is no need for inter-process communication, resource sharing is simplified.

For example, multithreading programming is usually used to achieve **graphical user interface** (**GUI**) responsiveness with fluid animations, in web servers to handle multiple clients' requests, or in data processing.

Event-driven programming

In event-driven programming, the control flow is driven by external events. The application detects events in real time and responds to these by invoking the appropriate event-handling method or callback.

An event signals an action that needs to be taken. This event is listened to by the event loop that continuously listens for incoming events and dispatches them to the appropriate callback, which will execute the desired action. As the code is only executed when an action occurs, this paradigm improves efficiency with resource usage and scalability.

Event-driven programming is useful to act on actions happening in user interfaces, real-time applications, and network connection listeners.

As with many of the other paradigms, the increased complexity, synchronization, and debugging make this paradigm complex to implement and apply.

As C++ is a low-level language, techniques such as callbacks or functors are used to write the event handlers.

Reactive programming

Reactive programming deals with data streams, which are continuous flows of data or values over time. A program is usually built using declarative or functional programming, defining a pipeline of operators and transformations applied to the stream. These operations happen asynchronously using schedulers and **backpressure** handling mechanisms.

Backpressure happens when the quantity of data overwhelms the consumers and they are not able to process all of it. To avoid a system collapse, a reactive system needs to use backpressure strategies to prevent system failures.

Some of these strategies include the following:

- Controlling input throughput by requesting the publisher to reduce the rate of published events. This can be achieved by following a pull strategy, where the publisher sends events only when the consumer requests them, or by limiting the number of events sent, creating a limited and controlled push strategy.
- Buffering the extra data, which is especially useful when there are data bursts or a high-bandwidth transmission over a short period.
- Dropping some events or delaying their publication until the consumers recover from the backpressure state.

Thus, reactive programs can be **pull-based** or **push-based**. Pull-based programs implement the classic case where the events are actively pulled from the data source. On the other hand, push-based programs push events through a signal network to reach the subscriber. Subscribers react to changes without blocking the program, making these systems ideal for rich user interface environments where responsiveness is crucial.

Reactive programming is like an event-driven model where event streams from various sources can be transformed, filtered, processed, and so on. Both increase code modularity and are suitable for real-time applications. However, there are some differences, as follows:

- Reactive programming reacts to event streams, while event-driven programming deals with discrete events.
- In event-driven programming, an event triggers a callback or event handlers. With reactive programming, a pipeline with different transformation operators can be created whereby the data stream will flow and modify the events.

Examples of systems and software using reactive programming include the X Windows system and libraries such as Qt, WxWidgets, and Gtk+. Reactive programming is also used in real-time sensors data processing and dashboards. Additionally, it is applied to handling network or file I/O traffic and data processing.

To reach full potential, there are some challenges to address when using reactive programming. For example, it's important to debug distributed dataflows and asynchronous processes or to optimize performance by fine-tuning the schedulers. Also, the use of declarative or functional programming makes developing software by using reactive programming techniques a bit more challenging to understand and learn.

Dataflow programming

With **dataflow programming**, a program is designed as a directed graph of nodes representing computation units and edges representing the flow of data. A node only executes when there is some available data. This paradigm was invented by Jack Dennis at MIT in the 1960s.

Dataflow programming makes the code and design more readable and clearer, as it provides a visual representation of the different computation units and how they interact. Also, independent nodes can run in parallel with dataflow programming, increasing parallelism and throughput. So, it is like reactive programming but offers a graph-based approach and visual aid to modeling systems.

To implement a dataflow program, we can use a hash table. The key identifies a set of inputs and the value describes the task to run. When all inputs for a given key are available, the task associated with that key is executed, generating additional input values that may trigger tasks for other keys in the hash table. In these systems, the scheduler can find opportunities for parallelism by using a topological sort on the graph data structure, sorting the different tasks by their interdependencies.

This paradigm is usually used for large-scale data processing pipelines for machine learning, real-time analysis from sensors or financial markets data, and audio, video, and image processing systems. Examples of software libraries using the dataflow paradigm are Apache Spark and TensorFlow. In hardware, we can find examples for digital signal processing, network routing, GPU architecture, telemetry, and artificial intelligence, among others.

A variant of dataflow programming is **incremental computing**, whereby only the outputs that depend on changed input data are recomputed. This is like recomputing affected cells in an Excel spreadsheet when a cell value changes.

Now that we have learned about the different parallel programming systems, models, and paradigms, it's time to introduce some **metrics** that help measure parallel systems' performance.

Exploring the metrics to assess parallelism

Metrics are measurements that can help us understand how a system is performing and to compare different improvement approaches.

Here are some metrics and formulas commonly used to evaluate parallelism in a system.

Degree of parallelism

Degree of parallelism (**DOP**) is a metric that indicates the number of operations being simultaneously executed by a computer. It is useful to describe the performance of parallel programs and multi-processor systems.

When computing the DOP, we can use the maximum number of operations that could be done simultaneously, measuring the ideal case scenario without bottlenecks or dependencies. Alternatively, we can use either the average number of operations or the number of simultaneous operations at a given point in time, reflecting the actual DOP achieved by a system. An approximation can be done by using profilers and performance analysis tools to measure the number of threads during a particular time period.

That means that the DOP is not a constant; it is a dynamic metric that changes during application execution.

For example, consider a script tool that processes multiple files. These files can be processed sequentially or simultaneously, increasing efficiency. If we have a machine with `N` cores and we want to process `N` files, we can assign a file to each core.

The time to process all files sequentially would be as follows:

$$t_{total} = t_{file1} + t_{file2} + t_{file3} + \cdots + t_{fileN} \cong N \cdot avg(t_{file})$$

And, the time to process them in parallel would be:

$$t_{total} = max(t_{file1}, t_{file2}, t_{file3}, \cdots, t_{fileN})$$

Therefore, the DOP is `N`, the number of cores actively processing separate files.

There is a theoretical upper bound on the speed-up that parallelization can achieve, which is given by **Amdahl's law**.

Amdahl's law

In a parallel system, we could believe that doubling the number of CPU cores could make the program run twice as fast, thereby halving the runtime. However, the speed-up from parallelization is not linear. After a certain number of cores, the runtime is not reduced anymore due to different circumstances such as context switching, memory paging, and so on.

The Amdahl's law formula computes the theoretical maximum speed-up a task can perform after parallelization as follows:

$$S_{max}(s) = \frac{s}{s + p(1 - s)} = \frac{1}{1 - p + \frac{p}{s}}$$

Here, `s` is the speed-up factor of the improved part and `p` is the fraction of the parallelizable part compared to the entire process. Therefore, `1-p` represents the ratio of the task not parallelizable (the bottleneck or sequential part), while `p/s` represents the speed-up achieved by the parallelizable part.

That means that the maximum speed-up is limited by the sequential portion of the task. The greater the fraction of the parallelizable task (`p` approaches `1`), the more the maximum speed-up increases up to the speed-up factor (`s`). On the other hand, when the sequential portion becomes larger (`p` approaches `0`), `Smax` tends to `1`, meaning that no improvement is possible.

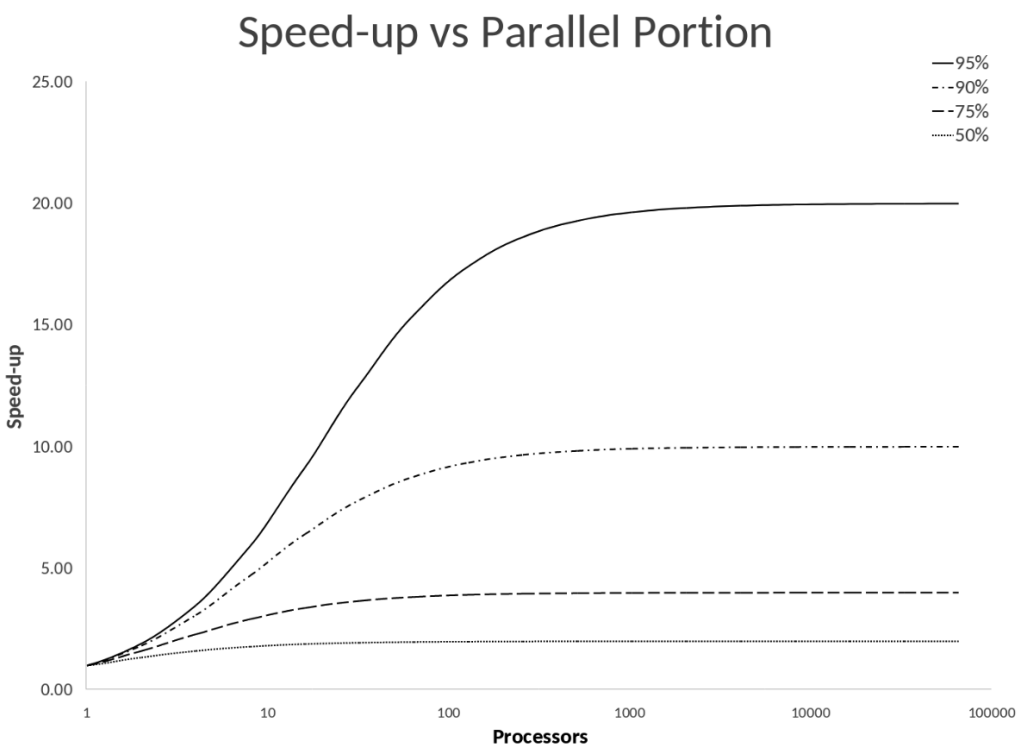

Figure 1.9: The speed-up limit by the number of processors and percentage of parallelizable parts

The **critical path** in parallel systems is defined by the longest chain of dependent calculations. As the critical path is hardly parallelizable, it defines the sequential portion and thus the quicker runtime that a program can achieve.

For example, if the sequential part of a process represents 10% of the runtime, then the fraction of the parallelizable part is `p=0.9`. In this case, the potential speed-up will not exceed 10 times the speed-up, regardless of the number of processors available.

Gustafson's law

The Amdahl's law formula can only be used with fixed-sized problems and increasing resources. When using larger datasets, time spent in the parallelizable part grows much faster than that in the sequential part. In these cases, the Gustafson's law formula is less pessimistic and more accurate, as it accounts for fixed execution time and increasing problem size with additional resources.