Chapter 29
Introduction to Concurrency and Parallelism

29.1 Introduction

In this chapter we will introduce the concepts of concurrency and parallelism. We will also briefly consider the related topic of distribution. After this we will consider process synchronisation, why object oriented approaches are well suited to concurrency and parallelism before finishing with a short discussion of threads versus processes.

29.2 Concurrency

Concurrency is defined by the dictionary as

> two or more events or circumstances happening or existing at the same time.

In Computer Science concurrency refers to the ability of different parts or units of a program, algorithm or problem to be *executed at the same time*, potentially on multiple processors or multiple cores.

Here a processor refers to the central processing unit (or CPU) or a computer while core refers to the idea that a CPU chip can have multiple cores or processors on it.

Originally a CPU chip had a single core. That is the CPU chip had a single processing unit on it. However, over time, to increase computer performance, hardware manufacturers added additional *cores* or processing units to chips. Thus a dual-core CPU chip has two processing units while a quad-core CPU chip has four processing units. This means that as far as the operating system of the computer is concerned, it has multiple CPUs on which it can run programs.

Running processing at the same time, on multiple CPUs, can substantially improve the overall performance of an application.

J. Hunt, *Advanced Guide to Python 3 Programming*,
Undergraduate Topics in Computer Science,
https://doi.org/10.1007/978-3-030-25943-3_29

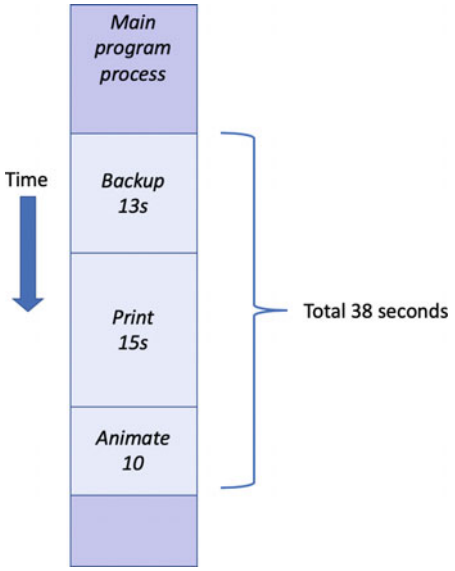
For example, let us assume that we have a program that will call three independent functions, these functions are:

- make a backup of the current data held by the program,
- print the data currently held by the program,
- run an animation using the current data.

Let us assume that these functions run sequentially, with the following timings:

- the *backup* function takes 13 s,
- the *print* function takes 15 s,
- the *animation* function takes 10 s.

This would result in a total of 38 s to perform all three operations. This is illustrated graphically below:

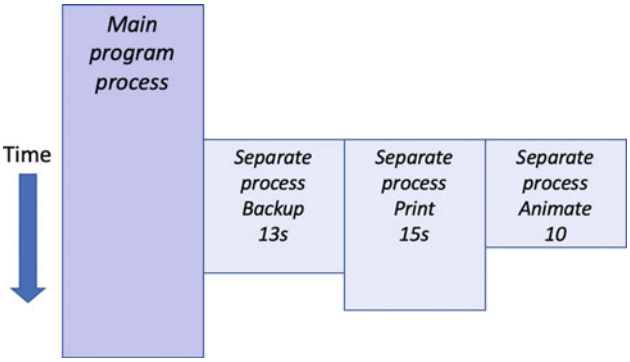


However, the three functions are all completely *independent* of each other. That is they do not rely on each other for any results or behaviour; they do not need one of the other functions to complete before they can complete etc. Thus we can run each function *concurrently*.

If the underlying operating system and program language being used support multiple processes, then we can potentially run each function in a separate process at the same time and obtain a significant speed up in overall execution time.

If the application starts all three functions at the same time, then the maximum time before the main process can continue will be 15s, as that is the time taken by the longest function to execute. However, the main program may be able to continue as soon as all three functions are started as it also does not depend on the

results from any of the functions; thus the delay may be negligible (although there will typically be some small delay as each process is set up). This is shown graphically below:



29.3 Parallelism

A distinction its often made in Computer Science between *concurrency* and *parallelism*.

In *concurrency*, separate independent tasks are performed potentially at the same time.

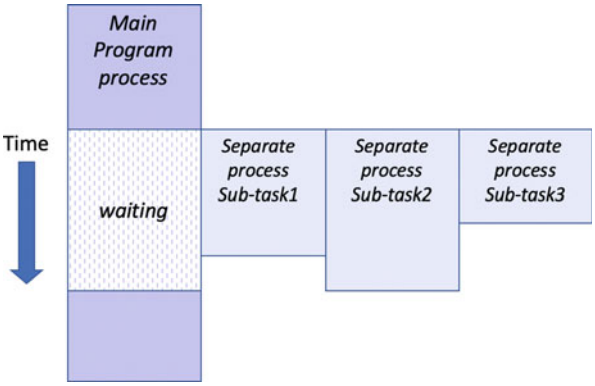
In *parallelism*, a large complex task is broken down into a set of *subtasks*. The subtasks represent part of the overall problem. Each subtask can be executed at the same time. Typically it is necessary to combine the results of the subtasks together to generate an overall result. These subtasks are also very similar if not functionally exactly the same (although in general each subtask invocation will have been supplied with different data).

Thus parallelism is when *multiple copies* of the same functionality are run at the same time, but on different data.

Some examples of where parallelism can be applied include:

- **A web search engine**. Such a system may look at many, many web pages. Each time it does so it must send a request to the appropriate web site, receive the result and process the data obtained. These steps are the same whether it is the BBC web site, Microsoft's web site or the web site of Cambridge University. Thus the requests can be run sequentially or in parallel.
- **Image Processing**. A large image may be broken down into slices so that each slice can be analysed in parallel.

The following diagram illustrates the basic idea behind parallelism; a main program fires off three subtasks each of which runs in parallel. The main program then waits for all the subtasks to complete before combining together the results from the subtasks before it can continue.



29.4 Distribution

When implementing a concurrent or parallel solution, where the resulting processes run is typically an implementation detail. Conceptually these processes could run on the same processor, physical machine or on a remote or distributed machine. As such distribution, in which problems are solved or processes executed by sharing the work across multiple physical machines, is often related to concurrency and parallelism.

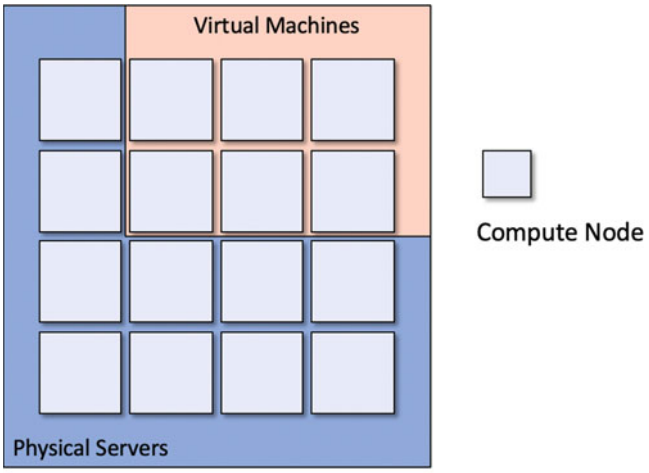
However, there is no requirement to distribute work across physical machines, indeed in doing so extra work is usually involved.

To distribute work to a remote machine, data and in many cases code, must be transferred and made available to the remote machine. This can result in significant delays in running the code remotely and may offset any potential performance advantages of using a physically separate computer. As a result many concurrent/parallel technologies default to executing code in a separate process on the same machine.

29.5 Grid Computing

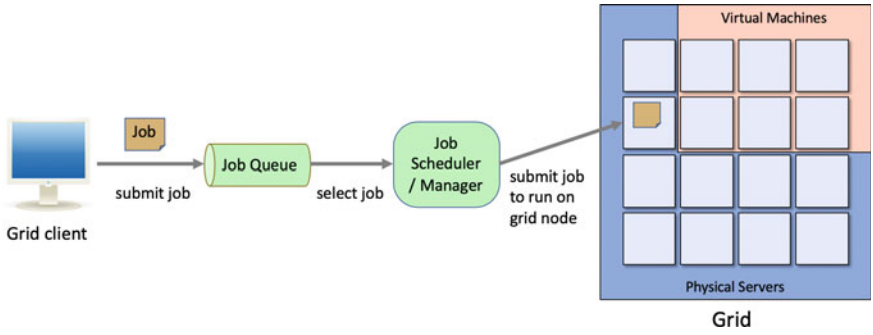
Grid Computing is based on the use of a network of loosely coupled computers, in which each computer can have a job submitted to it, which it will run to completion before returning a result.

In many cases the grid is made up of a *heterogeneous* set of computers (rather than all computers being the same) and may be geographically dispersed. These computers may be comprised of both physical computers and virtual machines.



A *Virtual Machine* is a piece of software that emulates a whole computer and runs on some underlying hardware that is shared with other virtual machines. Each Virtual Machine thinks it is the only computer on the hardware; however the virtual machines all share the resources of the physical computer. Multiple virtual machines can thus run simultaneously on the same physical computer. Each virtual machine provides its own virtual hardware, including CPUs, memory, hard drives, network interfaces and other devices. The virtual hardware is then mapped to the real hardware on the physical machine which saves costs by reducing the need for physical hardware systems along with the associated maintenance costs, as well as reducing the power and cooling demands of multiple computers.

Within a grid, software is used to manage the grid nodes and to submit jobs to those nodes. Such software will receive the jobs to perform (programs to run and information about the environment such as libraries to use) from clients of the grid. These jobs are typically added to a job queue before a job scheduler submits them to a node within the grid. When any results are generated by the job they are collected from the node and returned to the client. This is illustrated below:



The use of grids can make distributing concurrent/parallel processes amongst a set of physical and virtual machines much easier.

29.6 Concurrency and Synchronisation

Concurrency relates to executing multiple tasks at the same time. In many cases these tasks are not related to each other such as printing a document and refreshing the User Interface. In these cases, the separate tasks are completely independent and can execute at the same time without any interaction.

In other situations multiple concurrent tasks need to interact; for example, where one or more tasks produce data and one or more other tasks consume that data. This is often referred to as a *producer-consumer* relationship. In other situations, all parallel processes must have reached the same point before some other behaviour is executed.

Another situation that can occur is where we want to ensure that only one concurrent task executes a piece of sensitive code at a time; this code must therefore be protected from concurrent access.

Concurrent and parallel libraires need to provide facilities that allow for such synchronisation to occur.

29.7 Object Orientation and Concurrency

The concepts behind object-oriented programming lend themselves particularly well to the concepts associated with concurrency. For example, a system can be described as a set of discrete objects communicating with one another when necessary. In Python, only one object may execute at any one moment in time within a single interpreter. However, conceptually at least, there is no reason why this restriction should be enforced. The basic concepts behind object orientation still hold, even if each object executes within a separate independent process.

Traditionally a message send is treated like a procedural call, in which the calling object's execution is blocked until a response is returned. However, we can extend this model quite simply to view each object as a concurrently executable program, with activity starting when the object is created and continuing even when a message is sent to another object (unless the response is required for further processing). In this model, there may be very many (concurrent) objects executing at the same time. Of course, this introduces issues associated with resource allocation, etc. but no more so than in any concurrent system.

One implication of the concurrent object model is that objects are larger than in the traditional single execution thread approach, because of the overhead of having each object as a separate thread of execution. Overheads such as the need for a scheduler to handling these execution threads and resource allocation mechanisms means that it is not feasible to have integers, characters, etc. as separate processes.

29.8 Threads V Processes

As part of this discussion it is useful to understand what is meant by a process. A process is an instance of a computer program that is being executed by the operating system. Any process has three key elements; the program being executed, the data used by that program (such as the variables used by the program) and the state of the process (also known as the execution context of the program).

A (Python) Thread is a preemptive lightweight process.

A Thread is considered to be *pre-emptive* because every thread has a chance to run as the main thread at some point. When a thread gets to execute then it will execute until

- completion,
- until it is waiting for some form of I/O (Input/Output),
- sleeps for a period of time,
- it has run for 15 ms (the current threshold in Python 3).

If the thread has not completed when one of the above situations occurs, then it will give up being the executing thread and another thread will be run instead. This means that one thread can be interrupted in the middle of performing a series of related steps.

A thread is a considered a *lightweight* process because it does not possess its own address space and it is not treated as a separate entity by the host operating system. Instead, it exists within a single machine process using the same address space.

It is useful to get a clear idea of the difference between a thread (running within a single machine process) and a multi-process system that uses separate processes on the underlying hardware.

29.9 Some Terminology

The world of concurrent programming is full of terminology that you may not be familiar with. Some of those terms and concepts are outlined below:

- **Asynchronous versus Synchronous invocations**. Most of the method, function or procedure invocations you will have seen in programming represent synchronous invocations. A synchronous method or function call is one which blocks the calling code from executing until it returns. Such calls are typically within a single thread of execution. Asynchronous calls are ones where the flow of control immediately returns to the callee and the caller is able to execute in its own thread of execution. Allowing both the caller and the call to continue processing.
- **Non-Blocking versus Blocking code**. Blocking code is a term used to describe the code running in one thread of execution, waiting for some activity to complete which causes one of more separate threads of execution to also be delayed. For example, if one thread is the producer of some data and other threads are the consumers of that data, then the consumer treads cannot continue until the producer generates the data for them to consume. In contrast, non-blocking means that no thread is able to indefinitely delay others.
- **Concurrent versus Parallel code**. Concurrent code and parallel code are similar, but different in one significant aspect. Concurrency indicates that two or more activities are both making progress even though they might not be executing at the same point in time. This is typically achieved by continuously swapping competing processes between execution and non-execution. This process is repeated until at least one of the threads of execution (Threads) has completed their task. This may occur because two threads are sharing the same physical processor with each is being given a short time period in which to progress before the other gets a short time period to progress. The two threads are said to be sharing the processing time using a technique known as time slicing. Parallelism on the other hand implies that there are multiple processors available allowing each thread to execute on their own processor simultaneously.

29.10 Online Resources

See the following online resources for information on the topics in this chapter:

- https://en.wikipedia.org/wiki/Concurrency_(computer_science) Wikipedia page on concurrency.
- https://en.wikipedia.org/wiki/Virtual_machine Wikipedia page on Virtual Machines.

- https://en.wikipedia.org/wiki/Parallel_computing Wikipedia page on parallelism.
- http://tutorials.jenkov.com/java-concurrency/concurrency-vs-parallelism.html Concurrency versus Parallelism tutorial.
- https://www.redbooks.ibm.com/redbooks/pdfs/sg246778.pdf IBM Red Book on an Introduction to Grid Computing.