CPU-intensive data processing tasks have become crucial considering the complexity of the various big data applications that are used today. Reducing the CPU utilization per process is very important for improving the overall speed of applications.

This book will teach you how to perform parallel execution of computations by distributing them across multiple processors and multiple machines, thus improving the overall performance of a big data processing task. We will cover synchronous and asynchronous models, shared memory and file systems, communication between various processes, synchronization, and more.

Who this book is written for
This book is for Python developers who have developed Python programs for data processing and now want to learn how to write fast, efficient programs that perform CPU-intensive data processing tasks.
In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 1 'An Introduction to Parallel and Distributed Computing'
- A synopsis of the book’s content
- More information on Distributed Computing with Python
Francesco Pierfederici is a software engineer who loves Python. He has been working in the fields of astronomy, biology, and numerical weather forecasting for the last 20 years.

He has built large distributed systems that make use of tens of thousands of cores at a time and run on some of the fastest supercomputers in the world. He has also written a lot of applications of dubious usefulness but that are great fun. Mostly, he just likes to build things.
Preface

Parallel and distributed computing is a fascinating subject that only a few years ago developers in only a very few large companies and national labs were privy to. Things have changed dramatically in the last decade or so, and now everybody can build small- and medium-scale distributed applications in a variety of programming languages including, of course, our favorite one: Python.

This book is a very practical guide for Python programmers who are starting to build their own distributed systems. It starts off by illustrating the bare minimum theoretical concepts needed to understand parallel and distributed computing in order to lay the basic foundations required for the rest of the (more practical) chapters.

It then looks at some first examples of parallelism using nothing more than modules from the Python standard library. The next step is to move beyond the confines of a single computer and start using more and more nodes. This is accomplished using a number of third-party libraries, including Celery and Pyro.

The remaining chapters investigate a few deployment options for our distributed applications. The cloud and classic High Performance Computing (HPC) clusters, together with their strengths and challenges, take center stage.

Finally, the thorny issues of monitoring, logging, profiling, and debugging are touched upon.

All in all, this is very much a hands-on book, teaching you how to use some of the most common frameworks and methodologies to build parallel and distributed systems in Python.
What this book covers

Chapter 1, *An Introduction to Parallel and Distributed Computing*, takes you through the basic theoretical foundations of parallel and distributed computing.

Chapter 2, *Asynchronous Programming*, describes the two main programming styles used in distributed applications: synchronous and asynchronous programming.

Chapter 3, *Parallelism in Python*, shows you how to do more than one thing at the same time in your Python code, using nothing more than the Python standard library.

Chapter 4, *Distributed Applications – with Celery*, teaches you how to build simple distributed applications using Celery and some of its competitors: Python-RQ and Pyro.

Chapter 5, *Python in the Cloud*, shows how you can deploy your Python applications on the cloud using Amazon Web Services.

Chapter 6, *Python on an HPC Cluster*, shows how to deploy your Python applications on a classic HPC cluster, typical of many universities and national labs.

Chapter 7, *Testing and Debugging Distributed Applications*, talks about the challenges of testing, profiling, and debugging distributed applications in Python.

Chapter 8, *The Road Ahead*, looks at what you have learned so far and which directions interested readers could take to push their development of distributed systems further.
An Introduction to Parallel and Distributed Computing

The first modern digital computer was invented in the late 30s and early 40s (that is, arguably, the Z1 from Konrad Zuse in 1936), probably before most of the readers of this book—let alone the author—were born. These last seventy odd years have seen computers become faster and cheaper at an amazing rate, which was unique across industries. Just think that today's smartphones (for example, the latest iPhones or Android phones) are faster than the fastest computer in the world from just 20 years ago. Not to mention, the amazing feat of miniaturization: those supercomputers used to take up entire rooms; now they fit in our pockets.

These years have also seen, among others, two key inventions relevant to the topic at hand. One is the ability to cram more than one processor on a single motherboard (and even multiple CPU cores on a single processor). This development was crucial in allowing computations to be performed truly concurrently. As we know, processors are able to perform only one task at a time; however, as we will see later on in the chapter, they are fast enough to give the illusion of being able to run multiple tasks at the same time. To be able to perform more than one action exactly at the same time, you need access to more than one processor.

The other critical invention is high-speed computer networking. This allowed, for the first time, a potentially enormous number of computers to communicate with each other. These networked machines can either be located in the same office/building (the so-called Local Area Network (LAN)) or be spread out across different buildings, cities, or even across the planet (that is, WAN or wide area networking).
By now, most of us are familiar with multiprocessor/multicore computers, and indeed, the chances are pretty high that the phone in our pocket, the tablet in our hands, or the laptop we take on the road has a handful of cores already. The graphics card, also called Graphics Processing Unit (GPU) in these devices is more often than not massively parallel, with hundreds or even thousands of processing units. Computer networks too are all around us, starting from the most famous of them all: the Internet, to the Wi-Fi in our homes and coffee shops and the 4G mobile networks our phones use.

In the rest of this chapter, we will lay some working definitions of the topics that we will explore in the rest of the book. We will be introducing the concepts of parallel and distributed computing. We will give some examples of each that are taken from familiar topics and technologies. Some general advantages and disadvantages of each architecture and programming paradigm will be discussed as well.

Before proceeding with our definitions and a little bit of theory, let's clarify a possible source of confusion. In this and the following chapters, we will use the term processor and the term CPU core (or even simply core) interchangeably, unless otherwise specified. This is, of course, technically incorrect; a processor has one or more cores, and a computer has one or more processors as cores do not exist in isolation. Depending on the algorithm and its performance requirements, running on multiple processors or on a single processor using multiple cores can make a bit of difference in speed, assuming, of course, that the algorithm can be parallelized in the first place. For our intents and purposes, however, we will ignore these differences and refer to more advanced texts for further exploration of this topic.

Parallel computing
Definitions of parallel computing abound. However, for the purpose of this book, a simple definition will suffice, which is as follows:

Parallel computing is the simultaneous use of more than one processor to solve a problem.

Typically, this definition is further specialized by requiring that the processors reside on the same motherboard. This is mostly to distinguish parallel computing from distributed computing (which is discussed in the next section).

The idea of splitting work among many workers is as old as human civilization, is not restricted to the digital world, and finds an immediate and obvious application in modern computers equipped with higher and higher numbers of compute units.
Chapter 1

There are, of course, many reasons why parallel computing might be useful and even necessary. The simplest one is performance; if we can indeed break up a long-running computation into smaller chunks and parcel them out to different processors, then we can do more work in the same amount of time.

Other times, and just as often, parallel computing techniques are used to present users with responsive interfaces while the system is busy with some other task. Remember that one processor executes just one task at the time. Applications with GUIs need to offload work to a separate thread of execution running on another processor so that one processor is free to update the GUI and respond to user inputs.

The following figure illustrates this common architecture, where the main thread is processing user and system inputs using what is called an **event loop**. Tasks that require a long time to execute and those that would otherwise block the GUI are offloaded to a background or worker thread:

![Event Loop Diagram](image)

A simple real-world example of this parallel architecture could be a photo organization application. When we connect a digital camera or a smartphone to our computers, the photo application needs to perform a number of actions; all the while its user interface needs to stay interactive. For instance, our application needs to copy images from the device to the internal disk, create thumbnails, extract metadata (for example, date and time of the shot), index the images, and finally update the image gallery. While all of this happens, we are still able to browse images that are already imported, open them, edit them, and so on.

Of course, all these actions could very well be performed sequentially on a single processor—the same processor that is handling the GUI. The drawback would be a sluggish interface and an extremely slow overall application. Performing these steps in parallel keeps the application snappy and its users happy.
The astute reader might jump up at this point and rightfully point out that older computers, with a single processor and a single core, could already perform multiple things at the same time (by way of multitasking). What happened back then (and even today, when we launch more tasks than there are processors and cores on our computers) was that the one running task gave up the CPU (either voluntarily or forcibly by the OS, for example, in response to an IO event) so that another task could run in its place. These interrupts would happen over and over again, with various tasks acquiring and giving up the CPU many times over the course of the application's life. In those cases, users had the impression of multiple tasks running concurrently, as the switches were extremely fast. In reality, however, only one task was running at any given time.

The typical tools used in parallel applications are threads. On systems such as Python (as we will see in Chapter 3, Parallelism in Python) where threads have significant limitations, programmers resort to launching (oftentimes, by means of forking) subprocesses instead. These subprocesses replace (or complement) threads and run alongside the main application process.

The first technique is called multithreaded programming. The second is called multiprocessing. It is worth noting that multiprocessing should not be seen as inferior or as a workaround with respect to using multiple threads.

There are many situations where multiprocessing is preferable to multiple threads. Interestingly, even though they both run on a single computer, a multithreaded application is an example of shared-memory architecture, whereas a multiprocess application is an example of distributed memory architecture (refer to the following section to know more).

**Distributed computing**

For the remainder of this book, we will adopt the following working definition of distributed computing:

* Distributed computing is the simultaneous use of more than one computer to solve a problem.

Typically, as in the case of parallel computing, this definition is oftentimes further restricted. The restriction usually is the requirement that these computers appear to their users as a single machine, therefore hiding the distributed nature of the application. In this book, we will be happy with the more general definition.
Distributing computation across multiple computers is again a pretty obvious strategy when using systems that are able to speak to each other over the (local or otherwise) network. In many respects, in fact, this is just a generalization of the concepts of parallel computing that we saw in the previous section.

Reasons to build distributed systems abound. Oftentimes, the reason is the ability to tackle a problem so big that no individual computer could handle it at all, or at least, not in a reasonable amount of time. An interesting example from a field that is probably familiar to most of us is the rendering of 3D animation movies, such as those from Pixar and DreamWorks.

Given the sheer number of frames to render for a full-length feature (30 frames per second on a two-hour movie is a lot!), movie studios need to spread the full-rendering job to large numbers of computers (computer farms as they are called).

Other times, the very nature of the application being developed requires a distributed system. This is the case, for instance, for instant messaging and video conferencing applications. For these pieces of software, performance is not the main driver. It is just that the problem that the application solves is itself distributed.

In the following figure, we see a very common web application architecture (another example of a distributed application), where multiple users connect to the website over the network. At the same time, the application itself communicates with systems (such as a database server) running on different machines in its LAN:

Another interesting example of distributed systems, which might be a bit counterintuitive, is the CPU-GPU combination. These days, graphics cards are very sophisticated computers in their own right. They are highly parallel and offer compelling performance for a large number of compute-intensive problems, not just for displaying images on screen. Tools and libraries exist to allow programmers to make use of GPUs for general-purpose computing (for example CUDA from NVIDIA, OpenCL, and OpenAcc among others).
However, the system composed by the CPU and GPU is really an example of a distributed system, where the network is replaced by the PCI bus. Any application exploiting both the CPU and the GPU needs to take care of data movement between the two subsystems just like a more traditional application running across the network!

It is worth noting that, in general, adapting the existing code to run across computers on a network (or on the GPU) is far from a simple exercise. In these cases, I find it quite helpful to go through the intermediate step of using multiple processes on a single computer first (refer to the discussion in the previous section). Python, as we will see in Chapter 3, Parallelism in Python, has powerful facilities for doing just that (refer to the `concurrent.futures` module).

Once I evolve my application so that it uses multiple processes to perform operations in parallel, I start thinking about how to turn these processes into separate applications altogether, which are no longer part of my monolithic core.

Special attention must be given to the data—where to store it and how to access it. In simple cases, a shared filesystem (for example, NFS on Unix systems) is enough; other times, a database and/or a message bus is needed. We will see some concrete examples from Chapter 4, Distributed Applications – with Celery, onwards. It is important to remember that, more often than not, data, rather than CPU, is the real bottleneck.

**Shared memory versus distributed memory**

Conceptually, parallel computing and distributed computing look very similar—after all, they both are about breaking up some computation into several smaller parts and running those on processors. Some of you might ponder upon the fact that in one case the processors in use are part of the same computer, whereas in the other case they are physically on different computers; is this just a trivial technicality?

The answer is maybe. As we saw, some applications are fundamentally distributed. Others simply need more performance than they can get on a single box. For some of these applications, the answer is maybe yes—it does not really matter where the processing power comes from. However, in all cases, the physical location of the hardware resources in use has significant implications.

Possibly, the most obvious difference between parallel and distributed computing is in the underlying memory architecture and access patterns. In the case of a parallel application, all concurrent tasks can—in principle—access the same memory space. Here, we have to say *in principle* because as we have already saw, parallelism does not necessary imply the use of threads (which can indeed access the same memory space).
In the following figure, we see a typical shared-memory architecture where four processors (the four CPU boxes in the following diagram) can all access the same memory address space (that is, the Memory box). If our application were to use threads, then those would be able to access exactly the same memory locations, if needed:

In the case of a distributed application, however, the various concurrent tasks cannot normally access the same memory space. The reason being that some tasks run on one computer and others on another, physically separated, computer.

Since these computers are able to talk to each other over the network, one could imagine writing a software layer (a middleware) that could present our application with a unified logical (as opposed to physical) memory space. These types of middlewares do exist and implement what is known as distributed shared-memory architecture. We will not examine these systems in this book.

In the following figure, we have the same four CPUs as before, that are organized now in a shared-memory architecture. Each CPU has access to its own private memory and cannot see any other CPU memory space. The four computers (indicated by the boxes surrounding their CPU and memory) communicate through the network (the black line connecting them). Each data transfer between boxes happens over the network:
In reality, the computers one is likely to use nowadays are a hybrid of the two extremes that we just described in the previous section. Computers communicate over the network just like in a pure distributed-memory architecture. However, each computer has more than one processor and/or processor core, implementing a shared-memory architecture. The following figure schematically illustrates such a hybrid architecture that is shared within each individual computer (indicated by a box enclosing two CPUs and its own memory) and distributed across boxes (linked together via the network):

Each of these architectures has its own pros and cons. In the case of shared-memory systems, sharing data across various concurrent threads of a single executable is quite fast and tremendously faster than using the network. In addition, having a single, uniform memory address space makes writing the code arguably simpler.

At the same time, special care has to be exercised in the design of the program to avoid multiple threads from stepping on each other’s toes and changing variables behind each other’s backs.

A distributed memory system tends to be very scalable and cheap to assemble; need more power? Simply add another box. Another advantage is that processors can access their own memory in isolation without worrying about race conditions (while this is technically true, oftentimes, different tasks running in parallel need to read and write data in a common repository, for example, a database or a shared filesystem. In those cases, we still have to deal with race conditions). Disadvantages of this system include the fact that programmers need to implement their own strategy to move data around and they need to worry about issues of data locality. Also, not all algorithms easily map to these architectures.
Chapter 1

Amdahl's law

The last important concept of this chapter is a behavior known as Amdahl's law. In simple terms, Amdahl's law states that we can parallelize/distribute our computations as much as we want, gaining in performance as we add compute resources. However, our code cannot be faster than the speed of its combined sequential (that is, non parallelizable) parts on a single processor.

Put more formally, Amdahl's law has the following formulation. Given an algorithm that is partially parallel, let's call $P$ its parallel fraction and $S$ its serial (that is, non parallel) fraction (clearly, $S+P=100\%$). Furthermore, let's call $T(n)$ the runtime (in seconds) of the algorithm when using $n$ processors. Then, the following relation holds:

$$T(n) \geq S \times T(1) + \frac{P \times T(1)}{n}$$

The preceding relation, translated in plain English states the following:

The execution time of the algorithm described here on $n$ processors is equal—and generally greater—than the execution time of its serial part on one processor (that is, $S \times T(1)$) plus the execution time of its parallel part on one processor (that is, $P \times T(1)$) divided by $n$: the number of processors.

As we increase the number, $n$, of processors used by our code, the second term on the equation becomes smaller and smaller, eventually becoming negligible with respect to the first term. In those cases, the preceding relation simply becomes this:

$$T(\infty) \approx S \times T(1)$$

The translation of this relation in plain English is as follows:

The execution time of the algorithm described here on an infinite number of processors (that is, a really large number of processors) is approximately equal to the execution time of its serial part on a single processor (that is, $S \times T(1)$).

Now, let's stop for a second and think about the implication of Amdahl's law. What we have here is a pretty simple observation: oftentimes, we cannot fully parallelize our algorithms.
An Introduction to Parallel and Distributed Computing

Which means that, most of the time, we cannot have $S=0$ in the preceding relations. The reasons for this are numerous: we might have to copy data and/or code to where the various processors will be able to access them. We might have to split the data into chunks and move those chunks over the network. We might have to collect the results of all the concurrent tasks and perform some further processing on them, and so on.

Whatever the reason, if we cannot fully parallelize our algorithm, eventually the runtime of our code will be dominated by the performance of the serial fraction. Not only that, but even before that happens, we will start to see increasingly worse-than-expected speedups.

As a side note, algorithms that are fully parallel are usually called **embarrassingly parallel** or, in a more politically correct way, pleasantly parallel and offer impressive scalability properties (with speedups often linear with the number of processors). Of course, there is nothing embarrassing about those pieces of software! Unfortunately, they are not as common as we would like.

Let's try to visualize the entire Amdahl's law with some numbers. Assume that our algorithm takes 100 seconds on a single processor. Let's also assume that we can parallelize 99% of it, which would be a pretty amazing feat, most of the time. We can make our code go faster by increasing the number of processor we use, as expected. Let's take at look at the following calculation:

\[
T(1) = 100s
\]

\[
T(10) \approx 0.01 \times 100s + \frac{0.99 \times 100s}{10} = 10.9s \Rightarrow 9.2X \text{ speedup}
\]

\[
T(100) \approx 1s + 0.99s = 1.99s \Rightarrow 50.2X \text{ speedup}
\]

\[
T(1000) \approx 1s + 0.099s = 1.099s \Rightarrow 91X \text{ speedup}
\]

We can see from the preceding numbers that the speedup increase with increasing values of $n$ is rather disappointing. We start with a truly amazing 9.2X speedup using 10 processors, and then we drop to just 50X when using 100 processors and a paltry 91X when using 1,000 processors!
The following figure plots the expected best-case speedup for the same algorithm (calculated up to almost $n=10,000$). It does not matter how many processors we use; we cannot get a speedup greater than 100X, meaning that the fastest our code will run is one second, which is the time its serial fraction takes on a single processor, exactly as predicted by Amdahl's law:

Amdahl's law tells us two things: how much of a speedup we can reasonably expect in the best-case scenario and when to stop throwing hardware at the problem because of diminishing returns.

Another interesting observation is that Amdahl's law applies equally to distributed systems and hybrid parallel-distributed systems as well. In those cases, $n$ refers to the total number of processors across computers.

One aspect that should be mentioned at this point is that as the systems that we can use become more powerful, our distributed algorithms will take less and less time to run, if they can make use of the extra cycles.

When the runtime of our applications becomes reasonably short, what usually happens is that we tend to tackle bigger problems. This aspect of algorithm evolution, namely the expanding of the problem size (and therefore of the computing requirements) once acceptable performance levels are reached, is what is called Gustafson's law.
The mixed paradigm
Since most of the computers we buy today (2016) are multicore and often even both multiprocessor and multicore, any distributed application that we will write is likely to run on such systems. This brings us to being able to exploit both distributed computing and parallel computing techniques in our code. This mixed distributed-parallel paradigm is the de-facto standard nowadays when writing applications distributed over the network. As usual, reality is rarely binary.

Summary
We covered a lot of ground in this first chapter. We looked at parallelism and distributed computing. We saw some conceptual examples of both architectures and their pros and cons. We touched on their implications for memory access and noted that reality is oftentimes somewhere in between these two extremes. We finished the chapter by looking at Amdahl's law and its implications on scalability and the economics of throwing hardware at the problem. In the next chapters, we will put these concepts in practice and write some Python code!
Where to buy this book

You can buy Distributed Computing with Python from the Packt Publishing website.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

Click here for ordering and shipping details.