Chapter No. 1
"Contextualizing Parallel, Concurrent, and Distributed Programming"
In this package, you will find:

The author’s biography

A preview chapter from the book, Chapter no.1 "Contextualizing Parallel, Concurrent, and Distributed Programming"

A synopsis of the book’s content

Information on where to buy this book

About the Author

Jan Palach has been a software developer for 13 years, having worked with scientific visualization and backend for private companies using C++, Java, and Python technologies. Jan has a degree in Information Systems from Estácio de Sá University, Rio de Janeiro, Brazil, and a postgraduate degree in Software Development from Paraná State Federal Technological University. Currently, he works as a senior system analyst at a private company within the telecommunication sector implementing C++ systems; however, he likes to have fun experimenting with Python and Erlang—his two technological passions. Naturally curious, he loves challenges and learning new technologies, meeting new people, and learning about different cultures.

Acknowledgments

I had no idea how hard it could be to write a book with such a tight deadline among so many other things taking place in my life. I had to fit the writing into my routine, taking care of my family, karate lessons, work, Diablo III, and so on. The task was not easy; however, I got to the end of it hoping that I have generated quality content to please most readers, considering that I have focused on the most important thing based on my experience.

For More Information:

The list of people I would like to acknowledge is so long that I would need a book only for this. So, I would like to thank some people I have constant contact with and who, in a direct or indirect way, helped me throughout this quest.

My wife Anicieli Valeska de Miranda Pertile, the woman I chose to share my love with and gather toothbrushes with to the end of this life, who allowed me to have the time to create this book and did not let me give up when I thought I could not make it. My family has always been important to me during my growth as a human being and taught me the path of goodness.

I would like to thank Fanthiane Ketrin Wentz, who beyond being my best friend is also guiding me through the ways of martial arts, teaching me the values I will carry during a lifetime—a role model for me. Lis Marie Martini, dear friend who provided the cover for this book, and who is an incredible photographer and animal lover.

Big thanks to my former English teacher, reviser, and proofreader, Marina Melo, who helped along the writing of this book. Thanks to the reviewers and personal friends, Vitor Mazzi and Bruno Torres, who contributed a lot to my professional growth and still do.

Special thanks to Rodrigo Cacilhas, Bruno Bemfi ca, Rodrigo Delduca, Luiz Shigunov, Bruno Almeida Santos, Paulo Tesch (corujito), Luciano Palma, Felipe Cruz, and other people with whom I often talk to about technology. A special thanks to Turma B.

Big thanks to Guido Van Rossum for creating Python, which transformed programming into something pleasant; we need more of this stuff and less set/get.

For More Information:
Parallel Programming with Python

Months ago, in 2013, I was contacted by Packt Publishing professionals with the mission of writing a book about parallel programming using the Python language. I had never thought of writing a book before and had no idea of the work that was about to come; how complex it would be to conceive this piece of work and how it would feel to fit it into my work schedule within my current job. Although I thought about the idea for over a couple of days, I ended up accepting the mission and said to myself that it will be a great deal of personal learning and a perfect chance to disseminate my knowledge of Python to a worldwide audience, and thus, hopefully leave a worthy legacy along my journey in this life.

The first part of this work is to outline its topics. It is not easy to please everybody; however, I believe I have achieved a good balance in the topics proposed in this mini book, in which I intended to introduce Python parallel programming combining theory and practice. I have taken a risk in this work. I have used a new format to show how problems can be solved, in which examples are defined in the first chapters and then solved by using the tools presented along the length of the book. I think this is an interesting format as it allows the reader to analyze and question the different modules that Python offers.

All chapters combine a bit of theory, thereby building the context that will provide you with some basic knowledge to follow the practical bits of the text. I truly hope this book will be useful for those adventuring into the world of Python parallel programming, for I have tried to focus on quality writing.

What This Book Covers

Chapter 1, Contextualizing Parallel, Concurrent, and Distributed Programming, covers the concepts, advantages, disadvantages, and implications of parallel programming models. In addition, this chapter exposes some Python libraries to implement parallel solutions.

Chapter 2, Designing Parallel Algorithms, introduces a discussion about some techniques to design parallel algorithms.

Chapter 3, Identifying a Parallelizable Problem, introduces some examples of problems, and analyzes if these problems can be divided into parallel pieces.

Chapter 4, Using the threading and concurrent futures Modules, explains how to implement each problem presented in Chapter 3, Identifying a Parallelizable Problem, using the threading and concurrent futures modules.


Chapter 6, Utilizing Parallel Python, covers how to implement each problem presented in Chapter 3, Identifying a Parallelizable Problem, using the parallel Python module.

Chapter 7, Distributing Tasks with Celery, explains how to implement each problem presented in Chapter 3, Identifying a Parallelizable Problem, using the Celery distributed task queue.

Chapter 8, Doing Things Asynchronously, explains how to use the asyncio module and concepts about asynchronous programming.

For More Information:
Parallel programming can be defined as a model that aims to create programs that are compatible with environments prepared to execute code instructions simultaneously. It has not been too long since techniques of parallelism began to be used to develop software. Some years ago, processors had a single **Arithmetic Logic Unit (ALU)** among other components, which could only execute one instruction at a time during a time space. For years, only a clock that measured in hertz to determine the number of instructions a processor could process within a given interval of time was taken into consideration. The more the number of clocks, the more the instructions potentially executed in terms of KHz (thousands of operations per second), MHz (millions of operations per second), and the current GHz (billions of operations per second).

Summing up, the more instructions per cycle given to the processor, the faster the execution. During the ‘80s, a revolutionary processor came to life, **Intel 80386**, which allowed the execution of tasks in a pre-emptive manner, that is, it was possible to periodically interrupt the execution of a program to provide processor time to another program; this meant pseudo-parallelism based on **time-slicing**.

In the late ‘80s, there came **Intel 80486** that implemented a **pipelining system**, which in practice, divided the stage of execution into distinct substages. In practical terms, in a cycle of the processor, we could have different instructions being carried out simultaneously in each substage.

All the advances mentioned in the preceding section resulted in several improvements in performance, but it was not enough, as we were faced with a delicate issue that would end up as the so-called **Moore's law** (http://www.mooreslaw.org/).
The quest for high taxes of clock ended up colliding with physical limitations; processors would consume more energy, thereby generating more heat. Moreover, there was another as important issue: the market for portable computers was speeding up in the '90s. So, it was extremely important to have processors that could make the batteries of these pieces of equipment last long enough away from the plug. Several technologies and families of processors from different manufacturers were born. As regards servers and mainframes, Intel® deserves to be highlighted with its family of products Core®, which allowed to trick the operating system by simulating the existence of more than one processor even though there was a single physical chip.

In the Core® family, the processor got severe internal changes and featured components called core, which had their own ALU and caches L2 and L3, among other elements to carry out instructions. Those cores, also known as logical processors, allowed us to parallel the execution of different parts of the same program, or even different programs, simultaneously. The age core enabled lower energy use with power processing superior to its predecessors. As cores work in parallel, simulating independent processors, we can have a multi-core chip and an inferior clock, thereby getting superior performance compared to a single-core chip with higher clock, depending on the task.

So much evolution has, of course, changed the way we approach software designing. Today, we must think of parallelism to design systems that make rational use of resources without wasting them, thereby providing a better experience to the user and saving energy not only in personal computers, but also at processing centers. More than ever, parallel programming is in the developers' daily lives and, apparently, it will never go back.

This chapter covers the following topics:

- Why use parallel programming?
- Introducing the common forms of parallelization
- Communicating in parallel programming
- Identifying parallel programming problems
- Discovering Python's programming tools
- Taking care of Python Global Interpreter Lock (GIL)

For More Information:
Why use parallel programming?

Since computing systems have evolved, they have started to provide mechanisms that allow us to run independent pieces of a specific program in parallel with one another, thus enhancing the response and the general performance. Moreover, we can easily verify that the machines are equipped with more processors and these with plenty of more cores. So, why not take advantage of this architecture?

Parallel programming is a reality in all contexts of system development, from smartphones and tablets, to heavy duty computing in research centers. A solid basis in parallel programming will allow a developer to optimize the performance of an application. This results in enhancement of user experience as well as consumption of computing resources, thereby taking up less processing time for the accomplishment of complex tasks.

As an example of parallelism, let us picture a scenario in which an application that, amongst other tasks, selects information from a database, and this database has considerable size. Consider as well, the application being sequential, in which tasks must be run one after another in a logical sequence. When a user requests data, the rest of the system will be blocked until the data return is not concluded. However, making use of parallel programming, we will be allowed to create a new worker that which will seek information in this database without blocking other functions in the application, thus enhancing its use.

Exploring common forms of parallelization

There is a certain confusion when we try to define the main forms of paralleling systems. It is common to find quotations on parallel and concurrent systems as if both meant the same thing. Nevertheless, there are slight differences between them.

Within concurrent programming, we have a scenario in which a program dispatches several workers and these workers dispute to use the CPU to run a task. The stage at which the dispute takes place is controlled by the CPU scheduler, whose function is to define which worker is apt for using the resource at a specific moment. In most cases, the CPU scheduler runs the task of raking processes so fast that we might get the impression of pseudo-parallelism. Therefore, concurrent programming is an abstraction from parallel programming.

Concurrent systems dispute over the same CPU to run tasks.

For More Information:
The following diagram shows a concurrent program scheme:

![Diagram of Concurrent Programming Scheme]

Parallel programming can be defined as an approach in which program data creates workers to run specific tasks simultaneously in a **multicore environment** without the need for concurrency amongst them to access a **CPU**.

![Parallel Systems Run Tasks Simultaneously]

The following figure shows the concept of parallel systems:

![Diagram of Parallel Programming Scheme]

**Distributed programming** aims at the possibility of sharing the processing by exchanging data through messages between machines (nodes) of computing, which are physically separated.

Distributed programming is becoming more and more popular for many reasons; they are explored as follows:

---

**For More Information:**

• **Fault-tolerance**: As the system is decentralized, we can distribute the processing to different machines in a network, and thus perform individual maintenance of specific machines without affecting the functioning of the system as a whole.

• **Horizontal scalability**: We can increase the capacity of processing in distributed systems in general. We can link new equipment with no need to abort applications being executed. We can say that it is cheaper and simpler compared to vertical scalability.

• **Cloud computing**: With the reduction in hardware costs, we need the growth of this type of business where we can obtaining huge machine parks acting in a cooperative way and running programs in a transparent way for their users.

Distributed systems run tasks within physically-separated nodes.

The following figure shows a distributed system scheme:

![Distributed programming scheme](image)

**Communicating in parallel programming**

In parallel programming, the workers that are sent to perform a task often need to establish communication so that there can be cooperation in tackling a problem. In most cases, this communication is established in such a way that data can be exchanged amongst workers. There are two forms of communication that are more widely known when it comes to parallel programming: shared state and message passing. In the following sections, a brief description of both will be presented.

For More Information:
Understanding shared state

One the most well-known forms of communication amongst workers is *shared state*. Shared state seems straightforward to use but has many pitfalls because an invalid operation made to the shared resource by one of the processes will affect all of the others, thereby producing bad results. It also makes it impossible for the program to be distributed between multiple machines for obvious reasons.

Illustrating this, we will make use of a real-world case. Suppose you are a customer of a specific bank, and this bank has only one cashier. When you go to the bank, you must head to a queue and wait for your chance. Once in the queue, you notice that only one customer can make use of the cashier at a time, and it would be impossible for the cashier to attend two customers simultaneously without potentially making errors. Computing provides means to access data in a controlled way, and there are several techniques, such as *mutex*.

Mutex can be understood as a special process variable that indicates the level of availability to access data. That is, in our real-life example, the customer has a number, and at a specific moment, this number will be activated and the cashier will be available for this customer exclusively. At the end of the process, this customer will free the cashier for the next customer, and so on.

There are cases in which data has a constant value in a variable while the program is running, and the data is shared only for reading purposes. So, access control is not necessary because it will never present integrity problems.

Understanding message passing

*Message passing* is used when we aim to avoid data access control and synchronizing problems originating from shared state. Message passing consists of a mechanism for message exchange in running processes. It is very commonly used whenever we are developing programs with distributed architecture, where the message exchanges within the network they are placed are necessary. Languages such as *Erlang*, for instance, use this model to implement communication in its parallel architecture. Once data is copied at each message exchange, it is impossible that problems occur in terms of concurrence of access. Although memory use seems to be higher than in shared memory state, there are advantages to the use of this model. They are as follows:

- Absence of data access concurrence
- Messages can be exchange locally (various processes) or in distributed environments

For More Information:

• This makes it less likely that scalability issues occur and enables interoperability of different systems
• In general, it is easy to maintain according to programmers

Identifying parallel programming problems

There are classic problems that brave keyboard warriors can face while battling in the lands where parallel programming ghosts dwell. Many of these problems occur more often when inexperienced programmers make use of workers combined with shared state. Some of these issues will be described in the following sections.

Deadlock

Deadlock is a situation in which two or more workers keep indefinitely waiting for the freeing of a resource, which is blocked by a worker of the same group for some reason. For a better understanding, we will use another real-life case. Imagine the bank whose entrance has a rotating door. Customer A heads to the side, which will allow him to enter the bank, while customer B tries to exit the bank by using the entrance side of this rotating door so that both customers would be stuck forcing the door but heading nowhere. This situation would be hilarious in real life but tragic in programming.

Deadlock is a phenomenon in which processes wait for a condition to free their tasks, but this condition will never occur.

Starvation

This is the issue whose side effects are caused by unfair raking of one or more processes that take much more time to run a task. Imagine a group of processes, A, which runs heavy tasks and has data processor priority. Now, imagine that a process A with high priority constantly consumes the CPU, while a lower priority process B never gets the chance. Hence, one can say that process B is starving for CPU cycles.

Starvation is caused by badly adjusted policies of process ranking.

For More Information:
Race conditions
When the result of a process depends on a sequence of facts, and this sequence is broken due to the lack of synchronizing mechanisms, we face race conditions. They result from problems that are extremely difficult to filter in larger systems. For instance, a couple has a joint account; the initial balance before operations is $100. The following table shows the regular case, in which there are mechanisms of protection and the sequence of expected facts, as well as the result:

<table>
<thead>
<tr>
<th>Husband</th>
<th>Wife</th>
<th>Account balance (dollars)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>100</td>
<td></td>
</tr>
<tr>
<td>Read balance</td>
<td>100</td>
<td></td>
</tr>
<tr>
<td>Adds 20</td>
<td>100</td>
<td></td>
</tr>
<tr>
<td>Concludes operation</td>
<td>120</td>
<td></td>
</tr>
<tr>
<td></td>
<td>120</td>
<td></td>
</tr>
<tr>
<td>Read balance</td>
<td>120</td>
<td></td>
</tr>
<tr>
<td>Withdraws 10</td>
<td>120</td>
<td></td>
</tr>
<tr>
<td>Concludes operation</td>
<td>110</td>
<td></td>
</tr>
</tbody>
</table>

Presents baking operations without the chance of race conditions occurrence

In the following table, the problematic scenario is presented. Suppose that the account does not have mechanisms of synchronization and the order of operations is not as expected.

<table>
<thead>
<tr>
<th>Husband</th>
<th>Wife</th>
<th>Account balance (dollars)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>100</td>
<td></td>
</tr>
<tr>
<td>Read balance</td>
<td>100</td>
<td></td>
</tr>
<tr>
<td>Withdraws 100</td>
<td>100</td>
<td></td>
</tr>
<tr>
<td></td>
<td>100</td>
<td></td>
</tr>
<tr>
<td>Reads balance</td>
<td>100</td>
<td></td>
</tr>
<tr>
<td>Withdraws 10</td>
<td>100</td>
<td></td>
</tr>
<tr>
<td>Concludes operation</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>updating balance</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>90</td>
<td></td>
</tr>
</tbody>
</table>

Analogy to balance the problem in a joint account and race conditions

There is a noticeable inconsistency in the final result due to the unexpected lack of synchronization in the operations sequence. One of the parallel programming characteristics is non-determinism. It is impossible to foresee the moment at which two workers will be running, or even which of them will run first. Therefore, synchronization mechanisms are essential.

For More Information:
Non-determinism, if combined with lack of synchronization mechanisms, may lead to race condition issues.

Discovering Python's parallel programming tools

The Python language, created by Guido Van Rossum, is a multi-paradigm, multi-purpose language. It has been widely accepted worldwide due to its powerful simplicity and easy maintenance. It is also known as the language that has batteries included. There is a wide range of modules to make its use smoother. Within parallel programming, Python has built-in and external modules that simplify implementation. This work is based on Python 3.x.

The Python threading module

The Python threading module offers a layer of abstraction to the module _thread, which is a lower-level module. It provides functions that help the programmer during the hard task of developing parallel systems based on threads. The threading module's official papers can be found at http://docs.python.org/3/library/threading.html?highlight=threading#module-threading.

The Python multiprocessing module

The multiprocessing module aims at providing a simple API for the use of parallelism based on processes. This module is similar to the threading module, which simplifies alternations between the processes without major difficulties. The approach that is based on processes is very popular within the Python users' community as it is an alternative to answering questions on the use of CPU-Bound threads and GIL present in Python. The multiprocessing module's official papers can be found at http://docs.python.org/3/library/multiprocessing.html?highlight=multiprocessing#multiprocessing.

For More Information:

The parallel Python module

The parallel Python module is external and offers a rich API for the creation of parallel and distributed systems making use of the processes approach. This module promises to be light and easy to install, and integrates with other Python programs. The parallel Python module can be found at http://parallelpython.com. Among some of the features, we may highlight the following:

- Automatic detection of the optimal configuration
- The fact that a number of worker processes can be changed during runtime
- Dynamic load balance
- Fault tolerance
- Auto-discovery of computational resources

Celery – a distributed task queue

Celery is an excellent Python module that's used to create distributed systems and has excellent documentation. It makes use of at least three different types of approach to run tasks in concurrent form—multiprocessing, Eventlet, and Gevent. This work will, however, concentrate efforts on the use of the multiprocessing approach. Also, the link between one and another is a configuration issue, and it remains as a study so that the reader is able to establish comparisons with his/her own experiments.

The Celery module can be obtained on the official project page at http://celeryproject.org.

Taking care of Python GIL

GIL is a mechanism that is used in implementing standard Python, known as CPython, to avoid bytecodes that are executed simultaneously by different threads. The existence of GIL in Python is a reason for fiery discussion amongst users of this language. GIL was chosen to protect the internal memory used by the CPython interpreter, which does not implement mechanisms of synchronization for the concurrent access by threads. In any case, GIL results in a problem when we decide to use threads, and these tend to be CPU-bound. I/O Threads, for example, are out of GIL’s scope. Maybe the mechanism brings more benefits to the evolution of Python than harm to it. Evidently, we could not consider only speed as a single argument to determine whether something is good or not.

For More Information:
There are cases in which the approach to the use of processes for tasks sided with message passing brings better relations among maintainability, scalability, and performance. Even so, there are cases in which there will be a real need for threads, which would be subdued to GIL. In these cases, what could be done is write such pieces of code as extensions in C language, and embed them into the Python program. Thus, there are alternatives; it is up to the developer to analyze the real necessity. So, there comes the question: is GIL, in a general way, a villain? It is important to remember that, the PyPy team is working on an STM implementation in order to remove GIL from Python. For more details about the project, visit http://pypy.org/tmdonate.html.

Summary
In this chapter, we learned some parallel programming concepts, and learned about some models, their advantages, and disadvantages. Some of the problems and potential issues when thinking of parallelism have been presented in a brief explanations. We also had a short introduction to some Python modules, built-in and external, which makes a developer’s life easier when building up parallel systems.

In the next chapter, we will be studying some techniques to design parallel algorithms.
Where to buy this book


Free shipping to the US, UK, Europe and selected Asian countries. For more information, please read our shipping policy.

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