

1

Getting Started with Parallel Computing and Python

In this chapter, we will cover the following recipes:

- ▶ What is parallel computing?
- ▶ The parallel computing memory architecture
- ▶ Memory organization
- ▶ Parallel programming models
- ▶ How to design a parallel program
- ▶ How to evaluate the performance of a parallel program
- ▶ Introducing Python
- ▶ Python in a parallel world
- ▶ Introducing processes and threads
- ▶ Start working with processes and Python
- ▶ Start working with threads and Python

Introduction

This chapter gives you an overview of parallel programming architectures and programming models. These concepts are useful for inexperienced programmers who have approached parallel programming techniques for the first time. This chapter can be a basic reference for the experienced programmers. The dual characterization of parallel systems is also presented in this chapter. The first characterization is based on the architecture of the system and the second characterization is based on parallel programming paradigms. Parallel programming will always be a challenge for programmers. This programming-based approach is further described in this chapter, when we present the design procedure of a parallel program. The chapter ends with a brief introduction of the Python programming language. The characteristics of the language, ease of use and learning, and extensibility and richness of software libraries and applications make Python a valuable tool for any application and also, of course, for parallel computing. In the final part of the chapter, the concepts of threads and processes are introduced in relation to their use in the language. A typical way to solve a problem of a large-size is to divide it into smaller and independent parts in order to solve all the pieces simultaneously. A parallel program is intended for a program that uses this approach, that is, the use of multiple processors working together on a common task. Each processor works on its section (the independent part) of the problem. Furthermore, a data information exchange between processors could take place during the computation. Nowadays, many software applications require more computing power. One way to achieve this is to increase the clock speed of the processor or to increase the number of processing cores on the chip. Improving the clock speed increases the heat dissipation, thereby decreasing the performance per watt and moreover, this requires special equipment for cooling. Increasing the number of cores seems to be a feasible solution, as power consumption and dissipation are way under the limit and there is no significant gain in performance.

To address this problem, computer hardware vendors decided to adopt multi-core architectures, which are single chips that contain two or more processors (cores). On the other hand, the GPU manufactures also introduced hardware architectures based on multiple computing cores. In fact, today's computers are almost always present in multiple and heterogeneous computing units, each formed by a variable number of cores, for example, the most common multi-core architectures.

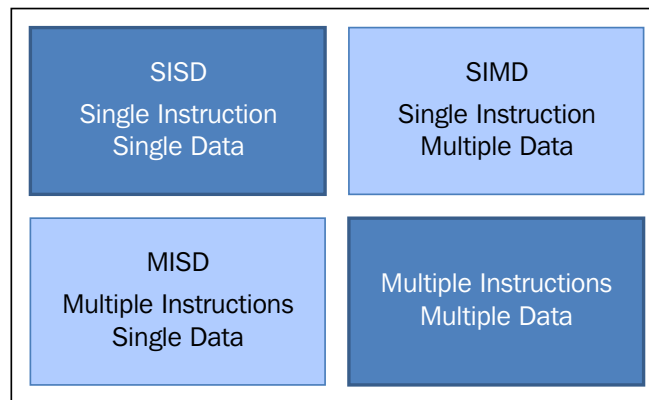
Therefore, it became essential for us to take advantage of the computational resources available, to adopt programming paradigms, techniques, and instruments of parallel computing.

The parallel computing memory architecture

Based on the number of instructions and data that can be processed simultaneously, computer systems are classified into four categories:

- ▶ **Single instruction, single data (SISD)**
- ▶ **Single instruction, multiple data (SIMD)**
- ▶ **Multiple instruction, single data (MISD)**
- ▶ **Multiple instruction, multiple data (MIMD)**

This classification is known as Flynn's taxonomy.



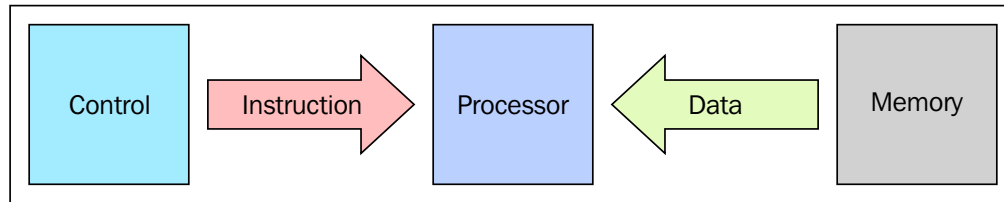
SISD

The SISD computing system is a uniprocessor machine. It executes a single instruction that operates on a single data stream. In SISD, machine instructions are processed sequentially.

In a clock cycle, the CPU executes the following operations:

- ▶ **Fetch:** The CPU fetches the data and instructions from a memory area, which is called a register.
- ▶ **Decode:** The CPU decodes the instructions.
- ▶ **Execute:** The instruction is carried out on the data. The result of the operation is stored in another register.

Once the execution stage is complete, the CPU sets itself to begin another CPU cycle.



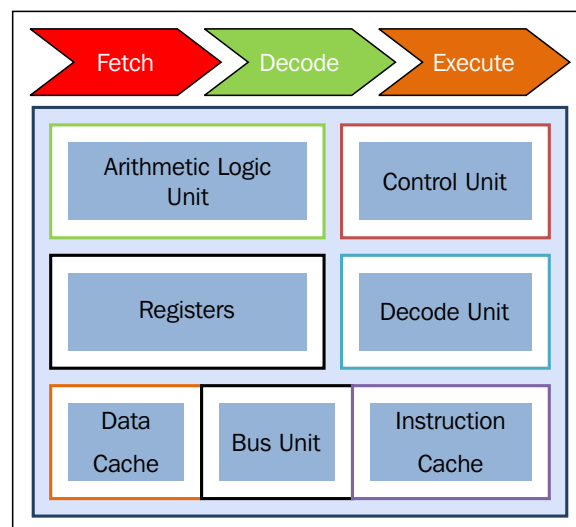
The SISD architecture schema

The algorithms that run on these types of computers are sequential (or serial), since they do not contain any parallelism. Examples of SISD computers are hardware systems with a single CPU.

The main elements of these architectures (Von Neumann architectures) are:

- ▶ **Central memory unit:** This is used to store both instructions and program data
- ▶ **CPU:** This is used to get the instruction and/or data from the memory unit, which decodes the instructions and sequentially implements them
- ▶ **The I/O system:** This refers to the input data and output data of the program

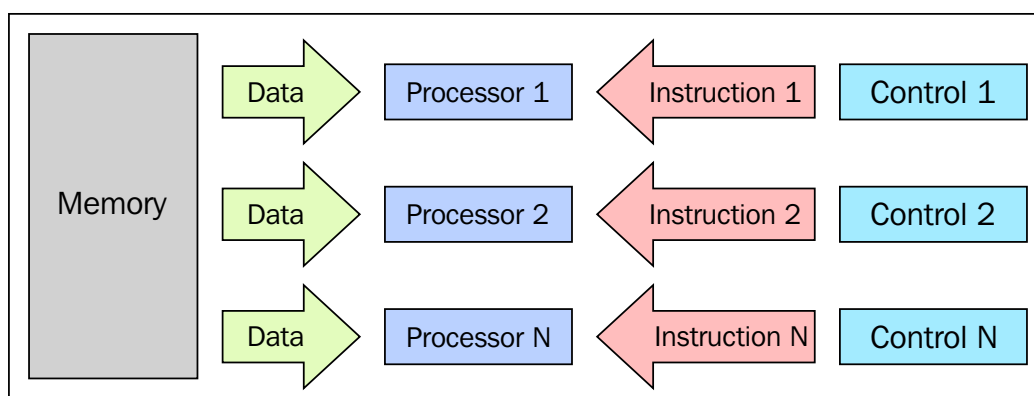
The conventional single processor computers are classified as SISD systems. The following figure specifically shows which areas of a CPU are used in the stages of fetch, decode, and execute:



CPU's components in the fetch-decode-execute phase

MISD

In this model, n processors, each with their own control unit, share a single memory unit. In each clock cycle, the data received from the memory is processed by all processors simultaneously, each in accordance with the instructions received from its control unit. In this case, the parallelism (instruction-level parallelism) is obtained by performing several operations on the same piece of data. The types of problems that can be solved efficiently in these architectures are rather special, such as those regarding data encryption; for this reason, the computer MISD did not find space in the commercial sector. MISD computers are more of an intellectual exercise than a practical configuration.



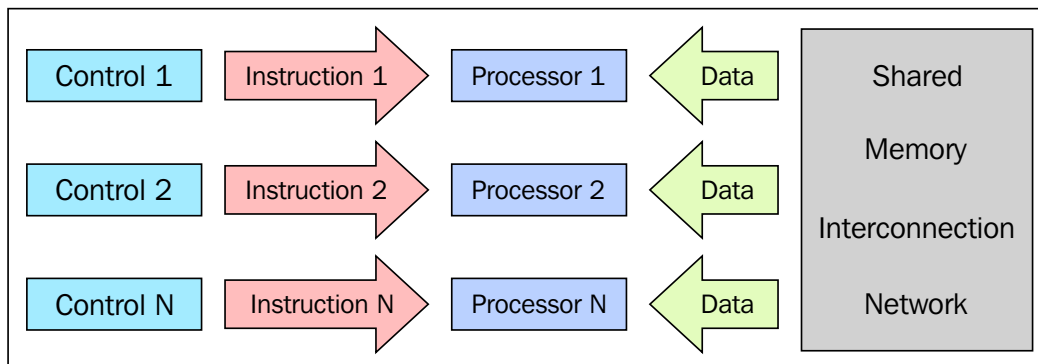
The MISD architecture scheme

SIMD

A SIMD computer consists of n identical processors, each with its own local memory, where it is possible to store data. All processors work under the control of a single instruction stream; in addition to this, there are n data streams, one for each processor. The processors work simultaneously on each step and execute the same instruction, but on different data elements. This is an example of data-level parallelism. The SIMD architectures are much more versatile than MISD architectures. Numerous problems covering a wide range of applications can be solved by parallel algorithms on SIMD computers. Another interesting feature is that the algorithms for these computers are relatively easy to design, analyze, and implement. The limit is that only the problems that can be divided into a number of subproblems (which are all identical, each of which will then be solved contemporaneously, through the same set of instructions) can be addressed with the SIMD computer. With the supercomputer developed according to this paradigm, we must mention the Connection Machine (1985 Thinking Machine) and MPP (NASA - 1983). As we will see in *Chapter 6, GPU Programming with Python*, the advent of modern **graphics processor unit (GPU)**, built with many SIMD embedded units has lead to a more widespread use of this computational paradigm.

MIMD

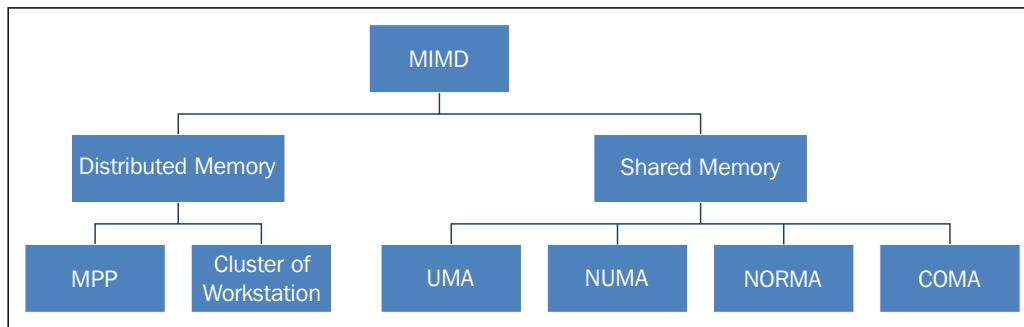
This class of parallel computers is the most general and more powerful class according to Flynn's classification. There are n processors, n instruction streams, and n data streams in this. Each processor has its own control unit and local memory, which makes MIMD architectures more computationally powerful than those used in SIMD. Each processor operates under the control of a flow of instructions issued by its own control unit; therefore, the processors can potentially run different programs on different data, solving subproblems that are different and can be a part of a single larger problem. In MIMD, architecture is achieved with the help of the parallelism level with threads and/or processes. This also means that the processors usually operate asynchronously. The computers in this class are used to solve those problems that do not have a regular structure that is required by the model SIMD. Nowadays, this architecture is applied to many PCs, supercomputers, and computer networks. However, there is a counter that you need to consider: asynchronous algorithms are difficult to design, analyze, and implement.



The MIMD architecture scheme

Memory organization

Another aspect that we need to consider to evaluate a parallel architecture is memory organization or rather, the way in which the data is accessed. No matter how fast the processing unit is, if the memory cannot maintain and provide instructions and data at a sufficient speed, there will be no improvement in performance. The main problem that must be overcome to make the response time of the memory compatible with the speed of the processor is the memory cycle time, which is defined as the time that has elapsed between two successive operations. The cycle time of the processor is typically much shorter than the cycle time of the memory. When the processor starts transferring data (to or from the memory), the memory will remain occupied for the entire time of the memory cycle: during this period, no other device (I/O controller, processor, or even the processor itself that made the request) can use the memory because it will be committed to respond to the request.

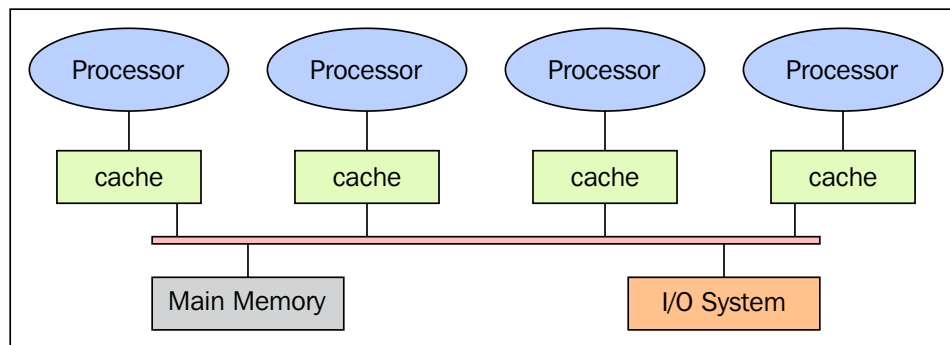


The memory organization in MIMD architecture

Solutions to the problem of access memory resulted in a dichotomy of MIMD architectures. In the first type of system, known as the shared memory system, there is high virtual memory and all processors have equal access to data and instructions in this memory. The other type of system is the distributed memory model, wherein each processor has a local memory that is not accessible to other processors. The difference between shared memory and distributed memory lies in the structure of the virtual memory or the memory from the perspective of the processor. Physically, almost every system memory is divided into distinct components that are independently accessible. What distinguishes a shared memory from a distributed memory is the memory access management by the processing unit. If a processor were to execute the instruction $R0, i$, which means load in the $R0$ register the contents of the memory location i , the question now is what should happen? In a system with shared memory, the i index is a global address and the memory location i is the same for each processor. If two processors were to perform this instruction at the same time, they would load the same information in their registers $R0$. In a distributed memory system, i is a local address. If two processors were to load the statement $R0$ at the same time, different values may end up in the respective register's $R0$, since, in this case, the memory cells are allotted one for each local memory. The distinction between shared memory and distributed memory is very important for programmers because it determines the way in which different parts of a parallel program must communicate. In a system, shared memory is sufficient to build a data structure in memory and go to the parallel subroutine, which are the reference variables of this data structure. Moreover, a distributed memory machine must make copies of shared data in each local memory. These copies are created by sending a message containing the data to be shared from one processor to another. A drawback of this memory organization is that sometimes, these messages can be very large and take a relatively long transfer time.

Shared memory

The schema of a shared memory multiprocessor system is shown in the following figure. The physical connections here are quite simple. The bus structure allows an arbitrary number of devices that share the same channel. The bus protocols were originally designed to allow a single processor, and one or more disks or tape controllers to communicate through the shared memory here. Note that each processor has been associated with a cache memory, as it is assumed that the probability that a processor needs data or instructions present in the local memory is very high. The problem occurs when a processor modifies data stored in the memory system that is simultaneously used by other processors. The new value will pass from the processor cache that has been changed to shared memory; later, however, it must also be passed to all the other processors, so that they do not work with the obsolete value. This problem is known as the problem of cache coherency, a special case of the problem of memory consistency, which requires hardware implementations that can handle concurrency issues and synchronization similar to those having thread programming.



The shared memory architecture schema

The main features of shared memory systems are:

- ▶ The memory is the same for all processors, for example, all the processors associated with the same data structure will work with the same logical memory addresses, thus accessing the same memory locations.
- ▶ The synchronization is made possible by controlling the access of processors to the shared memory. In fact, only one processor at a time can have access to the memory resources.

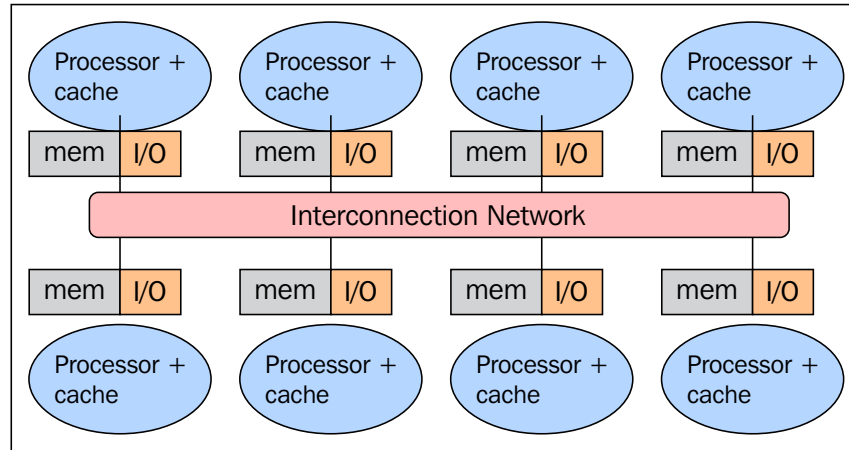
- ▶ A shared memory location must not be changed from a task while another task accesses it.
- ▶ Sharing data is fast; the time required for the communication between two tasks is equal to the time for reading a single memory location (it is depending on the speed of memory access).

The memory access in shared memory systems are as follows:

- ▶ **Uniform memory access (UMA):** The fundamental characteristic of this system is the access time to the memory that is constant for each processor and for any area of memory. For this reason, these systems are also called as **symmetric multiprocessor (SMP)**. They are relatively simple to implement, but not very scalable; the programmer is responsible for the management of the synchronization by inserting appropriate controls, semaphores, locks, and so on in the program that manages resources.
- ▶ **Non-uniform memory access (NUMA):** These architectures divide the memory area into a high-speed access area that is assigned to each processor and a common area for the data exchange, with slower access. These systems are also called as **Distributed Shared Memory Systems (DSM)**. They are very scalable, but complex to develop.
- ▶ **No remote memory access (NORMA):** The memory is physically distributed among the processors (local memory). All local memories are private and can only access the local processor. The communication between the processors is through a communication protocol used for exchange of messages, the message-passing protocol.
- ▶ **Cache only memory access (COMA):** These systems are equipped with only cache memories. While analyzing NUMA architectures, it was noticed that these architectures kept the local copies of the data in the cache and that these data were stored as duplication in the main memory. This architecture removes duplicates and keeps only the cache memories, the memory is physically distributed among the processors (local memory). All local memories are private and can only access the local processor. The communication between the processors is through a communication protocol for exchange of messages, the message-passing protocol.

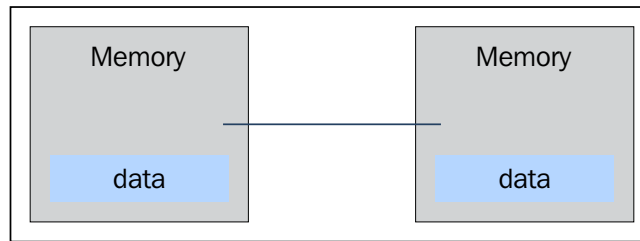
Distributed memory

In a system with distributed memory, the memory is associated with each processor and a processor is only able to address its own memory. Some authors refer to this type of system as "multicomputer", reflecting the fact that the elements of the system are themselves small complete systems of a processor and memory, as you can see in the following figure:



The distributed memory architecture scheme

This kind of organization has several advantages. At first, there are no conflicts at the level of the communication bus or switch. Each processor can use the full bandwidth of their own local memory without any interference from other processors. Secondly, the lack of a common bus means that there is no intrinsic limit to the number of processors, the size of the system is only limited by the network used to connect the processors. Thirdly, there are no problems of cache coherency. Each processor is responsible for its own data and does not have to worry about upgrading any copies. The main disadvantage is that the communication between processors is more difficult to implement. If a processor requires data in the memory of another processor, the two processors should necessarily exchange messages via the message-passing protocol. This introduces two sources of slowdown; to build and send a message from one processor to another takes time, and also, any processor should be stopped in order to manage the messages received from other processors. A program designed to work on a distributed memory machine must be organized as a set of independent tasks that communicate via messages.



Basic message passing

The main features of distributed memory systems are as follows:

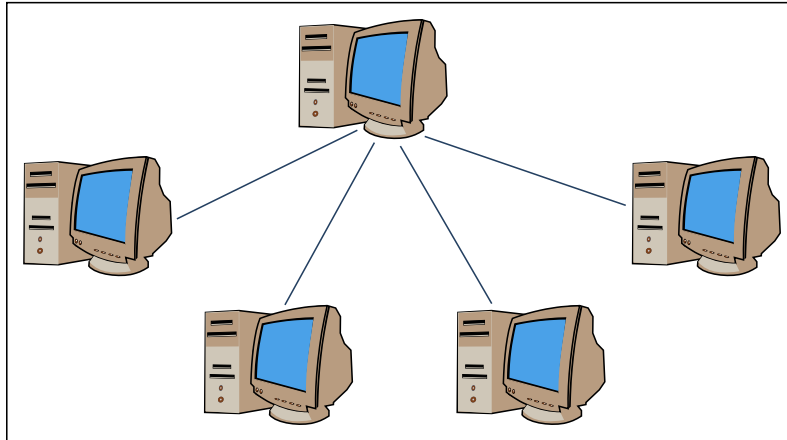
- ▶ Memory is physically distributed between processors; each local memory is directly accessible only by its processor.
- ▶ Synchronization is achieved by moving data (even if it's just the message itself) between processors (communication).
- ▶ The subdivision of data in the local memories affects the performance of the machine—it is essential to make a subdivision accurate, so as to minimize the communication between the CPUs. In addition to this, the processor that coordinates these operations of decomposition and composition must effectively communicate with the processors that operate on the individual parts of data structures.
- ▶ The message-passing protocol is used so that the CPU's can communicate with each other through the exchange of data packets. The messages are discrete units of information; in the sense that they have a well-defined identity, so it is always possible to distinguish them from each other.

Massively parallel processing

MPP machines are composed of hundreds of processors (which can be as large as hundreds of thousands in some machines) that are connected by a communication network. The fastest computers in the world are based on these architectures; some example systems of these architectures are: Earth Simulator, Blue Gene, ASCI White, ASCI Red, and ASCI Purple and Red Storm.

A cluster of workstations

These processing systems are based on classical computers that are connected by communication networks. The computational clusters fall into this classification.



An example of a cluster of workstation architecture

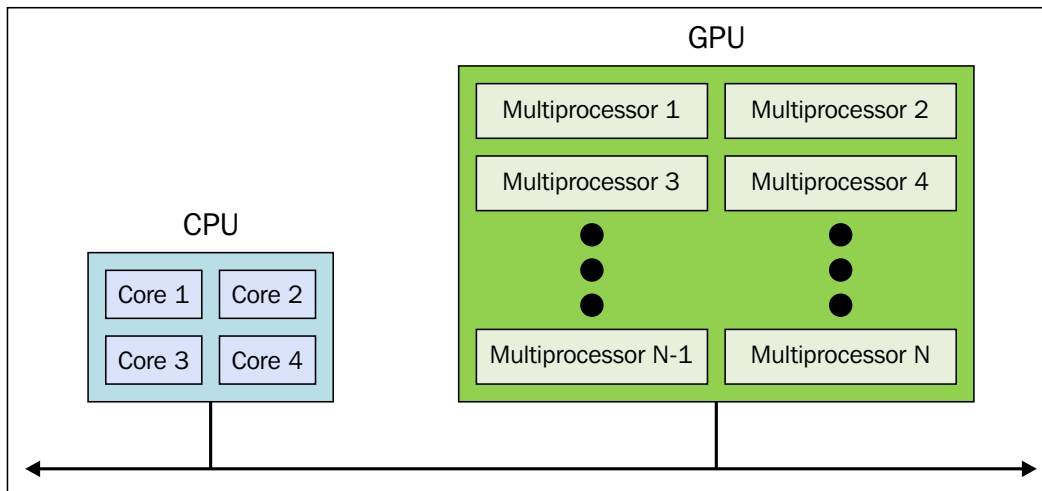
In a cluster architecture, we define a node as a single computing unit that takes part in the cluster. For the user, the cluster is fully transparent—all the hardware and software complexity is masked and data and applications are made accessible as if they were all from a single node.

Here, we've identified three types of clusters:

- ▶ **The fail-over cluster:** In this, the node's activity is continuously monitored, and when one stops working, another machine takes over the charge of those activities. The aim is to ensure a continuous service due to the redundancy of the architecture.
- ▶ **The load balancing cluster:** In this system, a job request is sent to the node that has less activity. This ensures that less time is taken to complete the process.
- ▶ **The high-performance computing cluster:** In this, each node is configured to provide extremely high performance. The process is also divided in multiple jobs on multiple nodes. The jobs are parallelized and will be distributed to different machines.

The heterogeneous architecture

The introduction of GPU accelerators in the homogeneous world of supercomputing has changed the nature of how supercomputers were both used and programmed previously. Despite the high performance offered by GPUs, they cannot be considered as an autonomous processing unit as they should always be accompanied by a combination of CPUs. The programming paradigm, therefore, is very simple; the CPU takes control and computes in a serial manner, assigning to the graphic accelerator the tasks that are computationally very expensive and have a high degree of parallelism. The communication between a CPU and GPU can take place not only through the use of a high-speed bus, but also through the sharing of a single area of memory for both physical or virtual. In fact, in the case where both the devices are not equipped with their own memory areas, it is possible to refer to a common memory area using the software libraries provided by the various programming models, such as CUDA and OpenCL. These architectures are called heterogeneous architectures, wherein applications can create data structures in a single address space and send a job to the device hardware appropriate for the resolution of the task. Several processing tasks can operate safely on the same regions to avoid data consistency problems, thanks to the atomic operations. So, despite the fact that the CPU and GPU do not seem to work efficiently together, with the use of this new architecture, we can optimize their interaction with and performance of parallel applications.



The heterogeneous architecture scheme

Parallel programming models

Parallel programming models exist as an abstraction of hardware and memory architectures. In fact, these models are not specific and do not refer to particular types of machines or memory architectures. They can be implemented (at least theoretically) on any kind of machines. Compared to the previous subdivisions, these programming models are made at a higher level and represent the way in which the software must be implemented to perform a parallel computation. Each model has its own way of sharing information with other processors in order to access memory and divide the work.

There is no better programming model in absolute terms; the best one to apply will depend very much on the problem that a programmer should address and resolve. The most widely used models for parallel programming are:

- ▶ The shared memory model
- ▶ The multithread model
- ▶ The distributed memory/message passing model
- ▶ The data parallel model

In this recipe, we will give you an overview of these models. A more accurate description will be in the next chapters that will introduce you to the appropriate Python module that implements these.

The shared memory model

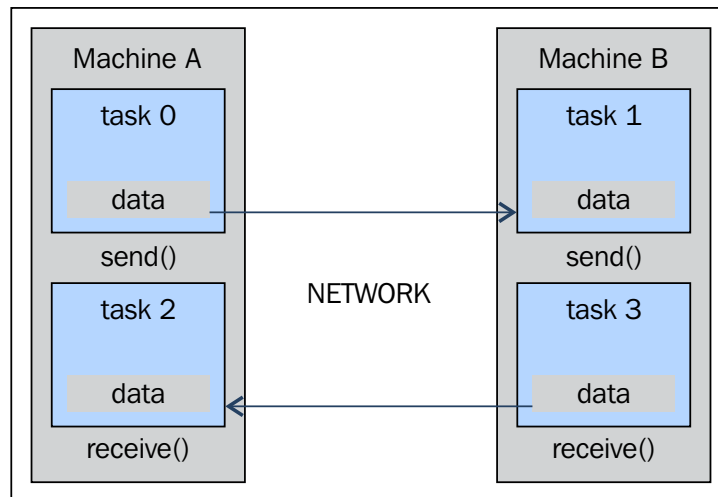
In this model the tasks share a single shared memory area, where the access (reading and writing data) to shared resources is asynchronous. There are mechanisms that allow the programmer to control the access to the shared memory, for example, locks or semaphores. This model offers the advantage that the programmer does not have to clarify the communication between tasks. An important disadvantage in terms of performance is that it becomes more difficult to understand and manage data locality; keeping data local to the processor that works on it conserves memory accesses, cache refreshes, and bus traffic that occur when multiple processors use the same data.

The multithread model

In this model, a process can have multiple flows of execution, for example, a sequential part is created and subsequently, a series of tasks are created that can be executed parallelly. Usually, this type of model is used on shared memory architectures. So, it will be very important for us to manage the synchronization between threads, as they operate on shared memory, and the programmer must prevent multiple threads from updating the same locations at the same time. The current generation CPUs are multithreaded in software and hardware. Posix threads are the classic example of the implementation of multithreading on software. The Intel Hyper-threading technology implements multithreading on hardware by switching between two threads when one is stalled or waiting on I/O. Parallelism can be achieved from this model even if the data alignment is nonlinear.

The message passing model

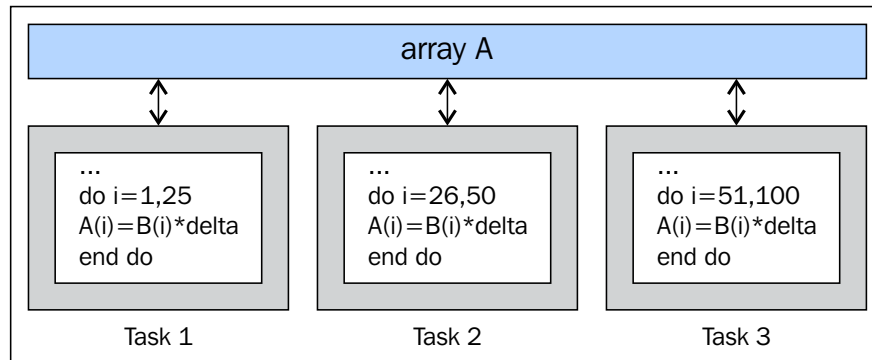
The message passing model is usually applied in the case where each processor has its own memory (distributed memory systems). More tasks can reside on the same physical machine or on an arbitrary number of machines. The programmer is responsible for determining the parallelism and data exchange that occurs through the messages. The implementation of this parallel programming model requires the use of (ad hoc) software libraries to be used within the code. Numerous implementations of message passing model were created: some of the examples are available since the 1980s, but only from the mid-90s, was created to standardized model, coming to a de facto standard called MPI (the message passing interface). The MPI model is designed clearly with distributed memory, but being models of parallel programming, multiplatform can also be used with a shared memory machine.



The message passing paradigm model

The data parallel model

In this model, we have more tasks that operate on the same data structure, but each task operates on a different portion of data. In the shared memory architecture, all tasks have access to data through shared memory and distributed memory architectures, where the data structure is divided and resides in the local memory of each task. To implement this model, a programmer must develop a program that specifies the distribution and alignment of data. The current generation GPUs operate high throughput with the data aligned.



The data parallel paradigm model

How to design a parallel program

The design of algorithms that exploit parallelism is based on a series of operations, which must necessarily be carried out for the program to perform the job correctly without producing partial or erroneous results. The macro operations that must be carried out for a correct parallelization of an algorithm are:

- ▶ Task decomposition
- ▶ Task assignment
- ▶ Agglomeration
- ▶ Mapping

Task decomposition

In this first phase, the software program is split into tasks or a set of instructions that can then be executed on different processors to implement parallelism. To do this subdivision, there are two methods that are used:

- ▶ **Domain decomposition:** Here, the data of the problems is decomposed; the application is common to all the processors that work on a different portion of data. This methodology is used when we have a large amount of data that must be processed.
- ▶ **Functional decomposition:** In this case, the problem is split into tasks, where each task will perform a particular operation on all the available data.

Task assignment

In this step, the mechanism by which the task will be distributed among the various processes is specified. This phase is very important because it establishes the distribution of workload among the various processors. The load balance is crucial here; in fact, all processors must work with continuity, avoiding an idle state for a long time. To perform this, the programmer takes into account the possible heterogeneity of the system that tries to assign more tasks to better performing processors. Finally, for greater efficiency of parallelization, it is necessary to limit communication as much as possible between processors, as they are often the source of slowdowns and consumption of resources.

Agglomeration

Agglomeration is the process of combining smaller tasks with larger ones in order to improve performance. If the previous two stages of the design process partitioned the problem into a number of tasks that greatly exceed the number of processors available, and if the computer is not specifically designed to handle a huge number of small tasks (some architectures, such as GPUs, handle this fine and indeed benefit from running millions or even billions of tasks), then the design can turn out to be highly inefficient. Commonly, this is because tasks have to be communicated to the processor or thread so that they compute the said task. Most communication has costs that are not only proportional with the amount of data transferred, but also incur a fixed cost for every communication operation (such as the latency which is inherent in setting up a TCP connection). If the tasks are too small, this fixed cost can easily make the design inefficient.

Mapping

In the mapping stage of the parallel algorithm design process, we specify where each task is to be executed. The goal is to minimize the total execution time. Here, you must often make tradeoffs, as the two main strategies often conflict with each other:

- ▶ The tasks that communicate frequently should be placed in the same processor to increase locality
- ▶ The tasks that can be executed concurrently should be placed in different processors to enhance concurrency

This is known as the mapping problem, and it is known to be NP-complete. As such, no polynomial time solutions to the problem in the general case exist. For tasks of equal size and tasks with easily identified communication patterns, the mapping is straightforward (we can also perform agglomeration here to combine tasks that map to the same processor.) However, if the tasks have communication patterns that are hard to predict or the amount of work varies per task, it is hard to design an efficient mapping and agglomeration scheme. For these types of problems, load balancing algorithms can be used to identify agglomeration and mapping strategies during runtime. The hardest problems are those in which the amount of communication or the number of tasks changes during the execution of the program. For these kind of problems, dynamic load balancing algorithms can be used, which run periodically during the execution.

Dynamic mapping

There exists many load balancing algorithms for various problems, both global and local. Global algorithms require global knowledge of the computation being performed, which often adds a lot of overhead. Local algorithms rely only on information that is local to the task in question, which reduces overhead compared to global algorithms, but are usually worse at finding an optimal agglomeration and mapping. However, the reduced overhead may reduce the execution time even though the mapping is worse by itself. If the tasks rarely communicate other than at the start and end of the execution, a task-scheduling algorithm is often used that simply maps tasks to processors as they become idle. In a task-scheduling algorithm, a task pool is maintained. Tasks are placed in this pool and are taken from it by workers.

There are three common approaches in this model, which are explained next.

Manager/worker

This is the basic dynamic mapping scheme in which all the workers connect to a the centralized manager. The manager repeatedly sends tasks to the workers and collects the results. This strategy is probably the best for a relatively small number of processors. The basic strategy can be improved by fetching tasks in advance so that communication and computation overlap each other.

Hierarchical manager/worker

This is the variant of a manager/worker that has a semi-distributed layout; workers are split into groups, each with their own manager. These group managers communicate with the central manager (and possibly, among themselves as well), while workers request tasks from the group managers. This spreads the load among several managers and can, as such, handle a larger amount of processors if all workers request tasks from the same manager.

Decentralize

In this scheme, everything is decentralized. Each processor maintains its own task pool and communicates with the other processors in order to request tasks. How the processors choose other processors to request tasks varies and is determined on the basis of the problem.

How to evaluate the performance of a parallel program

The development of parallel programming created the need of performance metrics and a software tool to evaluate the performance of a parallel algorithm in order to decide whether its use is convenient or not. Indeed, the focus of parallel computing is to solve large problems in a relatively short time. The factors that contribute to the achievement of this objective are, for example, the type of hardware used, the degree of parallelism of the problem, and which parallel programming model is adopted. To facilitate this, analysis of basic concepts was introduced, which compares the parallel algorithm obtained from the original sequence. The performance is achieved by analyzing and quantifying the number of threads and/or the number of processes used.

To analyze this, a few performance indexes are introduced: speedup, efficiency, and scaling.

The limitations of a parallel computation are introduced by the Ahmdal's law to evaluate the degree of the efficiency of parallelization of a sequential algorithm we have the Gustafson's law.

Speedup

Speedup is the measure that displays the benefit of solving a problem in parallel. It is defined as the ratio of the time taken to solve a problem on a single processing element, T_s , to the time required to solve the same problem on p identical processing elements, T_p .

$$S = \frac{T_s}{T_p}$$

We denote speedup by S . We have a linear speedup, where if $S=p$, it means that the speed of execution increases with the number of processors. Of course, this is an ideal case. While the speedup is absolute when T_s is the execution time of the best sequential algorithm, the speedup is relative when T_s is the execution time of the parallel algorithm for a single processor.

Let's recap these conditions:

- ▶ $S = p$ is linear or ideal speedup
- ▶ $S < p$ is real speedup
- ▶ $S > p$ is superlinear speedup

Efficiency

In an ideal world, a parallel system with p processing elements can give us a speedup equal to p . However, this is very rarely achieved. Usually, some time is wasted in either idling or communicating. Efficiency is a performance metric estimating how well-utilized the processors are in solving a task, compared to how much effort is wasted in communication and synchronization.

$$E = \frac{S}{p} = \frac{T_s}{pT_p}$$

We denote it by E and can define it as $E = \frac{S}{p} = \frac{T_s}{pT_p}$. The algorithms with linear speedup have the value of $E = 1$; in other cases, the value of E is less than 1. The three cases are identified as follows:

- ▶ When $E = 1$, it is a linear case
- ▶ When $E < 1$, it is a real case
- ▶ When $E \ll 1$, it is a problem that is parallelizable with low efficiency

Scaling

Scaling is defined as the ability to be efficient on a parallel machine. It identifies the computing power (speed of execution) in proportion with the number of processors. By increasing the size of the problem and at the same time the number of processors, there will be no loss in terms of performance. The scalable system, depending on the increments of the different factors, may maintain the same efficiency or improve it.

Amdahl's law

Amdahl's law is a widely used law used to design processors and parallel algorithms. It states that the maximum speedup that can be achieved is limited by the serial component of the

program: $S = \frac{1}{1-P}$, where $1 - P$ denotes the serial component (not parallelized) of a program. This means that for, as an example, a program in which 90 percent of the code can be made parallel, but 10 percent must remain serial, the maximum achievable speedup is 9 even for an infinite number of processors.

Gustafson's law

Gustafson's law is based on the following considerations:

- ▶ While increasing the dimension of a problem, its sequential parts remain constant
- ▶ While increasing the number of processors, the work required on each of them still remains the same

This states that $S(P) = P - \alpha (P - 1)$, where P is the number of processors, S is the speedup, and α is the non-parallelizable fraction of any parallel process. This is in contrast to Amdahl's law, which takes the single-process execution time to be the fixed quantity and compares it to a shrinking per process parallel execution time. Thus, Amdahl's law is based on the assumption of a fixed problem size; it assumes that the overall workload of a program does not change with respect to the machine size (that is, the number of processors). Gustafson's law addresses the deficiency of Amdahl's law, which does not take into account the total number of computing resources involved in solving a task. It suggests that the best way to set the time allowed for the solution of a parallel problem is to consider all the computing resources and on the basis of this information, it fixes the problem.

Introducing Python

Python is a powerful, dynamic, and interpreted programming language that is used in a wide variety of applications. Some of its features include:

- ▶ A clear and readable syntax
- ▶ A very extensive standard library, where through additional software modules, we can add data types, functions, and objects
- ▶ Easy-to-learn rapid development and debugging; the development of Python code in Python can be up to 10 times faster than the C/C++ code
- ▶ Exception-based error handling
- ▶ A strong introspection functionality
- ▶ Richness of documentation and software community

Python can be seen as a glue language. Using Python, better applications can be developed because different kinds of programmers can work together on a project. For example, when building a scientific application, C/C++ programmers can implement efficient numerical algorithms, while scientists on the same project can write Python programs that test and use those algorithms. Scientists don't have to learn a low-level programming language and a C/C++ programmer doesn't need to understand the science involved.



You can read more about this from <https://www.python.org/doc/essays/omg-darpa-mcc-position>.

Getting ready

Python can be downloaded from <https://www.python.org/downloads/>.

Although you can create Python programs with Notepad or TextEdit, you'll notice that it's much easier to read and write code using an **Integrated Development Environment (IDE)**.

There are many IDEs that are designated specifically for Python, including IDLE (<http://www.python.org/idle>), PyCharm (<https://www.jetbrains.com/pycharm/>), and Sublime Text, (<http://www.sublimetext.com/>).

How to do it...

Let's take a look at some examples of the very basic code to get an idea of the features of Python. Remember that the symbol `>>>` denotes the Python shell:

- ▶ Operations with integers:

```
>>> # This is a comment
>>> width = 20
>>> height = 5*9
>>> width * height
900
```

Only for this first example, we will see how the code appears in the Python shell:

```
Python Shell
File Edit Shell Debug Options Windows Help
Python 3.3.0 (v3.3.0:bd8afb90ebf2, Sep 29 2012, 10:55:48) [MSC v.1600 32 bit (Intel)]
on win32
Type "copyright", "credits" or "license()" for more information.
>>> #This is a comment
>>> width = 20
>>> height = 5*9
>>> width * height
900
>>>
```

Let's see the other basic examples:

► Complex numbers:

```
>>> a=1.5+0.5j
>>> a.real
1.5
>>> a.imag
0.5
>>> abs(a) # sqrt(a.real**2 + a.imag**2)
5.0
```

► Strings manipulation:

```
>>> word = 'Help' + 'A'
>>> word
'HelpA'
>>> word[4]
'A'
>>> word[0:2]
'He'
>>> word[-1] # The last character
'A'
```

► Defining lists:

```
>>> a = ['spam', 'eggs', 100, 1234]
>>> a[0]
'spam'
>>> a[3]
1234
>>> a[-2]
100
>>> a[1:-1]
['eggs', 100]
>>> len(a)
4
```

- The `while` loop:

```
# Fibonacci series:
>>> while b < 10:
...     print b
...     a, b = b, a+b
...
1
1
2
3
5
8
```

- The `if` command:

First we use the `input()` statement to insert an integer:

```
>>>x = int(input("Please enter an integer here: "))
Please enter an integer here:
```

Then we implement the `if` condition on the number inserted:

```
>>>if x < 0:
...     print ('the number is negative')
...elif x == 0:
...     print ('the number is zero')
...elif x == 1:
...     print ('the number is one')
...else:
...     print ('More')
...
```

- The `for` loop:

```
>>> # Measure some strings:
... a = ['cat', 'window', 'defenestrate']
>>> for x in a:
...     print (x, len(x))
...
cat 3
window 6
defenestrate 12
```


► Defining functions:

```
>>> def fib(n):    # write Fibonacci series up to n
...     """Print a Fibonacci series up to n."""
...     a, b = 0, 1
...     while b < n:
...         print (b),
...         a, b = b, a+b
...
>>> # Now call the function we just defined:
... fib(2000)
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597
```

► Importing modules:

```
>>> import math
>>> math.sin(1)
0.8414709848078965

>>> from math import *
>>> log(1)
0.0
```

► Defining classes:

```
>>> class Complex:
...     def __init__(self, realpart, imagpart):
...         self.r = realpart
...         self.i = imagpart
...
>>> x = Complex(3.0, -4.5)
>>> x.r, x.i
(3.0, -4.5)
```

Python in a parallel world

To be an interpreted language, Python is fast, and if speed is critical, it easily interfaces with extensions written in faster languages, such as C or C++. A common way of using Python is to use it for the high-level logic of a program; the Python interpreter is written in C and is known as CPython. The interpreter translates the Python code in an intermediate language called Python bytecode, which is analogous to an assembly language, but contains a high level of instruction. While a Python program runs, the so-called evaluation loop translates Python bytecode into machine-specific operations. The use of interpreter has advantages in code programming and debugging, but the speed of a program could be a problem. A first solution is provided by third-party packages, where a programmer writes a C module and then imports it from Python. Another solution is the use of a Just-in-Time Python compiler, which is an alternative to CPython, for example, the PyPy implementation optimizes code generation and the speed of a Python program. In this book, we will examine a third approach to the problem; in fact, Python provides ad hoc modules that could benefit from parallelism. The description of many of these modules, in which the parallel programming paradigm falls, will be discussed in subsequent chapters.

However, in this chapter, we will introduce the two fundamental concepts of threads and processes and how they are addressed in the Python programming language.

Introducing processes and threads

A process is an executing instance of an application, for example, double-clicking on the Internet browser icon on the desktop will start a process than runs the browser. A thread is an active flow of control that can be activated in parallel with other threads within the same process. The term "flow control" means a sequential execution of machine instructions. Also, a process can contain multiple threads, so starting the browser, the operating system creates a process and begins executing the primary threads of that process. Each thread can execute a set of instructions (typically, a function) independently and in parallel with other processes or threads. However, being the different active threads within the same process, they share space addressing and then the data structures. A thread is sometimes called a lightweight process because it shares many characteristics of a process, in particular, the characteristics of being a sequential flow of control that is executed in parallel with other control flows that are sequential. The term "light" is intended to indicate that the implementation of a thread is less onerous than that of a real process. However, unlike the processes, multiple threads may share many resources, in particular, space addressing and then the data structures.

Let's recap:

- ▶ A process can consist of multiple parallel threads.
- ▶ Normally, the creation and management of a thread by the operating system is less expensive in terms of CPU's resources than the creation and management of a process. Threads are used for small tasks, whereas processes are used for more heavyweight tasks—basically, the execution of applications.
- ▶ The threads of the same process share the address space and other resources, while processes are independent of each other.

Before examining in detail the features and functionality of Python modules for the management of parallelism via threads and processes, let's first look at how the Python programming language works with these two entities.

Start working with processes in Python

On common operating systems, each program runs in its own process. Usually, we start a program by double-clicking on the icon's program or selecting it from a menu. In this recipe, we simply demonstrate how to start a single new program from inside a Python program. A process has its own space address, data stack, and other auxiliary data to keep track of the execution; the OS manages the execution of all processes, managing the access to the computational resources of the system via a scheduling procedure.

Getting ready

In this first Python application, you'll simply get the Python language installed.



Refer to <https://www.python.org/> to get the latest version of Python.

How to do it...

To execute this first example, we need to type the following two programs:

- ▶ `called_Process.py`
- ▶ `calling_Process.py`

You can use the Python IDE (3.3.0) to edit these files:

The code for the `called_Process.py` file is as shown:

```
print ("Hello Python Parallel Cookbook!!")
closeInput = raw_input("Press ENTER to exit")
print "Closing calledProcess"
```

The code for the `calling_Process.py` file is as shown:

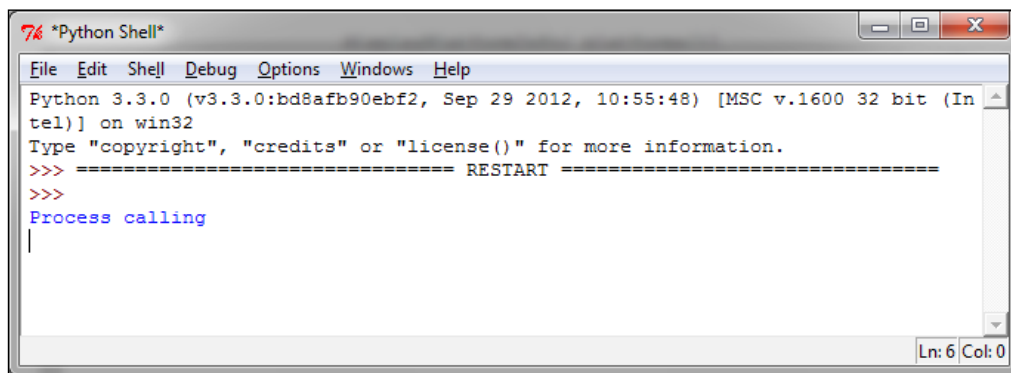
```
##The following modules must be imported
import os
import sys

##this is the code to execute
program = "python"
print("Process calling")
arguments = ["called_Process.py"]

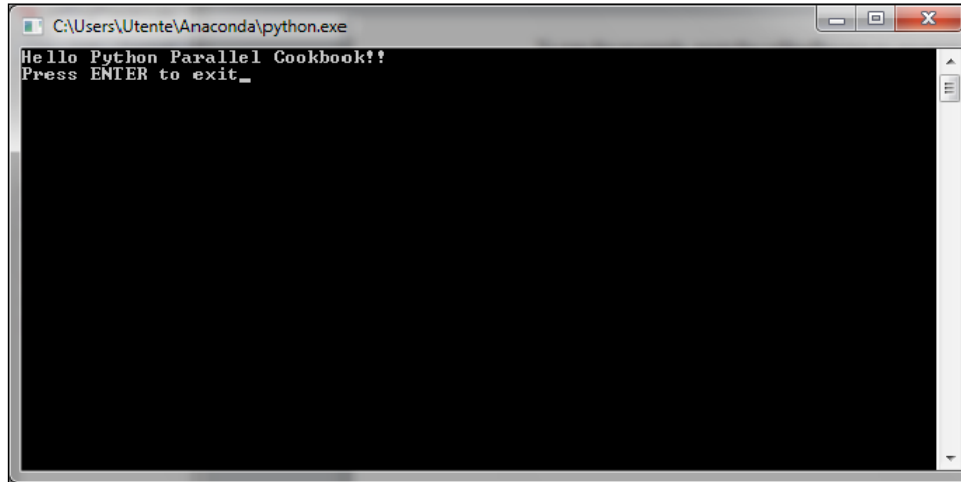
##we call the called_Process.py script
os.execvp(program, (program,) + tuple(arguments))
print("Good Bye!!")
```

To run the example, open the `calling_Process.py` program with the Python IDE and then press the `F5` button on the keyboard.

You will see the following output in the Python shell:



At same time, the OS prompt displays the following:



We have two processes running to close the OS prompt; simply press the *Enter* button on the keyboard to do so.

How it works...

In the preceding example, the `execvp` function starts a new process, replacing the current one. Note that the "Good Bye" message is never printed. Instead, it searches for the program called `_Process.py` along the standard path, passes the contents of the second argument tuple as individual arguments to that program, and runs it with the current set of environment variables. The instruction `input()` in `called_Process.py` is only used to manage the closure of OS prompt. In the recipe dedicated to process-based parallelism, we will finally see how to manage a parallel execution of more processes via the multiprocessing Python module.

Start working with threads in Python

As mentioned briefly in the previous section, thread-based parallelism is the standard way of writing parallel programs. However, the Python interpreter is not fully thread-safe. In order to support multithreaded Python programs, a global lock called the **Global Interpreter Lock (GIL)** is used. This means that only one thread can execute the Python code at the same time; Python automatically switches to the next thread after a short period of time or when a thread does something that may take a while. The GIL is not enough to avoid problems in your own programs. Although, if multiple threads attempt to access the same data object, it may end up in an inconsistent state.

In this recipe, we simply show you how to create a single thread inside a Python program.

How to do it...

To execute this first example, we need the program `helloPythonWithThreads.py`:

```
## To use threads you need import Thread using the following code:
from threading import Thread

##Also we use the sleep function to make the thread "sleep"
from time import sleep

## To create a thread in Python you'll want to make your class work as a
thread.
## For this, you should subclass your class from the Thread class
class CookBook(Thread):
    def __init__(self):
        Thread.__init__(self)
        self.message = "Hello Parallel Python CookBook!!\n"

##this method prints only the message
    def print_message(self):
        print (self.message)

##The run method prints ten times the message
    def run(self):
        print ("Thread Starting\n")
        x=0
        while (x < 10):
            self.print_message()
            sleep(2)
            x += 1
        print ("Thread Ended\n")

#start the main process
print ("Process Started")
```

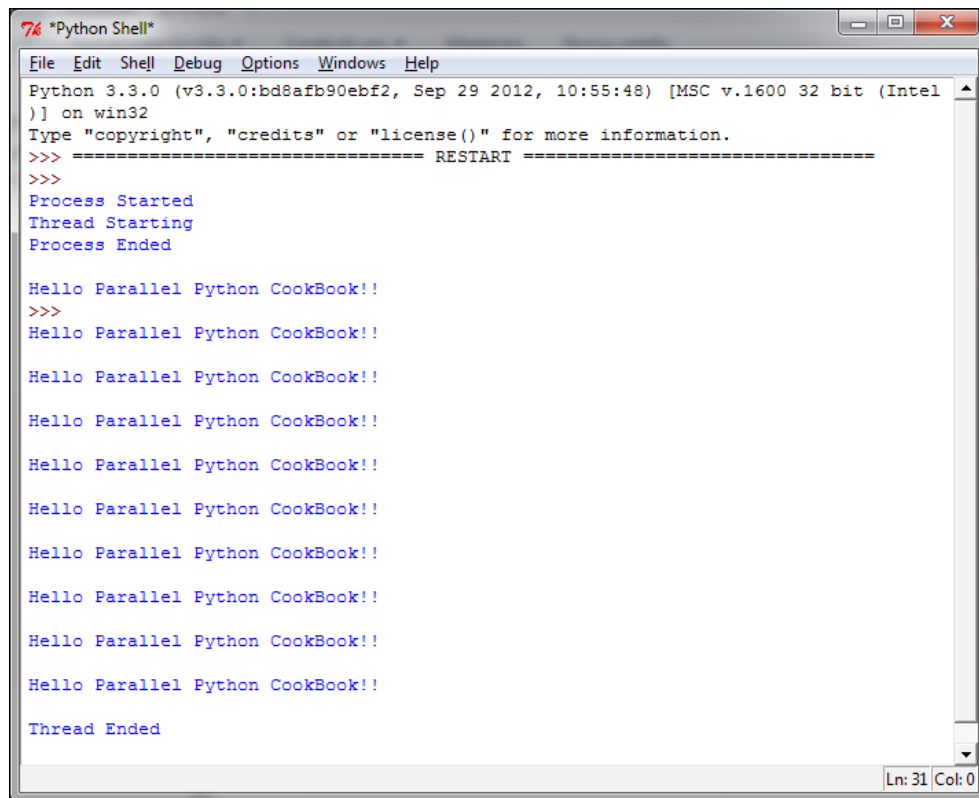
```
# create an instance of the HelloWorld class
hello_Python = CookBook()

# print the message...starting the thread
hello_Python.start()

#end the main process
print ("Process Ended")
```

To run the example, open the `calling_Process.py` program with the Python IDE and then press the `F5` button on the keyboard.

You will see the following output in the Python shell:



```
Python 3.3.0 (v3.3.0:bd8afb90ebf2, Sep 29 2012, 10:55:48) [MSC v.1600 32 bit (Intel
)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> ===== RESTART =====
>>>
Process Started
Thread Starting
Process Ended

Hello Parallel Python CookBook!!
>>>
Hello Parallel Python CookBook!!

Hello Parallel Python CookBook!!

Hello Parallel Python CookBook!!

Hello Parallel Python CookBook!!

Hello Parallel Python CookBook!!

Hello Parallel Python CookBook!!

Hello Parallel Python CookBook!!

Hello Parallel Python CookBook!!

Hello Parallel Python CookBook!!

Hello Parallel Python CookBook!!

Thread Ended
Ln: 31 Col: 0
```

How it works...

While the main program has reached the end, the thread continues printing its message every two seconds. This example demonstrates what threads are—a subtask doing something in a parent process.

A key point to make when using threads is that you must always make sure that you never leave any thread running in the background. This is very bad programming and can cause you all sorts of pain when you work on bigger applications.