Parallel Programming in Python

Contextualizing Parallel, Concurrent, and Distributed Programming

Parallel programming is a reality in all contexts of system development, from smart phones and tablets, to heavy duty computing in research centres. A solid basis in parallel programming will allow a developer to optimize the performance of an application.

The main functionality of the Parallel Programming independently of programming language, is improve the performance of the system where code is running. In addition, if we have a friendly programming language as Python with a modern syntax based of interpreted language and the best point: Python incorporates modules for process-based parallelism (*multiprocessing*), thread-based parallelism (*threading*) among others...

To briefly describe of **Parallel**, **Concurrent** and **Distributed** Programming, we are going to focus in the performance and enhancement that provides these to our programs.

Parallel Programming (1)

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.

Concurrent Programming (2)

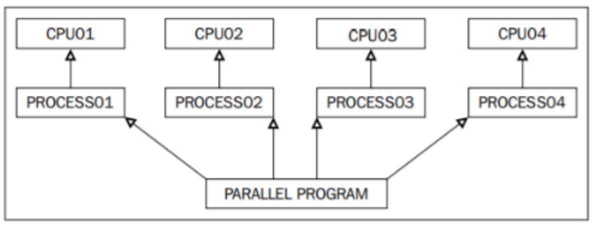
Concurrent programming is an programming paradigm based on abstraction from parallel programming, This is a form of modular programming, namely factoring an overall computation into sub computations that may be executed concurrently.

Distributed Programming (3)

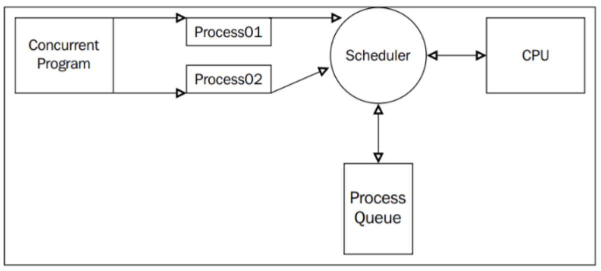
Distributed programming aims at the possibility of sharing the processing by exchanging data through messages between machines (nodes) of computing, which are physically separated.

Computers are incredibly fast, accurate, and stupid; Humans are incredibly slow, inaccurate and brilliant; together they are powerful beyond imagination.

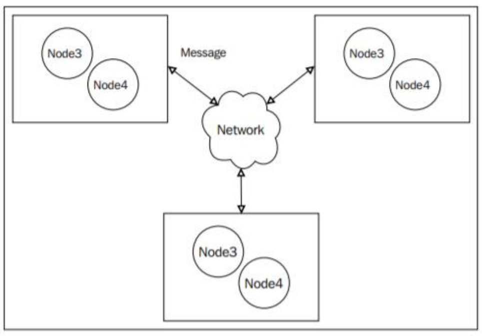
– Albert Einstein



Parallel programming scheme.[1]



Concurrent programming scheme.[2]



Distributed programming scheme.[3]

Identifying parallel programming problems

I. 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. *Example*: Traffic gridlock is an everyday example of a deadlock situation. *

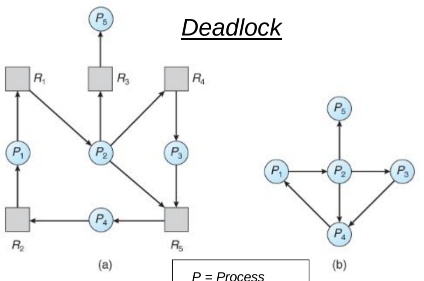
II. Starvation

Starvation describes a situation where a thread is unable to gain regular access to shared resources and is unable to make progress. *Example*: When 2 threads try to access to the same object will often be blocked. **

III. 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. Example: Take a look to the Race Condition in Hardware field, we have the same problem as in Software. ***

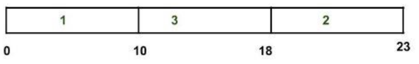
Deadlock



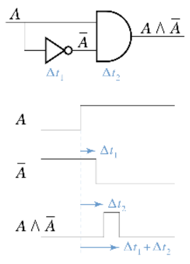
P = Process
R = Resource

Starvation

Process	Burst time	Priority
1	10	2
2	5	0
3	8	1



1967 Priority Scheduling was used in IBM 7094 at MIT



Race Condition

Race condition in a logic circuit. Here, ∆t1 and ∆t2 represent the propagation delays.

Rafael García Cuéllar ©

Discovering Python’s parallel programming tools & Python GIL

The Python language, created by Guido Van Rossum, is a multi-paradigm, multi-purpose language. Within parallel programming, Python (3.x version) has built-in and external modules that simplify implementation. As we can see in these modules:

The Python **threading** module:

The Python threading module offers a layer of abstraction to the module _thread, which is a lower-level module. We can see in this module, data structures and functional objects like Lock Objects, Thread Objects, RLock Objects, Condition Objects, Semaphore Objects, Events…

```
import 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.

```
import multiprocessing
```

The Python **concurrent.futures** package (*New in version 3.2*):

The concurrent.futures package provides a high-level interface for asynchronously executing callables. The asynchronous execution can be performed with threads, using ThreadPoolExecutor, or separate processes, usingProcessPoolExecutor.

```
import concurrent.futures
```

The Python **subprocess** module:

The subprocess module allows you to spawn new processes, connect to their input/output/error pipes, and obtain their return codes. This module provides clases like CompletedProcess (return value from *run()*), Popen Objects(Execute a child program in a new process.), Security Considerations…

```
import subprocess
```

The Python **sched** module:

The sched module defines a class, which implements a general-purpose event scheduler, in the Scheduler Objects.

```
import sched
```

The Python **queue** module:

The queue module implements multi-producer, multi-consumer queues. It is especially useful in threaded programming when information must be exchanged safely between multiple threads. We can implement this module with Queue and SimpleQueue Objects.

```
import queue
```

Python GIL

GIL or **GlobalInterpreterLock** is a mechanism that is used in implementing standard Python, known as CPython, to avoid bytecodes that are executed simultaneously by different threads. 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 the cases that we want to avoid the GIL functionality, what could be done is write such pieces of code as extensions in C language, and embed them into the Python program.

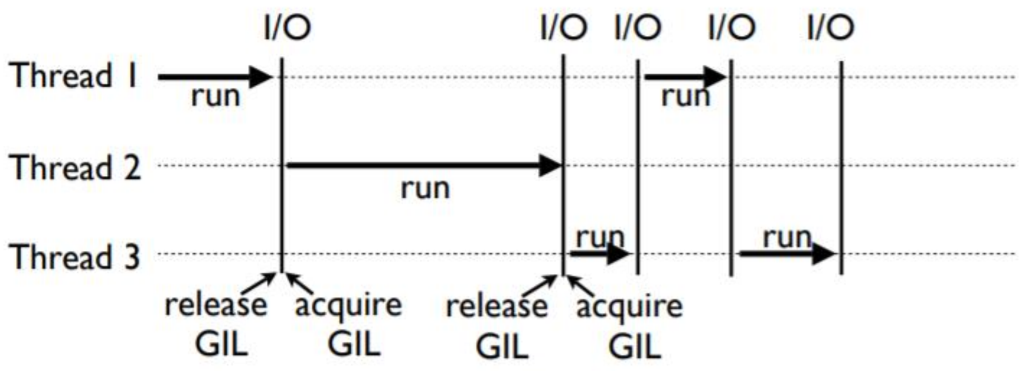
However, other alternatives will be the use of a compiler version of CPython: PyPy.

Most extension code manipulating the GIL has the following simple structure:

```
Save the thread state in a local variable.
Release the global interpreter lock.
... Do some blocking I/O operation...
Reacquire the global interpreter lock.
Restore the thread state from the local variable.
```

https://docs.python.org/3/c-api/init.html#thread-state-and-the-global-interpreter-lock

Thread Execution Model



- When a thread is running, it holds the GIL
- GIL released on I/O (read, write, send, recv, etc.)

Designing Parallel Algorithms

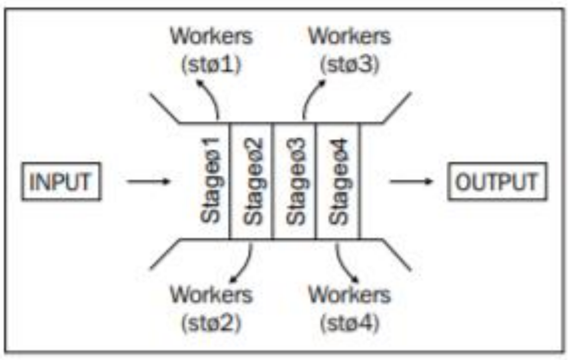
In some cases, we will have to apply the best algorithms for get the best performance and keep save the much time as we can. Python makes possible the application of this kind of algorithms like: The divide and conquer technique, Data decomposition, **Decomposing tasks with pipeline**, **Processing and mapping**. We are going to explain the two last ones:

Decomposing tasks with pipeline

The pipeline technique is used to organize tasks that must be executed in a collaborative way to resolve a problem. Pipeline breaks large tasks into smaller independent tasks that run in a parallel manner. As example, the pipeline model could be compared to an assembly line at a vehicle factory where each car is a task.

Each stage of the pipeline technique acts in an isolated way with its own workers. However, it establishes mechanisms of data communication so that there is an exchange of information.

The following diagram illustrates the pipeline concept:



The pipeline technique

```
if __name__ == "__main__":
    __spec__ = None          # Fix multiprocessing in Spyder's IPython

# 1. pool = ProcessPoolExecutor([max_workers])
# 2. pools[0].submit(worker, random.random()).add_done_callback(pipeline)
# 3. pool.shutdown()
    pools = instanceProcessPool()
    runThreadsInPipeline(pools)
    shutdownPools(pools)
```

PipelineDecomposing
Task.py

- Code show the Pipeline decomposing tasks works.

For the other hand, we can use the 3rd party modules like the Concurrent data pipelines with **pypeln** and the tiny Python module called **MPipe**

Processing and mapping

The number of workers is not always large enough to resolve a specific problem in a single step… We need more simple methods, for after decomposing data or tasks, manipulate the data how we want. In this topic we are going to focus in the comparison between the sequential *map()* and parallel *multiprocessing.map().*

```python
if __name__ == '__main__':
    numberList = [x for x in range(10_000_000)]
# Sequential Mapping
    startTime = datetime.now()
    sequentialMapping(numberList)
    print(getMiliseconds(startTime))

# Parallel Mapping
    startTime = datetime.now()
    parallelMapping(numberList)
    print(getMiliseconds(startTime))
```

MapCompare.py

- Output:
  - 3983.71 milisec.
  - 2615.993 milisec.

Using the parallel & concurrent Python modules

Threading Module

Threads are different execution lines in a process. There are some advantages like: the speed of communication, costless of creation thread, making the best use of data locality and other disadvantages: data sharing allows swift communications and limits in the flexibility of the solution…

Thread-Local Data

Thread-local data is data whose values are thread specific. To manage thread-local data, just create an instance of *local().*

Thread Objects

The Thread class represents an activity that is run in a separate thread of control. There are two ways to specify the activity: by passing a callable object to the constructor, or by overriding the run() method in a subclass. In this class we have methods like: *start(), run(), join(timeout=None), is_alive(), isDaemon()…* I don't need to explain them are auto-defined.

Lock Objects

It is a classic synchronisation lock and could be in one of two states, "locked" or "unlocked". Is created in the unlocked state. It has two basic methods, *acquire()* to lock and *release()*.

RLock Objects

A reentrant lock is a synchronization primitive that may be acquired multiple times by the same thread. Internally, it uses the concepts of "owning thread" and "recursion level" in addition to the locked/unlocked state used by primitive locks. Use *acquire()/release()*

Condition Objects

A condition variable is always associated with some kind of lock; this can be passed in or one will be created by default. *Class Condition(lock=None). wait()* Wait until notified or until a timeout occurs. *wait_for(), notify(), notify_all()* By default, wake up one thread waiting on this condition, if any.

Semaphore Objects
A semaphore manages an internal counter which is decremented by each *acquire()* call and incremented by each *release()* call. Invented by Edsger W. Dijkstra

Event, Timer & Barrier Objects
These objects improve and add parallel functionality in Python from the improvement in the mechanisms, for communication between threads with events passing for the Timer class, which represent an action that should be run only after a certain amount of time. Until, Barrier objects that provides a simple synchronization primitive for use by a fixed number of threads that need to wait for each other.

- ***Example - /Threading/ThreadingExample.py***

Multiprocessing & Multiprocessing.dummy Module
We must understand processes in operating systems as containers for programs in execution and their resources. All that is referring to a program in execution can be managed by means of the process it represents – data area, estates, communications…

The Process Class
In multiprocessing, processes are spawned by creating a Process object and then calling its *start()* method. Process follows the API of *threading.Thread*.

Queues
The Queue class is a near clone of *queue.Queue*. You can *put() a* process and *get()* method for extract the data. And other methods: *qsize(), empty(), full(), close(), join_thread(), cancel_join_thread, put_nowait(), get_nowait().*

Pipes
The *Pipe()* function returns a pair of connection objects connected by a pipe which by default is duplex (two-way). Each connection object has *send()* and *recv()* methods

Lock & RLock (multiprocessing.Lock - multiprocessing.RLock)
The same paradigm & functionality that the Threading module, but in this case with Process. Principal functions: *acquire()* and *release()*

Shared ctypes Objects
It is possible to create shared objects using shared memory, which can be inherited by child processes. We can access to the ctype object through *Value(typecode_or_type), Array(typecode_or_type).* Also, we could implement the *multiprocessing.sharedctypes* in order to provides functions for allocating *ctypes* objects…

Process Pools & Managers
A process pool object, which controls a pool of worker processes to which jobs can be submitted. It supports asynchronous results with timeouts and callbacks and has a parallel map implementation. We use the *Process* class, the main methods are *apply(), apply_async(), map(), map_async(), imap()* *[A lazier version of map()]*, *join(), terminate()…*

On another note, Managers provide a way to create data which can be shared between different processes, including sharing over a network between processes running on different machines. Implement the *Manager* class that return a started *SyncManager* where inside this we have the Barrier, *Condition*, *Event*, *BoundedSemaphore* Objects

- ***Example - /Multiprocessing/MultiProcessingExample.py***

Concurrent & concurrent.futures Module

The asynchronous execution can be performed with threads, using *ThreadPoolExecutor*, or separate processes, using *ProcessPoolExecutor*. Both implement the same interface, which is defined by the abstract Executor class.

Executor Objects

An abstract class provides methods to execute calls asynchronously. While *concurrent.futures.Executor* class we can reference the next methods: *submit()* to execute, *map(), shutdown()*

ThreadPoolExecutor

ThreadPoolExecutor is an Executor subclass that uses a pool of threads to execute calls asynchronously. Is commonly, see this object in *with statement.*

ProcessPoolExecutor

The *ProcessPoolExecutor* class is an Executor subclass that uses a pool of processes to execute calls *asynchronously.ProcessPoolExecutor* uses the multiprocessing module, which allows it to side-step the Global Interpreter Lock (GIL) but also means that only picklable objects can be executed and returned.

Future Objects

Encapsulates the asynchronous execution of a *callable.Future* instances are created by *Executor.submit()* and should not be created directly except for testing. Methods that we will have to take care: *cancel(), cancelled, running(), done(), result(), exception()*…

- ***Example - /Concurrent/PipelineDecomposingTask.py***

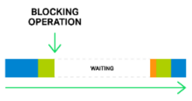
Doing Things Asynchronously

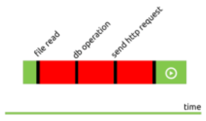
Understanding blocking, nonblocking, and asynchronous operations

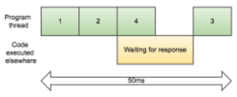
In the case of **blocking operation**, a process that is **blocked** is one that is waiting for some event, such as a resource becoming available or the completion of an I/O operation. The techniques of **mutual exclusion** are used to prevent this concurrent use. When the other task is blocked, it is unable to execute until the first task has finished using the shared resource. Example: Attending a customer at a bank counter.

A **non-blocking operator** is one that, at a minimal blocking sign, returns a control code or exception that tells the solicitor to retry later. Example: Imagine that among the clients waiting to be attended and one client needs to withdraw a benefit, but benefits are not available at the moment.

Asynchronous operations notify the end of solicitations by means of **callbacks**, coroutines, and other mechanisms. A callback function is a function that is called when a certain *condition occurs*. Example: When we finished a transaction the application mobile where we did this transaction raise a notification user in the mobile-app.







Understanding the Event Loop

We can define event loops as abstractions that ease up using polling functions to monitor events. Internally, event loops make use of poller objects, taking away the responsibility of the programmer to control the tasks of addition, removal, and control of events.

Polling functions work in the following steps:

I. A *poller* object is created.
II. We can register or not one or more resource descriptors in poller.
III. The polling function is executed in the created *poller* object

Using asyncio

We can define *asyncio* as a module that came to reboot asynchronous programming in Python. The *asyncio* module allows the implementation of asynchronous programming using a combination of the following elements:

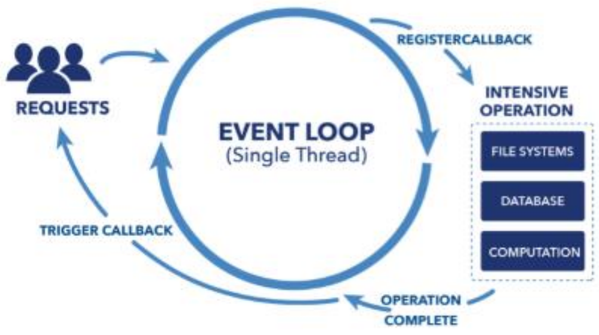
- **Event loop**: The asyncio module allows an event loop per process.
- **Coroutines**: As mentioned in the official documentation of asyncio, "A coroutine is a generator that follows certain conventions." It's most interesting feature is that it can be suspended during execution to wait for external processing (some routine in I/O) and return from the point it had stopped when the external processing is done.
- **Futures**: The asyncio module defines its own object Future. Futures represent a processing that has still not been accomplished.
- **Tasks**: This is a subclass of *asyncio.Future* to encapsulate and manage coroutines.

```python
import asyncio

async def main():
    print('Hello ...')
    await asyncio.sleep(1)
    print('... World!')

# Python 3.7+
asyncio.run(main())
```

- Output: "Hello World!"

asyncio provides a set of **high-level** APIs.

There are **low-level** APIs for *library and framework developers* to: networking, running *subprocesses*, handling *OS signal*, use *transports…*



Medium post by *Gaurav Balyan*

Using Python Parallel Programming Libraries

Dask – Parallel Computing with Task Scheduling

Dask provides advanced parallelism for analytics, enabling performance at scale for the tools you love. Also, Dask is open source and freely available, familiar for Python users which means that we don't have to completely rewrite our existing code, scale up to clusters and customizable letting build custom systems for in-house application. You can install everything required for most common uses of Dask (arrays, dataframes…)

```
pip install "dask[complete]"    # Install everything
```

Ipyparallel – Interactive Computing in Python

This architecture abstracts out parallelism in a general way, enabling IPython to support many different styles of parallelism, including single program, multiple data (SPMD) parallelism, multiple program: multiple data (MPMD) parallelism, task farming, data parallel, combinations of these approaches…

```
pip install ipyparallel
```

Celery – Distributed Task Queue

Celery is an asynchronous task queue/job queue based on distributed message passing. It is focused on real-time operation, but supports scheduling as well. The execution units, called tasks, are executed concurrently on a single or more worker servers using multiprocessing, *Eventlet*, or *gevent*. Tasks can execute asynchronously (in the background) or synchronously (wait until ready).

```
pip install Celery
```

Bibliography

Parallel Programming with Python Book

Parallel Programming with Python: Develop Efficient Parallel Systems Using the Robust Python Environment by Jan Palauch.
https://www.amazon.es/Parallel-Programming-Python-Efficient-Environment/dp/1783288396

Python Offical Documentation

- Concurrency Execution - https://docs.python.org/3.7/library/concurrency.html
- Threading Module - https://docs.python.org/3.7/library/threading.html
  https://github.com/python/cpython/blob/master/Lib/threading.py
- Multiprocessing Module - https://docs.python.org/3.7/library/multiprocessing.html
  https://github.com/python/cpython/tree/master/Lib/multiprocessing
- Concurrent Futures - https://docs.python.org/3.7/library/concurrent.futures.html
  https://github.com/python/cpython/tree/master/Lib/concurrent/futures
- Event Schedule - https://docs.python.org/3.7/library/sched.html
  https://github.com/python/cpython/blob/master/Lib/sched.py
- Synchronized Queue - https://docs.python.org/3.7/library/queue.html
  https://github.com/python/cpython/blob/master/Lib/queue.py