Parallel Computing

11

11.1 Introduction

The computer industry underwent a vigorous shake-up several years ago. Major chip manufacturers gave up trying to increase processor frequency. Each year, more and more transistors fit into the same space, but their clock speed cannot be increased without overheating. Thus, rather than trying to increase the clock speed, manufacturers turned to multicore architectures. A multicore processor is a single computing component with two or more processing units (called "cores") which read and execute program instructions. Multiple cores can run different instructions at the same time, thereby increasing the overall speed of programs susceptible to parallel computing. Within multicore systems, the cores communicate through hardware (the bus) in order to synchronize access to common resources such as RAM.

The operating system is the application that manages these multiple cores. If two computation-intensive processes (i.e., applications) are run on the computer, the operating system manages things so that each task is run on a different core. If we have a single computation-intensive task, it will only run on one core, even if our computer has multiple cores. If nothing is done explicitly, we will waste a lot of computation power!

Currently, in most parallel programming frameworks, the programmer has to manually split the computation work into multiple tasks so that each one is executed in different cores. The programmer has to perform the split and the operating system will then automatically execute each task on a different core. So, each task has to be run in different processes or threads. This is the principle behind parallel programming; harnessing multiple processors to work on a single task by dividing it into multiple (smaller) tasks.

In order to make the most of multicore capabilities, the number of processes should be equal to the number of processors. Within a parallel computing context, it does not make much sense to define more tasks than cores we have, e.g., defining eight computation-intensive tasks if our computer only has four cores. In this latter

L. Igual and S. Seguí, *Introduction to Data Science*,
Undergraduate Topics in Computer Science, DOI 10.1007/978-3-319-50017-1_11

case, the operating system will try to run eight tasks using four cores. This is done by switching between the tasks in such a way that each one gets approximately the same amount of computing time. Switching between tasks has a computational cost and thus overall performance may suffer if the number of simultaneous tasks is higher than the number of available cores.

Assume that a task takes T seconds to run on a single core (using standard serialized programming). Now assume that we have a computer with N cores and that we have divided our serialized application into N subtasks. By using the parallel capabilities of our computer we may be able to reduce the total computation time to T/N. This is the ideal case and usually we will not be able to reduce the computation time by a factor of N. This is due to the fact that cores, on the one hand, need to synchronize at the hardware level in order to access common resources such as RAM; and, on the other hand, the operating system needs some time to switch between all the tasks that run on the computer. However, using the multicore capabilities of the computer unit will result in a reduction of the computation time if the tasks are properly defined.

Parallelization can also be performed by means of distributed computing. While in multicore systems the cores communicate with each other through the bus at the hardware level, in distributed systems software communicates and coordinates the actions of computational entities located within a network. The computational entities are usually computers. In distributed computing, a large number of discrete computers, named *nodes*, distributed across a network (e.g., the Internet) devote some or all of their computation time to solving a common problem; each node receives and completes many small tasks, reporting the results to a central server which integrates the results into the overall solution. Each of the nodes has its own local memory and thus tasks that run on different computers do not need to coordinate access to it. However, since information is exchanged through the network, care must be taken in order to select the amount of information that is passed so as to optimize the computational performance.

In this chapter we will focus on IPython's capabilities for parallel computing, on both multicore and distributed systems. IPython does indeed offer an environment capable of dealing with both architectures in a transparent manner for the programmer. The user should be aware of the underlying architecture in which the application will be run in order to avoid loss of performance. We would like to point out that Python currently does not offer support for the parallel capabilities explained below. IPython, however, supports them.

11.2 Architecture

Figure 11.1 shows a simplified version of the IPython architecture for parallel computing (multicore and distributed).[1] The proposed architecture enables IPython to

[1]For a more detailed description please see http://ipyparallel.readthedocs.io/en/stable/intro.html. Last seen July 2016.

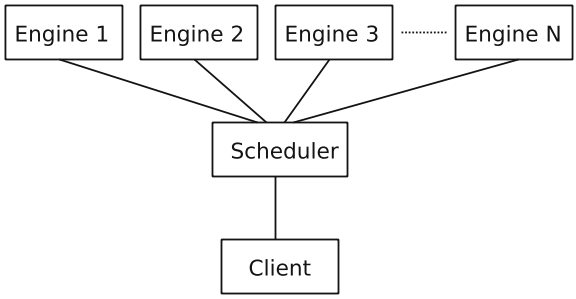

Fig. 11.1 IPython's architecture for parallel computing (multicore and distributed)

support many different styles of parallelism including those described in this chapter. Each of the blocks is explained below:

- Each engine is an instance of IPython, usually an IPython interpreter, that receives commands through a connection. When multiple engines are started, multicore and distributed computing becomes possible.
- The scheduler is an application that distributes the commands to the engines. We will see that there are two ways of distributing this work: the direct view and the load-balanced view, described in later sections.
- The client is an IPython object created at an IPython interpreter. This object will allow us to send commands to the IPython engines.

IPython uses the term *cluster* to refer to the scheduler and the set of engines that make parallelization possible. It should not be confused with the term cluster used in supercomputing. In addition, the reader should take into account that:

- Each engine is an independent instance of an IPython interpreter, i.e., it runs an independent process. None of the variables declared at, e.g., engine 1 are visible to the remaining engines or to the client. In a similar way, if we want to work with `numpy` functions, we should import this toolbox to every engine.
- We may be able to control at which engine each task is executed, but we will not be able to control on which core each engine is executed; this is the job of the operating system.

11.2.1 Getting Started

To use IPython's parallel capabilities, the first thing to do is to start the cluster. There are two ways of doing this:

- From the notebook interface. This is the simplest way of proceeding and is the recommended way for newbies in this topic. Within the IPython notebook, we can use the Clusters tab of the dashboard, and press Start with the desired number

of cores, under the desired profile.[2] This will automatically run the necessary commands to start the IPython cluster. In this case, the notebook will be used as the interface with the cluster; i.e., we will be able to send different tasks to the engines using the web interface.

- From the command line of a terminal. We can run the following command to start an IPython cluster:

```
$ ipcluster start
```

This command will create a cluster with N engines, where N equals the number of cores. If we want to create a cluster with a different number of engines, we just run:

```
$ ipcluster start -n 4
```

With this command we start a cluster with four engines. Once the engines are started, we may run an IPython interpreter.

```
$ ipython
```

11.2.2 Connecting to the Cluster (The Engines)

We have seen how to initialize the cluster. No matter which way we initialize the cluster, the following commands allow us to connect to it. These commands should either be introduced through the notebook or be typed into the IPython command line interpreter (the client):

In [1]:
```
from IPython import parallel
engines = parallel.Client()
engines.block = True
print engines.ids
```

Out[1]:
```
[0, 1, 2, 3, 4, 5, 6, 7]
```

These commands connect to the cluster and output the number of engines in it. If an error is shown when running the commands, the cluster has not been correctly created. We will explain later on the meaning of the `block` attribute.

The variable `engines` is an object that represents the available engines to which commands can be sent. Let us now see two different ways we can send tasks to the engines: the first, called the *direct view*, is simpler and allows the user to directly control which tasks are sent to which engines; the second, called the *load-balanced view*, delegates to the IPython scheduler the task of deciding which engines each task is sent to.

[2] More information on ipcluster profiles can be found at http://ipython.readthedocs.io/en/stable/.

As will be seen next, the former view is useful if a task can be evenly distributed computationally into smaller tasks; whereas the second is more useful if such subdivision cannot be easily done. For instance, if we have to analyze multiple data files, the direct view is a good approach if all the files have approximately the same size. But if the files differ (quite a lot) in size, the load-balanced view is the better approach. Let us now see both approaches.

11.3 Multicore Programming

11.3.1 Direct View of Engines

How do we send a command to the cluster? Recall that the `engines` variable just defined represents the engines in the cluster. Within the direct view, `engines[0]` represents the first engine, `engines[1]` the second engine, and so on. The following commands, executed on the client (i.e., the IPython interpreter), send commands to the first engine:

In [2]:
```
engines[0].execute('a = 2')
engines[0].execute('b = 10')
engines[0].execute('c = a + b')
```

We may retrieve the result by executing the following command on the client:

In [3]:
```
engines[0].pull('c')
```

Out[3]:
```
12
```

Note that we do not have direct access to the command line of the first engine. Rather, we may send commands to it through the client.

What about parallelization? Let us try the following:

In [4]:
```
engines[0].execute('a = 2')
engines[0].execute('b = 10')
engines[1].execute('a = 9')
engines[1].execute('b = 7')
engines[0:2].execute('c = a + b')
```

These commands initialize different values for `a` and `b` at engines 0 and 1 and execute the sum at both engines. Since each engine runs an independent process, the operating system may schedule each engine at different cores and thus execution is performed in parallel. Again, as before, we can retrieve both results using the pull command:

In [5]:
```
engines[0:2].pull('c')
```

Out[5]:
```
[12, 16]
```

Note that with these commands we are directly accessing the engines and that is why this type of approach is called the direct view.

In order to simplify the code, let us define the following variables:

In [6]:
```
dview2 = engines[0:2]
dview = engines.direct_view()
```

The variable `dview2` references the first two engines, whereas `dview` references all the current engines. This variable will be used later on, in Sect. 11.5.

Let us now try with matrix multiplication. Assume we have created four matrices $A0$, $B0$, $A1$, and $B1$ on the client. The objective is to compute the matrix products: $C0 = A0B0$ and $C1 = A1B1$.

The commands to be executed are as follows:

In [7]:
```
dview2.execute('import numpy as np')

engines[0].push(dict(A=A0, B=B0))
engines[1].push(dict(A=A1, B=B1))

dview2.execute('C = np.dot(A,B)')
dview2.pull('C')
```

Observe that the `import` command has to be run on each of the engines so that the scientific computing library becomes available on each engine. As before, the `push` and `pull` commands are used to send and retrieve data between the client and the engines, and the `execute` command computes the matrix product on both engines. It should be pointed out that the `push`, `execute`, and `pull` commands block (i.e., they do not return) until the engines have completed their corresponding task. This is due to the attribute `engines.block = True` we set when initializing the cluster, see Sect. 11.2.2. We may set the attribute to `False`, in which case the commands will return immediately, without waiting for the command to end. This feature may be very useful if we want to take full advantage of parallelization capabilities and performance. However, additional commands need to be introduced in order to ensure that, for instance, the `execute` command is not issued before the engines have received the corresponding matrices with the `push` command. The reader may find more information on this issue in the corresponding documentation.[3] An example of the non-blocking feature is shown in Sect. 11.5.

The previous examples show us how to execute commands on engines as if we were typing them directly into the command line. Indeed, we have manually sent,

[3] http://ipython.readthedocs.io/en/stable/.

executed, and retrieved the results of computations. This procedure may be useful in some cases but in many cases there will be no need for it. Indeed, the `apply` function allows us to simplify such procedure. Let us see this with the following example:

In [8]:
```
def mul(A, B):
        import numpy as np
        C = np.dot(A, B)
        return C

C = engines[0].apply(mul, A0, B0)
```

These commands, executed on the client, perform a remote call. The function `mul` is defined locally but is executed on the first engine. There is no need to use the `push` and `pull` functions explicitly to send and retrieve the results; it is done implicitly. All methods that communicate with the engines are built on top of the `apply` method. Note the `import numpy as np` inside the function. This is a common model, to ensure that the appropriate toolboxes are imported where the task is run.

If we execute `dview2.apply(mul, A0, B0)` we would execute the same command on engines 0 and 1. So, how can we call up the `mul` function and distribute parameters among the engines? The direct view (and load-balanced view, as we will see next) offers us the `map` method to tackle this issue:

In [9]:
```
[C0, C1] = dview2.map(mul,[A0, A1],[B0, B1])
```

The `map` call splits the tasks between the engines associated with `dview2`. In the previous example, the task `mul(A0,B0)` is executed on one engine and `mul(A1, B1)` is executed on the other one. Which command is executed on each engine? What happens if the list of arguments to map includes three or more matrices? We may see this with the following example:

In [10]:
```
engines[0].execute('my_id = "engineA"')
engines[1].execute('my_id = "engineB"')

def sleep_and_return_id(sec):
        import time
        time.sleep(sec)
        return my_id,sec

dview2.map(sleep_and_return_id, [3, 3, 3, 1, 1, 1])
```

Note that the `sleep_and_return_id` makes the function sleep for the specified amount of time and returns the identifier of the engine that has executed the function. The output is as follows:

Out[10]:
```
[('engineA', 3),
 ('engineA', 3),
 ('engineA', 3),
 ('engineB', 1),
 ('engineB', 1),
 ('engineB', 1)]
```

The previous output shows to which engine each task is assigned. The direct view distributes the tasks in a uniform way among the engines before executing them no matter which is the delay we pass as argument to the function `sleep_and_return_id`. Since the `block` attribute is set to `True`, the `map` function blocks until all engines have finished with their corresponding tasks. This is a good way to proceed if you expect each task to take the same amount of time. But if not, as is the case in the previous example, computation time is wasted and so we recommend to use the load-balanced view instead.

11.3.2 Load-Balanced View of Engines

The load-balanced view is an interface that allows, as does the direct view interface, parallelization of tasks. With load-balanced view, however, the user has no direct access to individual engines. It is the IPython scheduler that assigns work to each engine. This interface is simultaneously simpler and more powerful.

To create a load-balanced view we may use the following command:

In [11]:
```
engines.block = True
lview2 = engines.load_balanced_view(targets = [0, 1])
lview  = engines.load_balanced_view()
```

Again, we use the blocking mode since it simplifies the code. As can be seen, we have defined two variables: `lview2` is a variable that references the first two engines, whereas `lview` references all the engines.

Our example will be centered on the `sleep_and_return_id` function we saw in the previous subsection:

In [12]:
```
lview2.map(sleep_and_return_id, [3 ,3 ,3 ,1 ,1 , 1])
```

Observe that rather than using the direct view interface (`dview2` variable) of the `map` function, we use the associated load-balanced view interface (`lview2` variable). The output for our execution is as follows:

Out[12]:
```
[('engineB', 3),
 ('engineA', 3),
 ('engineB', 3),
 ('engineA', 1),
 ('engineA', 1),
 ('engineA', 1)]
```

As for the case of the direct view, the `map` function returns as soon as all the tasks have finished, since we are using the blocking mode. The output may vary each time the `map` function is executed. In this case, the tasks are assigned to the engines in a dynamic way. The `map` function of the load-balanced view begins by assigning one task to each engine in the order given by the parameters of the `map` function. By default, the load-balanced view scheduler then assigns a new task to an engine when it becomes free.[4] Since with the load-balanced view we do not know on which engine execution will take place, explicit data movement methods like `push` and `pull` functions are not provided in this view. The direct view should be used instead if needed.

The reader should have noticed the simplicity of the IPython interface to parallelize tasks. Once the cluster of engines has been set up, we may use the `map` function to execute tasks in parallel. This simplicity allows IPython's parallelization capabilities to be used in distributed computing. We next offer an overview of some of the associated issues.

11.4 Distributed Computing

The previous section introduced multicore computing; i.e., how to take advantage of the N multiple cores of a computer in order to speed up code execution. An application that takes T seconds to execute on a single core could be executed in T/N seconds if the tasks are properly defined. But what if we need to reduce the computation time even more?

One solution might be what is called as scale-up. That is, buying a new computer or a new processor with more cores, adding more memory to the system, buying faster storage, and so on.

Another solution is called scale-out: interconnecting multiple computers to make them work together to solve a problem. That is, create a grid of computers. Grids allow you to scale your system to meet your needs: add as many computers as you need, use all of them or only a few of them. Grids offer great scalability but low performance; whereas supercomputers give the best performance values but have scalability limitations.

In distributed computing, the nodes work together in order to solve a problem. As information is exchanged through the network, care must be taken to select the amount of information that is passed in order to optimize computational performance. One of the most prominent examples of distributed computing is the SETI@Home project: a project that searches for extraterrestrial life by analyzing radiotelescope signals. For that, the computational capacity of millions of computers belonging to volunteer users is used.

[4] Changing this behavior is beyond the scope of this chapter. You can find more details here: http://ipyparallel.readthedocs.io/en/stable/task.html#schedulers. Last seen November 2015.

IPython offers the possibility of setting up a cluster of engines running on different computers. One way to proceed is to use the `ipcluster` command (see Sect. 11.2.1) in SSH mode; the official documentation has examples of this. Configuring IPython to work with a grid of computers is not as easy as configuring it for multicore computing, so commercial platforms that offer the computational grid and ease the configuration process are also available.

All the commands that are discussed in Sect. 11.3 can also be used in distributed programming. However, it should be taken into account that the `push` and `pull` commands send data through the network. Sending many data through the network may drastically reduce the performance of the system; thus data movement is an important issue to tackle in distributed computing. Rather than using `push` and `pull` commands (either explicit or implicitly), engines may access the data they need directly on disk. Different approaches may be used in this case; data may be stored in a shared filesystem, for instance. This approach is useful and common if computers are interconnected within a local network but it is difficult to implement with computers connected in different networks. In a shared filesystem, the data are stored in a server and thus each computer has to connect with the server and retrieve the data needed from the same server. This can become a bottleneck when working with many data.

Another approach is to use a distributed filesystem. In this case, rather than storing all the data in a single server, data are divided into chunks and replicated between multiple computers. The data to be processed are distributed and thus the same computer that stores the chunk can work with it. This way of proceeding may be useful for Big Data: a broad term that refers to the processing of large datasets.

11.5 A Real Application: New York Taxi Trips

This section presents a real application of the parallel capabilities of IPython and discussion of several approaches to it. The dataset is a database of taxi trips in New York and it has been obtained through a Freedom of Information Law (FOIL) request from the New York City Taxi & Limousine Commission (NYCT&L) by the University of Illinois at Urbana-Champaign.[5] The dataset consists of 12 × 2 Gbyte CSV files. Each file has approximately 14 million entries (lines) and is already cleaned. Thus no special preprocessing is needed to be able to process it. For our purposes, we are only interested in the following information from each entry:

- `pickup_datetime`: start time of the trip, mm-dd-yyyy hh24:mm:ss EDT.
- `pickup_longitude` and `pickup_latitude`: GPS coordinates at the start of the trip.

[5] http://publish.illinois.edu/dbwork/open-data/.

Our objective is to analyze these data in order to answer the following questions: for each district, how many pickups are performed during week days and how many during weekends? And how many pickups are performed in the morning? For this issue, the city of New York is arbitrarily divided into nine districts: ChinaTown, WTC, Soho, Harlem, UpperTown, MidTown, DownTown, UpperEastSide, UpperWestSide, and Financial.

Implementing the previous classification is rather simple since it only requires checking, for each entry, the GPS coordinates of the start of the trip and the pickup date and time. Performing this task in a sequential way may take a rather long time, since the number of entries, for a single CSV file, is rather large. In addition, special care has to be taken when reading the file since a 2 Gbyte file may not fit into the computer's memory.

We may take advantage of parallelization capabilities in order to reduce the processing time. The idea is to divide the input data into chunks so that each engine takes care of classifying the entries in their corresponding chunks. A simple procedure may follow from the previous idea: we may explicitly divide the original 2 Gbyte file into multiple smaller files of approximately the same number of entries. Such splitting may be performed using, for instance, the Unix `split` command. Once performed, each engine reads and processes its chunks and the result may be collected by the client. Since we expect each chunk to be processed in the same amount of time the chunks may be distributed by the client using the `map` function of the direct view.

Although straightforward to implement, this has several drawbacks. Note that the new procedure includes a splitting stage that divides the input file into multiple smaller files. Splitting the file implies accessing a disk for reading and writing, and thus it may reduce the overall possible improvement, since accessing the disk is usually slow in comparison to CPUs computing capabilities. In addition, the splitting process reads the input file and afterwards each engine reads the split data again from the disk. There is no need to read data twice. We may avoid reading the data twice by letting each engine read their corresponding chunks from the original non-split file. However, this may also reduce the overall improvement since it may imply numerous movements of the disk brace when data are read from the disk by multiple engines. Finally, care should be taken when splitting the input file into smaller ones. Notice that each engine will read its assigned chunk and thus we must ensure that all chunks read by the engines fit into memory.

11.5.1 A Direct View Non-Blocking Proposal

We propose here a second approach which avoids reading the data twice by the computer. It is based on implementing a producer–consumer paradigm in order to distribute the tasks. The producer, associated with the client, reads the chunks from disk and distributes them among the engines using a round-robin technique. No explicit `map` function is used in this case. Rather, we simulate the behavior of the `map` function in order to have fine control of the parallel problem. Recall that each

engine runs an independent process. Since we assign different tasks to each engine, the operating system will try to execute each engine via a different process.

Assume engines are labeled with values 1 to N. The proposed solution, based on a round-robin algorithm, is as follows: the client begins by manually distributing a chunk to each engine in an ordered way, from engine 1 to engine N, and asking them to analyze its contents. This is performed in a non-blocking mode: the client will not wait for the task to finish on one engine in order to send a chunk to the next engine. Once a chunk has been distributed to each engine, the client then waits for the engine 1 to finish. Once finished, it sends a new chunk to it and asks it to analyze it without waiting for the engine to finish. The client then waits for the engine 2 to finish, sends it a new chunk and asks it to process it, and so on. The previous procedure is repeated until all the chunks have been sent to the engines. The engines accumulate the overall partial result of analyzing their chunks in a local variable. Once all the engines have finished, the client collects the partial results of each engine to compute the final result.

This round-robin technique is useful since each engine receives a chunk of the same size. Thus, each engine is expected to take the same amount of time to process its chunk. Indeed, if all engines are processing a chunk, the most likely engine to finish first is the one that, among all engines, is next in the round-robin queue.

Our solution is based on the direct view interface, see Sect. 11.3.1. We use the direct view since we would like to have explicit access to the engines in order to distribute the chunks. We also assume that one CSV file does not fit into memory. Therefore, the client (i.e., the producer) will split the input data into uniform chunks of appropriate size. The whole implementation of the solution is available as an IPython notebook. Here, we discuss only issues related to parallelization. Therefore, no number has been assigned to the input cells.

First, let `dview` be an IPython object associated with all the engines in the cluster. We set the `block` attribute to `True`, i.e., by default all the commands that are sent to the engines will not return until they are finished. In order to be able to send tasks to the engines in a round-robin-like fashion, an infinite iterator over the list of engines can be created. This can be done with a Cycle object:

```
from itertools import cycle
c_engines = cycle(engines.ids)
```

Our proposal then has the following steps, see Fig. 11.2:

1. We begin by sending each engine all the necessary functions that are needed to process the data. Of these functions, we just mention `init()`, which resets the (local) engine's variables, and `process(b)`, which classifies a chunk `b` of lines and groups the results into a `local_total` variable, which is local to each engine. After sending the necessary functions to the engines, in each engine we execute the `init()` function, in order to initialize the local variables in each engine:

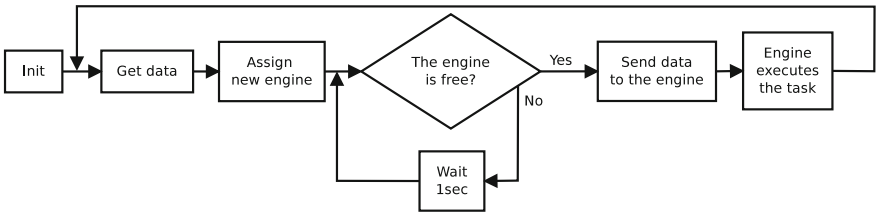

Fig. 11.2 Block diagram of the algorithm to process databases with taxi trips

```
for i in engines.ids:
    async_tasks[i] = engines[i].execute('init()',
                                        block = False)
```

Observe that it is executed in non-blocking mode. That is, the `init()` function is executed on each engine without waiting for the engine to finish and thus the `execute` command will return immediately. Thus, the loop can be executed for each engine in parallel. In order to know whether the execute command has finished for a given engine, we will need to check, when needed, the state of the corresponding `async_tasks` variable.

After performing this step the client enters a loop made up of steps 2 to 6 (see Fig. 11.2).

2. The client reads a chunk of the file and selects which engine the chunk will be sent to:

```
new_chunk = get_chunk(f, lines_per_block)
run_engine = c_engines.next()
```

These commands will be executed even if the `init()` function has not finished or if the engines have not finished processing their previous chunk. Each read chunk will have the same number of lines (with the exception of the last chunk read from the file) and thus we expect each chunk to be processed in the same amount of time by each engine. We therefore manually select the next engine in a round-robin fashion.

3. Once the chunk has been read and the engine that will process the chunk has been selected, we need to wait for the engine to finish its previous task. It may still be in the initialization state or it may be processing a previous chunk. While the engine has not finished, we wait:

```
while ( not async_tasks[run_engine].ready() ):
    time.sleep(1)
```

4. At this point, we are sure that the `run_engine` engine is free. Thus, we may send the data to the engine and ask it to process them:

```
mydict = dict(data = new_chunk)
engines[run_engine].push(mydict, block = True)
async_tasks[run_engine] = engines[run_engine].
                    execute('process(data)', block = False)
```

The `push` is performed with the default value of `block = True`. Thus the `push` function will not return until the chunk has arrived at the engine. Once it returns, we are sure that the chunk has been received by the engine and thus we may call the `execute` function. The latter function will process the data in non-blocking mode. Thus, the `execute` function will return immediately and meanwhile the engine will process its corresponding block.
It should be mentioned that the `process` function locally aggregates the results of analyzing each chunk in the variable `local_total`. At the end, the client will collect the local results from all the engines.

5. The algorithm then jumps again to step 2. The first time step 2 is executed the selected engine is engine 0. The second time it will be engine 1 and so on. After a chunk has been assigned to all engines the algorithm will again select engine 0; so it will wait until engine 0 has finished processing its previous chunk.
6. Once the loop (steps 2 to 5) has processed all the chunks in the file, the client gets the results from each engine and aggregates them into the `global_result` variable. Before reading the result we need to be sure that the engine has finished with its last chunk:

```
for engine in engines.ids:
    while (not async_tasks[engine].ready()):
        time.sleep(1)
    global_result += engines[engine].pull('local_total',
                                          block = True)
```

The `pull` is performed in blocking mode. After reading all the results from the engines the final result is stored in the dictionary `global_result`.

11.5.2 Results

The experiments were performed on an i7-4790 CPU with four physical cores with HyperThreading and 8 Gb of RAM. We performed experiments with different numbers of engines and different numbers of lines per block (i.e., the variable `lines_per_block` in the previous subsection). The performance results are shown in seconds and were obtained by computing the mean of three executions.

11.5.2.1 Lines per Block

The number of lines per block defines the number of data that will be sent to each of the engines to be processed. In order to test the performance of the algorithm, we performed tests with different values of lines per block and a reduced version of one CSV file: only 1 million lines were processed. The experiments used 8 engines; i.e.,

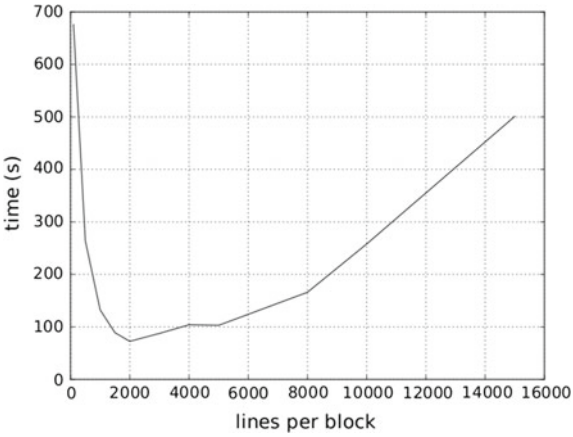

Fig. 11.3 Performance to process 1 million lines of a CSV file using 8 engines for different values of lines per block. Time is shown in seconds

the number of processors of the computer. Thus, in our environment, there will be a total of nine processes running: one producer, which is in charge of reading the CSV file and distributing the data among the engines in blocks defined by the variable associated with lines per block, and eight engines that will take the blocks of data from the producer and process them.

The results are shown in Fig. 11.3. As can be seen, an optimal execution time is located near 2,000 lines per block. With fewer lines per block, efficiency is lost because most of the time engines are idle (thus cores are also idle), and the system wastes lots of computational time managing short messages between processes. When working with more than 6,000 lines per block, the messages to be passed between processes are too big to be moved quickly.

Similar effects can be found by modifying the waiting time when an engine is busy; see step 3 in Sect. 11.5.1. Tests can be done to show that with a shorter waiting time the optimal number of lines per block value is reduced. Nevertheless, optimal execution time does not change because the optimal execution time is based on not having idle cores.

11.5.2.2 Number of Engines

The number of engines is associated with the level of parallelization that the code can reach. We tested our algorithm using 2,000 lines per block and different numbers of engines, again using a reduced version of one CSV file. In this case, 100,000 lines were processed. The result is shown in Fig. 11.4. As can be seen, for a given number of cores, the time that is needed to process the data reduces as the number of engines is increased, and the relation between the number of engines and time is not linear. The reason for this is that the operating system sees each engine as one process and thus each engine is expected to be scheduled on different processors of the computer. Note that for one engine the execution time is rather high; time is reduced if more engines are included in the environment until the number of engines

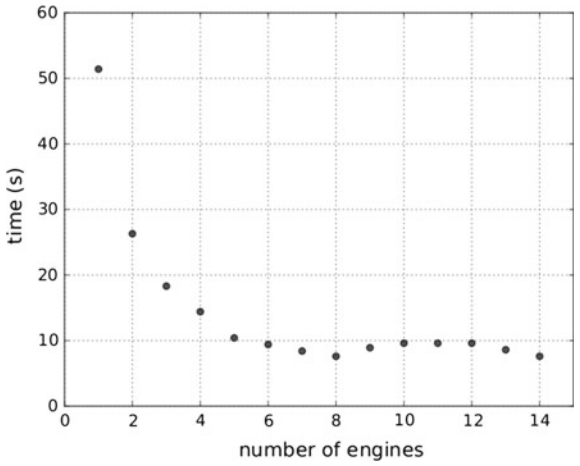

Fig. 11.4 Performance to process 100,000 lines for different numbers of engines

is close to the number of cores of the computer. Once the minimum is reached (in this case for eight cores) there is no benefit from parallelizing the job with more engines; on the contrary, with more processes, the operating system scheduler is going to spend more time managing processes so the execution time may increase. That is, the operating system scheduler may become a bottleneck. In addition, recall that the producer process in charge of distributing the data among the engines steals processing time from the engines.

11.5.2.3 Processing the Entire Dataset

With this optimal value of 2,000 for the lines per block variable we executed our algorithm over a whole CSV file made up of 14.7 million lines. The execution time with eight engines was 1009 seconds; and with four engines, that time increased to 1895 seconds.

As can be seen, increasing the number of engines by a factor of two does not divide the execution time by two. The reason of this can be explained by the fact that there is an additional process, the producer, that distributes the blocks of lines between the engines.

11.6 Conclusions

This chapter has focused on the parallel capabilities of IPython. As has been seen, IPython offers us an architecture that is capable of supporting many styles of parallelism, including multicore and distributed computing. In order to take advantage of such architecture, the user has to manually split the task to be performed into multiple subtasks. Each of these subtasks may then be executed on different engines.

The direct view offers the user the possibility of controlling which engine each task is sent to; whereas the load-balanced view leaves this issue to the scheduler. The former is useful if the tasks to be executed have similar computational cost or if a fine control over the tasks executed by each engine is needed. The latter is useful if the tasks have different computational costs and it does not matter which engine each task is executed on.

We used the IPython parallel capabilities to analyze a database made up of millions of entries. The tasks were created by dividing the database into chunks and assigning, in a cyclic manner, each of the chunks to an engine.

The framework explained in this chapter is not the only one currently available for IPython to take advantage of parallel computing capabilities. For instance, Hadoop and Apache Spark are cluster computing frameworks whose Application Programming Interface is available for the IPython notebook. Thus, these frameworks can be effectively used for data analysis.

Acknowledgements This chapter was co-written by Francesc Dantí and Lluís Garrido.

References

1. M. Herlihy, N. Shavit, *The art of multiprocessor programming* (Morgan Kaufmann, 2008)
2. T.K.G.B.G. Coulouris, J. Dollimore, *Distributed Systems* (Pearson, 2012)