Topics in Parallel and Distributed Computing:

Introducing Concurrency in Undergraduate Courses[1,2]

Chapter 3

Parallelism in Python for Novices

Steven Bogaerts
Department of Computer Science
DePauw University
stevenbogaerts@depauw.edu

Joshua Stough
Department of Computer Science
Washington & Lee University
stoughj@wlu.edu

[1]How to cite this book: Prasad, Gupta, Rosenberg, Sussman, and Weems. *Topics in Parallel and Distributed Computing: Introducing Concurrency in Undergraduate Courses*, 1st Edition, Morgan Kaufmann, ISBN : 9780128038994, Pages: 360.

[2]Free preprint version of the CDER book: http://grid.cs.gsu.edu/~tcpp/curriculum/?q=cedr_book.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 3

PARALLELISM IN PYTHON FOR NOVICES

Steven Bogaerts
Department of Computer Science
DePauw University
stevenbogaerts@depauw.edu

Joshua Stough
Department of Computer Science
Washington & Lee University
stoughj@wlu.edu

ABSTRACT

As a lightweight high-level language that supports both functional and object-oriented programming, Python provides many tools to allow programmers to easily express their ideas. This expressiveness extends to programming using parallelism and concurrency, allowing the early introduction of these increasingly critical concepts in the computer science core curriculum. Intended for instructors, this chapter provides pedagogical content on parallel programming in Python, including numerous illustrative examples and advice on presentation and evaluation.

Relevant core courses: CS0, CS1, CS2/DS

Relevant PDC topics: Concurrency (C), Tasks and threads (A), Decomposition into atomic tasks (A), Performance metrics (C), Divide & conquer (parallel aspects) (A), Recursion (parallel aspects) (A), Sorting (A)

Learning outcomes: The student should be able to apply the demonstrated techniques to new computational problems, and generally be able to maintain intelligent conversation on parallel computing.

Context for use: The very early introduction of parallel computing techniques in CS curricula and the reinforcement of concepts already covered in introductory curricula

3.1 Introduction

This chapter[1] provides practical advice and materials for instructors of novice programmers. The Python programming language and its `multiprocessing` module are used in a hands-on exploration of concepts of parallelism. The chapter starts with some background information, followed by specific materials and advice in how best to deliver them to students. Also note that many of the code examples are available online at [1], with more comments than are possible here.

3.2 Background

This section provides information about the target audience, our goals for the reader, and our reasons for selecting particular tools.

3.2.1 Target Audience of this Chapter

This chapter is intended for instructors of *novice* programmers. We expect this will most typically be instructors of CS1, CS2, and introductory computational science courses with a programming component. Instructors of other courses may also find some components of this material helpful, at least to organize an overview for their students.

3.2.2 Goals for the Reader

Our goals for the reader are as follows:

- The reader will share in our experience of prerequisites, best practices, and pitfalls of working with the `multiprocessing` module and integrating it into existing courses.
- Through numerous demonstration materials and example in-class exercises, the reader will be able to write programs and customize course materials that use

[1]Funding support in part from National Science Foundation grant CCF-0915805, SHF:Small:RUI:Collaborative Research: Accelerators to Applications – Supercharging the Undergraduate Computer Science Curriculum. Additional funding from NSF/TCPP CDER Center Early Adopter Awards and institutional sources.

`multiprocessing`, including fork/join, message passing, resource sharing, and locks.

- The reader will have experience with many practical applications of parallelism, including searching, sorting, pipelining, Monte Carlo simulations, and image processing.

3.2.3 Tools

Python Python is an increasingly popular choice as a first programming language in CS curricula. Its simple syntax allows students to quickly write interesting applications, minimizing time spent on syntactic rules. For example, Python requires no variable type declarations, variables are dynamic and can hold any time, garbage collection is automatic, and there is no explicit memory management. Clearly, students should learn such traditional programming mechanics at some point, but we argue that to require very early examination of such topics (i.e., in CS1) delays the exploration of interesting applications, possibly to the point that some students lose interest and do not take further computer science courses. We believe that interesting introductory experiences can be created in any programming language, but Python's simplicity greatly facilitates this effort.

We should mention also that we use Python 2.x in this chapter. We use 2.x here since, at time of writing, many third-party libraries still support only 2.x, and most current Linux distributions and Macs use 2.x as a default. This is not a significant problem, however, as the code examples in this chapter are essentially the same in Python 2.X and Python 3.X.

The `multiprocessing` Module Within the Python community, there are many tools available for the exploration of parallelism, including pprocess, Celery, MPI4Py, and Parallel Python [2]. Our primary need in this work is pedagogical clarity and ease of integration into existing courses, not all-out speed. For this reason we choose the `multiprocessing` module. This module is simpler than many other tools, features excellent documentation, and comes standard with Python 2.6+.

One small downside of the `multiprocessing` module is that it is not compatible with IDLE, the integrated development environment (IDE) that comes with Python installations.

The simplest workaround is to use a third-party IDE, such as spyder [3], for which this is not an issue. A listing of IDE's is available through the Python.org website [4].

3.3 Student Prerequisites

Here we discuss prerequisites for exploration of the `multiprocessing` module, both in terms of Python programming and a high-level overview of parallelism concepts. Some examples later in this chapter may have some additional prerequisites related to the area of application, as discussed in the context of those examples.

3.3.1 Motivating Examples Involving Parallelism Concepts

It is the experience of many instructors that students are not ready to absorb a new technique until it clearly solves a problem familiar to them. Fortunately for the introduction of concurrency and parallel computational techniques, *every* student will be very familiar with real-world situations that PDC thinking clearly addresses. Thus even in spite of the inevitable difficulties with implementation details (ameliorated though they are through Python), the PDC teacher is gifted with students who have been primed for this topic by their experience. Here we address some broad themes that invoke parallelism, along with particular scenarios that might serve as motivation. Just as with all of Computer Science, parallel algorithms and organization mimic human solutions to real-world problems.

- **Accelerating a large task by harnessing parallel hardware**: Real-world experiences exemplifying this theme include a group assembling a puzzle or toy, band or choir practice, or even any team sports. Among numerous computational examples involving this form of parallelism are animation (both film and video game), internet search, and data mining.

- **Communicating with remote entities that are inherently autonomous**: Everyone has experience with the concept of process coordination through message-passing, usually in the context of the real-world examples above. Additionally, many games (Go

Fish, Taboo, Trivial Pursuit) involve language or physical communication (Charades, Tag). While the students may not have previously thought about these scenarios as involving parallelism, the instructor may make the link by replacing key words, "process" for "player," or "pass a message" for "tell/say." In the world of computers, students can understand their use of email, web browsing, and social media clients as parallelism in this vein, with their device's client software communicating with the server software on the internet.

- **Isolating independent processes for security or resource management**: Most school cafeterias illustrate this concept through multiple independent counters or store fronts serving different fare, such as a pizza bar separate from the salad bar separate from the sandwich bar. This makes it possible for people who want pizza to avoid waiting behind the people who want a sandwich. Dealing with a particular line, Subway, Chipotle, Qdoba, and other establishments further use assembly-line production to improve efficiency, where isolated processes (employees) contribute to a job (order) and pass it along. Through this pipelining and the avoidance of context switching (no sandwich orders in the pizza line), a long line of customers can be served very quickly. This real-world organization is reflected in computers through chip design (memory unit separate from the register table separate from the arithmetic-logic unit) and the modularized organization of the computer itself.

The above provides the reader with some jumping-off points to motivate the use of parallelism, integral in Computer Science just as in our corporal reality. To further elicit this motivation in the students the authors are particularly fond of the physical exercises in [5]. Depending on time constraints and preferences, this can be covered in one to three hours of in-class time.

3.3.2 Python Programming

The bare minimum Python prerequisites for exploring the `multiprocessing` module are actually very few: just printing, variables, tuples, and writing and using functions with key-

word arguments. Theoretically, then, students could begin exploring the `multiprocessing` module extremely early in their study of programming (i.e., within the first few weeks). However, there are two problems with starting so early.

First, such limited programming ability would severely restrict what applications are feasible. Since an important part of this work is to apply parallelism to interesting problems, it would be wise to provide some introduction in traditional early topics like loops and conditionals.

Second, the `multiprocessing` module works through the use of classes (e.g., a Process class, a Lock class, etc.). Students are not required to understand object-oriented programming to use the `multiprocessing` module – they could simply be told "this is the syntax to make a new process", rather than "this is how you call the Process *constructor*, a special *method* defined in the Process *class*." Our experience has shown, however, that this would make student learning much more challenging. Thus it is preferable for students to have a small amount of experience in working with classes as well, before beginning a study of the `multiprocessing` module. The experience should cover the notion that a class defines methods and object state, constructors, method invocation, and that methods can modify state.

It is important to emphasize that only *introductory* experience in the above topics is necessary. That is, students should know the basics of what a loop is; they do not need to have mastered multiple kinds, less common forms, etc. Students should know some basics about classes as described above, but they do not need to understand object-oriented design and the language constructs that facilitate good designs. Thus, we expect that most introductory programming courses would be ready to consider the `multiprocessing` module after 50% – 75% of the course has been completed.

3.4 General Approach: Parallelism as a Medium

This section describes our general approach for integrating parallelism into introductory courses, based on work first reported in [6]. Detailed materials and explanation under this

approach will be provided following this section.

In considering how to integrate parallelism into introductory courses, it is useful to make an analogy. Consider object-oriented programming (OOP). At its core, OOP is a different paradigm from imperative programming, the primary paradigm in use decades ago. As OOP was developed, we can imagine posing questions similar to those we face today regarding parallelism: how can OOP be integrated into introductory programming courses? Is this wise, or even possible?

Of course there are still many variations in curricula, but in general we can see how these questions have been answered. While there is a place for a high-level OOP course, object-oriented concepts are by no means relegated only to such a course. CS1 typically includes the use of objects and basic class construction. A data structures course often includes the creation of abstract data types with classes. Graphics courses can make use of abstraction through classes as well as through functions. The inclusion of these OO topics has necessitated some additional time to learn mechanics, but it is fair to say that many of the topics of these courses are simply being taught through the medium of OO now, rather than solely through the medium of imperative programming. Furthermore, while perhaps some sacrifices have been made, most key concepts of imperative programming have not been sacrificed to achieve this OOP coverage.

We argue that the same can and will be said for questions of parallelism education. Our approach for integrating parallelism into introductory courses while not sacrificing large amounts of traditional content is to recognize parallelism as a complementary *medium* for learning various computer science topics. This does require some additional background in basic mechanics, but once these basics are covered, parallelism can be used in combination with traditional approaches for learning computer science. The key is that this can be done without significant elimination of other material; rather, other material is simply learned through the medium of parallelism. We will consider this theme explicitly in many of the course materials provided below.

3.5 Course Materials

The remaining sections of this chapter contain many examples, along with an explanation of what is happening and some tips on presenting the material in a course.

Also included with most examples is a list of "key ideas". We find that after an in-class example, it is useful to give students some time in class to enter into their copy of the file what they believe are the key ideas for that example. Students can then share key ideas with the class as a whole. While this explicit consideration of key ideas does take extra class time and may seem repetitious to the instructor, we find that it is worth it in the long run; students retain the material much better and are thus better prepared for later material, and therefore require less time reviewing prerequisite concepts. Some sample key ideas are provided for most concepts below, demonstrating what students should take away from the example. These are key ideas taken directly from students, meaning that occasionally some ideas listed were actually first introduced in earlier examples. The presence of such "review" key ideas reinforces the notion that students may need regular reminders of certain concepts. The sequence of examples is designed to provide these reminders.

Some course materials below are presented as in-class lecture-oriented examples, while others are presented as in-class exercises. We try to balance the two, keeping students as active as possible through the exercises while still providing adequate scaffolding through lecture. For the exercises, we will assume that students have access to computers in class, though the exercises could be adapted to individual pencil-and-paper exercises or class-wide collaborative activities.

3.6 Processes

Before getting into programming, a very brief discussion of processes is useful. While the reality of the execution of processes on various architectures can be quite complex, a simplified overview is all that is required here. The instructor can explain that a process is a running program, in which the current instruction and the data is maintained. On a

single-core processor, only one process actually runs at a time. This may be surprising to students, as they are accustomed to having many applications active at once. But the reality is that the operating system actually switches from one to the next and back very quickly via *context switches*, giving the *illusion* of multiple processes executing at once. Such context switching, sharing the single processor resource, is an example of *concurrency*. It can be fun to view a list of running processes on a machine (`crtl-alt-delete` to the Task Manager in Windows, for example) as an illustration of how this works.

This is in contrast to a multicore processor, in which multiple processes can be executed literally at the same time (limited by the number of cores, of course). Thus true *parallelism* is achieved in this case, rather than simply the illusion that concurrency provides. Of course, the reality of all this is more complex, but this is a useful abstraction at this level.

A potentially helpful side-note can be made here, comparing concurrency with the common human effort at multitasking. People often believe that in multitasking (e.g., texting and listening to a lecture...) that they are working in parallel, truly doing both tasks at the same time. Psychological studies suggest, however, that the reality is closer to concurrency – multitasking through context switches – as most "multitasker's" performance on both tasks suffers. This can be a revelation in study skills for students who are willing to take the message to heart.

3.6.1 Spawning a Process

We are now ready to work with the `multiprocessing` module itself. The first step is to import the module with `from multiprocessing import *`. Students can then view the first example, in Figure 3.1. Note that `current_process` returns a `Process` object corresponding to the currently running process. The `Process` class defines a `pid` instance variable for the process identifier. The `print` at line 5 prints the `pid` of the default *parent* process for the program.

Line 10 then constructs, but does not start, a *child* process. Note the use of keyword arguments. The `target` is `sayHi`, meaning that the child process, once started, will execute

the `sayHi` method. The `args` argument is a tuple of the arguments to pass to the `target`; since `sayHi` takes no arguments, `args` is an empty tuple in this example. Note the inclusion of comments for the assistance of the student in this constructor call.

Finally, the child process is started with the `start` method. At this point, both the parent and child processes want to run. On a multicore system, they can run in parallel; on a single-core system, they will run concurrently.

Immediately in this example we can see the idea of parallelism as a *medium* for traditional course topics. Ideally, at this point students will have had some experience with using classes, constructors, other methods, and instance variables. And yet they might not yet be fully comfortable with these ideas, and/or may not see their value. This early example gives students additional practice with these concepts. Through the `Process` class, a very significant amount of functionality is gained for very little programming effort. The `Process` class interacts with the operating system to create a new process registered with the scheduler, ready to access shared processor resources. This is an excellent illustration of the power of encapsulation and object-oriented programming.

Sample Key Ideas

- To create a child process, call the `Process` constructor, specifying a target (the function to be run – without parentheses), and also specifying the arguments that will go to that target function.
- The `Process` class has many instance variables, including `pid` and `name`, which are public and can be used to identify the process. `current_process().pid` gives the pid of the currently-running process (same idea for name).
- `current_process()` is a function defined in the `multiprocessing` module. It is an expression, giving a `Process` object representing the currently running process.
- The `Process` class has a start method that must be called for the process to start doing its task.

```
1  def sayHi():
2    print "Hi from process", current_process().pid
3
4  def procEx():
5    print "Hi from process", current_process().pid, "(main process)"
6
7    # Construct the process (but it doesn't start automatically).
8    # target must be a function, without the parentheses.
9    # args must be a tuple, containing the arguments to pass to the target,
10   #    or () if no arguments.
11   otherProc = Process(target=sayHi, args=())
12
13   # Start the process we just constructed.
14   otherProc.start()
```

Figure (3.1) Spawning a Single Process

3.6.2 Spawning Multiple Processes

Immediately after the example above, a short active exercise is helpful:

> Copy the "Spawning a Single Process" example above, and modify to create 3 processes, each of which says hi as above.

A solution is provided in Figure 3.2. It is important to point out to the students that each process uses the same `sayHi` function defined above, but each process executes that function independent of the others.

While this exercise is just a trivial extension of the previous example, we find it helpful to give the students hands-on experience in the `multiprocessing` module as soon as possible. This short exercise can give an early successful experience to the students.

Sample Key Ideas

- You can make multiple child processes simply by calling the Process constructor multiple times. These processes are independent of each other.

Immediately after this exercise, a trivial extension can introduce the passing of arguments to a child process' function, as shown in Figure 3.3.

```
1  def procEx2():
2    print "Hi from process", current_process().pid, "(main process)"
3
4    p1 = Process(target=sayHi, args=())
5    p2 = Process(target=sayHi, args=())
6    p3 = Process(target=sayHi, args=())
7
8    p1.start()
9    p2.start()
10   p3.start()
```

Figure (3.2) Exercise solution for spawning multiple processes

```
1  def sayHi2(n):
2    print "Hi", n, "from process", current_process().pid
3
4  def manyGreetings():
5    print "Hi from process", current_process().pid, "(main process)"
6
7    name = "Jimmy"
8    p1 = Process(target=sayHi2, args=(name,))
9    p2 = Process(target=sayHi2, args=(name,))
10   p3 = Process(target=sayHi2, args=(name,))
11
12   p1.start()
13   p2.start()
14   p3.start()
```

Figure (3.3) Passing arguments to child processes

Note that `sayHi2` takes a single argument, and that each call to the `Process` constructor passes a tuple with a single value inside of the `args` parameter. It may also be helpful here to remind students of the purpose of the comma in `(name, )`. Python requires this in order to interpret the parentheses as markers of a tuple rather than simple grouping.

Sample Key Ideas

- When you create multiple processes, they run independently of each other.
- The args tuple can be set to contain whatever data you want to pass to the target function.
- The args tuple must contain exactly as many items as the target takes arguments. The

arguments must be provided in the correct order (you can't use the keyword arguments style here).

3.6.3 Spawning Multiple Processes Using `Pool`

```
1  from multiprocessing import Pool, current_process
2
3  def sayHi2(n):
4      print('Hi ' + n + ' from process ' + str(current_process().pid))
5
6  def manyGreetings():
7      print('Hi from process ' + str(current_process().pid) + ' (main process)')
8
9      name = 'Jimmy'
10     p = Pool(processes = 3)
11
12     p.map(sayHi2, [name, name, name])
13
14 if __name__ == '__main__':
15     manyGreetings()
```

Figure (3.4) Using a `Pool` to perform the example of Figure 3.3.

While the `Process` constructor is useful for explicitly spawning independent processes, the `multiprocessing` module also offers multiple process spawning through the use of the `Pool` and `Pool.map` mechanism, as seen in Figure 3.4. Two syntactic constraints to this usage are first, that all processes execute the same function (`sayHi2` above), and second, that this function accept only one argument. (The second argument of `map` at line 12 is a list of the single arguments to distribute to the processes.) These are weak constraints however, in that both can be easily circumvented. Also in this simple example, the return argument of `Pool.map` is unassigned at line 12, though it would be the list of return arguments of the three independent expressions `sayHi2(name)`, or `[None, None, None]`. See Section 3.9 for more complicated and useful examples of the `Pool/map` paradigm, including Monte Carlo simulation, integration, and sorting.

```
1 def manyGreetings2():
2   name = raw_input("Enter your name: ")
3   numProc = input("How many processes? ")
4
5   for i in range(numProc):
6     #p = Process(target=sayHi2, args=(name,))
7     #p.start()
8     (Process(target=sayHi2, args=(name,))).start()
```

Figure (3.5) "Anonymous" processes

3.6.4 Anonymous Processes

At this point, we recommend another exercise:

> Write a function that first asks for your name, and then asks how many processes to spawn. That many processes are created, and each greets you by name and gives its pid.

See Figure 3.5 for a solution. Many students may devise a solution involving the commented-out approach (storing a new `Process` object in `p` and then calling `p.start()`), though for the purposes of this problem, the code in line 8 is more compact. While this may seem hardly worth a separate exercise to an experienced programmer, the *medium* of parallelism in this example actually offers a wealth of traditional topics for novice programmers to consider.

First, we find that novice programmers do not readily transfer knowledge from one context to another without explicit practice. It is not uncommon to find student attempts like:

```
1   for i in range(numProc):
2     pi = Process(target=sayHi2, args=(name,))
3     pi.start()
```

This would work, as it merely substitutes variable `p` in Figure 3.5 for `pi`. However, students may believe that this is actually creating several variables, `p0`, `p1`, `p2`, etc. This is not due to a complete lack of understanding of how loop variables work. Rather, it is a

failure to transfer knowledge to a new context. Students can be quickly corrected with the following example:

```
1  for a in range(10):
2    grade = 97
```

It will likely be obvious to students that this code does not generate the variables `gr0de`, `gr1de`, `gr2de`, etc. Similarly, then, this code analogy should make it clear that `pi` does not become `p0`, `p1`, `p2`, etc.

This exercise provides opportunities to consider other important concepts as well. For example, which approach is better, the explicit use of `p`, or the anonymous version? Which is more readable? Should the program be designed such that the programmer can refer to the object later, thus necessitating `p` or perhaps a list of all constructed processes? Students may not yet be accustomed to considering such questions in a wide variety of contexts, so this practice is very important.

This example also reinforces the notion that the `Process` constructor is an *expression* when called, and can be used as any other expression. Introductory programmers sometimes learn to do things more by rote than through deep understanding. A student might think "To make an instance of a class, I have to call the constructor and store it in a variable." Of course, in reality the programmer has more flexibility, and it is important for students to see this in various contexts.

The instructor can remind students that the `start` method is a *statement* when called. This means that `(Process(target=sayHi2, args=(name,))).start()` or `p.start()` are acceptable by themselves on a line. Furthermore, `p.start().pid`, for example, will *not* work. It would only work if the `start` method were an expression when called, returning a `Process` object. But this is not the case. Students will likely have wrestled with these ideas of expressions and statements earlier in the course, and this is another excellent opportunity to consider them in detail in a new context. The issue of expressions vs. statements is not just some unimportant distinction designed to give professors something to test students on. Rather, the understanding of this concept enables programmers to more effectively use

unfamiliar tools like the `multiprocessing` module!

Thus we see that, while the example is at its surface about creating processes, the most important lessons are about more general, traditional programming topics. These topics would need to be discussed in any event in introductory programming courses. Parallelism provides a very practical *medium* in which to do so.

Sample Key Ideas

- A process (or any object) doesn't have to be stored in a variable to be used. But if you don't store it in a variable, you won't be able to use it after the first time. An object not stored in a variable is called an anonymous object.
- If you need to access the process objects later, you could store them in a list when you make them. To do this, make a list accumulator (starting with []) that you append Process objects onto, so that you can loop through the list later.
- Once a process has been started, changing the variable that stored it won't stop the process.
- For loops don't substitute values into the middle of variable names.

3.6.5 Specifying Process Names

The next example (Figure 3.6), on its surface is about a small piece of `multiprocessing` functionality: specifying a name for a process. The more important lesson, using parallelism as a *medium* for traditional course topics, is that `name` is an instance variable of the `Process` class, thus providing a reminder of what instance variables are in general.

Another important issue brought up in this example is the idea of *resource contention*. It is likely that the output of this example will be very jumbled up; the reason why, and a solution, is discussed in the next example.

```
1  def sayHi3(personName):
2    print "Hi", personName, "from process", current_process().name, "-
3  pid", current_process().pid
4
5  def manyGreetings3():
6    print "Hi from process", current_process().pid, "(main process)"
7
8    personName = "Jimmy"
9    for i in range(10):
10     Process(target=sayHi3, args=(personName,), name=str(i)).start()
```

Figure (3.6) Specifying process names

3.6.6 Using a Lock to Control Printing

It is now time to consider an important part of parallelism: controlling access of shared resources. One excellent way to begin this process is by analogy to a concept from the novel Lord of the Flies by William Golding (or the 1963 and 1990 film adaptations). The novel tells the story of a group of boys shipwrecked on a deserted island with no adult survivors. Before an eventual breakdown into savagery, the boys conduct regular meetings to decide on issues facing the group. The boys quickly realize that, left unchecked, such meetings will be unproductive as multiple boys wish to speak at the same time. Thus a rule is developed: Only the boy that is holding a specially-designated conch shell is allowed to speak. When that boy is finished speaking, he relinquishes the conch so that another boy may speak. Thus order is maintained at the meetings as long as the boys abide by these rules. We can also imagine what would happen if this conch were not used: chaos in meetings as the boys try to shout above each other. (And in fact this does happen in the story.)

It requires only a slight stretch of the events in this novel to make an analogy to the coordination of multiple processes accessing a shared resource. In programming terms, each boy is a separate process, having his own things he wishes to say at the meeting. But the air around the meeting is a shared resource - all boys speak into the same space. So there is contention for the shared resource that is this space. Control of this shared resource is handled via the single, special conch shell. The conch shell is a *lock* – only one boy may hold

it at a time. When he releases it, indicating that he is done speaking, some other boy may pick it up. Boys that are waiting to pick up the conch can't say anything – they just have to wait until whoever has the conch releases it. Of course, several boys may be waiting for the conch at the same time, and only one of them will actually get it next. So some boys might have to continue to wait through multiple speakers.

The code in Figure 3.7 shows the analogous idea in Python. Several processes are created and started. Each wants to `print` somthing, in `sayHi4`. But `print` writes to `stdout` (standard output), a single resource that is shared among all the processes. So when multiple processes all want to print at the same time, their output would be jumbled together were it not for the lock, which ensures that only one process is able to execute its print at a time. Suppose process A `acquire`s the lock and begins printing. If processes B, C, and D then execute their `acquire` calls while A has the lock, then B, C, and D each must wait. That is, each will *block* on its `acquire` call. Once A `release`s the lock, one of the processes blocked on that lock acquisition will arbitrarily be chosen to acquire the lock and print. That process will then release the lock so that another blocked process can proceed, and so on.

Note that the lock must be created in the parent process and then passed to each child – this way each child process is referring to the same lock. The alternative, in which each child constructs its own lock, would be analogous to each boy bringing his own conch to a meeting. Clearly this wouldn't work.

It is important to point out to students that the order of execution of the processes is arbitrary. That is, the acquisition of the lock is arbitrary, and so subsequent runs of the code in Figure 3.7 are likely to produce different orderings. It is not necessarily the process that was created first, or that has been waiting the longest, that gets to acquire the lock next.

Sample Key Ideas

- Locks prevent multiple processes from trying to do something at the same time that they shouldn't. For example, multiple processes should not try to print (access stdout) at the same time.

```
1  def sayHi4(lock, name):
2    lock.acquire()
3    print "Hi", name, "from process", current_process().pid
4    lock.release()
5
6  def manyGreetings3():
7    lock1 = Lock()
8
9    print "Hi from process", current_process().pid, "(main process)"
10
11   for i in range(10):
12     Process(target=sayHi4, args=(lock1, "p"+str(i))).start()
```

Figure (3.7) A demonstration of a lock to handle resource contention

- Define the lock in the parent process, so it can be passed to all the children.
- Don't forget to use lock.release() sometime after every lock.acquire(), otherwise any other processes waiting for the lock will wait forever.

3.6.7 Digging Holes

The final example in this section is a simple exercise extending the concept of locks above:

> Imagine that you have 10 hole diggers, named A, B, C, D, E, F, G, H, I, and J. Think of each of these as a process, and write a function `assignDiggers()` that creates 10 processes with these worker names working on hole 0, 1, 2, ..., 9, respectively. Each one should print a message about what it's doing. When you're done, you should get output like the following (except that the order will be arbitrary):
>
> ```
> >>> assignDiggers()
> >>>
> Hiddy-ho!  I'm worker G and today I have to dig hole 6
> Hiddy-ho!  I'm worker A and today I have to dig hole 0
> ```

```
Hiddy-ho!  I'm worker C and today I have to dig hole 2
Hiddy-ho!  I'm worker D and today I have to dig hole 3
Hiddy-ho!  I'm worker F and today I have to dig hole 5
Hiddy-ho!  I'm worker I and today I have to dig hole 8
Hiddy-ho!  I'm worker H and today I have to dig hole 7
Hiddy-ho!  I'm worker J and today I have to dig hole 9
Hiddy-ho!  I'm worker B and today I have to dig hole 1
Hiddy-ho!  I'm worker E and today I have to dig hole 4
```

A solution is provided in Figure 3.8. This exercise provides a good medium to discuss the strengths and limitations of different looping constructs. Note that the provided solution uses a "loop by index" approach, in which the `holeID` index is the loop variable. An alternative that some students might attempt would be "loop by element" (a for-each loop), as in Figure 3.9. The issue here is that `workerNames.index(workerName)` is a linear time operation, so this is a far less efficient approach. Do note that actual execution time will be nearly instantaneous in both approaches, but it is nevertheless a good idea to reiterate the general principle of using the right programming constructs for maximum efficiency. Novice programmers may not be accustomed to considering such things, and so this is a nice opportunity to explore this important introductory programming concept arising in the medium of parallelism.

Sample Key Ideas

- If it doesn't matter what order the processes run in, then code like in this example can run very efficiently.
- This is a good example of a situation where you need to choose carefully between looping by index and looping by element. By index is ideal here.

```
1  def dig(workerName, holeID, lock):
2    lock.acquire()
3    print "Hiddy-ho!  I'm worker", workerName, "and today I have to dig hole",
         holeID
4    lock.release()
5
6  def assignDiggers():
7    lock = Lock()
8    workerNames = ["A", "B", "C", "D", "E", "F", "G", "H", "I", "J"]
9
10   for holeID in range(len(workerNames)):
11     Process(target=dig, args=(workerNames[holeID], holeID, lock)).start()
```

Figure (3.8) Digging holes: practice with locks

```
1    ... # Other code as before
2    for workerName in workerNames:
3      Process(target=dig, args=(workerName, workerNames.index(workerName),
           lock)).start()
```

Figure (3.9) Digging holes: a less-effective loop-by-element approach

3.7 Communication

This section provides several examples of interprocess communication. As in the other sections of this chapter, the purpose is not to exhaustively cover every possibility, but rather to allow students to gain some basic exposure. While the `multiprocessing` module does support some shared memory mechanisms, message passing is the primary means of communication for the module, and so that is the paradigm we recommend exploring in the limited time of an introductory course.

3.7.1 Communicating Via a Queue

The authors find it help to start immediately with a simple example. Figure 3.10 demonstrates the use of a *queue* for communication between processes. Note that the parent process creates the queue and passes it as an argument to the child process. So the child process is in fact using the same queue as the parent. In this case, the child will only receive (`get`) while the parent will only send (`put`). The child's `get` is a *blocking* command. This

```
1  def greet(q):
2    print "(child process) Waiting for name..."
3    name = q.get()
4    print "(child process) Well, hi", name
5
6  def sendName():
7    q = Queue()
8
9    p1 = Process(target=greet, args=(q,))
10   p1.start()
11
12   time.sleep(5) # wait for 5 seconds
13   print "(main process) Ok, I'll send the name"
14   q.put("Jimmy")
```

Figure (3.10) The use of a queue for communication between processes

means that the child process will go to sleep until it has a reason to wake up – in this case, that there is something to `get` off the queue. Since the parent sleeps for 5 seconds, the child ends up blocking for approximately 5 seconds as well. Finally the parent process sends the string `"Jimmy"`, the child process unblocks and stores `"Jimmy"` in the variable `name`, and prints.

Note also the addition of some print statements to give the programmer a visual indication of what is happening while the code is running. This is an important programming and debugging skill for introductory programmers to practice.

Sample Key Ideas

- A queue is like standing in line. The first thing in is the first thing out.
- `put` and `get` can be used for any processes to communicate by sending and receiving data.
- If a process tries to get from an empty queue, it will sleep until some other process puts something on the queue.

3.7.2 Extended Communication Via a Queue

After the simple example above, it is helpful to provide an immediate opportunity for student practice, even for a very small expansion of the problem. This enables students to do something active, thereby facilitating the transfer of these concepts into long-term memory. The following exercise is provided:

> Copy the code above as a basis for greet2 and sendName2. Modify the code so that greet2 expects to receive 5 names, which are sent by sendName2. Each function should accomplish this by sending/receiving one name at a time, in a loop.

While this exercise seems rather simple to experienced programmers, the newness of queues is sufficient to give many students pause. Thus this exercise provides an excellent opportunity to remind students of the utility of pseudocode. This crucial problem-solving technique frees students from most concerns of syntax, and so the overall program structure is then not too difficult to determine. And so, part way through this exercise, it is useful to work with the students in developing the pseudocode in Figure 3.11. Once this pseudocode is developed, the actual code comes much more easily. Figure 3.12 shows a solution.

```
'''
def greet2():
  for 5 times
    get name from queue
    say hello

def sendName2():
  queue
  make a child process, give it the queue
  start it

  for 5 times
    sleep for a bit
    put another name in the queue
'''
```

Figure (3.11) Pseudocode for an extended exercise for communicating via a queue

```
1  from random import randint
2
3  def greet2(q):
4    for i in range(5):
5      print
6      print "(child process) Waiting for name", i
7      name = q.get()
8      print "(child process) Well, hi", name
9
10 def sendName2():
11   q = Queue()
12
13   p1 = Process(target=greet2, args=(q,))
14   p1.start()
15
16   for i in range(5):
17     sleep(randint(1,4))
18     print "(main process) Ok, I'll send the name"
19     q.put("George"+str(i))
```

Figure (3.12) Python solution for an extended exercise for communicating via a queue

Sample Key Ideas

- Sometimes you can get by without using locks if processes are waiting for other reasons.
- `import moduleName` means I have to say `moduleName.functionName`.

  `from moduleName import functionName` means I can just say `functionName`

3.7.3 The Join Method

In parallel programming, a *join* operation instructs the executing process to block until the process on which the join is called completes. For example, if a parent process creates a child process in variable `p1` and then calls `p1.join()`, then the parent process will block on that join call until p1 completes. One important point to emphasize again in this example is that the *parent* process blocks, not the process on which `join` is called (`p1`). Hence the careful language at the start of this paragraph: the executing process blocks until the process on which the join is called completes.

The word "join" can be confusing for students sometimes. The example in Figure 3.13 provides a very physical analogy, of the parent process waiting (using `join`) for a "slowpoke"

```
1  def slowpoke(lock):
2    sleep(10)
3    lock.acquire()
4    print "Slowpoke: Ok, I'm coming"
5    lock.release()
6
7  def haveToWait():
8    lock = Lock()
9    p1 = Process(target=slowpoke, args=(lock,))
10   p1.start()
11   print "Waiter: Any day now..."
12
13   p1.join()
14   print "Waiter: Finally! Geez."
```

Figure (3.13) Using join for coordination between processes

child process to catch up. The child process is slow due to the `sleep(10)` call. Note also the use of a lock to manage the shared `stdout`.

It should be pointed out, however, that `join` is not always necessary for process coordination. Often a similar result can be obtained by blocking on a queue `get`, as described in Sections 3.7.4 and 3.7.5.

Sample Key Ideas

- p.join() makes the currently-running process wait for p to finish. Be careful! It's not the other way around.
- join will slow things down (since it makes a process wait), but it can be used to enforce a particular order of activity.
- Anonymous processes can't be used with join, because you have to both start it and join with it, which means you'll need a variable to be able to refer to the process.

3.7.4 Obtaining a Result from a Single Child

While earlier sections demonstrated a parent process sending data to a child via a queue, this exercise allows students to practice the other way around: a child that performs a

```
1  def addTwoNumbers(a, b, q):
2    # sleep(5) # In case you want to slow things down to see what is happening.
3    q.put(a+b)
4
5  def addTwoPar():
6    x = input("Enter first number: ")
7    y = input("Enter second number: ")
8
9    q = Queue()
10   p1 = Process(target=addTwoNumbers, args=(x, y, q))
11   p1.start()
```

Figure (3.14) Starter code for the exercise on getting a result from a child process

computation which is then obtained by the parent. Consider two functions: `addTwoNumbers`, and `addTwoPar`. `addTwoNumbers` takes two numbers as arguments, adds them, and places the result on a queue (which was also passed as an argument). `addTwoPar` asks the user to enter two numbers, passes them and a queue to `addTwoNumbers` in a new process, waits for the result, and then prints it.

The starting code provided to the students is shown in Figure 3.14. The completed exercise is in Figure 3.15. We choose to use starter code here because, as an in-class exercise with limited time available, it is important to get to the key points addressed by the example. Thus the starter code enables students to bypass side issues. Of course, given more time, these "side issues" could provide good review, or a relatively short homework problem, and so each instructor should consider personal goals and time allotted for the exercise.

The parent's use of `q.get()` is important, but is review from previous exercises. The new idea here is that, although `join` was just introduced, it should not be used thoughtlessly. `join` is not necessary in this example, because `get` will already cause the parent to block. The parent will not unblock until the child executes `put`, which is the child's last action. Thus, in this example, `get` is already accomplishing essentially what `join` would accomplish, while also obtaining the child's result.

Also note the commented-out `sleep` in the child process, which can be useful for seeing more clearly the sequencing created through the queue operations.

```
1  def addTwoNumbers(a, b, q):
2    # sleep(5) # In case you want to slow things down to see what is happening.
3    q.put(a+b)
4
5  def addTwoPar():
6    x = input("Enter first number: ")
7    y = input("Enter second number: ")
8
9    q = Queue()
10   p1 = Process(target=addTwoNumbers, args=(x, y, q))
11   p1.start()
12
13   # p1.join()
14   result = q.get()
15   print "The sum is:", result
```

Figure (3.15) Complete code for the exercise on getting a result from a child process

Sample Key Ideas

- This illustrates the bigger concept of making a child process to do some (possibly complex) task while the parent goes on to other things. You could even have multiple child processes working on parts of the problem.
- You don't need the join here, because the q.get() will wait until the child puts something on the queue anyway.

3.7.5 Using a Queue to Merge Multiple Child Process Results

The example in Figure 3.16 is a fairly straightforward extension of the one in Section 3.7.4. Here, two child processes are created, each given half of the work. The results are then merged by the parent through two `get` calls.

It can be interesting to ask the students which child's result will be in `answerA` and which in `answerB`. The answer is that this is indeterminate. Which child process finishes first will have its answer in `answerA`, and the other will be in `answerB`. This is not a problem for commutative merging operations, like the addition of this example, but of course could be a complication for non-commutative merging.

As in the previous exercise, the `join`s are not necessary and should not be used, so they

```
1  from random import randint
2  import time
3  def addManyNumbers(numNumbers, q):
4      s = 0
5      for i in range(numNumbers):
6          s = s + randint(1, 100)
7      q.put(s)
8
9  def addManyPar():
10     totalNumNumbers = 1000000
11
12     q = Queue()
13     p1 = Process(target=addManyNumbers, args=(totalNumNumbers/2, q))
14     p2 = Process(target=addManyNumbers, args=(totalNumNumbers/2, q))
15     startTime = time.time()
16     p1.start()
17     p2.start()
18
19 #     p1.join()
20 #     p2.join()
21
22     answerA = q.get()
23     answerB = q.get()
24     endTime = time.time()
25     timeElapsed = endTime - startTime
26     print "Time:", timeElapsed
27     print "Sum:", answerA + answerB
```

Figure (3.16) Demonstrates a parent process obtaining results from children

are commented out.

The use of some rudimentary timing mechanisms in this example is a precursor to further considerations of speedup in some examples below.

Sample Key Ideas

- This demonstrates the idea of splitting a task up evenly among multiple processes. Each one reports back to the parent processes via a queue.

3.7.6 Mergesort Using `Process` Spawning and `Queue` Objects

This subsection demonstrates parallel mergesort, extending the simpler example of Figure 3.16 above. There, each child process produces a single integer response, and the parent

```
#Sequential mergesort code
def merge(left, right):
    ret = []
    li = ri = 0
    while li < len(left) and ri < len(right):
        if left[li] <= right[ri]:
            ret.append(left[li])
            li += 1
        else:
            ret.append(right[ri])
            ri += 1
    if li == len(left):
        ret.extend(right[ri:])
    else:
        ret.extend(left[li:])
    return ret

def mergesort(lyst):
    if len(lyst) <= 1:
        return lyst
    ind = len(lyst)//2
    return merge(mergesort(lyst[:ind]), mergesort(lyst[ind:]))
```

Figure (3.17) Sequential mergesort code. Referred to as `seqMergesort` in Figures 3.28 and 3.27.

process combines the responses of the children. Here, each child process produces a list and the parent process merges the produced lists into a larger list. Figure 3.17 shows sequential mergesort code, while Figure 3.28 shows a parallel implementation using `Process` spawning and communication via `Queue` objects.

Mergesort is a deceptively simple procedure to intuitively understand, while at the same time the code is so simple it can make the student feel they must be missing something. Mergesort is simply "sort the left half; sort the right half; merge the results." One physical exercise the authors find useful involves sorting a deck of cards. To begin, sorting the entire deck as a whole is slow, given that at this stage in the curriculum only $O(N^2)$ sorts have been covered. However, if the instructor splits the deck in two parts and distributes one part to each of two students in the front row, the students quickly see the efficiency implied. If they do the same with their half-decks, distributing the work to students behind them, then soon many independent workers are sorting very small decks. As these students finish,

the merging operation takes place as the cards make their way back to the front. In-class discussion of this process determines that relative to the $O(N^2)$ sorts, speedup is obtained both by doing less work and by parallelizing that work among multiple processes/students.

The code of Figure 3.28 demonstrates how interactions of the physical exercise can be mapped almost directly to Python. The `mergeSortParallel` process, that represents any particular student in the physical exercise, receives as parameters both the list it is responsible for sorting and the `Queue` object on which it will put the sorted result. A third, integer parameter merely serves to limit the number of recursive process instantiations: i.e., can this student recruit two others or should they perform the sort themselves?

As an aside, the sequential mergesort of Figure 3.17 uses $O(N \log N)$ memory rather than the ideal $O(N)$, as a result of list slicing in argument passing. While not ideally efficient, this implementation allows the student to move easily from the intuitively understood mergesort idea to the code. Finally, see Section 3.9.3 for a 'flattened' mergesort implementation using a `Pool` of processes.

3.7.7 Sharing a Data Structure

The examples in Figures 3.18, 3.19, and 3.20 illustrate that memory is not shared between processes, so mutation of arguments does not occur in the same way as in a single-process program. First, Figure 3.18 reviews and reinforces the concept that in Python, object references are passed, and so mutations will be seen by the caller. That is, the changes to `ls` that `addItem` makes will be reflected in the `print` in `sequentialDS`.

Figure 3.19 then attempts to accomplish the same idea by passing a list to two child processes. If this worked similarly to the sequential version, then both child processes would append to the same list, and the parent would then print a list with two items. However, this is not what occurs. The actual result is that the parent prints an empty list. Thus the child processes do not share `ls`, they receive copies. These copies are mutated, but then thrown away when the child processes terminate. The parent process's `ls` is not affected.

Finally, Figure 3.20 demonstrates one correct way to accomplish this task. The use of

a queue as demonstrated here should be review. The necessity of the queue, however, may be new to the students, hence the utility of this example.

```
1 import random
2 def addItem(ls):
3   ls.append(random.randint(1,100))
4
5 def sequentialDS():
6   ls = []
7   addItem(ls)
8   addItem(ls)
9   print ls
```

Figure (3.18) Sequential program showing mutation of a parameter, to serve as a reminder to students.

3.8 Speedup

These examples demonstrate how to conduct basic timing in Python. This is then applied to examine the speedup of the use of multiple processors on a problem.

3.8.1 Timing the Summation of Random Numbers

The example in Figure 3.21 introduces students to the notion of timing the execution of a part of a program, and the generation of random numbers in Python. `time.time()` returns a float representing the number of seconds since the *epoch*, which in Python is defined as midnight UTC on 1/1/1970. Thus, the elapsed time can easily be computed as shown in `timeSum`, or with a small "trick" in `timeSum2` that saves the use of a `stopTime` variable.

3.8.2 Using Join to Time a Child Process

At this point, students may already be familiar with `join` from previous examples, but Figure 3.22 is a helpful illustration of its utility in timing a child process's activity. The start time is measured when the child process is started. Once the parent's `join` call unblocks, the elapsed time is immediately calculated.

```
# Doesn't work as you might hope!
def parallelShareDS1():
    ls = []

    p1 = Process(target = addItem, args = (ls,))
    p2 = Process(target = addItem, args = (ls,))
    p1.start()
    p2.start()

    p1.join()
    p2.join()

    print ls
```

Figure (3.19) Attempting to pass a parameter to child processes in a way analogous to the sequential case, but not obtaining the same results. Uses `addItem` in Figure 3.18.

3.8.3 Comparing Parallel and Sequential Performance

The example in Figure 3.23 demonstrates and allows students to experiment with the speedup that having multiple processors allows. This example is set up to make use of two processors. Note the clear comments marking the section for timing the parallel code, and the section for timing the sequential code. Interprocess communication is avoided in this example; i.e., the example is embarassingly parallel. The spawned processes also do not merge their results into a final computed sum, and therefore one less addition is carried out.

This problem can make a good base for an exercise in or out of the classroom. Students can be instructed to experiment with the code and perhaps write a report with graphs showing performance across multiple variables:

- Adjusting totalNumNumbers
- Modifying difficulty of the operation to be performed (e.g., summing square roots of numbers)

In such an exercise, students should observe the following key points:

- When operations are longer and/or more complex, working in parallel has the potential to bring more significant speedups. (This is definitely true in embarassingly parallel

```
1  def addItem2(ls, q):
2    q.put(random.randint(1,100))
3
4  def parallelShareDS2():
5    ls = []
6
7    q = Queue()
8    p1 = Process(target = addItem2, args = (ls,q))
9    p2 = Process(target = addItem2, args = (ls,q))
10
11   p1.start()
12   p2.start()
13
14   # Now do whatever we need to with the results from the spawned processes
15   ls.append(q.get())
16   ls.append(q.get())
17   print ls
```

Figure (3.20) Demonstrates that communication between processes must occur via special structures, such as a queue.

problems like this summation.)

- Speedup of n processors over 1 on the same algorithm will typically not exceed n (barring potential effects of caching, for example).

The version in Figure 3.24 uses n processors, instead of a hardcoded 2. It requires the use of accumulation in lists to manage the child processes. In experimenting with this code, students should see that simply adding more processes, when there are no dedicated processors for them, doesn't speed up the computation. In fact, the extra overhead may slow things down further.

3.9 Further Examples Using the `Pool/map` Paradigm

The following examples all take advantage of the `Pool/map` mechanism first described in Section 3.6.3. The examples are each embarassingly parallel, allowing the student to consider the following abstract procedure for parallel problem solving:

1. Split up the work into chunks.

```
1  import random
2  import time
3
4  def timeSum():
5    numNumbers = 10000
6
7    startTime = time.time()
8    s = 0
9    for i in range(numNumbers):
10     s = s + random.randint(1, 100)
11     # randint(x, y) gives a random integer between x and y, inclusive
12   stopTime = time.time()
13
14   elapsedTime = stopTime - startTime
15   print "The summation took", elapsedTime, "seconds."
16
17 # This version just shows a small shortcut in the timing code.
18 def timeSum2():
19   numNumbers = 10000
20
21   startTime = time.time()
22   s = 0
23   for i in range(numNumbers):
24     s = s + random.randint(1, 100)
25   elapsedTime = time.time() - startTime
26
27   print "The summation took", elapsedTime, "seconds."
```

Figure (3.21) Shows the use of the time module for simple timing of algorithms

2. Process the chunks independently in parallel using the parallel `map` mechanism.

3. Synthesize the overall solution using the results of the independently processed chunks.

This procedure is applicable to a many interesting problems, from computer graphics and image processing to brute-force security attacks, to distributed computing projects like Folding@home and computational science more generally.

3.9.1 Monte Carlo Estimation of π

A simple–if slowly converging–method for computing π is to generate N random pairs in the interval $[0, 1]$. When each pair is considered a point in the unit square, the fraction of generated points whose distance from the origin is less than or equal to 1 will approach $\pi/4$ as N grows large. Further, generating N points can be seen as the same random estimation

```
1  def addEmUp(maxNum):
2    s = 0
3    for i in range(maxNum):
4      s = s + i
5
6  def joinEx():
7    # Compare execution times of the two processes below
8    p1 = Process(target=addEmUp, args=(50,))
9    # p1 = Process(target=addEmUp, args=(5000000,))
10
11   startTime = time.time()
12   p1.start()
13   print "Still working..."
14   p1.join()
15   elapsedTime = time.time() - startTime
16
17   print "Done!"
18
19   print "The process took", elapsedTime, "seconds."
```

Figure (3.22) Shows the use of join in timing processes

as independently generating P sets of N/P points and combining the results. The program in Figure 3.25 shows such a sequential versus parallel timing. On an example quad-core machine, the typical output might be:

```
$ python parallelMontePi.py 10000000
Sequential: With 10000000 iterations and in 4.528216 seconds,
            pi is approximated to be 3.141044.
Parallel: With 10000000 iterations and in 1.106317 seconds,
          pi is approximated to be 3.141217.
```

To explain why the parallel process could run more than four times faster on a quad-core machine, recall that the time computed is 'real' or 'wall' time rather than time consumed only by the process. This example demonstrates the simple steps of the abstract procedure defined above, while also showing the `Pool/map` mechanism simply, where the spawned process accepts and returns a single integer argument.

3.9.2 Integration by Riemann Sum

A more complicated demonstration of the `Pool/map` mechanism is shown in the program of Figure 3.26, which computes an integral via Riemann sum. Here the sequential tt integrate function accepts four arguments, from the function to integrate to the limits and stepsize of that integration. However if we wish to call upon this function to integrate subdomains in parallel, recall that the `map` method requires a function that accepts only a single argument. This is accomplished with the `wrapIntegrate` function, where a single tuple containing the `integrate` arguments is unpacked and sent on (argument unpacking in the method header is deprecated). Typical output might be:

```
$ python integratePool.py
sequential integral ans, time: 0.459698, 3.232372 sec.
parallel integral ans, time: 0.459698, 0.665953 sec.
```

3.9.3 Mergesort

The code of Figure 3.27 demonstrates a variant on mergesort using a `Pool` of processes. Motivation follows from the intuitive mergesort algorithm of sorting two half-lists and merging the result. Why only split the list into two parts, since each sublist sort is independent, and each pair of sorted sublists can be merged independently as well? The algorithmic variant demonstrated here is first to sort P independent sublists in parallel (line 33 in Figure 3.27), then to merge pairs of sorted sublists *a*lso in parallel (line 35) until there is only one list left. This 'flattened' mergesort is in contrast to the recursive instantiation used in Section 3.7.6. Example output, again on a quad-core machine:

```
$ python mergesortPool.py
Sequential mergesort: 6.134139 sec
Parallel mergesort: 2.011089 sec
```

3.10 Conclusion

We hope with this chapter to have provided some guidance and a plethora of examples to consider for introducing parallel computing concepts with Python in early Computer Science curricula. Using the demonstration programs provided, instructors can expose students to such critical considerations in parallel computing as shared resources, divide and conquer, communication and speedup. The programs also demonstrate the basic syntax for process spawning using `Process` or `Pool`, allowing students and instructor to re-purpose the code to their own ends.

```
1  from multiprocessing import *
2
3  def addNumbers(numNumbers):
4      s = 0
5      for i in range(numNumbers):
6          s = s + random.randint(1, 100)
7      print s
8
9  def compareParSeq():
10     totalNumNumbers = 1000000
11
12     # START TIMING PARALLEL
13     startTime = time.time()
14     p1 = Process(target=addNumbers, args=(totalNumNumbers/2,))
15     p2 = Process(target=addNumbers, args=(totalNumNumbers/2,))
16     p1.start()
17     p2.start()
18
19     # Wait until processes are done
20     p1.join()
21     p2.join()
22
23     parTime = time.time() - startTime
24     # DONE TIMING PARALLEL
25     print "It took", parTime, "seconds to compute in parallel."
26
27     # START TIMING SEQUENTIAL
28     startTime = time.time()
29
30     s = 0
31     for i in range(totalNumNumbers):
32         s = s + random.randint(1, 100)
33
34     seqTime = time.time() - startTime
35     # DONE TIMING SEQUENTIAL
36     print "It took", seqTime, "seconds to compute sequentially."
37
38     print "Speedup: ", seqTime / parTime
```

Figure (3.23) Comparing parallel and sequential performance on 2 processors

```
1  def compareParSeqN():
2    totalNumNumbers = 1000000
3    numProcesses = 5
4
5    # START TIMING PARALLEL
6    startTime = time.time()
7    pList = []
8    for i in range(numProcesses):
9      pList.append(Process(target = addNumbers, args =
           (totalNumNumbers/numProcesses,)))
10     pList[-1].start()
11
12   # Wait until processes are done
13   for i in range(numProcesses):
14     pList[i].join()
15
16   parTime = time.time() - startTime
17   # DONE TIMING PARALLEL
18   print "It took", parTime, "seconds to compute in parallel."
19
20   # START TIMING SEQUENTIAL
21   startTime = time.time()
22
23   s = 0
24   for i in range(totalNumNumbers):
25     s = s + random.randint(1, 100)
26
27   seqTime = time.time() - startTime
28   # DONE TIMING SEQUENTIAL
29   print "It took", seqTime, "seconds to compute sequentially."
30
31   print "Speedup: ", seqTime / parTime
```

Figure (3.24) Comparing parallel and sequential performance on n processors. This uses `addNumbers` defined in Figure 3.23.

```
#Monte Carlo estimation of pi.
from multiprocessing import Pool, cpu_count
import random, time, sys

def main():
    N = 1000000 #default unless provided on command line.
    if len(sys.argv) > 1:
        N = int(sys.argv[1])

    #sequential timing
    start = time.time()
    result = montePi(N)
    pi_seq = 4.0*result/N
    elapsed = time.time() - start

    print("Sequential: With %d iterations and in %f seconds," % (N, elapsed))
    print("            pi is approximated to be %f." % (pi_seq))


    time.sleep(3) #To see the parallel effect on OS' process manager.

    #parallel timing
    start = time.time()

    #Step 1: split up the work: how many iterations to run each montePi.
    cpus = cpu_count()
    args = [N // cpus] * cpus
    #Distribute extra work if work cannot be evenly distributed.
    for i in range(N % cpus):
        args[i] += 1

    #Instantiate the pool of processes
    pool = Pool(processes = cpus)

    #Step 2: process chunks independently. Here, compute subtotals.
    subtotals = pool.map(montePi, args)

    #Step 3: Synthesize the overall solution.
    result = sum(subtotals)
    pi_par = 4.0*result/N
    elapsed = time.time() - start

    print("Parallel: With %d iterations and in %f seconds," % (N, elapsed))
    print("          pi is approximated to be %f." % (pi_par))

def montePi(num):  #returns the number of pairs in [0,1] lie within unit disk.
    numInCircle = 0
    for i in range(num):
        x, y = random.random(), random.random()
        if x**2 + y**2 <= 1:
            numInCircle += 1
    return numInCircle
```

Figure (3.25) Monte Carlo Estimation of π.

```
#Parallel integration by Riemann Sum.
from multiprocessing import Pool, cpu_count
from math import sin
import time, sys

def main():
    n = 10000000

    #First, sequential timing:
    start = time.time()
    ans = integrate(sin, 0,1,1.0/n)
    elapsed = time.time() - start
    print("sequential integral ans, time: %f, %f sec." % (ans, elapsed))

    #Now, the parallel solver
    start = time.time()
    stpsize = 1.0/n
    cpus = cpu_count()

    #Step 1: split up the work. Domain split into equal parts.
    args = []
    endpoints = linspace(0,1,cpus+1)
    for i in range(cpus):
        args.append((sin, endpoints[i], endpoints[i+1], stpsize))

    #Step 2: process chunks independently. Here, sub-integrals.
    pool = Pool(processes = cpus)
    results = pool.map(wrapIntegrate, args)

    #Step 3: Synthesize the overall solution. The integral of the domain
    #is the sum of the integrals over the non-overlapping partial domains.
    ans = sum(results)
    elapsed = time.time() - start
    print("parallel integral ans, time: %f, %f sec." % (ans, elapsed))

def integrate(f, a,b,h):
    """computes the integral of f from a to b in n steps"""
    s = 0
    x = a + h/2.0
    while x < b:
        s += h*f(x)
        x += h
    return s

def wrapIntegrate(fabh):
    (f, a,b,h) = fabh
    return integrate(f, a,b,h)

def linspace(a,b,nsteps):
    #returns linear steps from a to b in nsteps.
    ssize = float(b-a)/(nsteps-1)
    return [a + i*ssize for i in range(nsteps)]
    #list comprehension, equivalent to map(lambda i:a+i*ssize, range(nsteps))
```

Figure (3.26) Sequential versus parallel integration of sin over $[0, 1]$.

```
from multiprocessing import Pool
import time, random, sys
from seqMergesort import merge, mergesort

def main():
    N = 500000
    lystbck = [random.random() for x in range(N)]

    #Sequential mergesort a copy of the list.
    lyst = list(lystbck)
    start = time.time()              #start time
    lyst = mergesort(lyst)
    elapsed = time.time() - start    #stop time
    print('Sequential mergesort: %f sec' % (elapsed))

    #Now, parallel mergesort.
    lyst = list(lystbck)
    start = time.time()
    lyst = mergeSortParallel(lyst)
    elapsed = time.time() - start
    print('Parallel mergesort: %f sec' % (elapsed))

def mergeWrap(AandB):
    a,b = AandB
    return merge(a,b)

def mergeSortParallel(lyst, n = 3):
    numproc = 2**n #default 8 sublists to sort.
    endpoints = [int(x) for x in linspace(0, len(lyst), numproc+1)]
    args = [lyst[endpoints[i]:endpoints[i+1]] for i in range(numproc)]

    pool = Pool(processes = numproc)
    sortedsublists = pool.map(mergesort, args)

    while len(sortedsublists) > 1:
        #get sorted sublist pairs to send to merge
        args = [(sortedsublists[i], sortedsublists[i+1]) \
        for i in range(0, len(sortedsublists), 2)]
        sortedsublists = pool.map(mergeWrap, args)
    return sortedsublists[0]

def linspace(a,b,nsteps):
    """
    returns list of simple linear steps from a to b in nsteps.
    """
    ssize = float(b-a)/(nsteps-1)
    return [a + i*ssize forig:pool-merges i in range(nsteps)]
```

Figure (3.27) Parallel mergesort using `Pool/map`. The code relies on sequential mergesort from Figure 3.17.

```
1  from multiprocessing import Process, Queue
2  import random
3  from seqMergesort import merge, mergesort
4
5  def main():
6      N = 500000
7      lyst = [random.random() for x in range(N)]
8      n = 3 #2**(n+1) - 1 processes will be instantiated.
9
10     #Instantiate a Process and send it the entire list,
11     #along with a Queue so that we can receive its response.
12     q = Queue()
13     p = Process(target=mergeSortParallel, \
14                 args=(lyst, q, n))
15     p.start()
16     #get blocks until there is something (the sorted list) to receive.
17     lyst = q.get()
18     p.join()
19
20 def mergeSortParallel(lyst, q, procNum):
21     #Base case, this process is a leaf or the problem is very small.
22     if procNum <= 0 or len(lyst) <= 1:
23         q.put(mergesort(lyst))
24         q.close()
25         return
26
27     ind = len(lyst)//2
28
29     #Create processes to sort the left and right halves of lyst,
30     #with queues to communicate the sorted sublists back to us.
31     qLeft = Queue()
32     leftProc = Process(target=mergeSortParallel, \
33                        args=(lyst[:ind], qLeft, procNum - 1))
34     qRight = Queue()
35     rightProc = Process(target=mergeSortParallel, \
36                        args=(lyst[ind:], qRight, procNum - 1))
37     leftProc.start()
38     rightProc.start()
39
40     #Receive the left and right sorted sublists (each get blocks, waiting to
41     #finish), then merge the two sorted sublists, then return through our q.
42     q.put(merge(qLeft.get(), qRight.get()))
43     q.close() #ensure no more data.
44
45     #Join the left and right processes to finsh.
46     leftProc.join()
47     rightProc.join()
```

Figure (3.28) Parallel mergesort using `Process` and `Pipe`. Uses the sequential mergesort code of Figure 3.17.

REFERENCES

[1] S. Bogaerts and J. Stough. (2013) Strategies for introducing parallelism with python. [Online]. Available: sc13.cs.wlu.edu

[2] (2013) Parallel processing and multiprocessing in python. [Online]. Available: https://wiki.python.org/moin/ParallelProcessing

[3] (2014) Spyder: the scientific python development environment. [Online]. Available: https://bitbucket.org/spyder-ide/spyderlib

[4] (2014) Python.org:integrateddevelopmentenvironments. [Online]. Available: https://wiki.python.org/moin/IntegratedDevelopmentEnvironments

[5] B. R. Maxim, G. Bachelis, D. James, and Q. Stout, "Introducing parallel algorithms in undergraduate computer science courses (tutorial session)," in *ACM SIGCSE Bulletin*, vol. 22, no. 1. ACM, 1990, p. 255.

[6] S. Bogaerts, K. Burke, B. Shelburne, and E. Stahlberg, "Concurrency and parallelism as a medium for computer science concepts," in *Curricula for Concurrency and Parallelism workshop at Systems, Programming, Languages, and Applications: Software for Humanity (SPLASH)*, Reno, NV, USA, October 2010.