Module 14: Concurrent and Parallel Programming in Python

March 31, 2021

Last time we looked into building graphical user interfaces (GUIs) and executables with Python, which can be very useful for making functionality available for third parties when for some reason sharing code as .py files or Jupyter Notebooks is not possible or not desirable.

Today we will discuss concurrent and parallel programming in Python, to cover the last control-flow structure we introduced in the beginning of the course.

Next week we will conclude the program with some guest lectures and tips and suggestions for programming and software development after this course.

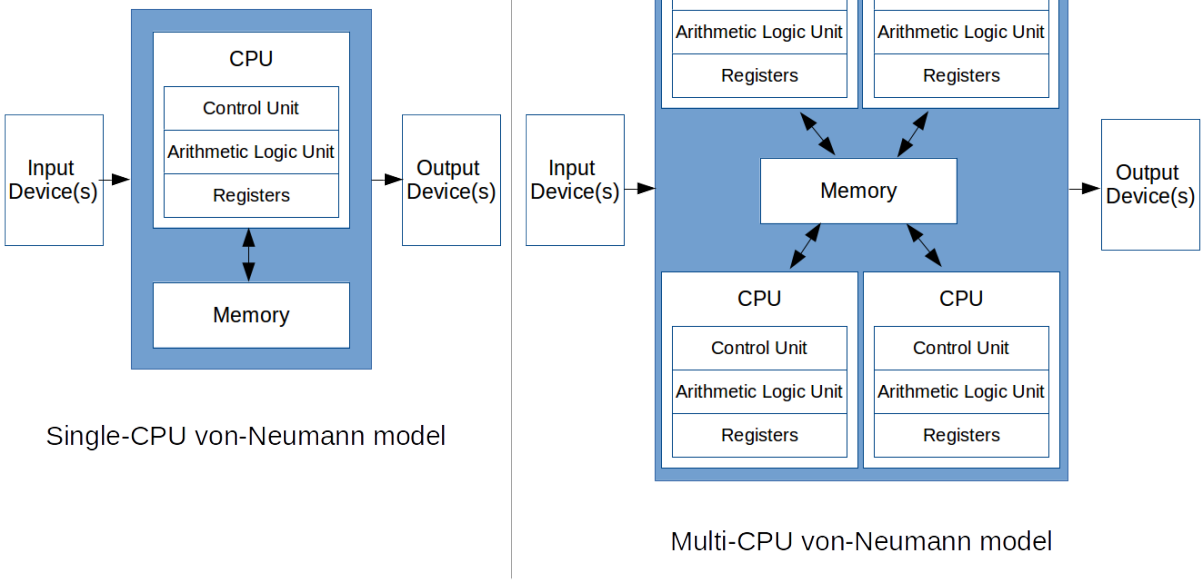
Background: Von Neumann Model

The von Neumann model (also known as von Neumann architecture) is an early description of a design architecture for computers. John von Neumann and others proposed it in 1945, and the principles still hold for today's computers.

The model describes a computer to consist of input devices (think of keyboard, mouse, scanner, microphone, camera, ...), output devices (screens, printers, sounds, ...), random-access memory (RAM) for storing data and instructions (including programs), and a central processing unit (CPU) comprising of control unit (for managing instructions), an arithmetic logic unit (for carrying out operations) and registers (CPU-internal memory). The CPU is ultimately reponsible for carrying out all operations needed for running a computer. This includes the operating system with all its processes, the installed software that is being used, as well as own programs that are executed.

CPUs are extremely fast. For example, the Intel Core i7 CPU in my laptop is said to have a speed of 4.9 GHz, roughly meaning that it can carry out up to $4.9 * 10^9$ instructions per second. Note that this refers to machine-level instructions, which are of finer granularity than the instructions we have in Python. A single line of Python code typically translates into several lines of machine code.

In the original formulation the model assumed a single-CPU system, but as the figure below illustrates, it easily extends to multiple CPUs.



How many CPUs does your computer have available? This piece of Python code will tell you:

In [20]:
```
import multiprocessing
print(f"This system has {multiprocessing.cpu_count()} CPUs.")
```

```
This system has 4 CPUs.
```

Concurrency vs. Parallelism

The terms *concurrency* and *parallelism* are often used interchangeably, but it is useful to distinguish them carefully, as the difference can have important implications for programming concurrent and parallel behaviour. Note that not all programmers use the same definitions, so you might come across different meanings. We use the definitions as in https://realpython.com/python-concurrency/ (https://realpython.com/python-concurrency/).

Concurrency is the ability of different (parts of) programs to be executed in arbitray order and thereby producing the same final outcome. This sounds abstract, but we use concurrent programs all the time. Concurrency makes programs more usable. For example, even if our computer, laptop or phone has only one processor, we usually run several programs at the same time. The browser has several tabs open for the latest news, the e-mail app is regularly checking for new messages, and in between we use different messenger apps for communicating with friends. Thereby it does not matter for the final outcome if we read the news first (or in which order), read our e-mail after we finished that, and only then go to our messenger app, or if we do all these things interleavingly with frequent switching between the programs. The processes are concurrent. Also within a single program there can be concurrency. For example, in the

messenger app we have different chats to which we can attend in arbitary order, or we can interrupt writing a message by setting a new status or profile picture, or just do it aftwards, the outcome would be the same. If you think about it a bit, you will discover many more concurrent processes in the software you use on a day-to-day basis. Many of them are visible at the user interface level, but there are also many more "under the hood".

Concurrent (parts of) programs can, but don't have to, be executed in parallel. If there is only one computing resource (processor) available, the concurrent processes will take turns in using it for taking the next steps. If there is more than one available, they can also (in parts) be distributed over the different resources and run at the same time, in parallel. In such multi-processor or multi-core systems, parallel execution can significantly improve the overall speed of exeuction.

Thus, *parallelism* in programming refers to techniques to make programs faster by perfoming multiple computations at the same time, exploiting the capabilites of a multi-processor or multi-core system. Typically, but not necessarily, this is used for problems that have can be divided into sub-computations of the same structure. For example, applying the same complex computation to all elements of a list can be done in parallel. Also graphic computations on GPUs are parallelism (and in fact GPUs are also used for massively parallelizing non-graphics computations in various areas, as they can be extremely fast). A key problem in parallelism is to identify sub-computations that have no or minimal data depencies between each other, so that they can be performed independeltly or with minimal communication between them.

Basic Python does not support concurrent or parallel programming, but there are various libraries available that make it possible (see, e.g. https://wiki.python.org/moin/ParallelProcessing (https://wiki.python.org/moin/ParallelProcessing)). We will have a look at two options from the standard library: thread-based parallelism with `threading` and process-based parallelism with `multiprocessing`.

Two (Serial) Example Programs

We will use the following two programs as examples in this module. Below their serial (sequential) variant, in the next sections we will discuss how to make them concurrent and parallel with Python.

The Coffeehouse

The idea here is to mimick what happens in a coffehouse. The processes that happen at each table are actually very similar: guests arrive, study the menu, order drinks and food, receive and consume them, pay and leave. This process is represented by a function that does simple printouts and pauses a few seconds between them (using seconds for what could be minutes in real life). The main program here then starts the process for three tables (it's a small coffeehouse). In addition, it measures the time needed for completing all three processes. This will be useful later to compare the execution time with concurrent and parallel implementations of the coffeehouse.

In [23]:
```
import time

# function mimicking the standard process at a coffeehouse table
def table(number):
    print(f"Table {number}: Guests arrive.")
    time.sleep(2)
    print(f"Table {number}: Guests study menu.")
    time.sleep(3)
    print(f"Table {number}: Guests order drinks.")
    time.sleep(5)
    print(f"Table {number}: Guests receive drinks.")
    time.sleep(1)
    print(f"Table {number}: Guests order food.")
    time.sleep(10)
    print(f"Table {number}: Guests receive food.")
    time.sleep(1)
    print(f"Table {number}: Guests consume.")
    time.sleep(15)
    print(f"Table {number}: Guests pay.")
    time.sleep(1)
    print(f"Table {number}: Guests leave.")


# main program
starttime = time.time()
table(1)
time.sleep(random.randint(0,5))
table(2)
time.sleep(random.randint(0,5))
table(3)
endtime = time.time()
print(f"Total time elapsed: {endtime-starttime} seconds.")
```

```
Table 1: Guests arrive.
Table 1: Guests study menu.
Table 1: Guests order drinks.
Table 1: Guests receive drinks.
Table 1: Guests order food.
Table 1: Guests receive food.
Table 1: Guests consume.
Table 1: Guests pay.
Table 1: Guests leave.
Table 2: Guests arrive.
Table 2: Guests study menu.
Table 2: Guests order drinks.
Table 2: Guests receive drinks.
Table 2: Guests order food.
Table 2: Guests receive food.
Table 2: Guests consume.
Table 2: Guests pay.
Table 2: Guests leave.
Table 3: Guests arrive.
Table 3: Guests study menu.
Table 3: Guests order drinks.
Table 3: Guests receive drinks.
Table 3: Guests order food.
```

```
Table 3: Guests receive food.
Table 3: Guests consume.
Table 3: Guests pay.
Table 3: Guests leave.
Total time elapsed: 120.09408473968506 seconds.
```

Text Analyses

The coffehouse example is quite instructive for a start, but in order to observe real speedup through parallelization we need something more computation-heavy. The program below defines three different text analysis functions (finding the longest word, the most frequent word, and all palindromes in the text). The main program applies these functions to a pretty long text, making sure that it needs some time to execute. As above, it also measures the execution time as a referene for later comparisons.

In [21]:

```
import time
import re

# helper function for retrieving all words (uninterrupted sequences of w
def get_all_words(text):
    all_words = re.findall("\w+", text)
    return all_words

# function to find the longest word in a text
def get_longest_word(text):
    words = get_all_words(text)
    longest_word = ""
    for word in words:
        if len(word) > len(longest_word):
            longest_word = word
    print(f"The longest word in the text is: {longest_word}")

# function to find the most frequent word in a text
def get_most_frequent_word(text):
    words = get_all_words(text)
    word_frequencies = {}
    for word in words:
        if word in word_frequencies:
            word_frequencies[word] += 1
        else:
            word_frequencies[word] = 1
    max_frequency = 0
    most_frequent_word = ""
    for word in words:
        if word_frequencies[word] > max_frequency:
            max_frequency = word_frequencies[word]
            most_frequent_word = word
    print(f"The most frequent word in the text is: {most_frequent_word}"

# function to find all palindromes (length > 1) in a text
def get_palindromes(text):
    words = get_all_words(text)
    all_palindromes = set()
    for word in words:
        if len(word) > 1 and word == word[::-1]:
            all_palindromes.add(word)
    print(f"There are {len(all_palindromes)} palindromes in the text.")
    #for palindrome in all_palindromes:
    #    print("\t", palindrome)


# main program
textfile = "data/big.txt" # downloaded from https://norvig.com/big.txt

with open(textfile, "r") as file:
    text = file.read()

starttime = time.time()

get_longest_word(text)
get_most_frequent_word(text)
```

```
get_palindromes(text)

endtime = time.time()
print(f"Total time elapsed: {endtime-starttime}")
```

```
The longest word in the text is: disproportionately
The most frequent word in the text is: the
There are 133 palindromes in the text.
Total time elapsed: 1.4100768566131592
```

Concurrent Programming in Python with `threading`

The `threading` library makes it possible to start concurrent processes from within a Python program. These processes are then called *threads*. Even if more than one CPU is available, all threads run on one processing unit. Technically, the Global Interpreter Lock (GIL) is responsible for this. GIL is a mechanism in Python that allows the intepreter to run only one instruction at a time. While it has certain advantages (like e.g. a shared memory that simplifies inter-process communication) it does not help much to make programs faster. (We will see examples of that later.)

As the variation of the Coffeehouse example below illustrates, we can use `threading` to simply create a set of Threads, launch and join them. As the output shows, the order of activities is more realistic, and the overall process is faster (because threads can `sleep` without requiring attention).

In [22]:
```
import time
import threading
import random

# function mimicking the standard process at a coffeehouse table
def table(number):
    print(f"Table {number}: Guests arrive.")
    time.sleep(2)
    print(f"Table {number}: Guests study menu.")
    time.sleep(3)
    print(f"Table {number}: Guests order drinks.")
    time.sleep(5)
    print(f"Table {number}: Guests receive drinks.")
    time.sleep(1)
    print(f"Table {number}: Guests order food.")
    time.sleep(10)
    print(f"Table {number}: Guests receive food.")
    time.sleep(1)
    print(f"Table {number}: Guests consume.")
    time.sleep(15)
    print(f"Table {number}: Guests pay.")
    time.sleep(1)
    print(f"Table {number}: Guests leave.")


# main program
starttime = time.time()

table1 = threading.Thread(target=table, args=(1,))
table2 = threading.Thread(target=table, args=(2,))
table3 = threading.Thread(target=table, args=(3,))

table1.start()
time.sleep(random.randint(0,5))
table2.start()
time.sleep(random.randint(0,5))
table3.start()

table1.join()
table2.join()
table3.join()

endtime = time.time()
print(f"Total time elapsed: {endtime-starttime}")
```

```
Table 1: Guests arrive.
Table 1: Guests study menu.
Table 2: Guests arrive.
Table 1: Guests order drinks.
Table 2: Guests study menu.
Table 3: Guests arrive.
Table 2: Guests order drinks.
Table 3: Guests study menu.
Table 1: Guests receive drinks.
Table 1: Guests order food.
Table 3: Guests order drinks.
```

```
Table 2: Guests receive drinks.
Table 2: Guests order food.
Table 3: Guests receive drinks.
Table 3: Guests order food.
Table 1: Guests receive food.
Table 1: Guests consume.
Table 2: Guests receive food.
Table 2: Guests consume.
Table 3: Guests receive food.
Table 3: Guests consume.
Table 1: Guests pay.
Table 1: Guests leave.
Table 2: Guests pay.
Table 2: Guests leave.
Table 3: Guests pay.
Table 3: Guests leave.
Total time elapsed: 44.046300649642944
```

In the same way, we can turn the text analysis example into a concurrent variant. Here the speedup is less pronounced, mainly because there are no `sleep` phases that would allow for time in a concurrent setting.

In [24]:

```
import time
import re
import threading

# helper function for retrieving all words (uninterrupted sequences of w
def get_all_words(text):
    all_words = re.findall("\w+", text)
    return all_words

# function to find the longest word in a text
def get_longest_word(text):
    words = get_all_words(text)
    longest_word = ""
    for word in words:
        if len(word) > len(longest_word):
            longest_word = word
    print(f"The longest word in the text is: {longest_word}")

# function to find the most frequent word in a text
def get_most_frequent_word(text):
    words = get_all_words(text)
    word_frequencies = {}
    for word in words:
        if word in word_frequencies:
            word_frequencies[word] += 1
        else:
            word_frequencies[word] = 1
    max_frequency = 0
    most_frequent_word = ""
    for word in words:
        if word_frequencies[word] > max_frequency:
            max_frequency = word_frequencies[word]
            most_frequent_word = word
    print(f"The most frequent word in the text is: {most_frequent_word}"

# function to find all palindromes (length > 1) in a text
def get_palindromes(text):
    words = get_all_words(text)
    all_palindromes = set()
    for word in words:
        if len(word) > 1 and word == word[::-1]:
            all_palindromes.add(word)
    print(f"There are {len(all_palindromes)} palindromes in the text.")
    #for palindrome in all_palindromes:
    #    print("\t", palindrome)


# main program
textfile = "data/big.txt" # downloaded from https://norvig.com/big.txt

with open(textfile, "r") as file:
    text = file.read()

starttime = time.time()

longest = threading.Thread(target=get_longest_word, args=(text,))
```

```
most_frequent = threading.Thread(target=get_most_frequent_word, args=(te
palindromes = threading.Thread(target=get_palindromes, args=(text,))

longest.start()
most_frequent.start()
palindromes.start()

longest.join()
most_frequent.join()
palindromes.join()

endtime = time.time()
print(f"Total time elapsed: {endtime-starttime}")
```

```
The longest word in the text is: disproportionately
There are 133 palindromes in the text.
The most frequent word in the text is: the
Total time elapsed: 1.3880736827850342
```

Parallel Programming in Python with `multiprocessing`

For real parallel programming, making use of different available CPUs, we can use the `multiprocessing` package. We can define, launch and join parallel processes very much like we defined, launched and joined concurrent threads above:

In [25]:

```
import time
import multiprocessing
import random

# function mimicking the standard process at a coffeehouse table
def table(number):
    print(f"Table {number}: Guests arrive.")
    time.sleep(2)
    print(f"Table {number}: Guests study menu.")
    time.sleep(3)
    print(f"Table {number}: Guests order drinks.")
    time.sleep(5)
    print(f"Table {number}: Guests receive drinks.")
    time.sleep(1)
    print(f"Table {number}: Guests order food.")
    time.sleep(10)
    print(f"Table {number}: Guests receive food.")
    time.sleep(1)
    print(f"Table {number}: Guests consume.")
    time.sleep(15)
    print(f"Table {number}: Guests pay.")
    time.sleep(1)
    print(f"Table {number}: Guests leave.")


# main program
starttime = time.time()

table1 = multiprocessing.Process(target=table, args=(1,))
table2 = multiprocessing.Process(target=table, args=(2,))
table3 = multiprocessing.Process(target=table, args=(3,))

table1.start()
time.sleep(random.randint(0,5))
table2.start()
time.sleep(random.randint(0,5))
table3.start()

table1.join()
table2.join()
table3.join()

endtime = time.time()
print(f"Total time elapsed: {endtime-starttime} seconds.")
```

```
Table 1: Guests arrive.
Table 2: Guests arrive.
Table 1: Guests study menu.
Table 2: Guests study menu.
Table 3: Guests arrive.
Table 3: Guests study menu.
Table 1: Guests order drinks.
Table 2: Guests order drinks.
Table 3: Guests order drinks.
Table 1: Guests receive drinks.Table 2: Guests receive drinks.
```

```
Table 1: Guests order food.
Table 2: Guests order food.
Table 3: Guests receive drinks.
Table 3: Guests order food.
Table 1: Guests receive food.
Table 2: Guests receive food.
Table 1: Guests consume.
Table 2: Guests consume.
Table 3: Guests receive food.
Table 3: Guests consume.
Table 1: Guests pay.
Table 2: Guests pay.
Table 1: Guests leave.
Table 2: Guests leave.
Table 3: Guests pay.
Table 3: Guests leave.
Total time elapsed: 41.112473011016846 seconds.
```

For this example there is not much additional speedup, which lies however again in the nature of the program (low-complexity computations, sleep phases). Another interesting observation here is that sometimes line breaks do not happen at exactly the foreseen places. That happens becaue on the machine-language level the printout of the line and the newline are different instructions, so that another, parallel process might carry out another printout instruction in between.

The text analysis program is however significantly faster when parallized, and the output also indicates which functions are faster than the others:

In [26]:

```
import time
import re
import multiprocessing

# helper function for retrieving all words (uninterrupted sequences of w
def get_all_words(text):
    all_words = re.findall("\w+", text)
    return all_words

# function to find the longest word in a text
def get_longest_word(text):
    words = get_all_words(text)
    longest_word = ""
    for word in words:
        if len(word) > len(longest_word):
            longest_word = word
    print(f"The longest word in the text is: {longest_word}")

# function to find the most frequent word in a text
def get_most_frequent_word(text):
    words = get_all_words(text)
    word_frequencies = {}
    for word in words:
        if word in word_frequencies:
            word_frequencies[word] += 1
        else:
            word_frequencies[word] = 1
    max_frequency = 0
    most_frequent_word = ""
    for word in words:
        if word_frequencies[word] > max_frequency:
            max_frequency = word_frequencies[word]
            most_frequent_word = word
    print(f"The most frequent word in the text is: {most_frequent_word}"

# function to find all palindromes (length > 1) in a text
def get_palindromes(text):
    words = get_all_words(text)
    all_palindromes = set()
    for word in words:
        if len(word) > 1 and word == word[::-1]:
            all_palindromes.add(word)
    print(f"There are {len(all_palindromes)} palindromes in the text.")
    #for palindrome in all_palindromes:
    #    print("\t", palindrome)


# main program
textfile = "data/big.txt" # downloaded from https://norvig.com/big.txt

with open(textfile, "r") as file:
    text = file.read()

starttime = time.time()

longest = multiprocessing.Process(target=get_longest_word, args=(text,))
```

```
most_frequent = multiprocessing.Process(target=get_most_frequent_word, a
palindromes = multiprocessing.Process(target=get_palindromes, args=(text

longest.start()
most_frequent.start()
palindromes.start()

longest.join()
most_frequent.join()
palindromes.join()

endtime = time.time()
print(f"Total time elapsed: {endtime-starttime}")
```

```
There are 133 palindromes in the text.
The longest word in the text is: disproportionately
The most frequent word in the text is: the
Total time elapsed: 0.850968599319458
```

Parallelization with Process Pools

In the previous examples, we worked with fixed numbers of threads and processes. However, in practice parallelization is used to apply the same function to many data items at the same time, such as (chunks of) a long list of input variables. For example, instead of only managing a small cafe with three tables, an actual coffee house might have tenths of tables (and the simulation program run the same process for all of them). Similarly, we might want to apply the individual functions of the text analysis program to a whole list of input texts, where parallelization could yield a significant speedup.

The easiest solution that Python offers here is probably the pooling feature of `multiprocessing` (note that something like this is not available in `threading`). It allows for parallelizing the execution of a function across multiple input values, by distributing the input data across parallel processes.

The typical pattern of pooling with `multiprocessing` is shown below for the Coffeehouse example. Application to the Text Analysis along the same lines.

In [27]:
```
import time
import multiprocessing

# function mimicking the standard process at a coffeehouse table
def table(number):
    print(f"Table {number}: Guests arrive.")
    #time.sleep(2)
    print(f"Table {number}: Guests study menu.")
    #time.sleep(3)
    print(f"Table {number}: Guests order drinks.")
    #time.sleep(5)
    print(f"Table {number}: Guests receive drinks.")
    #time.sleep(1)
    print(f"Table {number}: Guests order food.")
    #time.sleep(10)
    print(f"Table {number}: Guests receive food.")
    #time.sleep(1)
    print(f"Table {number}: Guests consume.")
    #time.sleep(15)
    print(f"Table {number}: Guests pay.")
    #time.sleep(1)
    print(f"Table {number}: Guests leave.")


# main program
tables = [x for x in range(1,21)]

cpus = multiprocessing.cpu_count()

with multiprocessing.Pool(processes=cpus) as pool:
    pool.map(table, tables)
```

```
Table 1: Guests arrive.
Table 3: Guests arrive.Table 1: Guests study menu.Table 7: Guests arr
ive.

Table 1: Guests order drinks.
Table 3: Guests study menu.
Table 7: Guests study menu.Table 1: Guests receive drinks.

Table 3: Guests order drinks.
Table 7: Guests order drinks.Table 3: Guests receive drinks.

Table 3: Guests order food.Table 7: Guests receive drinks.
Table 5: Guests arrive.Table 3: Guests receive food.

Table 3: Guests consume.
Table 7: Guests order food.Table 3: Guests pay.


Table 7: Guests receive food.Table 1: Guests order food.
Table 3: Guests leave
```

Again we can observe here the irregular distribution of newlines, for the same reasons as indicated above.

As another example with numerical computations, the following program parallelizes the celcius2fahrenheit conversion for a long number of (random) input values. The pooling pattern is as above, the only difference is that the return value of `pool.map()` is assigned to a variable. This converted list of values is printed afterwards.

In [19]:
```
import random

# function convert a celsius temperature to fahrenheit
def celsius2fahrenheit(temp):
    return (9/5) * temp + 32

# main program
temps_celsius = [random.randint(0,100) for x in range(0,25)]
print(temps_celsius)

cpus = multiprocessing.cpu_count()

with multiprocessing.Pool(processes=cpus) as pool:
    temps_fahrenheit = pool.map(celsius2fahrenheit, temps_celsius)

print(temps_fahrenheit)
```

```
[27, 54, 89, 87, 15, 57, 26, 23, 8, 37, 78, 16, 93, 9, 48, 39, 65, 43,
93, 74, 36, 49, 54, 74, 43]
[80.6, 129.2, 192.20000000000002, 188.6, 59.0, 134.60000000000002, 78.8
0000000000001, 73.4, 46.4, 98.60000000000001, 172.4, 60.8, 199.4, 48.2,
118.4, 102.2, 149.0, 109.4, 199.4, 165.20000000000002, 96.8, 120.2, 12
9.2, 165.20000000000002, 109.4]
```

More

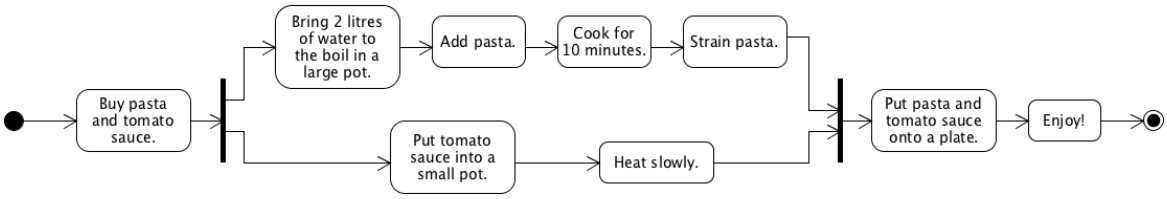
So far for a quick introduction to the basic principles of concurrent and parallel programming with Python. There are more things to consider when going for more complex applications and serious multi-core systems (high performance computing clusters, etc.). For example, exchange of objects and synchronization between processes becomes more important, but also more intricate. There are separate courses on concurrent and parallel programming, and also the web offers a lot of information.

If you'd like to challenge yourself a bit on this: On 12th and 13th April 2021, the Netherlands eScience Center will be running the next edition of their "Parallel Programming in Python" workshop (https://2y1rkjz.momice.events/page/851057).

Exercises

Please use Quarterfall to submit and check your answers.

1. Preparing Pasta with Tomato Sauce



This is the "how to prepare pasta with tomato sauce" Activity Diagram that we used in Module 2 to introduce the concept of parallel execution. Write a Python program that mimicks this process in a simple concurrent setting (assuming availability of only one cooking plate) and in a parallel setting (assuming availability of two cooking plates that can be used at the same time). Define two functions `prepare_pasta` and `prepare_sauce` for the concurrent parts of the overall process.

The output should look something like:

```
Concurrent version:
Buy pasta and tomato sauce.
Bring 2 lites of water to the boil in a large pot.
Add pasta.
Cook for 10 minutes.
Put tomato sauce into a small pot.
Heat slowly.
Strain pasta.
Put pasta and tomato sauce onto a plate.
Enjoy!
Total time elapsed: 10.013559341430664

Parallel version:
Buy pasta and tomato sauce.
Put tomato sauce into a small pot.
Heat slowly.
Bring 2 lites of water to the boil in a large pot.
Add pasta.
Cook for 10 minutes.
Strain pasta.
Put pasta and tomato sauce onto a plate.
Enjoy!
Total time elapsed: 10.08153748512268
```

Execute the program a few times. What differences can you observe? Can you explain them?

2. Many Fibonacci Numbers

In an earlier lecture we defined a (recursive) function for computing the Fibonacci number for in integer n:

```
def fib(n):
    if n > 1:
        return fib(n-1) + fib(n-2)
    elif n == 1:
        return 1
    else:
        return 0
```

Using this function, write a program that:

- generates a list of 20 random integer numbers (start with numbers between 1 and 10) as inputs
- prints the list of inputs
- implements two variants for computing the Fibonacci number for all inputs, storing the results in a new list:
  - serial: using a loop or list comprehension
  - parallel: using pooled parallel processes
- measures and prints out the execution time for both variants

The output should look something like:

```
Inputs: [5, 9, 4, 5, 2, 5, 2, 9, 6, 2, 10, 8, 7, 8, 8, 1, 9, 8,
 8, 2]

Starting serial computation.
Time elapsed: 0.0004374980926513672 seconds.
Result: [5, 34, 3, 5, 1, 5, 1, 34, 8, 1, 55, 21, 13, 21, 21, 1,
 34, 21, 21, 1]

Starting parallel computation.
Time elapsed: 0.050551414489746094 seconds.
Result: [5, 34, 3, 5, 1, 5, 1, 34, 8, 1, 55, 21, 13, 21, 21, 1,
 34, 21, 21, 1]
```

Probably you will see that the parallel version is slower than the serial one. Start playing around with the ranges of the input values (e.g., try n between 1 and 20, or n between 10 and 20, etc.). For which values does the parallel version start to be faster than the serial version? Can you explain why?

Extras

The "Dining Philosophers" are a classic and popular problem related to parallel and interdependent processes. We don't have time to discuss it in the lecture, but if you are curious, this website introduces the problem and possible solutions in various programming languages, including Python: https://rosettacode.org/wiki/Dining_philosophers (https://rosettacode.org/wiki/Dining_philosophers)

In [ ]: