**Lab 10 Parallel Computing in Python**

This practice consists of a mini tutorial about the Python multiprocessing package and few concrete examples, which might contribute to your coursework.

**Task 1: “Hello World” to process-based parallelism**

We’ll execute our first parallelised program, which is the foundation for more advanced features.

1. import multiprocessing as mp

2. def myfunc(message):

print(message)

3. if \_\_name\_\_ == '\_\_main\_\_':

4. print("Number of cpu : ", mp.cpu\_count())

5. messages=["Hello", "World", "Python", "Parallel"]

6. processes = [mp.Process(target=myfunc, args=(messages[x],)) for x in range(4)]

7. for p in processes:

8. p.start()

9. for p in processes:

10. p.join()

Line 1 imports the multiprocessing package, which is a build-in Python package. Line 2 is your function where you can do some operations even based on the parameter. Here, we simply print a message from the parameter. Line 3 is important if you want to run this program. Of course, we don’t need Line 3 if we’ll call this program from somewhere else.

Line 4 just prints out how many parallelised execution units in fact we can use. For example, Intel Core i5 8th Gen CPU has 4 cores, and each core has two independent threads (Hyper-threading Technology). Therefore, this CPU has 8 independent units. There is no number limit for software processes in programming languages, but all the processes are allocated to these hardware units finally. Bear in mind, parallel computing is partially based on hardware.

Line 5 creates a list consisting of four messages, which will be printed out by the different processes in parallel. Line 6 creates a list of processes. Here, we use the list comprehension, and you can also create a process individually and add them into the list. Usually, we pass a function that will do some work into the process with some parameters.

Line 7 and Line 8 launch these processes one by one using their built-in start function. Line 9 and Line 10 basically terminate all processes using the join function. Here, join() means ‘wait a process to terminate’. In fact, most parallelised programs are the master-worker pattern. For example, the current Python program is the master process, and those created processes are the worker processes. Then, the master process uses p.start() to launch a process (which leaves the master process), and uses p.join() to wait the process to join the master process when the worker process terminates. We use this mechanism to prevent the master process from terminating before the worker processes.

Please run the above program few times, and you may find out the different orders of those messages.

**Task 2: Parallelised Naïve search**

We use a simple linear search to demonstrate the parallelism. For example, we have a big text file, and unfortunately, one typo is inserted somewhere in the file, and your task is to find out where is the typo’s location. Clearly, a linear search can solve the task easily, and we just start from the beginning of the file and stop until the typo is encountered.

Please download the text file “The Deathly Hallows.txt” from BB and find the line NO of the typo word “KunWei”. It should be straightforward to finish the task, or you can take a look at the following example code.

def serialSearch(pattern, text, index):

found = False

for lineno in range(len(text)):

#str.find(pattern) returns the position of the pattern in the string and returns -1 if no match

if text[lineno].find('KunWei') != -1:

print("Found the keyword in line "+str(index + lineno+1))

found = True

break

if found is False:

print("The text didn't contain this keyword!")

where pattern is the word that we are searching such as ‘KunWei’, text is the text file which contains the pattern and index is the start position of the text for searching. Here, we check the text line by line, and so we simply need to know the line number of the typo.

We use a built-in function in a string, and str.find(pattern) will return a number to denote the position of the pattern in the line or -1 if dismatch.

Then, you can use the following code to test this function.

def main():

pattern = 'KunWei'

with open("The Deathly Hallows.txt", encoding='utf8') as file:

text = file.readlines()

serialSearch(pattern, text, 0)

file.close()

We won’t calculate the execution time here because the text file is relatively small, and there is no difference between the serial version and the parallel version.

For the parallelised program, we consider dividing the text into four portions, and create four processes to search one portion each. This is a typical example of input data decomposition and no communication among the processes. You can take the following code as an example to figure out how this is done.

def parallelisedSearch(pattern, text):

n = 4 # number of processes

steps = len(text)//n # size of each portion for the text file

# first three processes get the equal portion of the text

processes = [mp.Process(target=serialSearch, args=(pattern,text[i\*steps:(i+1)\*steps:],i\*steps)) for i in range(n-1)]

# add the last one which might have a different size of text

processes.append(mp.Process(target=serialSearch, args=(pattern,text[(n-1)\*steps::],(n-1)\*steps)))

for p in processes:

p.start()

for p in processes:

p.join()

In fact, we reuse the serialSearch(), but just give different values for the index parameter for different processes. Note that usually we treat the last process especially because of no guarantee for the equal division of the text file. Here, the last process will pick up the left part of the text. In fact, the parameter index will not be used for computation, but it will give the global line number when the pattern is found. For the worst case, the parallelised program can use 25% of the execution time of the serial program (WHY?).

**Task 3: Communication in processes via Pipe**

Very often processes need to communicate with each other to coordinate the work. Pipe is a module in the multiprocessing package, and it provides a synchronised communication channel between two processes. This mechanism is like a walkie talkie, and one process sends a message to the other and terminates only if the other has received the message. Please read and try the following example.

from multiprocessing import Process, Pipe

def childFunc(conn):

print(conn.recv())

conn.send(["Hello, parent!"])

conn.close()

def parentFunc(conn):

conn.send(['Hello, child!'])

print(conn.recv())

conn.close()

if \_\_name\_\_ == '\_\_main\_\_':

parent\_conn, child\_conn = Pipe()

parent = Process(target=parentFunc, args=(parent\_conn,))

child1 = Process(target=childFunc, args=(child\_conn,))

parent.start()

child1.start()

child1.join()

parent.join()

So, parent\_conn and child\_conn are two ends of the pipe. Then, we create one parent process and one child process which obtain the end individually. Now, they can do the synchronised talking.

The ends of the pipe can be obtained by multiple processes. For example, we create another child process which takes the child\_conn. However, the message from the parent process can reach ONE child process only. To make both child processes get the message, the parent process needs to send the message twice. Can you implement the new example yourself?

Note that you may experience a deadlock if the communication is not synchronised.

Now, we’ll redo Task 9.2 but with the communication via a pipe. Here, we modify the serialSearch() and parallelisedSearch() somewhat, and you can still use the same main function in Task 9.2. Please read and try the following code.

def serialSearch(pattern, text, conn):

message = conn.recv()

start = int(message[0])

end = int(message[1])

found = False

for lineno in range(start, end):

if text[lineno].find(pattern) != -1:

print("Found the keyword in line "+str(lineno+1))

found = True

break

if found is False:

print("The text didn't contain this keyword!")

def parallelisedSearch(pattern, text):

n = 4 # number of processes

steps = len(text)//n

master\_conn, worker\_conn = mp.Pipe()

# creat n processes

processes = [mp.Process(target=serialSearch, args=(pattern,text, worker\_conn)) for i in range(n)]

# start all processes

for p in processes:

p.start()

#send the boundaries of the text to all processes

for i in range(n-1):

master\_conn.send([i\*steps, (i+1)\*steps])

#send the last part of the text

master\_conn.send([(n-1)\*steps, len(text)])

**Task 4 Communication in processes via Queue**

We can also use Queue to communicate all processes. So, different from Pipe, Queue in multiprocessing is a shared memory space which can be got assess by all processes. For example, you use the parent and children greeting example again but with Queue. Please read and try the following code.

from multiprocessing import Process, Queue

def childFunc(conn):

print(conn.get())

if \_\_name\_\_ == '\_\_main\_\_':

q= Queue()

child1 = Process(target=childFunc, args=(q,))

child2 = Process(target=childFunc, args=(q,))

q.put("Hello, child1!")

q.put("Hello, child2!")

child1.start()

child2.start()

child1.join()

child2.join()

Note that we don’t know which process gets which message in fact because the order of the execution might be different from what you’ve seen here. Also, Queue.get() will terminate only if it can get a value, and will wait if the queue is empty.

Now, can you rewrite our parallelised searching via Queue?

Hint: you can let the main program to write the boundaries of the text into the queue and each process simply gets the value from the queue again.