Task 1

N.B: Our results shown in our attached notebook can only be replicated using a machine with 8 cores.

Quite unsurprisingly, and illustrating the rationale for implementing more than one core, we note that generally speaking increasing the number of cores reduces the time taken for our machines to perform calculations. What did strike us as surprising however, was the marginal increase in performance (i.e. reducing time) to which adding each additional core made. Initially, adding extra cores led to significant increases in performance. For instance, utilising two cores instead of one came close to halving runtime. However, diminishing marginal returns quickly set in and we often saw no increase in performance (sometimes adding an extra core even resulted in a longer computation). Ultimately, this leads us to the conclusion that, unless performing extremely complicated calculations that could be run on separate cores, additional resource allocation [the added cost (both financial and computational)] is not worth the slight uptick in performance.

It is worth framing the results we observed in the context of the type of calculations that were being performed. As it was largely a CPU bound program (it was simply conducting a large number of comparisons), as opposed to I/O bound, we would expect multiprocessing, instead of multithreading, to speed up the calculations. With a single core, Python cannot calculate any of the required comparisons concurrently, however splitting the workload between multiple cores enables quicker execution, unlike in a I/O dominated program where multithreading would lead to efficiencies while I/O operations were being performed and the CPU was idle.

Python Global Interpreter Lock (GIL)

By default, Python does not utilise more than one core at once. Interestingly, it is actually the GIL that both contributed to the rise and dominance of Python as a mainstream programming language, and means that only one thread can control the Python interpreter at any one time. The GIL is basically a lock on the interpreter that requires acquisition to conduct code execution. Around the time of Python's widespread adoption, the GIL was a simple solution to prevent inconsistencies with the integration of C libraries that are not thread safe. For many other reasons, such as maintaining backwards compatibility, maintaining performance for single threaded implementations and others, the GIL has remained apart of Python.

Although other solutions to thread-safe memory management (such as garbage collection using which multiple threads can be created) have been implemented in other languages, Python's approach allows for the performance benefits that accompany single threaded implementations, such as the avoidance of repeated acquisition and release of locks and the enablement of higher clock speeds.

Performance Improvements using Multithreading

Understanding the mechanism for the GIL is crucial to understanding why implementing multiprocessing as opposed to multithreading (which is still possible in Python) leads to performance increases. If instead we had initialised a single process with say 8 threads, each thread would have to wait on the one above it to execute before it could access the CPU. With few I/O operations there does not come a point whereby a thread can take over the GIL while another waits for I/O. A logical follow-on therefore would be that creating many instances of the interpreter which themselves have a single mutex may enable enhanced performance for CPU-bound processes.

As alluded to, with the requirement for the acquisition of the mutex, Python only allows one thread to be in execution at any one time. For CPU-bound programs such as ours this obviously leads to bottlenecks in performance, as although there is not a point in which the one thread holding the lock is not in execution, there is only one instance of the Python interpreter, meaning only one core in use.

The Multiprocessing library that we used navigates around the GIL issue by instantiating more than one operating system processes for the tasks to be carried out. In essence this gives each process a Python interpreter of its own, meaning a scenario whereby there are as many GILs as there are new processes. In this way, the new processes are executed separately and crucially, concurrently, in different cores and subsequently regrouped once all processes have executed.

The effects of this are striking, and explored in depth in our accompanying notebook. Proving how computationally intensive (and therefore CPU-bound) the program is, we see close to a 50% reduction in performance time when analysing the same set of primes using 2 cores instead of 1.

There was also an interesting observation in that the marginal uptick in execution for each additional core is far from linear and is more convex in nature. Indeed, for some of our tests we sometimes saw an increase in performance time, however we theorise this is due to external factors, such as other processes on our machines taking over a core (particularly as it does not happen all the time, nor does it happen in the average run time tests). Still, it is intuitive why the increases in performance are not distributed evenly for each extra core. Recalling what is actually occurring when we make a call for one more core, Python must create an extra process which is in itself computationally expensive and introduces I/O overhead. These overheads are what seem to be the reason for increasingly smaller amounts of performance optimisation through adding additional cores. As mentioned in the notebook, using 2 cores instead of 1 cut processing time by nearly 49%, using 4 cores reduced time by 73%, while 8 cores only reduced time by 79%, showing that the key to faster computation is not necessarily using more cores.

It would be quite interesting to conduct the same experiments on a machine with cores far in excess of the 8 we used, to see if the performance time continued to flat-line, or if we actually saw increased times due to the overheads created when spawning new processes. We also believe it would be interesting to explore combining multiprocessing with multithreading. Given that the creation of additional processes introduces process management (I/O) overhead, and it is largely the creation of these processes that cause the flatline in performance as the cores are increased, it may be the case that creating additional threads to handle this, and the additional processes to handle the CPU-bound work, leads to a more linear relationship.

Task 2.a

For the second task of this assignment, initially we considered downloading images, however, after further research we discovered that downloading is primarily an I/O bound task, since there is a wait time for the images to download. Therefore, to align with the assignment requirements we decided image processing was a more appropriate task as it is predominantly a CPU bound task.

We utilised the Pillow Python library to do some image processing (adding a Gaussian blur) on the pre-downloaded images. The `process_image` function simply opens the image, applies the filter and thumbnail size and then saves the image to the `images/processed` folder. A loop is utilised to loop over the range 1 – 8 which then records the length of time it takes the `pool_process` function to run with a given number of cores. The times are then appended to `processing_times` array which is used to plot the running times of each of the cores.

The results obtained in this task were similar to those observed in Part A. As can be seen in the accompanying notebook, it took 12.32 seconds to process the 15 images using only 1 core. Using 2 cores dropped this by over 47%, while 8 cores utilised led to a 75% decrease. The significant drops in time highlight how multiprocessing can greatly reduce the time necessary to perform CPU intensive tasks. As discussed in Task 1, it was also found that adding the first few additional cores greatly reduced processing times however, after a certain point diminishing marginal returns (caused by the increase in process management overhead) quickly set in leading to only minor reductions in processing times. Indeed it actually took longer to run on 8 cores than 7, likely due to external factors (interesting in of itself) and increased process management, but we saw in Task 1 that on average 8 cores should be faster.

Key takeaways:

The main learnings from these experiments we carried out are twofold: 1) utilising more cores, through in effect bypassing the restraints of the GIL by creating more Python interpreters (and therefore as many more GILs) leads to a reduction in overall processing time for CPU bound processes. 2) this reduction in overall processing time suffers from diminishing marginal returns, due to the increased process management (I/O) overhead of instantiating extra Python interpreters. Therefore, we speculate that it would be an interesting exploration to combine multiprocessing with multithreading, as multithreading can help to speed up I/O bound processes. This is something we plan to research ourselves outside the scope of this module.

A final additional learning that came as a by-product of this work, is that the same process ran twice on the same machine will not necessarily complete in the same time (or close to the same time), largely due to other processes demanding CPU time while the program being observed is running. Ultimately this shows the benefits of reducing running programs that demand CPU time when speed is required.