**Task 1:**

**N.B:** As is to be expected, our results as outlined in our 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 one more core made. Initially, adding extra cores ed to significant increases in performance. For example, 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, the added cost (both financial and \_\_\_\_\_) 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.

By default, python does not utilise more than one core at once

* Getting data to cores etc, more meaningful with more complex computations
* Talk about reason for marginal decrease due to overhead in creating multiprocessing processes – heavier than threads etc
* Could be interesting to see if time increases due to creation of extra processes when it is simplistic
* <https://www.linuxjournal.com/content/multiprocessing-python>
* https://www.quantstart.com/articles/Parallelising-Python-with-Threading-and-Multiprocessing/

LESSONS

**Task 2:**

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 follow the assignment requirements we decided image processing was a more appropriate task as it is mainly 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 `normal\_image\_process` and `multiprocessing\_image\_process` functions are identical except for where they save the processed images. It would have been preferable to merge this into one function which takes the folder route as a parameter, but the pool process function does not permit this as the function provided to it must only take one argument.

As can been seen in the accompanying notebook, it took 17 seconds to process the 15 images using only 1 core. When 4 cores were used the time took to process the images was 7 seconds. This increase in core utilisation lead to a 58.8% decrease in processing time. This is a significant drop in time and highlights how multiprocessing can greatly reduce the time necessary to perform CPU intensive tasks.