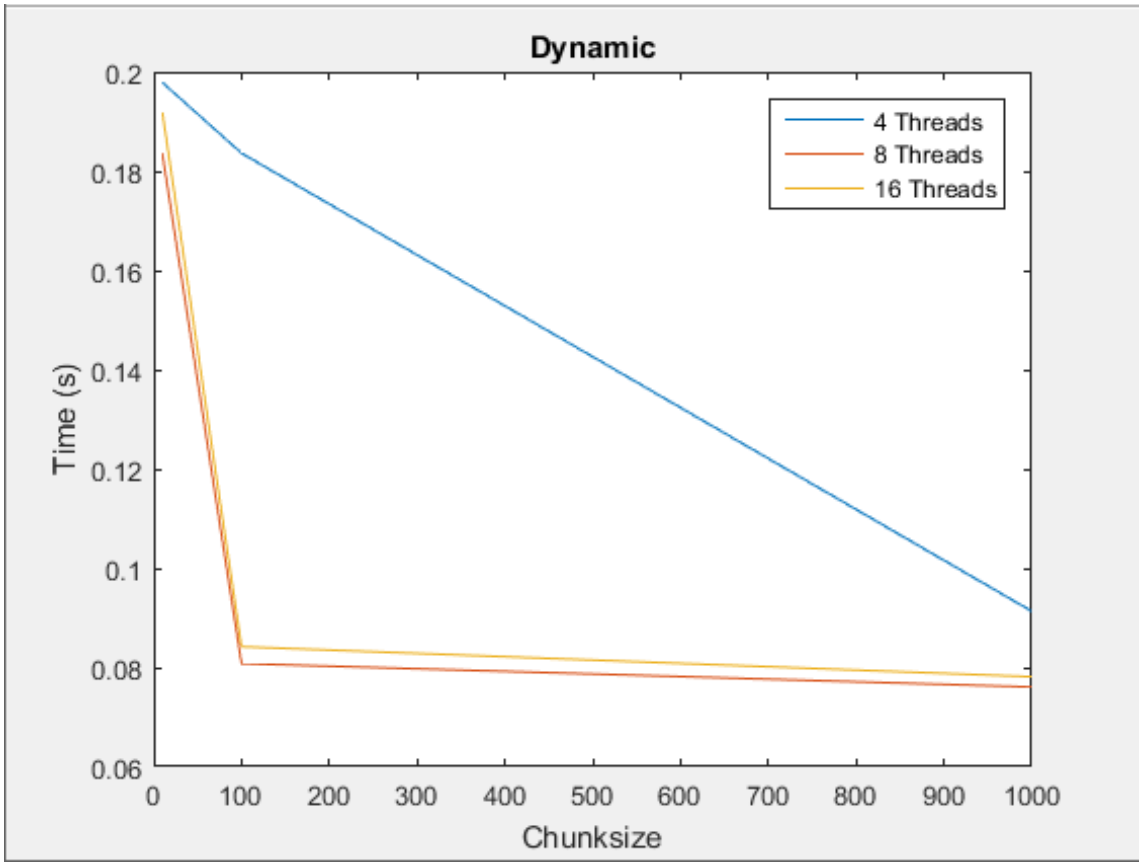
OpenMP Coursework

Our task was to download and run a piece of code and time how long it takes to run sequentially, we were then to parallelise the same program and repeat while changing certain parameters and note down the times. I had to do this for both dynamic and static methods and vary the chunksize and number of threads.

I conducted these experiments using Bash from the command line, this creates a linux environment, on my windows machine. My machine is running windows 10 on a 2.1GHz AMD processor with four cores.

For the experiments itself I first had to run the unaltered code and obtain a time for comparison, to ensure accuracy I decided to run this code 10 times and take an average, for both static and dynamic scheduling. I decided on using 4, 8 and 16 threads, and a chunksize of 10, 100 and 1000, and for each of these combinations the code was run 10 times and an average taken. My initial time for the experiment was 0.132315 seconds.

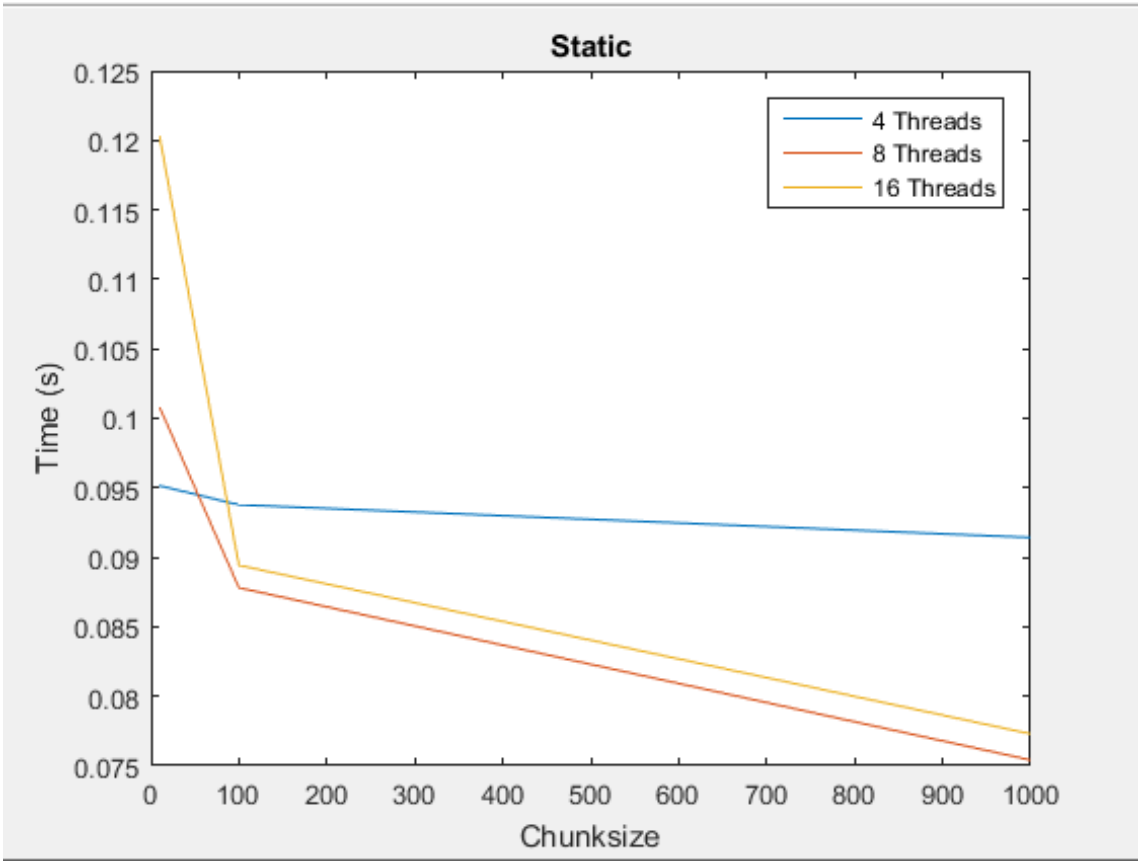


This first graph represents dynamic scheduling, each line corresponds to 4, 8 and 16 threads, the x axis is the chunksize and the y axis is the time taken. The time taken ranged from just under 0.2 seconds to around 0.07 seconds.

We can see that for a very small chunksize there is only a small discernible difference in runtime, we would expect to find that more threads would run faster but in this instance 8 threads is actually marginally faster than 16.

When we increase the chunksize to 100 4 threads has a small Improvement but both the 8 and 16 threads have a huge jump from between 0.18 – 0.2 seconds down to approximately 0.09 seconds, this is again concurrent with what we expect, a larger chunksize means that the code is split up into less parts and more threads can process these chunks leading to a faster overall time. Interestingly the line for 8 threads is still marginally faster than that of 16 threads.

Finally at a large chunk size of 1000 we see a rather large but linear improvement for 4 threads but only a small improvement for 8 and 16 threads, this could be due to the chunksize now being too large and the extra threads are not getting fully utilised. This could also explain why the 8 thread code is running slightly faster than the 16, all 16 threads may not be required for this size of task and is actually hindering the runtime slightly by trying to distribute it between too many resources.



The second graph is for static scheduling, the main difference between static and dynamic is that static scheduling is computing during compilation and dynamic is computed during run time. This explains our first major difference in that for a small chunksize 4 threads actually ran fastest of all as these resources were better utilised from the start. It should also be noted that the slowest static times were significantly quicker than dynamic but the fastest times are marginally slower.

Overall the 4 thread line did not show much improvement with increasing chunksize, only improving by around 0.005 seconds.

Initially the 16 threads was the slowest by quite a large margin, but when the chunksize is increased to 100, the 8 and the 16 threads both improve and give very similar times. When the chunksize is

increased further the 8 and 16 both continue to improve at a much slower rate and the final times are slower than that of the dynamic scheduling.

When starting this experiment I expected to find that more threads and larger chunksizes would give faster run times. However my results show me that this is not strictly true. Other than at a very small chunksize on the static graph both 8 and 16 threads were faster than 4 threads in every other case, but contrastingly to my expectation 8 threads was in fact faster than 16. Too many threads can cause bottlenecks to occur but the best number of threads can vary greatly by the size and nature of the task performed and the resources available. Generally as the chunksize increased the runtime was quicker in all cases, from 0 to 100 showed significant improvements while from 100 to 1000 showed much more gradual improvements and I believe that any further increase would cause the graph to level off. Finally from my graphs I have also included that for a small chunksize where resources are limited static scheduling produces better results, but as the chunksize and number of threads increases dynamic scheduling produced the fastest times.

If I were to run this experiment again I would take many more data points at much more frequent chunksize and thread number intervals, hopefully producing a smooth curve and highlighting any key areas that produce a particularly fast runtime.