Parallel Computing – Pthreads in C with Balena

This is a writeup on the relaxation method in parallel for CM30225 coursework 1 – I will keep it as short as possible! Note that the writeup can be considered a little informal as it focuses more on results than this being an official writeup. To prevent the writeup from becoming a collection of graphs and code snippets, anytime I reference code or a graph I will also reference the file (and line/page) where these can be found.

Relaxation method:

The objective of the assignment is averaging 4 neighbouring values in an array. Starting off with doing it sequentially to "get it in my head" as to how I'm going to do the primary algorithm behind it, **AverageNumbers.c** was created. A quick run-through of how it works:

Giving it array dimensions, it creates an array of random values, then just grabs the vals of one row above, below, one column left, right, then averages those 4 values. Computer Science students have done similar things in the Convolution Theorem on images, hence this wasn't an unfamiliar item. **AverageNumbers.c** is a debug piece of code, and has everything hardcoded in it, I could make it better, but I'd already moved on as it was just a "keepsake" of the algorithm, to have some building blocks on how it works. If you run the program it will create an outputSingle.csv with the output, easy.

Next up was unfamiliar fields: pthreads! Online I found a small snippet of code (see **AveragePthreads.c**) that taught the basic ropes of running two threads in parallel, nothing too fancy. For testing purposes I have attached genericMatrix.m and 5Matrix.txt, two files that can be used to test out avgArray, but this is covered later on (AverageNumber.c is a debug/test file that's mostly hardcoded).

Parallelisation strategy:

So as my parallelisation strategy I chose to simply distribute my array over x amount of threads – for example 5 threads, 100 rows, each thread gets 20 rows. This was done by sending off a thread with a pointer to the start of its delegated row, and allowing to to go through the amount of times necessary, which is rows * dimensionsize for every thread except the final one. If we had 105 rows, I simply let the last (for which I used the main) thread take the modulus (in this example, 25 threads) as well. It started the pointer in the same position, but went on for a few more points.

In all fairness, this approach is somewhat subject to race conditions – as one thread may be working on the row above the next one, it doesn't know whether or not that row has already been edited through. Hence the array may sometimes return *slightly* different outputs at large dimension values depending on the threads timing, as they are not held by semaphores or mutexes/barriers. As I had already sent off Balena jobs before noting this it is not covered here.

Another interesting issue to note is that I had originally planned to split my 2d array into multiple 2d arrays – without much use of pointers – and pass these on in structs towards new threads. Eventually I came to the realisation that I could flatten my array into a 1D

vector, and just use the arraydimensions to find out how to go up, down, left and right. This was easily incorporated and made multithreading a lot easier.

Correctness testing:

To test for correctness, I pushed a 5x5 array through at 0.01 precision and manually checked each step on whether the values worked out – I chose 2-3 values per iteration rather than doing the entire thing. Slow, but it's the only way to ensure my written code wasn't faulty! Another idea would be to use an identity matrix (or any given matrix) and comparing it to my peers their work, this was a very limited method for the method I'd opted for due to the fact that I had race conditions that would influence this output.

In terms of code "correctness" (aka how good it looks and how legible it is) testing I had some peers look at it on whether it looks good, especially attempting to get people to read it that haven't done the module and ensure they understand what exactly it does., and compiled with as many warnings as possible (Wall, Wextra, Wconversion). Although I assume this isn't what is meant with correctness testing, I thought it to be worth mentioning.

Graphs, scalability and using Balena:

This is the most interesting part! This covers primarily the output, any future use of this and why parallel is, to this day, such a headache for most programmers (including myself), as averaging 4 values, for a computer, is an incredibly simple and fast thing to do, but in parallel a bit more of an adventure. And essentially everything contained in **avgArray.c**, the main file of this coursework, other than the excel file with all my data.

So to ensure I had enough data I essentially created a large batch queue in Balena using a script, testing each node configuration around 120 times. Essentially Balena has the limitation given to us of 4 nodes at maximum:

For 1 node, I tested using 1 up to 16 threads (tasks per node), from array size 100 to 900. Once I went beyond 900 array size (except for 1 task/node), Balena booted my task out as it took longer than 15 minutes (the maximum given to us for this coursework). I attempted to do 100 to 1000 but an array of size 1000 only worked for 1 or 2 tasks per node, at certain node configurations, due to these time constraints.

The output of this batch of Balena runs is to be found in **ParallelAnalysis.xlsx**. This data was harvested out of Balena logs using MATLAB scripts – I originally had opted for Python but found MATLAB slightly easier to write in due to its lax writing and use of libraries. This is an opinion, not a fact! I also used small things in the code that make it easier, like the output csv printing a semi-colon after every row (except the last one) so it was easier to read in the files.

To expand on the graphs in the analysis: it is very quickly clear that at 1 task per node, with 1 node, it is often significantly faster due to the overhead caused by passing data between nodes, and increasing the tasks per node decreases the amount of time taken in a strange non-linear way (at 1 node it shoots up significantly from 1 tasks per node to 2, but then normalises at 3, very curious). For 2 & 3Nodes it however is a bowl-shaped graph, as it was

so slow at 1 task per node I couldn't even perform a 900x900 array without getting timed out. Starting at 4 Nodes I opted not to use 900x900 arrays anymore as half of them had failed and Balena cancelled them due to timeout, but they also seem to scale quite well.

Balena was an interesting experience – I have plenty of experience with computer and linux but not before with queuing up my work and similar! Luckily it allowed me to batch a bunch of batch files so I could leave it to perform the work overnight and all I had to do was strip the log files of the timing. (I say this as if it didn't took me several days to figure it all out).

Now, to the important part: scalability and speed. As is aid before, the relaxation operation is very quick, as it's just a load of averaging – but asking someone else to do it is definitely the slow part; it's easy to imagine 1 billion operations being split between two computers, but when scaled down, splitting 100 operations requires a lot of overhead that simply slows the computer down.

This reflects quite well on the analysis part, as I was limited to a constraint of 15 minutes per job, it wasn't easy to reach the "peak" where jobs start becoming faster due to their sheer size. At one node, with one task per node, it only took about 327 seconds (5 minutes) to run through a 900x900 array, but at 16 tasks per node this ramped up to more than twice (747). This data is found on the "1 Node" tab of **ParallelAnalysis.xlsx.**

This problem became less of a "gap" between tasks per node, on the other end of the extreme, an 800x800 array with 4 nodes working, but 1 thread/node takes 580 seconds, with 16 threads it goes up to 630 seconds, so the gap between threads per node becomes less and less apparent as your amount of nodes increases. But, the overall time has also slowed down drastically, as all this ran significantly faster with 1 Node.

Again, the issues is simply the amount of overhead – running threads or tasks on a single Node doesn't require all that much communications – between the memory and the CPU there is generally a fast bus. But inter-node communications is notoriously slow as it passes through a network, which even with Fibre is a lot slower than what a memory bus can achieve. This is really what shared memory programming boils down to – a massive amount of overhead due to the fact that nodes all need to know what the other has, rather than distributing it.

Although I haven't entirely worked out in my head how to distribute the memory properly for the next coursework on MPI, I have a few ideas and look forward to the second coursework as this was quite the learning exercise.