CAB401 PARALLELIZATION Assignment

By Alexander Santander – n9684026

QUT 2017

**The original sequential application:**

What is it?

The program that has been chosen is from the selection provided on blackboard. It has been implemented in C++ and ultimately calculates genome similarity using frequency vectors. The application itself is rather small with the source code itself only being 265 lines, however there are a large amount of resource files for it to read from.

How does it work?

As stated this program was designed to read data on various bacteria and then use vectors compare them and calculate their similarity.

Like any program this one begins in the “main” function. First the vectors which all the data will be stored in for comparison must be initialised. This is done through calling the “init()” function where the values of each are set. Note: AA\_NUMBER has a value of 20.

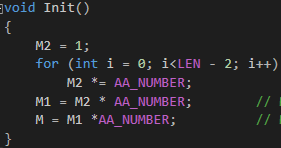


Figure - setting the vectors

From here the files with the information regarding the various bacteria must be read in and relevant data stored in variables. The function “ReadInputFile()” does exactly that by taking the file name as a parameter and storing the number of bacteria and their names in variables.

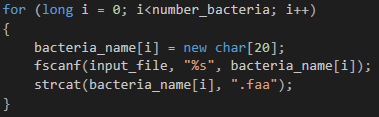


Figure - storing the number of bacteria

function “CompareAllBacteria()” is then called upon which initially creates a new list of class “Bacteria” with the length found from the input file. It is worth noting that the variable used to access this list is a pointer to a pointer. This extra level of abstraction will essentially make these new instantiated bacteria a “list” or “array” which may be accessed.



Figure - instantiating new bacteria

So, what does class “Bacteria” do? Answering that may take a while because more than half of the program is within this class. There are a number of functions within this class however when running it will immediately jump to the “Bacteria()” function which also takes the file name as an input. First this function sets the memory required for each vector through calling the “InitVectors()” function. Here some variables are set but most importantly the vectors are assigned a two-dimensional space with each address being the size of a “long”.



Figure - vectors being assigned memory

From here it reads the file until “EOF” is true and stores the relevant data into buffers (the data is simply a bunch of capitalized letters). It also keeps track of the total number of characters and divisions, so the data may be properly analysed. The next part is where the random distribution percentage (the “stochastic” variable) is calculated. Essentially the aforementioned buffers are iterated through and used to calculate a new “stochastic” on each iteration. However the calculated value is only saved if it is of sufficient size, this being greater than 1e-010. If “stochastic” is not of at least this size, then its value is simply set to zero.

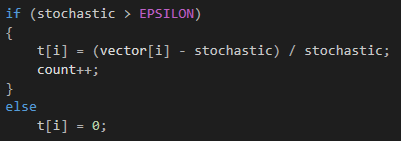


Figure - saving value if above 1e-010

The unneeded vectors are then deleted, and relevant data is saved for the next section.

Now that all the bacteria have been created and the necessary data passed in the comparison begins. For this a nested for loop is used, calling upon function “CompareBacteria()” which takes the bacteria in as arguments. Then within this function there are a series of while loops which essentially continuously compare the vectors calculated in the last section and add the value to one variable should the first be greater than the second (and vice versia.) Through this a correlation can be found and at the end of the function some simple division is performed to find a percentage similarity.

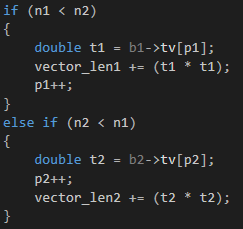


Figure - sorting the vectors based on value

Finally the percentage correlation is printed to screen and thus ends the program.

Profiling results and bottlenecks

Over an average of 10 tries the running time of the sequential program came to 58.3 seconds. Considering the number of loops there are, there is certainly a lot of room for speedup through parallelization.

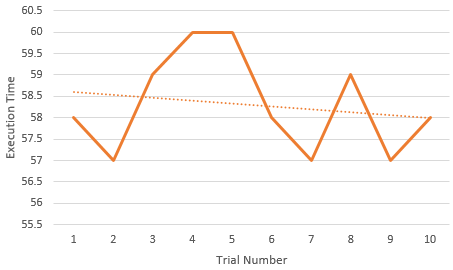


Figure - Average execution time of sequential

After running the profiler in Visual Studio a number of critical sections have been identified which the program spends a large percentage of its time on, these are prime candidates for parallelization and will be identified below.

1/ Below is the most impactful section of the code, taking up well over half the amount of computing time of the entire program. This is no surprise considering this is where all the files are passed in and analysed. Upon studying this loop it becomes apparent there are no data dependencies from iteration to iteration and so parallelization should be relatively simple.

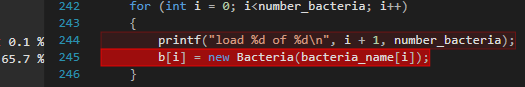


Figure - CompareAllBacteria() loop 1

2/ This nested for loop is used to compare each and every bacteria and so is expected to have a large impact. Upon analysation the outer loop has a flow dependency with respect to the inner one due to the inner loops value being “j = I + 1”. This means the outer loop is not parallelizable which is especially unfortunate due to it being much higher granularity than the inner loop.

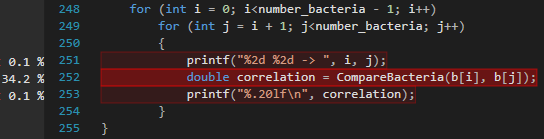


Figure - CompareAllBacteria() loop 2

3/ Below is the segment of code which fills the arrays which will be used for comparison later on. However, the value is only stored if it is non-zero. This means there will be an incredible amount of iterations as shown by the 6.8% impact on running time. Simply parallelizing the loop is not possible due to the clear flow dependency that the “pos” variable creates because it is read from and then written to on specific iterations. This means that the only conceivable approach for parallelization would be to perform an atomic add, however this would be very difficult and perhaps not even significantly decrease running time so it is unlikely this will be implemented.

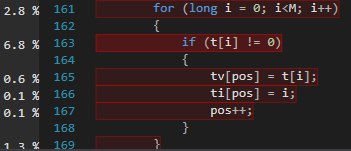


Figure - Bacteria() checking if vector == 0

**The process of parallelization:**

What tools were used?

Open MP was the library used to parallelize this application. All the tools used for speedup can be found within it.

The primary tool was Open MP’s go-to way for parallelizing for loops “#pragma omp parallel for”. As time went on there were various phrases added to this statement so the threads behaviour could be specified. This includes specifying the number of threads with “num\_threads,”, the type of scheduling to be used with “scheduling(#{scheduling type})” and the chunk size each thread would have to handle by simply adding a second parameter to “scheduling()”.

Analysing and parallelizing of sections

This section will detail all the changes made to parallelize the program

The first section to be parallelized was detailed in the above section (figure 8). It was concluded that there were no data dependencies and so parallelization would be an easy task. However when the loop was parallelized using “#pragma omp parallel for” the program would crash with an error stating there was not enough memory. After some investigating it was discovered that Visual Studio was running on a 32 bit environment. Due to the huge amount of data being used Visual Studio was set to run at x64 which seemingly fixed the issue.

The next bottleneck to be addressed was the nested for loop that compared the bacteria (figure 9). As was analysed in the previous section it is not possible to parallelize the outer loop due to the flow dependency. After some research a possible solution was found in Open MP’s “collapse” directive. Essentially this would combine both loops into one and provide one single large iteration space. However the results of the program were undoubtedly incorrect, this means that ultimately the data dependency was still there.

Upon analysing the loops in more depth the answer became clear

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| (outer loop) iteration | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | ...M-1 |
| (inner loop) iterations | 1->M | 2->M | 3->M | 4->M | 5->M | 6->M | 7->M | 8->M | …(M-1)->M |

The table above details the values that each loop iterates through at any given point. As you can see there is a clear data dependency (blue arrows) vertically between the two loops which means that each iteration of the outer must correspond with the inner, however there is no such restriction in the horizontal direction. This means it does not matter in what order the inner loop iterates and so may be parallelized.

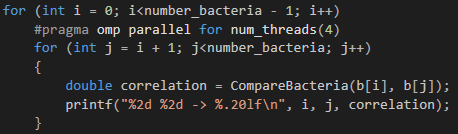


Figure - parallelized nested loop

Once this was implemented the results were correct however there was a disconnect between the two printf statements due to the unpredictable nature of threads. This was fixed by simply combining the two printf statements together as can be seen above in figure 11.

This particular section of code (figure 12 below) is not nearly as impactful as previous sections at only 10% but was still worth investigating nonetheless. At this point only the memory of the array “t” has been set and so it has no values. This if/else statement ensure that the current position does have a readable value even if the random probability distribution (stochastic) is below what is needed. As an attempt at speedup the else statement was removed and instead above the loop in which it is placed another loop was created to simply initialise the entire array to zero. This loop was then parallelized for maximum efficiency. However upon running the program actually took 4 more seconds to execute. It is believed this was due to the sheer number of iterations it had to perform (64,000,000). The granularity was simply too coarse to be an efficient change even after parallelization.

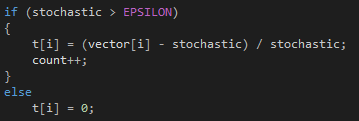


Figure - if/else stochastic

At this point all high impact sections had been addressed and so runtime had massively dropped. A new CPU profiler was run and as expected the CPU was certainly under more duress, however around ¾ through there was a very apparent dip in CPU intensity. It soon became clear that this corresponded with the beginning of the second loop in “CompareAllBacteria()” (see figure 9). This is because for each iteration the outer loop performs, the inner loop performs one less thus gradually reducing the amount of time each thread would be working before being set free to do another iteration. The most effective way to fix this was to use the scheduling capabilities of Open MP. Through this the scheduling of each thread was set to “dynamic” to ensure for much more flexibility.

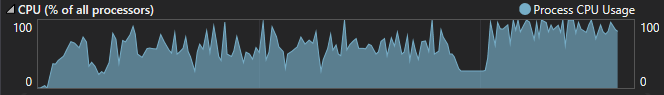


Figure - CPU usage profile before scheduling

**The Results:**

Timing and profiling results

Now that the program has been parallelized to the best of my ability the program was run with a slowly incremented number of cores to measure speedup as figure 14 displays.

Towards the beginning of this assignment it was stated that the average run time for the sequential program was 58.3 seconds. However the machine it was run on only possesses 4 cores in total. Therefore a more modern CPU had to be used to satisfy the criteria. It is interesting to note that this older CPU did have a similar speedup of 1.95 using 4 cores which meant that it was almost as efficient as the results you see below.

The sequential algorithm on this computer ran significantly faster at 25 seconds due to the much more modern core i7 CPU the computer possessed.

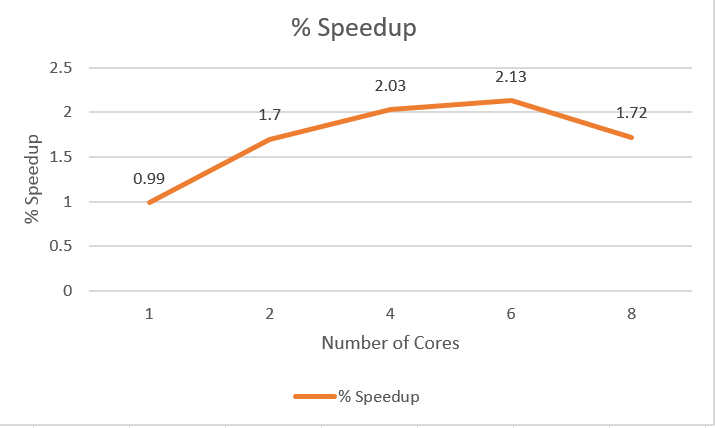


Figure - % speedup graph

The results in figure 14 were as expected, that is until it was run using 8 cores. When run on one thread the program ran an infinitesimally small amount time longer than sequentially. This is because running on one thread is essentially running the program sequentially however there is an initial overhead to create the thread itself, while this is small it does impact performance as can be seen. At two cores there was a 1.7 speedup which was anticipated because while it makes logical sense for the program to run twice as fast there are lots of factors such as thread overhead and hardware inefficiencies which will slow it down. From there the speedup significantly slows and as can be seen there is less speedup from 2 to 6 cores than there is from 1 to 2. It was thought that there would be more speedup at this point, however reflecting upon the changes made within the program reveals the problem. There were a total of 7 loops within the program and only 3 of these were significantly improved through parallelization. This means a lot of the program is still in sequential format and so the speedup was always going to be very limited.

However, there is an unexpected twist towards the end of this graph where the performance actually dips significantly when the cores are increased from 6 to 8. After some analysis it is believed this is because the program itself is not that large, relatively speaking. At a certain point any sort of parallelization will become inefficient, if there are too many threads being used then they will simply end up queueing behind each other, wasting precious time with needless overhead and context switching. This thresh hold is widely considered to be around number of CPU cores + 1 with the one waiting for the disk I/O. Despite there being 8 cores the speedup did decrease, this can only be attributed to the positives of adding threads being outweighed by the negative due to the program not containing enough coarse grain parallelism.

**Conclusion:**

Until this point in my degree I had only been creating pieces of software for assignments through following some criteria. However, this task forced me to flip that norm on its head and essentially reverse engineer the code of somebody else. It has occurred to me just how important this skill is considering in the real world almost nothing of significance is done individually and so when working with a team you must be able to understand and build upon others code.

The speedup of the program was lower than originally planned, perhaps more could have been done to achieve a more perfect parallelization. I believe that time itself played no part in this because this assignment was started well in advance. Despite some shortcomings the important thing is to recognize and understand why something unexpected occurred and learn from it. Through lots of research and analysis I believe I have achieved that and reflected it within my report.

# References

Mystical. (2011, September 11). *Why do C c ompilers specify long ot be 32-bit and long long ot be 64-bit?* Retrieved from stackoverflow: https://stackoverflow.com/questions/7279504/why-do-c-compilers-specify-long-to-be-32-bit-and-long-long-to-be-64-bit

Supercomputing Blog. (2009, July 13). *Turorial - Parallel For Loops with OpenMP*. Retrieved from The Supercomputing Blog: http://supercomputingblog.com/openmp/tutorial-parallel-for-loops-with-openmp/

University of Mary Washington. (n.d.). *OpenMP Scheduling*. Retrieved from http://cs.umw.edu/~finlayson/class/fall14/cpsc425/notes/12-scheduling.html

Zugzwang. (2010, May 8). *stativ cs dynamic scheduling*. Retrieved from ubuntu forums: https://ubuntuforums.org/showthread.php?t=1477278