High Performance Computing: Serial Optimisations

By Josh Philip Hamwee – jh17602
School of Computer Science, University of Bristol

(1.) Introduction

The aims of this assignment were to take a starting stencil code, and apply some serial optimisations to improve performance, on a single core. The code will be run on the Blue Crystal supercomputer at the University of Bristol, however not making use of the multi core capabilities.

There are three areas in which I have focused to try and improve the performance time of this code. I will be discussing the results and issues from all sections below.

(2.) Compiler & Flags

The first place I decided I would try and make up some time was with compiler optimisations. With modern day compilers, if you chose the right flags and compiler type you can drastically improve the performance. The first choice I would have to make would be GCC vs ICC. After some initial research [1] and testing the stencil program on both compilers I could see that the Intel compiler had a lot more potential.

Compiler	*Image size*	*Time/s*
gcc 4.8.5	1024x1024	5.62
icc 2018-u3	1024x1024	0.192

As you can see the Intel compiler was initially much more efficient that the gcc compiler. This is due to a number of reasons but mainly because the Intel compiler's auto vectorisation features increase the performance of the code by a large amount. From this point forward I did not look at the gcc compiler as I felt that the Intel compiler was going to outperform it in all areas.

The next steps in which I took to increase performance was that of the compiler flags. I looked through the documentation and figured out what flags would suit the program, and then ran many tests which can be seen below.

Compiler	*Flags*	*Image size*	*Time/s*
icc 2018-u3	-00	1024x1024	0.190
icc 2018-u3	-01	1024x1024	0.185
icc 2018-u3	-02	1024x1024	0.185
icc 2018-u3	-03	1024x1024	0.189
icc 2018-u3	-fast	1024x1024	0.110
icc 2018-u3	-xHOST	1024x1024	0.180

What we can take away from data is that the -fast flag is very good for optimising performance, however it may not be the most accurate. Due to it not making any errors in the comparison, I am going to keep it for the final flag. -O2, which is also the default flag, was the fastest, so I will also be using this for now. However, it may have just been an effect of noise as to why -03 was slower because later on in the research I discovered that it actually ran faster. This may have been to it performing under other changes which we will discuss later [2]

So, coming away from the compiler and the flags optimisations, we have taken the time down to 5.62s with gcc and no flags, all the way down to 0.15 with the Intel compiler and these flags: -fast -xHOST -O2. This is roughly a 35x speedup and have already hit the benchmark for our necessary speedup required.

(3.) Data Types

One area in which we were encouraged to look at to optimise the performance of our code was the data types used in the program. This can have a large effect on the performance of the code as different data types may require more storage and hence take longer to make calculations on. [3]

When analysing the code, I noticed that all the number variables were stored as doubles. This made me consider other options to store them. After a bit of research, I discovered that a double is a 64-bit datatype, whereas we would use an alternative such as a float which is only a 32-bit floating point number. After making the change of all variables, except the timing as we want to keep this as accurate as possible, to float the performance time made a significant speed up from 0.15s all the way down to 0.110s

This is the point in which I decided to use the -O3 flag as I discovered that it compiles more efficiently for heavy floating-point calculations and large data sets. This brought down our performance time all the way to 0.101s. By this point I feel that the code has reached a very optimised time, however I still felt that we could reach a sub 0.1s time with a few more adjustments.

(4.) Vectorisation

When we started out using the gcc compiler, there was no sign of loop vectorisation. Once we switched over to the Intel compiler, we could see a drastic increase in performance. I put this down to Intel's auto-vectorisation

features. I ran the compiler with Intel's inbuilt flag -qopt-report=2 -qopt-report-phase=vec. What this produced was a report on which loops were vectorised. The report told us that the **Permuted Loop was Vectorised**. This meant that in the stencil function, the loops were vectorised and needed no adjustment to increase optimisations. If we had continued using the gcc compiler then we would have had to implement vectorisation in a more concrete and specific way.

(5.) Data Access Pattern

The final way in which I tried to increase the performance of the code was that of the way in which data was accessed from an array. The choice was between Row-Major vs Column-Major. Making the right choice can increase the performance due to the way the array is stored in memory. [4]

? - Major	*Image size*	*Time/s*
Row	1024x1024	0.094
Column	1024x1024	0.101

Originally, we were iterating through in a column major layout, however it would be more efficient to switch to row major as each row is loaded into memory. This means that we do not have to reload the cache each time we want to access one item in the array.

(6.) Conclusion

Below I have shown the original timings that I first recorded with no flags on the gcc compiler, in comparison to the timings that I recorded after all the optimisations that I have mentioned above.

Image Size	*Optimised Time/s*
1024x1024	0.098
4096x4096	2.957
8000x8000	11.19

To conclude the results, and the rest of the research, I believe that the optimisations that I have chosen and implemented were, on the whole, very effective. Although I did not implement vectorisation as far as I would have liked to, the final results show that the timings are beyond what was expected of us. I am pleased with the findings and feel that there is plenty of more detail to explore if one was to look into MPI and multi-core programming.

References

[1] *Comparison of Different Compilers.* A Aggarwal,

[2] *Intel Compiler Documentation,* https://software.intel.com/en-us/articles/intel-cpp-compiler-release-notes

[3] *Efficient use of Data Types*, https://docs.microsoft.com/en-us/dotnet/visual-basic/programming-guide/language-features/data-types/efficient-use-of-data-types

[4] *Row-Major vs Column-Major*, https://www.geeksforgeeks.org/performance-analysis-of-row-major-and-column-major-order-of-storing-arrays-in-c/