Imperial College
London

High-performance Computing Coursework

Department: Department of Aeronautics
Course: MEng in Aeronautics
Module: AERO70011 High-performance Computing
Date: 22nd March 2023
Name: William Ho

1 Introduction

The aim of this coursework assignment is to solve the 2D shallow-water equations by writing code that where the focus lies in the computational performance, utilising parallelisation. The shallow-water problem was addressed in two ways, using a loop implementation and using BLAS for matrix-based evaluation. The code was then profiled using oracle developer studio to find areas of performance gain.

2 Results

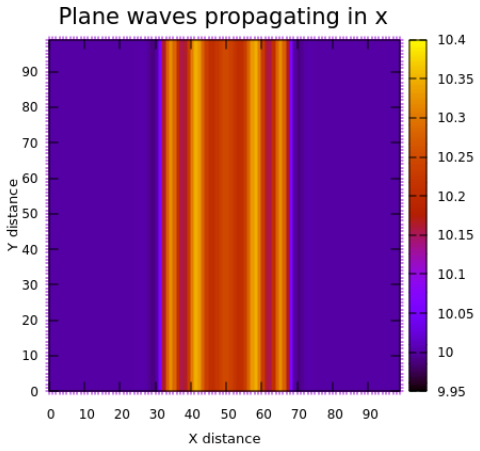

Figure 1: Height at T=80 with Case 1

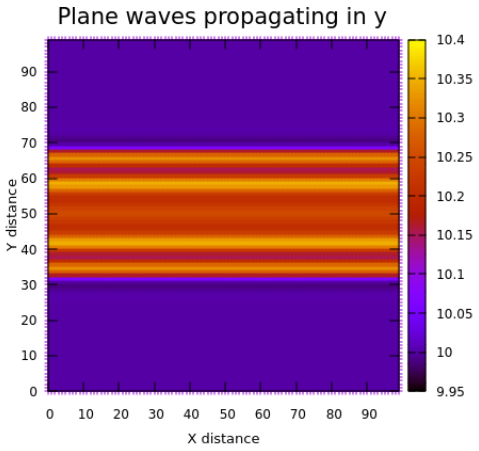

Figure 2: Height at T=80 with Case 2

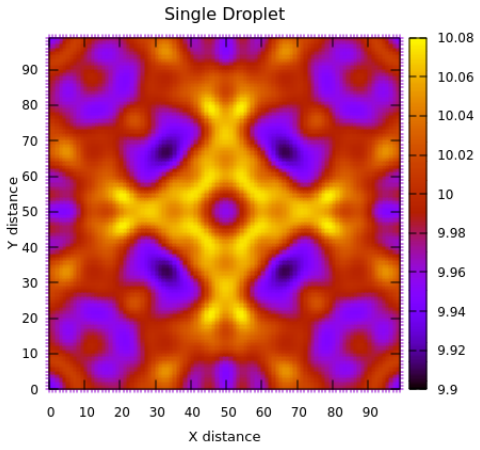

Figure 3: Height at T=80 with Case 3

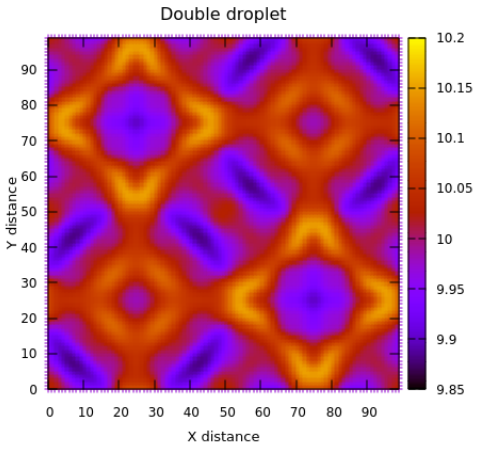

Figure 4: Height at T=80 with Case 4

Figures 1 and 2 show symmetry along their respective direction of propagation at T = 80. The contour of the shallow water equation for a single droplet and double droplet is also shown on figure 3 and figure 4, respectively.

3 Optimisation and Performance

There were several methods to increase the performance of the code, more specifically the run time of the code was analysed using Oracle Developer Studio to assist in identifying areas of code that is causing the

most computational effort. The following optimisation methods were utilised:

- Reordering loops to improve cache locality. This was enforced by making sure that nested loops utilised column-major method for matrices. Therefore, the loop was implemented to go down each row and finish the column before moving to the next column. This would then help cache locality as it will reduce the number of times the system will have to access the memory location.
- Simplify mathematical expressions. This was implemented by using multiplication signs over floating-point division as division is a more computationally intensive operation than multiplication. This was evidently used for the stencil matrix where coefficients such as $\frac{3}{4}$ and $\frac{3}{20}$ was replaced with 0.75 and 0.15, respectively.
- Pre-compute repeated expressions. This was implemented heavily throughout the code as there were many terms that repeated throughout the TimeIntegrate() function. By applying all these changes, the code was able to achieve a run time that was 3s faster than its previous version.
- Lift loop-independent expressions. There were very few scenarios where loop-independent expressions occurred, however where they did exist, it was combined with pre-computed expressions and i was found to produce marginal increase in performance.

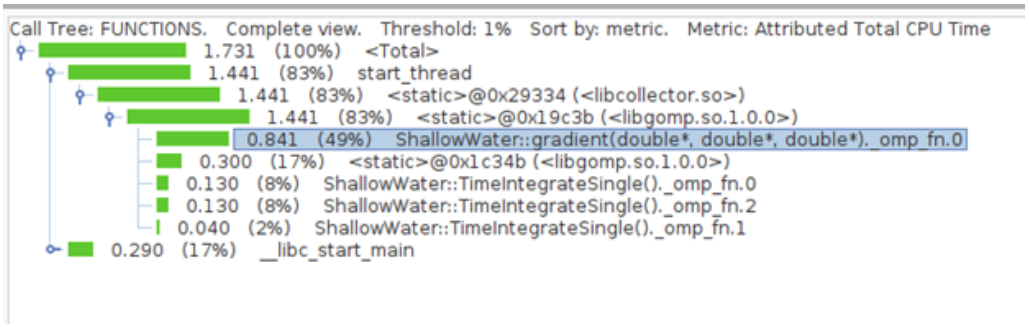

Figure 5: Oracle Developer Studio interface example when analysing the unoptimized code

Interestingly, a common method to aid further optimisation, inlining a function, actually increased the run time by 4 seconds. This may be due to the fact that the function was only called twelve times for each timestep and so the trade-off between calling the function which is not "free" and the computation for inline function was poor. There could be several reasons for this such as cache miss and increase memory usage due to the calling function becoming larger due to the inline function.

4 BLAS and loop-based versions

The loop-based method was found to outperform the BLAS version for serial and parallel at every amount of threads used. This could be because the size of the problem being solved. If the matrices were much larger, it may have been more favourable to use BLAS. This report covers the case for $N_x = N_y = 100$, and so the matrix size explored would not be considered large in industry. In addition, optimization was a lot more throrough with the loop-based method compared to BLAS as there were more features within the code to optimise. Also, there is no direct access to the BLAS subroutines which makes it more limited in terms of optimization. The computation of the spatial derivatives can also be changed slightly for the loop-based method such that there will be less computation in the sense that there will be three terms rather than six terms if the variables with the same coefficient magnitude are grouped together. Moreover, there are many versions of BLAS and the specific BLAS used for this report was developed by Netlib and may not be the most efficient or optimal form of BLAS. Figure 6 shows the comparison in performance between the loop-based implmentation and the BLAS method. This result may not reflect in the same manner for different size problems or using BLAS that was developed using a different algorithm.

5 Parallelisation

The loop-based method and BLAS were both parallelised using OpenMP. Given the time constraint for the project and the additional effort to implement MPI, it was deemed unecessary to implement it as distributed memory and networked nodes was not a requirement. The aim was to effectively utilise parallel programming which would make the work of each thread equal. The aspect of the code where it was calculating the Runge-Kutta coefficients, it worked on each grid point of the U, V and H matrices for each timestep, putting the output in a new matrix, U2, V2 and H2 which would then be used for the next k coefficient. This process as well as other region of code had high use of for loops. This made it possible to use a better strategy because all threads could operate on the original matrices before they were changed by having the matrices updated at the end of the timestep. The use of '#pragma omp for' allowed work sharing methodology for the loops. As the loop used for this specific case was iteration independent, this allowed different threads to work on different areas of the loop and combine at the end. The distribution of the work to the threads were assigned as fixed sized chunks utilising 'schedule (static)'. This was distributed at the start to minimise overhead. For the BLAS method, the use of '#pragma omp parallel section' was also utilised.

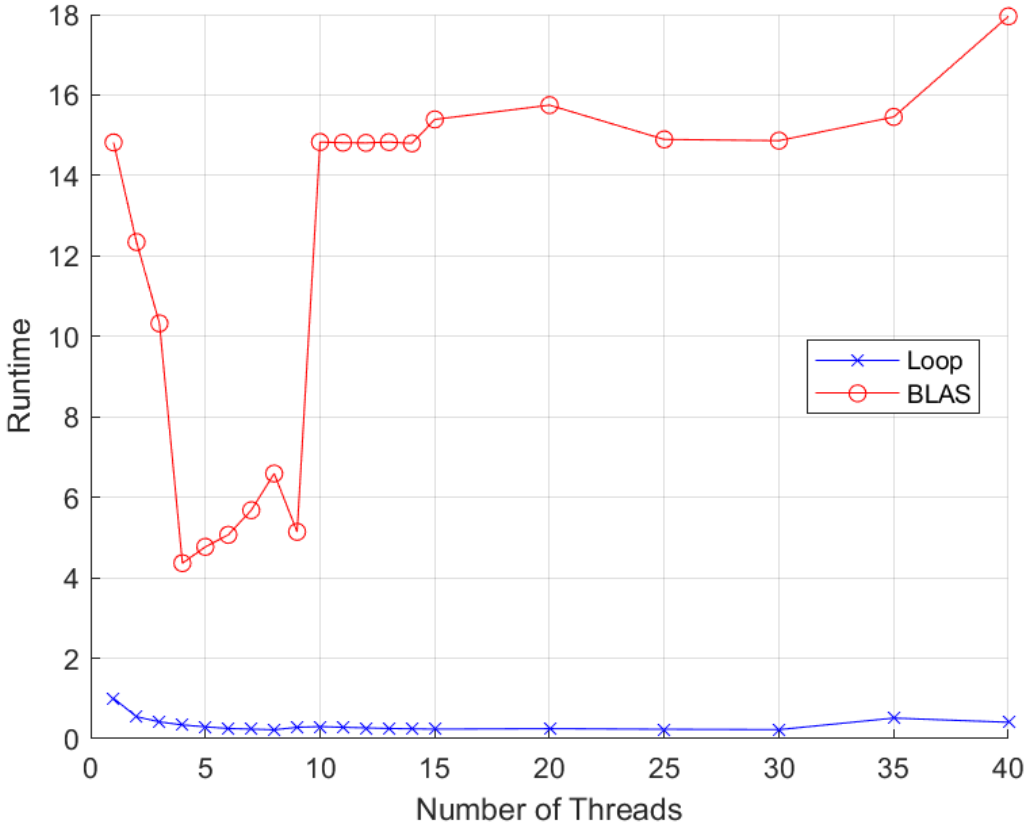

Figure 6: For-loop Runtime vs BLAS up to 40 threads

5.1 Parallel Scaling for Double Droplet Case

The ideal scenario for parallelisation is when increasing the number of threads using OpenMP would proportionally decrease the time to run the program. In other words, if the number of threads used went from one to two, the time to run tbe program is expected to halve, and if it went from one to three, the run time will be third of the runtime for the serial case. Therefore, the ideal case would be represented by equation 1

$$y = \frac{A}{x} + C \tag{1}$$

where A is the runtime for the serial case, x is the number of threads being used and C is the run time for the case if all the parallelised code managed to reach zero.. The chances of the parallel code working ideally

is rare in real life. This is usually because there will be aspect of code that cannot be parallelised. Computers have limited amount of cores and processors to handle threads in parallel. The parallelisation itself is not free as there will be associated overhead costs in running OpenMP such as initialisation, communication, etc.

When comparing the run time between methods of between number of threads being used, real time was used. During the testing, there would be fluctuations in run time due to other users connecting to the same remote desktop location. To mitigate these fluctuations, analysis for times were executed when the CPU percentage usage was low. Even during the low CPU usage, there were still some discrepancy and so the runtime was explored multiple times and a mean value was used.
The variation in run time to the number of threads can be seen on figure 7 and figure 8. From this plot it can be shown that, for low number of threads, both the loop-based and the BLAS methodology followed the expected trend of decreasing runtime with an increase in the number of threads. They also both exhibit behaviours where the runtime would jump up and increase the runtime from the previous thread value.

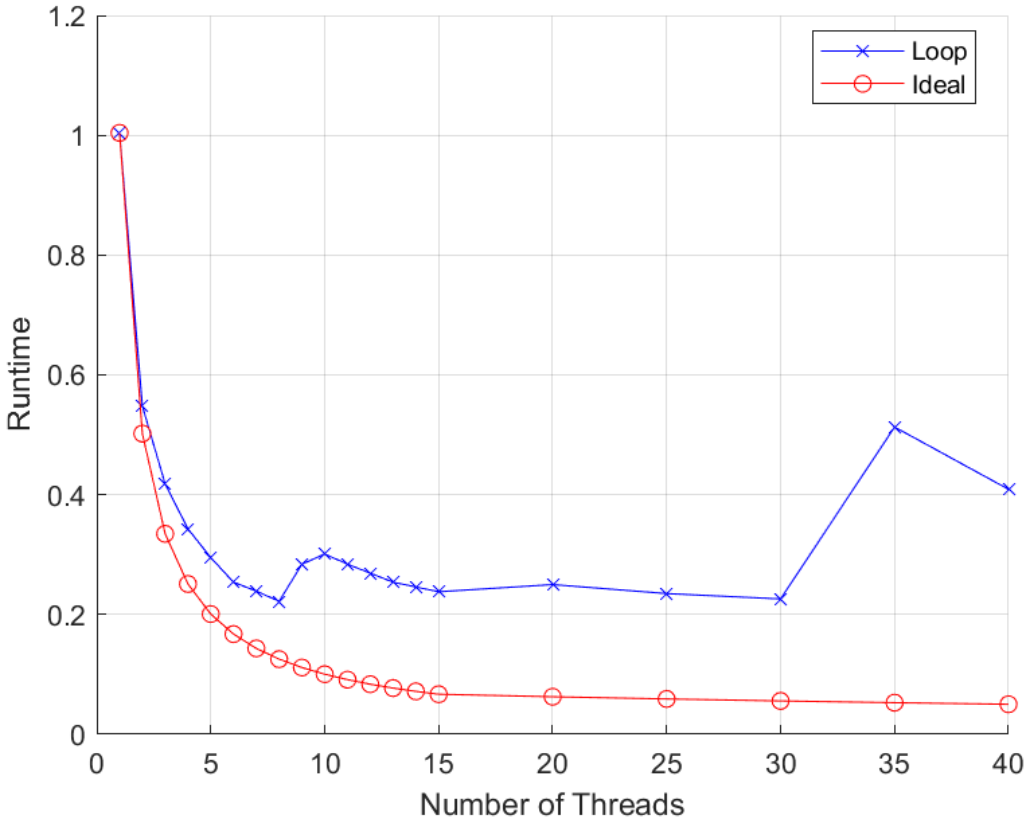

Figure 7: For-loop Runtime vs Number of Threads up to 8

The BLAS code stopped decreasing runtime after four threads were used, whilst the loop-based method found the lowest runtime using eight threads. The BLAS method even found a way to reach a runtime above the serial case, which may suggest that increasing the threads too much will lead the program to reach a bottleneck. The same can be seen for the loop-based method as it can be seen at 35 threads, the run time significantly increased.

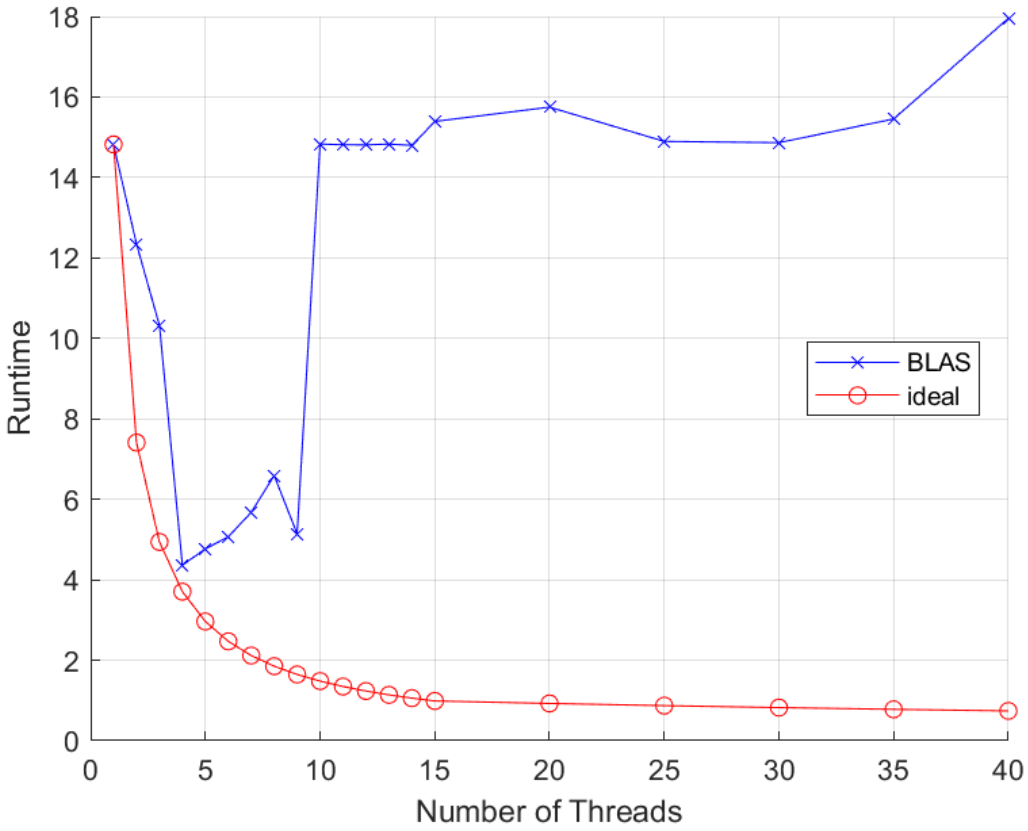

Figure 8: For-loop Runtime vs Number of Threads up to 8

References

[1] Dr. C. Cantwell. High-Peformance Computing Coursework Assignment. Available from: https://bb.imperial.ac.uk/webapps/blackboard/content/listContent.jsp?course_id=_35728_1&content_id=_2555436_1&mode=reset [Accessed 09/03/23]

[2] Dr. C. Cantwell. High-Performance Computing Course Notes. Available from: https://bb.imperial.ac.uk/webapps/blackboard/content/listContent.jsp?course_id=_35728_1&content_id=_2555435_1 [Accessed 09/03/23]

6 Appendix