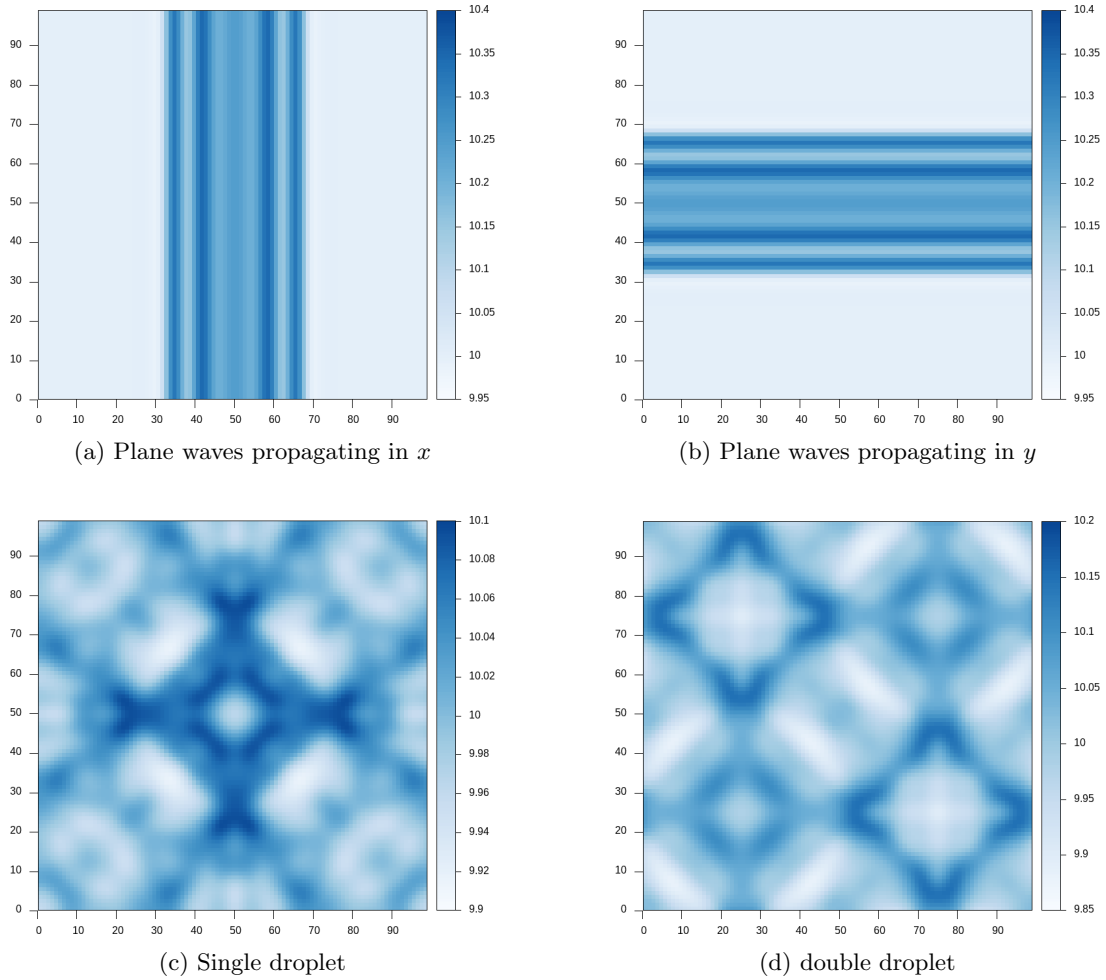
High Performance Computing (AERO70011) Coursework Report

By Chung-Hsun Chang Chien (CID:01771697)

1 Simulation Results



(a) Plane waves propagating in x

(b) Plane waves propagating in y

(c) Single droplet

(d) double droplet

Figure 1: 2D Contour Solution Plots

2 Optimisation Approach

Some Optimisation was taken place for both the serial and parallel code that aims to improve the performance, and are listed as following:

- Minimise the number of function called during each iteration of the time integration. Originally, functions (f_u, f_v, f_h) were created to calculate the right hand side function f for u, v and h field to make ease when calculating k_1, k_2, k_3, k_4 during the algorithm of 4th order Runge-Kutta[1]. The functions in the for-loop are now *inline*.

- The inner loop for the nested for loops that populates the matrices u, v, and h iterates through the rows first, exploiting cache locality for a column major storage matrix.

- The highest level of optimisation flag `-o3` was turned on to make the compiler attempt to improve

[1] An explicit numerical integration scheme that pre-calculates k, where $k_1 = f(y_n)$, $k_2 = f(y_n + \Delta t k_1/2)\ldots$. Runge-Kutta utilises the discretised time step Δt to update the values by formulating $y_{n+1} = y_n + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4)\Delta t$

the performance and/or code size at the expense of compilation time and possibly the ability to debug the program.[2].

- Uses `#pragma for nowait` in the `Spatial_Derivatives` function to remove the implicit boundary. Uses `collapse(2)` when parallelising nested for loops, this allows parallelising multiple loops in a nest without introducing nested parallelism.

- Minimises the number of `BLAS` function and iterate with for loops instead. This case we have more control on the algorithm and can better optimising by applying multiple threads in the process.

3 `For-loops` or `BLAS`

The program prompts the user to choose whether to proceed the calculation of spatial derivatives either using `for-loops`(1) or `BLAS`(2) (`--choice` (default = 1)). During the parallelisation process, `for-loops` overall have a higher performance and lower runtime compare to `BLAS`, enables us to distribute workloads more efficiently across threads, and easier to apply parallelisation for for-loops. On the other hand, it is with more complexity and difficulty to parallelise `BLAS` function (particularly `dgemv`), this is mainly due to the following reasons.

1. Data dependencies when performing linear algebra operations in `BLAS`. Matrix-vector multiplication requires intermediate steps (such as multiplying first row to first column before moving to the next) to be computed in specific order, therefore making parallelisation challenging.

2. Load balancing, `BLAS` operations may not distribute work evenly, leading to an imbalance load and reduced performance.

3. Synchronisation overhead, parallelisation often requires synchronisation between threads or processes, which can add extra overhead and reduce performance.

4. Requires additional programming efforts which also introduce additional complexities, such as race conditions and deadlocks.

Total CPU Time		Name
EXCLUSIVE sec.	INCLUSIVE sec.	
16.662	16.662	*<Total>*
11.458	11.468	dgemv_
3.142	14.770	ShallowWater::Partial_Derivative(double*, double*, double*, int)
0.290	0.290	ShallowWater::TimeIntegrate()._omp_fn.3
0.230	0.230	ShallowWater::TimeIntegrate()._omp_fn.1
0.190	0.190	ShallowWater::TimeIntegrate()._omp_fn.2
0.180	0.180	ShallowWater::TimeIntegrate()._omp_fn.0

Figure 2: Runtime for `dgemv` in Oracle Profiler

Figure 2 shows the inclusive and exclusive runtime when calling the function `dgemv` during every time iterations using parts of *Oracle Developer Studio suite* [3]. We can see that the time for our code to iterate through `dgemv` takes up almost half of our computational time.

[2] https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html

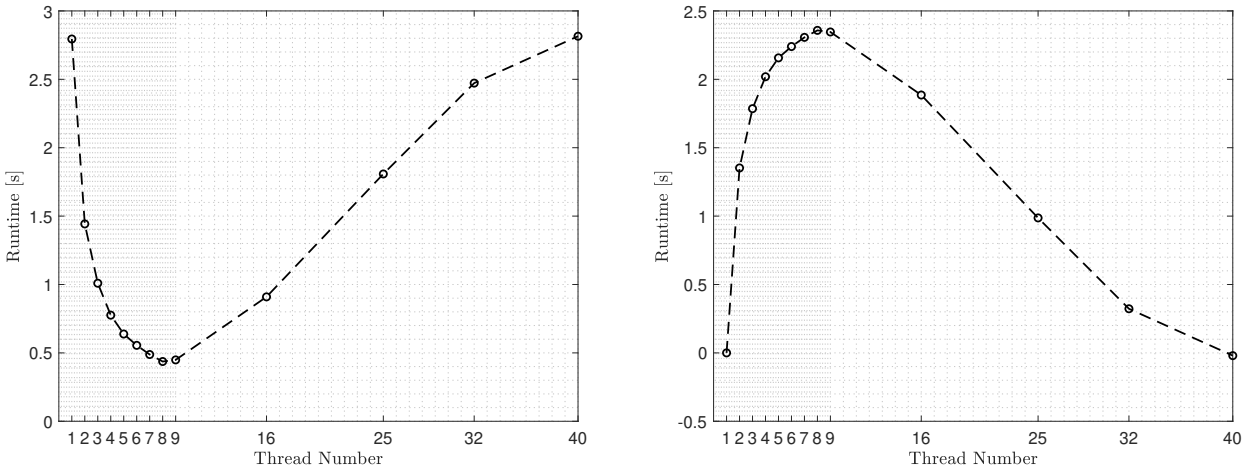
[3] A profiler analysis the performance and its resource usage to provide feedback on how well our code performs and what improvements can be done. Oracle dev studio includes a command-line tool for collecting performance statistics and a graphical tool for examining and analysing the collected data. The command line used is the following, `collect -o test.er ./main`

4 Parallelisation Approach

The simulation code is parallelised with *shared-memory* paradigm using the OpenMP API, which uses multi-threaded shared memory to achieve parallelism within a single process. Parallelisation using *message-passing* following the MPI standard, however, is not been incorporated in this program, due to the complexity and the nature of having potential latency between communications, and non-receiving message that leads to deadlocks. It is possible though, to combine both shared-memory and message-passing parallelism to leverage the performance by combining both advantages.

The objective of parallelisation for this simulation is to effectively reduce the time to solution. There are several approaches in terms of the architecture of openMP codes that's being distributed in the serial code that will be address in this section. One of the biggest advantage of parallelising using openMP is the ability to easily implement and add parallel code to an already working serial code, without the need of establishing communications between ranks. For the sake of making the code more modular and easy to maintain, a function calculating `Spatial_Derivatives` is preserved and will be constantly called during the `Time_Integrate` function. Despite this helps to reduce code duplication and increase reusability, it might have a negative implication on the performance metrics. As each function is called, it introduces overheads that will be aggregated up, including the cost of calling the function and returning arguments back.

In the `Time_Integrate`, `#pragma omp parallel for collapse(2)` were called when iterating through a nested loop when updating the k matrices. A parallel region was also created in the `Spatial_Derivatives`. To pre-define the first and last three columns/rows when implementing the periodic boundary conditions, `#pragma omp for nowait` directives was used to to assign loop iterations to threads in a round-robin fashion. The `nowait` directives suggested that each thread updates a node and there's no implicit synchronisation at the end of the loop to boost performance.



(a) Runtime

(b) Speed up (relative to serial)

5 Parallel Scaling

By incorporating parallelisation with multiple threads distributing the workload, it is also worth discussing on the performance in terms of runtime and speed-up relative to serial code when using different numbers of threads. This section of analysis was done using `std::chrono::steady_clock::time_point begin = std::chrono::steady_clock::now()` with `end` at the end of the time loop. This is also verified by using the departmental Linux machine *Typhoon* with the thread feature in *Oracle Developer Studio*. Noteworthy that the interactive Linux systems are throttled to 8 cores per user, so 10 threads can still only offer a maximum of 8x speed-up over 1 thread.

From figure 3a, we can see that the code benefits from parallelism by speeding up almost 2.5 sec when compare to serial (Figure 3b). Ideally, with more threads provided, the code should maintain the same (or better) level of efficiency, however, this is not the real case. As the number of threads set in the

environment exceed the number of CPU cores, the performance actually started degrading. With more threads, the partitioning a fixed amount of work among exceed numbers of threads give too less work that the overhead of starting and terminating threads swamps the useful work[4]. Time-slicing in threading ensures that all threads can do at least a minimum amount of work, but with an over number a threads, fair distribution can cause overhead. One of the more significant overhead of time slicing is saving and restoring a thread's cache state. As computer tends to prefer accessing cache memory because of the speed than accessing from the main memory. However, with an over number of threads might cause the eviction of data from cache due to the cache is full. Threads tend to fight over each other's eviction of data, where the cache fighting from too many threads can hurt performance.

With more threads also indicates more time were spent in idle. It also creates more forking and joining while designing the parallel region for performance, which will potentially hinder the overall performance as well.

[4]https://www.codeguru.com/cplusplus/why-too-many-threads-hurts-performance-and-what-to-do-about-it/