AERO70011 HIGH PERFORMANCE COMPUTING
Coursework Solutions

Author: Priyam Gupta CID: 02110124

1. Figure 1-4 are the plots for the test cases the required four test cases at T = 100.

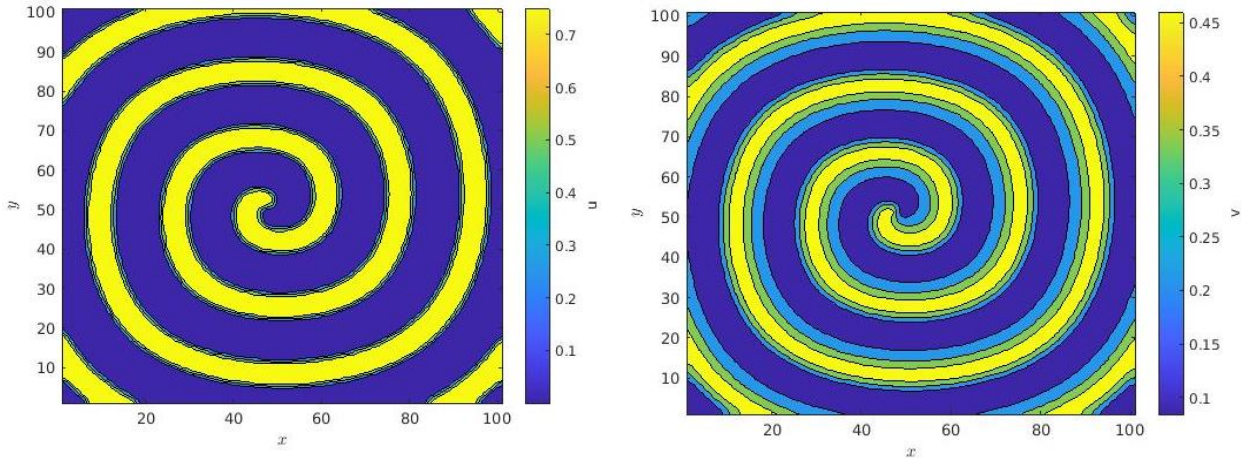

Figure 1. *Left: Contour plot of u for* ***Test 1****. Right: Contour plot of v for* ***Test 1****.*

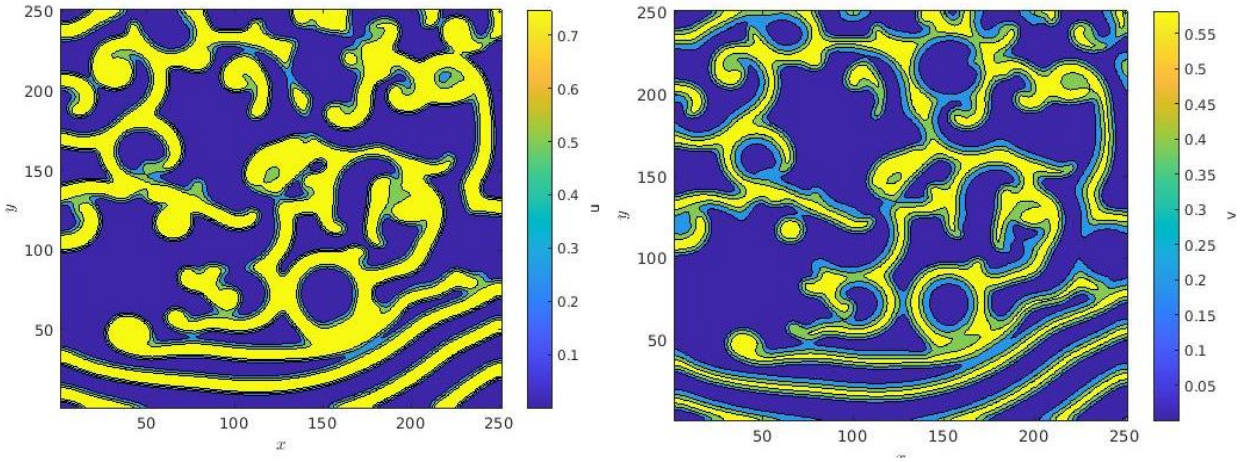

Figure 2. *Left: Contour plot of u for* ***Test 2****. Right: Contour plot of v for* ***Test 2****.*

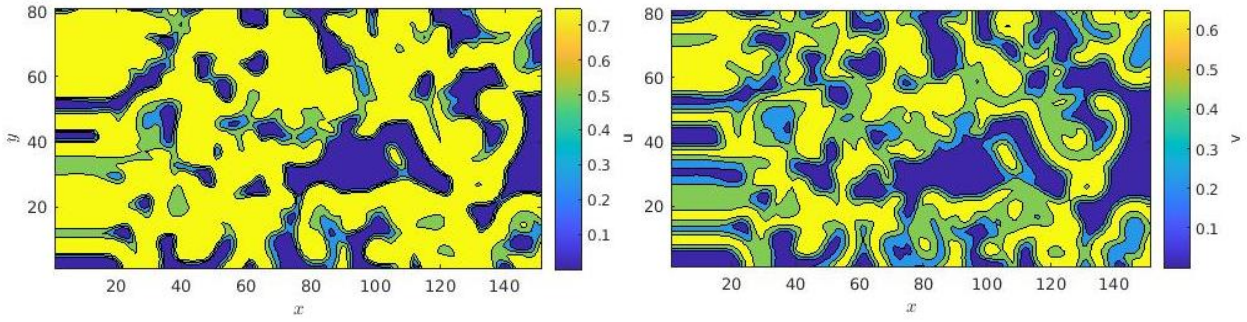

Figure 3. *Left: Contour plot of u for* ***Test 4****. Right: Contour plot of v for* ***Test 4****.*

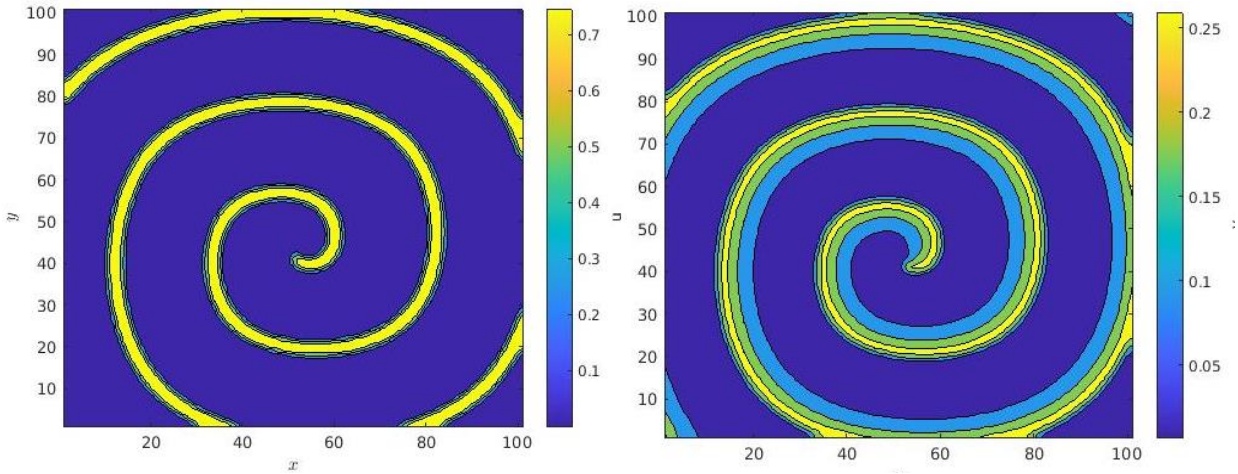

Figure 4. *Left: Contour plot of u for* ***Test 3****. Right: Contour plot of v for* ***Test 3****.*

2. OpenMP was used in this work to parallelise the program implemented for solving the Barkley model. The scheme used for this purpose is *explicit* which makes the calculation of values at the next timestep spatially independent of each other. This independence makes a ***do/for*** **work sharing construct** a suitable method to parallelise the given problem. The program was designed such that maximum processing can be done within a single loop to ensure optimal parallelisation. The *for* work sharing construct was used at following stages:

i. Initialization of the U and V matrix
ii. Calculating the velocity values for the next time step
iii. Copying the updated velocity values to the matrix containing previous time step values.

The 2D nature of the problem meant that all the *for* loops are nested. An attempt was made to collapse the nested loops manually to a single loop by calculating the indices using the modulus and division of iterator. However, an inbuilt *collapse* attribute proved to be much faster and was utilised in the final implementation. The default **static scheduling** which evenly distributes the iterations over the threads was used since it minimises overhead when the workload is approximately equally distributed amongst all the iterations which is true for the spatial iterations involved in this problem. Message passing model like MPI wasn`t used as it required heavily refactoring of the serial code and was more difficult to optimise as compared to OpenMP leading to subpar performance.

3. Figure 5 shows the parallel scaling for Test 1(b) and 1(d).

4. In an ideal case the execution time should decrease with an increasing number of threads as more work is being processed simultaneously.

Test 1(d) `s performance improves with increasing threads but an increment in execution time is observed beyond 16 threads. This significant overhead at higher number of threads can be attributed to the non-square nature of the matrix in Test 1(d). The matrix is stored in a column major format therefore the non-symmetric matrix can lead to sub-optimal cache locality due to non-uniform distribution of the columns. Threads that receive incomplete columns will have to

access the whole cache line from the main memory which takes multiple clock cycles thereby increasing the execution time.

In Test 1(b) the execution time decreases initially but starts to converge to an average of 5.3 seconds at higher threads. This could be due to increased latency caused by distribution of iterations amongst higher number of threads. Another potential reason can be distribution of resources amongst other users leads to interruption when more threads are used in spitfire.

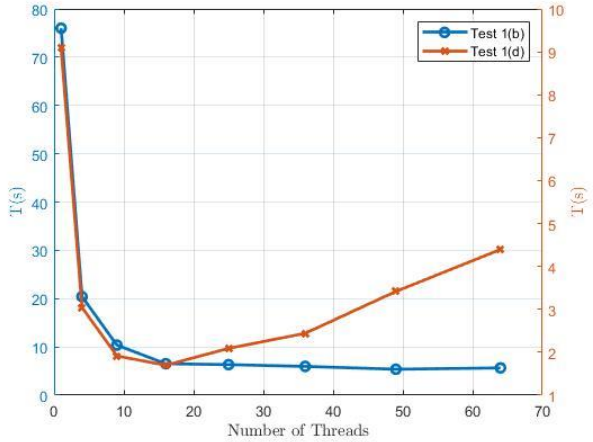

Figure 5. *Parallel Scaling for Test 1(b) [Left scale in blue] and Test 1(d) [Right scale in Red] on spitfire.*

5. Following optimizations were made to improve the code performance:

- To minimise the number of conditional statements required to impose the boundary conditions a dummy variable approach was used. This involved using an extra layer of dummy points along the boundary which had the same value as that on the boundary. The scheme used in the grid can also be used at the boundary while ensuring that the Neumann condition is still imposed. This reduced the computation time by half at a cost of minute increase in runtime memory. This change had the maximum impact on the performance of the program since it substantially reduced the number of times conditional statements are accessed within the Time Integration Loop.
- Time Integration loop took maximum processing time in the whole program therefore it was ensured that minimum number of functions are accessed within the loop. This was done by calculating the forcing function during spatial iterations instead of creating a separate matrix and function. It helped improve time and reduced the runtime memory.
- The matrix indices were precalculated which reduced the repetitive computation done to access the same element multiple times.
- The matrices were stored in a column major format so the *for* loops were ordered such that columns are accessed first in order to maximise cache locality.

6. In this coursework, BLAS and LAPACK weren`t utilised for computing the solution. Since the problem is to time integrate the discretised partial differential equation over time, and not to solve a linear system of equations or an eigenvalue problem, there is no need for LAPACK. In order to ensure maximum parallelization of the *for* loop iterations all the BLAS operations were replaced with C++ implementations and a *for* construct was applied. Although these measures increased the serial computation time but a substantial improvement was observed in parallel implementation.