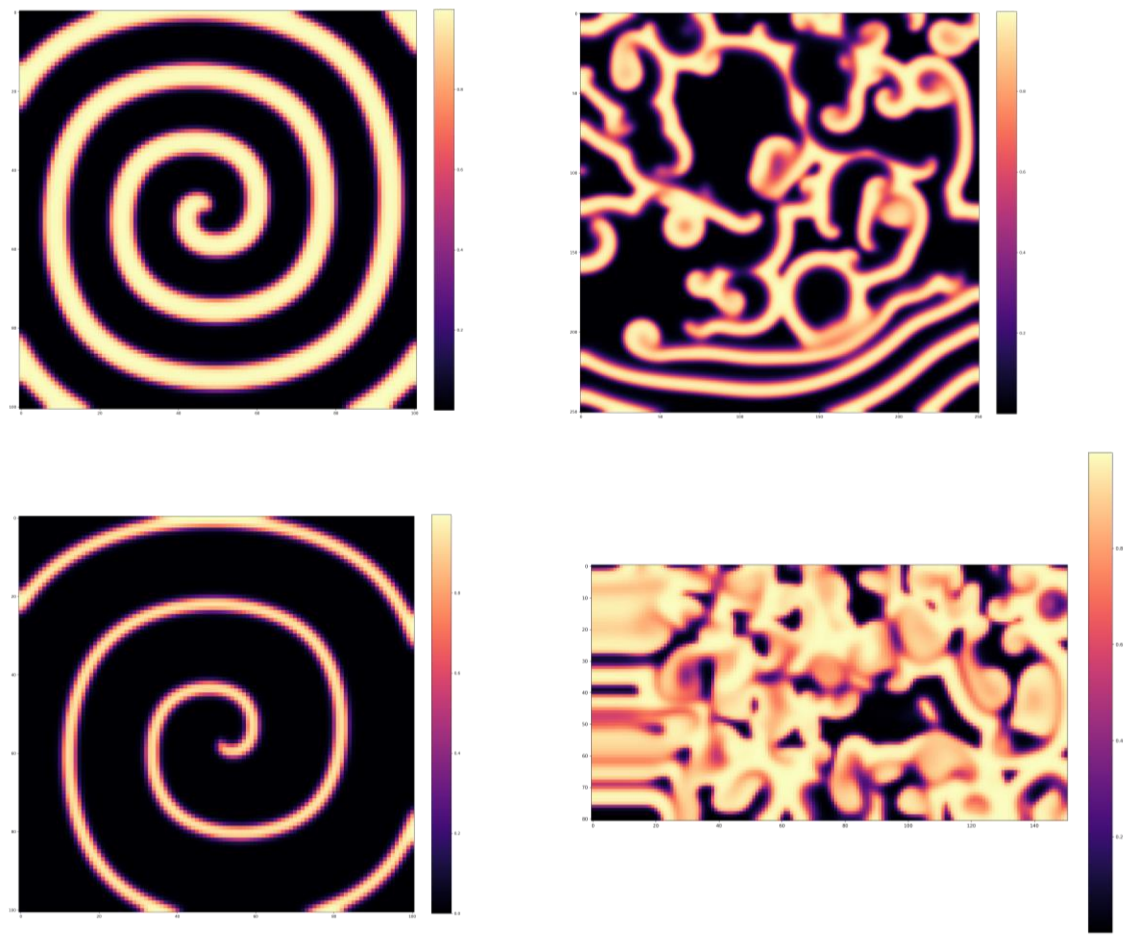
CID - 01707131

High Performance Computing – Coursework Report

This was a really fun coursework project! Admittedly, I did spend an outrageous amount of time pursuing a cleaner matrix solution (found 2 actually!) but it was the for-loop implementation that won out in the end – the short answer is because it is so dang easy to spread the work over threads with OpenMP! [Code available on https://github.com/raihaan123]

1) Solutions to the three test cases evaluated at T=100s with a timestep dt=1e-3s:



2) I chose to parallelize the code with OpenMP for a few reasons – most importantly, the code was already designed to work efficiently in serial and extracting performance from work sharing over a shared domain of nodes is incredibly easy. Knowing that temporal parallelization was impossible, spatial parallelization is where performance gains are expected, and separating chunks of the domain to different processes entirely is unintuitive given the shared boundaries. The messaging and overhead would be more expensive and less intuitive than a well-designed threaded algorithm. OpenMP also has a simple API which makes it easier to parallelize existing for-loops and other parallel sections. Parallelized sections were also chosen based on the relative performance gained by adding them, so I *did not* spam #pragma's everywhere!

3) Plot is in ‘CPP/src/runtime_plot.ipynb’ - almost finished but unfinished due to time-pressure ahaha and a high current CPU usage/inconsistent timings

4) Ideally the runtimes increase linearly with the excess threads spawned (beyond the optimal 16 threads) - threads are spawned in linear time whilst getting diminishing returns for the actual Laplacian evaluations. With 64 threads spawned, however, a significantly larger jump in runtime is observed. The computer I used for testing was Typhoon, and it is known it has a total of 48 hardware cores. To go beyond 48 threads, logical threads would be spawned which severely downgrade performance – dedicated hardware connections are replaced by logical equivalents for thread communication and data storage.

5) Several optimizations were made after the serial code was tested in parallel – most of these were driven by the results from profiler studies and focusing on bottlenecks – and others were made from better understanding the OpenMP paradigm.
    a. Added the –O3 optimization flag for the compiler
    b. Reordered the for-loops sweeping the spatial domain to minimize O(n^3) operations and make use of cache locality
    c. Replaced division arithmetic with multiplication + moved outward in the solver loop
    d. Minimized functions and temporary variables – made use of pointers and references
    e. Removed unnecessary verbosity to shell – good for debugging but expensive!
    f. Timed the various for-loops in the spatial domain with and without parallelization and only applied the directives where necessary

- Of these, (a) (b) and (e) I found the most effective - (a) works the best by ‘unrolling’ inner loops to an expanded state, consuming more memory but maximizing performance – for our hardware, memory does not pose a limitation. Also, no undefined behaviors were relied upon for the code to work, so I had full confidence in –O3's ability to not produce bugs. For this problem, it would reduce runtimes between ~0.5s to 1s.

6) LAPACK was ruled out quite early on since the PDE system was clearly non-linear and would not work with the linear system solver (Ax=B) - without f1 or f2, it was totally an option using ScaLAPACK for parallelization! Initially, I was using BLAS with a symmetric banded shift matrix approach to solving the system (details of which are in the Python/solution.ipynb notebook). While it did work well in the serial case, it did not outperform the for-loop implementation. There was also a case of false sharing from within parallel threads passing references of submatrices of the shift matrix to BLAS, resulting in even longer execution times compared to the serial counterpart. Additionally, for a second, more parallel-friendly technique I developed, I found a

custom function would perform 2x2 matrix multiplication faster than the equivalent dgemm_ method.