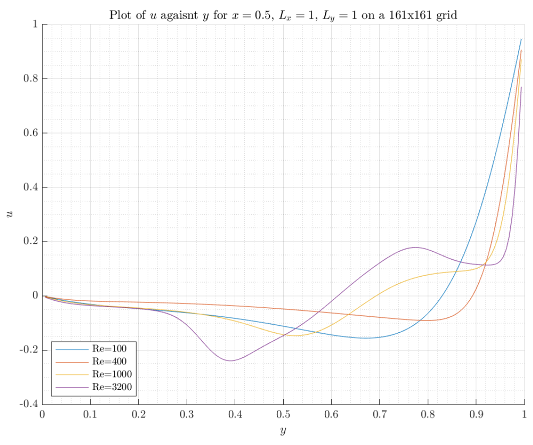
Imperial College London
Department of Aeronautics
Third Year HPC Coursework Report 2020

High Performance Computing Coursework

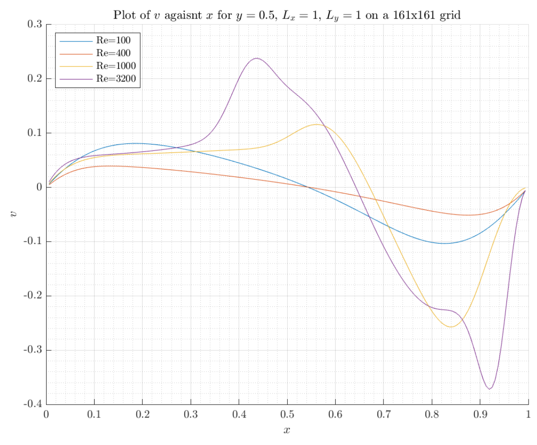
Done by:
Chai Jun, Sean
CID: 01327446

1 Result of running Lid Driven Cavity Solver

1.1 Velocity Plots



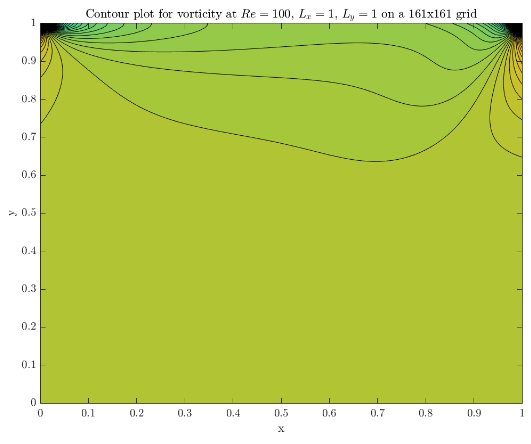
(a) Horizontal velocity u against y along $x = 0.5$



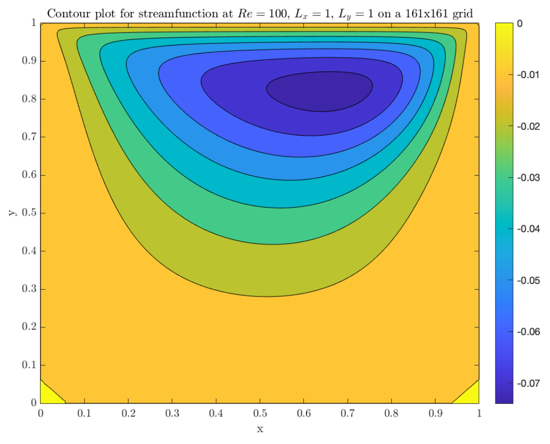
(b) Vertical velocity v against x along $y = 0.5$

Figure 1: Plot of velocities of the steady state solution for Reynolds numbers of 100, 400, 1000 and 3200 using a 161×161 grid and $L_x = L_y = 1$.

1.2 Vorticity and Streamfunction



(a) Vorticity contour plot



(b) Streamfunction contour plot

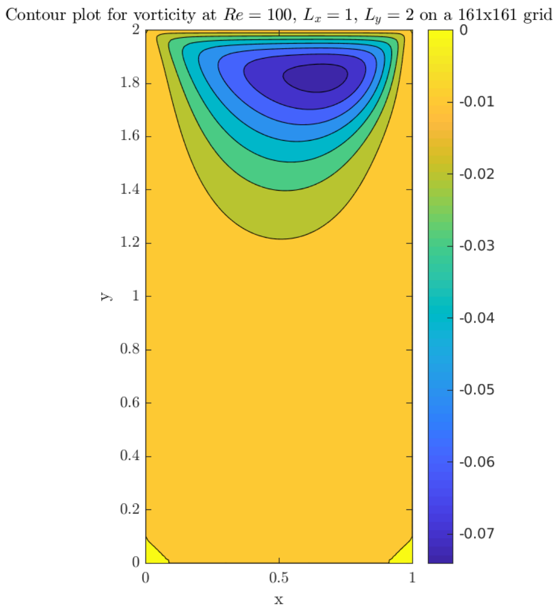
Figure 2: Contour plots of vorticity and streamfunction at $Re = 100$ on a 161×161 grid and $L_x = L_y = 1$.

1.3 Minimum Streamfunction

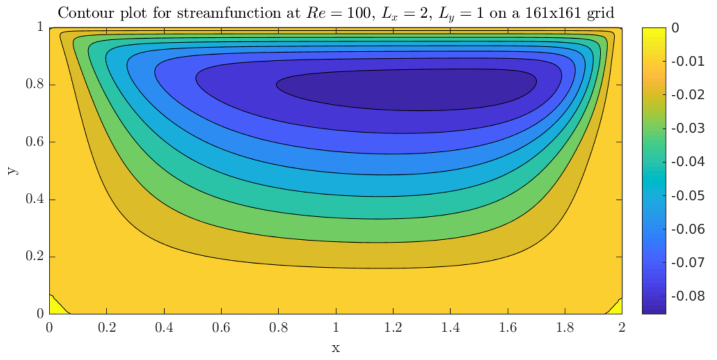
Reynolds Number	100	400	1000	3200
x-coordinate	0.6562	0.8250	0.6875	0.6500
y-coordinate	0.8250	0.8812	0.6875	0.6188

Table 1: Table of x-y coordinates of the streamfunction minimum at the different Reynolds numbers for a 161×161 grid and $L_x = L_y = 1$.

1.4 Streamfunction Plots for Varying L_x and L_y



(a) $L_x = 1$, $L_y = 2$



(b) $L_x = 2$, $L_y = 1$

Figure 3: Contour plots of streamfunction at $Re = 100$ on a 161×161 grid.

2 Discussion of Parallel Code

For the code, there are 4 main steps as outlined in the brief. For the the calculation of the vorticity boundary conditions, it was chosen not to parallelise this segment of the code as the additional time penalty required for communication outweighed the performance gain in distributing the task among processes, since only simple calculations on the edges of the matrix are required in this step.

For the second step, the calculation of the interior vorticity at time t, the calculation was distributed among the processes present. The entire inner matrix was segmented out into chunks, whereby the size of each chunk is optimised such that the array is evenly distributed (no overloading of a single process). The index location of these chunks (i,j coordinates of the matrix) are then distributed to the

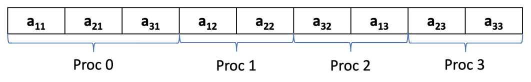


Figure 4: Example partitioning of column-major storage matrix among 4 processes.

individual processes to be solved. Since the size of the matrix remains constant throughout the solve, these coordinates only need to be calculated once. An example distribution of the process is visualised in Figure 4. The same distribution was used for the third step as well.

Additionally, the chunks are allocated in the same format as the storage format, in our case column major format. This allowed for faster memory access due to the reduced need of traversing through the array to reach the memory address. Also, as the same chunk of the array is allocated to the process each time, hence allowing for memory caching which results in faster memory access.

For the fourth step, ScaLAPACK was used to solve the system of linear equations (Ax=b). The laplacian matrix, A, was stored in a banded matrix storage format to save space. Additionally, this A matrix was prefactored via LU factorisation so as to speed up subsequent solves. The solve step was then done in parallel via ScaLAPACK.

One key decision in implementing the code was to swap around the steps of the algorithm. Since the calculation of the inner vorticity (step 2) and the boundary vorticity (step 1) are independent of one another, we can swap the steps around. By calculating the interior vorticity before the boundary, it allowed for easier parallelising of code. This is because the individual calculated segments can be summed together, as all values with index not within the range of the particular process are initialised as 0, then subsequently have the boundary condition applied. This compared to accessing the inner matrix and updating it individually, was found to be a lot more convenient and efficient.

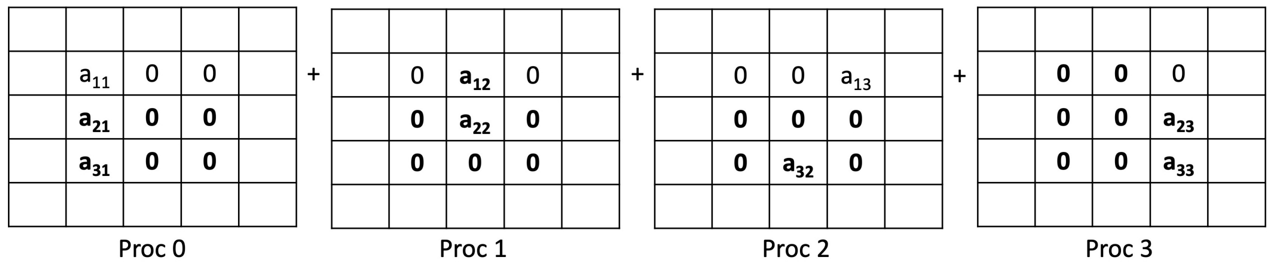


Figure 5: Example of “overlaying” matrices to get total value

Another method of optimising the serial code was to separate it from the parallel one. This reduced the need for the serial code to send and receive and removed the steps of determining the local chunk of (i,j) coordinates.

From Figure 6, it is clear that parallelising the code has led to significant improvements as compared to the serial one. However, one point to note is that while moving from 4 processes to 9 processes improved the performance, beyond this, performance begins to decrease. This can be attributed to the fact that as more processes are created, more communication is required between them. In the latter case, the performance gained from splitting the work between more processes is outweighed by the performance loss due to more communication.

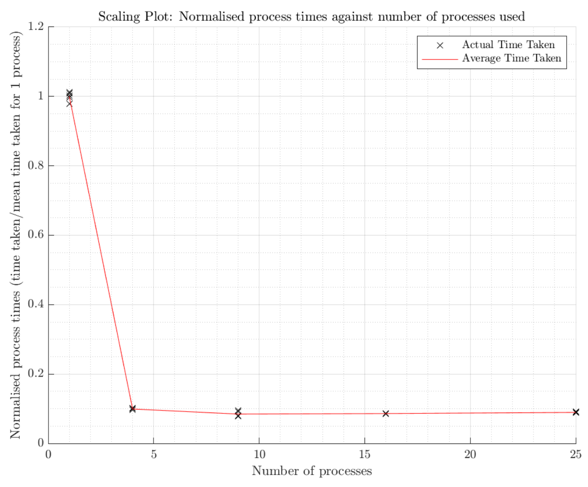


Figure 6: Scale plot of time taken against number of processes