Imperial College London

Department of Aeronautics

High Performance Computing Coursework

Student:	Cristian Vegas Medina
CID:	01874108
Date:	23/06/24
Department:	Department of Aeronautics
Course:	MEng Aeronautical Engineering
Module:	High Performance Computing
Academic Year:	2023/2024

1 Unit Tests

Unit tests have been created for the LidDrivenCavity and SolverCG classes to provide evidence that the code continues to execute correctly when tackling the following tasks. The Boost Test framework was used for each unit test, as it simplifies the process of creating tests. The LidDrivenCavity unit test computes the analytical vorticity and compares the value to the ones obtained by the LidDrivenCavity class through `BOOST_CHECK_CLOSE()`, ensuring they are equal to a tolerance of 1e-5. The same is done with SolverCG, comparing the analytical stream-function to the numerical result.

2 MPI Parallel Code

To parallelise the code, each dimension of the domain was partitioned p ways, creating a total of p^2 sub-domains. Each sub-domain was assigned to an MPI process, thus requiring an MPI initialisation with $P = p^2$ ranks. Other choices of P would result in the code terminating. These partitions were performed through a function which divides N_x and N_y by p, computing the start and end position in the x and y directions of each sub-domain, and sending these to their respective rank. In the case that N_x and N_y were not multiples of p, the remainder was computed and spread one by one along the initial ranks to ensure the work done by each process was as balanced as possible.

Before parallelising the code, the serial code was profiled to better understand which parts of the code were the most time-consuming and where parallelism was required the most. This showed how virtually the whole runtime was spent within solverCG. Thus, parallelising the LidDrivenCavity.cpp code would result in a marginal improvement in overall runtime. Hence, for the sake of simplicity, a simple and slightly in-efficient approach was used to parallelise LidDrivenCavity, where the entire domain was partitioned as described previously, but each process received an array of size equal to the entire domain conserving the local position of the sub-domain as represented below.

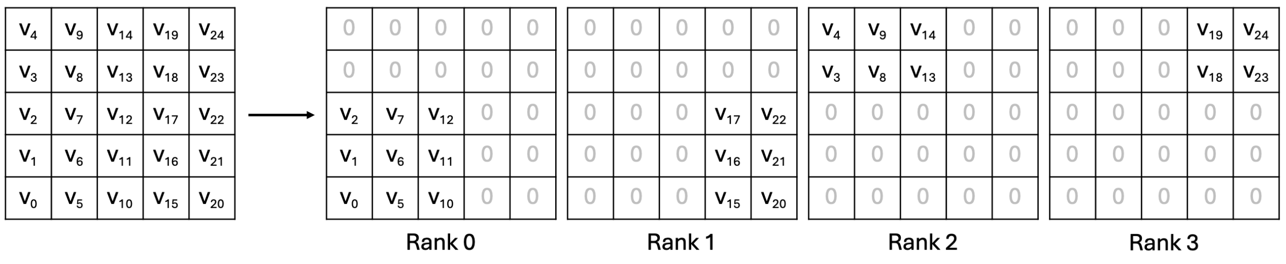
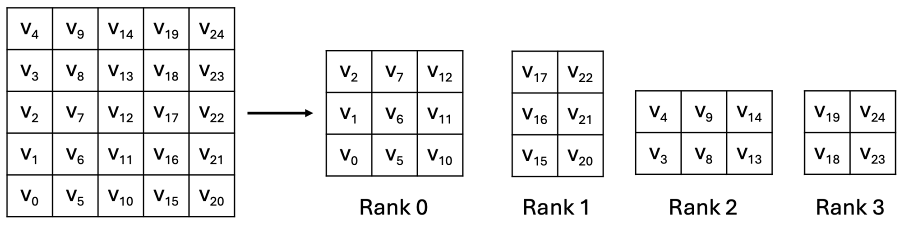


Figure 1: Domain partitions in LidDrivenCavity

The vorticity boundary conditions and the inner vorticity are computed in parallel for each sub-domain, where the stream-function of the entire domain is stored on each process. However, the time-advanced interior vorticity equation requires vorticity values one layer outside of each sub-domain. This is solved by using `MPI_Allreduce()` before the calculation, to combine all the vorticity sub-domains together and share the entire domain to each process. After the third equation, `MPI_Allreduce()` is used to combine all the updated vorticity arrays into the full domain. Finally, the vorticity (tmp) and stream-function (s) arrays of the entire domain are sent into SolverCG.

For SolverCG a more complex but efficient parallelization is performed, where the input v and s arrays of the entire domain are partitioned as described previously, but this time the array sent to each process only contains the sub-domain entries as shown in Figure 2. This leads to a loss of local position, and thus the partition function is modified to provide additional parameters to each process which deliver the required information to perform certain calculations.



v_4 v_9 v_{14} v_{19} v_{24}
v_3 v_8 v_{13} v_{18} v_{23}
v_2 v_7 v_{12} v_{17} v_{22}
v_1 v_6 v_{11} v_{16} v_{21}
v_0 v_5 v_{10} v_{15} v_{20}
Rank 0
Rank 1
Rank 2
Rank 3

Figure 2: Domain Partitions in SolverCG

In addition, as the `ApplyOperator` function requires values one layer outside of the sub-domain, a new function is designed to create (before the `ApplyOperator` function) a new augmented array containing this layer, through a series of `MPI_Send()` and `MPI_Recv()`'s between neighbouring ranks. However, after the `ApplyOperator` function, the arrays for each sub-domain remain within their initial (unlayered) size. `MPI_Allreduce()` is used to add all the alpha and beta values and share the sum to all the processes. The epsilon value is obtained by calculating eps^2 by performing a dot product with cblas and using `MPI_Allreduce()` to add the values and share the sum. After convergence, the stream-function arrays of each rank are combined into the full domain and are fed back to LidDrivenCavity.

The following test cases with increasing number of grid points were used to test the speedup of the code when run in parallel.

Test Case 1:	–Lx 1	–Ly 1	–Nx 25	–Ny 25	–Re 1000	–dt 0.005	–T 50
Test Case 2:	–Lx 1	–Ly 1	–Nx 50	–Ny 50	–Re 1000	–dt 0.005	–T 50
Test Case 3:	–Lx 1	–Ly 1	–Nx 100	–Ny 100	–Re 1000	–dt 0.005	–T 50
Test Case 4:	–Lx 1	–Ly 1	–Nx 200	–Ny 200	–Re 1000	–dt 0.005	–T 50
Test Case 5:	–Lx 1	–Ly 1	–Nx 400	–Ny 400	–Re 1000	–dt 0.001	–T 1

The results of the test cases for up to $P = 16$ ranks are shown in Table 1. The normalised runtimes (relative to serial) against the number of MPI processes used can be visualised in Figure 3. The results show a satisfactory scaling, where the reduction in runtime with increasing number of processes becomes more significant with increasing number of grid points. As expected, it also shows the opposite trend for a small number of grid points, where increasing number of processes results in an increase in runtime, as in this case the time penalty required for communication outweighs the performance gain in distributing the task among processes.

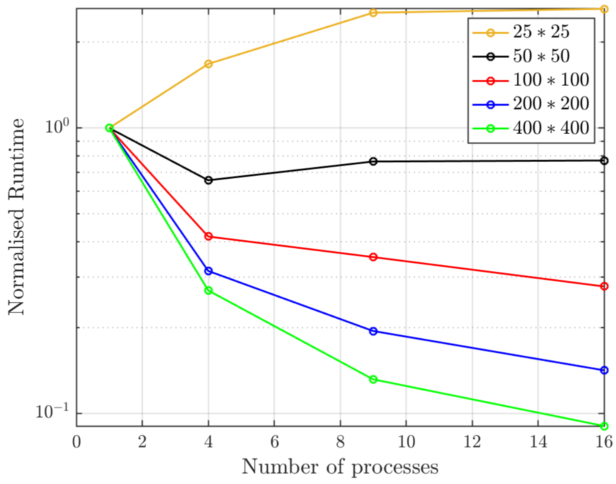


Figure 3: Normalised MPI runtimes plot

Table 1: MPI runtimes with increasing n° of processes

Test Case	$P = 1$	$P = 4$	$P = 9$	$P = 16$
1	3.1 s	5.3 s	7.9 s	8.2 s
2	31.7 s	20.8 s	24.2 s	24.4 s
3	259.2 s	108.0 s	91.5 s	72.2 s
4	2666 s	747.1 s	460.7 s	335.2 s
5	2180 s	588.2 s	287.9 s	197.0 s

3 OpenMP Parrallel Code

Moreover, multi-threading was added to the code with OpenMP. No partitions were applied to the domain for this part as only one MPI process was used. In this case, parallelism was achieved by applying `#pragma omp parallel for` statements before all the for loops to divide the iterations into chunks and distribute them between the threads. Nested for loops contain an additional `collapse(2)` term to parallelise both loops. Only SolverCG for loops were parallelised as only these resulted in a reduced total runtime. In addition, as `cblas_ddot`, `cblas_daxpy` and `cblas_dcopy` operations didn't perform any better with multiple threads, new functions were defined to perform the same array operations using for loops, which were parallelised as stated above.

Test cases 3, 4 and 5 were used to test for the speedup of the code when run in parallel, using up to 16 threads. The normalised runtimes (relative to serial) against the number of threads used are represented in Figure 4, with exact runtimes shown in Table 2. The results showed a similar satisfactory trend to MPI, where the speedup achieved with parallelism increased with increasing number of grid points. However, increasing the number of threads showed a slightly reduced speedup compared to increasing the number of processes, and a significant speedup was only obtained for domains with more than 100 $\times$ 100 grid points.

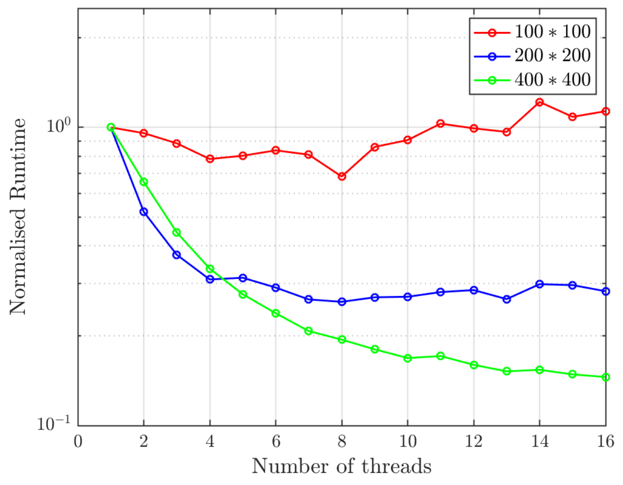


Figure 4: Normalised OpenMP runtimes plot

Table 2: OpenMP runtimes with increasing n° of threads (in seconds)

threads	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
TC3	259.2	248.1	229.3	203.1	208.3	217.1	210.1	177.4	223.0	23.51	266.9	257.4	250.4	315.6	281.4	293.8
TC4	2711	1413	1014	838.8	849.6	787.5	718.5	705.1	730.0	733.3	760.1	772.0	720.9,	808.3	802.8	765.0
TC5	1874	1230	833.2	629.7	516.7	446.1	388.8	363.9	337.6	315.7	320.5	299.5	284.8	288.2	278.1	272.6

4 Profiling and Optimisations

The code was profiled using one MPI process and one OpenMP thread with Test Case 4 parameters. The results are shown in the first row of Table 3, which shows that the most time-consuming parts of the code are the BLAS, ApplyOperator and Precondition functions within the while loop in solverCG, as they are used for a large number of iterations.

The first optimisation of the code was done by selecting the most efficient optimisation flag, which was initially set to -O2. However, -O3 showed an improved performance with almost a 10% decrease in runtime and was therefore selected.

The next optimisation was done by removing an alpha and a beta `cblas_ddot` calculations as they were identical to other `cblas_ddot` calls in the while loop. This reduced the time spent within `cblas_ddot` by 573 s, which is an additional 21% reduction in total runtime.

As the while loop performs +500 iterations at each time step (for domains with a large number of grid points), the epsilon calculation at each iteration using `cblas_dnrm2` becomes unnecessary as it isn't used for the linear solver calculations but rather for the convergence criteria. Hence, the epsilon calculation was calculated once every 4 iterations instead, reducing the overall runtime by 272 s. This change requires in the worst-case scenario 3 more iterations to converge, which virtually doesn't affect the runtime for domains with a large number of grid points. However, for domains with a small number of grid points it could be penalising and hence not convenient.

Overall the optimisations performed reduced the total runtime by 40% from 2713 s to 1638 s.

Table 3: Comparison of profiler results for each optimisation

Total	cblas_ddot	cblas_dnrm2	cblas_daxpy	ApplyOperator	Precondition	cblas_dcopy	Other
2713 s	1152 s	358 s	278 s	306 s	311 s	163 s	143 s
2487 s	1149 s	360 s	284 s	203 s	136 s	171 s	183 s
1910 s	576 s	370 s	272 s	189 s	154 s	168 s	180 s
1638 s	578 s	98 s	279 s	208 s	143 s	160 s	172 s