Imperial College London

Lid Driven Cavity Problem Simulation Report

High Performance Computing Coursework

Supervisor: Dr Chris Cantwell
Department: Department of Aeronautics
Course: 96021
Author: Jason Zhao
Date: 26/08/2024

Department of Aeronautics
South Kensington Campus
Imperial College London
London SW7 2AZ
U.K.

Contents

1 Unit Tests

The unit tests are done using Boost Test framework to simplify the process of creating tests.

The Boost unit tests are designed to be able to be implemented for the MPI and OpenMP test case. 3 unit tests are created with one for SolverCG class, and two for LidDrivenCavity class.

For the SolverCG class, the Boost unit test named *Solver* is implemented on member function *SolverCG::Solve(double* b, double* x)*, because it is the entry point of the SolverCG.cpp, and it either calls the other member functions (ImposeBC, Precondition, ApplyOperator...) in SolverCG.cpp or requires some of them as a prerequisite to run correctly. Thus testing *SolverCG::Solve(double* b, double* x)* can make sure the SolverCG class is running correctly.

The *Solver* Boost test take vorticity v using the provided sinusoidal test case and streamfuction s and compare their output values from *SolverCG::Solve(double* b, double* x)* to the analytical values of v and s.

For the LidDrivenCavity class, the Boost unit test named *Advance* is the major test, implemented on member function *LidDrivenCavity::Advance(double* v,double* s, SolverCG* cg, double dx, double dy)*. It is the main part through the iteration in member function *LidDrivenCavity::Integrate()*. The test *Advance* takes the initial value of v and s, implemented with 1 time step $NSteps = 1$ and compare it with the output using the given base code.

The Boost test named *subIDXglobal* is used to test the member function *subIDXglobal(int iSub, int rank)* which convert the local index of the point in the current sub process to the global index. It is tested for each point of sample array double *v[Npts]*.

2 MPI

2.1 MPI Approach

The MPI is mainly implemented on SolverCG.cpp and LidDrivenCavity.cpp files.

The grid points are divided for each processes in the *setSubCore()* member function, using *MPI_Cart* with Cartesian topology is used for easy referencing.

A macro to converting local coordinates to local index, and a function subIDXglobal(int iSub, int rank) converting local index to global index are defined.

MPI_Bcast() is used to broadcast the global matrix from Process 0 to all the other processes, and convert the global index to local index.

MPI_Allreduce() function is also used to gather and sum up the sum values from each sub processes to do the cblas_ddot() operations.

A member function is first defined to communicate between neighbouring sub processes, which is later inline for optimisations.

For the algorithms equations which have i±1 or j±1, it is necessary to consider whether the index exceeds the boundary of the sub process core. Therefore, the following sectional codes are created:

The code determining the location of the sub process core. Whether it is at the edges, corners, or interior. Then assign the unique boundaries of the index (beginning and ending of i and j) for each sub processes. Theses boundaries are used for formulas excluding the global boundaries e.g. (int i=1; i<Nx – 1; i ++).

The coding exchanging boundaries data with neighbouring processes in 4 directions, using MPI_send() and MPI_recv(). This code sorts out the sub processes in same way as previous part, to not send data out of global boundaries.

The code determining whether the local point with coordinates i±1 or j±1 is out of the boundary of the local matrix. If it is out of the local boundary, this data will be set equal to the exchanging data in 2. Otherwise, it will stay the same value.

For the algorithms calculating the boundaries, there is no need to exchange data between sub processes. Note that although Nx-2 and Ny-2 exist, there is no need to consider them for NP up to 64, because the last sub matrix of the row and column always has more than 2 grid points on that side.

2.2 Workload Balancing

To balance the workload between each sub processes, I let the Process 0 to gather all the local data and do the assemble job, because according to the definition in the *setSubCore()* member function, the processes at the top or right in the Cartesian topology may need to include the remainder grid points, thus having more grid points to compute. For example, in the case of NP = 4, Process 0 has the least amount of grid shown by figure 3, which has the least workload. For other NPs, Process is one of the processes with least grids as well.
Future improvements can be made for this code is to use master process to assign the task [1] to the process that are free, known as Master-Slave model.

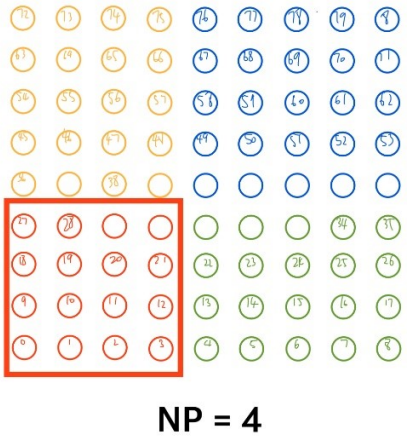

Figure 1. Span-wise Maximum Shear Stress Distribution

2.3 Scaling Plot of MPI

Table 1 in the appendix records the time elapsed of running the code with number of process = 1, 4, 9, and 16, and the grid points $9 \times 9 = 81$. Figure 2 shows the scaling plot of it.

However, it should be noticed that, compared to the serial case, the running time is longer. Besides, the running time increases with the number of processes.

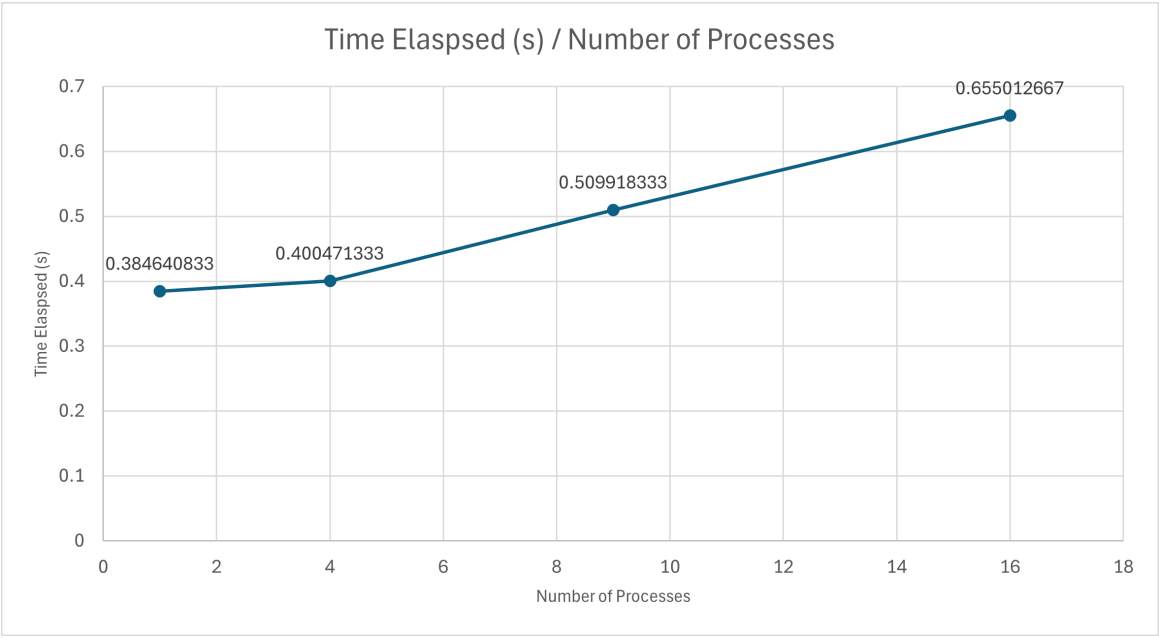

Figure 2. Running Time of MPI with Different Number of Processes

The major reason is that the grid size needs to be calculated is too small to show the advantages of MPI, as the reduction of number of grid point calculated by a single process is negligibly small.

In comparison, the time taken by communications between each sub process takes longer time. The more processes there are, the more frequent the processes need to communicate.

In additions, the synchronising process required by functions like MPI_Allreduce() also takes up longer time.

Moreover, MPI also creates iterated variables to each sub process as each processes cannot access the variables from the other sub processes directly. This takes up more computer resources in total.

Apart from that, the unbalanced workload also makes some of processes waiting for the slowest process.

3 OpenMP

3.1 Scaling Plot of OpenMP

The figure 3 shows the scaling plot of elapsed time versus number of threads from 1 to 16 using OpenMP. Table 3 in the appendix shows the experimental values.

It should be noticed that there is a dramatic increase since number of threads = 13. One of the possible reason of it is that when number of threads reaches 13, it also reaches the barrier of bandwidth of the memory [2].

Another finding is that when number of threads = 8, there is a slight increase in running time, it then drop down when number increases to 9. This might due to the do-while loop in SolverCG.cpp. The small error in t+dt when parallelizing the code (mentioned in the Ed Discusion), accidentally increases the number of iterations needed to converge when the data is equal to 8, as the number required to converge increased at this case.

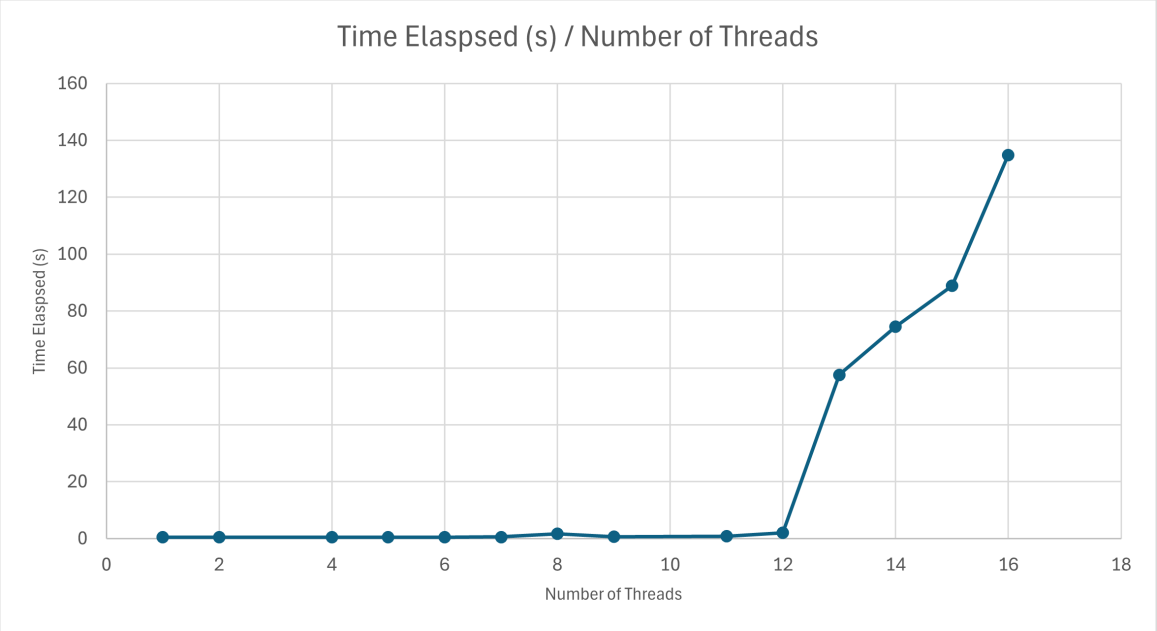

Figure 3. Running Time of OpenMP with Different Number of Processes

3.2 OpenMP Approach

The OpenMP is primarily done for the for loops, with the number of threads specified using the environmental variable OMP_NUM_THREADS defined in the Makefile.

omp_get_thread_num() and omp_get_num_threads() functions are used to visualise the total number of threads and the sequence of current thread for the probe when developing the code.

4 Optimisations

4.1 Analysis of the Time Consuming Parts

Environmental variable of version control of compiler used by mpicxx is tried, but source code is still not visible in the profiler. Hence the Call Tree, the Functions and the Timeline are refereed for optimisations.

Figure 4 is the Call Tree before the optimisation, showing the time consuming and their percentages. From the figure, it can be concluded that *LidDrivenCavity::Advance()* is the most time consuming part of LidDrivenCavity class which takes 99%. Surprisingly, the algorithms within *LidDrivenCavity::Advance()* only takes 2%, while the most time consuming part is the invoked member function *SolverCG::Solve(double* b, double* x)*, which is 64.755 s, taking up 97% of the running time .

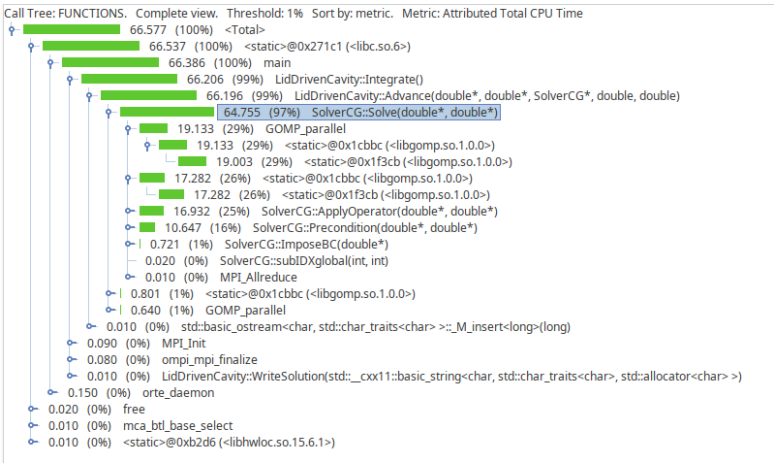

Figure 4. Call Tree Before Optimisation

Within *SolverCG::Solve(double* b, double* x)*, there are 4 time consuming parts: member functions Precondition() and ApplyOperator(), which takes 16% and 26%, and the GOMP_parallel and <static>@0x1cbbc parts, which takes 29% and 26%. Without further information from Source view, it is difficult to determine which parts that these two referring to. However, it can be inferred that GOMP_parallel refers to the OpenMP operations.

This might due to the reason that OpenMP needs to take up resources to assign tasks even only one thread is used. In additions, OpenMP still needs to create potential threads which takes up the time. Besides, the for loop used by OpenMP has a hidden barrier to synchronize [3], which also leads to delay.

Another reason that greatly contributes to the sever delay in the *SolverCG::Solve(double* b, double* x)* is the do-while loop, which iterative calls the functions of Precondition() and ApplyOperator(), and the parallel parts using MPI and OpenMP.

4.2 Inline Simple functions

For the Optimisations, firstly, simple member functions like *SolverCG::ImposeBC()*, the member function defined earlier for communication with neighbouring processes in the Cartesian topology, and*Precondition()* (which is iterated heavily in the do-while loop), are inserted directly into the code section calling them. This cut down the time calling them, which requires stack operations [3].

4.3 Lift loop-independent expression

Another optimisation made is by checking the code and moving some of the expressions that is constant within the for-loop to outside the for loop, in order to avoid unwanted iterative calculations.

4.4 Making Each Processes more Independent

The code was first parallelised using MPI in different sections, with local data assembled at process 0 and distributed to other processes for the next step for easy debugging. This takes massive time by exchanging data, converting data from global to sub grid and vice versa. Thus code is optimised by reducing the reducing the frequency of this unwanted MPI communication.

4.5 Optimised Results

The optimised results has total running time 59.382, which is 10.8% faster than the code before optimisations. The major contributions are from the member function *SolverCG::Solve(double* b, double* x)*, which improved from 64.755 s to 57.400 with a percentage improvement of 11.3%.

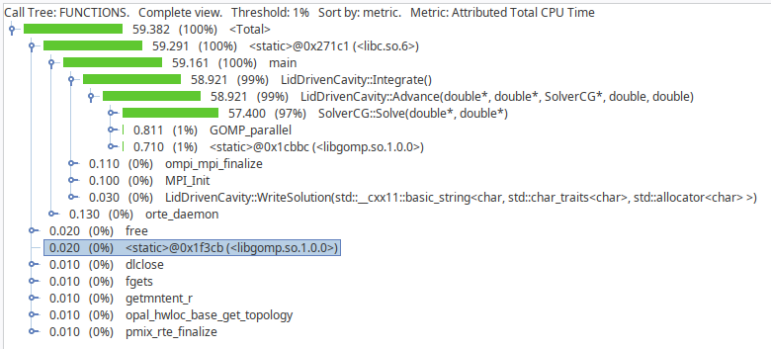

Figure 5. Call Tree After Optimisation

References

[1] GeeksforGeeks. Parallel Algorithm Models in Parallel Computing;. Accessed: 2023-08-26. https://www.geeksforgeeks.org/parallel-algorithm-models-in-parallel-computing/.

[2] ARCHER. OpenMP Programming; 2019. Accessed: 2023-08-25. http://www.archer.ac.uk/training/course-material/2019/07/sgl-node-imp/L06-openmp.pdf.

[3] Cantwell C. Imperial College London: Aero Department, Imperial College London; 2023/2024.

Appendix

Number of Processes	Time Elapsed / s, Test 1	Time Elapsed / s, Test 2	Time Elapsed / s, Test 3	Time Elapsed / s, Test 4	Time Elapsed / s, Test 5	Time Elapsed / s, Test 6	Time Elapsed / s, Average
1	4.00E-01	4.00E-01	3.51E-01	3.94E-01	4.07E-01	3.54E-01	0.384640833
4	3.94E-01	3.75E-01	3.79E-01	4.12E-01	4.31E-01	4.12E-01	0.400471333
9	4.67E-01	5.44E-01	5.43E-01	5.24E-01	5.10E-01	4.70E-01	0.509918333
16	6.23E-01	6.30E-01	7.49E-01	6.92E-01	6.32E-01	6.04E-01	0.655012667

Table 1. MPI Running Time

Table 2. Time Elapsed for Different Number of Threads

Number of Threads	Time Elapsed / s, Test 1	Time Elapsed / s, Test 2	Time Elapsed / s, Test 3	Time Elapsed / s, Test 4	Time Elapsed / s, Test 5	Time Elapsed / s, Test 6	Time Elapsed / s, Average
1	4.16E-01	4.18E-01	4.47E-01	4.17E-01	4.21E-01	4.44E-01	0.427294833
2	4.49E-01	4.73E-01	4.58E-01	4.53E-01	4.47E-01	4.52E-01	0.4552945
4	4.50E-01	4.48E-01	4.53E-01	4.39E-01	4.42E-01	4.80E-01	0.451921333
5	4.87E-01	5.00E-01	4.99E-01	5.01E-01	4.84E-01	4.77E-01	0.491010833
6	5.14E-01	5.34E-01	5.34E-01	5.33E-01	5.21E-01	5.19E-01	0.525879333
7	5.70E-01	5.47E-01	5.67E-01	6.33E-01	5.21E-01	5.65E-01	0.5763284
8	1.55765	2.04484	1.73636	1.81188	1.42303	1.66983	1.707265
9	0.665177	0.718511	0.684797	0.714285	0.686732	0.650677	0.6866965
11	0.862833	0.870693	0.892061	0.845645	0.846126	0.803651	0.8535015
12	3.34018	1.98234	1.72529	1.15933	2.39508	1.5921	2.032386667
13	57.2564	57.9364	-	-	-	-	57.5964
14	77.3044	71.5776	-	-	-	-	74.441
15	86.3934	91.4144	-	-	-	-	88.9039
16	120.531	149.026	-	-	-	-	134.7785

Table 3. OpenMP Running Time