AERO70011 - High-Performance Computing

Lid-Driven Cavity Simulation
Coursework assignment - Report

Student: Kee Kwan Liu
CID: 01847234
Date March 19, 2024

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

1 Introduction

The lid-driven cavity problem is a classical benchmark in computational fluid dynamics that provides valuable insights into the behaviour and efficiency of the CFD algorithm. In this problem, a square cavity is filled with fluid, and the motion of the fluid is driven by the continuous movement of the top lid, while the other boundaries remain stationary.

The objective and aims of developing the parallelised numerical solver are to demonstrate the effects and benefits of applying various parallelisation techniques to simulate the fluid behavior in this benchmark problem. Parallelisation techniques can be roughly divided into two classes: distributed memory approach and shared memory approach. These two different approaches to parallelisation can be realised via the usage of the MPI and OpenMP libraries.

Additionally, a class of supporting tools, such as build systems, documentation software, version control software, and profilers, is utilised to support the development of this high-performance tool.

2 Unit tests

Unit tests are software that validate the correctness of a class, function, or the entire software. This is especially useful when building new features or optimising the code, as it prevents accidentally breaking the code or producing unpredictable behavior without understanding the root cause and being unable to revert to a previous working version.

	Test Entry Point	LidDrivenCavity Class	SolverCG Class
test (folder)	TestMain.cpp	TestLidDrivenCavity.cpp	TestSolverCG.cpp

These test files are stored in the 'test' folder, which contains three files: 'TestMain.cpp', 'TestLidDrivenCavity.cpp', and 'TestSolverCG.cpp'. 'TestLidDrivenCavity.cpp' contains test cases for the LidDrivenCavity class, while 'TestSolverCG.cpp' contains test cases for the SolverCG class. Due to a considerable number of tests, it is deemed more appropriate to have a main test as entry point that combines all the test files into a single test executable.

2.1 LidDrivenCavity Class

1. DomainCheck	2. ReynoldsCheck	3. InitialConditionCheck	4. BoundaryconditionCheck
5. FinalResultCheck	6. FinalResult2Check	7. FinalResult3Check	

The LidDrivenCavity class file encompasses various scenarios aimed at thoroughly validating the functionality and accuracy of the LidDrivenCavity class. Out of the seven tests, the first two focus on the program arguments and whether the fluid solver configuration is set up properly. DomainCheck verifies whether the solver has the correct domain size, while ReynoldsCheck verifies whether the fluid properties (viscosity and heat transfer rate) are set up properly.

For the third and fourth tests, these check whether the fluid is at rest and whether boundary conditions are applied, accounting for the continuous movement of the top lid and the no-slip condition. In the event where the initial u and v velocity are not zero or the u velocity at the top lid is not U (the top velocity), an error message is returned and the test has failed.

The fifth to seventh tests compare the final simulation results of the parallel solver with those from the baseline serial solver. Due to floating-point issues and different computation approaches, an exact floating-point representation is not expected after millions of computations. Thus, if the final value falls within a certain tolerance limit (absolute tolerance of 1e-8 and relative tolerance of 0.1%), it is deemed correct. The fifth, sixth, and seventh test cases evaluate a 9x9, 49x49, and 201x201 mesh, respectively. These three tests are arguably the most important throughout the process of profiler code optimisation and updating and optimising certain code features.

2.2 SolverCG Class

1. SolveArbitraryCheck	2. SolveRealCheck

The SolveArbitraryCheck test case verifies whether the conjugate gradient solver outputs the correct result for an arbitrary vorticity and stream function matrix. The vorticity and stream function consist of arbitrary values with no real physical meaning. The SolveRealCheck test case evaluates the output of the CG solver with real vorticity and stream function values extracted from the real simulation and stored in a test file.

3 MPI Parallelisation

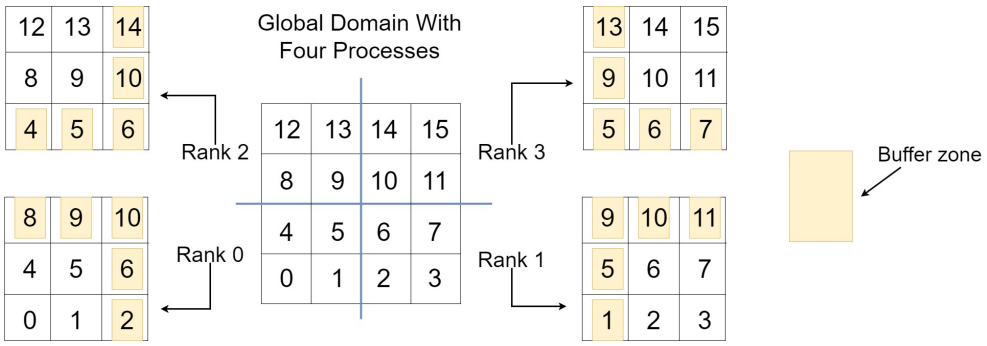
The Message Passing Interface (MPI) is a powerful technique that enables simultaneous computation of sections of the same problem in a distributed memory fashion. Each process exchanges information with others for a more efficient program.

3.1 Domain Decomposition

For optimal computational efficiency, our goal is to evenly distribute the workload among p^2 processes. The following code segment illustrates how a global computational domain is partitioned into smaller local domains in the x-direction (similar approach for y-direction). In both the x and y directions, the remainder r of Nx_local/Ny_local grid points divided by p is added to the first r processes in the respective dimension. Each process is assigned their respective local domain depending on their coordinate in the virtual Cartesian topology.

```
if(rank % Nprocs_sqrt < Nx_remainder){
    Nx_local = Nx / Nprocs_sqrt + 1;
    offset_x = Nx_local * (rank % Nprocs_sqrt);}
else{
    Nx_local = Nx / Nprocs_sqrt;
    offset_x = Nx_local * (rank % Nprocs_sqrt) + Nx_remainder;}
```

A buffer zone is then added to the local domain, which accounts for the information in other domain. E.g. at the top right corner, Nx_local and Ny_local will have an increment of one as it has a bottom neighbour and a left neighbour. In the case of 9 or 16 processes, a rank may have four ranks neighbouring it, Nx_local and Ny_local will have an increment of two.



3.2 MPI Parallelisation Strategy

Multiple equations are computed throughout the program. While the localised quantity (e.g., local stream function) can be computed with localised resources (e.g., local vorticity before the conjugate gradient solver), inter-domain communication is strategically placed to ensure that each local domain has enough information from other sub-domains to function properly.

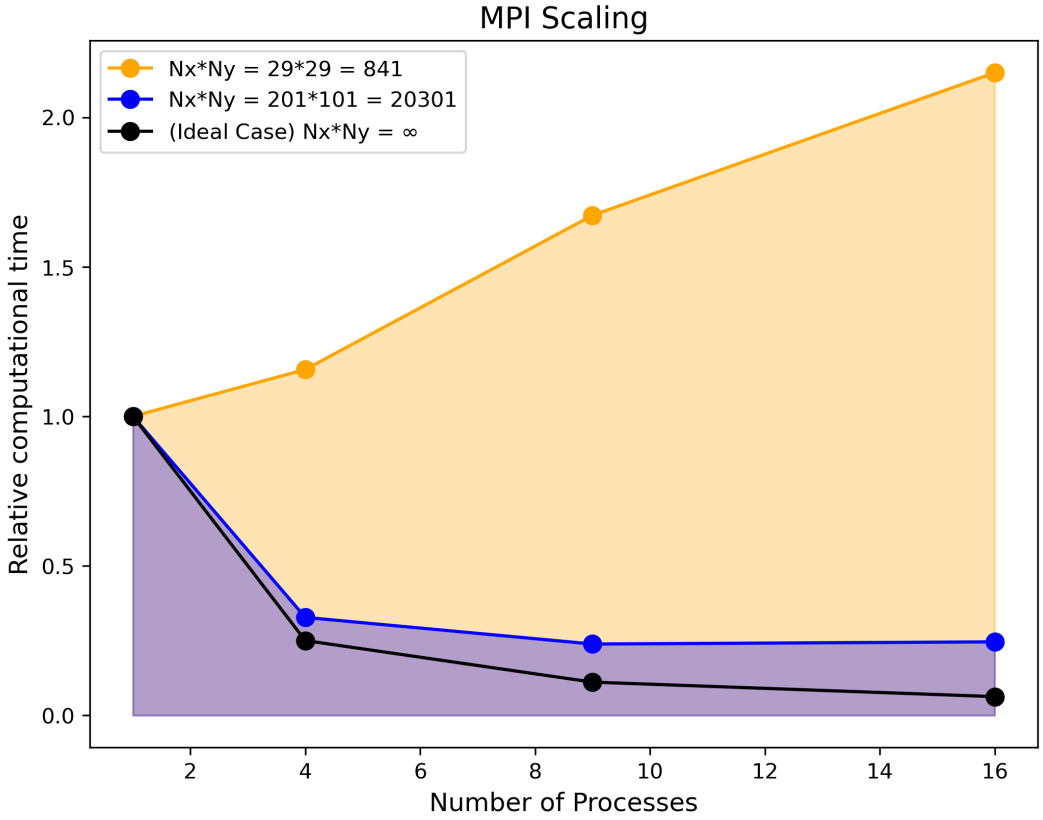
In the code, inter-domain communication is realised through the private function DomainInterCommunication(double* A_local). For each process, it sends the 'edge' of its local domain excluding the buffer zone to its corresponding neighbor if it is not a boundary. At the same time, it will receive the neighbor's 'edge' information and store it in the buffer zone. This provides enough information for each process to calculate the next equation.

When enforcing boundary conditions for both vorticity in the LidDrivenCavity class and preconditioning in the SolverCG class, it is crucial to check the coordinates of each process in the virtual Cartesian topology and to avoid incorrectly imposing boundary conditions on non-boundary values.

Similar to imposing boundary conditions, when computing interior values such as in the interior vorticity step in the LidDrivenCavity class and applying the Laplace operator in the conjugate gradient solver, it is important to account for the buffer zone. In areas where it is not a boundary, these elements should also be computed.

3.3 MPI Scaling

MPI scaling plots provide valuable insights into the performance and scalability of parallel applications. An ideal scaling plot would should a significant reduction of computational time to justify the increased amount of computational resources and to save time.



The above scaling plot is generated by varying the number of processes (with one thread) for the same fluid solver configuration. Ideally, with an extremely fine mesh where the cost of communication is negligible compared to the cost of the solver integration, the relative computational time would be approximately 1/processes. However, this experiment is a medium-sized problem where communication cost is non-negligible.

In this 201x101 case, there is a 67.3% reduction in computational time from 1 process to 4 processes, and a further 8.9% reduction going from 4 to 9 processes. However, the computational time remains approximately the same for 9 and 16 processes despite using more computational resources. This is explained by the increased cost of inter-process communication, which offsets the benefits of having less computation in each process.

A big problem requires a big solution; the use of Message Passing Interface is not justified for small cases like 29x29. Increasing the number of processes would lead to increased computational time when the solver integration cost is low. In such scenarios, a serial numerical solver would be a more desirable solution.

In conclusion, the scaling plot demonstrates good scaling, especially when the number of nodes exceeds 10,000. Overall, the trend from 1 to 9 processes resembles that of the ideal case. Therefore, it justifies the usage of more computational resources.

4 OpenMP Parallelisation

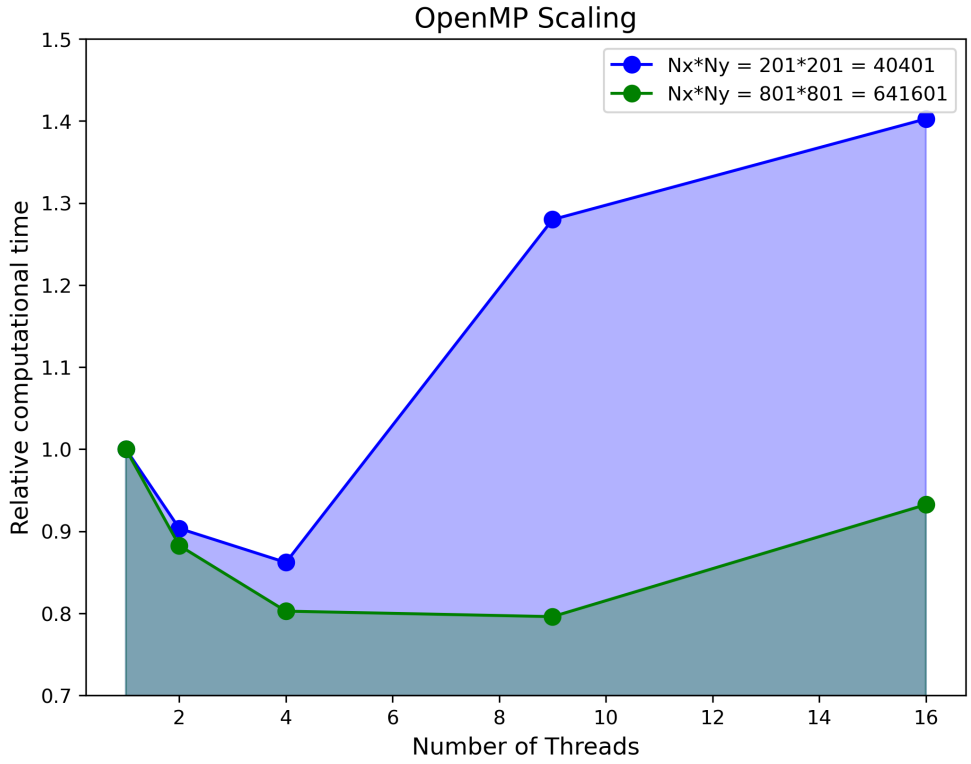
Shared-memory parallelisation is another approach to achieving parallel programming. In this approach, multiple threads are spawned within a single process. Different threads can work concurrently, and thus, they are capable of breaking down problems into smaller tasks.

4.1 OpenMP Parallelisation strategy

A holistic view of the parallelisation strategy would be to parallelise loop-based computations in the conjugate gradient solver that are most frequently used. From the profiler (which will be discussed in more detail later), the conjugate gradient solver consumes the majority of computational time due to the sheer number of iterations required for convergence. It is not implemented in the LidDrivenCavity class, as the low computational cost does not justify the introduction of additional overhead.

Due to the necessity of computing the step size (alpha), beta, and error with multiple dot products excluding the local buffer zone, the OpenMP reduction technique is used to parallelise the nested loop. In nested loops, the collapse(2) option is activated to increase the granularity of work being parallelised. There is also the need for scalar and vector operations such as cblas_daxpy and cblas_dcopy. They are replaced with parallel for loops due to their nature, which is highly amenable to vectorisation.

4.2 OpenMP Scaling



Similar to MPI scaling, ideally, the relative computational time would decrease with an increasing number of threads; however, there is a balance between solver integration cost and the overhead of multi-threading, just as in MPI scaling. For both solver configurations (201x201 and 801x801), as the number of threads increases, the relative computational time decreases until reaching a certain point. After that, the effects of overhead increase, offsetting the benefits of parallel computing.

For the 801x801 mesh: from 1 to 4 threads, a decreasing trend in relative computational time is observed, with an 11.7% reduction from 1 to 2 threads and a further 8% reduction from 2 to 4 threads. However, after 4 threads, the relative computation time increases and eventually having the speed of only two threads. This is explained by the increased overhead introduced by the higher number of threads.

Overall, for both simulation configuration, the scaling plot doesn't exhibit good scaling, as there is no significant reduction in time with an increasing number of threads. However, this isn't due to any poor decisions in parallelisation, but rather the inherent nature of this simulation problem. A 801x801 grid isn't large enough or computationally expensive enough to demonstrate significant time reduction. It's anticipated to perform better on much larger problems, such as a 10001x10001 grid, where the solver integration cost outweighs the overhead costs. However, due to time constraints and the necessity to invest multiple hours into the simulation of a 10001x10001 grid, this isn't realised in the end.

5 Profiler and Code Optimisation

5.1 Profiler Analysis

Out of the two classes, Lid-driven cavity and conjugate gradient solver class, the conjugate gradient class consumes the absolute majority of the CPU time (almost 100%). This is due to the significantly heavier computational demand for solving the Poisson equation compared to computing the vorticity equations. On average, for a 201x201 grid simulation, 500-600 iterations are required for convergence, while the vorticity equations only need to be computed once in each time step.

Within the conjugate gradient solver, computing the global dot product (29%), applying the Laplace operator (23%), computing the global error (9%), and preconditioning (9%) are the top 4 most computationally expensive functions that can be optimised in the code, collectively consuming up to 70% of the total computational time. Once identified, the code optimisation process will focus mostly on these functions to achieve the greatest possible reduction in time. Although the overhead introduced by implementing multi-threading is non-negligible in coarse mesh problems, it is believed to be significantly reduced in real world scenario where a very fine mesh is applied.

5.2 Code Optimisation

The code optimisation process can be generalised as follows: Reordering nested for loops to address cache locality issues, vectorising vector operations that are highly amenable to vectorisation, pre-computing repeated expressions, and simplifying complex mathematics in the conjugate gradient solver.

Reordering nested for loops : In the baseline serial code upon which the code is based, the preconditioning step loops first in the x-direction and subsequently in the y-direction. This introduces undesirable effects since the x-direction is the leading dimension, and the y-coordinate is the fastest-changing dimension. By reordering the nested loop (for int j=0....) and then (for int i=0....), one can utilise the nearby data temporarily stored in the cache much more efficiently, without needing to fetch from the main memory, which takes significantly longer. This optimisation reduces the computational time for preconditioning (PreconditionParallel()) by 36.2% with the same solver configuration.

Computing repeated expression : When applying the Laplace operator, the macro function IDX(i,j) is originally called 6 times, which is very computationally expensive, given that it's the top 2 most expensive function. A local variable called index_local is declared to store the local index by calling IDX(i,j) once. This saves approximately 18.8% of computational time when applying the Laplace operator in the conjugate gradient solver.

The original code also declares new dx2i and dy2i variables and calculates their values in both the ApplyOperatorParallel() and PreconditionParallel() functions. This is not the most efficient way of computing the Laplace operator and preconditioning, as they can be easily computed only once during the initialisation of the conjugate gradient solver. This saves approximately 2.7% when computing the overall code.

Simplifying mathematical expression : Division is typically considered more computationally expensive than multiplication due to the higher hardware complexity involved. In the preconditioning step where the input would have to be divided by factor, this is replaced by another factor which is the inverse of it so multiplication is done. This saves approximately 1.8% in computation.

Vectorisation : The computation of the residual vector (r), search direction vector (p), solution vector (x), z, and t involves vector and scalar operations that are highly amenable to vectorisation. Therefore, one can take advantage of how the data is organised in memory. pragma for loops are used to leverage the available threads and also when the mesh is very fine.

Failed attempts : Although more optimisations have been implemented, such as precomputing the coordinates of the inner domain (the local domain without buffer), altering the Laplace operator equation, and using omp parallel for instead of omp for, all intended to reduce computational time, their effects are not visible and thus have been abandoned.

Compiler optimisation : Throughout the development process, o0 optimization, which involves no compiler optimisation, is used as it cannot be debugged. An o3 optimization is used when the code has finished debugging. This saves approximately 58.8% of total computational time compared to no compiler optimisation.

Appendix

Configuration: ./solver –Lx 1 –Ly 1 –dt 0.005 –T 10 –Re 1000 –Nx 201 –Ny 201 –verbose true