Parallelization of Computational Fluid Dynamic Simulation For Data Center Air-Conditioning System

Jin Hu(jinh), Siliang Lu(siliang1)

Dec 2016

Project Report

1 Introduction

We are going to implement a 2D CFD solver to simulate airflow patterns of a Data Center Air-Conditioning System with CUDA and MPI model.

1.1 Data Center Cooling

Data center is a specially designed space, which houses numerous servers with high power intensity. Therefore, data center requires cooling all year round. However, without proper airflow design of the air-conditioning system, it will take risks of failure due to high temperature zone. Therefore, CFD simulation is necessary for engineers to detect the potential problem due to bad air flow design.

In order to detect the potential problem due to bad air flow design, computational fluid dynamic (CFD) simulation is necessary. Our 2D Navier-Stokes solver is aimed to predict the interior air flow patterns and temperature distribution with different inlet velocities and validate the results with Ansys. The simulation results can be used to help computer room air conditioning(CRAC) system designers to identify better CRAC configurations.

1.2 Theories

Computational Fluid Dynamics is using computing machinery to solve/simulate physical phenomena with fundamental theory in the field of fluid dynamic/mechanics. With the governing equations of fluid mechanics: Navier-Stokes (N-S) equations (as shown below), one can simulate the flow patterns in a fluid space. Continuity equation:

$$\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} = 0 \tag{1}$$

Figure 1: Update of A Cell

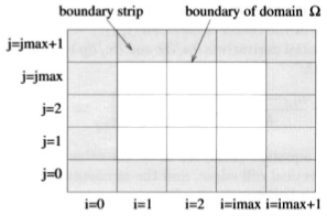

Figure 2: Overall Discretized Domain

Momentum equation:

$$\frac{\partial u}{\partial t} + \frac{\partial p}{\partial x} = \frac{1}{Re}(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}) - \frac{\partial (u^2)}{\partial x} - \frac{\partial uv}{\partial y} + g_x \tag{2}$$

$$\frac{\partial v}{\partial t} + \frac{\partial p}{\partial y} = \frac{1}{Re}(\frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2}) - \frac{\partial uv}{\partial x} - \frac{\partial (v^2)}{\partial y} + g_y \tag{3}$$

Energy equation:

$$\frac{\partial T}{\partial t} + \frac{\partial uT}{\partial x} + \frac{\partial vT}{\partial x} = \frac{1}{Re}\frac{1}{Pr}(\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2}) + q''' \tag{4}$$

where $Re = \frac{\rho V L}{\mu}$.

With finite difference method, a quadrilateral mesh grid can be created. However, there are many other topological meshes, which can be categorized as structured mesh and unstructured mesh. Instead of finite difference method, finite difference volume is a dominant discretized method used in CFD package since unstructured mesh can be easily implemented with it.

In addition, numerical flow simulation is a dominant method to solve the unsteady incompressible N-S equations. With this method, the continuous domain is discretized and apply the discretized domain to the continuous N-S equations, namely a finite-dimensional problem. The simplest discretization of N-S equations is to use finite difference method to update the values of each cell until it converged. For specific fluid problem, a cell of a staggered 2D grid is shown in figure 1. Besides, the overall discretized domain is shown in figure 2.

1.3 Problem & Opportunity

Since the finer mesh will increase the accuracy of CFD simulation, scientists expect to reduce the mesh size. However, the computational cost will increase dramatically as the mesh gets finer. Therefore, parallel computing becomes useful

Table 1: Velocity Boundary Conditions

	boundary conditions
inlet	inflow
outlet	outflow
axis	symmetry
others	no-slip

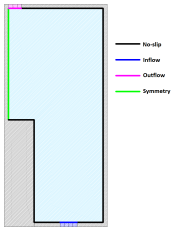

Figure 3: Datacenter Rack Model

in CFD simulation. In addition, due to complex geometry and boundary conditions of data center air-conditioning system, it is usually a time-consuming task to simulate the airflow patterns inside the data center. Hence, our problem is to parallelize the serial version of simulating airflow patterns of the air-conditioning system in a data center. physical model and boundary conditions (data-center; lid-driven cavity) The geometry is shown in figure 3 and the velocity boundary conditions are shown in table 1. Since the data center contains many racks, we just investigate the airflow patterns around one rack. In addition, since the geometry is symmetric, only half of the domain is calculated.

In order to simplify the geometry, the lid-driven cavity model (without obstacle) was firstly developed to ensure the correctness after parallelization. The physical model and the initial conditions of lid-driven cavity is shown in figure 4.

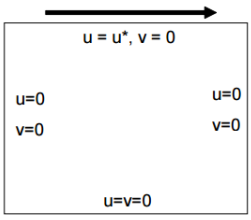

Figure 4: Visualization of underfloor air-distribution in data center

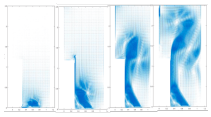

Figure 5: Lid Driven Model

1.4 Our Approach

We parallelize the 2D simulation of Data Center Air Conditioning System using Cuda and MPI. The parallel speedup is to be measured with the baseline of serial version in C. In addition, the correctness is to be guaranteed with the Matlab version, which has already been verified with commercial software of Ansys Workbench. The visualization result is shown in figure 5. More details about our implementation is discussed later.

1.5 Related Work

1.5.1 Parallelizing Gauss-Seidel with Red/Black Relaxation

Among different parts involved in numerical flow simulation, the most costly part is calculating pressure with Poisson equations. For the solution of such large, sparse linear systems, Gauss-Seidel method is applied that each cell (i,j) is successively processed in every cycle until the pressure value is satisfied.

1.5.2 Communication

Besides Red/Black relaxation, slave/master communication, namely message passing is also widely used in numerical flow simulation. Y.P. Chen et al. [2] estimated the communication cost for parallel CFD using variable time-stepping algorithm. In addition, Mattis Sellen presents an evaluation of parallel performance of CFD with unstructured grids and is parallelized by domain decomposition using MPI as the communication library [3]. It was found that the high performance network becomes significant when the system scales up. In other words, the node configuration where memory bandwidth saturation limits the performance with fewer nodes.

1.6 Contribution

We explored the possibility of parallellizing the CFD simulation with Cuda and message passing model and achieved a good speedup in our MPI implementation. It can be demonstrated that with better modular design some common

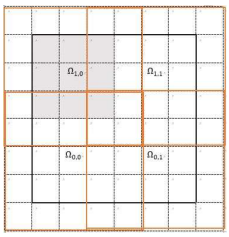

Figure 6: Overlapping Subdomains

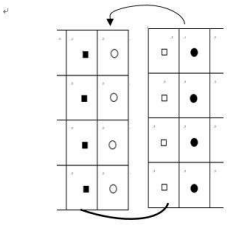

Figure 7: Exchange of Pressure Values

functions or models in physics can be parallellized and a programming framework for similar problems can be built with fast implementations with Cuda or MPI if needed.

2 Details on Design & Implementation

2.1 Message Passing

For message passing algorithms, we followed the approach to divide the underlying domain Ω into subdomains $\Omega_1, \Omega_2, \Omega_3, ...\Omega_N$ and have each processor treat one subdomain. This first domain decomposition method was proposed by Hermann Amandus Schwarz. In our solver, we used overlapping subdomains, as shown in figure 5. In each time-stepping loop, each subdomain is assigned to

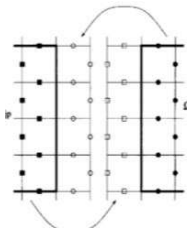

Figure 8: Exchange of Velocity Values

a processor and communicate with its neighbor processors or master processors when necessary.

2.1.1 Communication in Boundary Values Exchange

Since the subdomains overlaps with each other, the exchange of data in the boundary strips must be done in each time-stepping loop. The pressure values in the boundary strips and the velocity values in the boundary strips must be exchanged as shown in figure 7 and figure 8, respectively. The data exchange is performed four times (to the left, right, up and down).

2.1.2 Communication in Master/Slave Mode

Once each processor completes the update of pressure, it has to calculate the partial sum of the residual independently. Then the respective partial sums of the current residuals are all sent to the master processor so that it can add them up. Similarly to serial version, the master processor will decide whether to terminate the pressure iteration and broadcasts a message to all other processors to update their pressure iteration. Besides pressure iteration, master/slave mode is also implemented in time step size control, namely calculating the time interval in time-stepping loop. Instead of using constant time interval, we followed method of the adaptive stepsize control scheme proposed by Tome & McKee to calculate the interval using the following equation:

$$\delta t = \tau min(\frac{Re}{2}(\frac{1}{\delta x^2} + \frac{1}{\delta y^2})^{-1}, \frac{\delta x}{|u_{max}|}\frac{\delta y}{|v_{max}|} \tag{5}$$

where $\tau from[0, 1]$. Therefore, when calculating the time interval, the local maxima of u and v must be sent to the master processor, which determines the global maxima and calculate the new time interval and then broadcasts the latter to all processors.

2.2 Cuda Based method

The cuda based parallellization is implemented such that all the matrix related computation is done in the GPU memory. For all the for loop that is to update the matrix or matrix rows/columns, it can be computed in parallel in cuda threads without ordering restrictions. One exception is the Red-Black Gauss-Seidel parallellization step that is to be discussed more in detailed later. There are some other parts of the code that requires serialized computation, one is a sum of all the elements of a matrix, another is taking the max value of a matrix. The final implementation used third part cuBlas and thrust libraries for this. Multiple implementations are tried for improving this part of the computation and it turns out to be a significant part of cost of the whole simulation.

We also use one optimization to improve cache performance. We write new values to one copy of the matrix in memory while reading from another stale

copy of the matrix in memory. At the end of each iteration or where we need the new matrix, we swap the pointers to the two matrix.

The Red-Black Gauss-Seidel parallellization is based on the idea that the update of a cell u[i][j] is based on the value of u[i-1][j], u[i+1][j], u[i][j-1], u[i][j+1]. From the graph below, we can find that for all the cells u[i][j], if i+j is odd, it only depends on the even indices. If i+j is even, it only depends on the odd indices. This makes it possible to update half of the matrix in parallel without ordering restrictions but still keeping the correctness. The algorithm is as follows,

```
At each step
1. Send black ghost cells
2. Update red cells
3. Send red ghost cells
4. Update black ghost cells
```

With Red-Black Gauss-Seidel parallellization, the most repeated computation part of the simulation can be significantly speeded up.

For the max value of a given matrix, we tried using thrust max_element function in the extrema utilities. We also tried cublas's cublasIdamax function. We also implemented our own reduce based max element function. For the sum of a matrix, it is very similar to the max value function, we also used thrust reduce based implementation, cuBlas vector multiplication by creating a vector of all ones, and our own naive reduce based sum function.

Other parts of the code can be executed in parallel without other special interference or change of logic.

3 Experiments

3.1 Environment Setup

Our C/C++ code should be able to run with different machines specs that has Nvidia GPU and cuda libraries installed or has MPI libraries installed. We created Makefiles that shoule be straight forward to run to generate the simulation results. No other special setup is required to run our code. However, CFD simulation is computationally heavy and will benefit from good performance multi-core GPU/CPU hardware.

To measure the performance, we performed most of the tests and experiments on the latedays cluster. (There are some time that latedays are down or does not work properly, so we also used a AWS g2.2xlarge with GPU instance to run our experiments.)

For more details, please refer to readme file in the source code.

If running on CMU machines, PATH needs to be modified according to `http://15418.courses.cs.cmu.edu/fall2016/article/4`.

Table 2: Speedup of MPI Based Implementation

Nodes	Processor per node	32*32	128*128	256*256
1	1	1	1	1
1	4	1.74	1.98	3.53
1	16	1.78	2.51	7.01
2	2	1.823	0.86	2.04
2	8	0.2336	1.09	4.04

3.2 Experimental Evaluation

3.2.1 MPI experiments

The MPI based method speedup result is shown in the table 2. (We measure the end to end time and calculated the speedup based on the end to end time. This is not a good measure for the exact speed up for the parallellization part(because of Amdahl's law), but it does illustrate how the MPI method perform under different scenarios and provide a different and more practical insight.)

3.2.2 MPI Analysis

For the MPI implementation that works on a 20 seconds simulation, the time for a 256x256 mesh takes 1047.4 to run serially, 198 to run with 1*4 core, 38.16 seconds to run with 1*16 cores, which is pretty fast taking into consideration complexity of simulation.

As shown in the table, we've found the following results: 1. With the same amount of processors in only one node, the speed up increases as the mesh gets finer. 2. With the same amount of processors in two nodes, the speed up also increases as the mesh gets finer except for the mesh size of 128x128. The reason for the results above is that when mesh gets finer, each processor will have more computing within its own subdomain. Therefore, the percent of computational cost due to communication will be smaller. Because of that, message-passing parallelization could be more useful with large-scale mesh grid (finer or bigger). 3. With the different amount of processors either with only one node or with two nodes, the speed up is not proportional to the amount of processors used. The reason for that is that the memory bandwidth limits the performance of parallelization. 4. Compared to the speed up with only one node, the speed up with two nodes is not so satisfactory. That's because the network between different nodes will also result in extra computational cost.

3.2.3 Cuda experiments

The Cuda based method speedup is shown in the table.

Table 3: Speedup of Cuda Based Implementation

Mesh Size	16*16	64*64	128*128	512*512	1024*1024
Time Cuda	0.335330	0.587100	3.309735	40.260183	594.488618
Time Serial	0.001527	0.042056	0.426158	9.943977	142.390774
Speedup	1.00	1.82	3.29	5.22	7.88

3.2.4 Analysis

The Cuda based implementation does not outperform the CPU based code. The results are shown above.

We can find that the cuda based code is consistently slower than the CPU based code. We are not sure exactly what makes the cuda based code slow. There is no sing point or function that makes the cuda code slow, but each function takes longer than the serial one. There are a few points that may address the problem.

1. We may not correctly handle the GPU memory correctly such that memory is when using third part libraries, the conversion between device memory pointers might actually result in data copied from device to host and then back to device memory to conform to the thrust or cublas memory layout specification. 2. Another obvious drawback is that we did not manage to use the shared memory well in GPU. So even though we created two copies of each matrix to improve cache performance, we did not use shared memory explicitly where certain GPU cores are supposed to access certain parts of matrix. 3. Also, another important point is that most of the parallellized part of the code compose only a small proportion of the total execution time of the program. There are some intrinsic computations that can only be partially parallellized like the max(sum) and the Gauss Seidel relaxation part. The serial code uses SOR(another relaxation) in the iterations to reduce the computation to achieve a faster convergence speedup.

4 Surprises and Lessons Learned

1. Our first implementations of Cuda based parallellization encounters the problem that we later found to be the Red Black relaxation problem. Our original assumption was that all the cells in the pressure matrix p in the Poisson computation step can be updated in parallel without restrictions, which turned out to be wrong. It can be proved theoritically that some order must be respected when updating the matrix to achieve convergance. Then we found that we can use the Red Black relaxation, which is one popular way to parallel such problems. However, our original serial implementation is based on another relaxation(SOR), so we did not have enough time to re-structure the entire code base to support the Red Black relaxation.

The important lesson is that we as programmers still need to learn some theories behind the program to parallel before we start blindly to speedup the

code. We often have some wrong assumptions that we may find it hard to discover at first. The ordering restriction actually occurred in the renderer assignment, but I took it for granted and only thought that was an artificial restriction that does not happen often in the real world. So I spent a long time figuring out why our initial random ordered parallellized code does not converge.

2. When implementing message-passing model, one of the important things to keep in mind is that the order to MPI_Send and MPI_Recv. Since each subdomain has to communicate with all of its neighbor subdomain, it may result in never response if all of the processors responsible for subdomains in one row or in one column are sending message to its next processors at the same time. In addition, for some of the subdomains, the subdomain boundary coincides with the boundary of the base domain. Therefore, the processor responsible for such subdomain may not have to communicate with all four directions. Lastly but not least, since the base domain is decomposed into several subdomains, the corresponding boundary values set in the serial version may have to change since not all of subdomains coincides with the boundary of the base domain or obstacles.

5 Conclusions

Our project have implemented the parallelization of lid-driven cavity model. We demonstrated both our parallellized approach can generate the correct simulation results. The MPI based approach outperforms the serial code as the number of cores grow. It works well on a single node with multiple cores.

6 Future Work

6.1 For Cuda Based parallellization

1. We would study more third party libraries and consider re-implement it with thrust libraries given more time and compare the speedup. 2. We will also investigate other theoritcally sound relaxations that may is suitable for parallellization.

6.2 For CFD Model

We are facing a problem due to free boundary, which lies in the symmetric axis since we are computing half of the domain so as to save computing time. To parallelize the algorithm for solving free boundary value problems. For free boundary, each time the particles are advanced, it has to be determined which particles have crossed which subdomain boundary. Therefore, an additional particle list have to be added for each of the four subdomain boundaries. Therefore, for future work, we shall continue to parallelize the data center model by considering the special conditions for free boundary.

7 Distribution of Total Credit

We both contributed great effort to the project and worked happily as a team.

8 Website

Please refer to `http://thuhujin.github.io/cfd.html` for more information.

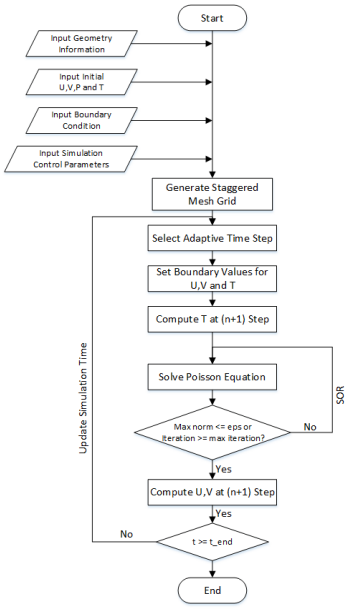

Figure 9: Simulation Algorithm Flowchart

9 Appendix

9.1 Simulation Algorithm

The simulation begins with geometry, boundary type and simulation control parameter input. To avoid serious oscillations of pressure value, a staggered grid mesh is then generated . The main time-stepping simulation loop starts after initialization. An adaptive time step is calculated for stability purpose. Boundary values of the matrices of x-velocity (U), y-velocity (V) is updated at each time step n. Then the values of temperature (T) at each cell in fluid domain at (n+1) time step is solved. With that, a linear system of pressure values (P) is solved with Poisson solver with successive over-relaxation (SOR). With pressure values (P) at (n+1) time step, x-velocity (U) and y-velocity at (n+1) time step can be solved. Simulation time and step are updated after each main iteration. The whole simulation terminates when simulation time is greater or equals to desired end time. The algorithm is shown in the flowchart. figure 9

9.2 Inputs and Output

The detailed initialization parameters are shown in the table below.

Input Parameters	Value
Inlet X-velocity U_{inlet}	Case 1:-1m/s; Case 2:0
Inlet Y-velocity V_{inlet}	Case 1:1m/s; Case 2:1m/s
Gravity at vertical direction g_y	0
Outlet relative pressure	0
Reynolds Number Re	17000
End time t	0-30s
SOR maximum iteration $Iter_{max}$	100
SOR relaxation parameter w	0.7
SOR tolerance	0.001

The outputs are the airflow velocity and temperature of each cell at the end time. In the end, a velocity plot of airflow distribution and temperature contour can be plotted.

References

[1] Griebel, M., Dornseifer, T., Neunhoeffer *Numerical simulation in fluid dynamics: a practical introduction (Vol. 3). Siam.*, 1997.

[2] Chien, Y. P., A. Ecer, H. U. Akay, S. Secer, and J. D. Chen. *Cost function for dynamic load balancing of explicit parallel CFD solvers with variable time-stepping strategies.* International Journal of Computational Fluid Dynamics 15, no. 3 (2001): 183-195

[3] Sillén, Mattias *Evaluation of Parallel Performance of an Unstructured CFD Code on PC-Clusters.* Journal of Aerospace Computing, Information, and Communication 2, no. 1 (2005): 109-119.