

# 3-Dimensional Navier-Stokes in C++ Framework and Parallelization Techniques Comparison

Christoph Neuhauser

Computer Science, Technical University of Munich

Stefan Haas

Computer Science, Technical University of Munich

Kwok Ping Ng

Data Engineering and Analytics, Technical University of Munich

July 22, 2019

## 1 Introduction

After experiencing 2D Navier-Stokes Equation, Navier-Stokes Equation in 3D has been implemented in our group project. C++ framework was replacing the traditional C programming. Then 3D calculation for velocity(u,v,w), force(F,G,H), pressure, residual and temperature has been reformed. Due to increase in the complexity of arbitrary geometry in 3D, a new algorithm was implemented. Our another goal was to elaborate how the CFD solver could be parallelized in a very efficient way. For this, we chose to compare a solver written in C++ with OpenMP (short: cpp solver), a solver using MPI for distributed-memory parallelism, a solver using CUDA on NVIDIA GPUs and a solver using OpenCL, which can be run on NVIDIA GPUs, Intel iGPUs and AMD GPUs. In the following section, we will compare these solvers regarding their best performance results, their scalability and their parallel efficiency.

## 2 3D incompressible Navier-Stokes equation

Non-stationary incompressible viscous fluid flow is described in 3-dimensional Navier-Stokes Equation. The analysis was carried out in Cartesian coordinates. The quantities are computed as u,v,w and F,G,H as x,y,z directions.

### 2.1 3D Momentum equations

$$\frac{\partial u}{\partial t} + \frac{\partial p}{\partial x} = \frac{1}{Re} \left( \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2} \right) - \frac{\partial(u^2)}{\partial x} - \frac{\partial(uv)}{\partial y} - \frac{\partial(uw)}{\partial z} + g_x \quad (1)$$

$$\frac{\partial v}{\partial t} + \frac{\partial p}{\partial y} = \frac{1}{Re} \left( \frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2} + \frac{\partial^2 v}{\partial z^2} \right) - \frac{\partial(uv)}{\partial x} - \frac{\partial(v^2)}{\partial y} - \frac{\partial(vw)}{\partial z} + g_y \quad (2)$$

$$\frac{\partial w}{\partial t} + \frac{\partial p}{\partial z} = \frac{1}{Re} \left( \frac{\partial^2 w}{\partial x^2} + \frac{\partial^2 w}{\partial y^2} + \frac{\partial^2 w}{\partial z^2} \right) - \frac{\partial(uw)}{\partial x} - \frac{\partial(vw)}{\partial y} - \frac{\partial(w^2)}{\partial z} + g_z \quad (3)$$

## 2.2 3D Continuity equation

$$\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} + \frac{\partial w}{\partial z} = 0 \quad (4)$$

## 2.3 Force F and velocity u calculation in x direction

Velocity w is essentially considered into the calculation then, the discretization for derivatives of u,v and w has to be with respect to directions x,y and z respectively.

F:

$$\begin{aligned} F_{i,j,k} = & u_{i,j,k} + \delta t \left( \frac{1}{Re} \left( \left[ \frac{\partial^2 u}{\partial x^2} \right]_{i,j,k} + \left[ \frac{\partial^2 u}{\partial y^2} \right]_{i,j,k} + \left[ \frac{\partial^2 u}{\partial z^2} \right]_{i,j,k} \right) - \left[ \frac{\partial(u^2)}{\partial x} \right]_{i,j,k} - \left[ \frac{\partial(uv)}{\partial y} \right]_{i,j,k} - \left[ \frac{\partial(uw)}{\partial z} \right]_{i,j,k} \right. \\ & \left. + g_x - \frac{\beta}{2} \left( T_{i,j,k}^{(n+1)} + T_{i+1,j,k}^{(n+1)} \right) g_x \right) \\ & i = 1, \dots, imax - 1; \quad j = 1, \dots, jmax; \quad k = 1, \dots, kmax \end{aligned} \quad (5)$$

u:

$$\begin{aligned} u_{i,j,k}^{(n+1)} = & F_{i,j,k}^{(n)} - \frac{\delta t}{\delta x} \left( p_{i+1,j,k}^{(n+1)} - p_{i,j,k}^{(n+1)} \right) \\ & i = 1, \dots, imax - 1; \quad j = 1, \dots, jmax; \quad k = 1, \dots, kmax \end{aligned} \quad (6)$$

## 2.4 Discretization for F

The midpoints of 3 directions will be towards to evaluate the derivative of u,v and w:

$$\left[ \frac{\partial^2 u}{\partial x^2} \right]_{i,j,k} = \frac{u_{i+1,j,k} - 2u_{i,j,k} + u_{i-1,j,k}}{(\delta x)^2} \quad (7)$$

$$\left[ \frac{\partial^2 u}{\partial y^2} \right]_{i,j,k} = \frac{u_{i,j+1,k} - 2u_{i,j,k} + u_{i,j-1,k}}{(\delta y)^2} \quad (8)$$

$$\left[ \frac{\partial^2 u}{\partial z^2} \right]_{i,j,k} = \frac{u_{i,j,k+1} - 2u_{i,j,k} + u_{i,j,k-1}}{(\delta z)^2} \quad (9)$$

$$\begin{aligned} \left[ \frac{\partial(u^2)}{\partial x} \right]_{i,j,k} = & \frac{1}{\delta x} \left( \left( \frac{u_{i,j,k} + u_{i+1,j,k}}{2} \right)^2 - \left( \frac{u_{i-1,j,k} + u_{i,j,k}}{2} \right)^2 \right) + \\ & \frac{\gamma}{\delta x} \left( \frac{|u_{i,j,k} + u_{i+1,j,k}|}{2} \frac{(u_{i,j,k} - u_{i+1,j,k})}{2} - \frac{|u_{i-1,j,k} + u_{i,j,k}|}{2} \frac{(u_{i-1,j,k} - u_{i,j,k})}{2} \right) \\ \left[ \frac{\partial(uv)}{\partial y} \right]_{i,j,k} = & \frac{1}{\delta y} \left( \frac{(v_{i,j,k} + v_{i+1,j,k})}{2} \frac{(u_{i,j,k} + u_{i,j+1,k})}{2} - \frac{(v_{i,j-1,k} + v_{i+1,j-1,k})}{2} \frac{(u_{i,j-1,k} + u_{i,j,k})}{2} \right) + \\ & \frac{\gamma}{\delta y} \left( \frac{|v_{i,j,k} + v_{i+1,j,k}|}{2} \frac{(u_{i,j,k} - u_{i,j+1,k})}{2} - \frac{|v_{i,j-1,k} + v_{i+1,j-1,k}|}{2} \frac{(u_{i,j-1,k} - u_{i,j,k})}{2} \right) \end{aligned} \quad (10)$$

(11)

$$\left[ \frac{\partial(uw)}{\partial z} \right]_{i,j,k} = \frac{1}{\delta z} \left( \frac{(w_{i,j,k} + w_{i+1,j,k})}{2} \frac{(u_{i,j,k} + u_{i,j,k+1})}{2} - \frac{(w_{i,j,k-1} + w_{i+1,j,k-1})}{2} \frac{(u_{i,j,k-1} + u_{i,j,k})}{2} \right) +$$

$$\frac{\gamma}{\delta z} \left( \frac{|w_{i,j,k} + w_{i+1,j,k}|}{2} \frac{(u_{i,j,k} - u_{i,j,k+1})}{2} - \frac{|w_{i,j,k-1} + w_{i+1,j,k-1}|}{2} \frac{(u_{i,j,k-1} - u_{i,j,k})}{2} \right)$$
(12)

## 2.5 Energy equation and Discretization

The energy equation in 3D and its discretization are easy to compute:

Energy Equation:

$$T_{i,j,k}^{(n+1)} = T_{i,j,k}^{(n)} + \delta t \left( \frac{1}{Re} \frac{1}{Pr} \left( \left[ \frac{\partial^2 T}{\partial x^2} \right]_{i,j,k} + \left[ \frac{\partial^2 T}{\partial y^2} \right]_{i,j,k} + \left[ \frac{\partial^2 T}{\partial z^2} \right]_{i,j,k} \right) - \left[ \frac{\partial(uT)}{\partial x} \right]_{i,j,k} - \left[ \frac{\partial(vT)}{\partial y} \right]_{i,j,k} - \left[ \frac{\partial(wT)}{\partial z} \right]_{i,j,k} \right)$$
(13)

Discretization:

$$\left[ \frac{\partial^2 T}{\partial x^2} \right]_{i,j,k} = \frac{T_{i+1,j,k} - 2T_{i,j,k} + T_{i-1,j,k}}{(\delta x)^2}$$
(14)

$$\left[ \frac{\partial(uT)}{\partial x} \right]_{i,j,k} = \frac{1}{\delta x} \left( u_{i,j,k} \frac{T_{i,j,k} + T_{i+1,j,k}}{2} - u_{i-1,j,k} \frac{T_{i-1,j,k} + T_{i,j,k}}{2} \right) +$$

$$\frac{\gamma}{\delta x} \left( |u_{i,j,k}| \frac{T_{i,j,k} - T_{i+1,j,k}}{2} - |u_{i-1,j,k}| \frac{T_{i-1,j,k} - T_{i,j,k}}{2} \right)$$
(15)

## 2.6 SOR Solver

Pressure:

$$p_{i,j,k}^{it+1} = (1 - \omega) p_{i,j,k}^{it} + \frac{\omega}{2 \left( \frac{1}{(\delta x)^2} + \frac{1}{(\delta y)^2} + \frac{1}{(\delta z)^2} \right)} \left( \frac{p_{i+1,j,k}^{it} + p_{i-1,j,k}^{it+1}}{(\delta x)^2} + \frac{p_{i,j+1,k}^{it} + p_{i,j-1,k}^{it+1}}{(\delta y)^2} + \frac{p_{i,j,k+1}^{it} + p_{i,j,k-1}^{it+1}}{(\delta z)^2} - r s_{i,j,k} \right)$$

$$it = 1, \dots, itmax; \quad i = 1, \dots, imax; \quad j = 1, \dots, jmax; \quad k = 1, \dots, kmax$$
(16)

Residual:

$$res := \left( \sum_{i=1}^{imax} \sum_{j=1}^{jmax} \sum_{k=1}^{kmax} \left( \frac{p_{i+1,j,k} - 2p_{i,j,k} + p_{i-1,j,k}}{(\delta x)^2} + \frac{p_{i,j+1,k} - 2p_{i,j,k} + p_{i,j-1,k}}{(\delta y)^2} + \frac{p_{i,j,k+1} - 2p_{i,j,k} + p_{i,j,k-1}}{(\delta z)^2} \right. \right.$$

$$\left. \left. - r s_{i,j,k} \right)^2 / (imax \cdot jmax \cdot kmax) \right)^{1/2}$$
(17)

## 3 Boundary Conditions

### 3.1 No Slip

Under the no-slip conditions, zero velocities at boundary. So we can set it to zero lying right on the boundary. Moreover, the boundary value zero is achieved by averaging the values on both sides of the boundary. For example, left wall case:  $u_{0,j,k} = 0$ ,  $v_{0,j,k} = -v_{1,j,k}$ ,  $w_{0,j,k} = -w_{1,j,k}$ .

### 3.2 Free Slip

It is the case when fluid flow freely parallel to the boundary but cannot across the boundary. In 3D cases, the fluid freely flow along two dimension but cannot across remaining direction. For example, the velocity  $u$  of left wall which is normal(perpendicular) to the wall is 0. Its boundary condition is to set as  $u_{0,j,k} = 0$ . And other boundary conditions to set  $v_{0,j,k} = v_{1,j,k}$ ,  $w_{0,j,k} = w_{1,j,k}$ . See Figure 1 below.

**\*\*picture\*\***

### 3.3 Outflow

the normal velocity derivatives are set to 0 at boundary. So the total velocity does not change in the direction normal to the boundary. The values of velocities at boundary can be set equal to the neighbouring velocities inside the region. For example, left wall case:  $u_{0,j,k} = u_{1,j,k}$ ,  $v_{0,j,k} = v_{1,j,k}$ ,  $w_{0,j,k} = w_{1,j,k}$ .

### 3.4 Boundary conditions for pressure and temperature

The boundary conditions for the pressure derived from momentum equation, resulting in discrete Neumann conditions. The boundary conditions for temperature derived from energy equation, resulting in discrete Dirichlet Condition for cold wall and hot wall separately. For example, left wall case:  $P_{0,j,k} = P_{1,j,k}$ .  $T_{0,j,k} = 2 * T_c / T_h - T_{1,j,k}$ .

## 4 Performance

### 4.1 CUDA

First of all, on consumer hardware, the CUDA solver and the OpenCL solver had considerable advantages over the CPU solvers. They can be seen in figure 1 which where run on on a NVIDIA GeForce GTX 1070, a 6-core Intel Core i7-8700 and on a 28-core Intel Xeon E5-2697 from the Linux Cluster of the LRZ. Graphics cards offer a much higher number of cores available for applications to utilize compared to similar prized-processors. However, GPUs execute workloads in a SIMD-fashion in lockstep, which means that a thread warp (that usually consists of 32 or 64 individual threads) would need to go the same route in the code without any diverging branches to reach the maximum possible performance. Furthermore, another problem is that especially NVIDIA limits their double-precision performance on consumer-level graphics cards to create an additional sales rationale for their high-priced workstation cards.

In theory, according to [1], the throughput of arithmetic double-precision floating point instructions on the GTX 1070 would be 1 : 32. However, the CUDA solver proved to be only approximately

48% slower in our test case for double-precision floating points values compared to single-precision floating point values, which can be seen in figure .... This can be probably attributed to the application being memory throughput limited for floating point operations rather than being limited by the throughput of arithmetic instructions.

On modern CPU architectures, the floating point units (FPUs) don't offer separate modules for single-precision and double-precision instructions anymore. Thus, as expected, the CPU solvers were only approximately 2% slower for double-precision floating-point numbers compared to single-precision floating-point numbers. The slight reduce in performance can probably also be attributed to the higher memory throughput of 64-bit wide double-precision floating-point numbers.

## 4.2 MPI

When comparing the scalability of the solvers, it is of course clear that the MPI solver offers the best scalability, as a memory-distributed program can be scaled to an arbitrary amount of cores. The weakness of the MPI solver comes into play when comparing it to a shared-memory implementation of the CFD solver on systems with a moderate number of cores. On our 6-core test system, the MPI solver was approximately 35% slower than the OpenMP solver. Figure 1 shows this difference. This is of course due to the communication overhead for the MPI solver. The advantage of MPI comes into play when utilizing multiple nodes of a cluster at once with distributed memory. When using two nodes and 56 cores, the MPI solver (with a domain decomposition of  $7 \times 4 \times 2$ ) gave by far the best performance results of all CPU solvers, but the CUDA solver still was 85% faster on the NVIDIA GeForce GTX 1070 even when using double-precision performance. There are probably two reasons for that. First of all, we used strong scaling with a fixed global problem size of  $80 \times 20 \times 10$  cells. It is of course comprehensible that the overhead for communication becomes really large for this high number of processes and a small local domain size. Secondly, the CUDA solver is able to very efficiently utilize the SIMD functionality of the GPU to provide a really large number of threads accessing very fast shared GDDR5 memory without hardly any communication overhead.

## 4.3 SOR and alternatives

When parallelizing the CFD solver, it is also necessary to parallelize the solver used for the linear system of equations of the Pressure Poisson Equation (PPE).

For the parallel solvers, we decided to implement and compare a Jacobi, Gauss-Seidel (GS) and successive over-relaxation (SOR) solver. Each of the solvers has its own advantages and disadvantages.

Jacobi is the solver to converge slowest (measured by the number of iterations, not the runtime on multiple cores). However, its greatest advantage is that it can be easily parallelized, as it has no data dependencies within one iteration.

Gauss-Seidel converges in less iterations than Jacobi, but is harder to parallelize. The vanilla SOR algorithm without any adaptations generally cannot run in parallel. Thus, Gauss-Seidel only has an

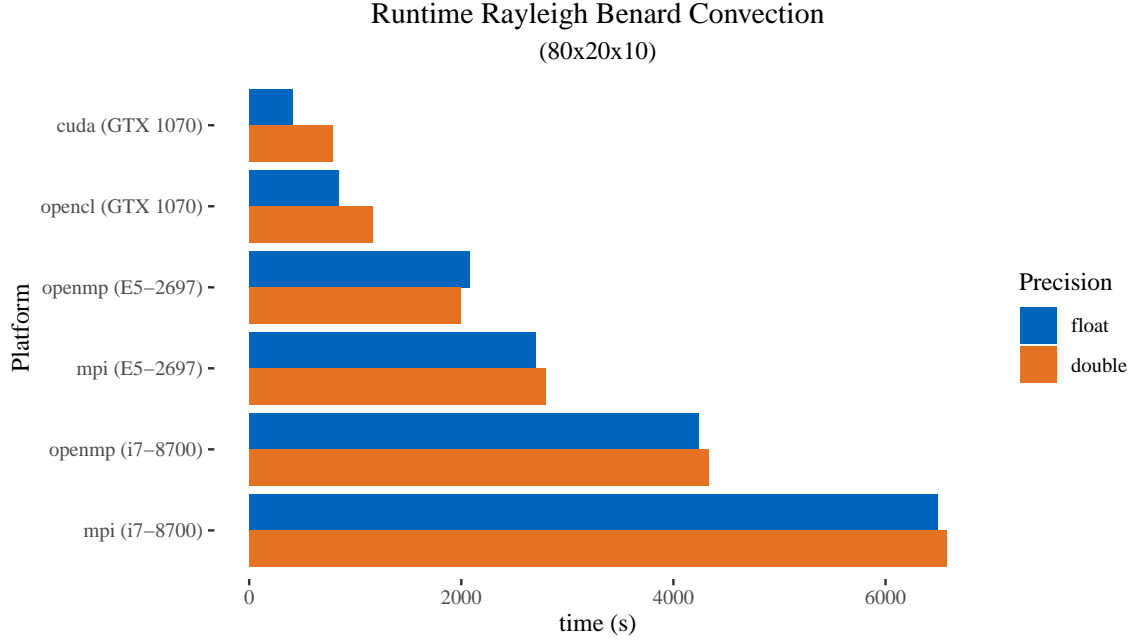


Figure 1: Write caption here.

advantage over Jacobi when using a low number of cores. We tested the parallel SOR solver by Cunha and Hopkins [2], but for sparse system matrices like in our case, the sequential update step in the end of the algorithm quickly becomes limiting for a high number of cores due to Amdahls law.

SOR is an algorithm based on Gauss-Seidel that couples the algorithm with an over-relaxation factor that can increase the convergence speed. In our program, we tested multiple values, and got the best results for an over-relaxation factor of 1.5. In our tests, which can be seen in figure 2, Gauss-Seidel was faster than SOR, which we didn't expect. Again, SOR suffers from the same weakness as Gauss-Seidel that it cant be parallelized easily. Thus, again, for a high number of cores, the Jacobi solver was able to beat the SOR solver.

Another method for solving a system of linear equations mentioned less often is the Jacobi over-relaxation (JOR), which transfers the principle of SOR to the Jacobi solver. However, for over-relaxation factors greater than 1, the solution would not converge. Thus, we rejected the JOR solver, as it can only increase convergence speed for a factor larger than 1. In the literature, we couldnt find any other source having success with JOR coupled with a factor greater than 1.

Taking all factors mentioned above into account, we decided to select the Jacobi solver as the standard solver in our program, as it converges considerably faster in the same time compared to the SOR solver for a high number of cores. In figure 2 it can be seen, that the difference between the solver when using MPI is really small, since all solvers are parallelised. In the case of Openmp our tests show that our decision to use Jacobi was correct.

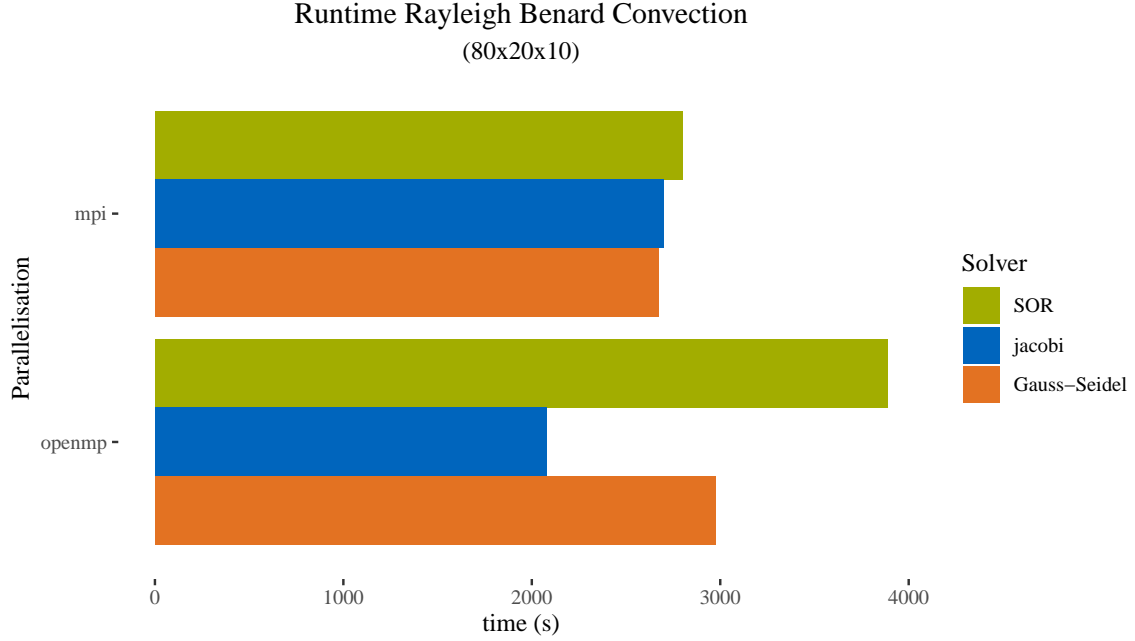


Figure 2: Write caption here.

#### 4.4 Speedup

After we chose the best solver we tested the performance when using a different number of cores. These tests were run on the Linux Cluster of the LRZ, where we used one node with a Intel Xeon E5-2697 for the speedup tests with OpenMP and up to two nodes with the same processor for the speedup tests with MPI.

The results of the tests with OpenMP can be seen in figure 3. The speedup is exactly as we expected. At the beginning the performance scales very good while the speedup at the end degrades. At 28 cores the speedup is approximately 14.

To test the MPI implementation we used one node for the first test. The reason we used 27 out of the 28 cores is that the domain could be split better with this amount of mpi processes, since the domain can be split in 3x3x3 parts, whereas with 28 processes it would be split into 7x2x2 part. We also tested both variants and the version with 27 processes was faster. With 27 cores the speedup is around ten which is not as good as with OpenMP, but when using two nodes with 56 cores altogether, the performance scales almost linearly with an speedup of approximately 19, even though the domain is split in 7x4x2 part, which is not optimal.

## 5 Arbitrary Geometries

### 5.1 u,v,w

Unlike the 2D cases which is necessarily to set the arbitrary geometries of velocities for 8 directions(4 lines + 4 corners of a square), there are total 26 directions (12 lines, 8 corners and 6 faces of a cube) to be set. Then setting it in 3D brought difficulties if we used the method from ws2. Therefore, we

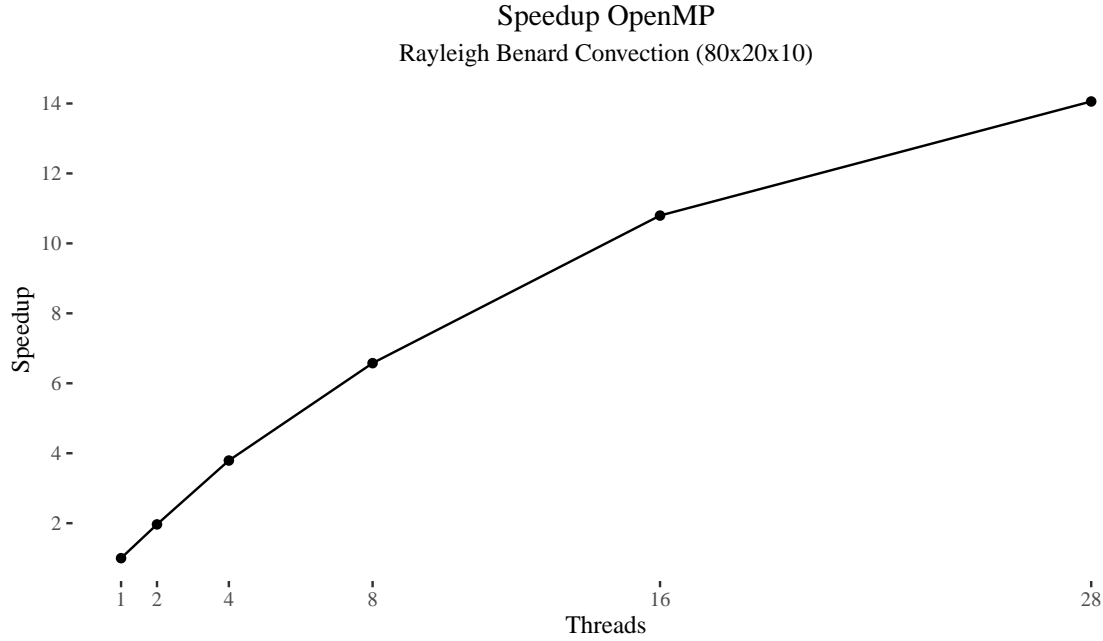


Figure 3: Write caption here.

designed a new algorithm to set arbitrary geometries.

We observed that we could set velocity to zero lying right on the boundary. Moreover, the boundary value zero was achieved by averaging the values on both sides of the boundary. It was the similar case to set no-slip boundary condition on walls. We set  $u, v$  and  $w$  separately. The flag of direction was checked if the velocity was lying on. If yes, the velocity was set to zero. Then an indicator was inserted that this velocity was not changed anymore. Otherwise, the velocity was set to equal to the negative of the neighboring velocity. For example, here is pseudo code of the arbitrary geometry setting of  $u$  on left wall.

---

**Algorithm 1** My algorithm

---

```

if  $i \geq maxval$  then
     $i \leftarrow 0$ 
else
    if  $i + k \leq maxval$  then
         $i \leftarrow i + k$ 

```

---

## References

- [1] Rudnei Dias da Cunha, Tim Hopkins. 1991. Parallel Overrelaxation Algorithms for Systems of Linear Equations.



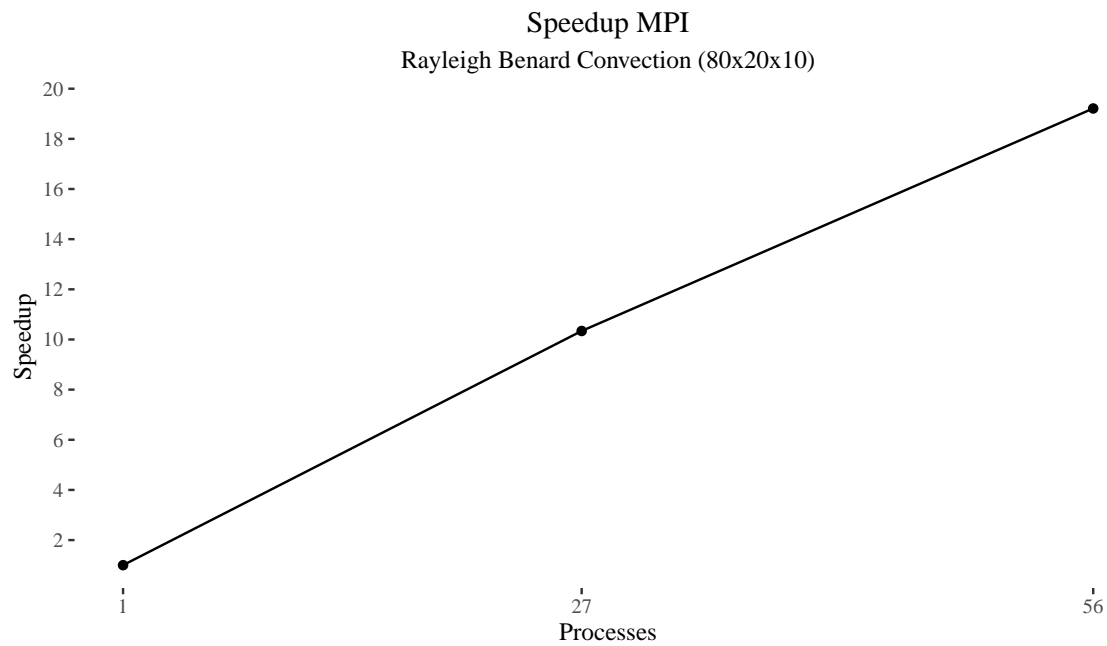


Figure 4: Write caption here.

- [2] M. Griebel, T. Dornseifer, and T. Neunhoeffler. SIAM, Philadelphia, 1998. Numerical Simulation in Fluid Dynamics, a Practical Introduction.
- [3] NVIDIA. 2019. CUDA C Programming Guide. <https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#arithmetic-instructions> visited on 2019-07-18.