Parallelized Incompressible Navier-Stokes Equations Solvers in a Modern C++ Framework

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 studying the incompressible Navier-Stokes equations in 2D in the practical course, our goal was to implement the incompressible Navier-Stokes equations in 3D as our group project. To replace the C framework from the practical course, a modern C++ framework was created. Our first step was to rewrite the momentum, continuity and energy equations. In 3D, it is necessary to add a third velocity component, called W. More information on the equations in 3D can be found in [1].

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 describe the code and how to use it, as well as compare these solvers regarding their best performance results, their scalability and their parallel efficiency.

Finally, we simulated multiple scenarios like Rayleigh-Benard convection, and multiple real-world scenarios like the air flow around Mt. Fuji in Japan or the German mountain Zugspitze.

2 Code description

2.1 Code framwork

The C++ Code is structured as follows. In the file main.cpp, which is located directly in the source directory, the main function is defined. There the core of the program resides, which is basically of the same structures as the 2D CFD code. Furthermore, there also is the Defines.hpp file, which defines the floating-point precision (float or double) as well as some define statements for the mapping of 3D indices in the staggered grid to 1D array indices. We couldn't use multi-dimensional arrays directly, as they aren't supported by CUDA and OpenCL.

Then there is the IO folder, which contains everthing that has to do with input and output functionality. Among others there are the VTK writer as well as the NetCDF writer and some classes to read and process different input formats for the geometry.

The folder ParticleTracer contains classes for tracing the characteristic lines of the flow. It was created for directly exporting the streamlines of the flow if ParaView isn't used for visualization.

The most important part of the program is the CdfSolver folder, where all solvers as well as the solver interface is located. The interface defines the following methods:

```
void initialize(
    const std::string &scenarioName,
    LinearSystemSolverType linearSystemSolverType,
    bool shallWriteOutput, Real Re, Real Pr, Real omg, Real eps,
    int itermax, Real alpha, Real beta, Real dt, Real tau,
    Real GX, Real GY, Real GZ, bool useTemperature, Real T_h, Real T_c,
    int imax, int jmax, int kmax, Real dx, Real dy, Real dz,
    Real *U, Real *V, Real *W, Real *P, Real *T, uint32_t *Flag)
```

which copies the passed initial values of U, V, W, P, T and Flag to the internal representation of the solver.

```
void setBoundaryValues()
```

which sets the boundary condition values of U, V, W and T using the Flag array.

```
void setBoundaryValuesScenarioSpecific()
```

which sets special boundary conditions (typically something like inflow) specific to the different scenarios.

```
Real calculateDt()
```

which calculates the smallest possible (plus some safety margin) time step for the simulation at the current state.

```
void calculateTemperature()
```

which updates the temperature values (using an intermediate copy of the temperature from the last iteration).

```
void calculateFgh()
```

which computes the values in the helper arrays F, G and H necessary to compute the right-hand side of the Pressure.

```
void calculateRs()
```

which computes the right-hand side of the Pressure Poisson Equation (PPE).

```
void executeSorSolver()
```

which executes the SOR solver (successive over-relaxation) for solving the Pressure Poisson Equation (PPE).

```
void calculateUvw()
```

which updates the values in the arrays U, V and W.

```
void getDataForOutput(Real *U, Real *V, Real *W, Real *P, Real *T)
```

which copies the values of the internal representations of U, V, W, P and T to the specified arrays. This is necessary when outputting the simulation results at certain time intervals. This function abstracts the process of either copying the values of the internal representation in VRAM on GPUs when using CUDA or OpenCL to the main memory or copying the values of the internal representation of the MPI or OpenMP solvers in main memory to the representation in the main function also in main memory.

The CfdSolver directory also features classes to work with the flag array as well as to initialize the arrays for the different variables.

Finally, as sub-folders of the CfdSolver folder, there are the different folders which contain the individual solvers. Each solver has the same basic structure even though some have additional files and classes for different solver-specific functions. Each solver has a file to manage the boundary values, the actual CfdSolver class, which mostly initializes variables and calls the function or kernels of the other files, the SOR solver file, where the SOR, Jacobi and/or the Gauss-Seidel solver are defined, and finally the UVW file, which calculates everything what has to do with u, v and w, the temperature or the timestep.

2.2 Building the code

The program is compiled with cmake. In order to build it, some packages have to installed (see the README file for more information on how to install them on Ubuntu) and the following commands have to be issued within the repository directory:

```
mkdir build
cd build
cmake ..
make
```

In order to use the different solvers, some arguments have to be given to cmake.

- For MPI support: cmake .. -DUSE_MPI=ON
- For CUDA support: cmake .. -DUSE_CUDA=ON
- For OpenCL support: cmake .. -DUSE_OPENCL=ON

These arguments can also be used together for the program to support multiple solvers at once.

2.3 Running the program

When executing the program, multiple arguments can be passed on the command line. These arguments are shown in the following list as well as their respective valid values.

- scenario:
  - driven_cavity
  - flow_over_step
  - natural_convection
  - rayleigh_benard_convection_8-2-1
  - rayleigh_benard_convection_8-2-2
  - rayleigh_benard_convection_8-2-4
  - rayleigh_benard_convection_2d
  - single_tower
  - fuji_san
  - zugspitze
- solver:
  - cpp
  - mpi
  - cuda
  - opencl
- outputformat:
  - netcdf
  - vtk (= vtk-binary)
  - vtk-binary
  - vtk-ascii
- output:
  - true
  - false (whether to write an output file)
- linsolver:
  - jacobi
  - sor
  - gauss-seidel
- tracestreamlines:
  - false
  - true

- numparticles:
  - any positive integer number
- numproc
  - Must be used with the MPI solver. Specifies the number of processes in x, y and z direction and must match the total number of MPI processes.
  - possible values: positive integer positive integer positive integer
  - example: –numproc = 2 2 2
- blocksize
  - Can be used with the CUDA or the OpenCL solver. Specifies the block size in x, y and z direction.
  - possible values: positive integer positive integer positive integer
  - example: –blocksize = 4 4 4
- platformid
  - Can be used with the OpenCL solver. Specifies the ID of the OpenCL platform to use. If the ID is not specified, it is set to zero. Which platform corresponds to which ID can be found out with the command line tool 'clinfo'.
  - possible values: integer

The standard values for the arguments are:

- scenario: driven_cavity
- solver: cpp
- outputformat: vtk
- outputformat: true
- linsolver: jacobi
- tracestreamlines: false
- numparticles: 500
- blocksize: 8 8 4
- platformid: 0

Here are some examples on how the code can be run:

```
./cfd3d --scenario driven_cavity --solver cpp
mpirun -np 8 ./cfd3d --solver mpi --numproc 2 2 2
./cfd3d --scenario driven_cavity --solver cuda
```

3 3D Incompressible Navier-Stokes equation

The flow of non-stationary incompressible viscous fluid is described by the three dimensional incompressible Navier-Stokes equations. The quantities u, v, w and F, G, H are computed as x, y, z directions, respectively.

3.1 3D Momentum equations

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

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

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

3.2 3D Continuity equation

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

3.3 Force F and velocity u calculation in x direction

The velocity w was added to the calculations from the 2D case and then, the discretization for the derivatives of u, v and w was derived with respect to directions x, y and z, respectively. Below, the computation of F and u is exemplified. The computation of G, H and v, w is analogous.

F:

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

u:

$$\begin{aligned} u^{(n+1)}_{i,j,k} = & F^{(n)}_{i,j,k} - \frac{\delta t}{\delta x}\left(p^{(n+1)}_{i+1,j,k} - p^{(n+1)}_{i,j,k}\right) \\ & i = 1, \ldots, imax - 1; \quad j = 1, \ldots, jmax; \quad k = 1, \ldots, kmax \end{aligned} \tag{6}$$

3.4 Discretization for F

The derivatives of the three directions are discretized as follows using the midpoint rules and the donor cells scheme (again, exemplified only for u).

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

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

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

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

$$\begin{aligned}\left[\frac{\partial(uv)}{\partial y}\right]_{i,j,k} =& \frac{1}{\delta y}\left(\frac{(v_{i,j,k}+v_{i+1,j,k})}{2}\frac{(u_{i,j,k}+u_{i,j+1,k})}{2} - \frac{(v_{i,j-1,k}+v_{i+1,j-1,k})}{2}\frac{(u_{i,j-1,k}+u_{i,j,k})}{2}\right) + \\ & \frac{\gamma}{\delta y}\left(\frac{|v_{i,j,k}+v_{i+1,j,k}|}{2}\frac{(u_{i,j,k}-u_{i,j+1,k})}{2} - \frac{|v_{i,j-1,k}+v_{i+1,j-1,k}|}{2}\frac{(u_{i,j-1,k}-u_{i,j,k})}{2}\right)\end{aligned} \tag{11}$$

$$\begin{aligned}\left[\frac{\partial(uw)}{\partial z}\right]_{i,j,k} =& \frac{1}{\delta z}\left(\frac{(w_{i,j,k}+w_{i+1,j,k})}{2}\frac{(u_{i,j,k}+u_{i,j,k+1})}{2} - \frac{(w_{i,j,k-1}+w_{i+1,j,k-1})}{2}\frac{(u_{i,j,k-1}+u_{i,j,k})}{2}\right) + \\ & \frac{\gamma}{\delta z}\left(\frac{|w_{i,j,k}+w_{i+1,j,k}|}{2}\frac{(u_{i,j,k}-u_{i,j,k+1})}{2} - \frac{|w_{i,j,k-1}+w_{i+1,j,k-1}|}{2}\frac{(u_{i,j,k-1}-u_{i,j,k})}{2}\right)\end{aligned} \tag{12}$$

3.5 Energy equation and Discretization

The energy equation in 3D and its discretization are given as follows.

Energy Equation:

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

Discretization:

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

$$\begin{aligned}\left[\frac{\partial(uT)}{\partial x}\right]_{i,j,k} =& \frac{1}{\delta x}\left(u_{i,j,k}\frac{T_{i,j,k}+T_{i+1,j,k}}{2} - u_{i-1,j,k}\frac{T_{i-1,j,k}+T_{i,j,k}}{2}\right) + \\ & \frac{\gamma}{\delta x}\left(|u_{i,j,k}|\frac{T_{i,j,k}-T_{i+1,j,k}}{2} - |u_{i-1,j,k}|\frac{T_{i-1,j,k}-T_{i,j,k}}{2}\right)\end{aligned} \tag{15}$$

3.6 SOR Solver

Pressure:

$$p_{i,j,k}^{it+1} = (1-\omega)p_{i,j,k}^{it} + \frac{\omega}{2(\frac{1}{(\delta x)^2}+\frac{1}{(\delta y)^2}+\frac{1}{(\delta z)^2})}\left(\frac{p_{i+1,j,k}^{it}+p_{i-1,j,k}^{it+1}}{(\delta x)^2}+\frac{p_{i,j+1,k}^{it}+p_{i,j-1,k}^{it+1}}{(\delta y)^2}+\frac{p_{i,j,k+1}^{it}+p_{i,j,k-1}^{it+1}}{(\delta z)^2}-rs_{i,j,k}\right)$$
$$it=1,\ldots,itmax; \quad i=1,\ldots,imax; \quad j=1,\ldots,jmax; \quad k=1,\ldots,kmax \tag{16}$$

Residual:

$$res := \left(\sum_{i=1}^{imax}\sum_{j=1}^{jmax}\sum_{k=1}^{kmax}\left(\frac{p_{i+1,j,k}-2p_{i,j,k}+p_{i-1,j,k}}{(\delta x)^2}+\frac{p_{i,j+1,k}-2p_{i,j,k}+p_{i,j-1,k}}{(\delta y)^2}+\frac{p_{i,j,k+1}-2p_{i,j,k}+p_{i,j,k-1}}{(\delta z)^2}\right.\right.$$
$$\left.\left.-rs_{i,j,k}\right)^2/(imax\cdot jmax\cdot kmax)\right)^{1/2} \tag{17}$$

4 Boundary Conditions

4.1 No Slip

When using no-slip conditions, zero velocities are imposed at the boundaries. In the staggered grid, it may be necessary to set the negative value at the boundary from within the domain, as averaging the values on both sides of the boundary would then give exactly zero. For example, in the left wall case we have: $u_{0,j,k} = 0$,$v_{0,j,k}$ = -$v_{1,j,k}$, $w_{0,j,k}$ = -$w_{1,j,k}$.

4.2 Free Slip

Free-slip means that the fluid can flow freely parallel to the boundary but cannot have a perpendicular flow component at the boundary. In 3D cases, the fluid freely flows along two dimensions, but cannot flow along the remaining direction. For example, the velocity at the left wall is imposed to be 0. Its boundary values are set to $u_{0,j,k} = 0$, $v_{0,j,k} = v_{1,j,k}$, and $w_{0,j,k} = w_{1,j,k}$. For more details see figure 1 below.

4.3 Outflow

For the outflow boundary condition, the normal velocity derivatives are set to 0 at the boundary. This means that the total velocity does not change in the direction normal to the boundary. The values of velocities at boundaries can be set equal to the neighbouring velocities inside the domain. For example, in the left wall case we have: $u_{0,j,k} = u_{1,j,k}$,$v_{0,j,k} = v_{1,j,k}$, $w_{0,j,k} = w_{1,j,k}$

4.4 Boundary conditions for pressure and temperature

The boundary conditions for the pressure are derived from the momentum equation and result in discrete Neumann conditions. The boundary conditions for the temperature, on the other hand,

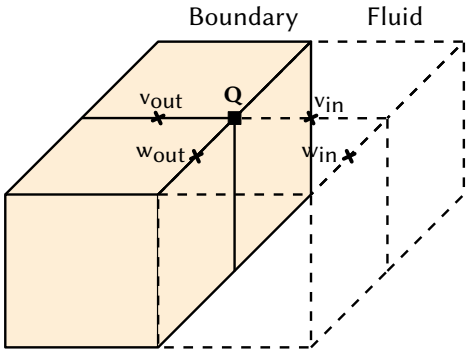

Figure 1: Boundary conditions for an obstacle wall on the left.

are derived from the energy equation and result in discrete Dirichlet boundary conditions for cold walls and hot walls, respectively. For example, in the left wall case we have: $P_{0,j,k} = P_{1,j,k}$. $T_{0,j,k} = 2 * T_c/T_h - T_{1,j,k}$.

5 Performance

5.1 CUDA

First of all, on consumer hardware, the CUDA solver and the OpenCL solver had considerable advantages over the CPU solvers. The results can be seen in figure 2, which where generated 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 [2], 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 point values compared to single-precision floating point values, which can be seen in figure 2. This can probably be 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 in our test case. The slight reduce in performance can probably also be attributed to the higher memory throughput caused by 64-bit wide double-precision floating-point numbers.

5.2 MPI

When comparing the scalability of the solvers, it is of course evident 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 2 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 7x4x2) 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 comparing double-precision performance. There are probably two reasons for that. First of all, we used strong scaling with a fixed global problem size of 80x20x10 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 almost any communication overhead.

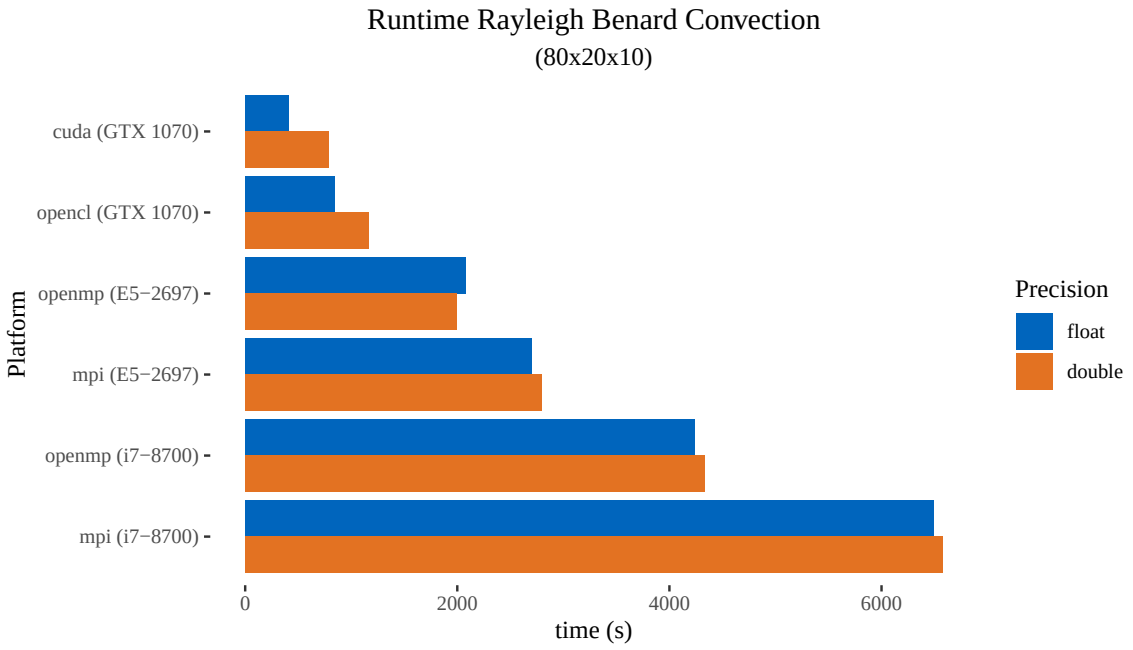

Figure 2: Performance results for different solvers. E2-2697: 28 cores, i7-8700: 6 cores. All results with Jacobi solver.

5.3 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 adaptions generally cannot run in parallel. Thus, Gauss-Seidel only has an advantage over Jacobi when using a low number of cores. We tested the parallel SOR solver by Cunha and Hopkins [3], 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 Amdahl's 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 generally got the best results for an over-relaxation factor of 1.5. In our tests, which can be seen in figure 3, Gauss-Seidel was faster than SOR, which is not what one might expect. However, when testing multiple over-relaxation factors, we noticed that the test scenario we chose (Rayleigh-Benard convection) needs a much lower over-relaxation factor than other scenarios. For an over-relaxation factor of 1.2, it performed slightly better than Gauss-Seidel. Again, SOR suffers from the same weakness as Gauss-Seidel that it cannot 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 could not 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 3 it can be seen that the difference between the solvers when using MPI is really small, since all solvers are parallelised (as the Gauss-Seidel scheme is replaced with a Jacobi scheme at the local domain boundaries of the processes for SOR and Gauss-Seidel). In the case of OpenMP, our tests show that our decision to use Jacobi was correct.

5.4 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 4. The speedup is exactly as we expected. At the beginning the performance scales very well 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 parts. 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

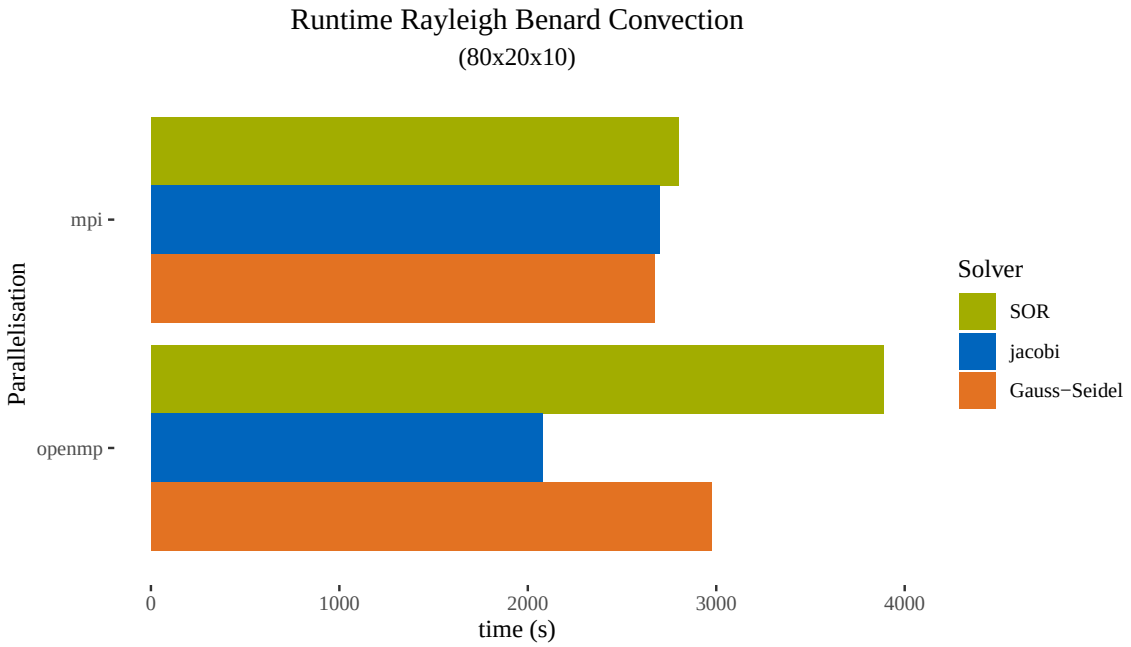

Figure 3: All cases tested on a E2-2697 with 28 cores.

cores altogether, the performance scales almost linearly with an speedup of approximately 19, even though the domain is split in 7x4x2 parts, which is not optimal. The results of the MPI speedup can be seen in figure 5.

6 Arbitrary Geometries

6.1 u,v,w

Unlike in the 2D cases, where it is necessary to set the arbitrary geometry boundary conditions for 8 directions (4 lines + 4 corners of a square), there are a total of 26 directions (12 edges, 8 corners and 6 faces of a cube) to be set. Setting the boundary values in 3D with the same method as utilized in worksheet 2 proved to be difficult. Therefore, we designed a different approach to set the boundary conditions for arbitrary geometries.

We observed that we could set velocities lying right on the boundary to zero. Moreover, for the other values, the boundary value being zero can be achieved by averaging the values on both sides of the boundary. This is an analogous case as when setting no-slip boundary condition on walls. We chose to set u, v and w separately. The direction flag is used to check if the velocity is lying on the wall. If it is, the velocity is set to zero. Then an indicator is inserted that this velocity may not be overwritten anymore. Otherwise, the velocity is set to equal the negative of the neighboring velocity. For example, below is pseudo code of the arbitrary geometry boundary conditions of u on the left wall.

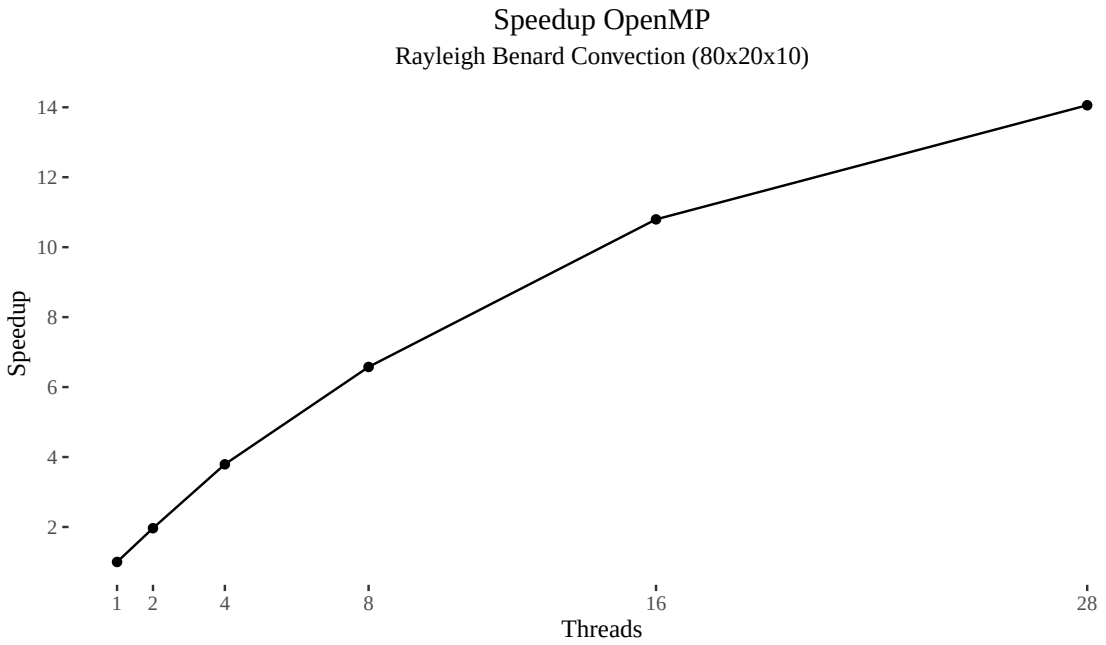

Figure 4: All cases tested on a E2-2697 with 28 cores.

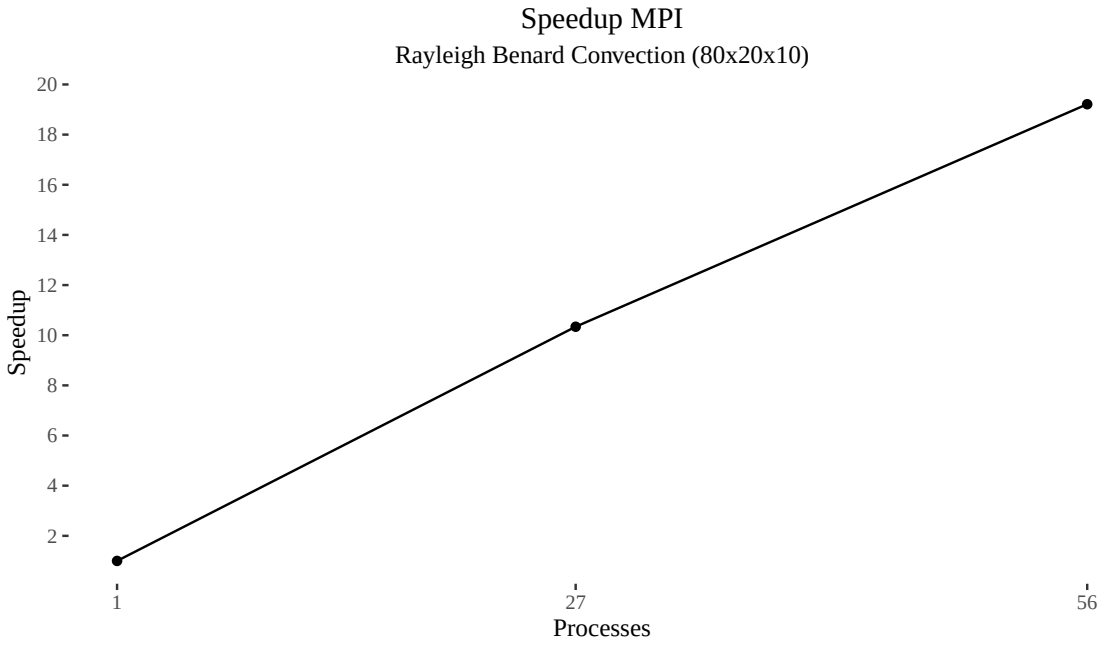

Figure 5: All cases tested on one or two nodes using a E2-2697 with 28 cores.

Algorithm 1 Algorithm on the left boundary

for over i,j,k **do**
  $a \leftarrow 0$
  $b \leftarrow 0$
  if Flag is not fluid **then**
    if Left flag is obstacle **then**
      $u_{i-1,j,k} \leftarrow 0$
      $a \leftarrow 1$
    if Up flag is obstacle **then**
      if $a == 0$ **then**
        $u_{i-1,j,k} \leftarrow -u_{i-1,j+1,k}$
        $b \leftarrow 1$
    if Down flag is obstacle **then**
      if $a == 0$ **then**
        $u_{i-1,j,k} \leftarrow -u_{i-1,j-1,k}$
        $b \leftarrow 1$
    if Back flag is obstacle **then**
      if $a == 0$ and $b == 0$ **then**
        $u_{i-1,j,k} \leftarrow -u_{i-1,j,k-1}$
        $b \leftarrow 1$
    if Front flag is obstacle **then**
      if $a == 0$ and $b == 0$ **then**
        $u_{i-1,j,k} \leftarrow -u_{i-1,j,k+1}$
        $b \leftarrow 1$

6.2 Pressure and Temperature

When setting the arbitrary geometry boundary conditions of pressure and temperature, the same methodology is used. As the pressure and temperature at the direction are the average of the values of the fluid neighbors involved, for example, B_URF (up, right and front)'s pressure is $P_{ijk} = (P_{i+1jk} + P_{ij+1k} + P_{ijk+1})/3$. So each flag of direction is checked separately and the values of the neighboring fluid cells are averaged.

Algorithm 2 Algorithm for pressure boundary conditions

```
for over i,j,k do
    numDirectFlag ← 0
    P_temp ← 0
    if Flag is not fluid then
        if Left flag is obstacle then
            P_temp+ = P_{i+1,j,k}
            numDirectFlag++
        ...
        P_{i,j,k} = P_temp/numDirectFlag
```

References

[1] M. Griebel, T. Dornseifer, and T. Neunhoeffer. SIAM, Philadelphia, 1998. Numerical Simulation in Fluid Dynamics, a Practical Introduction.

[2] NVIDIA. 2019. CUDA C Programming Guide. `https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#arithmetic-instructions` visited on 2019-07-18.

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