Batuhan Erden, Erfan Mashayekh, Judith Pauen, Nigel Bruce Khan

Report Worksheet 2

Merge Request: https://gitlab.lrz.de/erdenbatuhan/hpc_turbulence_code_skeleton/-/merge_requests/2

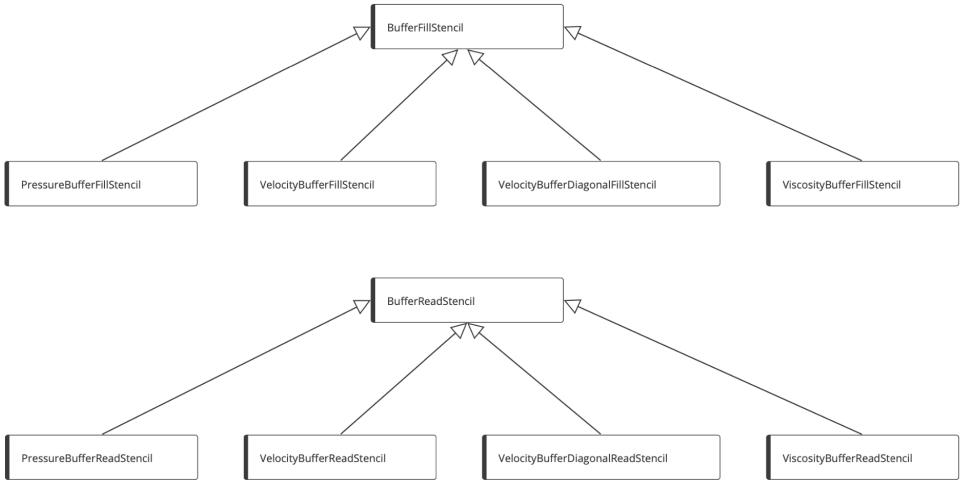
Communication of Flow Quantities

After updating the structure of the existing VTKStencil to make it ready for parallelization, we started by thinking about the ways of exchanging the pressure and the velocity values, later also the viscosity values, between the processes as the first part of the parallelization task. We have decided to use *MPI_Sendrecv* for all communications, and it was required for us to carefully implement buffer fillers and readers.

The subdomains were divided in such a way that on the boundaries, there were values that we needed to communicate, which requires the action of filling and reading the buffers. In our code, we decided to work with *vectors* as it would have led to cleaner code, especially in the read stencils. That is, working with *vectors* provided us with access to *vector iterators* where at each step, we could read the next values in the iterators without even worrying about the lower and upper bounds.

Implementing the stencils and working with them, we have implemented different logics for both that required us to implement the construction, destruction, access and modification operations for the *vectors* and *their iterators*. This is where we have used the first inheritance in our project. By creating a parent class "*BufferFillStencil"*, for *BufferVelocityFillStencil* and *BufferPressureFillStencil, and* a parent class "*BufferReadStencil"*, for *BufferVelocityReadStencil* and *BufferPressureReadStencil,* we were able to apply abstraction and encapsulation for all of the common function. With the help of the inheritance, one basically needs to implement only the *apply* functions. The inheritance relationship can be seen in the diagram below.

Hint: Not only did this method help us create new *Fill* and *Read* stencils in a simpler way without worrying about the baggage of the functionality, but it also prevented us from doing code duplication, which would increase the *technical debt* if this project ever got deployed.



Global Synchronization of the Time-step Size

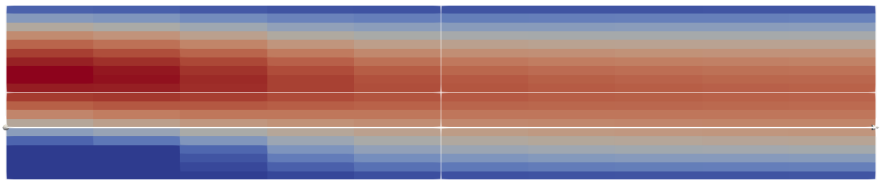
In the parallel simulation, the computational domain is divided into Px × Py × Pz number of subdomains, in which the simulations take place separately. Each subdomain computes a local time step related to its domain. As we need a time step obtained from the velocity values of the whole domain, we need to "Gather" all of these values in one rank and compute the minimum of all of them and then "Scatter" the calculated time step back to all ranks. This process can be done using MPI_Allreduce function in MPI.
Firstly, the maximum time-step size is evaluated for each subdomain, e.g. each processor. Afterwards, the minimum of all these values is chosen to be the global maximum time step.

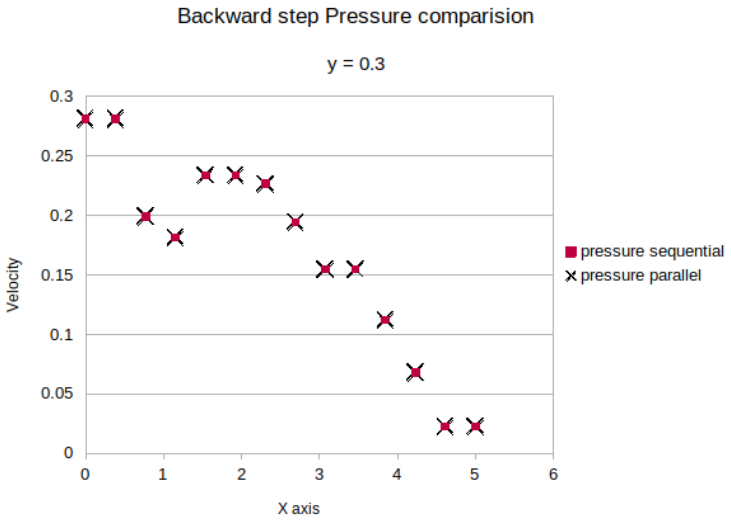
Task 2.3: Parallelization Validation

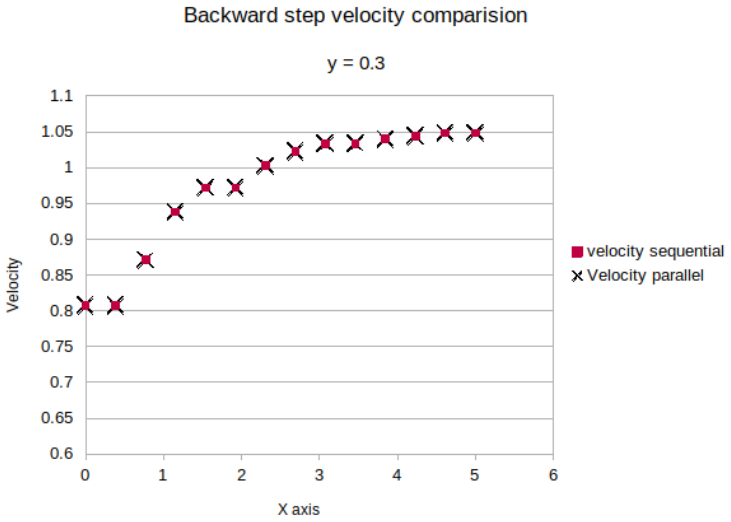
Validate your parallel implementation with different choices Px × Py × Pz, and domain sizes. Use cavity, channel and backward-facing step simulations for this purpose. Compare the solutions to your sequential program. What do you observe?

We compared the parallel code with different settings over X and Y directions in two and three-dimensional cases. It seems that there is a good agreement between the parallelized version and the sequential NS-EOF code. In the following, velocity and pressure in the two-dimensional backward-facing step case at Y=0.3 over X axis are being compared with the sequential code. The parallelization conduction with four processes is a 2x2 grid.

DNS Parallel Simulation: compute pressure and velocity at the white section







Towards Scaling Experiments

With the Channel 2D example case and domain size equal to 100x200, the speed up for 2 processors, in the x-direction, that divides the grid into a right and left half, is about 1.6 times the sequential, which is relatively good. However, the speed up is only 1.1 when we have 4 processors, with 2 in the x direction and 2 in the y direction. This did not make sense because the communication is lesser while dividing the grid into a top and bottom part due to there being twice as many y-grid points as there are in the x direction, so the dramatic drop in efficiency should not have been caused just by the introduction of the 2 processors that further divide the domain into a top and bottom half. We tested another grid size 50x100 and saw that the grid again the x-splitting of the domain and noticed that the speed up is more with the same number of processors limited to only the x-splitting of the domain. We think this is because of two things, one, the domain is much larger in the x side than the y-side so splitting left to right causes less communication than top-bottom splitting. Secondly, The computation to communication ratio increases as the grid points become larger, which is also noticed by the less dramatic drop in speedup in the table, so the speed up trend would be better if we used more grid points.

From the weak scaling we can see that increasing the x-grid points is much more efficient than increasing the y grid points. Physically since the change in the velocity and other parameters is more important in the y-direction we should have either a stretched mesh or more grid-points in the y-axis. However, doubling the grid points in the relatively smaller geometric side while splitting into top-bottom not only increases the computations drastically but also increases the communication needed exponentially. This is the reason why we think the weak scaling falls considerably as soon as more grid points and domain splitting in the y-direction is introduced. Thus, it takes less time to solve the problem in the x-direction than in the y-direction.

The solution could also have poor weak scaling due to the conjugate gradient solver used by PETSC. According to the paper “Simulation of Nf=2+1 Lattice QCDat Realistic Quark Masses” by Y. Nakamura et.al.[1], the conjugate gradient has very poor weak scaling in their matrix inversion tasks at least, and it could also apply to our iterative calculations as well.

Scaling the general Navier-Stokes equation is hard because it is spatially dependent, both in the quantity in the material derivative and in the strain tensor. Not only does parallelizing depend on the size of the geometry but greatly on the shape and also the type of problem. For example, for a flow near a plate the grid needs to be smaller near the plate and larger further away because of the variations in the fluid more near the plate. Or depending on the shape or Reynold's number the flow could be varying more in one direction compared to the other, thus it's hard to find an optimal way to divide the domain and a wrong division can cause the solver to have convergence problems which leads to greater time needed to solve the problem even with more processors. Thus, it is hard to have good weak and strong scaling in the Navier-Stokes equations.

Strong and weak scaling

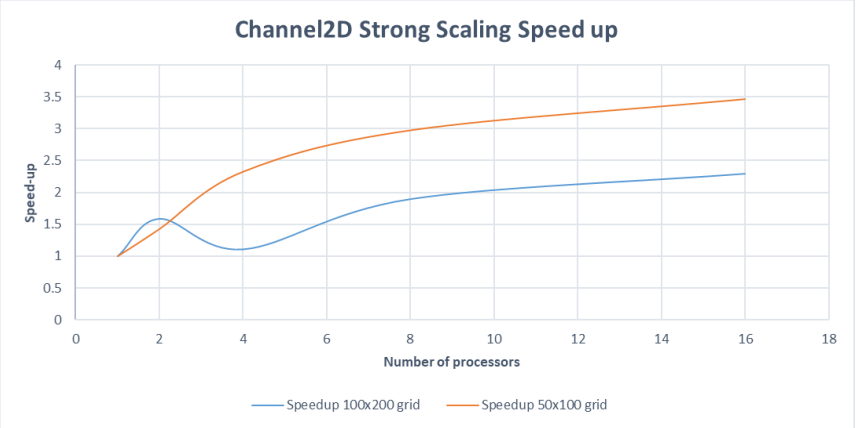
We chose the cavity and the channel scenario for scaling measurements because the channel-backward step faces accuracy and convergence errors in algebraic models and because the cavity is symmetric in grid points and shape in all directions. The channel was chosen because it shows us the effects of different grid divisions due to the shape of the domain. When dividing the grid, it is possible to distribute the same number of fluid cells to each of the processors in these two scenarios but with the backward facing step it would be a complicated task. For the strong scaling, we also chose a high number of cells, which resolves the solution quite well. For weak scaling we started with a coarser resolution to restrict the simulation time and see the changes over a large range as we increase the grid points.

We ran the code in 3D as well to test for different combinations of grid decompositions. For the 3D cavity simulation, we also chose a coarse resolution to restrict the simulation time. A good indication of how the scaling works is also able to be obtained with a coarse resolution in the cavity cases. The graphs and tables obtained for speedup and parallel efficiency with weak and strong scalings are shown below. The graphs are arguably not as important as the tables because the tables give more information on how the grid points were changed and in which direction the domain decomposition for parallelization was done but they do tell us the trends of the speedup and efficiency in a visual manner. For the weak scaling, since we could not find how much of the code percentage was parallel we used a formula that uses the time taken with 1 processor on a grid versus n-processor on a refined grid to find the percentage efficiency in the weak scaling.

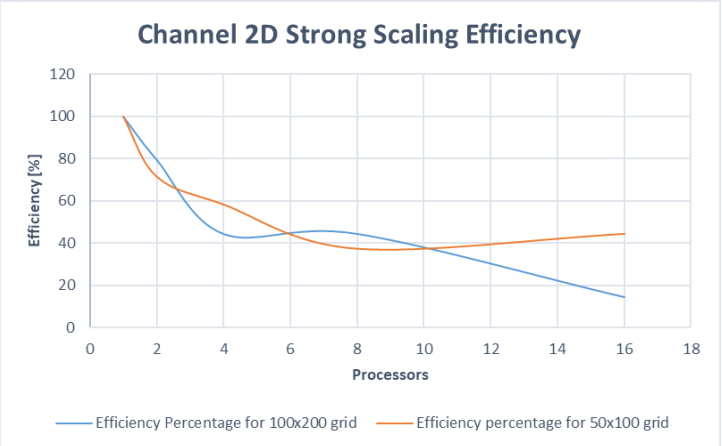
Performance can be improved by analyzing these tables and seeing what domain decomposition causes the least simulation time in a certain scenario without compromising on accuray. Optimizing the resolution of the grid to only refine the areas with small movements as the region after the step in the backward facing scenario or having a stretched grid would also improve performance for the channel cases. The code could be optimized by trying to reduce the blocking conditions, superior domain decomposition algorithms according to the geometry. Analyzing the code using gprof also revealed that the function "NSEOF::Stencils::mapd(int, int, int, int)" takes up the most time (19% in a sequential run), could also be made parallel and be written better. Finally, the PETSC solver itself could be written better as will be done in the project for this course by using Eigen.

Index	Length [x]	Height [y]	Processors[x,y,z]	Time [secs]	Speedup [t1/tN]	Efficiency [speedup/processors*100%]
1	100	200	1,1,1	410.699	1	100
2	100	200	2,1,1	258.753	1.59	79.5
3	100	200	2,2,1	368.586	1.11	27.6
4	100	200	4,2,1	215.733	1.9	23.8
5	100	200	4,4,1	178.474	2.30	14.38
1	50	100	1,1,1	21.886	1	100
2	50	100	2,1,1	15.357	1.43	71.5
3	50	100	4,1,1	9.3733	2.33	58.3
4	50	100	8,1,1	7.350	2.98	37.3
5	50	100	16,1,1	6.305	3.47	21.7

Table 1: Channel 2D Strong Scaling



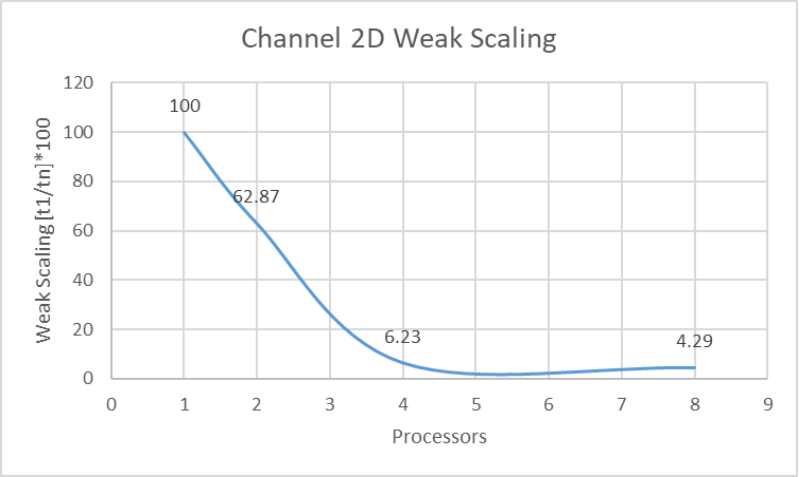
Strong scaling speedup for Channel 2D



Strong scaling Efficiency for Channel 2D

Index	Length [x]	Height [y]	Processors [x,y,z]	Time [secs]	Weak Scaling percentage [t1/tn*100%]
1	50	100	1,1,1	22.994	100
2	100	100	2,1,1	36.573	62.87
3	100	200	2,2,1	368.586	6.23
4	200	200	4,2,1	535.768	4.29
5	200	400	4,4,1	cluster stopped working here (PETSC problem)	

Table 2: Channel 2D Weak Scaling

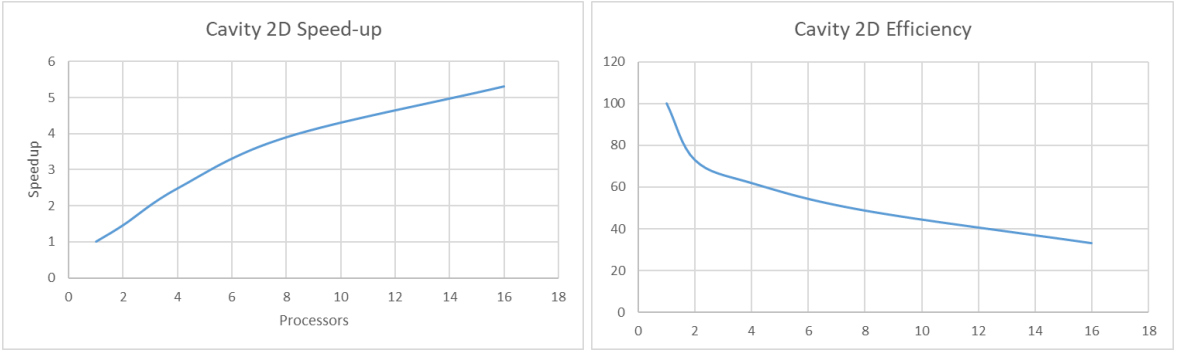


Weak Scaling for Channel 2D

When we increase grid points in x, we should increase processors in the x side as well i.e. divide the grid in half vertically. This limits the communications needed, while trying to keep the computation to number of processors ratio constant.

Index	Length[x]	Height[y]	Processors[x,y,z]	time [secs]	Speedup[t1/tN]	Efficiency (speedup/processors)*100%
1	100	50	1,1,1	33.321	1	100
2	100	50	2,1,1	22.8578	1.46	72.88
3	100	50	2,2,1	13.456	2.48	61.91
4	100	50	4,2,1	8.544	3.90	48.75
5	100	50	4,4,1	6.279	5.31	33.17

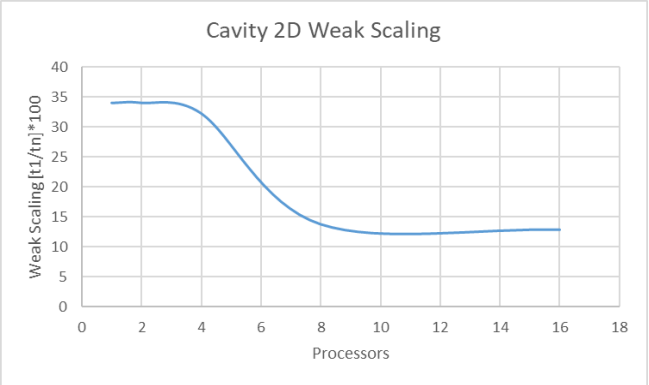
Table 3: Cavity 2D Strong Scaling



Strong Scaling speedup and efficiency for Cavity 2D

Index	Length [x]	Height [y]	Processors[x,y,z]	time [secs]	Weak Scaling percentage [t1/tn*100%]
1	100	50	1,1,1	33.320855	33.96
2	200	50	2,1,1	98.114527	33.96
3	200	100	2,2,1	103.162006	32.20
4	400	100	4,2,1	243.249895	13.70
5	400	200	4,4,1	260.13107	12.80

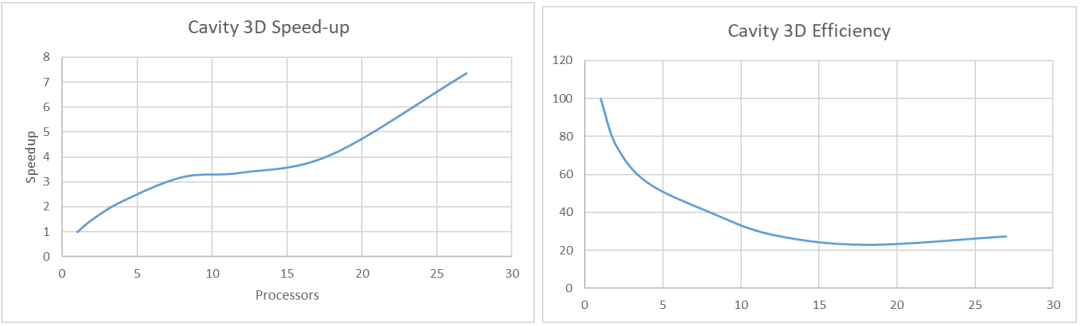
Table 4: Cavity 2D Weak Scaling



Weak Scaling for Cavity 2D

Index	Length [x]	Height [y]	Width [z]	Processors [x,y,z]	time [secs]	Speedup[t 1/tN]	Efficiency (speedup/proces sors)*100%
1	20	10	20	1,1,1	12.197	1	100
2	20	10	20	2,1,1	8.121	1.50	75.10
3	20	10	20	2,2,1	5.479	2.23	55.65
4	20	10	20	2,2,2	3.811	3.20	40.00
5	20	10	20	3,2,2	3.612	3.38	28.14
6	20	10	20	3,3,2	2.967	4.11	22.84
7	20	10	20	3,3,3	1.658	7.36	27.25

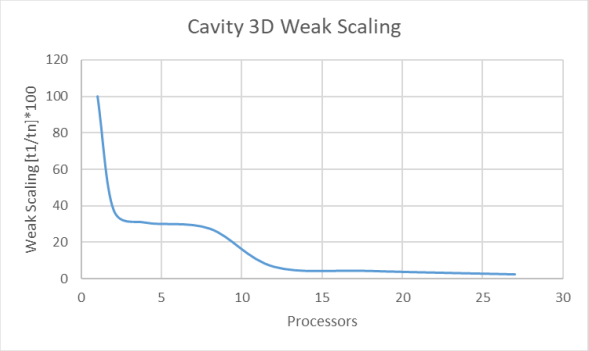
Table 5: Cavity 3D Strong Scaling



Strong Scaling speedup and efficiency for Cavity 3D

Inde x	Length [x]	Height [y]	Width [z]	Processors [x,y,z]	time [sec]	Weak Scaling percentage [t1/tn*100%]
1	20	10	20	1,1,1	12.197	100
2	40	10	20	2,1,1	32.305	37.75
3	40	20	20	2,2,1	39.670	30.75
4	40	20	40	2,2,2	44.467	27.43
5	80	20	40	3,2,2	186.303	6.55
6	80	40	40	3,3,2	285.224	4.27
7	80	40	80	3,3,3	513.314	2.38

Table 6: Cavity 3D Weak Scaling



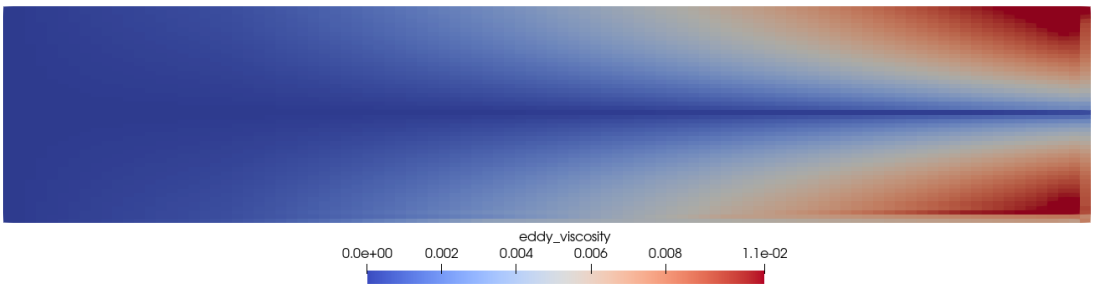
Weak Scaling for Cavity 3D

Implementation of algebraic turbulence model

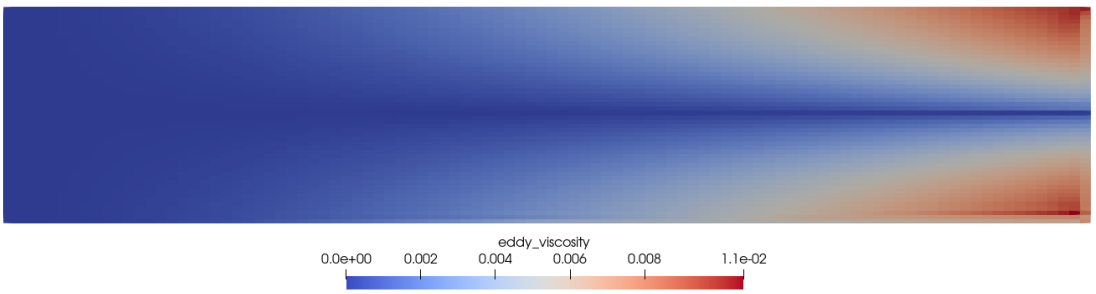
For the algebraic turbulence model, we implemented the mixing length calculated by $l_m = \kappa h$ and named it model 0. The computation of the boundary thickness assuming a laminar flat plate is named model 1 and the boundary thickness assuming a turbulent flat plate is named model 2. In our opinion, extracting the thickness of the boundary layer from a laminar reference case is an inexact model, therefore we did not implement it. Model 0 is converging very slowly, which is why we prefer the models 1 and 2.

For model 1, we would expect a greater boundary thickness than for model 2. A turbulent boundary layer is smaller than a laminar one because the turbulent velocity profile is more flat, the velocity gradient is bigger and the angle between the wall and the flow is smaller than for the laminar case.

The eddy viscosity for model 1 at a stationary state:



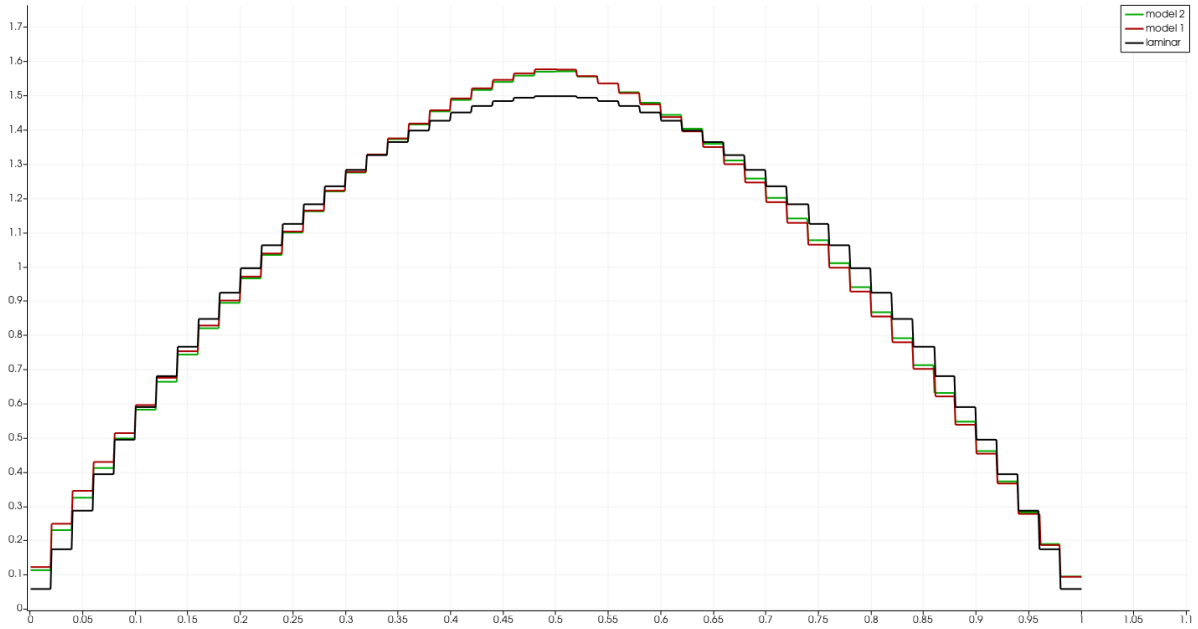
The eddy viscosity for model 2 at a stationary state:



When comparing the pictures of eddy viscosity for model 1 and 2, it can be seen that the expectations are fulfilled. At the end of the channel, the boundary layer for model 1 is bigger than for model 2.

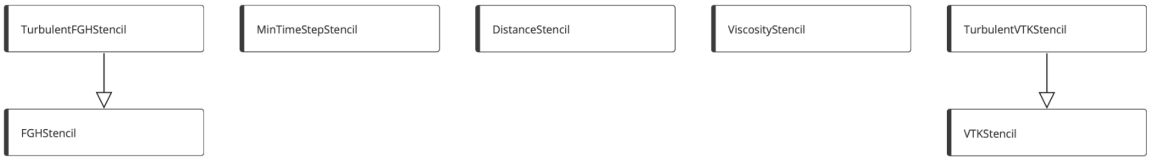
In the graph below, the velocity profile of the channel scenario in 2D at x=4.5 can be seen. The black line is the one for the laminar simulation, the red for model 1 and the green line for model 2. For the turbulence models, the profile overlaps, whereas the laminar profile reaches a smaller maximal velocity and is more flat. This result is different than expected, because a turbulent profile is more flat than a laminar. But this effect can depend on the growth of the boundary layer for the turbulence model to the end of the channel which happens only for the turbulence models in the simulation. In reality, this should happen also to the laminar simulation case.

The two coarse resolution is the reason for the steps in the graph below.



Turbulent viscosity and wall distance

We have extended our data structures for the turbulent case. As can be seen in the diagram below, we have implemented new data structures among which both of them inherit from already existing classes because we are reusing most of the functionalities of the parent classes (TurbulentFGHStencil and TurbulentVTKStencil).



The calculation of the distance function was easy to implement for the sequential Channel and Cavity scenarios. To capture the step in the backward facing step scenario, for every cell the surrounding mesh with Nx/2 in x-direction and Ny/2 in y-direction has to be investigated to find the nearest wall. Therefore, this special case was neglected for the distance calculation. When parallized, the distance function caused some problems for the top wall. Somehow the information of the wall was not communicated within the processors.

Momentum equations

The stretched mesh is needed especially for the channel models to have better resolution towards the walls where the flow tends to have more interesting flows. However, we could not find the time to implement the stretched grid on all the derivatives in the StencilFunctions.hpp file. There are two algorithms to find the new derivatives for ∂uv/∂y for example:

1. (((hxLong-hxShort)/hxLong*v00 +hxShort/hxLong*v10) *
   ((hyLong1-hyShort)/hyLong1*u00 +hyShort/hyLong1*u01) -
   ((hxLong-hxShort)/hxLong*v0M1 +hxShort/hxLong*v1M1)
   *((hyLong0-hyShort)/hyLong0*u00 +hyShort/hyLong0*u0M1)) / (2.0*hyShort);

Where hxLong and hxShort are the x-edges of two adjacent cells, hyLong1 and hyShort are the corresponding y-edges, v00, u00, v10, u01 are the v-velocity and u-velocity in the current cell and v10, u01 are the v- and u- velocities in the next x-direction cell, and v0M1 and v1M1 are the v-velocities in the middle of the x-edge and the y-edge. The formula basically, scales the partial derivatives by expanding them into separate terms and scaling them according to their lengths so that the sides further away have less effect on the derivative of the term and vice versa.

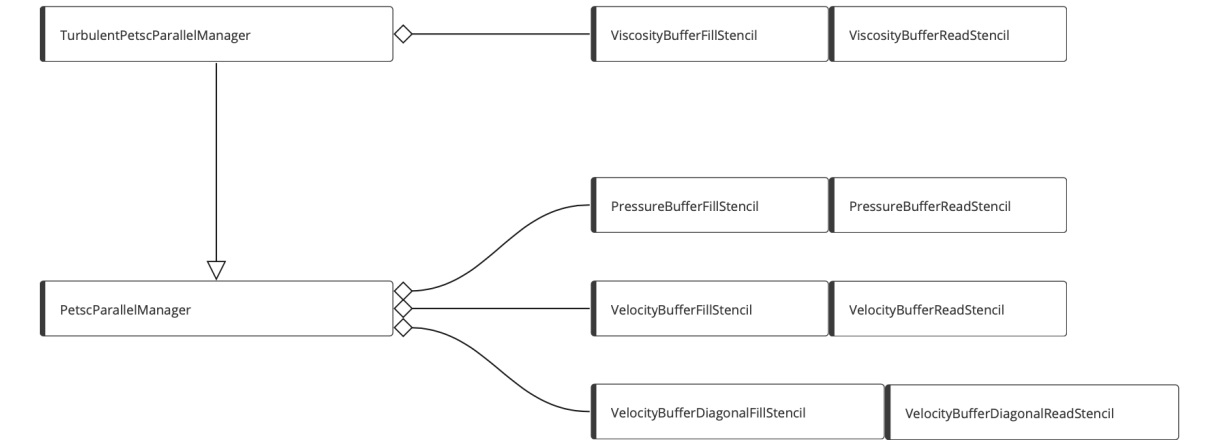
2. The second method would use the formula for the stretched grid given in the tanh grid stretching function "*computeCoordinate()*" in the file Meshsize.hpp that converts a uniform grid into a stretched one. According to the resource, "REVIEW Lecture 21: • End of Time-Marching Methods: higher-order methods -Runge-Kutta Methods, n.d" [3] we would first invert the formula in *computeCoordinate()* to find the normal coordinate (**σ**) with respect to the stretched one(y) i.e. σ(y) and then compute the derivatives according to the product rule as:

   ∂ρu/∂y = (∂ρu/∂σ)*(∂σ/∂y)

The formula above would thus enable us to calculate the derivatives on the stretched grid.

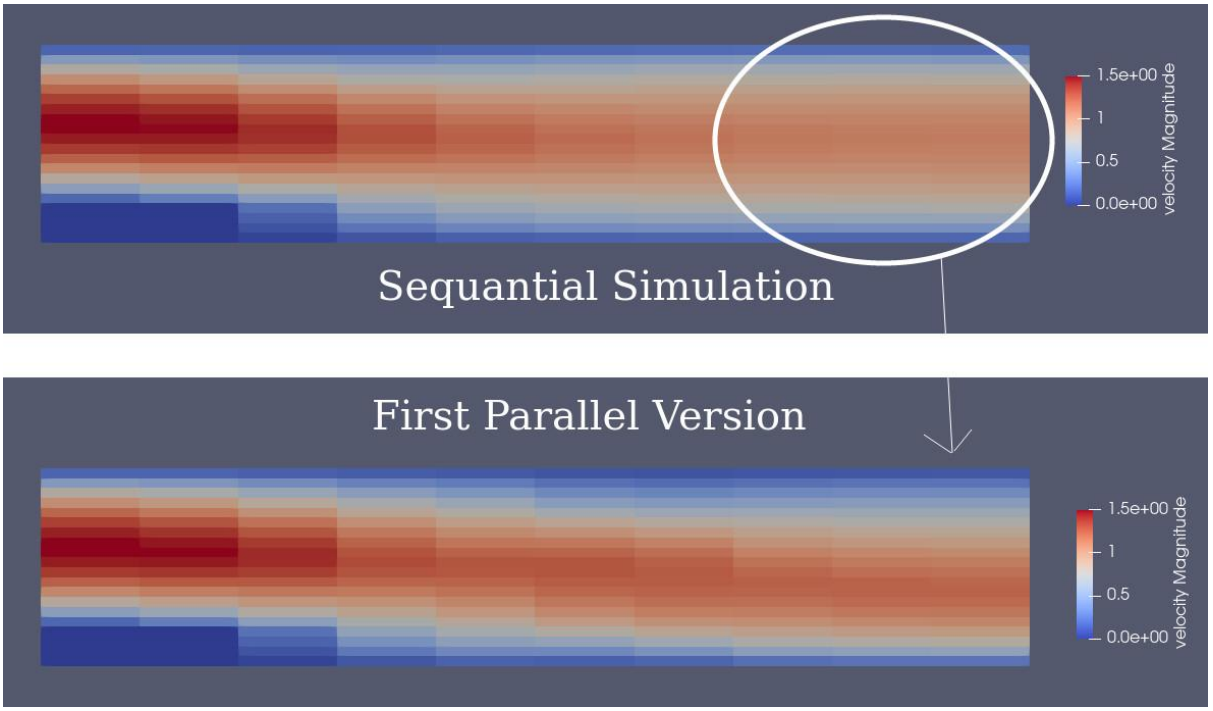
Parallelization

In the parallelization of the turbulence model, we first tried to refactor PetscParallelManager to create a common function for communicatePressure and communicateVelocity so that we would not have code duplication. In doing so, we have created an easy road for applying inheritance. For the parallelization of the turbulent model, we have created the TurbulentPetscParallelManager class that basically handles only the communication of viscosity, but also inherits the other communications from PetscParallelManager.



As one can see in the graph above, we have 2 extra classes that we did not mention, VelocityBuffer**Diagonal**FillStencil and VelocityBuffer**Diagonal**ReadStencil.

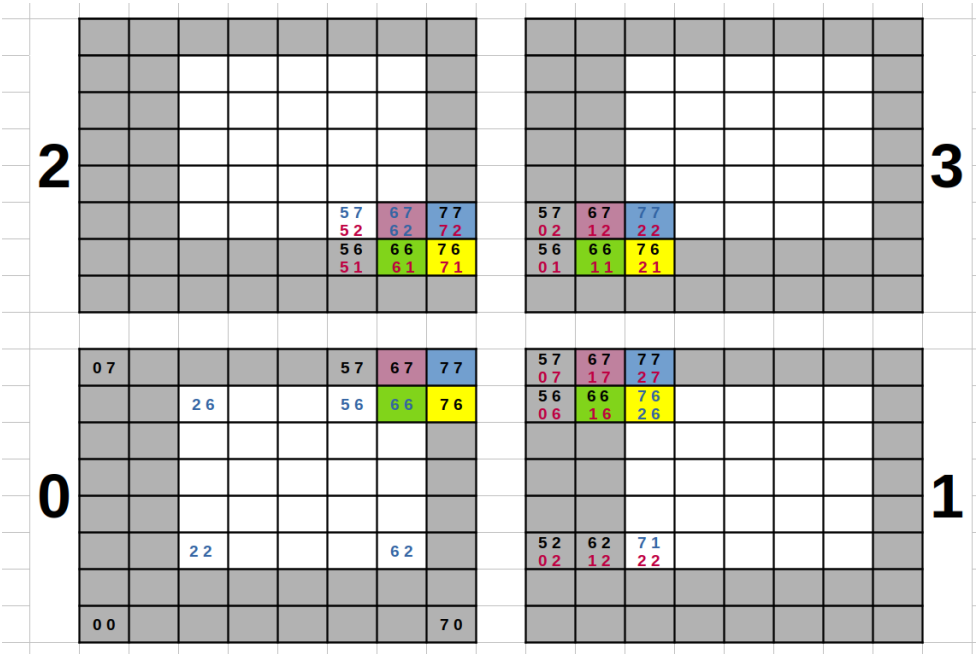
In the first attempts, we could not get good results for parallelization in turbulent simulation. In this case, we used our parallel manager class to communicate viscosity values after each iteration. However, we realized a problem with parallelization in two or more directions, and the results are different from the sequential turbulent reference. In other words, there was not a problem for parallelization, including 4x1x1 or 1x2x1, but in the 2x2x1 or 2x2x2.



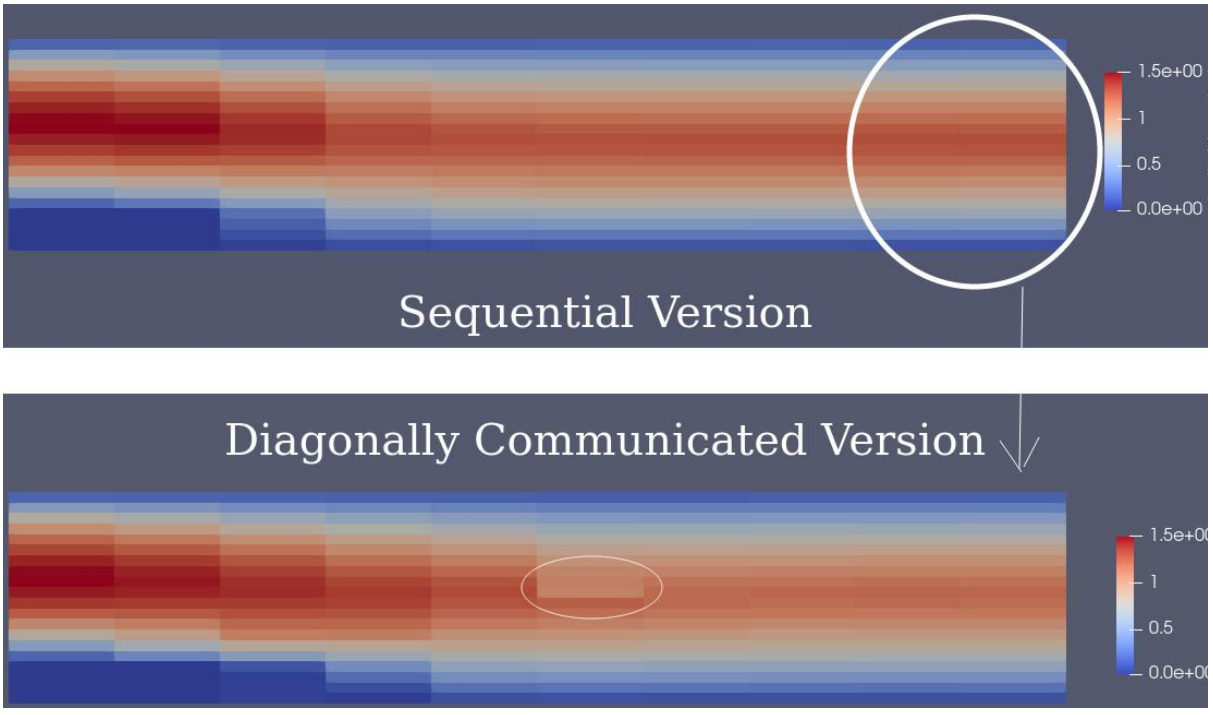
Backward step case for a 2x2 parallelization first version

Finally, we understood that in computations of FGH stencil at each grid point in the turbulent simulation, the number of neighbor velocities is much more than in the DNS simulation.

Therefore, we had to communicate velocity values even diagonally in intersections with top or bottom subdomains.



Although there are some minor problems in the diagonally communicated version, this code has almost good agreement with the sequential turbulent simulation. As can be seen there is a small mistake in one of the diagonally communicated velocities that must be solved.



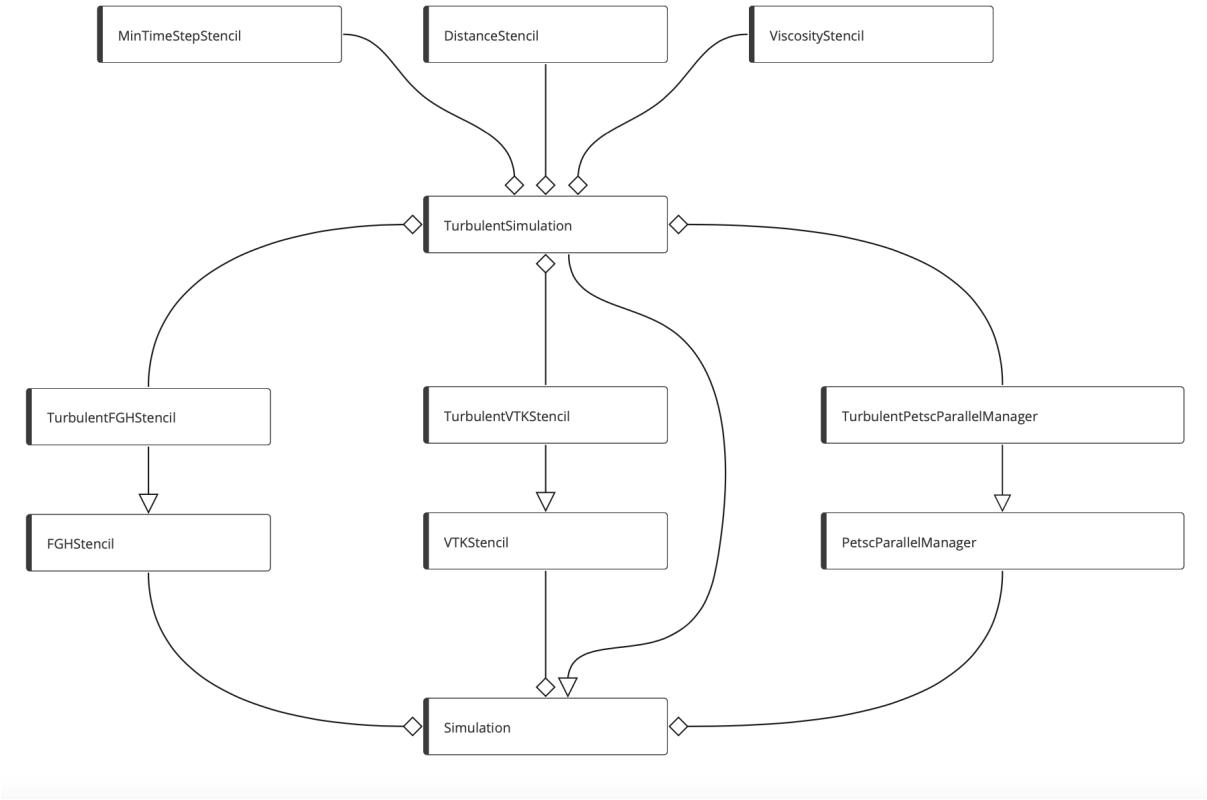
Backward step case for a 2x2 parallelization diagonally communicated version

Plugging things together

Now that we have everything ready, it is time to put all of these together. We have created TurbulentSimulation as a child class of Simulation in order for it to inherit everything from DNS simulation, and have the extra things needed for the turbulence model, such as minimum time step, distance and viscosity. TurbulentSimulation only overrides the necessary functions of Simulation and most of the Simulation class seems intact.

We choose between Simulation and TurbulentSimulation in the main file according to the simulation type given in the input files.

The relationships between the classes is demonstrated in the diagram below, where the arrows are the *"Inheritance"* paths and the diamonds are the *"Aggregation"* paths (Just like in the diagrams above).



The new time step is calculated similar to the original time step. The formula (12) of worksheet 2 is changed to:

$$\delta t < \frac{Re}{2}\left(\frac{1}{\delta x^2}+\frac{1}{\delta y^2}+\frac{1}{\delta z^2}+\frac{1}{\delta x\,\delta y}+\frac{1}{\delta x\,\delta z}+\frac{1}{\delta y\,\delta z}\right)^{-1}$$

The Reynolds number is computed as the inverse of the total viscosity (kinetic plus turbulent) via $Re = \frac{1}{\nu+\nu_T}$. The terms added in the brackets are the results of the added

terms in the FGH formulas for turbulent simulation. The MinTimeStepStencil calculates the maximal time step of each cell and in the end, the minimal value of these is chosen as the overall time step.

Testing

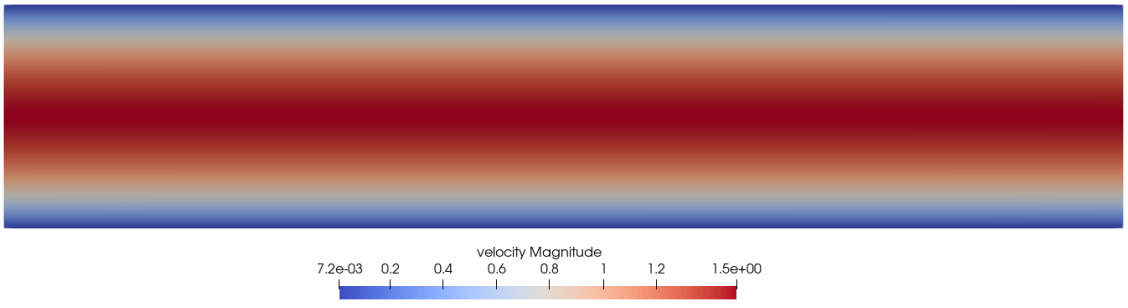
- parallel simulation of different channel flow settings
- verification of all finite difference expressions and parallel extensions
- experiment with different turbulence models. observations? Behaviour?
- backward-facing step

The verification of the finite difference expressions was done using chapter 6 of the textbook ‘Numerical Simulation in Fluid Dynamics, A Practical Introduction [2]. It was also done visually through paraview software by increasing the grid points till visual resolution and by printing the results and comparing to hand calculations for comparison. A mistake in the stencil for the FHG-terms leads directly to an unphysical behavior of the flow compared to other experiments or other flow simulations stated in literature and usually due to problems in the FGH stencils, even slight ones, the solver tends to diverge or not reach convergence in a suitable timestep.

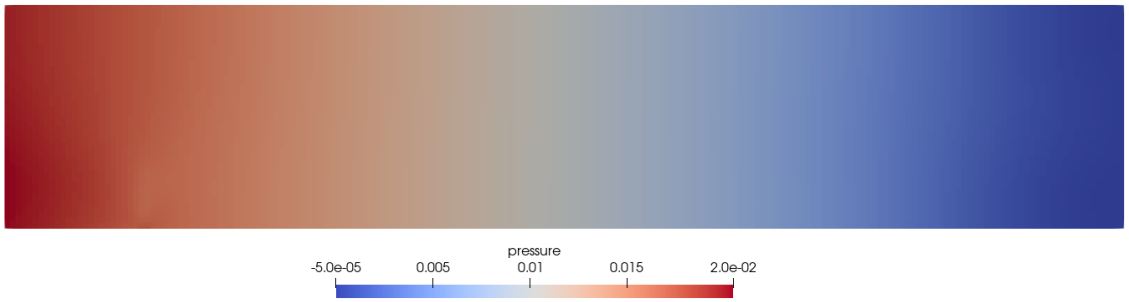
At the cluster, we simulated the channel as well as the backward facing step scenario in 2D and in 3D with a high resolution. Unfortunately, we had many issues running on the cluster. The code delivered good results without any errors at our own machines, whereas at the cluster communicating between the processors did not work well. The chosen turbulence model was not communicated well and the eddy viscosity output in the vtk files was only done for rank 0. Due to time issues, we were not able to find the error.

But we have been able to gain results with a high resolution. In the simulation of the backward facing step, the vortices directly after the step and at the top end of the channel can clearly be seen.

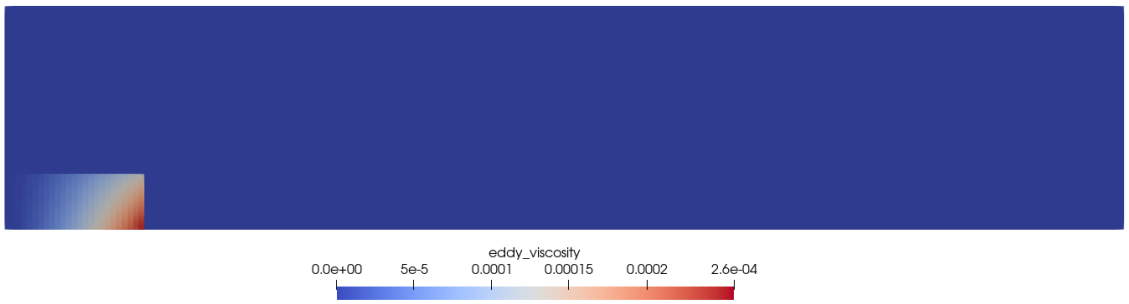
Velocity field of Channel 2D with Re = 3000 and 200 cells in x-direction and 400 in y-direction:



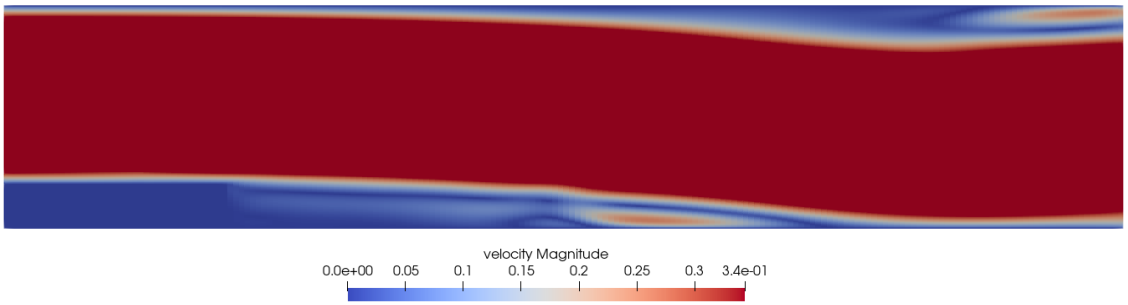
Pressure field of Channel 2D with Re = 3000 and 200 cells in x-direction and 400 in y-direction:



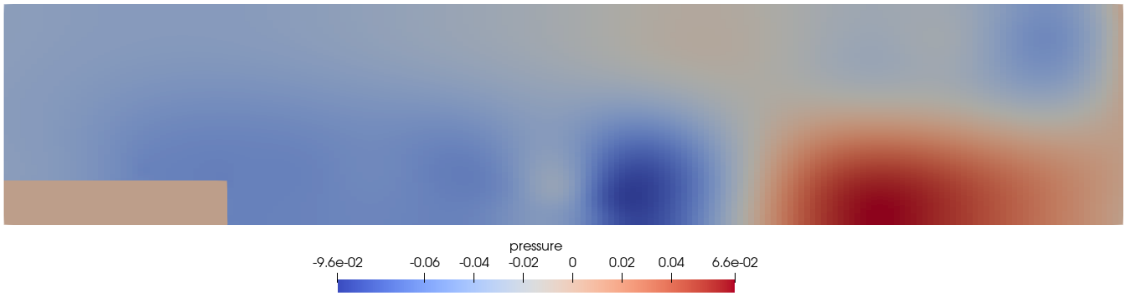
Eddy viscosity for Channel in 2D with Re = 3000 and 200 cells in x-direction and 400 in y-direction:



Velocity field of Channel Backward Facing Step in 2D with Re = 3000 and 200 cells in x-direction and 400 in y-direction:



Pressure field of Channel Backward Facing Step in 2D with Re = 3000 and 200 cells in x-direction and 400 in y-direction:



Team Organization

Our team was organized mainly through our group on whatsapp. Coding was done in an agile working condition. We solved arising problems immediately, some of which have git issues on GitLab, without focusing on a fixed time table. Also, we had regular online meetings. There we reported the things we achieved since the last meeting and talked about the further tasks we have to do. As well we splitted them regularly to give everyone their own responsibility. The tools needed were Trello, Zoom, Whatsapp, VirtualBoxes, different IDEs and Git as the main version control system. The teamwork happened in a friendly atmosphere, where everyone brought themselves in, and we all tried to help each other wherever possible.

A big issue was getting the code to run at the cluster, which consumed much time. Overall, we are happy with our progress and learned much while solving this worksheet.

References:

1. Nakamura, Y., Schierholz, G., Streuer, T., & Stüben, H. (2009). Simulation of N f= 2+ 1 Lattice QCD at Realistic Quark Masses. In *High Performance Computing in Science and Engineering, Garching/M*unich 2007 (pp. 627-638). Springer, Berlin, Heidelberg.
2. Griebel, M., Dornseifer, T., & Neunhoeffer, T. (1998). Numerical simulation in fluid dynamics: a practical introduction. Society for Industrial and Applied Mathematics.
3. REVIEW Lecture 21: • End of Time-Marching Methods: higher-order methods -Runge-Kutta Methods. (n.d.). [online] Available at: https://dspace.mit.edu/bitstream/handle/1721.1/100852/2-29-fall-2011/contents/lecture-notes/MIT2_29F11_lect_22.pdf [Accessed 17 Dec. 2021].