Solution Worksheet 2

Vishal Sontakke, Felix Späth, Michael Zellner, Christopher Zöller

1 Parallelization

We implemented the MPI_Communication via MPI_Datatypes. Our datatypes are the following ones:

```
1      MPI_Datatype _pres_right;  MPI_Datatype _pres_left;
2      MPI_Datatype _pres_top;    MPI_Datatype _pres_bottom;
3      MPI_Datatype _pres_front;  MPI_Datatype _pres_back;
4
5      MPI_Datatype _velo_right;  MPI_Datatype _velo_left;
6      MPI_Datatype _velo_top;    MPI_Datatype _velo_bottom;
7      MPI_Datatype _velo_front;  MPI_Datatype _velo_back;
```

The definition of the datatypes are for example the following ones (in 2D):

```
1
2  MPI_Type_vector(y,
3                  1,
4                  x+3,
5                  MY_MPI_FLOAT,
6                  &_pres_left
7                  );
8
9  MPI_Type_vector(y,
10                 2,
11                 x+3,
12                 MY_MPI_FLOAT,
13                 &_pres_right
14                 );
```

By defining the datatypes like this it is not necessary to fill and read the pressure and velocity buffers. Additionally, an iterator is not necessary anymore.

1.1 Communication of Flow Quantities

- **Check out the implementation of the iterator ParallelBoundaryIterator.**
  Done.
- **Read the implementation and infer its functional meaning: which cells does the iterator operate on?**
  The iterator will move over the boundary layer if there is a neighboring rank available. For each cell it applies a BoundaryStencil. One can specify an offset so it can iterate over inner cells.

- **Which functions does a respective stencil object need to provide?**
  The following functions are necessary because they are defined as virtual.

```
1      void applyLeftWall   (FlowField & flowField, int i, int j);
2      void applyRightWall  (FlowField & flowField, int i, int j);
3      void applyBottomWall (FlowField & flowField, int i, int j);
4      void applyTopWall    (FlowField & flowField, int i, int j);
5      void applyLeftWall   (FlowField & flowField, int i, int j, int k);
6      void applyRightWall  (FlowField & flowField, int i, int j, int k);
7      void applyBottomWall (FlowField & flowField, int i, int j, int k);
8      void applyTopWall    (FlowField & flowField, int i, int j, int k);
9      void applyFrontWall  (FlowField & flowField, int i, int j, int k);
10     void applyBackWall   (FlowField & flowField, int i, int j, int k);
```

- **Can you reuse existing stencil formats, or are new stencil types required?**
  Since every function in BoundaryStencil is virtual we would have to define new ones.
- **Implement a boundary stencil PressureBufferFillStencil which reads the pressure values in each of the six (3D) boundary faces of a subdomain (i.e. the domain of one process) and stores them consecutively in one-dimensional buffer arrays (one array for each of the six faces). For traversal of the respectivesub-domain cells, cf. 1.**
  Not necessary because we are using MPI_Datatypes.
- **Implement a boundary stencil PressureBufferReadStencil which reads data from one- dimensional arrays (one array for each of the six faces) and writes them into the correct cells of the boundary.**
  Not necessary because we are using MPI_Datatypes.
- **Integrate the read- and write operations into a class PetscParallelManager. The Petsc- ParallelManager should provide a method communicatePressure() (...)**
  Our communicatePressure() and communicateVelocity() methods consists of simple MPI_SendRecv operations. Filling and reading buffer is not necessary because the MPI_Datatypes will take care of that.
- **Implement the exchange of velocities analogously to the steps 1-4. Implement write- and read-stencils VelocityBufferFillStencil and VelocityBufferReadStencil and integrate them into the PetscParallelManager. The respective method communicate- Velocities() should further handle the exchange of the velocity buffers between the processes.**
  Done.
- **Integrate the PetscParallelManager and respective calls to its communication methods into the class Simulation. Test the exchange of flow quantities in cavity, channel and backward-facing step flows. It is sufficient to only consider the first time step (the choice of a maximum time step should not have severe impact on the testing).**
  Done. Tests were executed for 2D and 3D cases on every scenario.

1.2 Global Synchronization of the Time Step

- **Check out the implementation of setTimeStep() in the class Simulation. How is the maximum time step evaluated in the parallel simulation?**
  Each rank computes its minimum timestep afterwards the global minimum is derived from those values. MPI_Allreduce(...) takes care of the reduction and broadcast.

1.3 Validation

- **Validate your parallel implementation for different domain decompositions, that is different choices P_x x P_y x P_z , 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?**
  Done.
  Corner cells need to be send, too. These are needed for the derivatives as they contain the values of their right boarder. We tested the dimensional splitting in different 2D and 3D channel flow test-cases, with and without turbulence. The communication works as specified and gives the same results as the sequential code, even the numerical problem in the 3D channel flow is independent from the parallelization. The cavity scenario was tested with 2D and 3D, but without turbulence. We observed, that it is hard to achieve a speedup by adding more MPI-ranks as the number of iterations petsc needs increases dramatically, see next chapter for more details.

2 Scaling and Efficiency

2.1 Theory: Towards Scaling Experiments

- **How can weak and strong scaling be measured for NS-EOF? Which problems do you expect?**
  Strong scaling should be easy to measure by choosing a large scenario. We should be able to increase the number of processors working on the problem in order to achieve a lower execution time.
  Weak scaling will be tricky because the number of iterations of PETSC is dependent on the number of cells. The increase of computational work is therefore higher than the increase of cell-count. A good metrics would probably meassure the time per itreation and include this. We will just keep the number of cells per process constant.

2.2 NS-EOF in the MAC-cluster

2.2.1 Environment and Login

Done.

2.2.2 Compiling

Due to the fact that PETSC is not fully supported by the MAC cluster we compiled our own version for this worksheet. We used the version 3.7.4 the commands to compile it are the following ones:

Listing 1: Debug version

```
wget http://ftp.mcs.anl.gov/pub/petsc/release-snapshots/petsc-3.7.4.tar.gz
mkdir $HOME/petsc
tar -xvf petsc-3.7.4.tar.gz
cd petsc-3.7.4

#maccluster
python2 ./configure \
--with-blas-lapack-dir=$MKLROOT \
--with-mpi=1 \
--with-mpi-dir=$MPI_BASE  \
--prefix=$HOME/petsc/petsc-3.7.4-debug \
--CC=mpiicc \
--CXX=mpiicpc \
--FC=mpiifort \
--CPPFLAGS="-g -O3" \ # -xHOST does not work with debugging=1
--CFLAGS="-g -O3" \
--CXXFLAGS="-g -O3" \
--with-vendor-compilers=intel \
--known-mpi-shared-libraries=1 \
--with-make-np=28 \
--with-batch=1 \
--with-debugging=1 \ # Set to zero for release version
--PETSC_ARCH=petsc-3-7-4-debug

salloc -n 1 -p snb
cd $HOME/petsc/petsc-3.7.4
./conftest-petsc-3-7-4-release

#[Ctr + D]

./reconfigure-petsc-3-7-4-release.py

make PETSC_DIR=$HOME/petsc-3.7.4 PETSC_ARCH=petsc-3-7-4-release all
make PETSC_DIR=$HOME/petsc-3.7.4 PETSC_ARCH=petsc-3-7-4-release install

#in .bashrc
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$HOME/petsc/petsc-3.7.4/lib
export PETSC_DIR=$HOME/petsc/petsc-3.7.4
```

2.2.3 Execution

Done

2.3 Scaling Measurements

- **Choose suitable simulation scenarios for the scaling measurements with respect to domain size, domain decomposition and the flow problem under consideration.**

Strong scaling results:

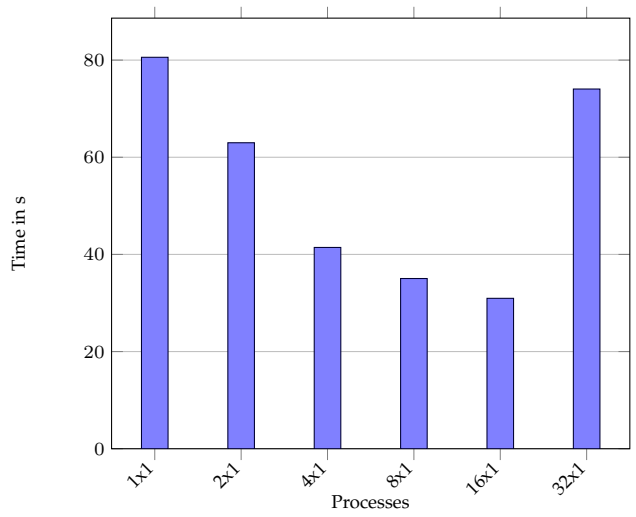


Figure 1: On one Node with domain size 120x120. Increase number of ranks in x-direction

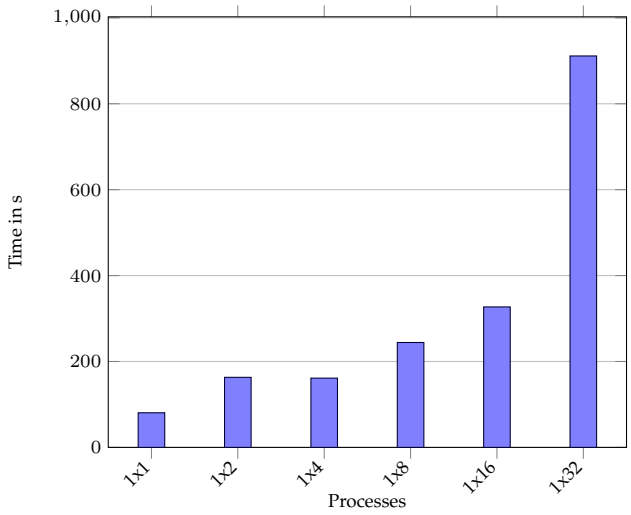


Figure 2: On one Node with domain size 120x120. Increase number of ranks in y-direction

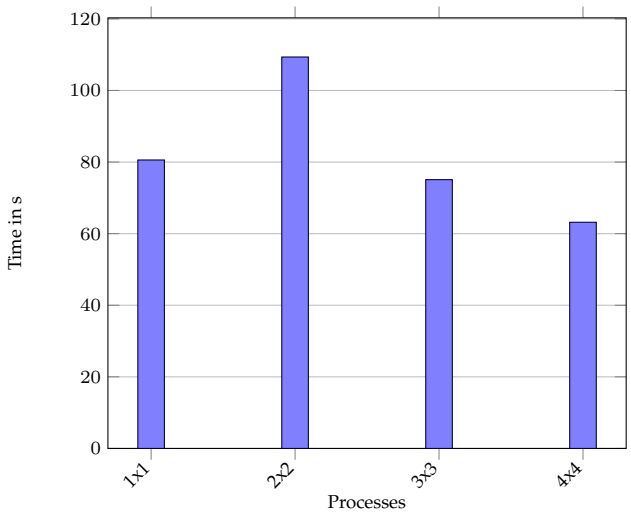


Figure 3: On one Node with domain size 120x120. Increase number of ranks in x- and y-direction

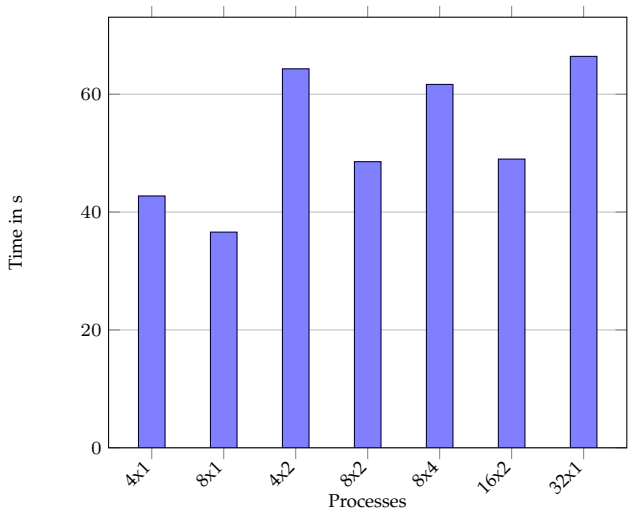


Figure 4: On two Node with domain size 120x120

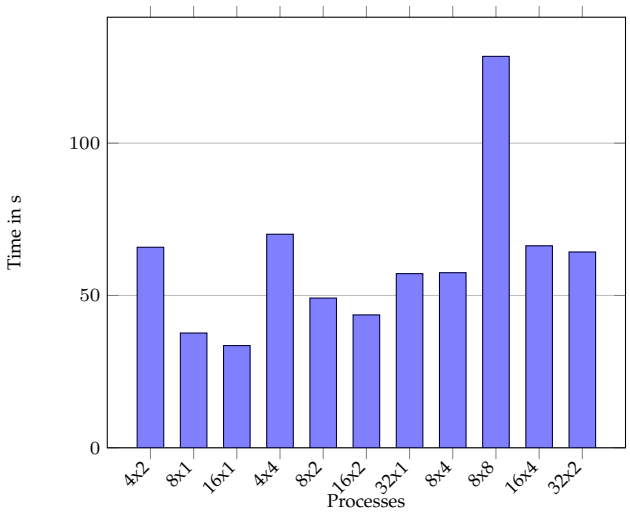


Figure 5: On four Node with domain size 120x120

Weak scaling results:

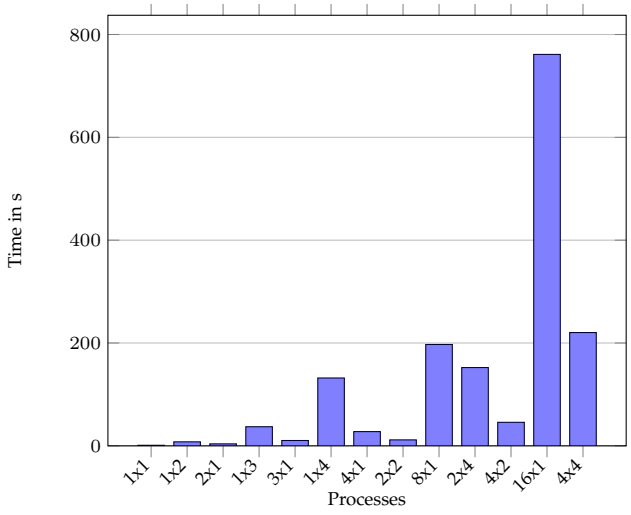


Figure 6: On one node. 40x40 per rank.

- **Run your parallel simulations for various numbers of processors and architectures. Plot your speedup and parallel efficiency in respective graphs (processes vs. speedup, e.g.). Hint: Deactivate vtk output in the scaling experiments since our output is not optimized for parallel execution yet.**
  TODO: Out VTK output is per rank. Moreover maybe we should compare binary and ascii output here

- **How can you improve the performance of your program?**
  Since the Petsc part is the most compute intensive one. One should try to improve this. About 65% of the runtime is spend in Petsc. Probably a better parallelization of this part by using for example OpenMP could help to address this problem. Moreover another Preconditioner could help to decrease the time which is spend in petsc.
- **Optimize your software. You may work on both sequential and parallel improvements as well as on algorithmic optimizations (such as optimization of the pressure Poisson solver).**
  One optimization we did was to use a better implementation for the MPI communication. With our MPI_Datatypes unnecessary fill and read operation on the communication buffers are not a problem anymore. Additionally, our code is much cleaner as before because it is about 500 loc less. Another improvement is the BinaryVTK output. It is much faster and safes harddisk space.

3 Turbulence Modeling

3.1 Implementation of an algebraic turbulence model

- **Compute a laminar reference case. Therefore, set $v_T = 0$.**
  Done with 2D.
  3D has been implemented but there is a bug which we are not able to find out which causes PETSc to diverge after few time steps.
- **Implement the Prandtl mixing length model. Consider that you need to know the local distance to the nearest wall as well as S_{ij}.**
  Done.
- **Following these instructions,**
  - **experiment two of the four options.**
    We chose to implement equation 6 and 7 as well as the option without boundary layer thickness dependency of the mixing length. In the following we will discuss the two models using equation 6 and 7 respectively.
  - **explain your choices. What would you have expected and how does that differ from your findings?**
    The two equations resolve the evolution of the boundary layer thickness in flow direction which will lead to a more accurate description of the boundary layer development compared to a model where the boundary layer thickness depends solely on the wall normal distance. The Blasius boundary layer equation describes a boundary layer thickness which is inverse proportional to the square root of Re_x whereas the turbulent boundary layer model has the boundary layer thickness proportional to the inverse of the 5th root. We expect those proportionalities will show in the development of the bound-

ary layer thickness and therefore in the mixing length and turbulent viscosity.

- **discuss your results and compare the results of your two modeling approaches.**
  The results show the expected proportionalities to the boundary layer thickness functions. Since the Blasisus boundary layer thickness increases faster than the turbulent boundary layer thickness and therefore the turbulent viscosity is higher in the entrance area of the channel. At x/H = 20 both laminar and turbulent boundary layer thickness are larger than kappa*h and therefore both models calculate the same turbulent viscosity.
- **compare your findings to the laminar reference case.**
  In the laminar reference case the turbulent viscosity is set to zero which results in less exchange of momentum in the fluid and therefore the velocity gradients are weaker and the maximum velocity is higher.

3.2 Implementational Details

3.2.1 Prerequisites

- **Extend the configuration of the simulation such that all parameters required by your turbulence model implementation can also be read from the xml-files.**
  Done.

3.2.2 Turbulent viscosity and wall distance

- **Extend the data structures for the turbulent case. In order to keep things simple, you may add a new scalar field for both the distance to the (nearest) wall and a scalar field to store the turbulent viscosity vT . We further evaluate the turbulent viscosity in the cell center, i.e. at the same location as the pressure.**
  The turbulent viscosity calculation is implemented in the TurbLPModelStencil files. The Stencil consists of properties for the local velocities and local mesh sizes, the getShearStressTensorProduct which determines the square root in equation (4) and the apply methods that calculate the turbulent viscosity value for each cell. The derivatives for the shear stress tensor are implemented in the StencilFunctions.h file. The mixing length is stored in TurbFlowField and calculated in MixingLengthStencil.
- **Consider the basic interplay of stencils, iterators and data structures in NS-EOF. How can you apply the concept of inheritance so that you can naturally extend your data structures and support both old and potentially new data structures in case of turbulence- model-based simulations?**
  Done.

- **Implement a VTK visualization for the turbulent viscosity. You may base your implementations on the existing plotter (VTKStencil) for the turbulent case and create a new stencil for this purpose which operates on the new data structure.**
  Done.

3.2.3 Momentum equations

- **Replace the discretization of the Laplacian in the evaluation of F, G and H by discrete expressions for**
  The velocity terms F, G and H are being evaluated in TurbFGHStencil. The methods that compute F, G and H are implemented in StencilFunctions. The compute functions used for the DNS simulations are overloaded and the evaluation of the Laplacian is changed to the evaluation of the three summands of equation (8-10) respectively. The methods used to calculate these summands consist of the derivative-methods and the methods that determine the turbulent viscosity at cell edges.
- **Improve your implementation by extending it to stretched meshes. How do you need to adapt the formulas for F, G, H from above?**
  We extend the implementation to stretched meshes by looking at the cell sizes and weighing the respective cell with the respective ratio. This is similar to what is already implemented in the code for other derivatives.

3.2.4 Parallelization

- **Extend interprocess communication by communication routines for the viscosity field. The communication of turbulent viscosity is carried out analogously to the communication of the pressure. How can you use the concept of inheritance to extend and reuse the existing communication routines from the PetscParallelManager?**
  Done. Similar to the communication of pressure and velocity using MPI_DATA_TYPE.
- **Extend the configuration by broadcasting the new parameters to all processes. You may use the methods MPI Bcast(...) and broadcastString(...), provided by MPI and the class Configuration.**
  Done.

3.2.5 Plugging things together

- **Use the concept of inheritance to create a simulation class TurbulentSimulation which possesses all properties of Simulation and incorporates the features of the algebraic turbulence modeling.**
  Done.
- **Adapt the time step restriction in TurbulentSimulation. Due to your inhomogeneous viscosity, the original expression for diffusive time step**

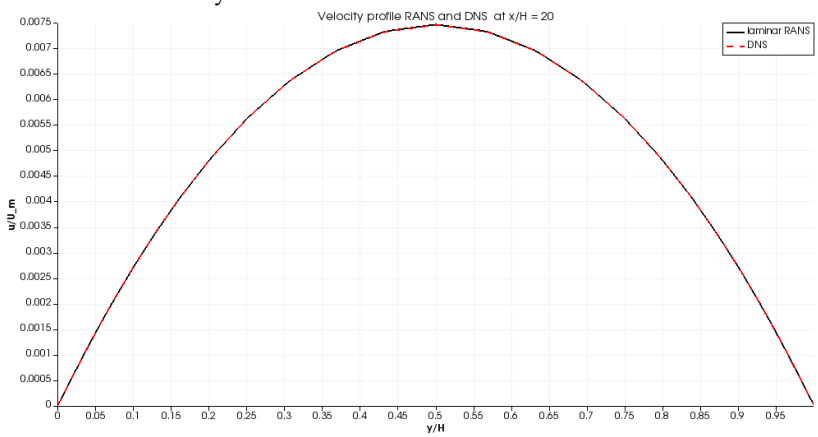
limitations
Done.

- **Implement the evaluation of the new time step criterion using the stencil-iterator concept.**
  Done.

4 Testing

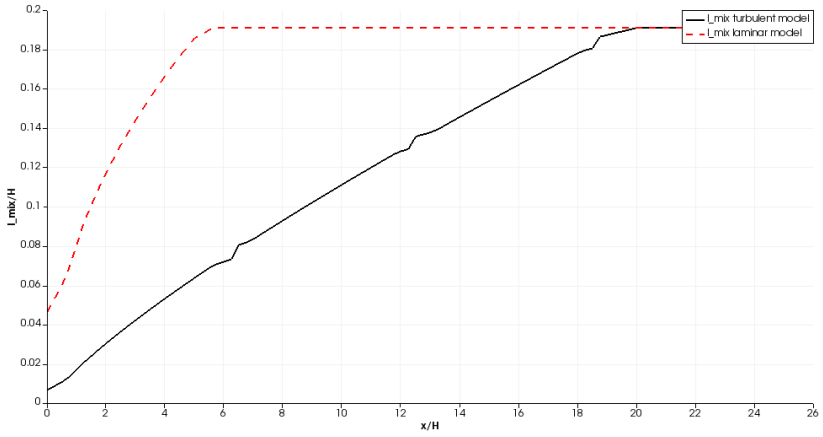
- **Test your parallel, turbulent flow simulation code in various channel flow scenarios. How can you verify that all finite difference expressions and parallel extensions are evaluated correctly?**
  Running the simulation with turbulent viscosity, ν_T, set to 0 if we observe that the output is the same as dns simulation then we conclude that the derivatives are correct. If the derivatives were wrong then we would not see this in the output.

- **Experiment with your choices on turbulence models in channel flows (cf. Sec. 3.1). What do you observe? How do the different models behave?**
  In the following graph the the normalized velocity profiles of a DNS simulation and a laminar RANS simulation for Re = 1000 are plotted ar x/H = 20 over the dimensionless wall normal distance. The profiles are congruent. Therefore the derivatives must be evaluated correctly. The domain has been decomposed into 4x2 blocks for the parallel compuation. The flow field throughout the channel does not show any unphysical sudden changes in flow properties and single core simulations show the same results. Therefore the parallel extensions are evaluated correctly.

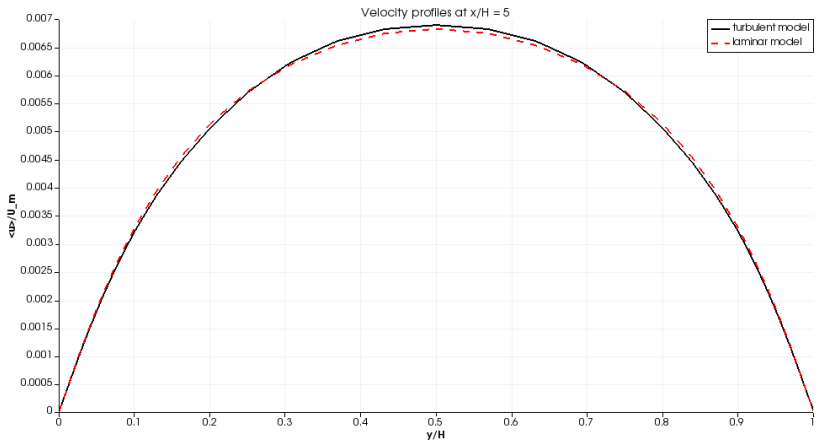


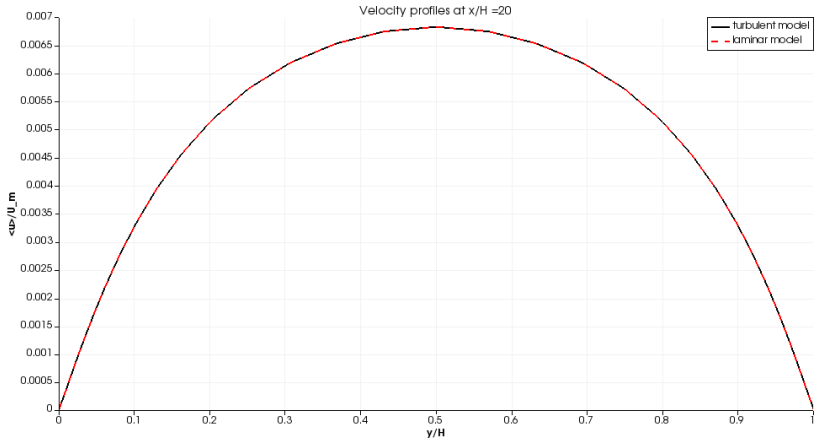
The mixing length development in flow direction is in accordance with the x-dependency of the two boundary layer thickness equations. Therefore the mixing length increases faster when the Blasius boundary layer equation is used than it does for the turbulent boundary layer

model. The graph below shows the evolution of the mixing length along the center line of the channel. The mixing length increases with the boundary layer thickness until κ *h is the restricting factor.

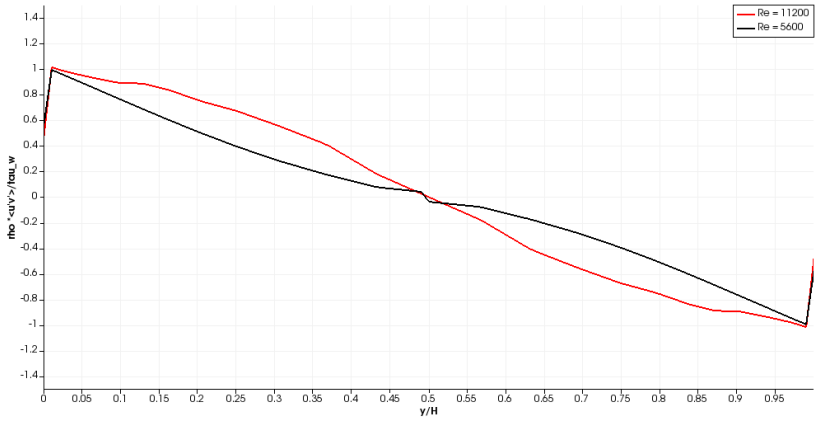


As the plot of the mixing length development has shown, differences in turbulent viscosity exist only until x/H = 20 is reached. Therefore the velocity profiles at x/H = 20 should be similar for both turbulence models whereas they should differ in the entrance region. This behavior is shown in the following two velocity profile graphs where the velocity profiles are plotted at x/H = 5 and x/H = 20.





The following plot shows the production term of the Reynolds stress tensor normalized with the wall shear stress over the normalized wall normal direction. With an increase in the Reynolds number the turbulence production increases. The decay of the turbulence production towards the channel center is not linear as expected. The turbulence production close to the wall is larger than expected. It reaches values close to one, which does not coincide with the results that are found in literature. We conclude there is an overproduction of turbulence close to the wall.



- **Run turbulence simulations for the backward-facing step scenario.**
  TODO:
  TODO: Attach your discussions and appropriate plots as a pdf-file to the code you submit!