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:

```
    MPI_Datatype _pres_right;  MPI_Datatype _pres_left;
    MPI_Datatype _pres_top;    MPI_Datatype _pres_bottom;
    MPI_Datatype _pres_front;  MPI_Datatype _pres_back;

    MPI_Datatype _velo_right;  MPI_Datatype _velo_left;
    MPI_Datatype _velo_top;    MPI_Datatype _velo_bottom;
    MPI_Datatype _velo_front;  MPI_Datatype _velo_back;
```

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

```

MPI_Type_vector(y,
                1,
                x+3,
                MY_MPI_FLOAT,
                &_pres_left
                );

MPI_Type_vector(y,
                2,
                x+3,
                MY_MPI_FLOAT,
                &_pres_right
                );
```

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. It operates on a BoundaryStencil.
- **Which functions does a respective stencil object need to provide?**
  This functions are necessary because they are defined as virtual ones.

```
    void applyLeftWall   (FlowField & flowField, int i, int j);
    void applyRightWall  (FlowField & flowField, int i, int j);
    void applyBottomWall (FlowField & flowField, int i, int j);
    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 need 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. It seems like there is a bug in the cavity scenario. Petsc wont converge if one chooses split in x and y at the same time.

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?**
  TODO:

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.
  TODO: Maybe some diagrams so that we can see the speedup

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?**
  TODO:

2.2 NS-EOF in the MAC-cluster

2.2.1 Environment and Login

Done.

2.2.2 Compiling

Done

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.**
  TODO:
- **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?**
  TODO:
- **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).**
  TODO:

3 Turbulence Modeling

3.1 Implementation of an algebraic turbulence model

- **Compute a laminar reference case. Therefore, set $\nu_T = 0$.**
  TODO:
- **Implement the Prandtl mixing length model. Consider that you need to know the local distance to the nearest wall as well as S_{ij}.**
  TODO:
- **Following these instructions,**
  - **experiment two of the four options.**
    TODO:
  - **explain your choices. What would you have expected and how does that differ from your findings?**
    TODO:
  - **discuss your results and compare the results of your two modeling approaches.**
    TODO:
  - **compare your findings to the laminar reference case.**
    TODO:
    TODO: Ensure to base all of your discussions on appropriate data!

  TODO:

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.**
  TODO:

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.**
  TODO:
- **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?**
  TODO:
- **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.**
  TODO:

3.2.3 Momentum equations

- **Replace the discretization of the Laplacian in the evaluation of F, G and H by discrete expressions for**
  TODO:
- **Improve your implementation by extending it to stretched meshes. How do you need to adapt the formulas for F, G, H from above?**
  TODO:

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?**
  TODO:
- **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.**
  TODO:

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.**
  TODO:
- **Adapt the time step restriction in TurbulentSimulation. Due to your inhomogeneous viscosity, the original expression for diffusive time step limitations**
  TODO:
- **Implement the evaluation of the new time step criterion using the stencil-iterator concept.**
  TODO:

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?**
  TODO:
- **Experiment with your choices on turbulence models in channel flows (cf. Sec. 3.1). What do you observe? How do the different models behave?**
  TODO:
- **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!