MueLu Tutorial

Release 2023

The MueLu Team

Nov 14, 2023

BEGINNERS TUTORIAL

This is the MueLu Tutorial. Additional resources can be found at the MueLu package web page, in the MueLu User's Guide and the Doxygen source code documentation.

CHAPTER

ONE

PREFACE

The MueLu tutorial is written as a hands-on tutorial for MueLu, the next generation multigrid framework in Trilinos. It covers the whole spectrum from absolute beginners' topics to expert level. Since the focus of this tutorial is on practical and technical aspects of multigrid methods in general and MueLu in particular, the reader should already have a basic understanding of multigrid methods and their general underlying concepts. Please refer to multigrid textbooks for the theoretical background.

CHAPTER

TWO

CONTENT

The tutorial is split into three parts. The first part contains four tutorials for beginners who are interested in using multigrid methods. No knowledge about C++ is required if the programs are used that come with the tutorial (in the Trilinos repository). If one uses the virtual box image one can even avoid the Trilinos compilation process. So, the tutorials in the first part can also be used for teaching purposes. One can easily study the smoothing effect of multigrid smoothers and perform some very basic experiments, which helps to gain a better understanding of multigrid methods. In the quick start tutorial, all steps are documented step by step such that it should be very easy to follow the tutorial. Different exercises may encourage the reader for performing some more experiments and tests. The following tutorials give an overview of the existing level smoothers and transfer operators, that can easily be used with the simple XML format, that MueLu uses for defining the multigrid hierarchies. In addition, it is explained how to visualize the aggregates and export the multigrid levels for a more in-depth analysis.

The second part consists of tutorials for users which are interested in some more background on the underlying techniques that are used in MueLu. The user still does not need explicit knowledge of C++ or any other programming language, but some interest in object-oriented design concepts may be helpful to understand the factory concept. The focus of the second part is on the introduction of the advanced XML interface for MueLu, which describes all internal building blocks of the multigrid setup procedures with its internal dependencies. In context of transfer operator smoothing, a brief introduction of the theory is given with some in-depth details on the algorithmic design in MueLu. More advanced topics are handled as well such as rebalancing or aggregation strategies. Additional exercises help the reader to perform some experiments in practice.

The third part is meant for expert users and for users, who want to use MueLu within their own software. Many detailed C++ examples show how to use MueLu from an user application as preconditioner for a Krylov subspace method or as a standalone multigrid solver. We expect the reader to be familiar with Trilinos, especially with the linear algebra packages Epetra and Tpetra as well as the linear solver packages AztecOO or Belos. For users who are already using ML, the predecessor multigrid package of MueLu in Trilinos, we provide a chapter describing the migration process from ML to MueLu.

2.1 Quick start

The first example is meant to quickly get into touch with MueLu.

2.1.1 Example problem

We generate a test matrix corresponding to the stencil of a 2D Laplacian operator on a structured Cartesian grid. The matrix stencil is

$$\frac{1}{h^2}\begin{pmatrix} & -1 & \\ -1 & 4 & -1 \\ & -1 & \end{pmatrix}$$

where h denotes the mesh size parameter. The resulting matrix is symmetric positive definite. We choose the right hand side to be the constant vector one and use a random initial guess for the iterative solution process. The problem domain is the unit square with a Cartesian (uniform) mesh.

2.1.2 User interface

For this tutorial there is an easy-to-use user interface to perform some experiments with multigrid methods for the given problem as described in *Example problem*. To use the user-interface run **./hands-on.py** in a terminal in the Trilinos build directory's **packages/muelu/test/tutorial** folder. From this point forward, we will provide all paths relative to that folder in the build directory. First one has to choose a problem. For this tutorial, select option 0 for the Laplace 2D problem on a 50×50 mesh.

```
*************************   PROBLEM   *****************************
Problem type:          Laplace 2D
Mesh:                  50x50

Solver xml parameters:              s2a.xml
Number of processors:               2
Number of Multigrid solving sweeps: 1
*************************   PROBLEM   *****************************


0. Laplace 2D (50x50)
1. Laplace 2D
2. Recirc 2D (50x50)
3. Recirc 2D
4. Challenge: Convection diffusion
5. Challenge: Elasticity problem
6. Exit
your choice?
```

Next one has to choose a xml file with the multigrid parameters. Choose option 2 and put in **s1_easy.xml** as filename for the xml file containing the xml parameters that are used for the multigrid method.

Note: Please make sure that you enter a filename that actually exists on your hard disk!

```
*************************   PROBLEM   ****************************
Problem type:          Laplace 2D
Mesh:                  50x50

Solver xml parameters:              s2a.xml
Number of processors:               2
Number of Multigrid solving sweeps: 1
*************************   PROBLEM   ****************************

DO NOT FORGET TO RUN THE EXAMPLE (option 0)

0. Rerun simulation
1. Show screen output
2. Change solver
3. Open xml file
4. Change procs
5. Change MG sweeps
6. Plot solution
7. Plot residual norm over cg solver iterations
8. Postprocess aggregates
9. Exit
your choice?
```

The s1_easy.xml file has the following content

Listing 2.1: ../../../test/tutorial/s1_easy.xml

```
<ParameterList name="MueLu">

  <Parameter name="verbosity" type="string" value="high"/>

  <Parameter name="max levels" type="int" value="3"/>
  <Parameter name="coarse: max size" type="int" value="10"/>

  <Parameter name="multigrid algorithm" type="string" value="sa"/>

  <!-- Smoothing -->
  <!-- Comment/uncomment different sections to try different smoothers -->

  <!-- Jacobi -->
  <Parameter name="smoother: type" type="string" value="RELAXATION"/>
  <ParameterList name="smoother: params">
    <Parameter name="relaxation: type"  type="string" value="Jacobi"/>
    <Parameter name="relaxation: sweeps" type="int" value="1"/>
    <Parameter name="relaxation: damping factor" type="double" value="0.9"/>
  </ParameterList>

  <!-- Aggregation -->
  <Parameter name="aggregation: type" type="string" value="uncoupled"/>
  <Parameter name="aggregation: min agg size" type="int" value="3"/>
  <Parameter name="aggregation: max agg size" type="int" value="9"/>

</ParameterList>
```

As one can easily find from the xml parameters, a multigrid method with not more than 3 levels and a damped Jacobi method for level smoothing shall be used. Next, choose option 0 and run the example. That is, the linear system is created and iteratively solved both by a preconditioned CG method with a MueLu multigrid preconditioner and a standalone multigrid solver (again using MueLu) with the given multigrid parameters.

```
**************************   PROBLEM   ****************************
Problem type:          Laplace 2D
Mesh:                  50x50

Solver xml parameters:              s2a.xml
Number of processors:               2
Number of Multigrid solving sweeps: 1
**************************   PROBLEM   ****************************

DO NOT FORGET TO RUN THE EXAMPLE (option 0)

0. Rerun simulation
1. Show screen output
2. Change solver
3. Open xml file
4. Change procs
5. Change MG sweeps
6. Plot solution
7. Plot residual norm over cg solver iterations
8. Postprocess aggregates
9. Exit
your choice? 2
XML file name: s1_easy.xml
```

Note that the line **mpirun -np 2 MueLu_tutorial_laplace2d.exe -nx ...** is the command that is executed in the background. The default is 2 processors used. After pressing a key we are ready for a first analysis as it is stated by the green letters **Results up to date!**

```
**************************   PROBLEM   ****************************
Problem type:          Laplace 2D
Mesh:                  50x50

Solver xml parameters:              s1_easy.xml
Number of processors:               2
Number of Multigrid solving sweeps: 1
**************************   PROBLEM   ****************************

DO NOT FORGET TO RUN THE EXAMPLE (option 0)

0. Rerun simulation
1. Show screen output
2. Change solver
3. Open xml file
4. Change procs
5. Change MG sweeps
6. Plot solution
7. Plot residual norm over cg solver iterations
8. Postprocess aggregates
9. Exit
```

(continues on next page)

(continued from previous page)

```
your choice? 0
PREPARE SIMULATON
RUN EXAMPLE
mpirun -np 2 MueLu_Tutorial_laplace2d.exe --nx=50 --ny=50 --mgridSweeps=1 --xml=s1_easy.
↪xml | tee output.log 2>&1
POSTPROCESSING...
COMPLETE
Press any key to continue...
```

Note: If the results are not up to date always choose option 0 first to recalculate the results.

To check the output select option 1. This should produce the following output on screen.

Note: Depending on the number of lines in your terminal you may have to scroll up to the top of the file

These lines give you some information about the setup process with some details on the aggregation process and the transfer operators. Note that for this example three levels are built: Level 0 for the finest level, level 1 as intermediate level and level 2 for the coarsest level. Then an overview of the different multigrid levels is given by

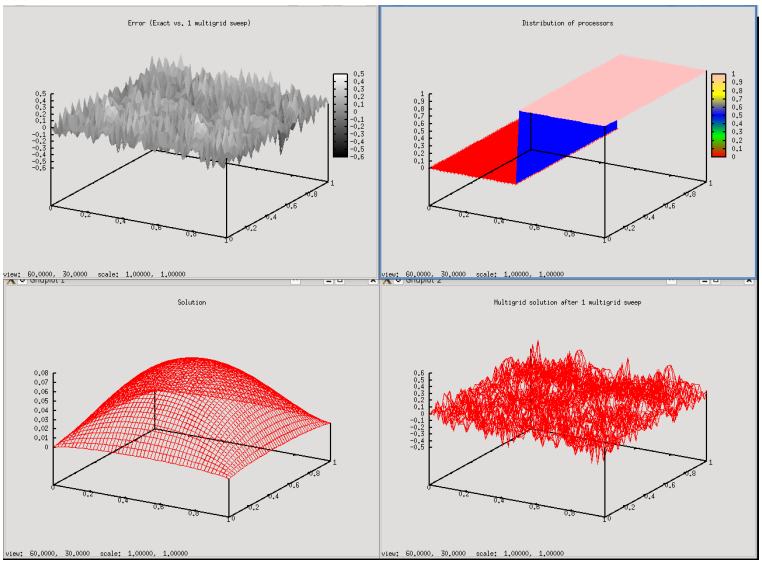
Warning: Insert screen output

One can see that a three level multigrid method is used with a direct solver on the coarsest level and Jacobi level smoothers on the fine and intermediate level. Furthermore some basic information is printed such as the operator complexity.

In the end the CG convergence is printed when applying the generated multigrid method as preconditioner within a CG solver. The numbers give the relative residual after the corresponding number of iterations as well as the solution time in seconds.

Warning: Insert screen output

Selecting option 6 gives you four plots.



The lower left plot shows the exact solution of the linear system (using a direct solver from the Amesos package). The lower right plot shows the multigrid solution when 1 sweep with a V-cycle of the multigrid method as defined in the xml parameter file is applied to the linear system as a standalone multigrid solver. As one can see, the multigrid solution with a random initial guess is far away from the exact solution. The upper left plot shows the difference between the multigrid solution and the exact solution. Finally, the upper right plot shows the distribution of the fine level mesh nodes over the processors (in our example we use 2 processors).

Note: The plots do not show the solution of the preconditioned CG method! The solution of the CG method is always exact up to a given tolerance as long as the multigrid preconditioner is sufficient. This can be checked by the screen output under option 1.

As a first experiment we change the number of multigrid sweeps for the stand alone multigrid smoother. Let's choose option 5 and use 10 multigrid sweeps.

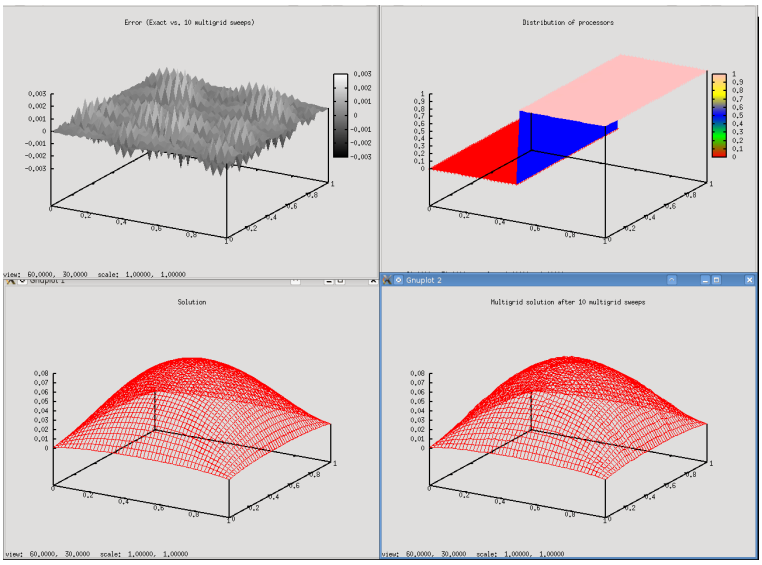
```
Datei  Bearbeiten  Ansicht  Lesezeichen  Einstellungen  Hilfe
*****************************   PROBLEM   *****************************
Problem type:          Laplace 2D
Mesh:                  50x50

Solver xml parameters:              xml/s1_easy.xml
Number of processors:               2
Number of Multigrid solving sweeps: 1
*****************************   PROBLEM   *****************************

Results up to date!

0. Rerun simulation
1. Show screen output
2. Change solver
3. Open xml file
4. Change procs
5. Change MG sweeps
6. Plot solution
7. Plot residual norm over cg solver iterations
8. Postprocess aggregates
9. Exit
your choice? 5
Number of Multigrid sweeps: 10
```

Then, do not forget to rerun the examples to update the results. That is, choose option 0 and wait for the simulation to finish. Then plot again the results using menu option 6 and you should obtain



As one can see is the multigrid solution rather close to the exact solution. In the error plot one finds some low and high frequency error components.

The XML input deck - multigrid parameters

After we have learned the basics of the driver program for our experiments we now perform some experiments with our multigrid methods. We again use the simple 2D Laplace problem. First, we create a copy of the solver parameters using

cp s1_easy.xml mysolver.xml

Then, we run the driver program again using

./hands-on.sh

and choose option 0 for the 2D-Laplace example on the **50times50** mesh. Use the xml parameters from the **mysolver.xml****`file, that is, choose option 2 and put in **mysolver.xml**. Make sure that the problem can be solved with the parameters (option 0) and verify the solver output. Once that is done it is time for some first experiments. Open your **mysolver.xml** file in a text editor. You can try option 3 for doing that, but alternatively you can also do it by hand choosing your favorite text editor.

Neu | Öffnen | Speichern | Speichern unter | Schließen | Rückgängig | Wiederherstellen

```
<ParameterList name="MueLu">

  <Parameter name="verbosity" type="string" value="low"/>

  <Parameter name="max levels" type="int" value="3"/>
  <Parameter name="coarse: max size" type="int" value="10"/>

  <Parameter name="multigrid algorithm" type="string" value="sa"/>

  <!-- Smoothing -->
  <!-- Comment/uncomment different sections to try different smoothers -->

  <!-- Jacobi -->
  <Parameter name="smoother: type" type="string" value="RELAXATION"/>
  <ParameterList name="smoother: params">
    <Parameter name="relaxation: type"  type="string" value="Jacobi"/>
    <Parameter name="relaxation: sweeps" type="int" value="1"/>
    <Parameter name="relaxation: damping factor" type="double" value="0.9"/>
  </ParameterList>

  <!-- Aggregation -->
  <Parameter name="aggregation: type" type="string" value="uncoupled"/>
  <Parameter name="aggregation: min agg size" type="int" value="3"/>
  <Parameter name="aggregation: max agg size" type="int" value="9"/>

</ParameterList>
```

Now, let's change the maximum number of multigrid levels from 3 to 10 in the xml file, that is, change the value of the parameter **max levels** from 3 to 10. Do not forget to save the file and rerun the example by choosing option 0 in the driver program. The screen output should be the following

Warning: Insert missing output

Note: Even though we allow for at maximum 10 multigrid levels the coarsening process stops after level 4. The reason is that the linear operator on multigrid level 4 has only 4 lines and therefore is smaller than the coarse: max size parameter in the xml parameter list which defines the maximum size of the linear operator on the coarsest level.

The option sa for **smoothed aggregation** in the multigrid algorithm parameter can be considered to be optimal for symmetric positive definite (SPD) problems. We can compare it with the option unsmoothed as a robust but slower alternative. Let's choose a 3 level multigrid method with unsmoothed transfer operators (i.e., max levels = 3, multigrid algorithm = unsmoothed), then we obtain

Warning: Insert missing output

Compared with the smoothed aggregation method (multigrid algorithm = sa) which uses some smoothed transfer operator basis functions within the multigrid method, the unsmoothed multigrid algorithm needs a significantly higher number of iterations. The same method with smoothed transfer operator basis functions gives

Warning: Insert missing output

Note: You can find the corresponding xml files also in **../../../test/tutorial/s1_easy_3levels_smoothed.xml**

2.2 Level smoothers

From the last tutorial, we have learned that the used multigrid algorithm may have a significant influence in the convergence speed. When comparing the error plots for the standalone multigrid smoothers with unsmoothed and smoothed aggregation multigrid, one finds also a notable difference in the **smoothness** of the error.

2.2.1 Background on multigrid methods

Obviously, there are cases where some highly oscillatory error modes are left and overlaying some low frequency modes. In other cases there are only low frequency error modes left. Theses are two typical cases one might find in practice.

Multigrid methods are built upont the observation that (cheap) level smoothing method often are able to smooth out high oscillatory error components, whereas they cannot reduce low frequency error components very well. These low frequency error components are then transferred to a coarse level, where they can be seen as high frequency error component for a level smoother on the coarse level.

Warning: Display an image of a sine wave appearing oscillatory on a coarse grid

One should not forget that for an efficient multigrid method both the so-called coarse level correction method and the level smoothers have to work together. That is, one has to choose the right multigrid method (e.g., **unsmoothed** or **sa**) in combination with an appropriate level smoothing strategy.

In context of multigrid level smoothers, we have to define both the level smoothers and the coarse solver. Usually, a direct solver is used as coarse solver that is applied to the coarsest multigrid levels. However, it is also possible to apply any other kind of iterative smoothing method or even no solver at all (even though this would be non-standard). The following XML file shows how to use a Jacobi smoother both for level smoothing and as a coarse solver.

Listing 2.2: ../../../test/tutorial/s1_easy_jacobi.xml

```
<ParameterList name="MueLu">

  <Parameter name="verbosity" type="string" value="low"/>

  <Parameter name="max levels" type="int" value="10"/>
  <Parameter name="coarse: max size" type="int" value="10"/>

  <Parameter name="multigrid algorithm" type="string" value="unsmoothed"/>

  <!-- Jacobi -->
  <Parameter name="smoother: type" type="string" value="RELAXATION"/>
  <ParameterList name="smoother: params">
    <Parameter name="relaxation: type"  type="string" value="Jacobi"/>
    <Parameter name="relaxation: sweeps" type="int" value="1"/>
    <Parameter name="relaxation: damping factor" type="double" value="0.9"/>
  </ParameterList>

  <!-- Jacobi -->
  <Parameter name="coarse: type" type="string" value="RELAXATION"/>
  <ParameterList name="coarse: params">
    <Parameter name="relaxation: type"  type="string" value="Jacobi"/>
    <Parameter name="relaxation: sweeps" type="int" value="1"/>
    <Parameter name="relaxation: damping factor" type="double" value="0.9"/>
  </ParameterList>

</ParameterList>
```

The corresponding multigrid hierarchy is

Table 2.1 and Table 2.2 show the multigrid effect of different number of Jacobi smoothers on all multigrid levels.

Table 2.1: 2D Laplace equation on 50 x 50 mesh after 1 V-cycle with an AMG multigrid solver and Jacobi smoothers on all multigrid levels (2 processors)

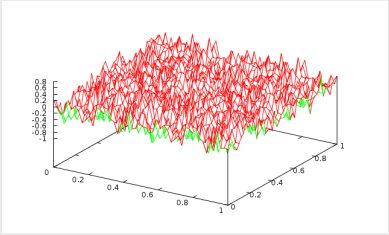


Fig. 2.1: 1 level with 1 Jacobi sweep ($\omega = 0.9$)

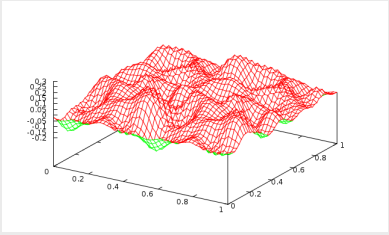


Fig. 2.2: 1 level with 10 Jacobi sweeps ($\omega = 0.9$)

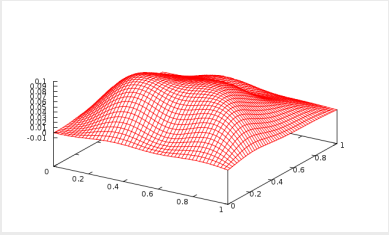


Fig. 2.3: 1 level with 100 Jacobi sweeps ($\omega = 0.9$)

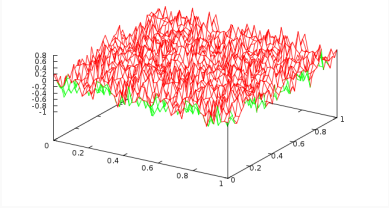


Fig. 2.4: 2 levels with 1 Jacobi sweep ($\omega = 0.9$)

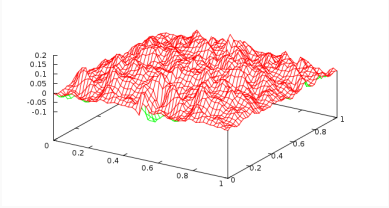


Fig. 2.5: 2 levels with 10 Jacobi sweeps ($\omega = 0.9$)

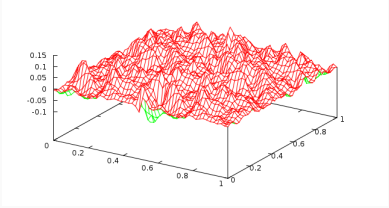


Fig. 2.6: 2 levels with 100 Jacobi sweeps ($\omega = 0.9$)

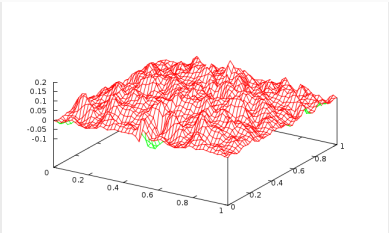


Fig. 2.7: 3 levels with 1 Jacobi sweep ($\omega = 0.9$)

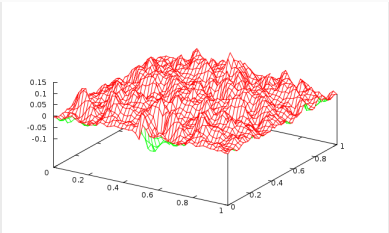


Fig. 2.8: 3 levels with 10 Jacobi sweeps ($\omega = 0.9$)

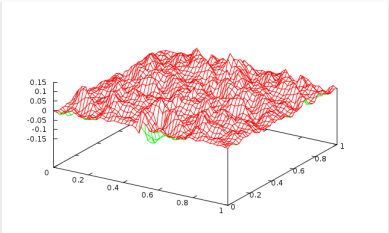


Fig. 2.9: 3 levels with 100 Jacobi sweeps ($\omega = 0.9$)

Table 2.2: 2D Laplace equation on 50 x 50 mesh after 5 V-cycle with an AMG multigrid solver and Jacobi smoothers on all multigrid levels. (2 processors)

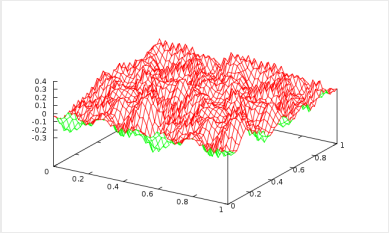


Fig. 2.10: 1 level with 1 Jacobi sweep ($\omega = 0.9$)

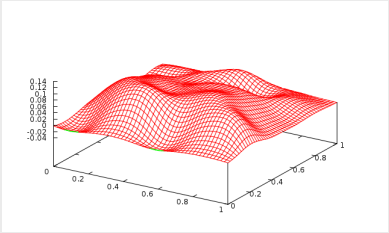


Fig. 2.11: 1 level with 10 Jacobi sweeps ($\omega = 0.9$)

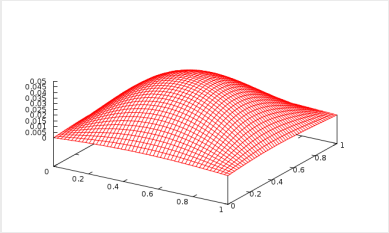


Fig. 2.12: 1 level with 100 Jacobi sweeps ($\omega = 0.9$)

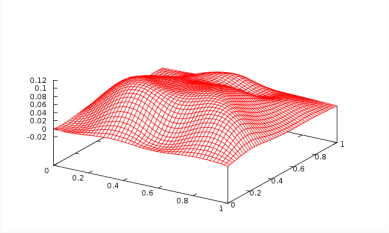


Fig. 2.13: 2 levels with 1 Jacobi sweep ($\omega = 0.9$)

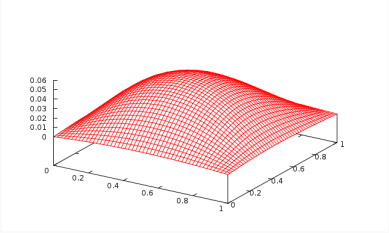


Fig. 2.14: 2 levels with 10 Jacobi sweeps ($\omega = 0.9$)

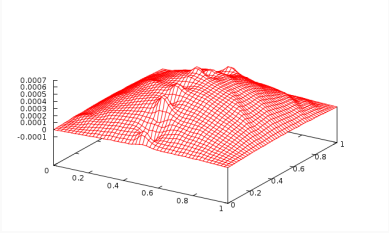


Fig. 2.15: 2 levels with 100 Jacobi sweeps ($\omega = 0.9$)

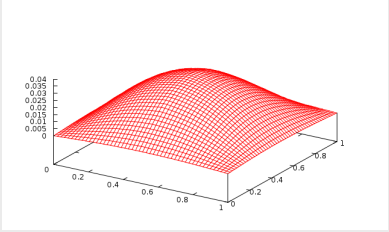


Fig. 2.16: 3 levels with 1 Jacobi sweep ($\omega = 0.9$)

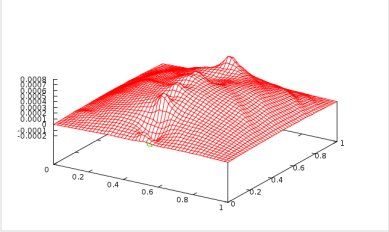


Fig. 2.17: 3 levels with 10 Jacobi sweeps ($\omega = 0.9$)

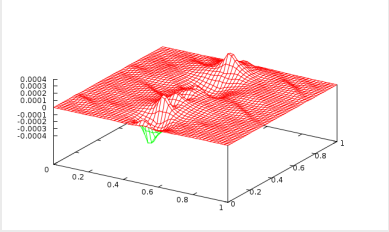


Fig. 2.18: 3 levels with 100 Jacobi sweeps ($\omega = 0.9$)

One has even more fine-grained control over pre- and post-smoothing as shown in the following example, where we use different damping parameters for pre- and post-smoothing (and a direct solver on the coarse grid):

Listing 2.3: ../../../test/tutorial/s1_easy_jacobi2.xml

```
<ParameterList name="MueLu">

  <Parameter name="verbosity" type="string" value="low"/>

  <Parameter name="max levels" type="int" value="10"/>
  <Parameter name="coarse: max size" type="int" value="10"/>

  <Parameter name="multigrid algorithm" type="string" value="unsmoothed"/>

  <!-- Jacobi as pre-smoother with damping of 0.6 -->
  <Parameter name="smoother: pre type" type="string" value="RELAXATION"/>
  <ParameterList name="smoother: pre params">
    <Parameter name="relaxation: type"  type="string" value="Symmetric Gauss-Seidel"/>
```

(continues on next page)

(continued from previous page)

```
    <Parameter name="relaxation: sweeps" type="int" value="3"/>
    <Parameter name="relaxation: damping factor" type="double" value="0.6"/>
  </ParameterList>

  <!-- Jacobi as pre-smoother with damping of 0.9 -->
  <Parameter name="smoother: post type" type="string" value="RELAXATION"/>
  <ParameterList name="smoother: post params">
    <Parameter name="relaxation: type" type="string" value="Gauss-Seidel"/>
    <Parameter name="relaxation: sweeps" type="int" value="1"/>
    <Parameter name="relaxation: damping factor" type="double" value="0.9"/>
  </ParameterList>

  <!-- Direct solver on the coarsest level -->
  <Parameter name="coarse: type" type="string" value="DIRECT"/>

</ParameterList>
```

This produces the following multigrid hierarchy

Warning: Insert missing output

Note: Note that the relaxation-based methods provided by the Ifpack/Ifpack2 package are embedded in an outer additive Schwarz method.

Of course, there exist other smoother methods such as polynomial smoothers (Chebyshev) and ILU-based methods. A detailed overview of the different available smoothers can be found in the MueLu User's Guide ([1]).

Exercise 1

Play around with the smoother parameters and study their effect on the error plot and the convergence of the preconditioned CG method. For all available smoothing options and parameters refer to the MueLu user guide ([1]). Hint: use **unsmoothed** transfer operator basis functions (i.e., **multigrid algorithm = unsmoothed**) to highlight the effect of the level smoothers.

Exercise 2

Use the following parameters to solve the 50×50 Laplace 2D problem on 2 processors:

Listing 2.4: ../../../test/tutorial/s1_easy_exercise.xml

```
<ParameterList name="MueLu">

  <Parameter name="verbosity" type="string" value="low"/>
```

(continues on next page)

[1] L. Berger-Vergiat, C. A. Glusa, G. Harper, J. J. Hu, M. Mayr, P. Ohm, A. Prokopenko, C. M. Siefert, R. S. Tuminaro, and T. A. Wiesner. MueLu User's Guide. Technical Report SAND2023-12265, Sandia National Laboratories, Albuquerque, NM (USA) 87185, 2023.

(continued from previous page)

```
  <Parameter name="max levels" type="int" value="3"/>
  <Parameter name="coarse: max size" type="int" value="10"/>
  <Parameter name="multigrid algorithm" type="string" value="sa"/>

  <!-- Jacobi -->
  <Parameter name="smoother: type" type="string" value="RELAXATION"/>
  <ParameterList name="smoother: params">
    <Parameter name="relaxation: type"  type="string" value="Jacobi"/>
    <Parameter name="relaxation: sweeps" type="int" value="1"/>
    <Parameter name="relaxation: damping factor" type="double" value="0.9"/>
  </ParameterList>

  <!-- Jacobi -->
  <Parameter name="coarse: type" type="string" value="DIRECT"/>

</ParameterList>
```

That is, we change to smoothed aggregation AMG (SA-AMG). You can find the xml file also in **../../../test/tutorial/s1_easy_exercise.xml**. Run the example on two processors and check the number of linear iterations and the solver timings in the screen output. Can you find smoother parameters which reduce the number of iterations? Can you find smoother parameters which reduce the iteration timings?

2.2.2 Footnotes

2.3 Multigrid for non-symmetric problems

2.3.1 Test example

The **Recirc2D** example uses a matrix corresponding to the finite-difference discretization of the problem

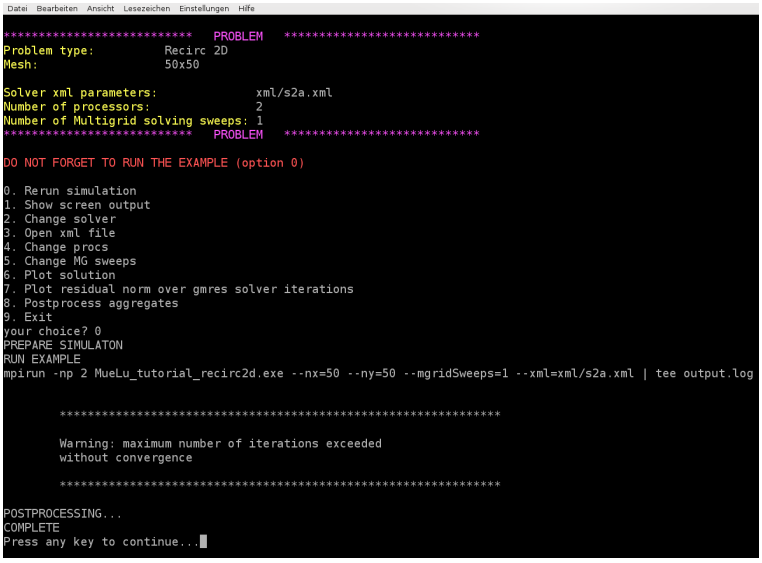
$$-\varepsilon\Delta u + (v_x, v_y)\cdot\nabla u = f$$

on the unit square, with $\varepsilon = 1e-5$ and homogeneous Dirichlet boundary conditions. It is $v_x = 4x(x-1)(1-2y)$ and $v_y = -4y(y-1)(1-2x)$. The right hand side vector f is chosen to be the constant vector 1. Due to the convective term the resulting linear system is non-symmetric and therefore more challenging for the iterative solver. The multigrid algorithm has to be adapted to the non-symmetry to obtain good convergence behavior.

2.3.2 User interface

For this tutorial again we can use the easy-to-use user interface. Run the **hands-on.py** script in your terminal and choose option 2 for the **Recirc 2D** example on a 50×50 mesh. Note that the default values from the file **../../../test/tutorial/s2a.xml** do not lead to a convergent multigrid preconditioner.

The convergence of the used unsmoothed transfer operators (**multigrid algorithm = unsmoothed**) is not optimal. In case of symmetric problems one can reduce the number of iterations using smoothed aggregation algebraic multigrid methods. In context of non-symmetric problems, especially when arising from problems with (highly) convective phenomena, one should use a Petrov-Galerkin approach for smoothing the prolongation and restriction operators more carefully.



In MueLu one can choose a Petrov-Galerkin approach for the transfer operators by setting **multigrid algorithm = pg**. Furthermore, one has to state that the system is non-symmetric by setting **problem: symmetric = false**. In addition, you have to set **transpose: use implicit = false** to make sure that the prolongation and restriction are built separately. This is highly important for non-symmetric problems since $R = P^T$ is not a good choice for non-symmetric problems (see, e.g.,[1][2]).

The role of the **transpose: use implicit** and the **problem: symmetric** parameters are the following:

Description

- **transpose: use implicit** Use $R = P^T$ for the restriction operator and do not explicitly build the operator R.

This can save a lot of memory and might be very performant when building the multigrid Galerkin product. However, for non-symmetric problems this is not working and has to be turned off. * **problem: symmetric** If **true**, use $R = P^T$ as restriction operator. Depending on the **transpose: use implicit** parameter the restriction operator is explicitly built. If **false** a Petrov-Galerkin approach as described in[Page 18, 1] is used to build the restriction operator separately. Note, that for the Galerkin approach it is necessary to build the restriction operator explicitly and store it.

Note: One can also use unsmoothed transfer operators (**multigrid algorithm = unsmoothed**) for non-symmetric problems. These might not give optimal results with respect to the iteration count, but they can be used with **transpose: use implicit = true** for non-symmetric problems, too, without disturbing the convergence. This way one can save a significant amount of memory compared to the smoothed aggregation method with Petrov-Galerkin for non-symmetric problems.

Exercise 1

Choose the parameters from the **n1_easy.xml** file. If you run the example you might find that the GMRES method did not converge within 50 iterations. Use **multigrid algorithm = pg** and compare the results with **multigrid algorithm = unsmoothed**. Do not forget to set the other parameters correctly for Petrov-Galerkin methods as described before.

[1] Sala, M. and Tuminaro, R. S., A new Petrov-Galerkin Smoothed Aggregation Preconditioner for nonsymmetric Linear Systems, SIAM J. Sci. Comput., 2008, 31, p. 143–166

[2] Wiesner, T. A., Tuminaro, R. S., Wall, W. A. and Gee, M. W., Multigrid transfers fornonsymmetric systems based on Schur complements and Galerkin projections., Numer.Linear Algebra Appl., 2013, doi: 10.1002/nla.1889

What is the difference in the number of GMRES iterations? What is changing in the multigrid setup?

Exercise 2

For slightly non-symmetric problems the **sa** method often performs satisfactorily. Change the verbosity to high (**verbosity = high**) and compare the results of the **multigrid algorithm = pg** option with the **multigrid algorithm = sa** option. Check the role of the **transpose: use implicit** parameter. What is changed by the **problem: symmetric** parameter? Try different values between 0 and 1.5 for the damping parameter within the smoothed aggregation method (i.e., try values 0.0, 0.5, 1.0, 1.33 and 1.5 for **sa: damping factor**). What do you observe?

2.3.3 Footnotes

2.4 Useful tools for analysis

2.4.1 Visualization of aggregates

Technical prerequisites

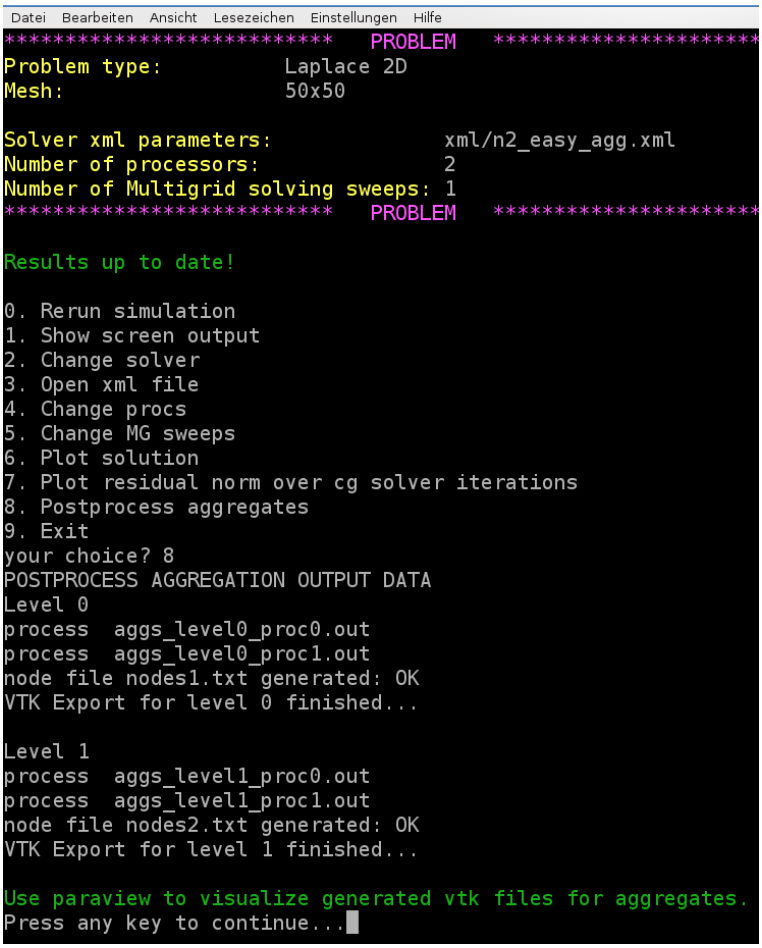
MueLu allows one to export plain aggregation information in simple text files that may be interpreted by post-processing scripts to generate pictures from the raw data. The post-processing script provided with the MueLu tutorial is written in python and produces VTK output. Please make sure that you have all necessary python packages installed on your machine (including **python-vtk**).

Note: The visualization script has successfully been tested with VTK 5.x. Note that it is not compatible to VTK 6.x.

Visualization of aggregates with MueLu using VTK

We can visualize the aggregates using the vtk file format and a visualization program called ParaView. First add the parameter **aggregation: export visualization data = true** to the list of aggregation parameters. Use, e.g., the following xml file **../../../test/tutorial/n2_easy_agg.xml**.

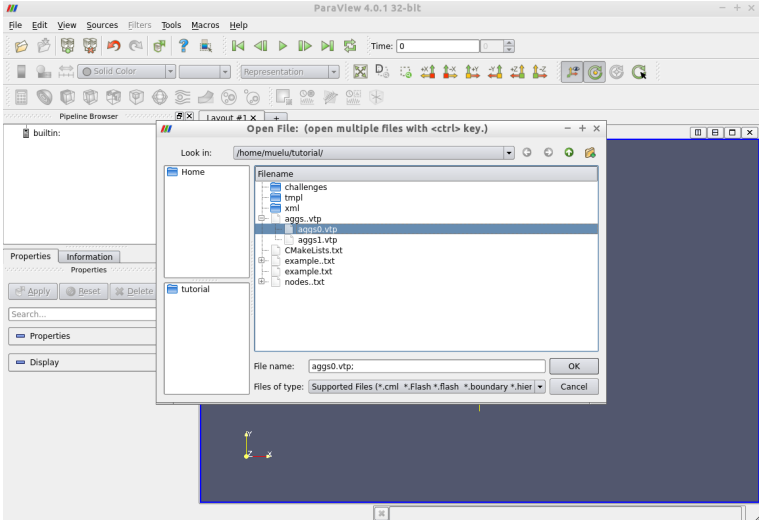
Run the **hands-on.py** script and select, e.g., the Laplace 2D example on a 50×50 mesh. Select above xml file for the multigrid parameters with the **aggregation: export visualization data** enabled. Run the program and then choose option 8 for post-processing the aggregates.



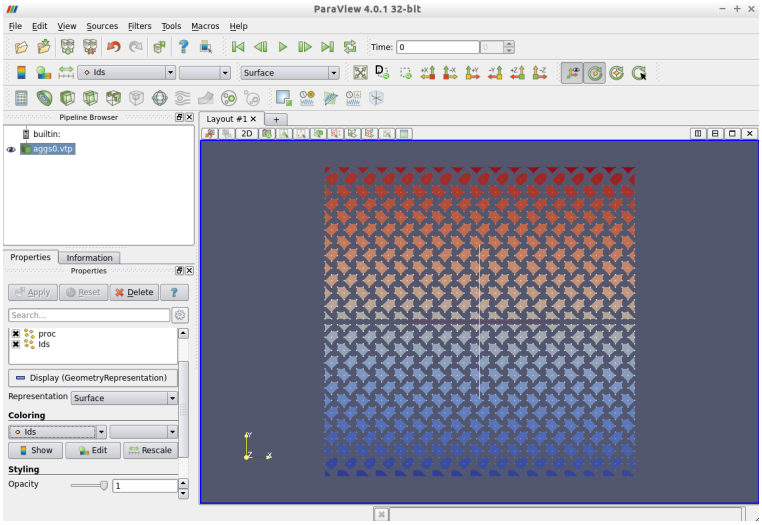
Note: Be aware that without **aggregation: export visualization data = true** the post processing step for the aggregates will fail.

Once the visualization data is exported and post-processed you can run *ParaView* (if it is installed on your machine) and open the files **aggs0.vtp** and **aggs1.vtp** for visualization.

Start *ParaView* and open the files **aggs0.vtp** and/or **aggs1.vtp**. Do not forget to press the *Apply* button to show the aggregates on screen.



Then the aggregates should be visualized as follows.



Here the colors represent the unique aggregate id. You can change the coloring in the left column from *Ids* to *proc* which denotes the owning processor of the aggregate.

Figure *Aggregates for Laplace2D example on 50\times 50 mesh without dropping.* shows the aggregates for the Laplace2D problem on the different multigrid levels starting with an isotropic 50×50 mesh. No dropping of small entries was used when building the matrix graph (**aggregation: drop tol=0.0**). For visualization purposes the *midpoint* of each aggregate defines the coordinate of the supernode on the next coarser level. Be aware that these supernodes are purely algebraic. There is no coarse mesh for algebraic multigrid methods. As one can see from the colors an uncoupled aggregation strategy has been applied using 2 processors. The aggregates do not cross the processor boundaries.

Fig. 2.19: Aggregates for Laplace2D example on 50×50 mesh without dropping.

Exercise 1

Repeat above steps for the *Recirc2D* example on a 50×50 mesh. Compare the aggregates from the **../../../test/tutorial/n2_easy_agg.xml** parameter file with the aggregates when using the **../../../test/tutorial/n2_easy_agg2.xml** parameter file, which drops some small entries of the fine level matrix A

when building the graph.

Exercise 2

Vary the number of processors. Do not forget to export the aggregation data (option 7) after the simulation has rerun with a new number of processors. In *ParaView* choose the variable *proc* for the coloring. Then the color denotes the processor the aggregate belongs to. How do the aggregates change when switching from 2 to 3 processors? * Try the solver parameters from **../../../test/tutorial/s4c.xml** or the *Recirc2D* example on a 50×50 mesh and compare them with the results for the **../../../test/tutorial/s4a.xml** and **../../../test/tutorial/s4b.xml** parameters. Which differences do you observe?

Figure *Aggregates for Recirc2D example on 50\times 50 mesh with dropping.* shows the aggregates for the Recirc2D problem. When building the matrix graph, entries with values smaller than 0.01 were dropped. Obviously the shape of the aggregates follows the direction of convection of the example. Using an uncoupled aggregation method (i.e., **aggregation: type = uncoupled**) as default the aggregates do not cross processor boundaries.

Fig. 2.20: Aggregates for Recirc2D example on 50×50 mesh with dropping.

2.4.2 Export data

For debugging purposes it can be very helpful to have a look at the coarse level matrices as well as the transfer operators. MueLu allows one to export the corresponding operators to the matrix market format such that the files can be imported, e.g., into MATLAB (or FreeMat) for some in-depth analysis.

The following xml file writes the fine level operator and the coarse level operator as well as the prolongation and restriction operator to the hard disk using the filenames **A_0.m**, **A_1.m** as well as **P_1.m** and **R_1.m**

Listing 2.5: ../../../test/tutorial/n2_easy_export.xml

```
<ParameterList name="MueLu">

  <Parameter name="verbosity" type="string" value="high"/>

  <Parameter name="max levels" type="int" value="3"/>
  <Parameter name="coarse: max size" type="int" value="10"/>

  <Parameter name="multigrid algorithm" type="string" value="unsmoothed"/>

  <Parameter name="smoother: type" type="string" value="RELAXATION"/>
  <ParameterList name="smoother: params">
    <Parameter name="relaxation: type"  type="string" value="Gauss-Seidel"/>
    <Parameter name="relaxation: sweeps" type="int" value="3"/>
    <Parameter name="relaxation: damping factor" type="double" value="0.7"/>
  </ParameterList>

  <!-- Export data -->
  <ParameterList name="export data">
    <Parameter name="A" type="string" value="{0,1}"/>
    <Parameter name="P" type="string" value="{0,1}"/>
    <Parameter name="R" type="string" value="{0}"/>
```

(continues on next page)

(continued from previous page)

```
  </ParameterList>
</ParameterList>
```

Note: Be aware that there is no prolongator and restrictor on the finest level (level 0) since the transfer operators between level ℓ and $\ell+1$ are always associated with the coarse level $\ell+1$ (for technical reasons). So, be not confused if there is no **P_0.m** and **R_0.m**. Only the operators are written to external files which really exist and are requested in the corresponding list in the xml parameters.

The exported files can easily imported into MATLAB and used for some in-depth analysis (determining the eigenvalue spectrum, sparsity pattern,...).

2.5 Challenge: CD example

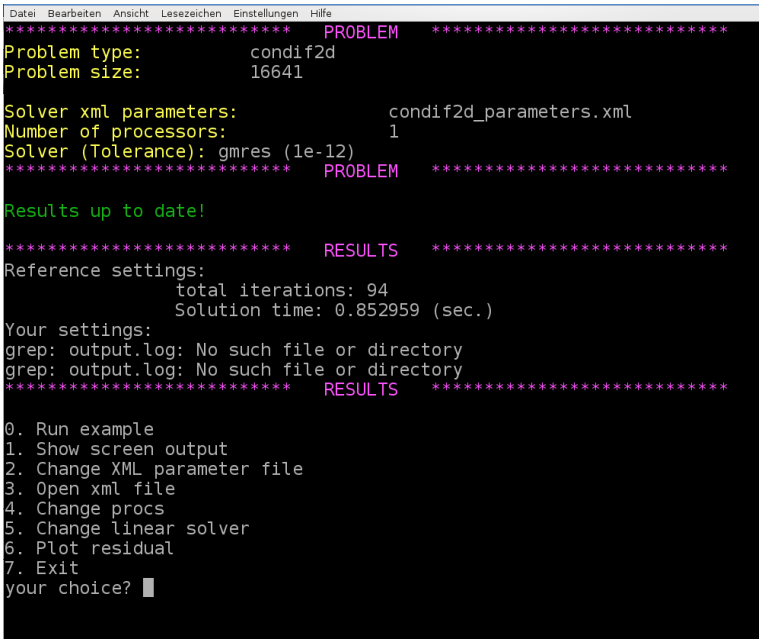
2.5.1 Practical example

Often one has only very rough information about the linear system that is supposed to be effectively solved using iterative methods with multigrid preconditioners. Therefore, it is highly essential to gain some experience with the solver and preconditioner parameters and learn to optimize the multigrid parameters just by looking at the convergence behavior of the linear solver.

Here, we consider a convection-diffusion example with 16641 degrees of freedom. No further information is provided (geometry, discretization technique, ...).

2.5.2 User-interface

Run the *hands-on.sh* script and choose the option 4 for the convection-diffusion example. The script automatically generates a XML file with reference multigrid parameters which are far from being optimal.



When using the reference settings for the multigrid preconditioner we need 94 linear iterations. The challenge is to find optimized multigrid settings which results in a significantly lower number of linear iterations and – even more important – a lower computational time.

Note: Please notice that we have automatically chosen GMRES as solver as the linear systems arising from convection-diffusion problems are non-symmetric (due to the convective term). A CG methods would not converge.

Exercise 1

Open the *condif2d_parameters.xml* file by pressing option 3. Try to find optimized multigrid settings using your knowledge from the previous tutorials. Save the file and rerun the example (using option 0). Compare your results with the reference results. With option 6 you can plot the convergence of the relative residual of the iterative solver (for comparison).

2.5.3 General hints

There is a very simple strategy for optimizing the solver and preconditioner parameters iteratively that works for many examples surprisingly well.

Linear solver settings

The parameters for the linear solver usually are fixed. Just make sure that you consider the non-symmetry in the choice of your iterative method and choose the solver tolerance in a reasonable way. Before you think about finding good preconditioner parameters you should be absolutely sure that your linear solver is chosen appropriately for your problem.

General multigrid settings

Next, one should choose the multigrid settings. This includes the desired number of multigrid levels and the stopping criterion for the coarsening process. An appropriate choice here is mainly dominated by the size of the problem and the discretization. The multigrid parameters should be chosen such that one obtains a reasonably small problem on the coarsest level which is solved directly.

Transfer operators

Then, one should think about the transfer operators. In the symmetric case one can try smoothed aggregation transfer operators. If unsure, the non-smooth transfer operators always should be a safe and robust starting point.

Level smoothers

Once the multigrid skeleton is fixed by the choice of transfer operators one can start with optimizing the level smoothers. When using relaxation based level smoothers one should first try different smoothing parameters and increase the number of smoothing sweeps only when necessary.

Fine tuning

Sometimes it is very helpful to have a look at the multigrid matrices. First of all, one should check whether the aggregation is working properly. This can be done by checking the screen output for the coarsening rate and the aggregation details (this is often the only way to do it if aggregates cannot be visualized due to missing node coordinates). If there is some problem with the aggregation one should try to adapt the aggregation parameters. Here it might make sense to export the coarse level matrices first and study their properties. For finding aggregation parameters one should, e.g., check the number of non-zeros in each row and choose the minimum aggregation size accordingly.

2.6 XML interface for advanced users

This tutorial introduces the more advanced (and more flexible) XML interface that can be used for setting up multigrid hierarchies in MueLu. Again we use the 2D Laplace problem on a 50×50 mesh as introduced in *Example problem*. That is, in the **hands-on.py** script you have to choose option 0 for the problem type.

2.6.1 One-level method

Before applying a multigrid method as solver, we start with a simple Jacobi iteration as solver and look at the error. By setting the maximum number of multigrid levels to 1 and using a Jacobi smoother as coarse solver we obtain a pseudo multigrid method which corresponds to a simple Jacobi iteration.

Listing 2.6: ../../../test/tutorial/s2_adv_a.xml

```
<ParameterList name="MueLu">

  <!-- Factory collection -->
  <ParameterList name="Factories">

    <!-- Note that ParameterLists must be defined prior to being used -->
    <ParameterList name="myJacobi">
      <Parameter name="factory"                                type="string" value=
↪"TrilinosSmoother"/>
      <Parameter name="type"                                   type="string" value=
↪"RELAXATION"/>
      <ParameterList name="ParameterList">
        <Parameter name="relaxation: type"                     type="string" value="Jacobi"/
↪>
        <Parameter name="relaxation: sweeps"                   type="int"    value="1"/>
        <Parameter name="relaxation: damping factor"           type="double" value="0.9"/>
      </ParameterList>
    </ParameterList>
  </ParameterList>

  <!-- Definition of the multigrid preconditioner -->
  <ParameterList name="Hierarchy">

    <Parameter name="max levels"                               type="int"      value="1"/>
    <Parameter name="coarse: max size"                         type="int"      value="10"/>
    <Parameter name="verbosity"                                type="string"   value="Low"/>

    <ParameterList name="All">
```

(continues on next page)

(continued from previous page)

```
      <Parameter name="Smoother"                                type="string"  value=
↪"myJacobi"/>
      <Parameter name="CoarseSolver"                            type="string"  value=
↪"myJacobi"/>
    </ParameterList>

  </ParameterList>
</ParameterList>
```

The advanced XML format is more hierarchical in the structure. Each XML file in the advanced format consists of two major blocks. First there is a set of "Factory" blocks which describe the building blocks within your multigrid methods. In above example there is only one building block specified for the Jacobi method. Each building block needs a (unique) name. In above example the building block has the name **myJacobi**. It is a factory of type **TrilinosSmoother** and describes a damped Jacobi method as declared by the internal parameters.

Later we will see examples for other building blocks describing transfer operators or the aggregation strategy. In the "Factories" list, the user has to declare all building blocks of the multigrid method that are used for the setup. The user cannot specify all building blocks involved in the setup process. MueLu will take care of that and use default building blocks for all parts of the setup process where the user makes no explicit statement. This way the user only has to describe what he explicitly needs.

It is not sufficient just to declare some building blocks. One also has to register them in the setup process. This is done in the second part of the XML file. The so-called **Hierarchy** block describes the setup process. First there are some basic multigrid parameters that are well-known from the easy XML interface (cf. the previous tutorials). Then, there is an additional list of parameters **All** which encapsulates the information which factory block is responsible to provide certain data. In above example you can see, that the building block **myJacobi** shall be used both for the level smoother and the coarse solver. Since it is only a 1 level problem it would be sufficient to define a coarse solver only. The name of the parameter list **All** can be chosen by the user. It basically describes the user-specified parts of the setup process for all multigrid levels. In this case we just overwrite the internal default factories both for the level smoother and the coarse solver by our Jacobi smoother.

Note: Be aware that we can have different parameter list sets for different levels, that is, we can use different factories on certain levels.

Exercise 1

Run the **hands-on.py** script and choose the XML file **../../../test/tutorial//s2_adv_a.xml** which contains above XML parameters. Use only 1 processor and visualize the error for an increasing number of multigrid cycles (e.g. 1, 5, 10, 30, 100). What do you observe?

Exercise 2

Note that the relaxation based smoothers are based on a Schwarz method (see Ifpack documentation). Repeat above steps using 2 processors. What do you observe in the error plots?

2.6.2 Multigrid method

The next step is to introduce a full multigrid algorithm. First one should increase the number of multigrid levels. Second, we switch to a direct solver on the coarsest level.

Exercise 3

Create your own copy of the **../../../test/tutorial/s2_adv_a.xml** parameter file. Adapt it to obtain a 3 level multigrid method. Check how this affects the error plots.

Exercise 4

Change to a direct solver on the coarsest level. You can do this by using **<Parameter name="CoarseSolver" type="string" value="DirectSolver"/>** in the **Hierarchy** block of the xml file. Check the output of the multigrid hierarchy.

2.6.3 Level smoothers

Next, we give some building blocks for different types of level smoothers that you can use. Note that all these xml blocks can be put into the **Factories** block of the advanced MueLu XML file format. Then you can use them by adding the corresponding link into the **Hierarchy** block using the name of the parameter block.

Standard level smoothers

Here is a list of the standard level smoothers and how to define them in the XML format. All these standard smoothers are generated by the **TrilinosSmoother** factory class (set in the **factory** parameter in the xml snippets below).

- Chebyshev smoother:

```
<ParameterList name="Chebyshev">
<Parameter name="factory"                    type="string" value=
↪"TrilinosSmoother"/>
<Parameter name="type"                              type="string" value=
↪"CHEBYSHEV"/>

<ParameterList name="ParameterList">
        <Parameter name="chebyshev: degree"                type="int"    ␣
↪value="2"/>
        <Parameter name="chebyshev: ratio eigenvalue"      type="double" ␣
↪value="20"/>
        <Parameter name="chebyshev: min eigenvalue"        type="double" ␣
↪value="1.0"/>
        <Parameter name="chebyshev: zero starting solution" type="bool"   ␣
↪value="true"/>
</ParameterList>
</ParameterList>
```

- Jacobi smoother:

```
<ParameterList name="myJacobi">
        <Parameter name="factory"                          type="string" value=
↪"TrilinosSmoother"/>
        <Parameter name="type"                                  type="string"␣
↪value="RELAXATION"/>
                <ParameterList name="ParameterList">
                <Parameter name="relaxation: type"                  type=
↪"string" value="Jacobi"/>
                <Parameter name="relaxation: sweeps"                type=
↪"int"    value="1"/>
                <Parameter name="relaxation: damping factor"        type=
↪"double" value="0.9"/>
                </ParameterList>
        </ParameterList>
```

- Gauss-Seidel smoother variants:

```
<ParameterList name="SymGaussSeidel">
        <Parameter name="factory"                          type="string" value=
↪"TrilinosSmoother"/>
        <Parameter name="type"                                  type="string"␣
↪value="RELAXATION"/>
        <ParameterList name="ParameterList">
        <Parameter name="relaxation: type"                  type="string"␣
↪value="Symmetric Gauss-Seidel"/>
                <Parameter name="relaxation: sweeps"                type=
↪"int"    value="1"/>
                <Parameter name="relaxation: damping factor"        type=
↪"double" value="1.0"/>
</ParameterList>
</ParameterList>
```

```
<ParameterList name="ForwardGaussSeidel">
        <Parameter name="factory"                           type="string"␣
↪value="TrilinosSmoother"/>
        <Parameter name="type"                                  type="string"␣
↪ value="RELAXATION"/>

        <ParameterList name="ParameterList">
                <Parameter name="relaxation: type"                  type=
↪"string"  value="Gauss-Seidel"/>
        <Parameter name="relaxation: backward mode"         type="bool"   ␣
↪value="false"/>
        <Parameter name="relaxation: sweeps"                type="int"    ␣
↪value="2"/>
                <Parameter name="relaxation: damping factor"        type=
↪"double"  value="1"/>
</ParameterList>
</ParameterList>
```

```
<ParameterList name="BackwardGaussSeidel">
        <Parameter name="factory"                           type="string"␣
```

(continues on next page)

(continued from previous page)

```
↪value="TrilinosSmoother"/>
        <Parameter name="type"                                   type="string"␣
↪ value="RELAXATION"/>

        <ParameterList name="ParameterList">
                <Parameter name="relaxation: type"                  type=
↪"string"  value="Gauss-Seidel"/>
        <Parameter name="relaxation: backward mode"         type="bool"   ␣
↪value="true"/>
                <Parameter name="relaxation: sweeps"                type=
↪"int"     value="2"/>
                <Parameter name="relaxation: damping factor"        type=
↪"double"  value="1"/>
</ParameterList>
</ParameterList>
```

- ILU smoothers:

```
<ParameterList name="IfpackILU">
        <Parameter name="factory" type="string" value="TrilinosSmoother"/>
        <Parameter name="type" type="string" value="ILU"/>
        <ParameterList name="ParameterList">
                <Parameter name="fact: level-of-fill" type="int" value="0"/>
        </ParameterList>
        </ParameterList>
```

Above listing shows how to create an ILU(0) level smoother (using Ifpack). Please refer to the Ifpack documentation of all parameters on how to choose, e.g. overlapping.

Note: There is an inconsistency between **Ifpack** (for the **Epetra** stack) and **Ifpack2** (for the **Tpetra** stack) with respect to the **type** parameter in above listing. Not all types of ILU methods available in **Ifpack** are available in **Ifpack2** and vice versa. **Ifpack2** has an implementation of ILUT (**type = ILUT**), but there ILUT parameters may also be slightly different. Please check and adapt the parameters if you switch between the **Epetra** and **Tpetra** stack. Please refer to the **Ifpack** and **Ifpack2** documentation for the details.

For Ifpack2 ILUT you can use, e.g., the following settings

```
<ParameterList name="Ifpack2ILUT">
  <Parameter name="factory" type="string" value="TrilinosSmoother"/>
  <Parameter name="type" type="string" value="ILUT"/>
  <ParameterList name="ParameterList">
    <Parameter name="fact: ilut level-of-fill" type="int" value="0"/>
  </ParameterList>
</ParameterList>
```

In Ifpack the parameters might be

```
<ParameterList name="IfpackILUT">
<Parameter name="factory" type="string" value="TrilinosSmoother"/>
<Parameter name="type" type="string" value="ILUT"/>
<ParameterList name="ParameterList">
<Parameter name="fact: ilut level-of-fill" type="double" value="0.0"/>
```

(continues on next page)

(continued from previous page)

```
</ParameterList>
</ParameterList>
```

Exercise 5

Pick out one level smoother from above and use them for your problem. Note that you may have to adapt the **relaxation: damping factor** for reasonable results.

Level smoothers for the Epetra and Tpetra stack

Generally, the **TrilinosSmoother** factory class tries to provide the same set of standard level smoothers for the **Epetra** and **Tpetra** stack as far as possible. It uses **Ifpack** or **Ifpack2** under the hood. **MueLu** partially tries to internally translate the parameters from **Ifpack2** to **Ifpack**, but this is not always possible and error-prone. The available values for the **type** parameter are **RELAXATION**, **CHEBYSHEV**, **ILUT**, **RILUK** and **ILU**.

Note: In order to define a multigrid hierarchy that is working for both the **Epetra** and **Tpetra** stack it is recommended to use **CHEBYSHEV** or **RELAXATION** based smoothers.

The **TrilinosSmoother** factory may provide support more types of smoothers, but these have to be considered as experimental and often are only available for the **Tpetra** branch. Please check the source code for a list of all available smoothers and options. More advanced smoothing strategies will also introduced later in this guide (e.g. *Line-smoohing*).

2.6.4 Advanced features

MueLu allows full control over the behavior of the multigrid levels. Here, we demonstrate the capabilities of MueLu using the level smoothers. Take a look at the following example XML parameter list

Listing 2.7: ../../../test/tutorial/s2_adv_b.xml

```
<ParameterList name="MueLu">

  <!-- Factory collection -->
  <ParameterList name="Factories">

    <!-- Note that ParameterLists must be defined prior to being used -->
    <ParameterList name="BackwardGaussSeidel">
      <Parameter name="factory"                          type="string" value=
↪"TrilinosSmoother"/>
      <Parameter name="type"                                  type="string"  value=
↪"RELAXATION"/>

      <ParameterList name="ParameterList">
        <Parameter name="relaxation: type"                    type="string"  value="Gauss-
↪Seidel"/>
        <Parameter name="relaxation: backward mode"           type="bool"    value="true"/>
        <Parameter name="relaxation: sweeps"                  type="int"     value="50"/>
        <Parameter name="relaxation: damping factor"          type="double"  value="0.6"/>
```

(continues on next page)

(continued from previous page)

```
    </ParameterList>
  </ParameterList>
</ParameterList>

<!-- Definition of the multigrid preconditioner -->
<ParameterList name="Hierarchy">

  <Parameter name="max levels"                              type="int"      value="4"/>
  <Parameter name="coarse: max size"                        type="int"      value="10"/>
  <Parameter name="verbosity"                               type="string"   value="Low"/>

  <ParameterList name="Finest">
    <Parameter name="PreSmoother"           type="string" value="BackwardGaussSeidel"/>
    <Parameter name="PostSmoother"          type="string" value="NoSmoother"/>
    <Parameter name="CoarseSolver"          type="string"  value="DirectSolver"/>

  </ParameterList>

  <ParameterList name="Remaining">
    <Parameter name="startLevel"            type="int" value="1"/>
    <Parameter name="PreSmoother"           type="string" value="NoSmoother"/>
    <Parameter name="PostSmoother"          type="string" value="BackwardGaussSeidel"/>
    <Parameter name="CoarseSolver"          type="string"  value="DirectSolver"/>
  </ParameterList>

</ParameterList>
</ParameterList>
```

You can find the parameters in **../../../test/tutorial/s2_adv_b.xml**. We have one building block **BackwardGaussSeidel** representing the level smoother that we want to use in our multigrid hierarchy. As one can see from the **Hierarchy** block we request a 4 level multigrid method. There are two blocks called **Finest** and **Remaining** describing the behavior of the different multigrid levels. Note the **startLevel** parameter in the block **Remaining**. This parameter is missing in the **Finest** block (where it is assumed to be the default value which is zero). That is, in this example we use the backward Gauss-Seidel method as pre-smoother on the finest level (note the keyword **NoFactory** for **PostSmoother**). The parameter **startLevel=1** in the **Remaining** block means that for level 1 and all coarser levels (unless there is another block with **startLevel > 1**) the building blocks from **Remaining** shall be used for the multigrid setup. That is, on the multigrid levels 1 and 2 the backward Gauss-Seidel method is used for post smoothing only. The corresponding multigrid hierarchy has the form

Warning: printScreenOutput{s2_adv_b.txt_3.fragment_3.fragment} -> Also replace in Exercise 6.

Exercise 6

Create an XML file in advanced format which produces the following multigrid layout printScreenOutput{s2_adv_c.txt_3.fragment_3.fragment}.

Hint: create a copy of the file **../../../test/tutorial/s2_adv_b.xml** and extend it accordingly. A possible solution can be found in **../../../test/tutorial/s2_adv_c.xml**.

2.7 MueLu factories for transfer operators

For this example, we reuse the **Recirc2D** example as introduced in *Test example*. The resulting linear systems are (slightly) non-symmetric and classical smoothed aggregation methods may be able to solve the problem but are not optimal in sense of convergence.

2.7.1 Multigrid setup phase - algorithmic design

Smoothed aggregation based algebraic multigrid methods originally have not been designed for non-symmetric linear systems. Inappropriately smoothed transfer operators may significantly deteriorate the convergence rate or even break convergence completely.

Unsmoothed transfer operators

Warning: Insert and link missing figures

Before we introduce smoothed aggregation methods for non-symmetric linear systems, we first go back one step and demonstrate how to use non-smoothed transfer operators which are eligible for non-symmetric linear systems. Figure *XML Interface* gives a simplified example how to build the coarse level matrix A_c using the fine level matrix A only. First, we "somehow" build aggregates using the information of the fine level matrix A. The aggregates are then used to build the tentative non-smoothed prolongation operator. The restrictor is just the transpose of the (tentative) prolongator and finally the coarse level matrix A_c is calculated by the triple product $A_c = RAP$.

In Figure *XML Interface*, the **SaPFactory** has been added after the **TentativePFactory**. Therein the non-smoothed transfer operator from the **TentativePFactory** is smoothed using information of the fine level matrix A. This transfer operator design is used per default when the user does not specify its own transfer operator design. The default settings are optimal for symmetric positive definite systems. However, they might be problematic for our non-symmetric problem.

Smoothed transfer operators for non-symmetric systems

Warning: Insert missing figures

In case of non-symmetric linear systems it is $A \neq A^T$. Therefore it is a bad idea just to use the transposed of the smoothed prolongation operator for the restrictor. Let $\widehat{P}$ be the non-smoothed tentative prolongation operator. Then the smoothed prolongation operator P is built using

$$P = \big(I - \omega A\big)\widehat{P}$$

with some reasonable smoothing parameter $\omega > 0$. The standard restrictor is

$$R = P^T = \widehat{P}^T - \omega \widehat{P}^T A^T = \widehat{P}^T\big(I - \omega A^T\big).$$

That is, the restrictor would be smoothed using the information of A^T. However, for non-symmetric systems we want to use the information of matrix A for smoothing the restriction operator, too. The restriction operator shall we built by the formula

$$R = P^T = \widehat{P}^T - \omega \widehat{P}^T A.$$

This corresponds to apply the same smoothing strategy to the non-smoothed restriction operator $\widehat{R} = \widehat{P}^T$, which is applied to the (tentative) prolongation operator with using A^T as input instead of matrix A. Figure

muelu_factories_for_transfer_operators/figure_simpledesignpamg shows the changed factory design. The dashed line denotes, that the same smoothing strategy is used than for the prolongation operator. The concept is known as Petrov-Galerkin smoothed aggregation approach in the literature. A more advanced transfer operator smoothing strategy for non-symmetric linear systems that is based on the Petrov-Galerkin approach is described in[1]. Another approach based on SchurComplement approximations can be found in[2].

Insert missing figures here

2.7.2 XML Interface

Unsmoothed transfer operators

To construct a multigrid hierarchy with unsmoothed transfer operators one can use the following XML file (stored in ../../../test/tutorial/s3a.xml)

```
<ParameterList name="MueLu">

  <!-- Factory collection -->
  <ParameterList name="Factories">

    <ParameterList name="UncoupledAggregationFact">
      <Parameter name="factory"              type="string" value=
↪"UncoupledAggregationFactory"/>
      <Parameter name="aggregation: ordering"                   type="string" value=
↪"natural"/>
      <Parameter name="aggregation: max selected neighbors"            type="int"   ␣
↪value="0"/>
      <Parameter name="aggregation: min agg size"           type="int"    value="4"/>
    </ParameterList>

    <ParameterList name="myTentativePFact">
      <Parameter name="factory"                        type="string" value=
↪"TentativePFactory"/>
    </ParameterList>

  </ParameterList>

  <!-- Definition of the multigrid preconditioner -->
  <ParameterList name="Hierarchy">

    <Parameter name="use kokkos refactor"                  type="bool"     value="false
↪"/>
    <Parameter name="max levels"                           type="int"      value="10"/>
    <Parameter name="coarse: max size"                     type="int"      value="10"/>
    <Parameter name="verbosity"                            type="string"   value="High"/
↪>

    <ParameterList name="All">
      <Parameter name="Aggregates"           type="string"   value=
```

(continues on next page)

[1] Sala, M. and Tuminaro, R. S., A new Petrov-Galerkin Smoothed Aggregation Preconditionerfor nonsymmetric Linear Systems, SIAM J. Sci. Comput., 2008, 31, p. 143–166

[2] Wiesner, T. A., Tuminaro, R. S., Wall, W. A. and Gee, M. W., Multigrid transfers fornonsymmetric systems based on Schur complements and Galerkin projections., Numer.Linear Algebra Appl., 2013, doi: 10.1002/nla.1889

(continued from previous page)

```
↪"UncoupledAggregationFact"/>
      <Parameter name="Nullspace"                    type="string"   value=
↪"myTentativePFact"/>
      <Parameter name="P"                            type="string"   value=
↪"myTentativePFact"/>
      <Parameter name="CoarseSolver"                     type="string"   value=
↪"DirectSolver"/>
    </ParameterList>

  </ParameterList>
</ParameterList>
```

Beside the **TentiativePFactory** which is responsible to generate the unsmoothed transfer operators we also introduce the **UncoupledAggregationFactory** with this example. In the **Factories** section of the XML file, you find both an entry for the aggregation factory and the prolongation operator factory with its parameters. In the **Hierarchy** section the defined factories are just put in into the multigrid setup algorithm. That is, the factory with the name **UncoupledAggregationFact** is used to generate the **Aggregates** and the **myTentativePFact** is responsible for generating both the (unsmoothed) prolongation operator P and the (coarse) near null space vectors **Nullspace**.

Note: For some more details about the (hidden) **NullspaceFactory**, which is internally used to handle the null space information and the dependencies, the reader might refer to *Setup of block transfer operators*.

Note: One can also use the **Ptent** variable for registering a **TentativePFactory**. This makes the **TentativePFactory** somewhat special in its central role for generating an aggregation based multigrid hierarchy. MueLu is smart enough to understand that you want to use the near null space vectors generated by the factory registered as **Ptent** for setting up the transfer operators.

That is, the following code would explicitly use the **TentativePFactory** object that is created as **myTentativePFact**. Since no factory is specified for the prolongation operator **P**, MueLu would decide to use a smoothed aggregation prolongation operator (represented by the **SaPFactory**), which correctly uses the factory for **Ptent** for the unsmoothed transfers with all its dependencies.

```
<ParameterList name="MueLu">
<ParameterList name="Factories">
        <ParameterList name="myTentativePFact">
        <Parameter name="factory" type="string" value="TentativePFactory"/>
        </ParameterList>
</ParameterList>

<ParameterList name="Hierarchy">
        <ParameterList name="Levels">
        <Parameter name="Ptent" type="string" value="myTentativePFact"/>
        </ParameterList>
</ParameterList>
</ParameterList>
```

Exercise 1

Create a sublist in the **Factories** part of the XML file for the restriction operator factory. Use a **TransPFactory** which builds the transposed of **P** to generate the restriction operator **R**. Register your restriction factory in the **Hierarchy**

section to generate the variable **R**.

Smoothed aggregation for non-symmetric problems

Next, let's try smoothed transfer operators for the non-symmetric linear system and compare the results of the transfer operator designs. Take a look at the XML file (in **../../../test/tutorial/s3b.xml**).

```
<ParameterList name="MueLu">

  <!-- Factory collection -->
  <ParameterList name="Factories">

    <!-- Note that ParameterLists must be defined prior to being used -->

    <ParameterList name="UncoupledAggregationFact">
      <Parameter name="factory"                               type="string" value=
↪"UncoupledAggregationFactory"/>
      <Parameter name="aggregation: ordering"                 type="string" value="natural
↪"/>
      <Parameter name="aggregation: max selected neighbors" type="int"    value="0"/>
      <Parameter name="aggregation: min agg size"           type="int"    value="4"/>
    </ParameterList>

    <ParameterList name="myTentativePFact">
      <Parameter name="factory"                               type="string" value=
↪"TentativePFactory"/>
    </ParameterList>
    <ParameterList name="myProlongatorFact">
      <Parameter name="factory"                               type="string" value=
↪"SaPFactory"/>
      <Parameter name="P"                                     type="string" value=
↪"myTentativePFact"/>
      <Parameter name="sa: damping factor"                    type="double" value="1.0"/>
    </ParameterList>
    <ParameterList name="myTentRestrictorFact">
      <Parameter name="factory"                               type="string" value=
↪"TransPFactory"/>
      <Parameter name="P"                                     type="string" value=
↪"myTentativePFact"/>
    </ParameterList>
    <ParameterList name="mySymRestrictorFact">
      <Parameter name="factory"                               type="string" value=
↪"TransPFactory"/>
      <Parameter name="P"                                     type="string" value=
↪"myProlongatorFact"/>
    </ParameterList>
    <ParameterList name="myNonsymRestrictorFact">
      <Parameter name="factory"                               type="string" value=
↪"GenericRFactory"/>
      <Parameter name="P"                                     type="string" value=
↪"myProlongatorFact"/>
    </ParameterList>
```

(continues on next page)

(continued from previous page)

```
    <ParameterList name="SymGaussSeidel">
      <Parameter name="factory"                                type="string" value=
↪"TrilinosSmoother"/>
      <Parameter name="type"                                   type="string" value=
↪"RELAXATION"/>
      <ParameterList name="ParameterList">
        <Parameter name="relaxation: type"                     type="string" value=
↪"Symmetric Gauss-Seidel"/>
        <Parameter name="relaxation: sweeps"                   type="int"    value="10"/>
        <Parameter name="relaxation: damping factor"           type="double" value="0.8"/>
      </ParameterList>
    </ParameterList>

  </ParameterList>

  <!-- Definition of the multigrid preconditioner -->
  <ParameterList name="Hierarchy">

    <Parameter name="use kokkos refactor"                      type="bool"     value="false
↪"/>
    <Parameter name="max levels"                               type="int"      value="10"/>
    <Parameter name="coarse: max size"                         type="int"      value="10"/>
    <Parameter name="verbosity"                                type="string"   value="High"/
↪>

    <ParameterList name="All">
      <Parameter name="Smoother"                    type="string"   value="SymGaussSeidel
↪"/>
      <Parameter name="Aggregates"                  type="string"   value=
↪"UncoupledAggregationFact"/>
      <Parameter name="Nullspace"                   type="string"   value=
↪"myTentativePFact"/>
      <Parameter name="P"                           type="string"   value=
↪"myProlongatorFact"/>
      <Parameter name="R"                           type="string"   value=
↪"mySymRestrictorFact"/>
      <Parameter name="CoarseSolver"                type="string"   value="DirectSolver"/
↪>
    </ParameterList>

  </ParameterList>
</ParameterList>
```

The interesting part is the **Factories** section where several different factories for the restriction operator are defined

Description

- **myTentRestrictorFact:** just uses the transposed of the unsmoothed prolongator for restriction.
- **mySymRestrictorFact:** uses the transposed of the smoothed prolongator for restriction.
- **myNonsymRestrictorFact:** uses the special non-symmetric smoothing for the restriction operator (based on the

SaPFactory smoothing factory).

Note: The MueLu framework is very flexible and allows for arbitrary combinations of factories. However, be aware that the **TentativePFactory** cannot be used as input for the **GenericRFactory**. That is no problem since this combination not really makes sense. If you are using the **TentativePFactory** as your final prolongation operator you always have to use the **TransPFactory** for generating the restriction operators.

Exercise 2

Run the **Recirc2D** example with the different restriction operator strategies and compare the results for the iterative solver. What do you observe? What is the best choice for the transfer operators in the non-symmetric case?

Exercise 3

Change the **myProlongatorFact** from type **SaPFactory** to **PgPFactory**, which uses automatically calculated local damping factors instead of a global damping factor (with some user parameter **sa: damping factor**). Note that the **PgPFactory** might not accept the **sa: damping factor** parameter such that you have to comment it out (using **<!– ... –>**).

Exercise 4

Try to set up a multigrid hierarchy with unsmoothed transfer operators for the transition from the finest level to level 1 and then use smoothed aggregation for the coarser levels (starting from level 1).

2.7.3 Footnotes

2.8 Rebalancing - Hypergraph repartitioning

2.8.1 Basic concepts and parameters

Especially when using the uncoupled aggregation strategy it is essential to reduce the number of processors used on the coarser levels. A natural strategy is to make sure that each processor gets a minimum number of equations to solve and reduce the number of active processors accordingly.

In this tutorial we use a hypergraph-based repartitioning for the coarse level matrices A_c to rebalance the coarse level problems. The repartitioning algorithm is implemented in the Zoltan2 package of Trilinos. The advantage of the hypergraph-based repartitioning methods is, that they do not need additional information such as node coordinates and therefore are the consequent choice within algebraic multigrid preconditioners.

Note: Hypergraph partitioning algorithms as PHG are not available in the new Zoltan2 package of Trilinos, yet. Therefore we can use this type of repartitioning only in context of Epetra based applications. If you use the new templated Tpetra stack, you have to use repartitioning algorithms which are available in Zoltan2 such as RCB.

Repartitioning algorithms are a very wide field of research and can be very complicated. Here, we cannot go into details and just focus on how to use them. Basically there are only a few really important parameters that the user has to set properly:

Description

repartition: min rows per proc
The minimum number of rows each processor shall handle. This parameter is used to reduce the number of involved processors on the coarser levels. If for example the parameter value is chosen to be 1000 and the fine level problem has 10000 rows whereas the coarse level problem has 2000 rows, then the fine level problem is solved on not more than 10 processors at maximum and for the coarse level problem there are not more processors than at maximum 2 being used.

repartition: max imbalance
This parameter defines the maximum allowed imbalance ratio of nonzeros on all processors. If the value is set to 1.2, and there is one processor with more than 20% nonzeros compared to another processor, then the problem will be rebalanced.

repartition: start level
Start rebalancing on the given level and coarser levels. This allows to avoid the costs of rebalancing on the finer levels (where it is not really necessary).

2.8.2 Transfer operator design

Warning: Insert and link missing figures

Figure **ref{fig:rebalanceddesignpgamg}** gives the extended factory design for smoothed aggregation based AMG for non-symmetric linear systems with rebalancing enabled. Nothing has changed in the upper part where the non-rebalanced Galerkin product has been calculated using the *RAPFactory*. The coarse level matrix A_c as output from the *RAPFactory* then is checked for its partition and rebalanced.

The *AmalgamationFactory* amalgamates the matrix, i.e. it generates some mapping between the actual degrees of freedom and the corresponding nodes or supernodes. In fact the *AmalgamationFactory* is only important if there are more than one degree of freedom per node. Otherwise the mappings are trivial to build.

The *RepartitionHeuristicFactory* contains the rebalancing logic. Depending on the chosen repartitioning parameters, it determines the number of partitions (variable name *number of partitions*) for the coarse level problem. The *IsorropiaInterface* class first builds internally the graph of the coarse level matrix A_c using the information from the *AmalgamationInformation* and then calls the repartitioning algorithm from Zoltan through the Isorropia interface. The output is an amalgamated repartitioning information. Then the *RepartitionInterface* factory resembles the un-amalgamated repartitioning information which is put into the *RepartitionFactory*.

The *RepartitionFactory* creates the communication "plan" that is used to rebalance the transfer operators and the coarse level matrix.

Insert missing figure here

2.8.3 XML interface

The corresponding XML parameter file looks like

Listing 2.8: ../../../test/tutorial/s5a.xml

```
<ParameterList name="MueLu">

  <!-- Factory collection -->
  <ParameterList name="Factories">

    <ParameterList name="myTentativePFact">
      <Parameter name="factory"                                type="string" value=
↪"TentativePFactory"/>
    </ParameterList>
    <ParameterList name="myProlongatorFact">
      <Parameter name="factory"                                type="string" value=
↪"PgPFactory"/>
      <Parameter name="P"                                      type="string" value=
↪"myTentativePFact"/>
    </ParameterList>
    <ParameterList name="myRestrictorFact">
      <Parameter name="factory"                                type="string" value=
↪"GenericRFactory"/>
      <Parameter name="P"                                      type="string" value=
↪"myProlongatorFact"/>
    </ParameterList>

    <ParameterList name="myAggExportFact">
      <Parameter name="factory"                                type="string" value=
↪"AggregationExportFactory"/>
      <Parameter name="Output filename"                        type="string" value="aggs_
↪level%LEVELID_proc%PROCID.out"/>
    </ParameterList>

    <ParameterList name="myRAPFact">
      <Parameter name="factory"                                type="string" value=
↪"RAPFactory"/>
      <Parameter name="P"                                      type="string" value=
↪"myProlongatorFact"/>
      <Parameter name="R"                                      type="string" value=
↪"myRestrictorFact"/>
      <ParameterList name="TransferFactories">
        <Parameter name="Visualization"                        type="string" value=
↪"myAggExportFact"/>
      </ParameterList>
    </ParameterList>

    <!-- ======================  REPARTITIONING  ======================= -->
    <!-- amalgamation of coarse level matrix -->
    <ParameterList name="myRebAmalgFact">
      <Parameter name="factory"                          type="string" value=
↪"AmalgamationFactory"/>
```

(continues on next page)

(continued from previous page)

```
      <Parameter name="A"                                type="string" value="myRAPFact"/>
    </ParameterList>

    <ParameterList name="myRepartitionHeuristicFact">
      <Parameter name="factory"                          type="string" value=
↪"RepartitionHeuristicFactory"/>
      <Parameter name="A"                                type="string" value="myRAPFact"/>
      <Parameter name="repartition: min rows per proc"      type="int"    value="2000"/>
      <Parameter name="repartition: max imbalance"          type="double" value="1.1"/>
      <Parameter name="repartition: start level"            type="int"    value="1"/>
    </ParameterList>

    <ParameterList name="myIsorropiaInterface">
      <Parameter name="factory"                          type="string" value=
↪"IsorropiaInterface"/>
      <Parameter name="A"                                type="string" value="myRAPFact"/>
      <Parameter name="UnAmalgamationInfo"               type="string" value=
↪"myRebAmalgFact"/>
      <Parameter name="number of partitions"             type="string" value=
↪"myRepartitionHeuristicFact"/>
    </ParameterList>

    <ParameterList name="myRepartitionInterface">
      <Parameter name="factory"                          type="string" value=
↪"RepartitionInterface"/>
      <Parameter name="A"                                type="string" value="myRAPFact"/>
      <Parameter name="AmalgamatedPartition"             type="string" value=
↪"myIsorropiaInterface"/>
      <Parameter name="number of partitions"             type="string" value=
↪"myRepartitionHeuristicFact"/>
    </ParameterList>


    <ParameterList name="myRepartitionFact">
      <Parameter name="factory"                          type="string" value=
↪"RepartitionFactory"/>
      <Parameter name="A"                                type="string" value="myRAPFact"/>
      <Parameter name="Partition"                        type="string" value=
↪"myRepartitionInterface"/>
      <Parameter name="repartition: remap parts"         type="bool"   value="false"/>
      <Parameter name="number of partitions"             type="string" value=
↪"myRepartitionHeuristicFact"/>
    </ParameterList>

    <ParameterList name="myRebalanceProlongatorFact">
      <Parameter name="factory"                          type="string" value=
↪"RebalanceTransferFactory"/>
      <Parameter name="type"                             type="string" value="Interpolation
↪"/>
      <Parameter name="P"                                type="string" value=
↪"myProlongatorFact"/>
      <Parameter name="Nullspace"                        type="string" value=
```

(continues on next page)

(continued from previous page)

```
↪"myTentativePFact"/>
    </ParameterList>

    <ParameterList name="myRebalanceRestrictionFact">
      <Parameter name="factory"                          type="string" value=
↪"RebalanceTransferFactory"/>
      <Parameter name="type"                             type="string" value="Restriction"/
↪>
      <Parameter name="R"                                type="string" value=
↪"myRestrictorFact"/>
    </ParameterList>

    <ParameterList name="myRebalanceAFact">
      <Parameter name="factory"                          type="string" value=
↪"RebalanceAcFactory"/>
      <Parameter name="A"                                type="string" value="myRAPFact"/>
    </ParameterList>

    <!-- ========================  SMOOTHERS  ========================== -->
    <ParameterList name="SymGaussSeidel">
      <Parameter name="factory"                          type="string" value=
↪"TrilinosSmoother"/>
      <Parameter name="type"                             type="string" value="RELAXATION"/>
      <ParameterList name="ParameterList">
        <Parameter name="relaxation: type"               type="string" value="Symmetric␣
↪Gauss-Seidel"/>
        <Parameter name="relaxation: sweeps"             type="int"    value="20"/>
        <Parameter name="relaxation: damping factor"     type="double" value="1.2"/>
      </ParameterList>
    </ParameterList>

  </ParameterList>

  <!-- Definition of the multigrid preconditioner -->
  <ParameterList name="Hierarchy">

    <Parameter name="max levels"                           type="int"      value="4"/>
    <Parameter name="coarse: max size"                     type="int"      value="10"/>
    <Parameter name="verbosity"                            type="string"   value="High"/
↪>

    <ParameterList name="All">
      <Parameter name="Smoother"                           type="string"   value=
↪"SymGaussSeidel"/>
      <Parameter name="Nullspace"                          type="string"   value=
↪"myRebalanceProlongatorFact"/>
      <Parameter name="P"                                  type="string"   value=
↪"myRebalanceProlongatorFact"/>
      <Parameter name="R"                                  type="string"   value=
↪"myRebalanceRestrictionFact"/>
      <Parameter name="A"                                  type="string"   value=
↪"myRebalanceAFact"/>
```

(continues on next page)

(continued from previous page)

```
      <Parameter name="Importer"                                type="string"   value=
↪"myRepartitionFact"/>
      <Parameter name="CoarseSolver"                            type="string"   value=
↪"DirectSolver"/>
    </ParameterList>

  </ParameterList>
</ParameterList>
```

It is stored in **../../../test/tutorial/s5a.xml**. In this example we define a smoothed aggregation transfer operator strategy (using the *PgPFactory*) for non-symmetric systems. The level smoother is chosen to be an over-relaxed symmetric Gauss–Seidel method. A direct solver is applied on the coarsest level. Please compare the building blocks in the xml file with Figure **ref{fig:rebalanceddesignpgamg}**. Be aware that the *Nullspace* variable now is also generated by the *myRebalanceProlongatorFact*.

Exercise 1 Choose option 1 in the problem menu of hands-on.py to run the *Laplace 2D* example on a 300×300 mesh. Change the solver to ../../../test/tutorial/s5a.xml. Use a reasonable number of processors. For demonstration purposes 4 processors should be fine for the 300×300 mesh. Run the example and check the screen output to see the effect of rebalancing. Try to visualize the ownership of the aggregates.

Note: The XML parameters in **../../../test/tutorial/s5a.xml** write out the aggregation data for debugging. See the next tutorial for some more background information on aggregation and debugging.

You should observe a multigrid hierarchy as follows

> **Warning:** Insert missing Screen output

printScreenOutput{s5a.txt_3.fragment_3.fragment}

2.9 Advanced concepts

As already mentioned in the beginning, MueLu is designed as a multigrid framework, and, even though initiated as an aggregation-based algebraic multigrid method, it can also be used for other kinds of coarsening methods. In this chapter we demonstrate the combination of a semi-coarsening method with an aggregation-based coarsening on the coarser levels. Semi-coarsening is combined with a line-smoothing method which then changes to a point-relaxation smoothing once no further semi-coarsening is possible. In both cases, the semi-coarsening and the line-smoothing, the key element here is the dynamic switch from one to the another coarsening or smoothing strategy during runtime.

2.9.1 Semi-coarsening

Basic idea

Assuming that you have a 3D problem which is based on an extruded 2D mesh, semi-coarsening might be an interesting option. That is, on the finer levels we apply a semi-coarsening transfer operator which basically reduces the problem to a pseudo 2D problem which then is handled by any other type of (smoothed aggregation based) transfer operator the usual way.

Factory layout without rebalancing

The semi-coarsening is provided by the **SemiCoarsenPFactory** for generating the semi-coarsening transfer operators in combination with the **LineDetectionFactory** which performs the line detection, i.e. it searches for the vertical node lines along the extrusion axis of the 2D mesh. Once all mesh layers are reduced to one by the **SemiCoarsenPFactory**, we switch to the aggregation-based standard coarsening process. There is a **TogglePFactory** which allows to switch back and forth between two different transfer operator strategies (such as semi-coarsening and standard aggregation-based coarsening). In principle, any combination of two different transfer operator strategies is allowed. However, the current implementation only contains decision criteria to switch between semi-coarsening and aggregation-based types of coarsening.

Figure advanced_concepts/figure_rebalancedtoggledesign shows a typical factory layout for the combination of semi-coarsening with a standard (non-smoothed) aggregation-based coarsening with repartitioning enabled. First, one can see the **TogglePFactory** which has knowledge about the two different transfer operator branches and makes a decision which transfer operator is used and provided to the restriction operator factory and the **RAPFactory**. Depending on the line detection algorithm in **LineDetectionFactory**, one might also need some information about the fine level mesh (such as the fine level coordinates). The user only has to provide the mesh on the finest level. On the coarser levels the **LineDetectionFactory** uses a standard ordering of the degrees of freedom which corresponds to a vertical node ordering.

Warning: Insert missing figure here

Factory layout with rebalancing

Figure advanced_concepts/figure_rebalancedtogglerebalancingdesign gives the extended factory layout when rebalancing is enabled. There is a new **ToggleCoordinatesTransferFactory** which is controlled by the **TogglePFactory** and appropriately generates the coarse coordinates depending on the used transfer operator. In case of semi-coarsening, the **SemiCoarsenPFactory** provides the coarse coordinates, which are then piped through the **ToggleCoordinatesTransferFactory**. In case of standard aggregation, the **CoordinatesTransferFactory** calculates the coarse coordinates using the aggregation information provided by the **AggregationFactory**. The coarse coordinate information is finally rebalanced by the **RebalanceTransferFactory** based on the rebalancing information provided by the **RepartitionFactory**.

Note: Note, that the **LineDetectionFactory** algorithm expects all nodes of a vertical line along the extrusion axis of the underlying 2D mesh to be owned by the same processor. Do not allow for rebalancing before semi-coarsening is complete! Alternatively, you can implement a new interface class to replace the **ZoltanInterface** which makes sure that the nodes are rebalanced appropriately. This could be easily done by rebalancing the **VertLineIds** info that is provided by the **LineDetectionFactory** and reconstruct the node based **Partition** data. The **Chosen P** variable provided by the **TogglePFactory** would tell the new interface class whether we are in semi-coarsening mode or in standard aggregation mode.

Warning: Insert missing figure here

The following listing shows exemplary the content of the xml file to set up a factory hierarchy similar to the one shown in Figure advanced_concepts/figure_rebalancedtogglerebalancingdesign.

In the **Factories** section, first the **SemiCoarsenPFactory** and the line detection algorithm are defined representing the left transfer operator branch. Next, the smoothed aggregation coarsening branch is defined in Part II together with an instance of the **TransferCoordinatesTransferFactory** for calculating the corresponding coarse coordinates. In Part III, the **TogglePFactory** is defined. It contains a **TransferFactories** sublist where all different coarsening branches (i.e., the semi-coarsening and aggregation-based coarsening) are defined. The corresponding prolongation operators are listed using the variable name **P** with a number between 1 and 9. In addition to the prolongation operator factories, one has also to declare the factories providing the coarse level null space and the *tentative* prolongation operator. In case of semi-coarsening this is the **SemiCoarsenPFactory** and for the aggregation-based coarsening this is the **TentativePFactory**. These factories are declared for generating the **Nullspace** variable with a number between 1 and 9 corresponding to the associated transfer operator branch. Similar for the *tentative* prolongation operators denoted by the variable **Ptent**.

Note: **SemiCoarsenPFactory** provides this information for compatibility reasons, even though there is no tentative prolongation operator for the geometrically generated **SemiCoarsenPFactory** operator. But the **TogglePFactory** is designed to be more general and allows for combining different kinds of smoothed prolongation operators. These need information about the non-smoothed transfer operators in the variable **Ptent**.

Similarly, a **ToggleCoordinatesTransferFactory** is declared with an internal list of all factories providing the coarse level coordinates. This is the previously defined **CoordinatesTransferFactory** for the standard aggregation-based coarsening branch and the **SemiCoarsenPFactory** for the semi-coarsening branch.

Part IV contains the standard factories for the rebalancing.

Finally, it is important to declare all necessary main factories in the **Hierarchy** section of the xml file.

```
<ParameterList name="MueLu">
<ParameterList name="Factories">

    <!-- =======================  PART I  ======================== -->
    <ParameterList name="myLineDetectionFact">
    <Parameter name="factory" type="string" value="LineDetectionFactory"/>
    <Parameter name="linedetection: orientation" type="string" value="coordinates"/>
    </ParameterList>

    <ParameterList name="mySemiCoarsenPFact1">
    <Parameter name="factory" type="string" value="SemiCoarsenPFactory"/>
    <Parameter name="semicoarsen: coarsen rate" type="int" value="6"/>
    </ParameterList>

    <!-- =======================  PART II  ======================== -->
    <ParameterList name="UncoupledAggregationFact2">
    <Parameter name="factory" type="string" value="UncoupledAggregationFactory"/>
    <Parameter name="aggregation: ordering" type="string" value="graph"/>
    <Parameter name="aggregation: min agg size" type="int"    value="9"/>
    </ParameterList>

    <ParameterList name="MyCoarseMap2">
```

(continues on next page)

(continued from previous page)

```
    <Parameter name="factory" type="string" value="CoarseMapFactory"/>
    <Parameter name="Aggregates" type="string" value="UncoupledAggregationFact2"/>
    </ParameterList>


    <ParameterList name="myTentativePFact2">
    <Parameter name="factory"     type="string" value="TentativePFactory"/>
    <Parameter name="Aggregates"  type="string" value="UncoupledAggregationFact2"/>
    <Parameter name="CoarseMap"   type="string" value="MyCoarseMap2"/>
    </ParameterList>

    <ParameterList name="mySaPFact2">
    <Parameter name="factory"     type="string" value="SaPFactory"/>
    <Parameter name="P"           type="string" value="myTentativePFact2"/>
    </ParameterList>

    <ParameterList name="myTransferCoordinatesFact">
    <Parameter name="factory"     type="string" value="CoordinatesTransferFactory"/>
    <Parameter name="CoarseMap"   type="string" value="MyCoarseMap2"/>
    <Parameter name="Aggregates"  type="string" value="UncoupledAggregationFact2"/>
    </ParameterList>

    <!-- ======================  PART III  ======================== -->

    <ParameterList name="myTogglePFact">
    <Parameter name="factory"              type="string" value="TogglePFactory"/>
    <Parameter name="semicoarsen: number of levels"       type="int" value="2"/>
    <ParameterList name="TransferFactories">
        <Parameter name="P1"               type="string" value="mySemiCoarsenPFact1"/>
        <Parameter name="P2"               type="string" value="mySaPFact2"/>
        <Parameter name="Ptent1"           type="string" value="mySemiCoarsenPFact1"/>
        <Parameter name="Ptent2"           type="string" value="myTentativePFact2"/>
        <Parameter name="Nullspace1"       type="string" value="mySemiCoarsenPFact1"/>
        <Parameter name="Nullspace2"       type="string" value="myTentativePFact2"/>
    </ParameterList>
    </ParameterList>

    <ParameterList name="myRestrictorFact">
    <Parameter name="factory"   type="string" value="TransPFactory"/>
    <Parameter name="P"         type="string" value="myTogglePFact"/>
    </ParameterList>

    <ParameterList name="myToggleTransferCoordinatesFact">
    <Parameter name="factory"   type="string" value="ToggleCoordinatesTransferFactory"/>
    <Parameter name="Chosen P"  type="string" value="myTogglePFact"/>
    <ParameterList name="TransferFactories">
        <Parameter name="Coordinates1" type="string" value="mySemiCoarsenPFact1"/>
        <Parameter name="Coordinates2" type="string" value="myTransferCoordinatesFact"/>
    </ParameterList>
    </ParameterList>

    <ParameterList name="myRAPFact">
```

(continues on next page)

(continued from previous page)

```
    <Parameter name="factory" type="string" value="RAPFactory"/>
    <Parameter name="P"       type="string" value="myTogglePFact"/>
    <Parameter name="R"       type="string" value="myRestrictorFact"/>
    <ParameterList name="TransferFactories">
        <Parameter name="For Coordinates" type="string" value=
↪"myToggleTransferCoordinatesFact"/>
    </ParameterList>
    </ParameterList>

    <!-- ========================  PART IV (Repartitioning)  ======================== -->
    <ParameterList name="myZoltanInterface">
    <Parameter name="factory"      type="string" value="ZoltanInterface"/>
    <Parameter name="A"            type="string" value="myRAPFact"/>
    <Parameter name="Coordinates"  type="string" value="myToggleTransferCoordinatesFact"/
↪>
    </ParameterList>

    <ParameterList name="myRepartitionFact">
    <Parameter name="factory"    type="string" value="RepartitionFactory"/>
    <Parameter name="A"          type="string" value="myRAPFact"/>
    <Parameter name="Partition"  type="string" value="myZoltanInterface"/>
    <Parameter name="repartition: min rows per proc"  type="int"    value="800"/>
    <Parameter name="repartition: max imbalance"      type="double" value="1.1"/>
    <Parameter name="repartition: start level"        type="int"    value="3"/>
    <Parameter name="repartition: remap parts"        type="bool"   value="false"/>
    </ParameterList>

    <ParameterList name="myRebalanceProlongatorFact">
    <Parameter name="factory"      type="string" value="RebalanceTransferFactory"/>
    <Parameter name="type"         type="string" value="Interpolation"/>
    <Parameter name="P"            type="string" value="myTogglePFact"/>
    <Parameter name="Coordinates"  type="string" value="myToggleTransferCoordinatesFact"/
↪>
    <Parameter name="Nullspace"    type="string" value="myTogglePFact"/>
    </ParameterList>

    <ParameterList name="myRebalanceRestrictionFact">
    <Parameter name="factory"      type="string" value="RebalanceTransferFactory"/>
    <Parameter name="type"         type="string" value="Restriction"/>
    <Parameter name="R"            type="string" value="myRestrictorFact"/>
    </ParameterList>

    <ParameterList name="myRebalanceAFact">
    <Parameter name="factory"      type="string" value="RebalanceAcFactory"/>
    <Parameter name="A"            type="string" value="myRAPFact"/>
    </ParameterList>
</ParameterList>

<!-- Definition of the multigrid preconditioner -->
<ParameterList name="Hierarchy">
    <Parameter name="max levels"       type="int"      value="6"/>
    <Parameter name="coarse: max size" type="int"      value="100"/>
```

(continues on next page)

(continued from previous page)

```
    <Parameter name="verbosity"        type="string"   value="High"/>
    <ParameterList name="All">
    <Parameter name="P"              type="string"   value="myRebalanceProlongatorFact"/>
    <Parameter name="Nullspace"      type="string"   value="myRebalanceProlongatorFact"/>
    <Parameter name="CoarseNumZLayers" type="string"   value="myLineDetectionFact"/>
    <Parameter name="LineDetection_Layers" type="string"   value="myLineDetectionFact"/>
    <Parameter name="LineDetection_VertLineIds" type="string"   value=
↪"myLineDetectionFact"/>
    <Parameter name="A"              type="string"   value="myRebalanceAFact"/>
    <Parameter name="Coordinates"    type="string"   value="myRebalanceProlongatorFact"/>
    <Parameter name="Importer"       type="string"   value="myRepartitionFact"/>
    <!--<Parameter name="R"          type="string"   value="myRebalanceRestrictionFact"/>
↪-->
</ParameterList>
</ParameterList>
</ParameterList>
```

Warning: Include the above XML file into testing.

2.9.2 Line-smoohing

General idea

Semi-coarsening should be combined with line-smoothing as the complementary smoothing operation. Whereas semi-coarsening coarsens, e.g., along the z-axis trying to produce a 2D representation of a 3D problem, the line-smoothing operates orthogonal to the coarsening and smoothes in the x- and y-direction and interprets all vertical z-layers technically as nodes in a pseudo 2D problem.

Usage

The following listing shows how to choose a Jacobi line smoother. The reader might compare the xml code snippets with Section *Level smoothers* for a detailed description of the different smoothers and parameters.

```
<ParameterList name="MueLu">
<ParameterList name="Factories">
<ParameterList name="mySmoother1">
    <Parameter name="factory"   type="string" value="TrilinosSmoother"/>
    <Parameter name="type"      type="string" value="LINESMOOTHING_BANDEDRELAXATION"/>
    <Parameter name="smoother: pre or post"        type="string" value="pre"/>
    <ParameterList name="ParameterList">
    <Parameter name="relaxation: type"           type="string" value="Jacobi"/>
    <Parameter name="relaxation: sweeps"         type="int"    value="2"/>
    <Parameter name="relaxation: damping factor" type="double" value="0.3"/>
    </ParameterList>
</ParameterList>
<ParameterList name="Hierarchy">
    <ParameterList name="All">
    <Parameter name="Smoother"    type="string"   value="mySmoother1"/>
```

(continues on next page)

(continued from previous page)

```
    </ParameterList>
</ParameterList>
</ParameterList>
```

Warning: Include the above XML file into testing.

The parameters are standard except of **type**. The standard choice would be **RELAXATION** for relaxation based smoothers. To use line-smoothing instead one has the following options:

Description

- [LINESMOOTHINGBANDEDRELAXATION] Use banded containers to store the local block associated with one vertical line.

This is the recommended variant as it saves memory and is usually faster. * [LINESMOOTHINGBLOCKEDRELAXATION] Use a dense matrix container to store the local block associated with one vertical line. This is the safe fallback variant. Use the **LINESMOOTHING_BANDEDRELAXATION** variant instead.

All the other parameters in the parameter sublist correspond to the usual parameters for relaxation based smoothers such as Jacobi, Gauss-Seidel or Symmetric Gauss-Seidel methods. Refer to Section *Level smoothers* or the MueLu user guide[1] for an overview of all available parameters.

2.9.3 Footnotes

2.10 Aggregation

This tutorial provides some background information about the aggregation process in MueLu. A very detailed description of the aggregation algorithms with all internal details can be found in[1].

2.10.1 Building aggregates

Warning: Ref and include missing figures

The aggregates are built using the graph of the fine level matrix A. The graph is generated by the **CoalesceDropFactory**. Since we still only restrict ourselves to scalar problems with one degree of freedom per node (*DofsPerNode=1*), the graph of the fine level matrix is trivial to build. Figure aggregation/figure_simpledesignaggregates shows the extended transfer operator design with the additional **CoalesceDropFactory**.

Insert figure here

Especially for anisotropic or non-symmetric problems, it may be advantageous to drop small entries from the graph of A and use a filtered graph for generating aggregates.

[1]

L. Berger-Vergiat, C. A. Glusa, G. Harper, J. J. Hu, M. Mayr, P. Ohm, A. Prokopenko, C. M. Siefert, R. S. Tuminaro, and T. A. Wiesner. MueLu User's Guide. Technical Report SAND2023-12265, Sandia National Laboratories, Albuquerque, NM (USA) 87185, 2023.

[1] Wiesner, T. A., Flexible Aggregation-based Algebraic Multigrid Methods for Contact and Flow Problems., PhD thesis, Technische Universität München, 2014

The following listing shows the definition of the **myCoalesceDropFactory** which drops all values of the fine level matrix A with the absolute value smaller than 0.01. Of course, the **myCoalesceDropFactory** has to be registered to generate the variable **Graph**, which is used by the aggregation factory. The **Graph** and the variable **DofsPerNode** generated by the **myCoalesceDropFactory** are needed as input by the **UncoupledAggregationFact**. Note that the aggregation routine always works on the node-based information instead of DOF-based information. Therefore, we first have to build the graph of A which then can be processed by the aggregation algorithm.

Listing 2.9: ../../../test/tutorial/s4a.xml

```
<ParameterList name="MueLu">

  <!-- Factory collection -->
  <ParameterList name="Factories">

    <!-- Note that ParameterLists must be defined prior to being used -->
    <ParameterList name="myCoalesceDropFact">
      <Parameter name="factory"                               type="string" value=
↪"CoalesceDropFactory"/>
      <Parameter name="lightweight wrap"                      type="bool"   value="true"/>
      <!-- for aggregation dropping -->
      <Parameter name="aggregation: drop tol"                 type="double" value="0.01"/>
    </ParameterList>

    <ParameterList name="UncoupledAggregationFact">
      <Parameter name="factory"                               type="string" value=
↪"UncoupledAggregationFactory"/>
      <Parameter name="aggregation: ordering"                 type="string" value="natural
↪"/>
      <Parameter name="aggregation: max selected neighbors" type="int"    value="0"/>
      <Parameter name="aggregation: min agg size"             type="int"    value="4"/>
      <Parameter name="Graph"                                 type="string" value=
↪"myCoalesceDropFact"/>
      <Parameter name="DofsPerNode"                           type="string" value=
↪"myCoalesceDropFact"/>
    </ParameterList>

    <ParameterList name="myTentativePFact">
      <Parameter name="factory"                               type="string" value=
↪"TentativePFactory"/>
    </ParameterList>
    <ParameterList name="myProlongatorFact">
      <Parameter name="factory"                               type="string" value=
↪"PgPFactory"/>
      <Parameter name="P"                                     type="string" value=
↪"myTentativePFact"/>
    </ParameterList>
    <ParameterList name="myRestrictorFact">
      <Parameter name="factory"                               type="string" value=
↪"GenericRFactory"/>
      <Parameter name="P"                                     type="string" value=
↪"myProlongatorFact"/>
    </ParameterList>

    <ParameterList name="myRAPFact">
```

(continues on next page)

(continued from previous page)

```
      <Parameter name="factory"                                type="string" value=
↪"RAPFactory"/>
      <Parameter name="P"                                      type="string" value=
↪"myProlongatorFact"/>
      <Parameter name="R"                                      type="string" value=
↪"myRestrictorFact"/>
    </ParameterList>

    <!-- =======================  SMOOTHERS  ======================== -->
    <ParameterList name="SymGaussSeidel">
      <Parameter name="factory"                                type="string" value=
↪"TrilinosSmoother"/>
      <Parameter name="type"                                   type="string" value=
↪"RELAXATION"/>
      <ParameterList name="ParameterList">
        <Parameter name="relaxation: type"                     type="string" value=
↪"Symmetric Gauss-Seidel"/>
        <Parameter name="relaxation: sweeps"                   type="int"    value="1"/>
        <Parameter name="relaxation: damping factor"           type="double" value="1.0"/>
      </ParameterList>
    </ParameterList>

  </ParameterList>

  <!-- Definition of the multigrid preconditioner -->
  <ParameterList name="Hierarchy">

    <Parameter name="max levels"       type="int"    value="10"/>
    <Parameter name="coarse: max size" type="int"    value="10"/>
    <Parameter name="verbosity"        type="string" value="High"/>

    <ParameterList name="All">
      <Parameter name="Smoother"       type="string" value="SymGaussSeidel"/>
      <Parameter name="Graph"          type="string" value="myCoalesceDropFact"/>
      <Parameter name="Aggregates"     type="string" value="UncoupledAggregationFact"/>
      <Parameter name="Nullspace"      type="string" value="myTentativePFact"/>
      <Parameter name="P"              type="string" value="myProlongatorFact"/>
      <Parameter name="R"              type="string" value="myRestrictorFact"/>
      <Parameter name="A"              type="string" value="myRAPFact"/>
      <Parameter name="CoarseSolver"   type="string" value="DirectSolver"/>
    </ParameterList>
  </ParameterList>
</ParameterList>
```

The listing shows how the **Graph** and the variable **DofsPerNode** generated by the coalescing factory **myCoalesceDropFact** are explicitly used as input for the aggregation routine. This is an example for a direct link of variables from output to the corresponding input. In addition, the **myCoalesceDropFact** is registered to produce the variable **Graph** in the *Hierarchy* section of the XML file. One should also register **myCoalesceDropFact** to produce the **DofsPerNode** information. In our case it is not really necessary, since all factories which rely on information from **DofsPerNode** get the information directly in the XML file (see also **myAggExportFact** in above listing). So, one has in general two possibilities to declare inter-factory dependencies. One can either explicitly describe the input for each factory (as demonstrated for the **Graph** in **UncoupledAggregationFact**) or use the default factories (provided either by MueLu or explicitly set by the user in the *Hierarchy* section). MueLu uses the following ordering: first, the explicit input de-

pendencies within the factories are used by MueLu. If a user does not define input variables (e.g., there is no input for **Aggregates** in **myTentativePFact**), MueLu checks whether there is a default factory for the data variable set in the *Hierarchy* section (in above listing it will find **UncoupledAggregationFact** to be responsible to provide the **Aggregates**). Otherwise MueLu will use some internal default factory.

For demonstration purposes, we also introduced a **RAPFactory** which makes use of the user-defined transfer factories **myProlongatorFact** as well as **myRestrictorFact**. The full XML file can be found in **../../../test/tutorial/s4a.xml** (see `../../../test/tutorial/s4a.xml`).

2.10.2 Visualization of aggregates

For debugging purposes, it can be very useful to visualize the aggregates. MueLu provides several ways to graphically visualize the coarsening process and the aggregates. In order to visualize aggregates one needs the coordinates of the mesh nodes as geometric information. Whereas the user is expected to provide the mesh node coordinates in the **Coordinates** variable on the finest level, we have to transfer the mesh information to the coarser levels.

General data transfer

The **RAPFactory** is responsible to generate the coarse level matrix A_c, which is beside the transfer operators P and R the only information needed for an algebraic multigrid method to further coarsen the problem. However, in some situations the user might be able to transfer further user-specific information to coarser levels. The **RAPFactory** can be extended by further helper transfer functions. These helper factories have to be registered in the **RAPFactory** and then are called during the multigrid setup phase after the Galerkin product has been built. A typical example for such a helper factory is the **CoordinatesTransferFactory**, which transfers the **Coordinates** variable to the coarser level and builds coarse node coordinates using the aggregation information. Further examples for special helper transfer factories are the **AggregationExportFactory** and the **CoarseningVisualizationFactory**, hich are all introduced in the following sections.

Use the AggregationExportFactory

Insert figure here

If you have an aggregation-based algebraic multigrid method, the **AggregationExportFactory** is the first choice to export the aggregates for visualization purposes.

The file **../../../test/tutorial/s4av.xml** extends the multigrid hierarchy from file **../../../test/tutorial/s4a.xml** for support of visualization of aggregates. The important changes are the following:

```
<ParameterList name="myAggExportFact">
<Parameter name="factory"         type="string" value="AggregationExportFactory"/>
<Parameter name="aggregation: output filename"  type="string"
           value="aggs_level%LEVELID_proc%PROCID.out"/>
<Parameter name="aggregation: output file: agg style" type="string"
           value="Convex Hulls"/>
</ParameterList>

<ParameterList name="myCoordTransferFact">
<Parameter name="factory"         type="string" value="CoordinatesTransferFactory"/>
</ParameterList>

<ParameterList name="myRAPFact">
<Parameter name="factory"                  type="string" value="RAPFactory"/>
<Parameter name="P"                        type="string" value="myProlongatorFact"/>
```

(continues on next page)

(continued from previous page)

```
<Parameter name="R"                          type="string" value="myRestrictorFact"/>
<ParameterList name="TransferFactories">
    <Parameter name="CoordinateTransfer"     type="string" value="myCoordTransferFact"/>
    <Parameter name="Visualization"          type="string" value="myAggExportFact"/>
</ParameterList>
</ParameterList>
```

Warning: Include the above xml file into testing.

The **AggregationExportFactory** acts as a small helper factory within the **RAPFactory**, which writes out some aggregation information to VTK files on the hard disk (see Figure aggregation/figure_simpledesignaggregatesvis). For visualization, the user has to provide the coordinates associated with the mesh nodes in the Coordinates** variable. The coordinates are transferred to the coarse level by the **CoordinatesTransferFactory**, which is also a helper transfer factory called by the **RAPFactory** as the **AggregationExportFactory**. The **CoordinatesTransferFactory** needs the aggregation information to build a coarse coordinate by using the midpoint of each aggregate. Note, that the **RAPFactory** accepts a sublist **TransferFactories** to register all the additional helper transfer factories which are called after the Galerkin product is calculated. The helper transfer factories are called in the ordering in which they are registered in the **RAPFactory**, but for this example the ordering is not important.

To complete the xml file, one has to declare the **CoordinatesTransferFactory** to be the default factory for producing **coordinates** by making the following statements in the **Hierarchy** sublist of the xml file.

```
<ParameterList name="Hierarchy">
  <Parameter name="max levels"       type="int"    value="10"/>
  <Parameter name="coarse: max size" type="int"    value="10"/>
  <Parameter name="verbosity"        type="string" value="High"/>

  <ParameterList name="All">
    <Parameter name="Smoother"       type="string" value="SymGaussSeidel"/>
    <Parameter name="Graph"          type="string" value="myCoalesceDropFact"/>
    <Parameter name="Aggregates"     type="string" value="UncoupledAggregationFact"/>
    <Parameter name="Nullspace"      type="string" value="myTentativePFact"/>
    <Parameter name="P"              type="string" value="myProlongatorFact"/>
    <Parameter name="R"              type="string" value="myRestrictorFact"/>
    <Parameter name="A"              type="string" value="myRAPFact"/>
    <Parameter name="CoarseSolver"   type="string" value="DirectSolver"/>
    <Parameter name="Coordinates"    type="string" value="myCoordTransferFact"/>
  </ParameterList>
</ParameterList>
```

Warning: Include the above xml file into testing.

This way, the **myCoordTransferFact** is declared as the default factory to generate the coarse coordinates in **Coordinates** on the coarser levels.

Note: The fine level coordinates have to be provided by the user in the **Coordinates** variable on the finest level. They are automatically used as input for the **CoordinatesTransferFactory** to produce the coarse level coordinates for level 1. On the coarser levels then the **CoordinatesTransferFactory** serves as input factory for being responsible to produce the coarse coordinates vector. Therefore, it is not a problem to declare **myCoordTransferFact** as generating factory for

the **Coordinates** on all multigrid levels, as per default input data on level 0 provided by the user has always precedence over factory-generated data.

Different aggregation parameters and the corresponding aggregates

Table 2.3: Aggregates with settings from **../../../test/tutorial/s4a.xml**

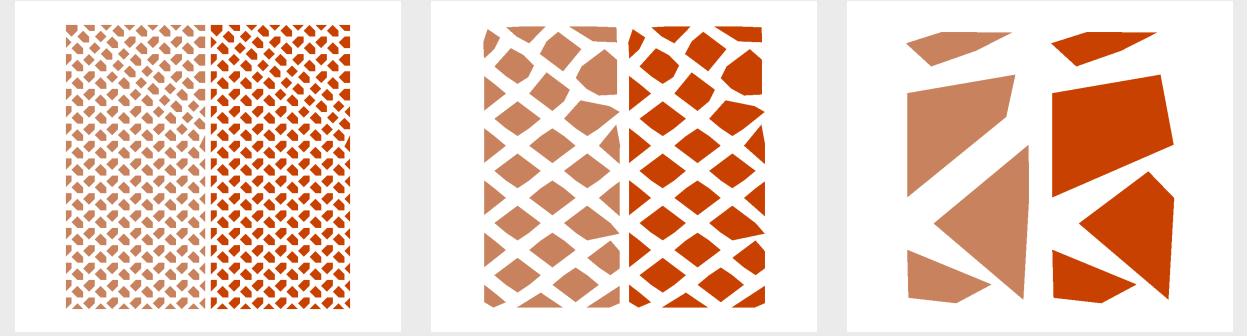
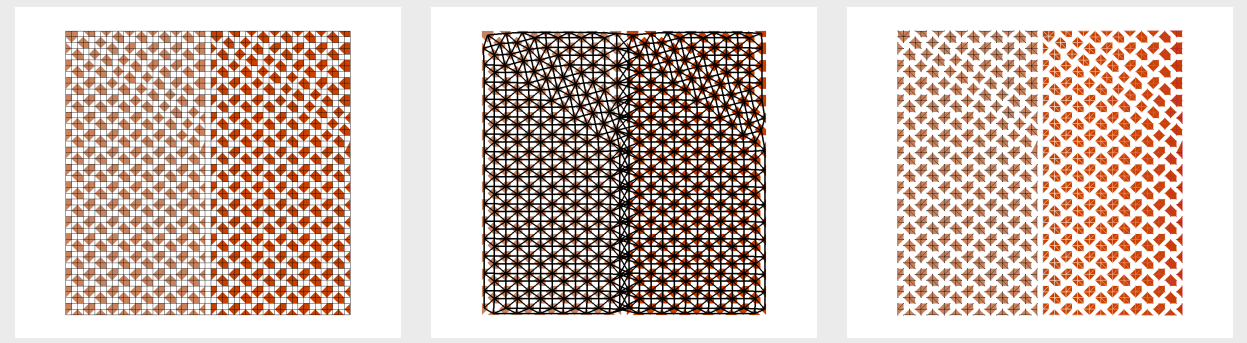


Table 2.4: Aggregates with settings from **../../../test/tutorial/s4b.xml** (default settings)



Exercise 1

Run the **Laplace 2D** example on a 50×50 mesh using the XML file **../../../test/tutorial/s4av.xml**. Open **paraview** and load the **aggs_level*_proc.out-master.pvtu** file.

Exercise 2

In the **../../../test/tutorial/s4av.xml** file, an uncoupled aggregation factory has been explicitly defined using with some user-chosen aggregation parameters. Make a copy of **../../../test/tutorial/s4av.xml** and use the default (uncoupled) aggregation routine that is provided by MueLu if no user-specified aggregation algorithm with parameters is prescribed. Which line in the XML file do you have to remove to obtain this behavior? Compare the results (screen output of aggregates, multigrid hierarchy). Try to visualize the aggregates.

In general it is a good idea to use the Hierarchy section to register the factories to generate the variables. It is very hard to declare all dependencies in the factory sections itself. In the worst case you declare, e.g., a **UncoupledAggregationFactory** and use it as input for the **TentativePFactory**, but forget to declare it also explicitly as input for the aggregation export factory **AggregationExportFactory**. If the **UncoupledAggregationFactory** is not declared as default for Aggregates in the **AggregationExportFactory**, the **AggregationExportFactory** will use default aggregates

provided by MueLu which are not identical to the aggregates usedfor building the transfer operators! Missing or wrong dependencies in the factory list are very hard to debug. Therefore one should always start with the *Hierarchy* section and only locally overwrite the dependencies where necessary.

2.10.3 Footnotes

2.11 Useful commands and debugging

2.11.1 Export level information

Of course, it is possible to export the multigrid hierarchy in matrix market format similar to *Export data* when using the advanced XML file format instead of the simple XML format.

To export the multigrid hierarchy one can use, e.g., the following parameters

```
<ParameterList name="MueLu">

  <!-- Factory collection -->
  <ParameterList name="Factories">
  </ParameterList>

  <!-- Definition of the multigrid preconditioner -->
  <ParameterList name="Hierarchy">

    <Parameter name="max levels"                          type="int"      value="3"/>
    <Parameter name="coarse: max size"                    type="int"      value="10"/>
    <Parameter name="verbosity"                           type="string"   value="Low"/>

    <ParameterList name="All">
    </ParameterList>

    <ParameterList name="DataToWrite">
      <Parameter name="Matrices" type="string" value="{0,1,2}"/>
      <Parameter name="Prolongators" type="string" value="{1,2}"/>
      <Parameter name="Restrictors" type="string" value="{1,2}"/>
    </ParameterList>
  </ParameterList>
</ParameterList>
```

2.11.2 Dependency trees

For debugging it can be extremely helpful to automatically generate the dependency tree of the factories for a given XML file. However, it shall be noticed that even with a graphical dependency tree it might be hard to find the missing links and dependencies without a sufficient understanding of the overall framework.

To write out the dependencies you just have to put in the **dependencyOutputLevel** parameter. The value gives you the fine level index that you are interested in (e.g., 1 means: print dependencies between level 1 and level 2).

Listing 2.10: ../../../test/tutorial/s6_dep.xml

```
<ParameterList name="MueLu">

  <!-- Factory collection -->
  <ParameterList name="Factories">
  </ParameterList>

  <!-- Definition of the multigrid preconditioner -->
  <ParameterList name="Hierarchy">

    <Parameter name="max levels"                              type="int"      value="3"/>
    <Parameter name="coarse: max size"                        type="int"      value="10"/>
    <Parameter name="verbosity"                               type="string"   value="Low"/>

    <ParameterList name="All">
    </ParameterList>

    <Parameter name="dependencyOutputLevel" type="int" value="1"/>

  </ParameterList>
</ParameterList>
```

After running the example you should find a file named **dep_graph.dot** in the current folder which you can transform into a graph using the **dot** utility from the graphviz package. Run, e.g. the commands

::

sed -i 's/label=Graph]/label="Graph"]/' dep_graph.dot sed -i 's/\"/"/g' dep_graph.dot sed -i 's/"</</' dep_graph.dot sed -i 's/>"/>/' dep_graph.dot dot -Tpng dep_graph.dot -o dep_graph.dot.png

in your terminal to obtain the graph as given in Figure *Visualization of aggregates using the CoarseningVisualizationFactory.*.

Note that the red arrows correspond to the fine level (level 1) and the blue arrows correspond to data on the coarse level (level 2).

Note: In case that the file **dep_graph.dot** is not generated you have to check the prerequisites. To be able to auto-generate the dependency graphs you have to compile MueLu with Boost enabled. Furthermore you have to set the **MueLu_ENABLE_Boost_for_real:BOOL = ON** defines flag in your configuration script. If these requirements are not fulfilled you should find the error message *Dependency graph output requires boost and MueLu_ENABLE_Boost_for_real* in the screen output of MueLu.

As one can see from the dependency output there are also some internal factories which have not been visualized in the Figures *XML Interface*. A good example is the **NullspaceFactory** which seems to build a dependency cycle with the **TentativePFactory**. In fact, the **NullspaceFactory** is a helper factory which allows to use the user-provided near null space vectors as input on the finest level. On the coarser levels it just loops through the generated coarse set of near null space vectors from the **TentativePFactory**. This is a technical detail which sometimes can cause some problems when the corresponding dependency is not defined properly in the XML file. Another example for a special factory is the **NoFactory**. This special factory is used for all data which is kept in memory and needed by the level smoothers during the iteration. Usually, the final transfer operators P and R as well as the level matrix A are transformed to **NoFactory** objects after the setup phase has completed. However, expert users can also use the **NoFactory** mechanism for special data during the setup phase. But this is not recommended.

Exercise 1

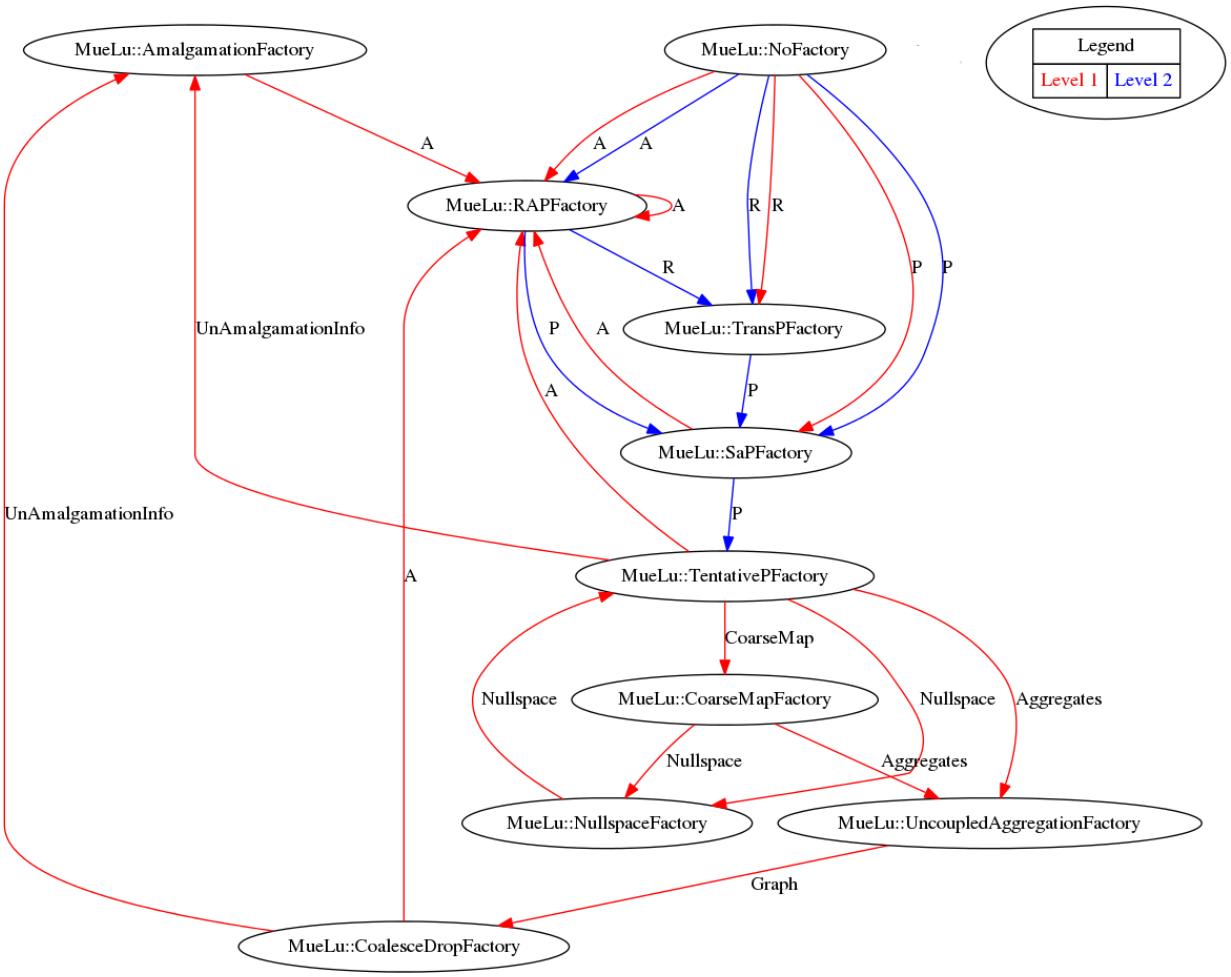


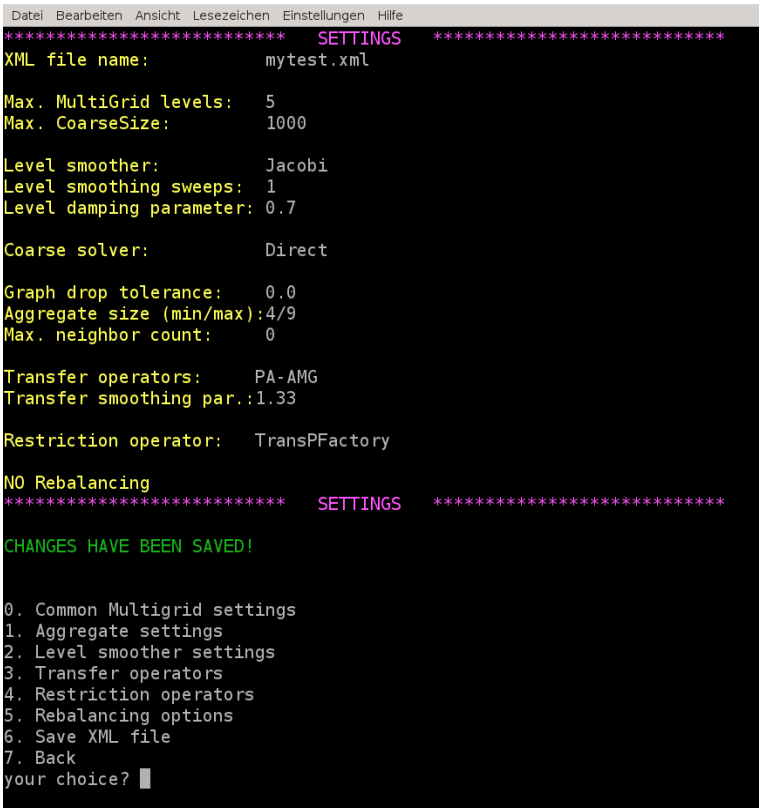
Fig. 2.21: Visualization of aggregates using the CoarseningVisualizationFactory.

Compare the factory layout in Figure aggregation/figure_simpledesignaggregates with above dependency graph. Try to read it from bottom to top. Which factories are missing in Figure aggregation/figure_simpledesignaggregates? Which variables are missing in Figure aggregation/figure_simpledesignaggregates?

2.11.3 Graphical assistant for XML file generation

The **hands-on.py** driver script contains a graphical assistant to generate new XML parameter files in the advanced MueLu file format.

Just run the **hands-on.py** script and choose a problem type from the list. Then choose option 2 to set the xml file. If you enter a filename that does not exist then the assistant is started to generate that new XML file.



Just go through the menu and make your choices for level smoothers, transfer operators and so on. Do not forget to call option 6 to save the XML file under the given name, that you have entered before. Then you can use option 7 to go back to the main menu for the example problem and try your new preconditioner with your parameter choices.

Note: Of course, we could have introduced this feature with the earlier tutorials, but the idea was to familiarize the user with the XML files.

2.12 Challenge: elasticity example

2.12.1 Practical example

For the second challenge, we consider an 2D elasticity example with 7020 degrees of freedom. No further information is provided (geometry, discretization technique, ...).

2.12.2 User-interface

Run the **hands-on.sh** script and choose the option 5 for the elasticity. The script automatically generates a XML file with reference multigrid parameters which are far from being optimal. The resulting problem matrix is symmetric. Therefore, we can use a CG method as outer linear solver.

Exercise 1

Open the **stru2d_parameters.xml** file by pressing option 3. Try to find optimized multigrid settings using your knowledge from the previous tutorials. We have 2 (displacement) degrees of freedom per node and 3 vectors describing the near null space components (rigid body modes). All this information is automatically set correctly by the **hands-on.py** script. Run the example. Check the screen output (using option 1) and verify **blockdim=2** on level 1 and **blockdim=3** on level 2.

In the screen output of the **CoalesceDropFactory** the **blockdim** denotes the number of degrees of freedom per node (or super node on the coarser levels). Since the number of near null space vectors differs from the number of PDE equations, the number of degrees of freedom per node changes on the different multigrid levels.

Exercise 2

Open the XML parameter file (choose option 3) and try to find optimized settings. Use the advanced XML file format. Save the file, rerun the example (option 0) and compare the output with the reference results.

Note: Use *General hints* for a general step-by-step procedure to optimize the multigrid parameters.

Exercise 3

How do the reference settings and your XML parameter settings perform when increasing the number of processors?

Exercise 3

Compare the results of the reference method and your preconditioner parameters when changing to a GMRES solver (instead of CG). What is changing? What about the solver timings?

2.13 Multigrid for Multiphysics

2.13.1 General Concept

Assuming we have a multiphysics application which generates a $n \times n$ block operator (e.g. a Fluid-Structure-Interaction problem, or a Navier-Stokes problem with the splitting into velocity and pressure degrees of freedom), the idea is to preserve the block structure on the coarse levels using block-diagonal segregated transfer operators and block smoothers on all multigrid levels. The blocks $P_{i,i}$ and $R_{i,i}$ in the block-diagonal transfer operators P and R usually are built using the diagonal blocks $A_{i,i}$ or - for applications where $A_{i,i}$ does not contain sufficient information to build aggregates - any other kind of application-specific method.

For the block smoothers we apply well-established block smoothers (e.g. block relaxation methods or Schur complement based methods) as well as application-specific adaptions.

2.13.2 Exemplary setup for a 2×2 problem

Setup of block transfer operators

Warning: Insert missing figure.

Figure ref{fig:transferoperatorsetup} shows a typical layout for the setup of a 2×2 blocked operator block transfer operators. We group factories for building transfer operators for the upper-left and lower-right blocks and build the corresponding blocked transfer operators. In general, we can distinguish volume-coupled and interface-coupled problems. In the simpler case of volume-coupled problems, we usually can use the same aggregates for the second block than we have built for the first block. That is, we reuse the same aggregates (built, e.g., by *myAggFact1*) for the second block. For interface-coupled problems we usually need a separate aggregation strategy for the interface DOFs. This can be a second standard aggregation factory object or an application-specific aggregation routine.

In the following we give more details on the corresponding xml files for setting up the transfer operator layout given in Figure ref{fig:transferoperatorsetup}.

Factory list

All the following definitions of factories are contained in the **Factories** sublist of the **MueLu** parameter list, which basically is meant as a factory collection for all used factories in the multigrid setup. To keep things simple, we only give the factories necessary to define the setup for the most upper-left block in the $n \times n$ block matrix.

```
<!-- sub block factories -->
 <!-- BLOCK 1 (for submatrix A_{00}) -->
 <ParameterList name="mySubBlockAFactory1">
   <Parameter name="factory" type="string" value="SubBlockAFactory"/>
   <Parameter name="block row"                 type="int"     value="0"/>
   <Parameter name="block col"                 type="int"     value="0"/>
   <Parameter name="Range map: Striding info"  type="string"  value="{ 2 }"/>
   <Parameter name="Domain map: Striding info" type="string"  value="{ 2 }"/>
 </ParameterList>

 <ParameterList name="myAggFact1">
   <Parameter name="factory" type="string" value="UncoupledAggregationFactory"/>
   <Parameter name="aggregation: min agg size" type="int" value="5"/>
```

(continues on next page)

(continued from previous page)

```
  <Parameter name="aggregation: max selected neighbors" type="int" value="1"/>
</ParameterList>

<!-- tell the tentative prolongator that we have 2 DOFs per node on the coarse levels --
↪>
<ParameterList name="myCoarseMap1">
  <Parameter name="factory" type="string" value="CoarseMapFactory"/>
  <Parameter name="Striding info" type="string" value="{ 2 }"/>
  <Parameter name="Strided block id" type="int" value="-1"/>
</ParameterList>

<ParameterList name="myTentativePFact1">
  <Parameter name="factory" type="string" value="TentativePFactory"/>
  <Parameter name="A" type="string" value="mySubBlockAFactory1"/>
  <Parameter name="Aggregates" type="string" value="myAggFact1"/>
  <Parameter name="CoarseMap" type="string" value="myCoarseMap1"/>
</ParameterList>

<!-- We have to use Nullspace1 here. If "Nullspace1" is not set the
     Factory creates the default null space containing of constant
     vectors -->
<ParameterList name="myNspFact1">
  <Parameter name="factory" type="string" value="NullspaceFactory"/>
  <Parameter name="Fine level nullspace" type="string" value="Nullspace1"/>
  <Parameter name="Nullspace1" type="string" value="myTentativePFact1"/>
</ParameterList>
```

Note: Please note, that the ordering of the factories is important in the sense that factories have to be defined before they are used as input for other following factories. As an example, the **myCoarseMap1** factory has to be defined before the **myTentativePFact1** as it is used as input for the **CoarseMap** variable. Switching the ordering of the factories in the xml file would result in an error that **myCoarseMap1** is unknown. Technically, one could avoid this restriction by a two pass reading process which first reads all factories and then resolves the dependencies. On the other side, this restriction helps to keep a straightforward linear design of the setup process.

The meaning of the factories is the following:

- **SubBlockAFactory**: Given a $n \times n$ block operator A, the **SubBlockAFactory** extracts the (i, j) block where i is defined by the parameter **block row** and j by the parameter **block col** where $0 \leq i, j < n$. Above example assumes a Thyra-style numbering of the global ids for a simple 2D Navier-Stokes example. That is, the matrix block A_{00} has two degrees of freedom per node (one for the velocity in x-direction and one in the y-direction). The **Range map: Striding info** contains this information (i.e. 2 dofs per node), since this information might get lost (or never was stored) when using Thyra block operators.
- **UncoupledAggregationFactory**: The aggregation factory is used to build the aggregates. In this case, the aggregates shall be built using the graph of A_{00} that is returned by the **SubBlockAFactory**. In this example, we only give the user parameters for the aggregation. Later it is shown how to declare the **FactoryManager** which makes sure that this concrete instance of an aggregation factory builds the aggregates for A_{00} only.
- **CoarseMapFactory**: The **CoarseMapFactory** is used in the **TentativePFactory** and basically is responsible to provide a proper domain map for the transfer operator block P_i. For P_0, this is usually a standard map. The only information that is important is **Striding info** which means that the coarse domain map has 2 dofs per node again. Note: we have 2 dofs per node (for the velocities in x and y-direction). We have 2 null space vectors. Therefore, the coarse problem has also 2 dofs per node which means the domain map of P_i has to be built for 2

dofs per node.

- **TentativePFactory**: Here, the **TentativePFactory** builds the P_i block for the blocked transfer operator P : $math$:. We explicitly give the names of the factories used to generate P_i, which include the previously defined factories for the **CoarseMap**, **Aggregates** and **A**. This information is not really needed in this place as we later define a **FactoryManager** for the i-th block, but often makes it easier to understand the dependencies. That is, the short version would just be

```
<ParameterList name="myTentativePFact1">
  <Parameter name="factory" type="string" value="TentativePFactory"/>
</ParameterList>
```

- **NullspaceFactory**: For defining multigrid methods for multiphysics problems, the **NullspaceFactory** is very important. In general, the **TentativePFactory** uses the provided fine level near null space vectors to generate the tentative prolongation operator P_i together with a coarse representation of the near null space vectors. That is, the **TentativePFactory** produces the **Nullspace** information that it needs itself on the next coarser level. That is a hidden dependency which usually automatically works without any changes necessary by the user. The user is only responsible to provide proper fine level near null space vectors as **Nullspace** variable on the finest level. The **NullspaceFactory** is just a helper factory which processes the null space information on the finest level and pipes it in into the global setup process. For multiphysics problems, the user has to provide n partial near null space vectors (one for each mathematical or physical field) using the variable names **Nullspace1** to **Nullspace9** on the finest level. The **Fine level nullspace** parameter in the **NullspaceFactory** then can be set to the corresponding variable name (e.g. texttt{Nullspace1}). That is, the **NullspaceFactory** checks the fine level variable container for a variable named **Nullspace1** and uses the content as fine level null space for input in the **TentativePFactory**. It is important, that besides of the **Fine level nullspace** parameter another parameter with the name of the near null space vector (in above case **Nullspace1**) is declared with the corresponding **TentativePFactory** name as value. This closes the circle for the null space generation for block P_i on all coarser levels. It is important that the **NullspaceFactory** is defined after the corresponding **TentativePFactory** class, such that the dependency circle can be closed correctly.

Note: Instead of **TentativePFactory** any other factory which generates a coarse null space can be used as well (e.g. **SemiCoarsenPFactory**).

Note: Of course, above factory list can be extended for smoothed aggregation. We also skipped the factories for the restriction operators.

Factory manager

Once the necessary factories for building P_i are defined in the **FactoryList** section of the xml file, we can group them together. Right after the factories, we can add a **FactoryManager** block in the **FactoryList** section.

```
<!-- Multigrid setup for velocity block (A_{00}) -->
<ParameterList name="myFirstGroup">
  <Parameter name="group" type="string" value="FactoryManager"/>
  <Parameter name="A" type="string" value="mySubBlockAFactory1"/>
  <Parameter name="P" type="string" value="myTentativePFact1"/>
  <Parameter name="Aggregates" type="string" value="myAggFact1"/>
  <Parameter name="Nullspace" type="string" value="myNspFact1"/>
  <Parameter name="CoarseMap" type="string" value="myCoarseMap1"/>
</ParameterList>
```

The name for the group can be chosen freely (e.g. **myFirstGroup**). Besides the declaration of the **FactoryManager** group using the

```
<Parameter name="group" type="string" value="FactoryManager"/>
```

parameter, it contains a list of all factories which are used in context of building the coarsening information for the corresponding block.

The group block defining a **FactoryManager** has a similar role than in the **Hierarchy** section in the xml file later. It allows to group together factories into subgroups that can be referred to by the common name (e.g. **myFirstGroup**) later. These groups help to organize the different factories. Note, that we basically need one group for each physical/mathematical field in our $n \times n$ block operator, that is we need n groups.

Note: The group block for the second row in the block operator could look like the following

```
<!-- Multigrid setup for pressure block (A_{11}) -->
<ParameterList name="mySecondGroup">
  <Parameter name="group" type="string" value="FactoryManager"/>
  <Parameter name="A" type="string" value="mySubBlockAFactory2"/>
  <Parameter name="P" type="string" value="myTentativePFact2"/>
  <!-- reuse aggs from velocity block! -->
  <Parameter name="Aggregates" type="string" value="myAggFact1"/>
  <Parameter name="Nullspace" type="string" value="myNspFact2"/>
  <Parameter name="CoarseMap" type="string" value="myCoarseMap2"/>
</ParameterList>
```

This assumes that all the factories have been defined before similar to the first group blocks. Note, that in some cases for certain applications, it is possible to reuse information from the first block in the second block. In this case, we use the abstract aggregation information that has been built using the velocity information for the associated pressure degrees of freedom. This is possible, since in our example each node has 2 velocity and 1 pressure degree of freedom.

Block prolongation operator

The diagonal block prolongation operator is built using

```
<!-- define block prolongation operator using above blocks -->
<ParameterList name="myBlockedPFact">
  <Parameter name="factory" type="string" value="BlockedPFactory"/>
  <!-- factory manager for block 1 -->
  <ParameterList name="block1">
    <Parameter name="group" type="string" value="myFirstGroup"/>
  </ParameterList>
  <!-- factory manager for block 2 -->
  <ParameterList name="block2">
    <Parameter name="group" type="string" value="mySecondGroup"/>
  </ParameterList>
</ParameterList>
```

in the **FactoryList** section of the xml file after all groups have been defined. It contains basically sublists with the names **block1** to **blockn**. Each of these sublists contains a parameter **group** with the group name defined before.

Note: Instead of using groups, you could also put all the factory definitions within the corresponding **blockn** parameter list. But this would mean that you have to set all inter-factory dependencies in the corresponding block by hand. You

cannot use the general defaults that are defined in the groups. It would also make it impossible to reuse information from factories belonging to a different block (e.g., you could not reuse the aggregation information built by the **myAggFact1** for the aggregates in block 2.

Block restriction operator

The following definitions should be the standard for nearly all multiphysics problems. We use the **GenericRFactory** for building the restriction operator out of the blocked prolongation factory.

```
<!-- define block restriction operator using above blocks -->
<!-- The block restriction operator is usually always of type
     GenericRFactory since we want to be able to combine, e.g.,
     SmoothedAggregation for block A_{00} with e.g. tentative
     prolongation for block A_{11} (or any other kind of transfer
     strategy for the subblocks -->
<ParameterList name="myBlockedRFact">
  <Parameter name="factory" type="string" value="GenericRFactory"/>
  <Parameter name="P" type="string" value="myBlockedPFact"/>
</ParameterList>
```

It uses the blocks P_i to generate the corresponding blocks R_i for the diagonal of the restriction operator. If PG-AMG is used for some or all blocks P_i this is automatically considered when generating R_i.

Note: Please note, that you cannot use **TransPFactory** as it has no support for block operators. However, this is not a problem, since the **TransPFactory** might be used locally for single blocks wherever possible.

Coarse level operator

Once, the block diagonal transfer operators P and R are set up, the **BlockedRAPFactory** builds the coarse $n \times n$ operator:

```
<ParameterList name="myBlockedRAPFact">
  <Parameter name="factory" type="string" value="BlockedRAPFactory"/>
  <Parameter name="P" type="string" value="myBlockedPFact"/>
  <Parameter name="R" type="string" value="myBlockedRFact"/>
</ParameterList>
```

2.13.3 Setup of block smoothers

Once the transfer operators are set up properly, it is time to define the block smoothing methods

Xpetra block smoothers

The Xpetra package contains a set of general block smoothers, including a block Gauss-Seidel method for $n \times n$ block operators and a Schur complement based SIMPLE variant for 2×2 block operators. Here we just explain the setup for the Schur complement smoother as an example.

The Schur complement based smoother internally needs two solvers/smoothers. The first smoother/solver is need for calculating a prediction for the velocities, the second solver/smoother then has to (approximately) solve the Schur complement equation.

We are still in the **FactoryList** section of the xml file and define the corresponding solver/smoother blocks:

```
<!-- block smoother for block A_{00} -->
<ParameterList name="mySmooFact1">
  <Parameter name="factory" type="string" value="TrilinosSmoother"/>
  <Parameter name="type" type="string" value="RELAXATION"/>
  <ParameterList name="ParameterList">
    <Parameter name="relaxation: type" type="string" value="Symmetric Gauss-Seidel"/>
    <Parameter name="relaxation: sweeps" type="int"    value="3"/>
    <Parameter name="relaxation: damping factor" type="double" value="0.8"/>
  </ParameterList>
  <Parameter name="A" type="string" value="mySubBlockAFactory1"/>
</ParameterList>

<!-- Build Schur Complement factory (for being used in SIMPLE type block smoother)-->
<ParameterList name="mySchurCompFact">
  <Parameter name="factory" type="string" value="SchurComplementFactory"/>
  <Parameter name="omega" type="double" value="0.8"/>
  <Parameter name="lumping" type="bool" value="false"/>
</ParameterList>

<!-- block smoother for block A_{11} respective the Schur complement operator -->
<ParameterList name="mySchurSmooFact">
  <Parameter name="factory" type="string" value="TrilinosSmoother"/>
  <Parameter name="type" type="string" value="RELAXATION"/>
  <ParameterList name="ParameterList">
    <Parameter name="relaxation: type" type="string" value="Symmetric Gauss-Seidel"/>
    <Parameter name="relaxation: sweeps" type="int"    value="1"/>
    <Parameter name="relaxation: damping factor" type="double" value="0.7"/>
  </ParameterList>
  <!-- You don't have to specify the input matrix A for the smoother -->
  <!-- It is clear from the subblocks in the BlockedSmoother below -->
  <!--<Parameter name="A" type="string" value="mySchurCompFact"/>-->
</ParameterList>
```

It is not necessary but helpful to declare variable **A** in **mySmooFact1** to be the diagonal block A_{00} of the blocked operator. This way it is obvious that this smoother is supposed to generate the prediction within the Schur complement approach. The second smoother (with the name **mySchurSmooFact** is supposed to solve the Schur complement equation, that is, the input matrix A for this smoother should be the Schur complement operator $A_{11} - A_{10}A_{00}^{-1}A_{01}$ or at least a good approximation of the Schur complement operator. This operator is provided by the **SchurComplementFactory**. Be aware, that the **SchurComplementFactory** uses the full 2×2 operator A as input to generate the approximation of the Schur complement operator. It is not defined as input variable **A** since the full 2×2 operator is the standard answer for variable **A**. It would make sense, though, to declare **mySchurCompFact** as variable **A** for **mySchurSmooFact**.

The Schur complement smoother then is defined by the block:

```
<!-- Use SIMPLE: -->
<!-- User has to define two blocks with each containing a smoother for
     the corresponding sub-block matrix (see above) -->
<ParameterList name="myBlockSmoother">
  <Parameter name="factory" type="string" value="SimpleSmoother"/>
  <Parameter name="Sweeps" type="int" value="1"/>
  <Parameter name="Damping factor" type="double" value="1.0"/>
  <Parameter name="UseSIMPLEC" type="bool" value="false"/>
  <!-- factory manager for block 1 -->
  <ParameterList name="block1">
    <!-- <Parameter name="group" type="string" value="myFirstGroup"/> -->
    <!-- It's enough to just provide the sub block matrix A_{00} needed
         as input for the smoother "mySmooFact1" as well as the smoother
         itself. Alternatively, one could add below strings to the
         factory group "myFirstGroup" and use it here instead of below lines -->
    <Parameter name="A" type="string" value="mySubBlockAFactory1"/>
    <Parameter name="Smoother" type="string" value="mySmooFact1"/>
  </ParameterList>
  <!-- factory manager for block 2 -->
  <ParameterList name="block2">
    <!-- The second equation in the SIMPLE smoother is the Schur complement
         equation that has to be solved. Therefore, provide the Schur complement
         operator together with the smoother. The smoother object takes the
         Schur complement operator as operator "A" for the internal smoothing process. --
↪>
    <Parameter name="A" type="string" value="mySchurCompFact"/>
    <Parameter name="Smoother" type="string" value="mySchurSmooFact"/>
  </ParameterList>
</ParameterList>
```

In this case, we use the **SimpleSmoother** as example. Besides the typical smoother parameters (number of sweeps, damping, ldots), the interesting part are the sublists **block1** and **block2**, which contain the information about the internal smoothers/solvers. In above example, we just declare the factories for **A** and **Smoother**. The variable **A** always gives the internal linear operator that is used within the solver/smoother. By defining **A** in this place, we do not really have to define it extra in the smoother blocks above.

Note: Please note, that instead of the explicit variable definitions in the **blockn** sublists, one could also have given just the group names. However, this only works if the **Smoother** variable is also contained in the corresponding groups. In above examples from the previous section we skipped the **Smoother** variable. This makes sense especially if the aggregation information is built using a different A operator as the smoother is using. In above example we do not build aggregates using the Schur complement operator, but want to reuse aggregates from the first block.

As a side note it shall be mentioned, that you can also directly make all definitions in the **blockn** parameter lists:

```
<ParameterList name="block1">
  <!-- <Parameter name="group" type="string" value="myFirstGroup"/> -->
  <ParameterList name="Smoother">
    <Parameter name="factory" type="string" value="TrilinosSmoother"/>
    <Parameter name="type" type="string" value="RELAXATION"/>
    <ParameterList name="ParameterList">
      <Parameter name="relaxation: type" type="string" value="Symmetric Gauss-Seidel"/>
      <Parameter name="relaxation: sweeps" type="int"    value="3"/>
```

(continues on next page)

(continued from previous page)

```
      <Parameter name="relaxation: damping factor" type="double" value="0.8"/>
    </ParameterList>
  </ParameterList>
  <!-- Note: this is not recommended as it creates a second instance of the sub block␣
↪AFactory -->
  <ParameterList name="A">
    <Parameter name="factory" type="string" value="SubBlockAFactory"/>
    <Parameter name="block row"                 type="int"     value="0"/>
    <Parameter name="block col"                 type="int"     value="0"/>
    <Parameter name="Range map: Striding info"  type="string"  value="{ 2 }"/>
    <Parameter name="Domain map: Striding info" type="string"  value="{ 2 }"/>
  </ParameterList>
</ParameterList>
```

However, this is not really recommended since it prevents the reuse of factories in several places. E.g., instead of the new **SubBlockAFactory** one should just reuse the **SubBlockAFactory**, which has been defined and used before for the block transfers. This drastically simplifies and shortens the factory definitions and reduces the number of potential errors.

Warning: Be aware, that each new block in the xml file means that a new instance of the corresponding factory is instantiated and built. In the worst case some expensive information is calculated twice, which might heavily impact the overall performance.

Teko block smoothers

Note: In Trilinos, the Teko package provides block preconditioners, that can be used as alternative to the existing Xpetra block smoothers. The Xpetra linear algebra layer also provides support for Thyra block operators, which allows us to use the Teko block smoothers within a MueLu multigrid hierarchy. In case of the Teko SIMPLE implementation, one again needs to internal solvers/smoothers (one for the prediction of the primary variables and one for the solution of the Schur complement equation). Teko uses the Stratimikos interface for defining the corresponding smoothers/solvers. So, instead of the **SimpleSmoother** object from the previous subsection, one can also use the SIMPLE implementation from Teko.

We define a **TekoSmoother** as block smoother using

```
<!-- Use SIMPLE: -->
<!-- User has to define two blocks with each containing a smoother for
     the corresponding sub-block matrix-->
<ParameterList name="myTekoSmoother">
  <Parameter name="factory" type="string" value="TekoSmoother"/>
  <Parameter name="Inverse Type" type="string" value="SIMPLE"/> <!-- contains name of␣
↪sublist within Teko parameters -->
  <ParameterList name="Inverse Factory Library">
    <ParameterList name="SIMPLE">
      <Parameter name="Type" type="string" value="NS SIMPLE"/>
      <Parameter name="Inverse Velocity Type" type="string" value="Amesos"/>
      <Parameter name="Inverse Pressure Type" type="string" value="Amesos"/>
    </ParameterList>
```

(continues on next page)

(continued from previous page)

```
  </ParameterList>
</ParameterList>
```

The **TekoSmoother** accepts the full 2×2 block operator as input (not declared above, since it is the default) and contains a sublist with the name **Inverse Factory Library**. Within this sublist, all local smoothers/solvers as well as the Teko block smoother (or several Teko block smoothers) are defined. In the above example, there is only one Teko block smoother (of type **NS SIMPLE**) declared, which internally uses direct solvers from the Amesos package for the velocity and pressure (Schur complement) problem. The **Inverse Type** parameter of the **TekoSmoother** defines the Teko block smoother from the **Inverse Factory Library**. For the available parameters and block smoothers in Teko, the reader is referred to the Teko documentation.

2.13.4 Multigrid setup

Last but not least, once both the transfers and the block smoothers are defined, the multigrid method itself has to be set up. Note, that all previous definitions and declarations have been made in the **Factories** section of the xml file. The multigrid setup is now done in the **Hierarchy** section of the xml file and looks like:

```
<ParameterList name="Hierarchy">

  <Parameter name="max levels"          type="int"      value="3"/>
  <Parameter name="coarse: max size"    type="int"      value="10"/>
  <Parameter name="verbosity"           type="string"   value="High"/>

  <ParameterList name="AllLevel">
    <Parameter name="startLevel"        type="int"      value="0"/>
    <Parameter name="Smoother"          type="string"   value="myTekoSmoother"/>
    <Parameter name="CoarseSolver"      type="string"   value="myTekoSmoother"/>
    <Parameter name="P"                 type="string"   value="myBlockedPFact"/>
    <Parameter name="R"                 type="string"   value="myBlockedRFact"/>
    <Parameter name="A"                 type="string"   value="myBlockedRAPFact"/>
  </ParameterList>
</ParameterList>
```

The interesting part is the **AllLevel** sublist (you can freely choose the name of this list), which - in some sense - corresponds to the groups introduced before to setup the block transfers and block smoothers. In fact, this sublist defines the master **FactoryManager** for the overall multigrid method. Note, that all variables (**A**, **P**, **R**, **...**) are generated by the block versions instead of the single block factories.

2.13.5 Exemplary setup for a 2×2 problem with rebalancing

Transfer operator setup

Warning: Include missing figure.

Figure ref{fig:transferoperatorsetuprebalancing} shows the basic setup for block transfer operators with rebalancing enabled. Please compare it with the complete XML input deck in Section ref{sec:xmlinputdeckrebalancing}.

As one can see from the upper part of Figure ref{fig:transferoperatorsetuprebalancing}, first we build blocked transfer operators and a blocked coarse level operator using sub-factory manager objects **myFirstGroup** and **mySecondGroup**

in the factories **myBlockedPFact**, **myBlockedRFact** and **myBlockedRAPFact**. Then, we rebalance the coarse level blocked operator A from **myBlockedRAPFact**.

The **myRepartitionHeuristicFact** object will decide whether rebalancing is necessary. If yes, then it will return the number of required partitions for the coarse level operator. This input is processed by the repartition interface and repartition factory objects that finally create **Xpetra::Importer** to do the rebalancing. The **myRebBlocked{P,R,Ac}Fact** objects use those **Importer** objects to perform the rebalancing.

Please note, that we build additional helper factory manager objects **myRebFirstGroup** and **myRebSecondGroup** which contain all factories relevant for rebalancing the two blocks.

Note: No changes are necessary when setting up the block smoothers, as they use the matrices on the current level as input (which may or may not be rebalanced in the previous transfer operator setup process).

Complete XML input deck

Listing 2.11: ../../../test/tutorial/blocked_rebalancing.xml

```
<ParameterList name="MueLu">

  <!-- Factory collection -->
  <ParameterList name="Factories">

    <!-- Note that ParameterLists must be defined prior to being used -->

    <!-- sub block factories -->


    <!-- BLOCK 1 (for submatrix A_{00}) VELOCITY PART -->
    <ParameterList name="mySubBlockAFactory1">
      <Parameter name="factory" type="string" value="SubBlockAFactory"/>
      <Parameter name="block row"                 type="int"     value="0"/>
      <Parameter name="block col"                 type="int"     value="0"/>
      <Parameter name="Range map: Striding info"  type="string"  value="{ 4 }"/>
      <Parameter name="Domain map: Striding info" type="string"  value="{ 4 }"/>
    </ParameterList>

    <!--<ParameterList name="myAggFact1">
      <Parameter name="factory" type="string" value="UncoupledAggregationFactory"/>
      <Parameter name="aggregation: min agg size" type="int" value="5"/>
      <Parameter name="aggregation: max selected neighbors" type="int" value="1"/>
    </ParameterList>-->

    <!-- tell the tentative prolongator that we have 2 DOFs per node on the coarse␣
↪levels -->
    <ParameterList name="myCoarseMap1">
      <Parameter name="factory" type="string" value="CoarseMapFactory"/>
      <Parameter name="Striding info" type="string" value="{ 4 }"/>
      <Parameter name="Strided block id" type="int" value="-1"/>
    </ParameterList>

    <ParameterList name="myAggFact1">
```

(continues on next page)

(continued from previous page)

```
    <Parameter name="factory" type="string" value="UncoupledAggregationFactory"/>
    <Parameter name="aggregation: min agg size" type="int" value="5"/>
    <Parameter name="aggregation: max selected neighbors" type="int" value="1"/>
  </ParameterList>

  <ParameterList name="myTentativePFact1">
    <Parameter name="factory" type="string" value="TentativePFactory"/>
    <Parameter name="A" type="string" value="mySubBlockAFactory1"/>
    <Parameter name="Aggregates" type="string" value="myAggFact1"/>
    <Parameter name="CoarseMap" type="string" value="myCoarseMap1"/>
  </ParameterList>

  <ParameterList name="myRFact1">
    <Parameter name="factory" type="string" value="TransPFactory"/>
    <Parameter name="P" type="string" value="myTentativePFact1"/>
  </ParameterList>

  <!-- We have to use Nullspace1 here. If "Nullspace1" is not set the
       Factory creates the default null space containing of constant
       vectors -->
  <ParameterList name="myNspFact1">
    <Parameter name="factory" type="string" value="NullspaceFactory"/>
    <Parameter name="Fine level nullspace" type="string" value="Nullspace1"/>
    <Parameter name="Nullspace1" type="string" value="myTentativePFact1"/>
  </ParameterList>

  <!-- BLOCK 2 (for submatrix A_{22})-->
  <ParameterList name="mySubBlockAFactory">
    <Parameter name="factory" type="string" value="SubBlockAFactory"/>
    <Parameter name="block row"                 type="int"     value="1"/>
    <Parameter name="block col"                 type="int"     value="1"/>
    <Parameter name="Range map: Striding info"  type="string"  value="{ 4 }"/>
    <Parameter name="Domain map: Striding info" type="string"  value="{ 4 }"/>
  </ParameterList>

  <!-- tell the tentative prolongator that we have 1 DOF per node on the coarse levels␣
↪-->
  <!-- We use the factory "CoarseMapFactory" which always creates a standard coarse
       level map starting with GIDs at 0. This is ok as we use Thyra like numbering
       for the block operators. To obtain unique GIDs one would use the
       BlockedCoarseMapFactory (see below) -->
  <ParameterList name="myCoarseMap2">
    <Parameter name="factory" type="string" value="CoarseMapFactory"/>
    <Parameter name="Striding info" type="string" value="{ 4 }"/>
    <Parameter name="Strided block id" type="int" value="-1"/>
  </ParameterList>

  <ParameterList name="myTentativePFact2">
    <Parameter name="factory" type="string" value="TentativePFactory"/>
    <Parameter name="A" type="string" value="mySubBlockAFactory2"/>
    <Parameter name="Aggregates" type="string" value="myAggFact1"/> <!-- use␣
↪aggregates from velocity block! -->
```

(continues on next page)

(continued from previous page)

```
    <Parameter name="CoarseMap" type="string" value="myCoarseMap2"/>
  </ParameterList>

  <!-- We have to use Nullspace2 here. If "Nullspace2" is not set the
       Factory creates the default null space containing of constant
       vectors (here only one constant vector) -->
  <ParameterList name="myNspFact2">
    <Parameter name="factory" type="string" value="NullspaceFactory"/>
    <Parameter name="Fine level nullspace" type="string" value="Nullspace2"/>
    <Parameter name="Nullspace2" type="string" value="myTentativePFact2"/>
  </ParameterList>

  <!-- FACTORY MANAGERS -->

  <!-- Multigrid setup for velocity block (A_{00}) -->
  <ParameterList name="myFirstGroup">
    <Parameter name="group" type="string" value="FactoryManager"/>
    <Parameter name="A" type="string" value="mySubBlockAFactory1"/>
    <Parameter name="P" type="string" value="myTentativePFact1"/>
    <Parameter name="R" type="string" value="myRFact1"/>
    <Parameter name="Aggregates" type="string" value="myAggFact1"/>
    <Parameter name="Nullspace" type="string" value="myNspFact1"/>
    <Parameter name="CoarseMap" type="string" value="myCoarseMap1"/>
  </ParameterList>

  <!-- Multigrid setup for pressure block (A_{22}) -->
  <ParameterList name="mySecondGroup">
    <Parameter name="group" type="string" value="FactoryManager"/>
    <Parameter name="A" type="string" value="mySubBlockAFactory2"/>
    <Parameter name="P" type="string" value="myTentativePFact2"/>
    <Parameter name="Aggregates" type="string" value="myAggFact1"/><!-- reuse aggs -->
    <Parameter name="Nullspace" type="string" value="myNspFact2"/>
    <Parameter name="CoarseMap" type="string" value="myCoarseMap2"/>
  </ParameterList>

  <!-- BLOCK TRANSFER operators -->

  <!-- define block prolongation operator using above blocks -->
  <ParameterList name="myBlockedPFact">
    <Parameter name="factory" type="string" value="BlockedPFactory"/>
    <!-- factory manager for block 1 -->
    <ParameterList name="block1">
       <Parameter name="group" type="string" value="myFirstGroup"/>
    </ParameterList>
    <!-- factory manager for block 2 -->
    <ParameterList name="block2">
      <Parameter name="group" type="string" value="mySecondGroup"/>
    </ParameterList>
  </ParameterList>

  <!-- define block restriction operator using above blocks -->
  <!-- The block restriction operator is usually always of type
```

(continues on next page)

(continued from previous page)

```
        GenericRFactory since we want to be able to combine, e.g.,
        SmoothedAggregation for block A_{00} with e.g. tentative
        prolongation for block A_{11} (or any other kind of transfer
        strategy for the subblocks -->
    <ParameterList name="myBlockedRFact">
      <Parameter name="factory" type="string" value="GenericRFactory"/>
      <Parameter name="P" type="string" value="myBlockedPFact"/>
    </ParameterList>

    <ParameterList name="myTransferCoordinatesFact">
      <Parameter name="factory" type="string" value="CoordinatesTransferFactory"/>
      <!-- we need the factories from the first group -->
      <Parameter name="Aggregates" type="string" value="myAggFact1"/>
      <Parameter name="CoarseMap" type="string" value="myCoarseMap1"/>

    </ParameterList>

    <ParameterList name="myBlockedRAPFact">
      <Parameter name="factory" type="string" value="BlockedRAPFactory"/>
      <Parameter name="P" type="string" value="myBlockedPFact"/>
      <Parameter name="R" type="string" value="myBlockedRFact"/>
      <ParameterList name="TransferFactories">
        <Parameter name="For Coordinates" type="string" value="myTransferCoordinatesFact
↪"/>
      </ParameterList>
    </ParameterList>

    <!-- REBALANCING -->

    <ParameterList name="myRepartitionHeuristicFactory">
      <Parameter name="factory" type="string" value="RepartitionHeuristicFactory"/>
      <Parameter name="A"                              type="string"  value=
↪"myBlockedRAPFact"/>
      <Parameter name="repartition: start level"       type="int"     value="2"/>
      <Parameter name="repartition: min rows per proc" type="int"     value="256"/>
      <Parameter name="repartition: nonzeroImbalance"  type="double"  value="1.2"/>
    </ParameterList>

    <ParameterList name="myRebSubBlockAFactory1">
      <Parameter name="factory" type="string" value="SubBlockAFactory"/>
      <Parameter name="A"                         type="string"  value="myBlockedRAPFact
↪"/>
      <Parameter name="block row"                 type="int"     value="0"/>
      <Parameter name="block col"                 type="int"     value="0"/>
      <Parameter name="Range map: Striding info"  type="string"  value="{ 4 }"/>
      <Parameter name="Domain map: Striding info" type="string"  value="{ 4 }"/>
    </ParameterList>

    <ParameterList name="myInputCoordsFact">
      <Parameter name="factory" type="string" value="FineLevelInputDataFactory"/>
      <Parameter name="Variable" type="string" value="Coordinates"/>
      <Parameter name="Variable type" type="string" value="MultiVector"/>
```

(continues on next page)

(continued from previous page)

```
      <Parameter name="Fine level factory" type="string" value="NoFactory"/>
      <!--<Parameter name="Coarse level factory" type="string" value=
↪"myTransferCoordinatesFact"/>-->
      <!--<Parameter name="Coarse level factory" type="string" value="NoFactory"/> TO BE␣
↪DEFINED LATER -->
    </ParameterList>

    <ParameterList name="myZoltanInterface1">
      <Parameter name="factory"                              type="string" value=
↪"ZoltanInterface"/>
      <Parameter name="A"                                    type="string" value=
↪"myRebSubBlockAFactory1"/>
      <Parameter name="number of partitions"                 type="string" value=
↪"myRepartitionHeuristicFactory"/>
      <Parameter name="Coordinates"                          type="string" value=
↪"myInputCoordsFact"/>
    </ParameterList>

    <ParameterList name="myRepartitionFactory1">
      <Parameter name="factory" type="string" value="RepartitionFactory"/>
      <Parameter name="A"                          type="string"  value=
↪"myRebSubBlockAFactory1"/>
      <Parameter name="Partition"                  type="string"  value=
↪"myZoltanInterface1"/>
      <Parameter name="number of partitions"       type="string"  value=
↪"myRepartitionHeuristicFactory"/>
      <Parameter name="repartition: print partition distribution" type="bool" value="true
↪"/>
      <Parameter name="repartition: remap parts" type="bool" value="true"/>
    </ParameterList>

    <ParameterList name="myRebSubBlockAFactory2">
      <Parameter name="factory" type="string" value="SubBlockAFactory"/>
      <Parameter name="A"                          type="string"  value="myBlockedRAPFact
↪"/>
      <Parameter name="block row"                  type="int"     value="1"/>
      <Parameter name="block col"                  type="int"     value="1"/>
      <Parameter name="Range map: Striding info"   type="string"  value="{ 4 }"/>
      <Parameter name="Domain map: Striding info"  type="string"  value="{ 4 }"/>
    </ParameterList>

    <ParameterList name="myRepartitionInterface2">
      <Parameter name="factory" type="string" value="CloneRepartitionInterface"/>
      <Parameter name="A"                          type="string"  value=
↪"myRebSubBlockAFactory2"/>
      <Parameter name="number of partitions"       type="string"  value=
↪"myRepartitionHeuristicFactory"/>
      <Parameter name="Partition"                  type="string"  value=
↪"myZoltanInterface1"/>
    </ParameterList>

    <ParameterList name="myRepartitionFactory2">
```

(continues on next page)

(continued from previous page)

```
      <Parameter name="factory" type="string" value="RepartitionFactory"/>
      <Parameter name="A"                         type="string"  value=
↪"myRebSubBlockAFactory2"/>
      <Parameter name="Partition"                 type="string"  value=
↪"myRepartitionInterface2"/>
      <Parameter name="number of partitions"      type="string"  value=
↪"myRepartitionHeuristicFactory"/>
      <Parameter name="repartition: print partition distribution" type="bool" value="true
↪"/>
      <Parameter name="repartition: remap parts" type="bool" value="false"/>
    </ParameterList>

    <!-- Rebalanced groups for velocity block (A_{00}) -->
    <ParameterList name="myRebFirstGroup">
      <Parameter name="group" type="string" value="FactoryManager"/>
      <Parameter name="A" type="string" value="mySubBlockAFactory1"/> <!-- non-
↪rebalanced A11! -->
      <Parameter name="Importer" type="string" value="myRepartitionFactory1"/>
      <Parameter name="Nullspace" type="string" value="myNspFact1"/>
      <Parameter name="number of partitions"      type="string"  value=
↪"myRepartitionHeuristicFactory"/>
      <Parameter name="Coordinates" type="string" value="myInputCoordsFact"/>
    </ParameterList>

    <!-- Rebalanced groups for magnetics block (A_{22}) -->
    <ParameterList name="myRebSecondGroup">
      <Parameter name="group" type="string" value="FactoryManager"/>
      <Parameter name="A" type="string" value="mySubBlockAFactory2"/> <!-- non-
↪rebalanced ! -->
      <Parameter name="Importer" type="string" value="myRepartitionFactory2"/>
      <Parameter name="number of partitions"      type="string"  value=
↪"myRepartitionHeuristicFactory"/>
      <Parameter name="Nullspace" type="string" value="myNspFact2"/>
      <Parameter name="Coordinates" type="string" value="myInputCoordsFact"/>
    </ParameterList>

    <ParameterList name="myRebBlockedPFact">
      <Parameter name="factory" type="string" value="RebalanceBlockInterpolationFactory"/
↪>
      <Parameter name="A" type="string" value="myBlockedRAPFact"/>
      <Parameter name="P" type="string" value="myBlockedPFact"/>
      <ParameterList name="block1">
         <Parameter name="group" type="string" value="myRebFirstGroup"/>
      </ParameterList>
      <ParameterList name="block2">
        <Parameter name="group" type="string" value="myRebSecondGroup"/>
      </ParameterList>
    </ParameterList>

    <ParameterList name="myInputCoordsFactDeps">
      <Parameter name="dependency for" type="string" value="myInputCoordsFact"/>
      <Parameter name="Coarse level factory" type="string" value=
```

(continues on next page)

(continued from previous page)

```
↪"myTransferCoordinatesFact"/>
    </ParameterList>

    <ParameterList name="myRebBlockedRFact">
      <Parameter name="factory" type="string" value="RebalanceBlockRestrictionFactory"/>
      <Parameter name="R" type="string" value="myBlockedRFact"/>
      <Parameter name="repartition: use subcommunicators" type="bool" value="false"/>
      <ParameterList name="block1">
         <Parameter name="group" type="string" value="myRebFirstGroup"/>
      </ParameterList>
      <ParameterList name="block2">
        <Parameter name="group" type="string" value="myRebSecondGroup"/>
      </ParameterList>
    </ParameterList>

    <ParameterList name="myRebBlockedAcFact">
      <Parameter name="factory" type="string" value="RebalanceBlockAcFactory"/>
      <Parameter name="A" type="string" value="myBlockedRAPFact"/>
      <Parameter name="repartition: use subcommunicators" type="bool" value="false"/>
      <ParameterList name="block1">
         <Parameter name="group" type="string" value="myRebFirstGroup"/>
      </ParameterList>
      <ParameterList name="block2">
        <Parameter name="group" type="string" value="myRebSecondGroup"/>
      </ParameterList>
    </ParameterList>

    <!-- BLOCK SMOOTHERS -->

    <ParameterList name="mySmooFact1">
      <Parameter name="factory" type="string" value="TrilinosSmoother"/>
      <Parameter name="type" type="string" value="RELAXATION"/>
      <Parameter name="overlap" type="int" value="0"/>
      <ParameterList name="ParameterList">
        <Parameter name="relaxation: type" type="string" value="symmetric Gauss-Seidel"/>
        <Parameter name="relaxation: sweeps" type="int"    value="1"/>
        <Parameter name="relaxation: damping factor" type="double" value="0.9"/>
      </ParameterList>
    </ParameterList>


    <!-- block smoother for block A_{00} -->

    <ParameterList name="mySmooFact2">
      <Parameter name="factory" type="string" value="TrilinosSmoother"/>
      <Parameter name="type" type="string" value="RELAXATION"/>
      <Parameter name="overlap" type="int" value="0"/>
      <ParameterList name="ParameterList">
        <Parameter name="relaxation: type" type="string" value="symmetric Gauss-Seidel"/>
        <Parameter name="relaxation: sweeps" type="int"    value="1"/>
        <Parameter name="relaxation: damping factor" type="double" value="0.9"/>
      </ParameterList>
```

(continues on next page)

(continued from previous page)

```
    </ParameterList>

    <!-- Use BGS -->
    <ParameterList name="myBlockSmoother">
      <Parameter name="factory" type="string" value="BlockedGaussSeidelSmoother"/>
      <Parameter name="Sweeps" type="int" value="1"/>
      <Parameter name="Damping factor" type="double" value="0.4"/>
      <!-- factory manager for block 1 -->
      <ParameterList name="block1">
        <Parameter name="A" type="string" value="mySubBlockAFactory1"/>
        <Parameter name="Smoother" type="string" value="mySmooFact1"/>
      </ParameterList>
      <!-- factory manager for block 3 -->
      <ParameterList name="block2">
        <Parameter name="A" type="string" value="mySubBlockAFactory2"/>
        <Parameter name="Smoother" type="string" value="mySmooFact2"/>
      </ParameterList>
    </ParameterList>

    <ParameterList name="myBlockDirectSolver">
      <Parameter name="factory" type="string" value="BlockedDirectSolver"/>
    </ParameterList>

  </ParameterList>
  <!-- end Factories -->

  <!-- Definition of the multigrid preconditioner -->
  <ParameterList name="Hierarchy">

    <Parameter name="max levels"          type="int"      value="10"/>
    <Parameter name="coarse: max size"    type="int"      value="2500"/>
    <Parameter name="verbosity"           type="string"   value="High"/>

    <ParameterList name="AllLevel">
      <Parameter name="startLevel"        type="int"      value="0"/>
      <Parameter name="Smoother"          type="string"   value="myBlockSmoother"/>
      <Parameter name="CoarseSolver"      type="string"   value="myBlockDirectSolver"/>
      <Parameter name="P"                 type="string"   value="myRebBlockedPFact"/>
      <Parameter name="R"                 type="string"   value="myRebBlockedRFact"/>
      <Parameter name="A"                 type="string"   value="myRebBlockedAcFact"/>
      <Parameter name="Coordinates"       type="string"   value="myRebBlockedPFact"/>
    </ParameterList>

  </ParameterList>

</ParameterList>
<!-- end "MueLu" -->
```

Note, that we are using a coordinate-based rebalancing method from the Zoltan package. The **myInputCoordsFact** provides the **Coordinates** variable to the **ZoltanInterface**. That is, **myInputCoordsFact** uses user-provided data on the finest level and switches to Coordinates provided by the **myTransferCoordinatesFact** on the coarser level.

Note: To deal with potential logical circular dependencies between factories, you can use the **dependency for** keyword as demonstrated for the **myTransferCoordinatesFact** and **myInputCoordsFact** in this example. Note: you can always use the **dependency for** keyword to extend/change factory dependencies in the xml file.

2.14 Using MueLu in User Applications

This tutorial demonstrates how to use MueLu from within user applications in C++. We will use the `Tpetra` linear algebra stack. We will read the configuration for MueLu from an XML file and then create a `MueLu::Hierarchy`. This will then be used as a preconditioner within Belos as well as a standalone solver.

Note: There is also support for Stratimikos. Please refer to the `examples` in the MueLu folder for more details.

2.14.1 XML Interface using CreateTpetraPreconditioner

The most comfortable way to declare the multigrid parameters for MueLu is using the XML interface. In fact, MueLu provides two different XML interfaces. There is a simplified XML interface for multigrid users and a more advanced XML interface for experts, which allows to make use of all features of MueLu as a multigrid framework. Both XML file formats are introduced in the previous sections of this hands on tutorial. However, for the C++ code it makes no difference which type of XML interface is used.

Note: In the next sections, we give some code snippets. THey are borrowed form the `laplace2d.cpp` file in the tutorial tests.

Preparations

First of all, we need to grab a communicator object. Therefore, it is easy to use some utilities from the Teuchos package:

```
RCP<const Teuchos::Comm<int>> comm = Teuchos::DefaultComm<int>::getComm();
int MyPID = comm->getRank();
int NumProc = comm->getSize();
(void) MyPID;   // unused void pointer cast to avoid unused variable warnings
(void) NumProc; // unused void pointer cast to avoid unused variable warnings
```

For the multigrid method, we need a linear operator A. For demonstration purposes, here we just generate a 2D Laplacian operator using the Galeri package (see *Example problem*). In this example, we use Tpetra (throught the Xpetra wrappers) for the underlying linear algebra framework. For convenience, we ask the Galeri package to create a matrix of a Laplace2D problem:

For aggregation-based algebraic multigrid methods, one has to provide a valid set of near null space vectors to produce transfer operators. In case of a Laplace problem, we just use a constant vector.

```
// Build null space vector for a scalar problem, i.e. vector with all ones
RCP<MultiVector> nullspace = rcp(new MultiVector(dofMap, 1));
nullspace->putScalar(one);
```

Setup phase

With a fine level operator A available as `Tpetra::CrsMatrix` object and a set of near null space vectors (available as `Tpetra::MultiVector`), all minimum requirements are fulfilled for generating an algebraic multigrid hierarchy. There are two different ways to setup a multigrid hierarchy in MueLu. One can either use a parameter list driven setup process which accepts either `Teuchos::ParameterList` objects or XML files in two different XML file formats. Alternatively, one can use the MueLu C++ API to define the multigrid setup at compile time. In the next sections we show both variants.

The most comfortable way to declare the multigrid parameters for MueLu is using the XML interface. In fact, MueLu provides two different XML interfaces. There is a simplified XML interface for multigrid users and a more advanced XML interface for expert which allows to make use of all features of MueLu as a multigrid framework. Both XML file formats are introduced in the previous sections of this hands on tutorial. However, for the C++ code it makes no difference which type of XML interface is used.

We first read the MueLu configuration from the XML file:

```
    // Read MueLu parameter list from xml file
    RCP<ParameterList> mueluParams =  Teuchos::getParametersFromXmlFile(xmlFileName);
```

Then, we store the near null space vectors in the "user data" sublist of this parameter list:

```
    // Register nullspace as user data in the MueLu parameter list
    ParameterList& userDataList = mueluParams->sublist("user data");
    userDataList.set<RCP<MultiVector>>("Nullspace", nullspace);
```

We then can create a MueLu object ready to be used as a preconditioner:

```
    // Create the MueLu preconditioner based on the Tpetra stack
    RCP<MueLu::TpetraOperator<SC,LO,GO,NO>> mueLuPreconditioner =
        MueLu::CreateTpetraPreconditioner(Teuchos::rcp_dynamic_cast<Operator>(matrix),␣
↪*mueluParams);
```

Iteration Phase

Once the setup phase is completed, the MueLu multigrid hierarchy is ready for being used.

There are several ways how to use the multigrid method. One can apply the multigrid method as standalone solver for linear systems. Multigrid methods are also known to be efficient preconditioners within iterative (Krylov) solvers such as CG or GMRES methods.

In the next subsections it is demonstrated how to use MueLu as standalone solver and as preconditioner for iterative solvers from the Belos and AztecOO package in Trilinos.

For solving a linear system $Ax = b$, we need a right hand side vector b. When using iterative solvers we also need an initial guess for the solution vector.

In this example we just create `Tpetra::MultiVectors` (with just a single vector each). The right-hand side vector b is initialized with ones and the solution vector x is filled with random values.

MueLu as multigrid solver

MueLu can be used as standalone solver. First, we create and initialize a solution vector:

```
        // Create solution vector and set initial guess (already stored in X)
        RCP<MultiVector> multigridSolVec = rcp(new MultiVector(dofMap, 1, true));
        multigridSolVec->update(0.0, *X, 1.0);
```

If necessary, extract the `MueLu::Hiearchy` from the Tpetra preconditioner object:

```
        // Extract the underlying MueLu hierarchy
        RCP<MueLu::Hierarchy<SC,LO,GO,NO>> hierarchy = mueLuPreconditioner->
↪GetHierarchy();
```

Then, the `MueLu::Hierarchy` object is set to the non-preconditioner mode:

```
        // Configure MueLu to be used as solver
        hierarchy->IsPreconditioner(false);
```

Finally, we solve the system by calling the `Iterate()` routine to perform `mgridSweeps` sweeps with the chosen multigrid cycle.

```
        // Solve
        hierarchy->Iterate(*Xpetra::toXpetra(B), *Xpetra::toXpetra(multigridSolVec),␣
↪mgridSweeps);
```

If successful, the `multigridSolVec` vector contains the solution.

MueLu as preconditioner for Belos

Belos is the Krylov package in Trilinos and works both for Epetra and Tpetra. Here, we demonstrate how to use MueLu as preconditioner for Belos solvers using Tpetra.

First, we create and initialize a solution vector:

```
        // Create solution vector and set initial guess (already stored in X)
        RCP<MultiVector> precSolVec = rcp(new MultiVector(dofMap, 1, true));
        precSolVec->update(0.0, *X, 1.0);
```

Then, we create a `Belos::LinearProblem` and hand in the MueLu preconditioner object:

```
        // Construct a Belos LinearProblem object and hand-in the MueLu preconditioner
        RCP<Belos::LinearProblem<SC,MultiVector,Operator>> belosProblem =
            rcp(new Belos::LinearProblem<SC,MultiVector,Operator>(matrix, precSolVec,␣
↪B));
        belosProblem->setLeftPrec(mueLuPreconditioner);
        bool set = belosProblem->setProblem();
```

Now, we define the linear solver configuration in a `Teuchos::ParameterList` and create the Belos solver with this configuration:

```
        // Belos parameter list
        RCP<ParameterList> belosList = Teuchos::parameterList();
        belosList->set("Maximum Iterations", 50); // Maximum number of iterations allowed
```

(continues on next page)

(continued from previous page)

```
        belosList->set("Convergence Tolerance", tol); // Relative convergence tolerance␣
↪requested
        belosList->set("Verbosity", Belos::Errors + Belos::Warnings +␣
↪Belos::StatusTestDetails);
        belosList->set("Output Frequency", 1);
        belosList->set("Output Style", Belos::Brief);

        // Create an iterative solver manager
        Belos::SolverFactory<SC,MultiVector,Operator> solverFactory;
        RCP<Belos::SolverManager<SC,MultiVector,Operator>> solver = solverFactory.create(
↪"Block GMRES", belosList);

        // Pass the linear problem to the solver
        solver->setProblem(belosProblem);
```

Finally, we solve the linear system:

```
        // Perform solve
        Belos::ReturnType retStatus = Belos::Unconverged;
        retStatus = solver->solve();
```

Full example using the XML interface

The reader may refer to `laplace2d.cpp` for a working example to study the source code. This demonstration program has some more features that are not discussed in this tutorial.

Exercise 1

Compile the example in `laplace2d.cpp` and then run the program in parallel using two processors

```
mpirun -np 2 ./MueLu_tutorial_laplace2d.exe --help
```

Study the screen output and try to run the example with an XML file as input for the multigrid setup.

Exercise 2

Create large scale examples using the `--nx` and `--ny` parameters for a finer mesh.

Choose reasonable numbers for `--nx` and `--ny` for your machine and make use of your knowledge about MueLu for generating efficient preconditioners.

2.14.2 C++ Interface

As an alternative to the XML interfaces, the user can also define the multigrid hierarchy using the C++ API directly. In contrary to the XML interface, which allows to build the layout of the multigrid preconditioner at runtime, the preconditioner is fully defined at compile time when using the C++ interface.

First, a `MueLu::Hierarchy` object has to be defined, which manages the multigrid hierarchy including all multigrid levels. It provides routines for the multigrid setup and the multigrid cycle algorithms (such as V-cycle and W-cycle).

```
// create new hierarchy
RCP<MueLu::Hierarchy<SC, LO, GO, NO> > H;
```

There are some member functions which can be used to describe the basic multigrid hierarchy. The `SetMaxCoarseSize` member function is used to set the maximum size of the coarse level problem before the coarsening process can be stopped.

```
// instantiate new Hierarchy object
H = rcp(new Hierarchy());
H->setDefaultVerbLevel(Teuchos::VERB_HIGH);
H->SetMaxCoarseSize((GO) optMaxCoarseSize);
```

Next, one defines an empty `MueLu::Level` object for the finest level. The `MueLu::Level` objects represent a data container storing the internal variables on each multigrid level. The user has to provide and fill the level container for the finest level only. The `MueLu::Hierarchy` object then automatically generates the coarse levels using the multigrid parameters. The absolute minimum requirements for the finest level that the user has to provide is the fine level operator A which represents the fine level matrix. MueLu is based on Xpetra. So, the matrix A has to be of type `Xpetra::Matrix`. In addition, the user should also provide a valid set of near null space vectors. For a Laplace problem we can just use the constant `nullspace` vector that has previously been defined. Some routines need additional information. For example, the user has to provide the node coordinates for repartitioning.

```
// create a fine level object
RCP<Level> Finest = H->GetLevel();
Finest->setDefaultVerbLevel(Teuchos::VERB_HIGH);
Finest->Set("A",           A);
Finest->Set("Nullspace",   nullspace);
```

Note:

When including the `MueLu_UseShortNames.hpp` header file,

the template parameters usually can be dropped for compiling. The most important template parameters are `SC` for the scalar type, `LO` for the local ordinal type and `GO` for the global ordinal type. For a detailed description of the template parameters, the reader may refer to the Tpetra documentation.

A `MueLu::FactoryManager` object is used for the internal management of data dependencies and generating algorithms of the multigrid setup. Even though not absolutely necessary, we show the usage of the `MueLu::FactoryManager` object as it allows for user-specific enhancements of the multigrid code.

```
FactoryManager M;
M.SetKokkosRefactor(false);
```

The user can define its own factories for performing different tasks in the setup process. The following code shows how to define a smoothed aggregation transfer operator and a restriction operator. The `MueLu::RAPFactory` is used for the (standard) Galerkin product to generate the coarse level matrix A.

```
// declare some factories (potentially overwrite default factories)
RCP<SaPFactory> PFact = rcp(new SaPFactory());
PFact->SetParameter("sa: damping factor", Teuchos::ParameterEntry(optSaDamping));

RCP<Factory>    RFact = rcp(new TransPFactory());

RCP<RAPFactory> AcFact = rcp(new RAPFactory());
AcFact->setVerbLevel(Teuchos::VERB_HIGH);
```

The user-defined factories have to be registered in the `FactoryManager` using the lines

```
// configure factory manager
M.SetFactory("P", PFact);
M.SetFactory("R", RFact);
M.SetFactory("A", AcFact);
```

Warning:

If you forget to register the new factories, the `FactoryManager` will use some internal default factories for being responsible to create the corresponding variables.

Then your user-specified factories are just ignored during the multigrid setup!

Note:

The `FactoryManager` is also responsible for resolving all dependencies between different factories.

That is, after the user-defined factories have been registered, all factories that request variable P are provided with the prolongation operator P that has been generated by the registered factory `PFact`. If there is some data requested for which no factory has been registered by the user, the `FactoryManager` manages an internal list for reasonable default choices and default factories.

Next, the user has to declare a level smoother. The following code can be used to define a symmetric Gauss-Seidel smoother. Other methods can be set up in a similar way.

```
// define smoother object
std::string ifpackType;
Teuchos::ParameterList ifpackList;
ifpackList.set("relaxation: sweeps", (LO) optSweeps);
ifpackList.set("relaxation: damping factor", (SC) 1.0);
if (optSmooType == "sgs") {
  ifpackType = "RELAXATION";
  ifpackList.set("relaxation: type", "Symmetric Gauss-Seidel");
}
```

Before the level smoother can be used, a `MueLu::SmootherFactory` has to be defined for the smoother factory. The `SmootherFactory` is used in the multigrid setup to generate level smoothers for the corresponding levels using the prototyping design pattern. Note, that the `SmootherFactory` has also to be registered in the `FactoryManager` object. If the user forgets this, the multigrid setup will use some kind of default smoother, i.e., the user-chosen smoother options are just ignored.

```
    // create smoother factory
    RCP<SmootherPrototype> smootherPrototype = rcp(new TrilinosSmoother(ifpackType,␣
↪ifpackList));
    M.SetFactory("Smoother", rcp(new SmootherFactory(smootherPrototype)));
```

Once the `FactoryManager` is set up, it can be used with the `Hierarchy::Setup` routine to initiate the coarsening process and set up the multigrid hierarchy.

```
  // setup multigrid hierarchy
  int startLevel = 0;
  H->Setup(M, startLevel, optMaxLevels);
```

2.14.3 Footnotes

2.15 ML ParameterList interpreter

2.15.1 Backwards compatibility

ML[1] is the predecessor multigrid package of MueLu in Trilinos and widely used in the community for smoothed aggregation multigrid methods. ML is implemented in C and known for its good performance properties. However, the disadvantage is that ML is harder to adapt to new applications and non-standard problems. Furthermore, ML uses its own internal data structure and is somewhat limited to the use with Epetra objects only. In contrast, MueLu provides a fully flexible multigrid framework which is designed to be adapted to any kind of new application with non-standard requirements. Furthermore, it is based on Xpetra and therefore can be used both with Epetra or Tpetra. Nevertheless, it is an important point to provide some kind of backwards compatibility to allow ML users to easily migrate to MueLu (or make experiments with MueLu without having to write to much new code).

In this tutorial we present the **MueLu::MLParameterListInterpreter** which provides support for the most important ML parameters to be used with MueLu.

2.15.2 C++ part

Preparations

In order to use MueLu (instead or aside of ML) you first have to add it to your application. Please refer to the MueLu user guide for information about compilation and linking (see[2]). Basically, if your application is already working with ML, you should only need to compile and install MueLu and make sure that the MueLu libraries are found by the linker.

[1] M. W. Gee, C. M. Siefert, J. J. Hu, R. S. Tuminaro, and M. G. Sala. ML 5.0 Smoothed Aggregation User's Guide, Sandia National Laboratories, 2006, SAND2006-2649

[2] L. Berger-Vergiat, C. A. Glusa, G. Harper, J. J. Hu, M. Mayr, P. Ohm, A. Prokopenko, C. M. Siefert, R. S. Tuminaro, and T. A. Wiesner. MueLu User's Guide. Technical Report SAND2023-12265, Sandia National Laboratories, Albuquerque, NM (USA) 87185, 2023.

C++ interface

In the following we assume that the linear operator A is available as **RCP<Xpetra::Matrix> A**.

Then we create a parameter list and fill it with ML parameters.

```
params = rcp(new Teuchos::ParameterList());

params->set("ML output", 10);
params->set("max levels", 2);
params->set("smoother: type", "symmetric Gauss-Seidel");
params->set("coarse: type", "Amesos-KLU");
```

Please refer to the ML guide[Page 82, 1] for a complete list of available parameters.

Note:

Be aware that the MLParameterListInterpreter does not support all ML parameters,

but only the most important ones (e.g., smoothers, transfer operators, rebalancing, ...).

Instead of defining the ML parameters by hand in the ParameterList, you can also read in XML files with ML parameters using

```
params = Teuchos::getParametersFromXmlFile(xmlFileName);
```

Next, you create a **MLParameterListInterpreter** object using the parameters and create a new **MueLu::Hierarchy** from it.

```
MLParameterListInterpreter mueLuFactory(*params);
RCP<Hierarchy> hierarchy = mueLuFactory.CreateHierarchy();
```

Of course, you have to provide all necessary information for the multigrid setup routine. This does not only include the fine level operator, but also the set of near null space vectors. Assuming that **numPDEs** stores the number of equations (and near null space vectors), the following code allows to produce piecewise constant standard near null space vectors (which should be valid for many PDE discretizations).

```
RCP<MultiVector> nullspace = MultiVectorFactory::Build(A->getDomainMap(), numPDEs);
for (int i = 0; i < numPDEs; ++i)
{
  Teuchos::ArrayRCP<Scalar> nsValues = nullspace->getDataNonConst(i);
  const int numBlocks = nsValues.size() / numPDEs;

  for (int j = 0; j < numBlocks; ++j)
    nsValues[j*numPDEs + i] = STS::one();
}
```

Then, we just feed in the information to the finest level:

```
hierarchy->GetLevel(0)->Set("Nullspace", nullspace);
hierarchy->GetLevel(0)->Set("A", A);
```

Finally, we call the **Setup** routine which actually builds the multigrid hierarchy:

```
mueLuFactory.SetupHierarchy(*hierarchy);
```

Once we have the multigrid hierarchy set up, we can use it the same way as described in *Iteration Phase*.

Exercise 1

Study the source code of **../../../test/tutorial/MLParameterList.cpp** and compile it.

Run the executable **MueLu_tutorial_MLParameterList.exe** with the **–help** command line parameter to get an overview of all available command line parameters.

Run the example on a 1D mesh with 256 elements using

./MueLu_tutorial_MLParameterList.exe –xml=xml/ml_ParameterList.xml –nx=256

and study the MueLu output.

Note: You will see a warning by the **MLParameterListInterpreter**, that the parameter list could not be validated. The reason is the follwing: Since this tutorial example runs with the Tpetra backend, ML, which is purely Epetra-based, cannot validate the parameter list.

Exercise 2

Play around with the parameters from **MueLu_tutorial_MLParameterList.exe**.

Change, e.g., the problem type to a 2D Laplace problem (**–matrixType=Laplace2D**) and adapt the **–nx** and **–ny** parameters accordingly.

2.15.3 Footnotes

2.16 A. Virtual box image

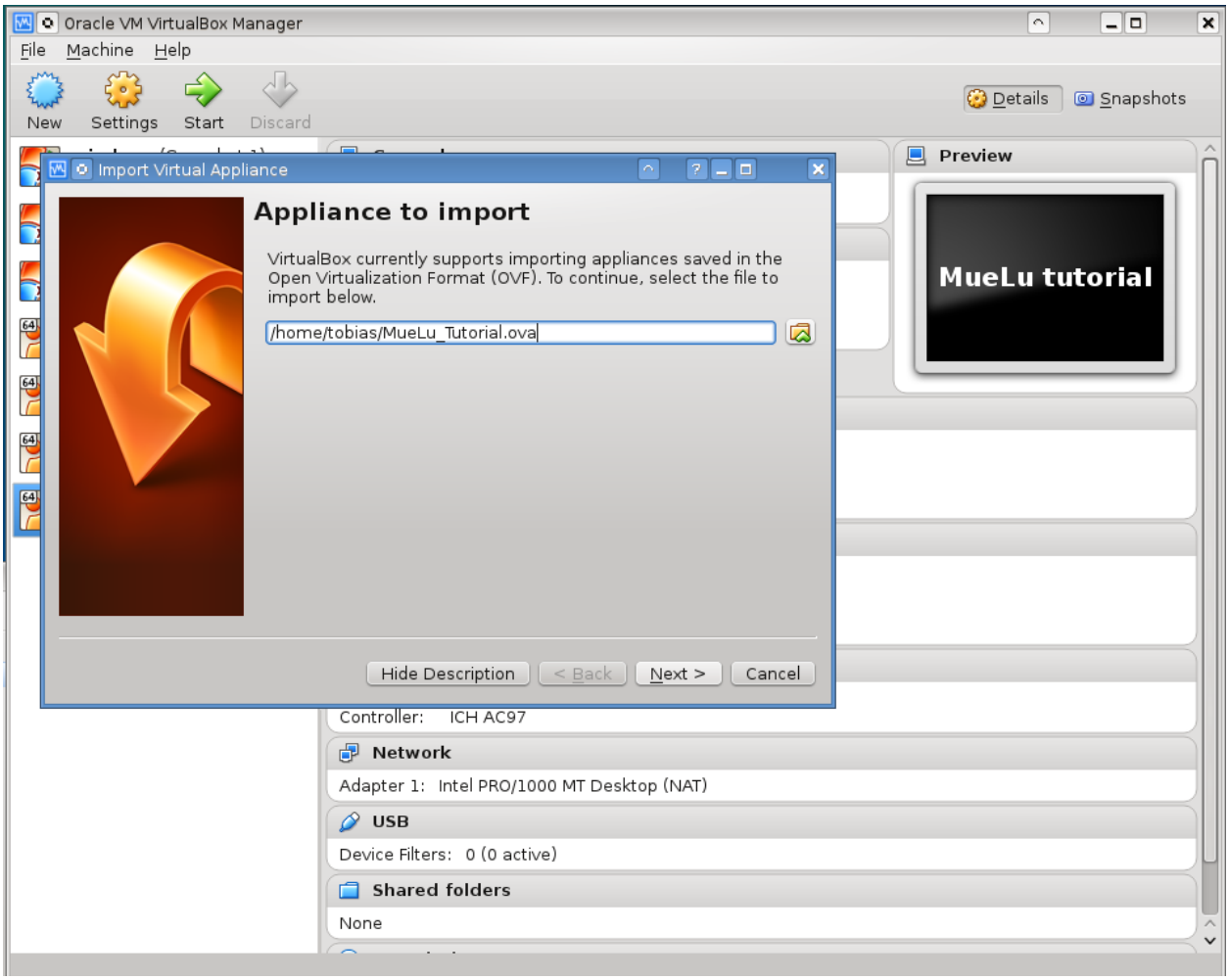
This chapter discusses the basics of the virtual box image that comes with this tutorial to allow the user to follow above explanations and do its own experiments with MueLu and Trilinos. A virtual machine has the advantage that it is rather easy to set up for a user. Even though compiling and installing got easier the last years by using a cmake based build system it is still a nightmare for not so experienced users. The virtual machine runs both on Linux and Windows as host and brings all the necessary tools for a quick start to MueLu.

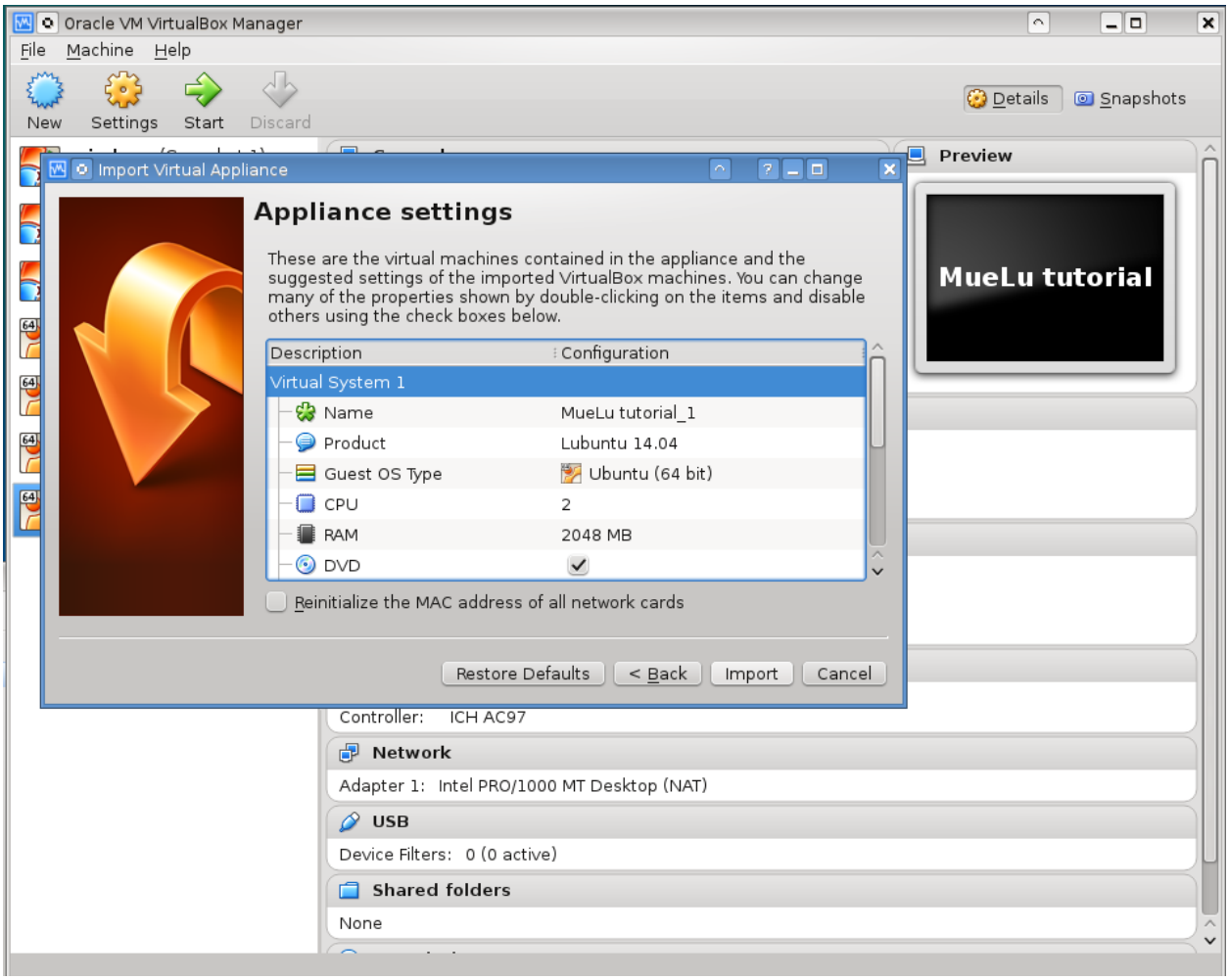
2.16.1 Preparations

To use the virtual machine you basically have to perform the following steps.

1. Install **VirtualBox** on your host machine. You can download it from **www.virtualbox.org**.
2. Download the **MueLu_Tutorial.ova** virtual machine. The image file has 4 GB.
3. Run **VirtualBox** and import the **MueLu_Tutorial.ova** machine.

   Then, check and adapt the settings of the virtual machine.


Oracle VM VirtualBox Manager
File Machine Help
New Settings Start Discard
Details Snapshots
Preview
MueLu tutorial
Import Virtual Appliance
Appliance to import
VirtualBox currently supports importing appliances saved in the Open Virtualization Format (OVF). To continue, select the file to import below.
/home/tobias/MueLu_Tutorial.ova
Hide Description
< Back
Next >
Cancel
Controller: ICH AC97
Network
Adapter 1: Intel PRO/1000 MT Desktop (NAT)
USB
Device Filters: 0 (0 active)
Shared folders
None


Oracle VM VirtualBox Manager
File Machine Help
New Settings Start Discard
Details Snapshots
Preview
MueLu tutorial
Import Virtual Appliance
Appliance settings
These are the virtual machines contained in the appliance and the suggested settings of the imported VirtualBox machines. You can change many of the properties shown by double-clicking on the items and disable others using the check boxes below.
Description Configuration
Virtual System 1
Name MueLu tutorial_1
Product Lubuntu 14.04
Guest OS Type Ubuntu (64 bit)
CPU 2
RAM 2048 MB
DVD
Reinitialize the MAC address of all network cards
Restore Defaults < Back Import Cancel
Controller: ICH AC97
Network
Adapter 1: Intel PRO/1000 MT Desktop (NAT)
USB
Device Filters: 0 (0 active)
Shared folders
None

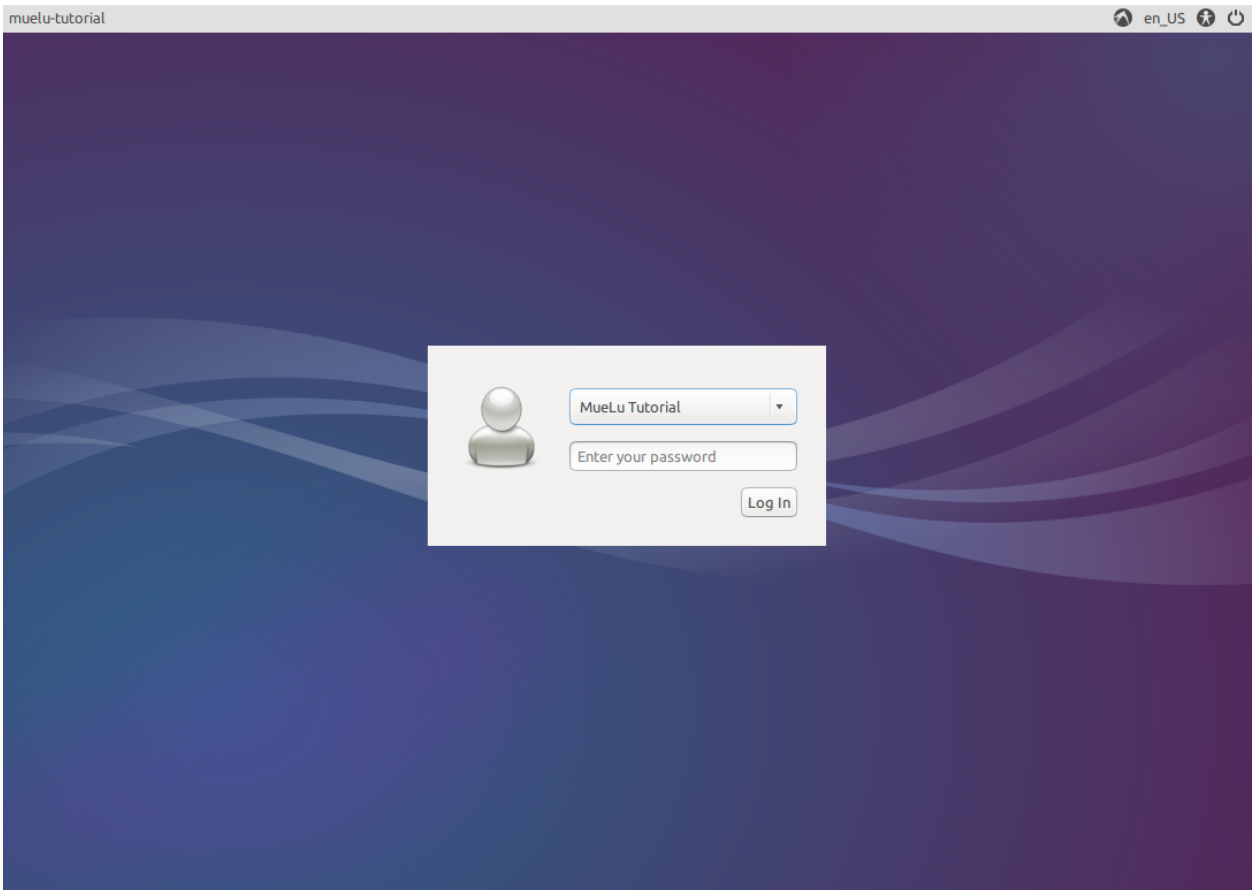
In general, one processor should be enough. But if you want to make some more reasonable tests with parallel multigrid you should increase the number of processors accordingly. Click import, to import the virtual machine.

4. With a click on the start button the virtual machine is booting.

2.16.2 First steps

Login and setup

Once the virtual machine is started you first have to login.



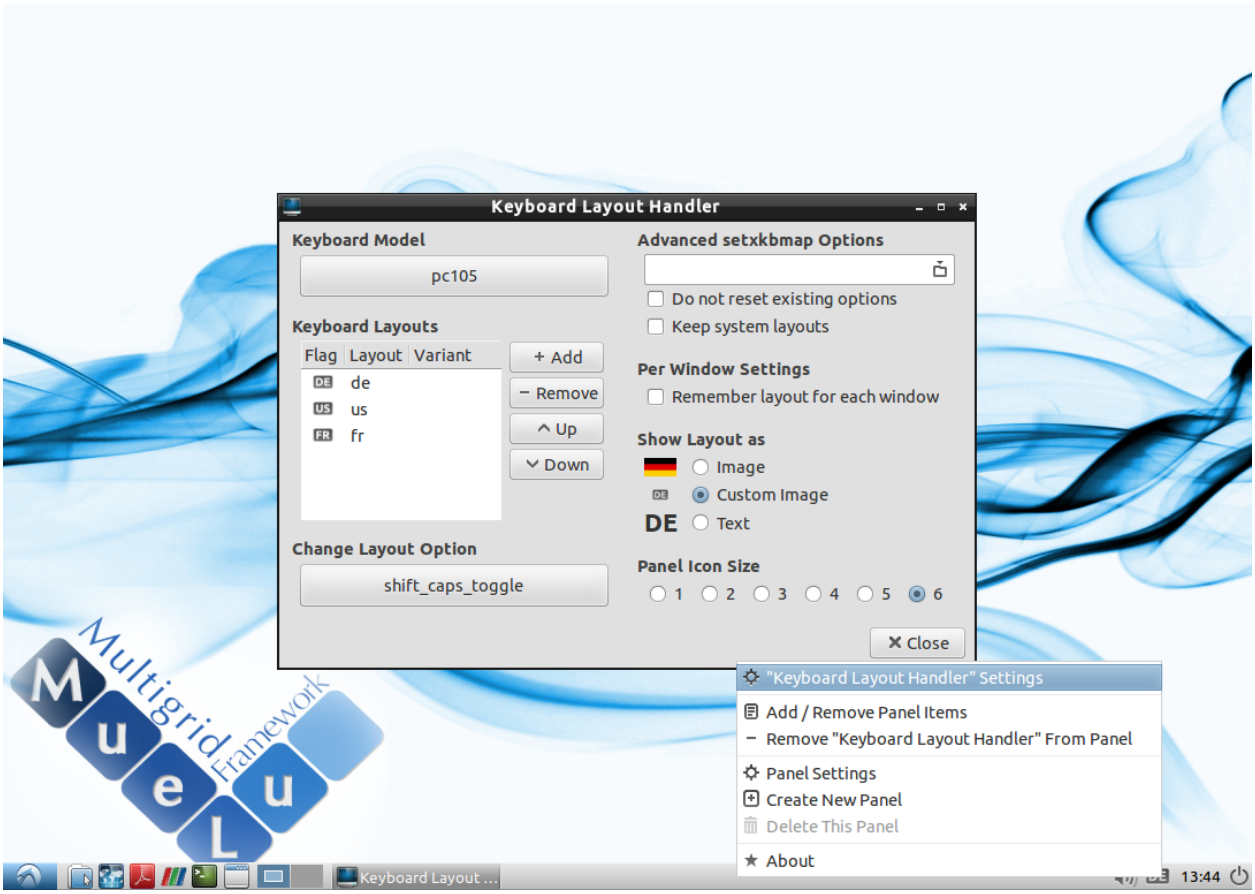
The login data is:

```
Username: muelu
Password: sandia
```

Note: You only need to enter the password in above screen.

After the login you should see the following desktop.

First, you should adapt the language settings and the keyboard layout. You can switch the keyboard layout by clicking on the logo in the lower right corner. A right click on the logo allows you to change more details

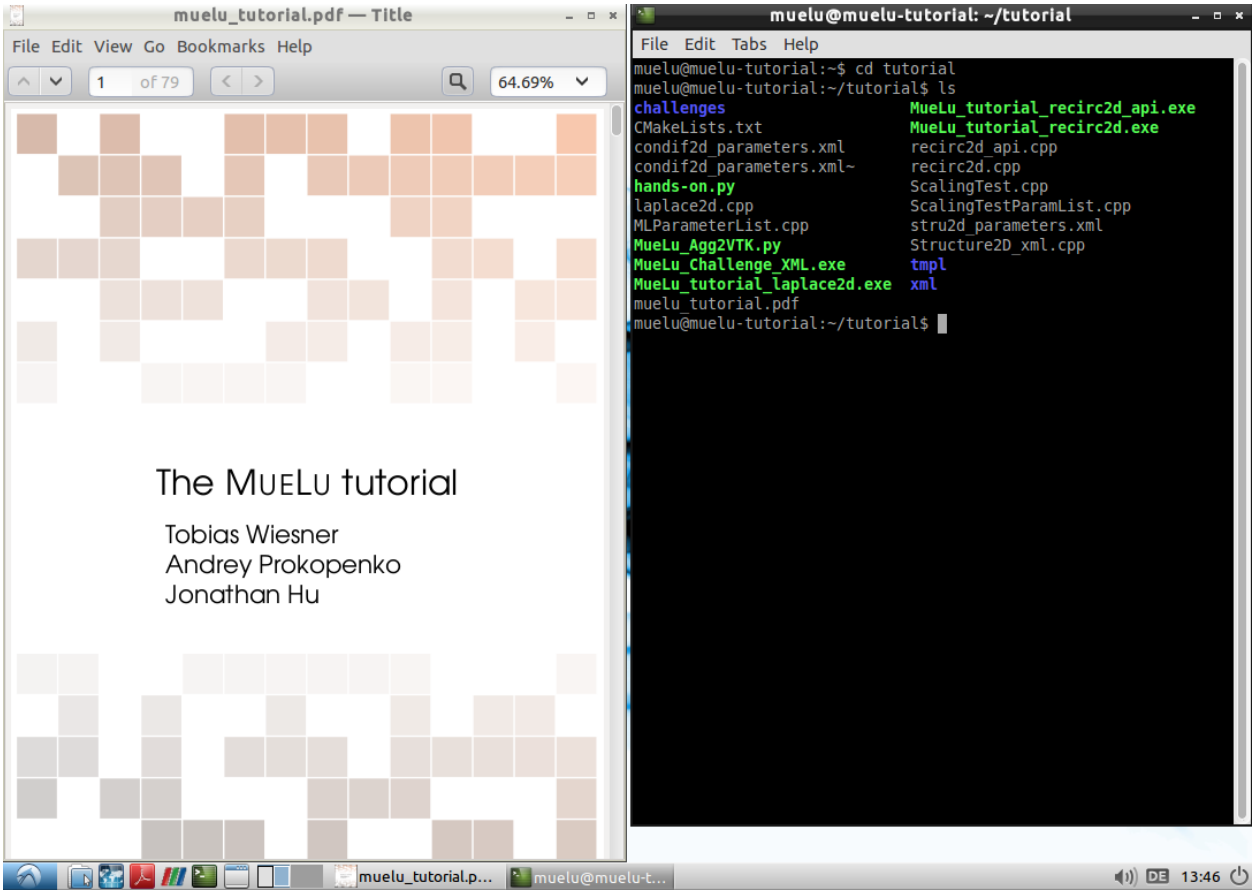


Then you are ready to go with the tutorial.

MueLu tutorial

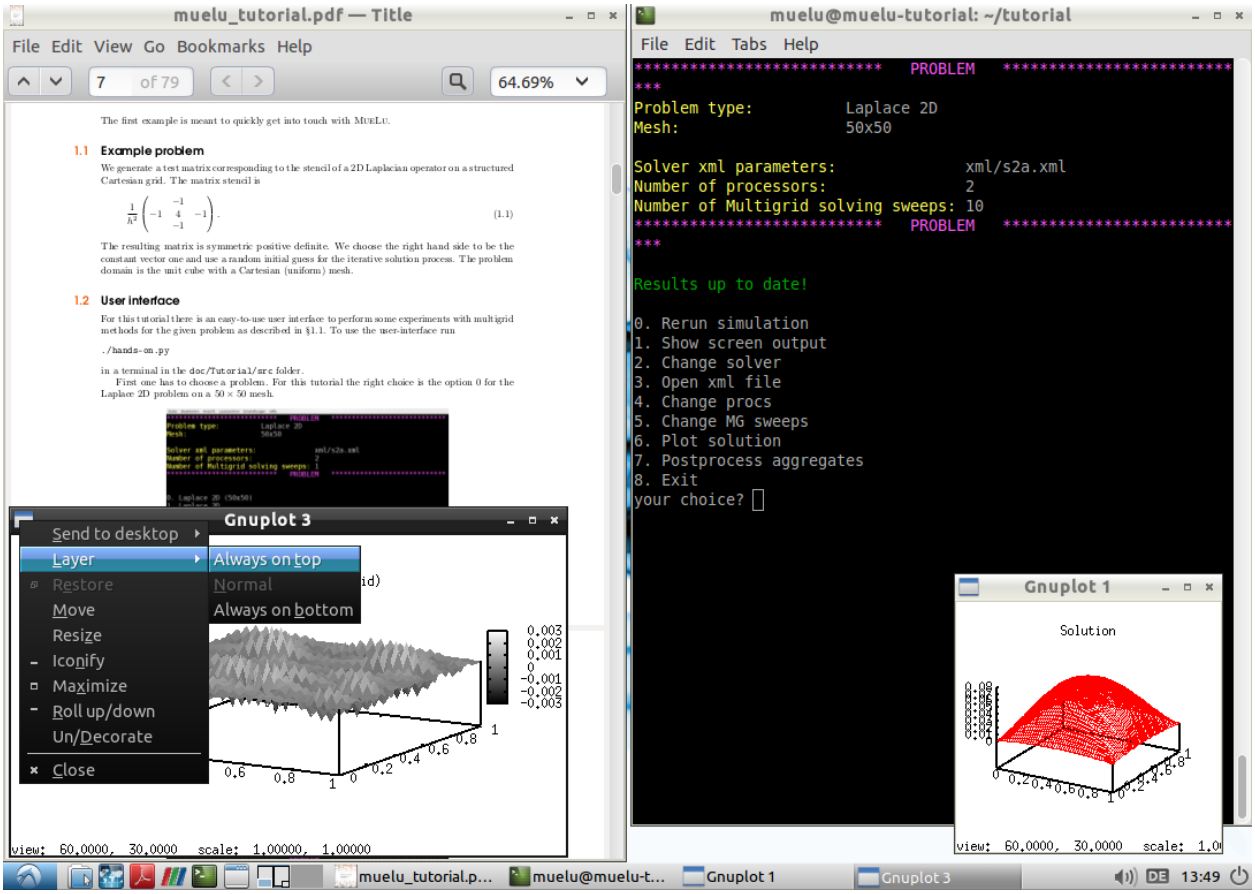
Open the tutorial with **evince** as pdf viewer. To open **evince** you can either use the shortcut in the lower left corner of your desktop or press `Alt + F2` to open the **Run** dialog and enter **evince**. Load the **muelu_tutorial.pdf** file in the **tutorial** folder of your home directory.

To open a terminal you have several option. Either use the shortcut button in the lower left corner. Alternatively you can open the **Run** dialog (`Alt + F2`) and enter **lxterminal**. As a third alternative you can just press `Ctrl`+ :kbd:`Alt + T`. In the terminal, change to the **tutorial** folder by entering **cd tutorial**. Therein you can find the **hands-on.py** script which is used throughout the whole MueLu tutorial.

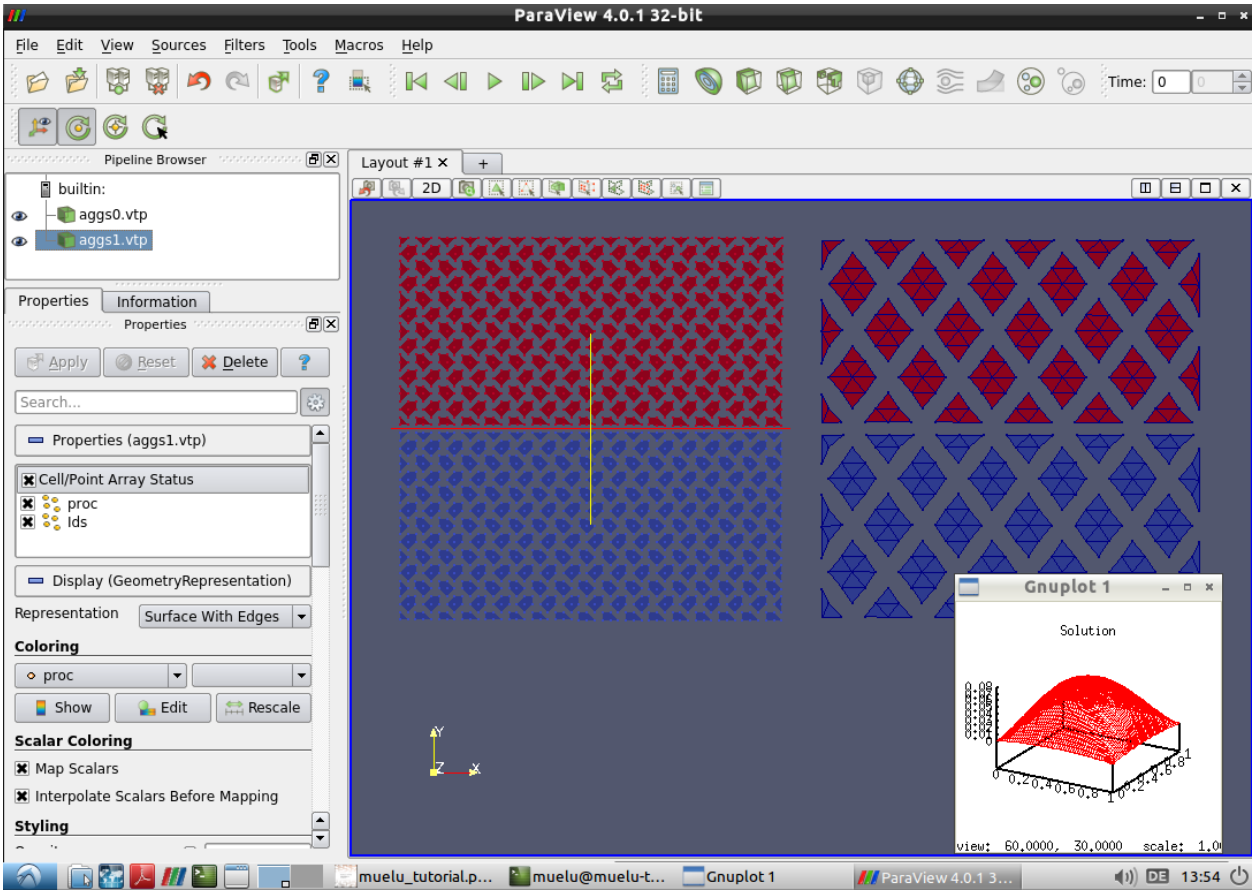


Note: Use the `Win` + `<-` and `Win` + `->` keys to arrange the windows in a split view as shown above. There are other useful keyboard shortcuts such as `Win` + `R` to open the **Run** dialog or `Win` + `E` to open the file manager.

When plotting the results with gnuplot from within the **hands-on.py** script it might be useful to make the plot windows to stay on top.



The virtual machine has all software installed that you need to follow the tutorial (including **paraview**)



2.16.3 Software

The virtual machine is based on a minimal installation of **Lubuntu 14.04**. The image file has 4 GB with about 250 MB free for the user.

The following software is pre-installed:

```
Web-browser: midori
PDF-viewer: evince
Terminal: LXTerminal
Visualization: paraview, gnuplot
File manager: PCManFM
Analysis: FreeMat v4.0
GNU octave 3.8.1
```

The following system libraries are installed:

```
Trilinos: Trilinos (developer branch: Oct 1, 2014)
Direct solver: SuperLU 4.3
VTK: VTK 5.8
MPI: OpenMPI 1.6.5
Python: Python 2.7.6
Compiler: gcc 4.8.2
```

2.17 B. Docker Container

This chapter discusses the basics of the Docker container that comes with this tutorial to allow the user to follow above explanations and do its own experiments with MueLu and Trilinos. A docker container has the advantage that it is rather easy to set up for a user. Even though compiling and installing is easier in recent years by using a cmake based build system, it may be difficult for not so experienced users. The Docker container runs on any machine with Docker installed and brings all the necessary tools for a quick start to MueLu.

2.17.1 Preparations

To use the Docker container you must perform the following steps.

1. Install **Docker** on your host machine. You can download it from **www.docker.com**.
2. Download the MueLu **docker-images** repository. It is located at **https://github.com/GrahamBenHarper/docker-images**.
3. In the **docker-images/muelu-tutorial** directory, run *build-container.sh*. This will take some time depending on your machine.

   Warning: Insert screen output of Docker building or pulling

4. Once the build is complete, run **run-container.sh** to run a new tutorial container. The tutorial is installed in the */opt/trilinos/build/packages/muelu/test/tutorial* directory of the Docker image.

   Warning: Insert screen output of Docker running

Note: The container does not have a graphical interface, so some of the visualization aspects of the tutorial may be unavailable.

2.17.2 Software

The Docker container is based on CentOS-Stream 9. The container requires approximately 8GB after Trilinos is built.

The following software is pre-installed:

```
Terminal: bash
Text editor: nano
Version control: git
```

The following system libraries are installed:

```
MPI: OpenMI 4.1.1
Python: Python 3.9.18
Compiler: gcc 11.4.1
CMake: 3.26.1
NetCDF
HDF5
```

(continues on next page)

(continued from previous page)

```
BLAS
LAPACK
Boost
Atlas
```

2.18 Error Messages

This section summarizes typical error messages and hints how to address the underlying problem.

2.18.1 Syntax errors

Parser errors

```
XML parse error at line 27: file ended before closing element 'ParameterList' from line 1
```

It seems like you forgot to close the **<ParameterList>** section that is opened in line 1 of the xml file. Close it by adding the line **</ParameterList>** at the end of the (sub-)list.

```
XML parse error at line 15: start element not well-formed: invalid character
```

Check line 15 for an invalid xml format. The reason can be, e.g., a missing closing character **/>** for a parameter.

Parameter list errors

```
All child nodes of a ParameterList must have a name attribute!
```

You probably forgot to add a name attribute in one or more elements of your xml file, that is you used, e.g.,

- <Parameter type="string" value="RELAXATION"/>

instead of

- <Parameter name="smoother: type" type="string" value="RELAXATION"/>

```
Error, the parameter {name="smoother: type",type="int",value="0"} in the parameter␣
↪(sub)list "ANONYMOUS" exists in the list of valid parameters but has the wrong type.
```

Choose the coorect type. The correct type depends on the parameter at hand. In this example, the correct type is "string".

```
Use the correct (proposed) value type for the given parameter name, i.e.,
```

- <Parameter name="smoother: type" type="string" value="RELAXATION"/>

instead of

- <Parameter name="smoother: type" type="int" value="RELAXATION"/>

2.18.2 MueLu errors

General errors

```
Throw test that evaluated to true: s_.is_null()

Smoother for Tpetra was not constructed
    during request for data "    PreSmoother" on level 0 by factory NoFactory
```

Failed to create a level smoother. Check the smoother blocks in your xml file. The error occurs, e.g., if there is a typing error in the **smoother: type** parameter. For example

- <Parameter name="smoother: type" type="string" value="REXATION"/>

would trigger above error since the smoother type should be **RELAXATION**.

```
IFPACK ERROR -2, ifpack/src/Ifpack_PointRelaxation.cpp, line 117
```

Errors like this indicate that it is a problem within the **smoother: params** section. Most likely a (relaxation) smoother is requested which is not existing (e.g., **Jadobi** instead of **Jacobi**).

Warning: Switch this example to Ifpack2.

```
The parameter name "smother: type" is not valid. Did you mean "smoother: type"?
```

There is a typo in your parameter list. Locate the parameter and fix it (possibly using the suggestions, that come with the error message).

```
Throw test that evaluated to true: maxNodesPerAggregate < minNodesPerAggregate
```

Choose the **aggregation: min agg size** parameter to be smaller than the **aggregation: max agg size** parameter for the aggregation routine.

Advanced XML file format

```
Throw test that evaluated to true: bIsZeroNSColumn == true

MueLu::TentativePFactory::MakeTentative: fine level NS part has a zero column
```

This error indicates that there is a problem with the provided near null space vectors. There are different reasons which can trigger this problem:

- The near null space vectors are not valid (containing zeros, wrong ordering of internal degrees of freedom).

Please check your near null space vectors. Maybe there is an empty vector or the ordering of degrees of freedom for the linear operator does not match with the ordering of the near null space vectors. * The near null space vectors are correct but used in a wrong way (e.g., a wrong number of degrees of freedom). Check the screen output for wrong block dimensions (CoalesceDropFactory). * There is a problem with the aggregates. Validate the screen output and look for unusual (e.g. very small or empty) aggregates.

```
Throw test that evaluated to true: factoryManager_ == null

MueLu::Level(0)::GetFactory(Aggregates, 0): No FactoryManager
```

This is a typical error when the dependency tree is screwed up. If aggregates and/or transfer operators are involved, one usually has forgotten some entries in the **Hierarchy** sublist of the extended XML file format for the internal factory managers. These errors can be quite tricky to fix. In general, it is a good idea to start with a working XML file and extend it step by step if possible. The following general strategies may help to track down the problem:

- Run the problem with **verbosity=extreme** to get as much screen output as possible.

Check for unusual screen output (such as **Nullspace factory**). * Try to generate a graphical dependency tree as described in *Dependency trees*.

For example, above error is caused by the following XML file

```
<ParameterList name="MueLu">
<ParameterList name="Factories">
    <ParameterList name="myTentativePFact">
    <Parameter name="factory" type="string" value="TentativePFactory"/>
    </ParameterList>
</ParameterList>

<ParameterList name="Hierarchy">
    <ParameterList name="Levels">
    <Parameter name="P" type="string" value="myTentativePFact"/>
    <!--<Parameter name="Nullspace" type="string" value="myTentativePFact"/>-->
    </ParameterList>
</ParameterList>
</ParameterList>
```

When looking at the error output, it seems to be a problem with aggregates. However, no special aggregation factory has been declared in the XML file. The only factory which has been introduced was a tentative prolongation factory for generating unsmoothed transfer operators. Therefore, one should start digging into the details of the **TentativePFactory** to find out, that the unsmoothed transfer operator factory is responsible both for creating the unsmoothed prolongator and the coarse level null space information. When looking at the screen output one should find that the last called/generated factory is a **NullspaceFactory**, which can also be a hint that the problem is the null space.

When looking at the XML file, one can see that the **myTentativePFact** factory has been registered to be responsible for generating the prolongator P, but the generating factory for the variable **Nullspace** is not declared. MueLu tries to generate the default null space, but since it does not know about **myTentativePFact** to be a **TentativePFactory**, which would already produce the needed information, the call ordering of the dependent factories (e.g., aggregation) gets mixed up.

Note that the **TentativePFactory** is special. If you declare an explicit instance of the **TentativePFactory**, you always have to register it for generating the **Nullspace** variable, too. Only in very special cases, this would not be necessary.

Note: This is a general rule: if a factory generates more than one output variables, always make sure that all these output variables are properly defined in the **FactoryManager** list (or **Hierarchy** sublist in the xml files, respectively).

To solve above problem there are two possibilities:

- Following above comment, just register **myTentativePFact** for generating **Nullspace**.

That is, just comment in the corresponding line in above xml file. * Alternatively, you can register **myTentativePFact** for generating **Ptent** (and **P**). This way you mark the **myTentativePFact** object to be used for generating the unsmoothed transfer operators (and state that they shall be used for the final prolongation operators). MueLu is smart enough to understand that the factory responsible for generating **Ptent** is also supposed to generate the null space vectors.

CHAPTER

THREE

INDICES AND TABLES

- genindex
- modindex
- search