The MueLu tutorial

Tobias Wiesner
Michael Gee
Andrey Prokopenko
Jonathan Hu

Tobias A. Wiesner
Institute for Computational Mechanics
Technische Universität München
Boltzmannstr. 15
85747 Garching
Germany

Michael W. Gee
Mechanics & High Performance Computing Group
Technische Universität München
Parkring 35
85748 Garching
Germany

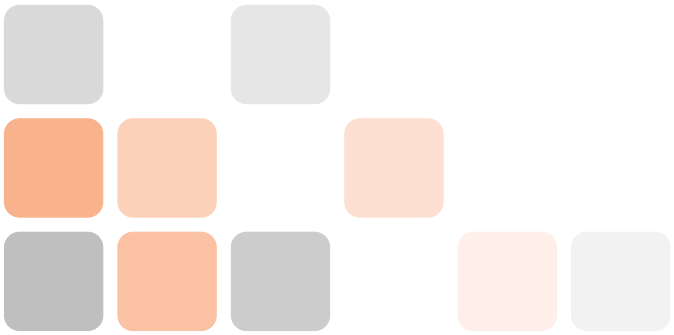
Andrey Prokopenko
Scalable Algorithms
Sandia National Laboratories
Mailstop 1318
P.O. Box 5800
Albuquerque, NM 87185-1318

Jonathan J. Hu
Scalable Algorithms
Sandia National Laboratories
Mailstop 9159
P.O. Box 0969
Livermore, CA 94551-0969

Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000
HTTP://TRILINOS.ORG/PACKAGES/MUELU/

SAND2014-18624 R
First printing, September 2014

Version: ed3342a



Preface

Idea and concept

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 is expected to have a basic understanding of multigrid methods and its general underlying concepts. Please refer to multigrid textbooks (e.g. [1]) for the theoretical background.

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 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 five tutorials which are 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 such as rebalancing are handled as well as aggregation strategies. Additional exercises help the reader to perform some experiments in practice.

The third part is meant for expert 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.

References

For a complete overview of all features and available parameters in MueLu the reader may refer to the MueLu user guide [2]. For the most current version of MueLu it is recommended to visit the homepage

`http://trilinos.org/packages/muelu/`

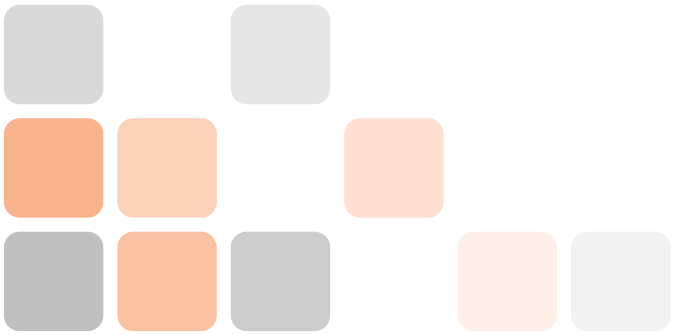
If you find errors in this tutorial, please contact the MueLu user list

`muelu-users@software.sandia.gov`

Acknowledgments

Many people have helped to develop MueLu, and we would like to acknowledge their contributions: Tom Benson, Julian Cortial, Jeremie Gaidamour, Axel Gerstenberger, Chetan Jhurani, Mark Hoemmen, Jonathan Hu, Paul Lin, Eric Phipps, Andrey Prokopenko, Chris Siefert, Paul Tsuji, Ray Tuminaro, and Tobias Wiesner.

Beginners tutorial



1. Quick start

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

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

$$\frac{1}{h^2}\begin{pmatrix} & -1 & \\ -1 & 4 & -1 \\ & -1 & \end{pmatrix}, \tag{1.1}$$

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.

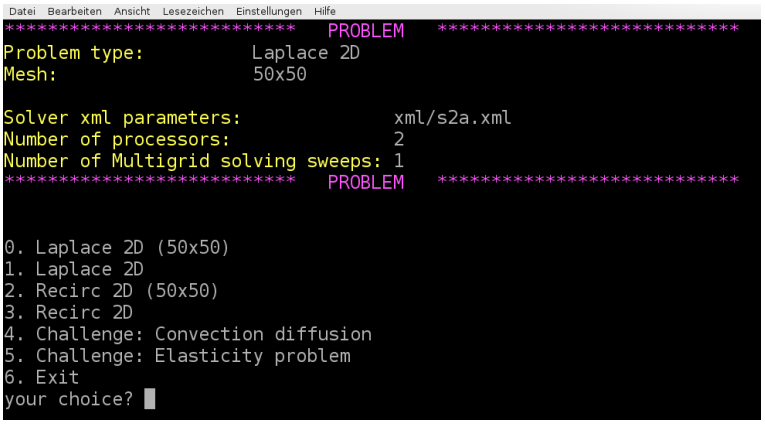
1.2 User interface

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 §1.1. To use the user-interface run

```
./hands-on.py
```

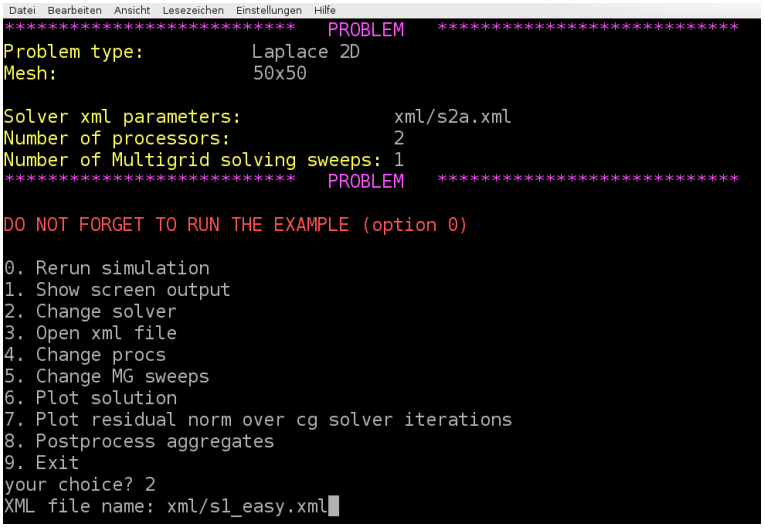
in a terminal in the `doc/Tutorial/src` folder.

First one has to choose a problem. For this tutorial the right choice is the option 0 for the Laplace 2D problem on a 50×50 mesh.



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

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



The `s1_easy.xml` file has the following content

```
<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>

```

```
21    <!-- Aggregation -->
22    <Parameter name="aggregation: type" type="string" value="uncoupled"/>
23    <Parameter name="aggregation: min agg size" type="int" value="3"/>
24    <Parameter name="aggregation: max agg size" type="int" value="9"/>
25
26  </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 (from the AztecOO package) with a MueLu multigrid preconditioner and a standalone multigrid solver (again using MueLu) with the given multigrid parameters.

```
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   **************************

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? 0
PREPARE SIMULATON
RUN EXAMPLE
mpirun -np 2 MueLu_tutorial_laplace2d.exe --nx=50 --ny=50 --mgridSweep
POSTPROCESSING...
COMPLETE
Press any key to continue...
```

Note that the line `mpirun -np 2 MueLu_tutorial_laplace2d.exe -nx ...` is the command that is executed in the background. Per default are 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!*

```
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?
```

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

To check the output select option 1 and you should find the following screen output:

```
Clearing old data (if any)
Level 0
 Setup Smoother (MueLu::IfpackSmoother{type = point relaxation stand-alone})
  IFPACK (Local Jacobi, sweeps=1, damping=0.9)
Level 1
 Prolongator smoothing (MueLu::SaPFactory)
  Matrix filtering (MueLu::FilteredAFactory)
   Build (MueLu::CoalesceDropFactory)
    lightweight wrap = 1
    algorithm = "classical": threshold = 0, blocksize = 1
    Detected 0 Dirichlet nodes
    Number of dropped entries in unamalgamated matrix graph: 0/12300 (0%)
   Filtered matrix is not being constructed as no filtering is being done
  Build (MueLu::TentativePFactory)
   Build (MueLu::UncoupledAggregationFactory)
    BuildAggregates (Phase - (Dirichlet))
    BuildAggregates (Phase 1 (main))
    BuildAggregates (Phase 2 (cleanup))
    BuildAggregates (Phase 3 (emergency))
    BuildAggregates (Phase - (isolated))
    "UC": MueLu::Aggregates{nGlobalAggregates = 442}
   Build (MueLu::AmalgamationFactory)
   Build (MueLu::CoarseMapFactory)
  Prolongator damping factor = 0.698709 (1.33 / 1.90351)
 Transpose P (MueLu::TransPFactory)
 Computing Ac (MueLu::RAPFactory)
 Setup Smoother (MueLu::IfpackSmoother{type = point relaxation stand-alone})
  IFPACK (Local Jacobi, sweeps=1, damping=0.9)
Level 2
 Prolongator smoothing (MueLu::SaPFactory)
  Matrix filtering (MueLu::FilteredAFactory)
   Build (MueLu::CoalesceDropFactory)
    lightweight wrap = 1
    algorithm = "classical": threshold = 0, blocksize = 1
    Detected 0 Dirichlet nodes
    Number of dropped entries in unamalgamated matrix graph: 0/3870 (0%)
   Filtered matrix is not being constructed as no filtering is being done
  Build (MueLu::TentativePFactory)
   Build (MueLu::UncoupledAggregationFactory)
    BuildAggregates (Phase - (Dirichlet))
    BuildAggregates (Phase 1 (main))
    BuildAggregates (Phase 2 (cleanup))
    BuildAggregates (Phase 3 (emergency))
    BuildAggregates (Phase - (isolated))
    "UC": MueLu::Aggregates{nGlobalAggregates = 60}
   Build (MueLu::AmalgamationFactory)
   Nullspace factory (MueLu::NullspaceFactory)
   Build (MueLu::CoarseMapFactory)
  Prolongator damping factor = 1.00326 (1.33 / 1.32567)
 Transpose P (MueLu::TransPFactory)
 Computing Ac (MueLu::RAPFactory)
 Setup Smoother (MueLu::AmesosSmoother{type = Superlu})
```

created with MueLu version ed3342a

☛ 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 inter-medium level and level 2 for the coarsest level). Then

an overview of the different multigrid levels is given by

```
Number of levels  = 3
Operator complexity = 1.36

matrix rows   nnz  nnz/row procs
A 0   2500  12300    4.92  2
A 1    442   3870    8.76  2
A 2     60    520    8.67  2

Smoother (level 0) both : IFPACK (Local Jacobi, sweeps=1, damping=0.9)

Smoother (level 1) both : IFPACK (Local Jacobi, sweeps=1, damping=0.9)

Smoother (level 2) pre : MueLu::AmesosSmoother{type = Superlu}
Smoother (level 2) post : no smoother
```

created with MueLu version ed3342a

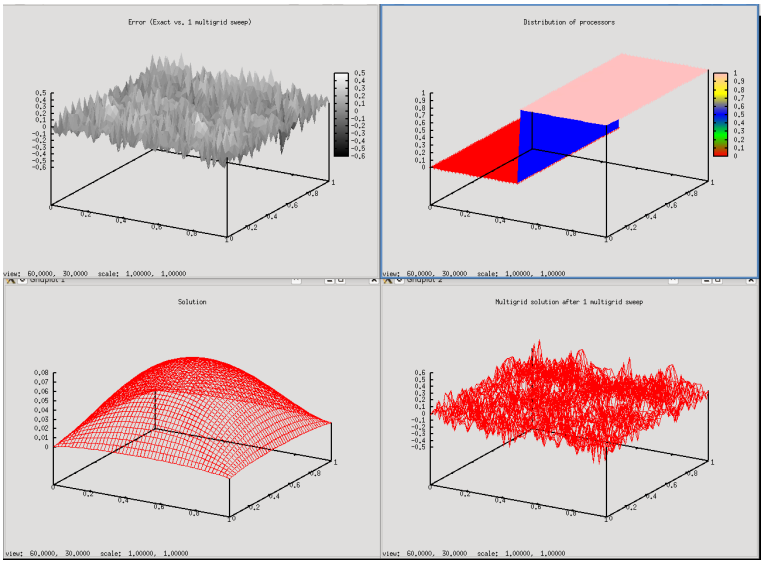
One can see that a three level multigrid method is used with a direct solver on the coarsest level and Jacobi level smoothers on the fine and inter-medium level. Furthermore some basic information is printed such as the operator complexity.

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

```
Use multigrid hierarchy as preconditioner within CG.

    ******************************************************
    ***** Problem: Epetra::CrsMatrix
    ***** Preconditioned CG solution
    ***** MueLu::Hierarchy
    ***** No scaling
    ******************************************************

               iter:   0          residual = 1.000000e+00
               iter:   1          residual = 1.948066e-01
               iter:   2          residual = 4.763987e-02
               iter:   3          residual = 1.132820e-02
               iter:   4          residual = 2.633924e-03
               iter:   5          residual = 6.648822e-04
               iter:   6          residual = 1.678971e-04
               iter:   7          residual = 3.924505e-05
               iter:   8          residual = 9.137502e-06
               iter:   9          residual = 2.138738e-06
               iter:  10          residual = 5.292825e-07
               iter:  11          residual = 1.343805e-07
               iter:  12          residual = 2.985605e-08
               iter:  13          residual = 7.104820e-09
               iter:  14          residual = 1.671271e-09
               iter:  15          residual = 3.909860e-10
               iter:  16          residual = 9.618993e-11
               iter:  17          residual = 2.174033e-11
               iter:  18          residual = 5.279059e-12
               iter:  19          residual = 1.314498e-12
               iter:  20          residual = 3.148552e-13


    Solution time: 0.048807 (sec.)
    total iterations: 20
```

created with MueLu version ed3342a

Selecting option 6 gives you four plots.



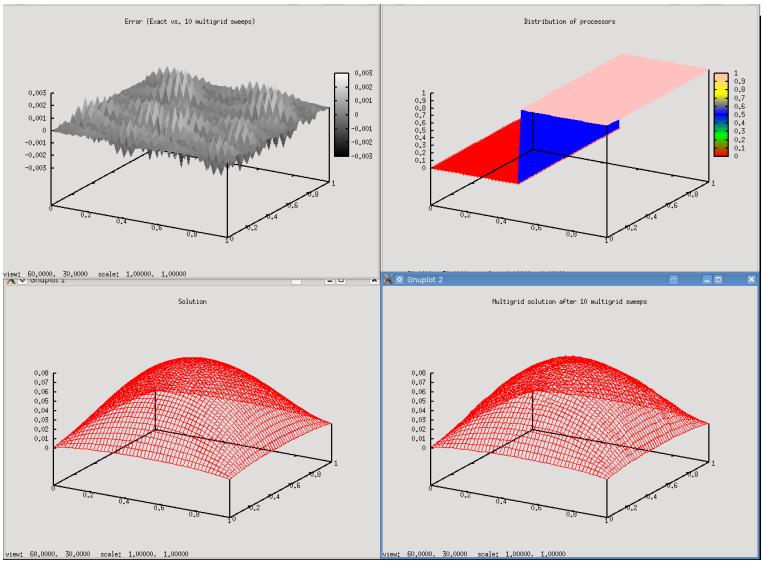
The lower left plot shows the exact solution of the linear system (using a direct solver from the AMESOS package). The lower right plot shows the multigrid solution when 1 sweep with a V-cycle of the multigrid method as defined in the xml parameter file is applied to the linear system as a standalone 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, that 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.

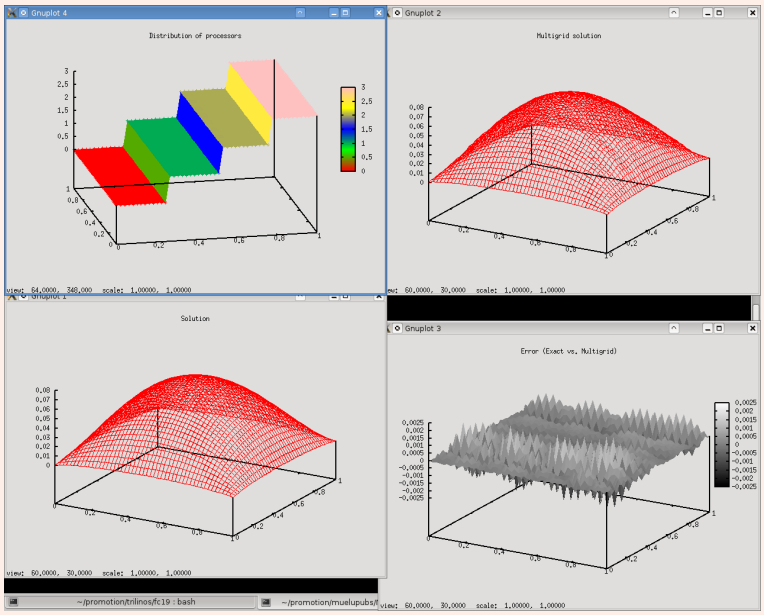
```
Datei  Bearbeiten  Ansicht  Lesezeichen  Einstellungen  Hilfe
*****************************   PROBLEM   *****************************
Problem type:          Laplace 2D
Mesh:                  50x50

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.

Exercise 1.1 Change the number of processors. Use option 4 and select for example 4 processors (instead of 2). Rerun the example and plot the results.



■

Exercise 1.2 Compare the output when using 4 processors with the output for only 2 processors. Is the number of aggregates changing? Is there some effect on the quality of the multigrid solution. How does the number and convergence history change for the preconditioned CG method? ■

Exercise 1.3 Choose option 9 to close the program. ■

1.3 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

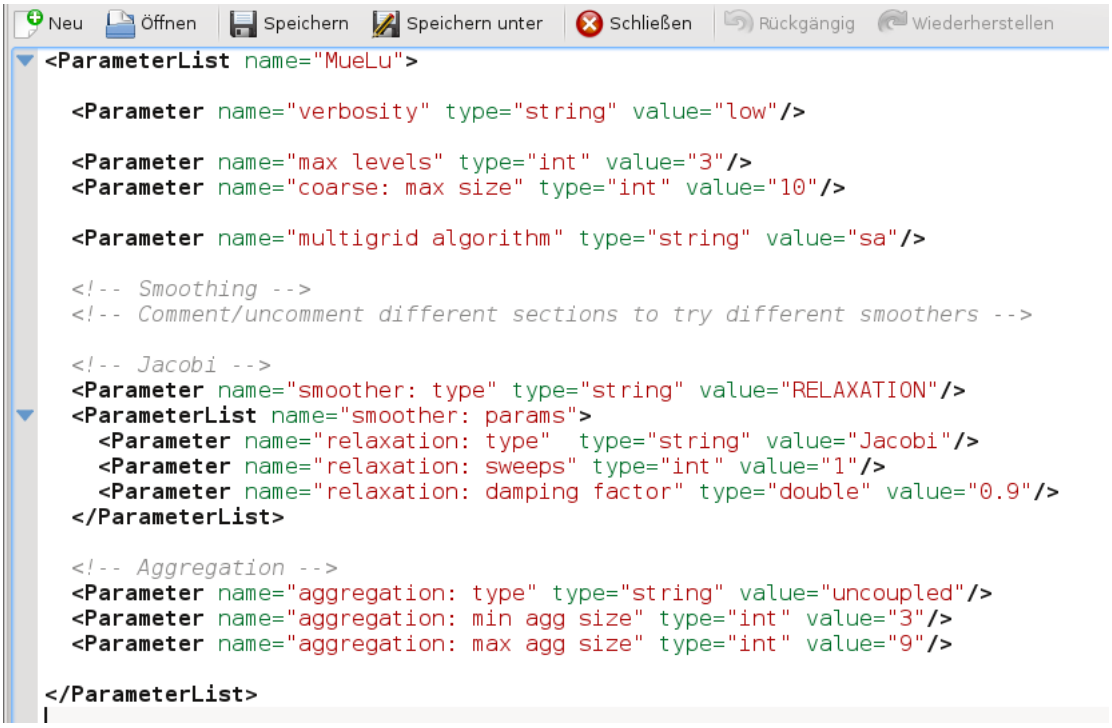
```
cp xml/s1_easy.xml mysolver.xml
```

Then, we run the driver program again using

```
./hands-on.sh
```

and choose option 0 for the 2D Laplace example on the 50×50 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 you `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.



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

```
Number of levels    = 5
Operator complexity = 1.35

matrix rows   nnz  nnz/row procs
A 0   10000 49600    4.96 2
A 1    1700 15148    8.91 2
A 2     208  2050    9.86 2
A 3      25   275   11.00 2
A 4       4    16    4.00 2

Smoother (level 0) both : IFPACK (Local Jacobi, sweeps=1, damping=0.9)

```

```
Smoother (level 1) both : IFPACK (Local Jacobi, sweeps=1, damping=0.9)

Smoother (level 2) both : IFPACK (Local Jacobi, sweeps=1, damping=0.9)

Smoother (level 3) both : IFPACK (Local Jacobi, sweeps=1, damping=0.9)

Smoother (level 4) pre : MueLu::AmesosSmoother{type = Superlu}
Smoother (level 4) post : no smoother
```

created with MueLu version ed3342a

Note that 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.

Exercise 1.4 How do you have to choose the `coarse: max size` parameter to obtain a 3 level multigrid method again? Increase the parameter accordingly, rerun the simulation and check your results. ■

Exercise 1.5 What happens if you allow only for a 1 level method (i.e., no multigrid)? How does this affect the preconditioned CG method? ■

The option `sa` for *smoothed aggregation* in the `multigrid algorithm` parameter can be considered to be optimal for symmetric positive definite 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

```
Use multigrid hierarchy as preconditioner within CG.

    *******************************************************
    ***** Problem: Epetra::CrsMatrix
    ***** Preconditioned CG solution
    ***** MueLu::Hierarchy
    ***** No scaling
    *******************************************************

               iter:   0         residual = 1.000000e+00
               iter:   1         residual = 1.726353e-01
               iter:   2         residual = 3.571945e-02
               iter:   3         residual = 1.101478e-02
               iter:   4         residual = 4.190157e-03
               iter:   5         residual = 2.378439e-03
               iter:   6         residual = 1.313185e-03
               iter:   7         residual = 6.982733e-04
               iter:   8         residual = 3.655210e-04
               iter:   9         residual = 2.069851e-04
               iter:  10         residual = 1.169833e-04
               iter:  11         residual = 7.085908e-05
               iter:  12         residual = 4.030429e-05
               iter:  13         residual = 2.453853e-05
               iter:  14         residual = 1.375104e-05
               iter:  15         residual = 7.741195e-06
               iter:  16         residual = 4.689433e-06
               iter:  17         residual = 2.600967e-06
               iter:  18         residual = 1.451541e-06
               iter:  19         residual = 8.757440e-07
               iter:  20         residual = 4.975362e-07
               iter:  21         residual = 2.641755e-07
               iter:  22         residual = 1.412237e-07
               iter:  23         residual = 8.373356e-08
```

```
34              iter:   24          residual = 4.714368e-08
35              iter:   25          residual = 2.465322e-08
36              iter:   26          residual = 1.339658e-08
37              iter:   27          residual = 7.170277e-09
38              iter:   28          residual = 4.338305e-09
39              iter:   29          residual = 2.438524e-09
40              iter:   30          residual = 1.458018e-09
41              iter:   31          residual = 7.910160e-10
42              iter:   32          residual = 4.096269e-10
43              iter:   33          residual = 2.343424e-10
44              iter:   34          residual = 1.263163e-10
45              iter:   35          residual = 6.958107e-11
46              iter:   36          residual = 3.744427e-11
47              iter:   37          residual = 2.094458e-11
48              iter:   38          residual = 1.107644e-11
49              iter:   39          residual = 6.050847e-12
50              iter:   40          residual = 3.482129e-12
51              iter:   41          residual = 1.943140e-12
52              iter:   42          residual = 1.111635e-12
53              iter:   43          residual = 6.438972e-13
54
55
56    Solution time: 0.150637 (sec.)
57    total iterations: 43
```

created with MueLu version ed3342a

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

```
1  Use multigrid hierarchy as preconditioner within CG.
2
3     *******************************************************
4     ***** Problem: Epetra::CrsMatrix
5     ***** Preconditioned CG solution
6     ***** MueLu::Hierarchy
7     ***** No scaling
8     *******************************************************
9
10              iter:    0          residual = 1.000000e+00
11              iter:    1          residual = 2.062059e-01
12              iter:    2          residual = 4.870154e-02
13              iter:    3          residual = 1.175258e-02
14              iter:    4          residual = 2.943278e-03
15              iter:    5          residual = 7.049471e-04
16              iter:    6          residual = 1.697510e-04
17              iter:    7          residual = 4.195728e-05
18              iter:    8          residual = 9.958322e-06
19              iter:    9          residual = 2.441874e-06
20              iter:   10          residual = 6.184789e-07
21              iter:   11          residual = 1.442079e-07
22              iter:   12          residual = 3.437446e-08
23              iter:   13          residual = 8.540255e-09
24              iter:   14          residual = 2.060443e-09
25              iter:   15          residual = 5.011928e-10
26              iter:   16          residual = 1.252446e-10
27              iter:   17          residual = 3.065168e-11
28              iter:   18          residual = 7.586370e-12
29              iter:   19          residual = 1.733282e-12
30              iter:   20          residual = 4.205447e-13
31
```

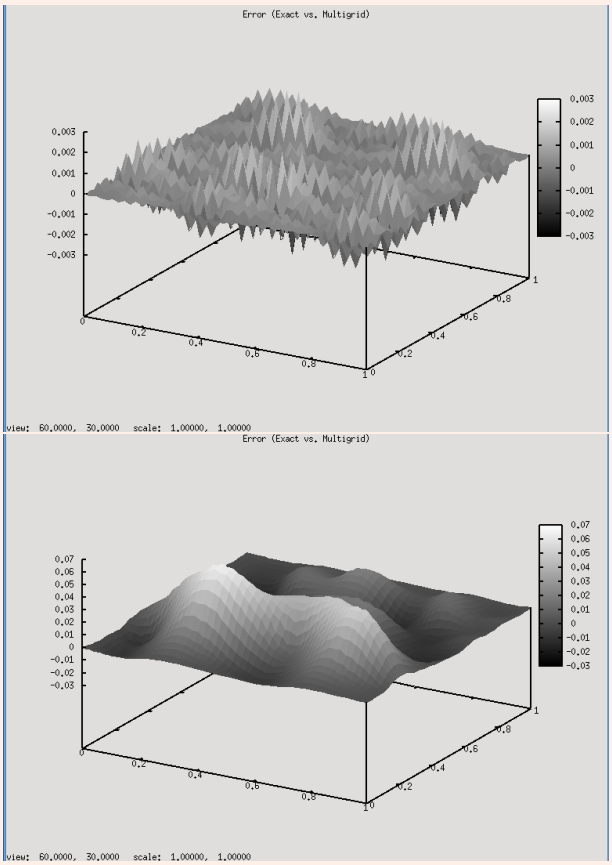
```
32
33    Solution time: 0.078784 (sec.)
34    total iterations: 20
```

created with MueLu version ed3342a

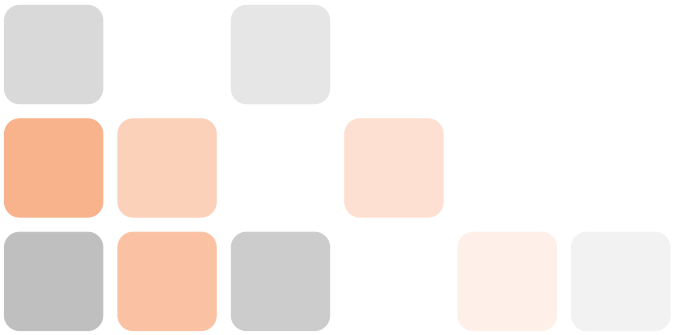
☛ You can find the corresponding xml files also in `xml/s1_easy_3levels_unsmoothed.xml` and `xml/s1_easy_3levels_smoothed.xml`.

Exercise 1.6 Compare the screen output of the unsmoothed multigrid method and the smoothed aggregation multigrid method. Which parts are missing in the multigrid setup for the unsmoothed multigrid method? How does the multigrid method affect the aggregates? ■

Exercise 1.7 Performing 10 multigrid sweeps both with the unsmoothed and the smoothed aggregation multigrid method gives the following error plots



Which one belongs to the unsmoothed multigrid method? ■



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.

2.1 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 basically the two typical cases one might find in practice.

Multigrid methods are based on the fact, 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 then are transferred to a coarse level where they can be seen as high frequency error component for a level smoother on the coarse level.

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.

2.2 Example

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 coarse solver.

```
1 <ParameterList name="MueLu">
2
3   <Parameter name="verbosity" type="string" value="low"/>
4
5   <Parameter name="max levels" type="int" value="10"/>
6   <Parameter name="coarse: max size" type="int" value="10"/>
7
8   <Parameter name="multigrid algorithm" type="string" value="unsmoothed"/>
9
```

```
  <!-- 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

```
Number of levels  = 5
Operator complexity = 1.26

matrix rows   nnz  nnz/row procs
A 0   10000 49600    4.96 2
A 1    1700 11476    6.75 2
A 2     216  1380    6.39 2
A 3      30   168    5.60 2
A 4       6    24    4.00 2

Smoother (level 0) both : IFPACK (Local Jacobi, sweeps=1, damping=0.9)

Smoother (level 1) both : IFPACK (Local Jacobi, sweeps=1, damping=0.9)

Smoother (level 2) both : IFPACK (Local Jacobi, sweeps=1, damping=0.9)

Smoother (level 3) both : IFPACK (Local Jacobi, sweeps=1, damping=0.9)

Smoother (level 4) pre : IFPACK (Local Jacobi, sweeps=1, damping=0.9)
Smoother (level 4) post : no smoother
```

created with MueLu version ed3342a

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

One has even more fine-grained control over pre- and post-smoothing.

```
<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: pre type" type="string" value="RELAXATION"/>
  <ParameterList name="smoother: pre params">
    <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="0.6"/>
  </ParameterList>

```

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

</ParameterList>
```

This produces the following multigrid hierarchy

```
Number of levels  = 5
Operator complexity = 1.26

matrix rows   nnz  nnz/row procs
A 0   10000 49600    4.96 2
A 1    1700 11476    6.75 2
A 2     216  1380    6.39 2
A 3      30   168    5.60 2
A 4       6    24    4.00 2

Smoother (level 0) pre : IFPACK (Local SGS, sweeps=3, damping=0.6)
Smoother (level 0) post : IFPACK (Local GS, sweeps=1, damping=0.9)

Smoother (level 1) pre : IFPACK (Local SGS, sweeps=3, damping=0.6)
Smoother (level 1) post : IFPACK (Local GS, sweeps=1, damping=0.9)

Smoother (level 2) pre : IFPACK (Local SGS, sweeps=3, damping=0.6)
Smoother (level 2) post : IFPACK (Local GS, sweeps=1, damping=0.9)

Smoother (level 3) pre : IFPACK (Local SGS, sweeps=3, damping=0.6)
Smoother (level 3) post : IFPACK (Local GS, sweeps=1, damping=0.9)

Smoother (level 4) pre : MueLu::AmesosSmoother{type = Superlu}
Smoother (level 4) post : no smoother
```

created with MueLu version ed3342a

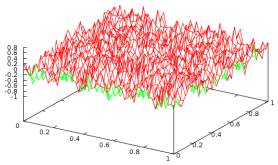
☛ Note that the relaxation based methods provided by the Ifpack package are embedded in an outer additive Schwarz method.

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 users guide ([2]).

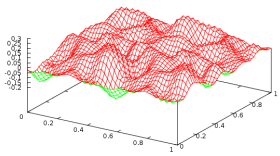
Exercise 2.1 Play around with the smoother parameters and study their effect on the error plot and the convergence of the preconditioned cg method. For all available smoothing options and parameters refer to the MueLu user guide ([2]). Hint: use `unsmoothed` transfer operator basis functions (i.e., `multigrid algorithm = unsmoothed`) to highlight the effect of the level smoothers. ■

Exercise 2.2 Use the following parameters to solve the 50×50 Laplace 2D problem on 2 processors

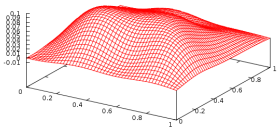
```
<ParameterList name="MueLu">

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



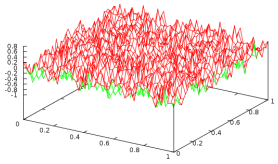
(a) 1 level with 1 Jacobi sweep ($\omega = 0.9$)



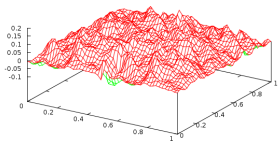
(b) 1 level with 10 Jacobi sweeps ($\omega = 0.9$)



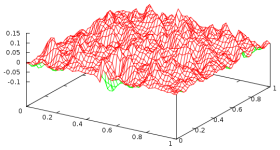
(c) 1 level with 100 Jacobi sweeps ($\omega = 0.9$)



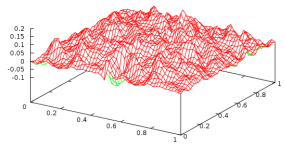
(d) 2 level with 1 Jacobi sweep ($\omega = 0.9$)



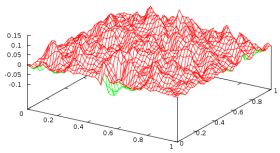
(e) 2 level with 10 Jacobi sweeps ($\omega = 0.9$)



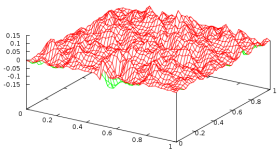
(f) 2 level with 100 Jacobi sweeps ($\omega = 0.9$)



(g) 3 level with 1 Jacobi sweep ($\omega = 0.9$)

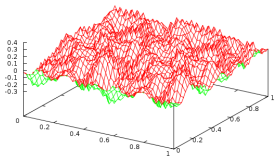


(h) 3 level with 10 Jacobi sweeps ($\omega = 0.9$)

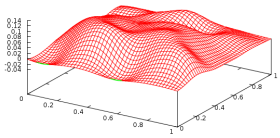


(i) 3 level with 100 Jacobi sweeps ($\omega = 0.9$)

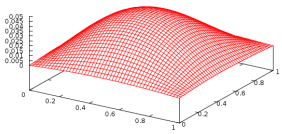
Figure 2.1: 2D Laplace equation on 50×50 mesh after 1 V-cycle with an AMG multigrid solver and Jacobi smoothers on all multigrid levels. (2 processors)



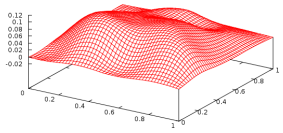
(a) 1 level with 1 Jacobi sweep ($\omega = 0.9$)



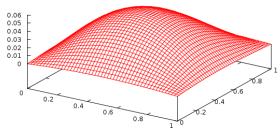
(b) 1 level with 10 Jacobi sweeps ($\omega = 0.9$)



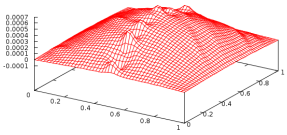
(c) 1 level with 100 Jacobi sweeps ($\omega = 0.9$)



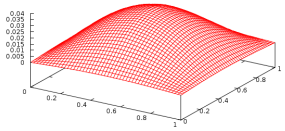
(d) 2 level with 1 Jacobi sweep ($\omega = 0.9$)



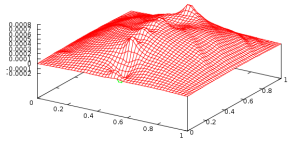
(e) 2 level with 10 Jacobi sweeps ($\omega = 0.9$)



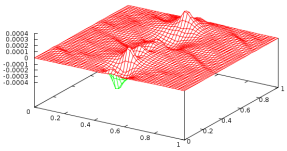
(f) 2 level with 100 Jacobi sweeps ($\omega = 0.9$)



(g) 3 level with 1 Jacobi sweep ($\omega = 0.9$)



(h) 3 level with 10 Jacobi sweeps ($\omega = 0.9$)

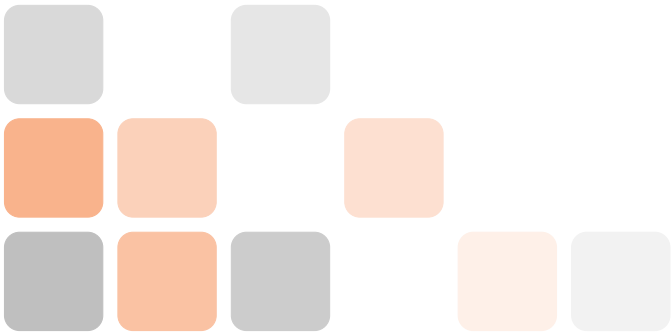


(i) 3 level with 100 Jacobi sweeps ($\omega = 0.9$)

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

```
4
5    <Parameter name="max levels" type="int" value="3"/>
6    <Parameter name="coarse: max size" type="int" value="10"/>
7    <Parameter name="multigrid algorithm" type="string" value="sa"/>
8
9    <!-- Jacobi -->
10   <Parameter name="smoother: type" type="string" value="RELAXATION"/>
11   <ParameterList name="smoother: params">
12     <Parameter name="relaxation: type" type="string" value="Jacobi"/>
13     <Parameter name="relaxation: sweeps" type="int" value="1"/>
14     <Parameter name="relaxation: damping factor" type="double" value="0.9"/>
15   </ParameterList>
16
17   <!-- Jacobi -->
18   <Parameter name="coarse: type" type="string" value="DIRECT"/>
19
20 </ParameterList>
```

That is, we change to smoothed aggregation AMG. You can find the xml file also in `xml/s1_easy_exercise.xml`. Run the example on 2 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? ■



3. Multigrid for non-symmetric problems

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.

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 `xml/s2a.xml` do not lead to a convergent multigrid preconditioner.

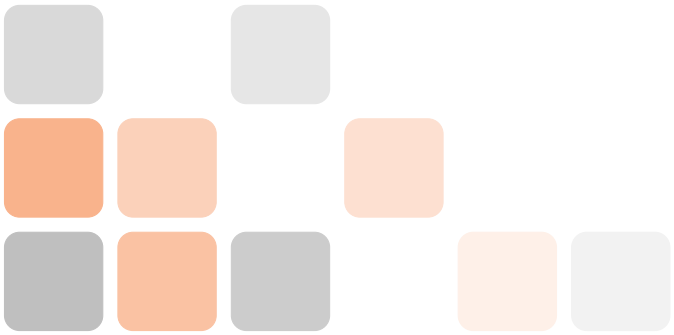
```
Datei  Bearbeiten  Ansicht  Lesezeichen  Einstellungen  Hilfe

*****************************   PROBLEM   *****************************
Problem type:          Recirc 2D
Mesh:                  50x50

Solver xml parameters:             xml/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 gmres solver iterations
8. Postprocess aggregates
9. Exit
your choice? 0
PREPARE SIMULATON
RUN EXAMPLE
mpirun -np 2 MueLu_tutorial_recirc2d.exe --nx=50 --ny=50 --mgridSweeps=1 --xml=xml/s2a.xml | tee output.log


        ************************************************************

        Warning: maximum number of iterations exceeded
        without convergence

        ************************************************************

POSTPROCESSING...
COMPLETE
Press any key to continue...
```

Choose the parameters from the `n1_easy.xml` file instead. If you run the example you might find that the GMRES method did not converge within 50 iterations.

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.

Exercise 3.1 Use `multigrid algorithm = pg` and compare the results with `multigrid algorithm = unsmoothed`. What is the difference in the number of GMRES iterations? What is changing in the multigrid setup? ■

Exercise 3.2 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. 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? ■



4. Useful tools for analysis

4.1 Visualization of aggregates

4.1.1 Technical prerequisites

MueLu allows to export plain aggregation information in simple text files that have to be interpreted by some 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`).

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

4.1.2 Visualization of aggregates with MueLu using VTK

We can visualize the aggregates using the vtk file format and paraview. First add the parameter `aggregation: export visualization data = true` to the list of aggregation parameters. Use, e.g., the following xml file

```
<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="pg"/>
  <Parameter name="sa: damping factor" type="double" value="1.33333"/>

  <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>

  <!-- 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"/>
  <Parameter name="aggregation: export visualization data" type="bool" value="true"/>
```

```
23
24 </ParameterList>
```

The file is stored in `xml/n2_easy_agg.xml`.

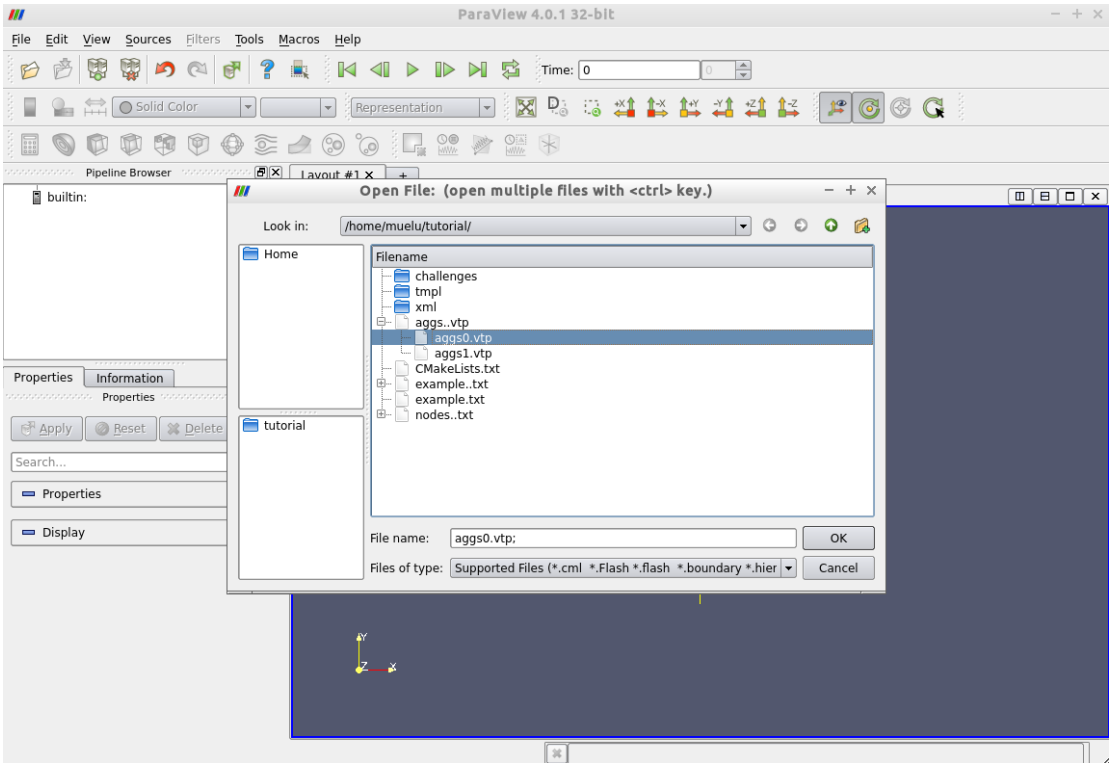
Run the `hands-on.py` script and select, e.g., the Laplace 2D example on a 50×50 mesh. Select above xml file for the multigrid parameters with the `aggregation: export visualization data` enabled. Run the program and then choose option 8 for post-processing the aggregates.

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

Solver xml parameters:              xml/n2_easy_agg.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? 8
POSTPROCESS AGGREGATION OUTPUT DATA
Level 0
process  aggs_level0_proc0.out
process  aggs_level0_proc1.out
node file nodes1.txt generated: OK
VTK Export for level 0 finished...

Level 1
process  aggs_level1_proc0.out
process  aggs_level1_proc1.out
node file nodes2.txt generated: OK
VTK Export for level 1 finished...

Use paraview to visualize generated vtk files for aggregates.
Press any key to continue...
```

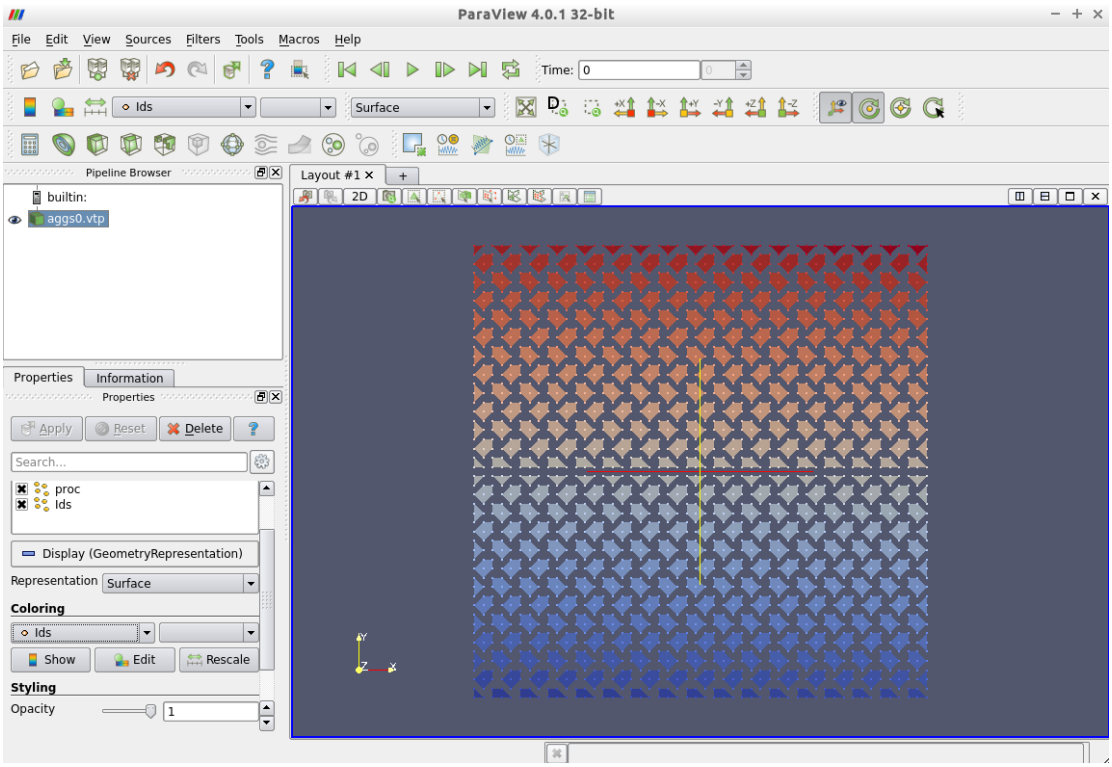
☛ Be aware that without `aggregation: export visualization data = true` the post processing step for the aggregates will fail.

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 4.1 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"

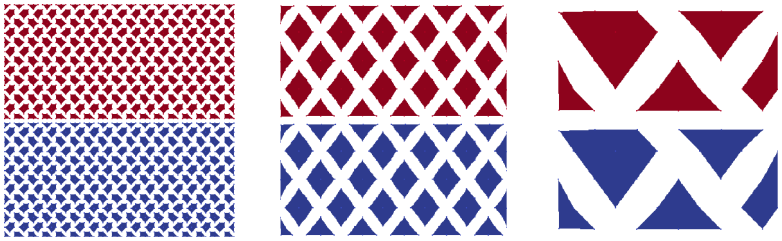


Figure 4.1: Aggregates for Laplace2D example on 50×50 mesh without dropping.

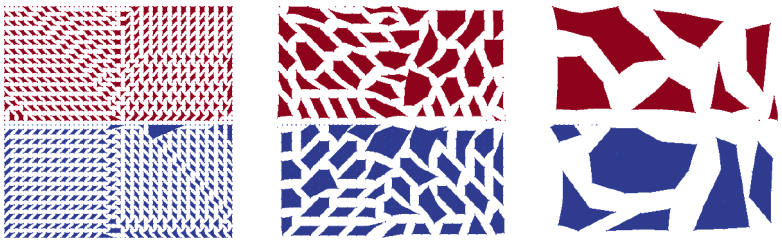


Figure 4.2: Aggregates for Recirc2D example on 50×50 mesh with dropping.

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.

Exercise 4.1 Repeat above steps for the `Recirc2D` example on a 50×50 mesh. Compare the aggregates from the `xml/n2_easy_agg.xml` parameter file with the aggregates when using the `xml/n2_easy_agg2.xml` parameter file, which drops some small entries of the fine level matrix A when building the graph. ■

Exercise 4.2 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? ■

Figure 4.2 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.

Note on coupled aggregation strategy:

Comparing Figures 4.1 and 4.3 one finds the difference between the `uncoupled` and the `coupled` aggregation method (`aggregation: type`). For the `coupled` aggregation strategy the aggregates can overlap processor boundaries.

☛ Using the `coupled` aggregation in general is not recommended, since

- the aggregation routine itself needs some global communication,
- building the tentative prolongation operator from the aggregates needs some global communication,
- prolongator smoothing is more expensive due to a higher overlap.

The implementation of a `coupled` aggregation method is much more complicated and therefore error-prone and less robust.

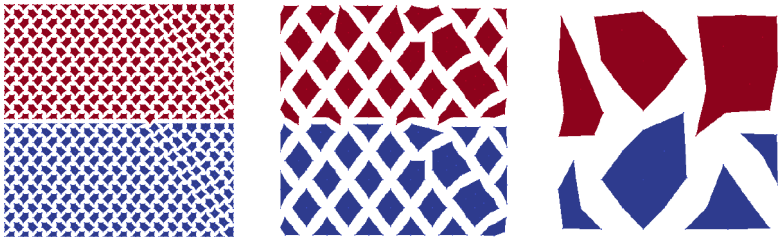


Figure 4.3: Aggregates for Laplace2D example on 50×50 mesh without dropping using a coupled aggregation strategy.

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 to export the corresponding operators to the matrix market format such that the files can be imported, e.g., into MATLAB (or FreeMat[1]) for some in-depth analysis.

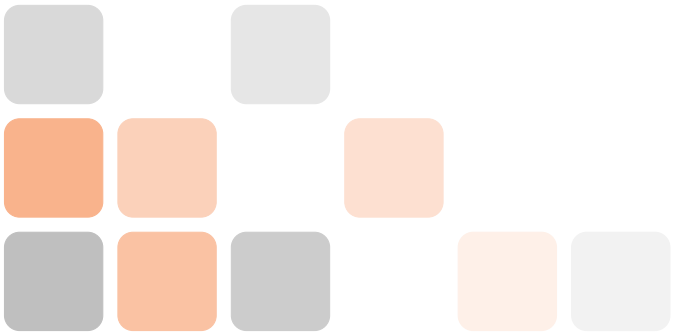
Using the following xml file writes the fine level operator and the coarse level operator as well as the prolongation and restriction operator to the hard disk using the filenames `A_0.m`, `A_1.m` as well as `P_1.m` and `R_1.m`

```
<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}"/>
  </ParameterList>
</ParameterList>
```

☛ 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,...).

[1] In the virtual image you find a FreeMat installation (`http://freemat.sourceforge.net/`).



5. Challenge: CD example

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, ...).

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.

```
Datei  Bearbeiten  Ansicht  Lesezeichen  Einstellungen  Hilfe
*************************   PROBLEM   *************************
Problem type:          condif2d
Problem size:          16641

Solver xml parameters:              condif2d_parameters.xml
Number of processors:               1
Solver (Tolerance): gmres (1e-12)
*************************   PROBLEM   *************************

Results up to date!

*************************   RESULTS   *************************
Reference settings:
               total iterations: 94
               Solution time: 0.852959 (sec.)
Your settings:
grep: output.log: No such file or directory
grep: output.log: No such file or directory
*************************   RESULTS   *************************

0. Run example
1. Show screen output
2. Change XML parameter file
3. Open xml file
4. Change procs
5. Change linear solver
6. Plot residual
7. Exit
your choice?
```

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.

☛ 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 5.1 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). ■

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.

5.3.1 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.

5.3.2 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.

5.3.3 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.

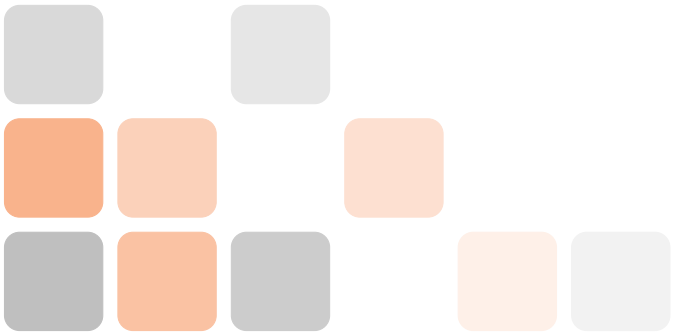
5.3.4 Level smoothers

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.

5.3.5 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.

Advanced topics



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 §1.1. That is, in the `hands-on.py` script you have to choose option 0 for the problem type.

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.

```
<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">
```

```
26      <Parameter name="Smoother"                    type="string" value="myJacobi"/>
27      <Parameter name="CoarseSolver"                type="string" value="myJacobi"/>
28    </ParameterList>
29
30  </ParameterList>
31 </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.

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

Exercise 6.1 Run the `hands-on.py` script and choose the XML file `xml/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 6.2 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? ■

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

Exercise 6.4 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. ■

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.

- 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" 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>
```

Beside above level smoothers there are more level smoothers such as ILU methods.

Exercise 6.5 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. ■

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

```
<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"/>
      </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="NoFactory"/>
      <Parameter name="CoarseSolver"      type="string" value="DirectSolver"/>

    </ParameterList>

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

  </ParameterList>
</ParameterList>
```

You can find the parameters in `xml/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

```
Number of levels  = 4
Operator complexity = 1.73

matrix rows   nnz  nnz/row procs
A 0   10000 49600    4.96 2
A 1    1700 35024   20.60 2
A 2      80  1328   16.60 2
A 3       4    16    4.00 2

Smoother (level 0) pre : IFPACK (Local Backward GS, sweeps=50, damping=0.6)
Smoother (level 0) post : no smoother

Smoother (level 1) pre : no smoother
Smoother (level 1) post : IFPACK (Local Backward GS, sweeps=50, damping=0.6)

Smoother (level 2) pre : no smoother
Smoother (level 2) post : IFPACK (Local Backward GS, sweeps=50, damping=0.6)

Smoother (level 3) pre : MueLu::AmesosSmoother{type = Superlu}
Smoother (level 3) post : no smoother
```

created with MueLu version ed3342a

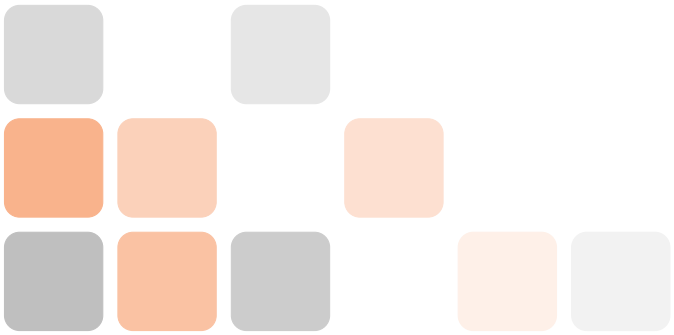
Exercise 6.6 Create an XML file in advanced format which produces the following multigrid layout

```
Number of levels  = 4
Operator complexity = 1.73

```

```
matrix rows   nnz  nnz/row procs
A 0   10000 49600    4.96 2
A 1    1700 35024   20.60 2
A 2      80  1328   16.60 2
A 3       4    16    4.00 2

Smoother (level 0) both : IFPACK (Local Jacobi, sweeps=5, damping=0.6)

Smoother (level 1) pre : IFPACK (Local GS, sweeps=3, damping=0.6)
Smoother (level 1) post : IFPACK (Local Backward GS, sweeps=3, damping=0.6)

Smoother (level 2) both : IFPACK (Local SGS, sweeps=3, damping=0.7)

Smoother (level 3) pre : MueLu::AmesosSmoother{type = Superlu}
Smoother (level 3) post : no smoother
```

created with MueLu version ed3342a .

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



7. MueLu factories for transfer operators

For this example we reuse the `Recirc2D` example as introduced in §3.1. 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.

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.

7.1.1 Unsmoothed transfer operators

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 7.1(a) 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 7.1(b) 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 for our non-symmetric problem they might be problematic.

7.1.2 Smoothed transfer operators for non-symmetric systems

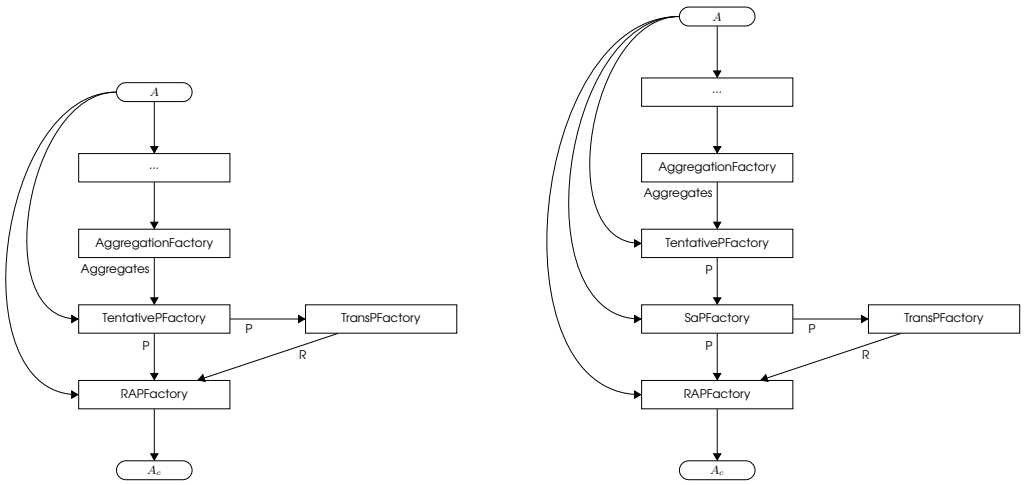
In case of non-symmetric linear systems it is $A \neq A^T$. Therefore it is a bad idea just to use the transposed of the smoothed prolongation operator for the restrictor. Let $\widehat{P}$ be the non-smoothed tentative prolongation operator. Then the smoothed prolongation operator P is built using

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

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



(a) Non-smoothed aggregation based AMG (b) Smoothed aggregation AMG (SA-AMG)

Figure 7.1: Simple factory design for aggregation based algebraic multigrid methods.

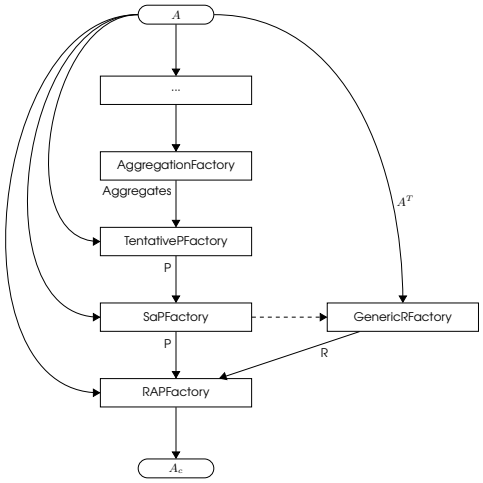


Figure 7.2: Simple factory design for smoothed aggregation based algebraic multigrid methods for non-symmetric systems.

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 7.2 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 [4]. Another approach based on SchurComplement approximations can be found in [5].

7.2 XML interface

7.2.1 Unsmoothed transfer operators

To construct a multigrid hierarchy with unsmoothed transfer operators one can use the following XML file (stored in `xml/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="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="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 `TentativePFactory` 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`.

☛ Be aware that it is highly important not to forget to register the `myTentativePFact` object for generating null space `Nullspace`. It is not visualized in the dependency trees, but the `TenativePFactory` factory both generates the unsmoothed prolongation operator and the set of near null space vectors for the coarse level. So, if you declare your own explicit instance of a

`TentativePFactory` you always have to register it for the near null space, too. In general it is a good idea to register a factory in the *Hierarchy* sublist for all output variables of the factory.

☛ Note that 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 7.1 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`. ■

7.2.2 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 `xml/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"/>
```

```
21      <Parameter name="sa: damping factor"            type="double" value="1.0"/>
22    </ParameterList>
23    <ParameterList name="myTentRestrictorFact">
24      <Parameter name="factory"                       type="string" value="
            TransPFactory"/>
25      <Parameter name="P"                             type="string" value="
            myTentativePFact"/>
26    </ParameterList>
27    <ParameterList name="mySymRestrictorFact">
28      <Parameter name="factory"                       type="string" value="
            TransPFactory"/>
29      <Parameter name="P"                             type="string" value="
            myProlongatorFact"/>
30    </ParameterList>
31    <ParameterList name="myNonsymRestrictorFact">
32      <Parameter name="factory"                       type="string" value="
            GenericRFactory"/>
33      <Parameter name="P"                             type="string" value="
            myProlongatorFact"/>
34    </ParameterList>
35
36
37    <ParameterList name="SymGaussSeidel">
38      <Parameter name="factory"                       type="string" value="
            TrilinosSmoother"/>
39      <Parameter name="type"                          type="string" value="RELAXATION"/>

40      <ParameterList name="ParameterList">
41        <Parameter name="relaxation: type"            type="string" value="Symmetric
              Gauss-Seidel"/>
42        <Parameter name="relaxation: sweeps"          type="int"   value="10"/>
43        <Parameter name="relaxation: damping factor"  type="double" value="0.8"/>
44      </ParameterList>
45    </ParameterList>
46
47  </ParameterList>
48
49  <!-- Definition of the multigrid preconditioner -->
50  <ParameterList name="Hierarchy">
51
52    <Parameter name="max levels"                      type="int"    value="10"/>
53    <Parameter name="coarse: max size"                type="int"    value="10"/>
54    <Parameter name="verbosity"                       type="string" value="High"/>
55
56    <ParameterList name="All">
57      <Parameter name="Smoother"              type="string" value="SymGaussSeidel"/>
58      <Parameter name="Aggregates"            type="string" value="
            UncoupledAggregationFact"/>
59      <Parameter name="Nullspace"             type="string" value="myTentativePFact"/>
60      <Parameter name="P"                     type="string" value="myProlongatorFact
            "/>
61      <Parameter name="R"                     type="string" value="mySymRestrictorFact
            "/>
62      <Parameter name="CoarseSolver"          type="string" value="DirectSolver"/>
63    </ParameterList>
64
65  </ParameterList>
66 </ParameterList>
```

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

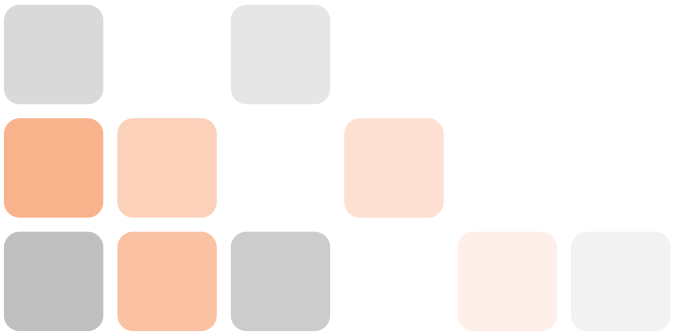
myTentRestrictorFact: just uses the transposed of the unsmoothed prolongator for restriction.
mySymRestrictorFact: uses the transposed of the smoothed prolongator for restriction.
myNonsymRestrictorFact: uses the special non-symmetric smoothing for the restriction operator (based on the `SaPFactory`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

☛ 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 7.2 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 7.3 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 7.4 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). ■



8. Rebalancing - Hypergraph repartitioning

8.1 Basic concepts and parameters

It's a natural thing to use lesser processors for the coarse level problem than for the fine level problem. This is an elegant way to avoid the problems of the coupled aggregation strategy, by just using the uncoupled aggregation strategy with rebalancing on the coarser levels.

In this tutorial we use a hypergraph based repartitioning for the coarse level matrices A_c to rebalance the problem. The repartitioning algorithm is implemented in the Zoltan 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.

☛ 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:

`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, than the problem will be rebalanced.

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

8.2 Transfer operator design

Figure 8.1 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 `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[1]. 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` is the most important factory and responsible for the repartitioning. Therein the decision is made whether repartitioning is necessary at all. It creates the communication "plan" that is used to rebalance the transfer operators and the coarse level matrix.

8.3 XML interface

The corresponding XML parameter file looks as

```
<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>
```

[1] Isorropia is a Trilinos package which provides an easy-to-use interface to many partitioning algorithms in Zoltan.

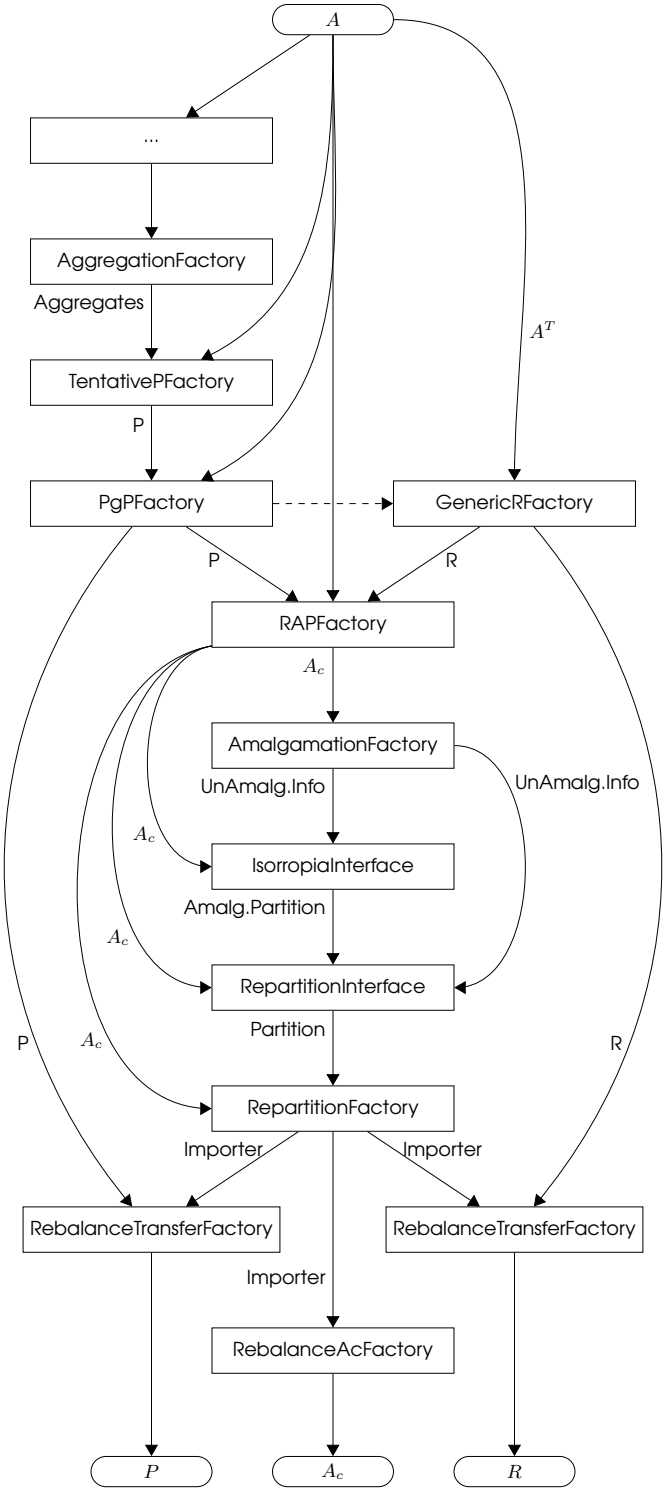


Figure 8.1: Factory design for smoothed aggregation based algebraic multigrid methods for non-symmetric systems with rebalancing.

```
31
32    <!-- ======================= REPARTITIONING ======================= -->
33    <!-- amalgamation of coarse level matrix -->
34    <ParameterList name="myRebAmalgFact">
35      <Parameter name="factory"                   type="string" value="
          AmalgamationFactory"/>
36      <Parameter name="A"                         type="string" value="myRAPFact"/>
37    </ParameterList>
38
39    <ParameterList name="myIsorropiaInterface">
40      <Parameter name="factory"                   type="string" value="
          IsorropiaInterface"/>
41      <Parameter name="A"                         type="string" value="myRAPFact"/>
42      <Parameter name="UnAmalgamationInfo"        type="string" value="myRebAmalgFact"/>
43    </ParameterList>
44
45    <ParameterList name="myRepartitionInterface">
46      <Parameter name="factory"                   type="string" value="
          RepartitionInterface"/>
47      <Parameter name="A"                         type="string" value="myRAPFact"/>
48      <Parameter name="AmalgamatedPartition"      type="string" value="
          myIsorropiaInterface"/>
49    </ParameterList>
50
51
52    <ParameterList name="myRepartitionFact">
53      <Parameter name="factory"                   type="string" value="
          RepartitionFactory"/>
54      <Parameter name="A"                         type="string" value="myRAPFact"/>
55      <Parameter name="Partition"                 type="string" value="
          myRepartitionInterface"/>
56      <Parameter name="repartition: min rows per proc" type="int" value="2000"/>
57      <Parameter name="repartition: max imbalance" type="double" value="1.1"/>
58      <Parameter name="repartition: start level"  type="int"   value="1"/>
59      <Parameter name="repartition: remap parts"  type="bool"  value="false"/>
60    </ParameterList>
61
62    <ParameterList name="myRebalanceProlongatorFact">
63      <Parameter name="factory"                   type="string" value="
          RebalanceTransferFactory"/>
64      <Parameter name="type"                      type="string" value="Interpolation"/>
65      <Parameter name="P"                         type="string" value="myProlongatorFact
          "/>
66      <Parameter name="Nullspace"                 type="string" value="myTentativePFact
          "/>
67    </ParameterList>
68
69    <ParameterList name="myRebalanceRestrictionFact">
70      <Parameter name="factory"                   type="string" value="
          RebalanceTransferFactory"/>
71      <Parameter name="type"                      type="string" value="Restriction"/>
72      <Parameter name="R"                         type="string" value="myRestrictorFact
          "/>
73    </ParameterList>
74
75    <ParameterList name="myRebalanceAFact">
76      <Parameter name="factory"                   type="string" value="
          RebalanceAcFactory"/>
77      <Parameter name="A"                         type="string" value="myRAPFact"/>
78    </ParameterList>
79
```

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

  </ParameterList>
</ParameterList>
```

It is stored in `xml/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 8.1. Be aware that the *Nullspace* variable now is also generated by the `myRebalanceProlongatorFact`.

Exercise 8.1 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 `xml/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.

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

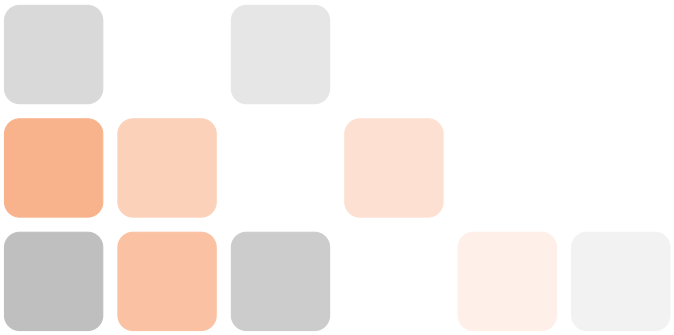
■

You should observe a multigrid hierarchy as follows

```
Number of levels  = 4
```

```
Operator complexity = 2.02

matrix rows    nnz  nnz/row procs
A 0   90000 448800    4.99 4
A 1   15296 447206   29.24 4
A 2     466   8525   18.29 1
A 3      24    178    7.42 1

Smoother (level 0) both : IFPACK (Local SGS, sweeps=20, damping=1.2)

Smoother (level 1) both : IFPACK (Local SGS, sweeps=20, damping=1.2)

Smoother (level 2) both : IFPACK (Local SGS, sweeps=20, damping=1.2)

Smoother (level 3) pre : MueLu::AmesosSmoother{type = Superlu}
Smoother (level 3) post : no smoother
```

created with MueLu version ed3342a



9. 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 [6, Chapter 3.3].

9.1 Building aggregates

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 9.1 shows the extended transfer operator design with the additional `CoalesceDropFactory`.

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.

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.

```
1  <ParameterList name="MueLu">
2
3    <!-- Factory collection -->
4    <ParameterList name="Factories">
5
6      <!-- Note that ParameterLists must be defined prior to being used -->
7      <ParameterList name="myCoalesceDropFact">
8        <Parameter name="factory"                          type="string" value="
             CoalesceDropFactory"/>
9        <Parameter name="lightweight wrap"                 type="bool"  value="true"/>
10       <!-- for aggregation dropping -->
11       <Parameter name="aggregation: drop tol"            type="double" value="0.01"/>
12     </ParameterList>
13
14     <ParameterList name="UncoupledAggregationFact">
15       <Parameter name="factory"                          type="string" value="
             UncoupledAggregationFactory"/>
```

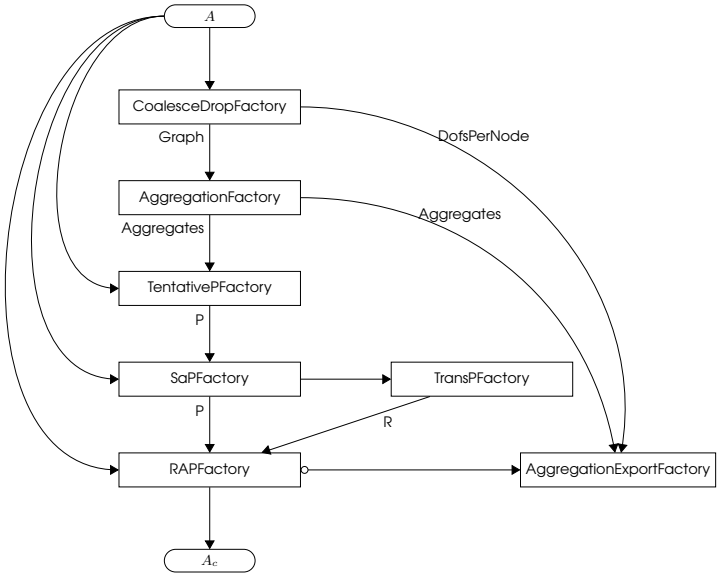


Figure 9.1: Simple factory design for building aggregates.

```
16      <Parameter name="aggregation: ordering"          type="string" value="natural"/>
17      <Parameter name="aggregation: max selected neighbors" type="int" value="0"/>
18      <Parameter name="aggregation: min agg size"      type="int"   value="4"/>
19      <Parameter name="Graph"                          type="string" value="
            myCoalesceDropFact"/>
20      <Parameter name="DofsPerNode"                    type="string" value="
            myCoalesceDropFact"/>
21    </ParameterList>
22
23    <ParameterList name="myTentativePFact">
24      <Parameter name="factory"                        type="string" value="
            TentativePFactory"/>
25    </ParameterList>
26    <ParameterList name="myProlongatorFact">
27      <Parameter name="factory"                        type="string" value="PgPFactory"/>

28      <Parameter name="P"                              type="string" value="
            myTentativePFact"/>
29    </ParameterList>
30    <ParameterList name="myRestrictorFact">
31      <Parameter name="factory"                        type="string" value="
            GenericRFactory"/>
32      <Parameter name="P"                              type="string" value="
            myProlongatorFact"/>
33    </ParameterList>
34
35    <ParameterList name="myAggExportFact">
36      <Parameter name="factory"                        type="string" value="
            AggregationExportFactory"/>
37      <Parameter name="Output filename"                type="string" value="aggs_level%
            LEVELID_proc%PROCID.out"/>
38      <Parameter name="DofsPerNode"                    type="string" value="
            myCoalesceDropFact"/>
39    </ParameterList>
40
41    <ParameterList name="myRAPFact">
42      <Parameter name="factory"                        type="string" value="RAPFactory"/>
```

```
43      <Parameter name="P"                                type="string" value="
            myProlongatorFact"/>
44      <Parameter name="R"                                type="string" value="
            myRestrictorFact"/>
45      <ParameterList name="TransferFactories">
46        <Parameter name="Visualization"                  type="string" value="
              myAggExportFact"/>
47      </ParameterList>
48    </ParameterList>
49
50    <!-- ======================= SMOOTHERS ======================= -->
51    <ParameterList name="SymGaussSeidel">
52      <Parameter name="factory"                          type="string" value="
            TrilinosSmoother"/>
53      <Parameter name="type"                             type="string" value="RELAXATION"/>

54      <ParameterList name="ParameterList">
55        <Parameter name="relaxation: type"               type="string" value="Symmetric
              Gauss-Seidel"/>
56        <Parameter name="relaxation: sweeps"             type="int"   value="1"/>
57        <Parameter name="relaxation: damping factor"     type="double" value="1.0"/>
58      </ParameterList>
59    </ParameterList>
60
61  </ParameterList>
62
63  <!-- Definition of the multigrid preconditioner -->
64  <ParameterList name="Hierarchy">
65
66    <Parameter name="max levels"    type="int"   value="10"/>
67    <Parameter name="coarse: max size" type="int" value="10"/>
68    <Parameter name="verbosity"     type="string" value="High"/>
69
70    <ParameterList name="All">
71      <Parameter name="Smoother"    type="string" value="SymGaussSeidel"/>
72      <Parameter name="Graph"       type="string" value="myCoalesceDropFact"/>
73      <Parameter name="Aggregates"  type="string" value="UncoupledAggregationFact"/>
74      <Parameter name="Nullspace"   type="string" value="myTentativePFact"/>
75      <Parameter name="P"           type="string" value="myProlongatorFact"/>
76      <Parameter name="R"           type="string" value="myRestrictorFact"/>
77      <Parameter name="A"           type="string" value="myRAPFact"/>
78      <Parameter name="CoarseSolver" type="string" value="DirectSolver"/>
79    </ParameterList>
80  </ParameterList>
81</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 dependencies 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 `xml/s4a.xml`.

9.2 Visualization of aggregates

For debugging purposes it can be very useful to visualize the aggregates. In MueLu there is a helper factory named `AggregationExportFactory` which can be used to export the aggregates for visualization purposes.

The `AggregationExportFactory` acts as a small helper factory within the `RAPFactory` which writes out some aggregation information to files on the hard disk (see figure 9.1). In a post-processing step one can use these files together with some mesh information to generate pictures of the aggregates on each level. For post-processing we use the `MueLu_Agg2VTK.py` python script which is stored in the `src` folder of this tutorial. It works both for 2D and 3D problems. You have to provide the output files from the `AggregationExportFactory` as well as a list of node coordinates, that are needed to produce the output in the vtk format which can be opened in paraview.

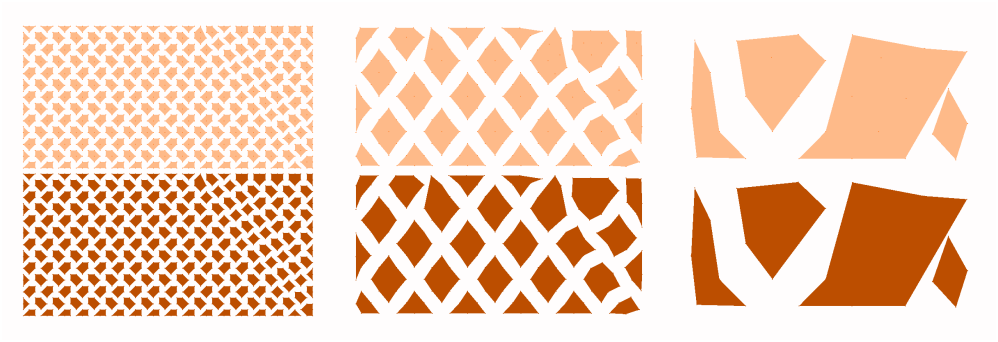
☛ The visualization script has successfully been tested with VTK 5.x. Note that it is not compatible to VTK 6.x.

Exercise 9.1 Run the `Laplace 2D` example on a 50×50 mesh using the XML file `xml/s4a.xml`. Visualize the aggregates. ■

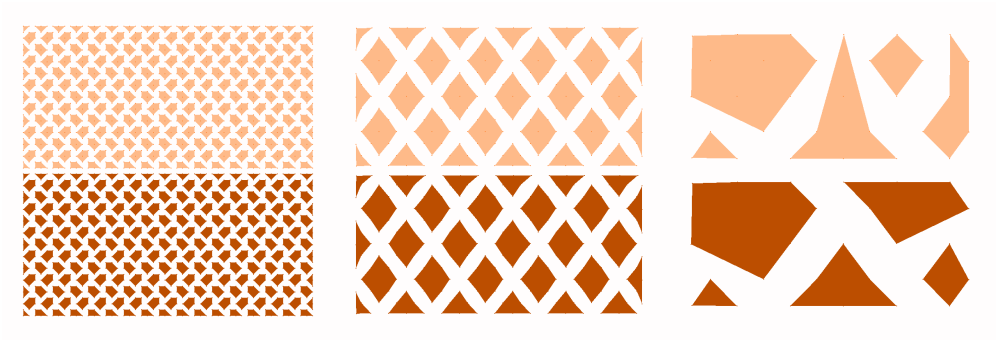
Exercise 9.2 In the `xml/s4a.xml` file an uncoupled aggregation factory has been explicitly defined using with some user-chosen aggregation parameters. Make a copy of `xml/s4a.xml` and use the default (uncoupled) aggregation routine that is provided by MueLu if no user-specified aggregation algorithm 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. ■

Before we close this tutorial there are some general remarks that might help in setting up valid XML files.

☛ 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 you `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 used for 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.

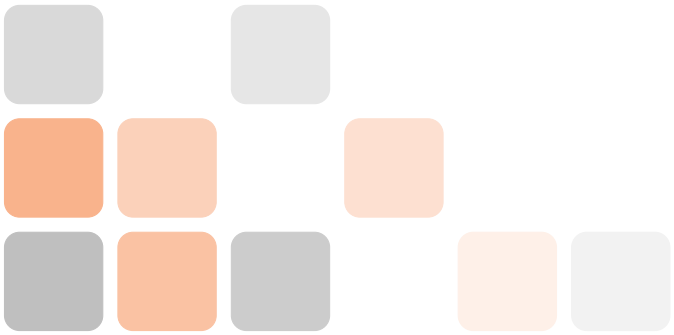


(a) Aggregates with settings from `xml/s4a.xml`



(b) Aggregates with settings from `xml/s4b.xml` (default parameters)

Figure 9.2: Different aggregation parameters and the corresponding aggregates



10. Useful commands and debugging

10.1 Export level information

Of course, it is possible to export the multigrid hierarchy in matrix market format similar to §4.2 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>
```

10.2 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.

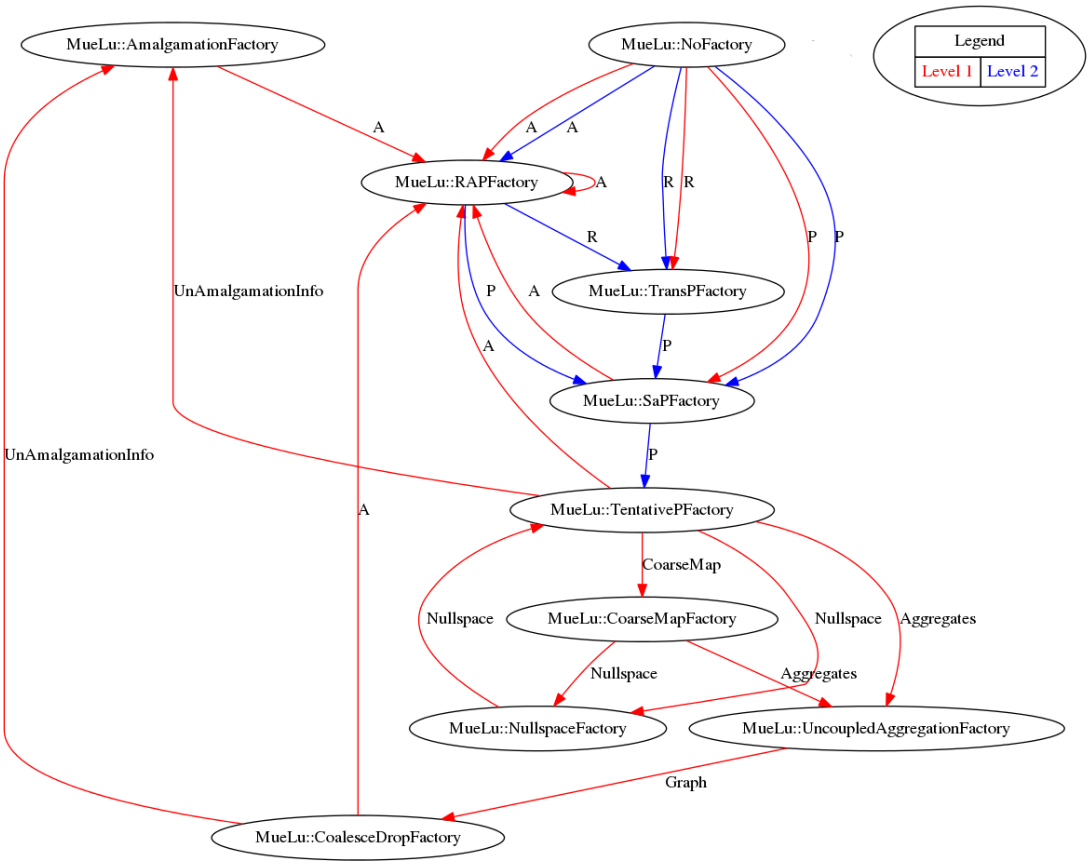
To write out the dependencies you just have to put in the `dependencyOutputLevel` parameter. The value gives you the fine level index that you are interested in (e.g., 1 means: print dependencies

between level 1 and level 2).

```
<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 following commands in your terminal to obtain the following graph:

```
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
```



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). Compare the factory layout in Figure 9.1 with above dependency graph. Try to read it from bottom to top.

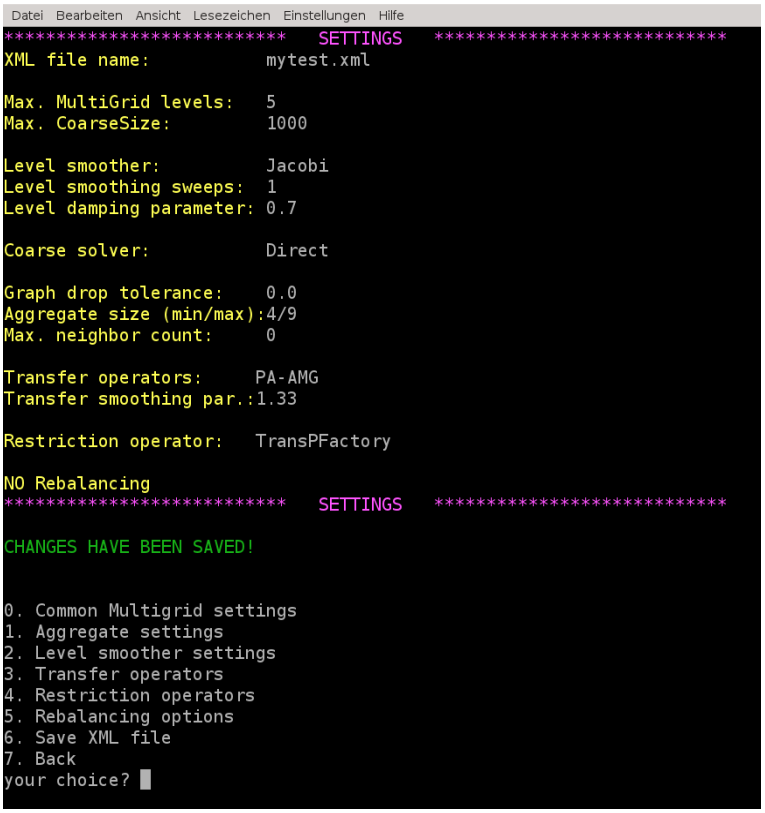
☛ In case that the file `dep_graph.dot` is not generated you have to check the prerequisites. To be able to auto-generate the dependency graphs you have to compile MueLu with Boost enabled. Furthermore you have to set the `MueLu_ENABLE_Boost_for_real:BOOL = ON` defines flag in your configuration script. 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 7.1. 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.

10.3 Graphical assistant for XML file generation

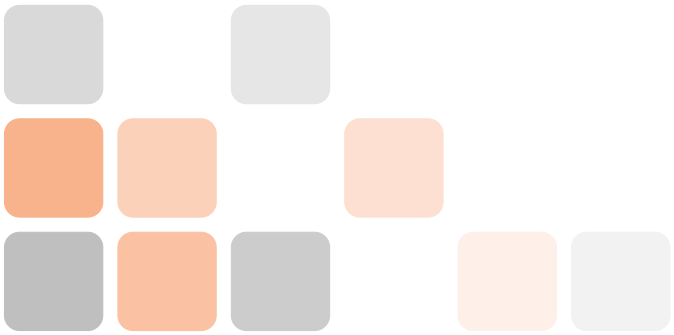
The `hands-on.py` driver script contains a graphical assistant to generate new XML parameter files in the advanced MueLu file format.

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.

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



11. Challenge: elasticity example

11.1 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, ...).

11.2 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 11.1 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.

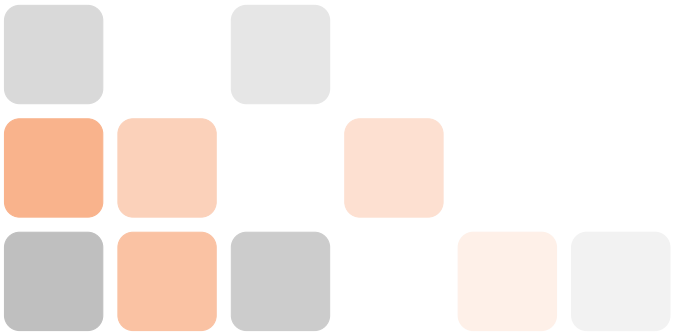
Exercise 11.2 Open the XML parameter file (choose option 3) and try to find optimized settings. Use the advanced XML file format. Save the file, rerun the example (option 0) and compare the output with the reference results. ■

☛ Use §5.3 for a general step-by-step procedure to optimize the multigrid parameters.

Exercise 11.3 How do the reference settings and your XML parameter settings perform when increasing the number of processors? ■

Exercise 11.4 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? ■

Expert tutorials



12. Using MueLu in user applications

This tutorial demonstrates how to use MueLu from within user applications in C++. In [2, Section 2.6] it is explained how to use MueLu through the `MueLu::CreateE/TpetraPreconditioner` interface. This interface is designed for beginners which want to try MueLu through standard Trilinos interfaces. Note that there is also support for Stratimikos. Please refer to the `examples` in the MueLu folder for more details.

This tutorial aims at more advanced methods to use MueLu, creating an explicit instance of some MueLu classes like `MueLu::Hierarchy` and `MueLu::HierarchyManager`. In the next sections we give some code snippets. Most of them are borrowed form the `laplace2d.cpp` file in the tutorial.

12.1 Preparations

First of all, we have to define a communicator object.

```
1
2  Teuchos::GlobalMPISession mpiSession(&argc, &argv, NULL);
3
4  bool success = false;
5  try {
6    RCP< const Teuchos::Comm<int> > comm = Teuchos::DefaultComm<int>::getComm();
7    int MyPID  = comm->getRank();
8    int NumProc = comm->getSize();
9
10   const Teuchos::RCP<Epetra_Comm> epComm = Teuchos::rcp_const_cast<Epetra_Comm>(Xpetra
         ::toEpetra(comm));
```

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 1.1). In this example we use Epetra for the underlying linear algebra framework, but it shall be mentioned that it works for Tpetra in a similar way (refer to the code examples in the MueLu examples folder).

```
1
2      // create map
3      epMap = Teuchos::rcp(Galeri::CreateMap("Cartesian2D", *epComm, GaleriList));
4
5      // create coordinates
6      epCoord = Teuchos::rcp(Galeri::CreateCartesianCoordinates("2D", epMap.get(),
           GaleriList));
7
```

```
8       // create matrix
9       epA = Teuchos::rcp(Galeri::CreateCrsMatrix("Laplace2D", epMap.get(), GaleriList));
10
11      double hx = 1./(nx-1); double hy = 1./(ny-1);
12      epA->Scale(1./(hx*hy));
```

MueLu is based on Xpetra which provides a common interface both for Epetra and Tpetra. Therefore we have to encapsulate our Epetra objects into Xpetra wrapper objects. This is done using the following code.

```
1
2   // Epetra -> Xpetra
3   Teuchos::RCP<CrsMatrix> exA = Teuchos::rcp(new Xpetra::EpetraCrsMatrix(epA));
4   Teuchos::RCP<CrsMatrixWrap> exAWrap = Teuchos::rcp(new CrsMatrixWrap(exA));
5
6   RCP<Matrix> A = Teuchos::rcp_dynamic_cast<Matrix>(exAWrap);
7   A->SetFixedBlockSize(1);
```

Note that the MueLu setup routines require a `Xpetra::Matrix` object. The wrapper class `Xpetra::CrsMatrixWrap` is just a wrapper derived from `Xpetra::Matrix` which manages a `Xpetra::CrsMatrix` object which is the common base class for both Epetra and Tpetra CRS matrix classes. The details are not really important as long as one understands that one needs a `Xpetra::Matrix` object for MueLu in the end. With the `SetFixedBlockSize` routine we state that there is only one degree of freedom per node (pure Laplace 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.

```
1
2   // build null space vector
3   RCP<const Map> map = A->getRowMap();
4   RCP<MultiVector> nullspace = MultiVectorFactory::Build(map, 1);
5   nullspace->putScalar(one);
```

12.2 Setup phase

With a fine level operator A available as `Xpetra::Matrix` object and a set of near null space vectors (available as `Xpetra::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.

12.2.1 XML Interface

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.

Assuming that we have a `Teuchos::ParameterList` object with valid MueLu parameters we can create a `MueLu::HierarchyManager` object

```
1
2       RCP<HierarchyManager> mueLuFactory = rcp(new ParameterListInterpreter(mueluList));
```

For an example how to fill the parameter list the reader may refer to [2, Section 2.3]. Note that there are routines to fill the parameter lists with the information from XML files. You can

also directly provide a file name of a XML file to the `Muelu::ParameterListInterpreter`. For details you may refer to the doxygen documentation or the example in `laplace2d.cpp`.

Next a new `MueLu::Hierarchy` object is generated

```
1
2     RCP<Hierarchy> H;
```

The `CreateHierarchy` creates a new empty multigrid hierarchy with a finest level only. The user has to feed in the linear operator A and the near null space vector. If further information is available, such as the node coordinates, they can be also stored in the finest level. The coordinates are needed, e.g., for rebalancing the coarse levels. Finally, the `SetupHierarchy` call initiates the coarsening process and the multigrid hierarchy is built according to the parameters from the `mueluList` parameters.

```
1
2       H = mueLuFactory->CreateHierarchy();
3       H->GetLevel(0)->Set("A",        A);
4       H->GetLevel(0)->Set("Nullspace", nullspace);
5       if (!coordinates.is_null())
6         H->GetLevel(0)->Set("Coordinates", coordinates);
7       mueLuFactory->SetupHierarchy(*H);
```

As XML parameter file any of the files shown in the previous tutorials can be used.

☛ As one can see from the last code snippet, the `Hierarchy` allows access to all important parts of the multigrid method before setup. So, if you have to feed in some non-standard information, this is the way how it works. Using the `CreateE/TpetraPreconditioner` interface may be easier but does not allow to access the finest level before setup.

Once the `SetupHierarchy` call is completed, the multigrid hierarchy is ready to use. The reader can skip the next section about the C++ interface and proceed with §12.3.3 for an example how to use the multigrid method as preconditioner within a Krylov subspace method from the Belos package.

12.2.2 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).

```
1  // create new hierarchy
2  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.

```
1    // instantiate new Hierarchy object
2    H = rcp(new Hierarchy());
3    H->setDefaultVerbLevel(Teuchos::VERB_HIGH);
4    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);
Finest->Set("Coordinates", coordinates); //FIXME: XCoordinates, YCoordinates, ..
```

☛ 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 (usually `int`) 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.

```
// define a factory manager
FactoryManager M;
```

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->SetDampingFactor(optSaDamping);

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

RCP<RAPFactory> AFact = rcp(new RAPFactory());
AFact->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", AFact);
```

☛ 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, that 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);
```

12.3 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.

```

  // set rhs and solution vector
  RCP<Epetra_Vector> B = Teuchos::rcp(new Epetra_Vector(*epMap));
  RCP<Epetra_Vector> X = Teuchos::rcp(new Epetra_Vector(*epMap));
  B->PutScalar(1.0);
  X->PutScalar(0.0);

  // Epetra -> Xpetra
  RCP<Vector> xB = Teuchos::rcp(new Xpetra::EpetraVector(B));
  RCP<Vector> xX = Teuchos::rcp(new Xpetra::EpetraVector(X));

  xX->setSeed(100);
  xX->randomize();
```

In this example we just create Epetra vectors and wrap them into Xpetra objects. The right hand side vector is initialized with one and the solution vector is filled with random values.

12.3.1 MueLu as multigrid solver

To use MueLu as standalone solver one can use the following code

```

  // use multigrid hierarchy as solver
```

```
3     RCP<Vector> mgridLsgVec = VectorFactory::Build(map);
4     mgridLsgVec->putScalar(0.0);
5     {
6       fancyout << "=======================================================\nUse
            multigrid hierarchy as solver." << std::endl;
7       tm = rcp (new TimeMonitor(*TimeMonitor::getNewTimer("ScalingTest: 5 - Multigrid
            Solve")));
8       mgridLsgVec->update(1.0,*xX,1.0);
9       H->IsPreconditioner(false);
10      H->Iterate(*xB, *mgridLsgVec, mgridSweeps);
11      comm->barrier();
12      tm = Teuchos::null;
13    }
```

In this code snippet a solution vector is created using the `Xpetra::VectorFactory` and initialized with the content from the solution vector `xX` containing the initial guess. Then, the `MueLu::Hierarchy` object is set to the non-preconditioner mode and the `Iterate` routine is called to perform `mgridSweeps` sweeps with the chosen multigrid cycle. If successful, the `mgridLsgVec` vector contains the solution.

12.3.2 MueLu as preconditioner for AztecOO

Commonly, multigrid methods are used as preconditioners for iterative linear solvers. Here, we show how to use the `MueLu::Hierarchy` as preconditioner within an AztecOO solver (using Epetra). After an Epetra solution vector has been created by

```
1     RCP<Epetra_Vector> precLsgVec = rcp(new Epetra_Vector(X->Map()));
```

the following code can be used to apply the MueLu hierarchy as preconditioner within the AztecOO CG solver

```
1     precLsgVec->PutScalar(0.0);
2     precLsgVec->Update(1.0,*X,1.0);
3     Epetra_LinearProblem epetraProblem(epA.get(), precLsgVec.get(), B.get());
4
5     AztecOO aztecSolver(epetraProblem);
6     aztecSolver.SetAztecOption(AZ_solver, AZ_cg);
7
8     MueLu::EpetraOperator aztecPrec(H);
9     aztecSolver.SetPrecOperator(&aztecPrec);
10
11    int maxIts = 50;
12
13    aztecSolver.Iterate(maxIts, tol);
```

Basically, the MueLu hierarchy is put into an `MueLu::EpetraOperator` object, which implements the Epetra interface for preconditioners. With the `SetPrecOperator` routine from the AztecOO solver the `MueLu::EpetraOperator` object then is defined as preconditioner.

12.3.3 MueLu as preconditioner for Belos

Belos is the successor package of AztecOO for linear solvers in Trilinos and works both for Epetra and Tpetra. Here we demonstrate how to use MueLu as preconditioner for Belos solvers using Xpetra. First, we have to declare objects for the solution vector and the right hand side vector in Xpetra. The following code just uses a random vector for the initial guess and solution variable.

```
1   // Define X, B
2   RCP<MultiVector> X = MultiVectorFactory::Build(map, 1);
3   RCP<MultiVector> B = MultiVectorFactory::Build(map, 1);
4
5   X->setSeed(846930886);
```

```
X->randomize();
A->apply(*X, *B, Teuchos::NO_TRANS, (SC)1.0, (SC)0.0);
B->norm2(norms);
B->scale(1.0/norms[0]);
```

In the following we demonstrate how to use the MueLu hierarchy as preconditioner within a Belos solver. There are special wrapper objects for wrapping the Xpetra matrix and the MueLu hierarchy to Belos compatible objects. These can be used to define a linear problem for use with Belos.

```
// Operator and Multivector type that will be used with Belos
typedef MultiVector       MV;
typedef Belos::OperatorT<MV> OP;
H->IsPreconditioner(true);

// Define Operator and Preconditioner
Teuchos::RCP<OP> belosOp = Teuchos::rcp(new Belos::XpetraOp<SC, LO, GO, NO>(A)); // Turns a Xpetra::Operator object into a Belos operator
Teuchos::RCP<OP> belosPrec = Teuchos::rcp(new Belos::MueLuOp<SC, LO, GO, NO>(H)); // Turns a MueLu::Hierarchy object into a Belos operator

// Construct a Belos LinearProblem object
RCP< Belos::LinearProblem<SC, MV, OP> > belosProblem = rcp(new Belos::LinearProblem<SC, MV, OP>(belosOp, X, B));
belosProblem->setLeftPrec(belosPrec);

bool set = belosProblem->setProblem();
if (set == false) {
  if (comm->getRank() == 0)
    std::cout << std::endl << "ERROR: Belos::LinearProblem failed to set up correctly!" << std::endl;
  return EXIT_FAILURE;
}
```

Then, one can set up the Belos solver. For a Belos GMRES solver one uses

```
// Belos parameter list
int maxIts = 100;
Teuchos::ParameterList belosList;
belosList.set("Maximum Iterations", maxIts); // Maximum number of iterations allowed
belosList.set("Convergence Tolerance", optTol); // Relative convergence tolerance requested
//belosList.set("Verbosity", Belos::Errors + Belos::Warnings + Belos::TimingDetails + Belos::StatusTestDetails);
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
RCP< Belos::SolverManager<SC, MV, OP> > solver = rcp(new Belos::BlockCGSolMgr<SC, MV, OP>(belosProblem, rcp(&belosList, false)));
```

Finally, we can solve the linear system using Belos with the MueLu multigrid preconditioner (left-preconditioning) by calling

```
    // solve linear system
    ret = solver->solve();
```

and perform some convergence checks

```
// Check convergence
if (ret != Belos::Converged) {
```

```
3     if (comm->getRank() == 0) std::cout << std::endl << "ERROR: Belos did not converge
          ! " << std::endl;
4   } else {
5     if (comm->getRank() == 0) std::cout << std::endl << "SUCCESS: Belos converged!" <<
          std::endl;
6   }
```

12.4 Full example

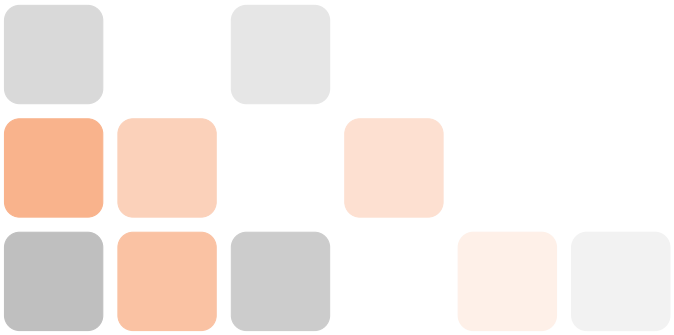
The reader may refer to `laplace2d.cpp` for a working example to study the source code. This demonstration program has some more features that are not discussed in this tutorial.

Exercise 12.1 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 12.2 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. ■



13. ML ParameterList interpreter

13.1 Backwards compatibility

ML [7] 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.

13.2 C++ part

13.2.1 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.

13.2.2 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. Please refer to the ML guide [7] for a complete list of available parameters.

```
1   params = rcp(new Teuchos::ParameterList());
2
3   params->set("ML output", 10);
4   params->set("max levels", 2);
5   params->set("smoother: type", "symmetric Gauss-Seidel");
6
7   if (xpetraParameters.GetLib() == Xpetra::UseTpetra)
8     params->set("coarse: type","Amesos-Superlu");
9   else
```

```
10      params->set("coarse: type","Amesos-KLU");
```

☛ Be aware that the `MLParameterListInterpreter` does not support all ML parameters but only the most important ones (e.g., smoothers, transfer operators, rebalancing, ...). There is, e.g., no support for the Maxwell specific enhancements in ML.

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

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

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

```
1    MLParameterListInterpreter mueLuFactory(*params);
2    RCP<Hierarchy> H = mueLuFactory.CreateHierarchy();
```

Of course, we 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).

```
1    RCP<MultiVector> nullspace = MultiVectorFactory::Build(A->getDomainMap(), numPDEs);
2
3    for (int i=0; i<numPDEs; ++i) {
4      Teuchos::ArrayRCP<Scalar> nsValues = nullspace->getDataNonConst(i);
5      int numBlocks = nsValues.size() / numPDEs;
6      for (int j=0; j< numBlocks; ++j) {
7        nsValues[j*numPDEs + i] = 1.0;
8      }
9    }
```

Then we just feed in the information to the finest level

```
1    H->GetLevel(0)->Set("Nullspace", nullspace);
2    H->GetLevel(0)->Set("A", A);
```

Finally we call the `Setup` routine which actually builds the multigrid hierarchy.

```
1    mueLuFactory.SetupHierarchy(*H);
```

Once we have the multigrid hierarchy set up we can use it the same way as described in §12.3.

Exercise 13.1 Study the source code of `../src/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 using

```
./MueLu_tutorial_MLParameterList.exe --ml=1 --muelu=0
   --xml=xml/ml_ParameterList.xml --linAlgebra=Epetra
```

and study the ML output. Compare the output and results when switching to MueLu using the same input file

```
./MueLu_tutorial_MLParameterList.exe --ml=0 --muelu=1
   --xml=xml/ml_ParameterList.xml --linAlgebra=Epetra
```

■

Exercise 13.2 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. Try to run both ML and MueLu and compare the results. Do you find significant differences? ■

Appendix

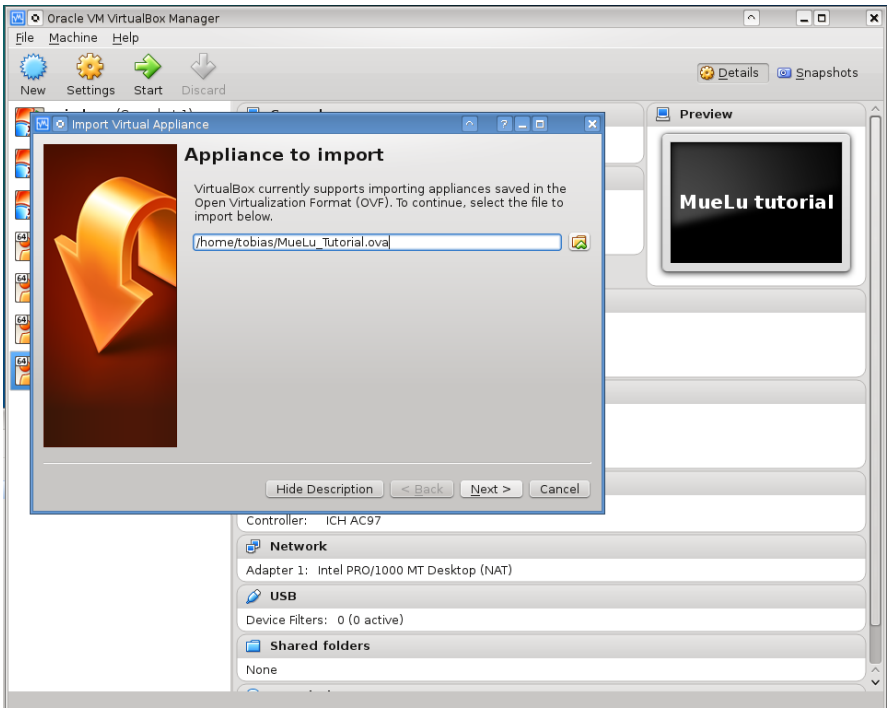
A. Virtual box image

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.

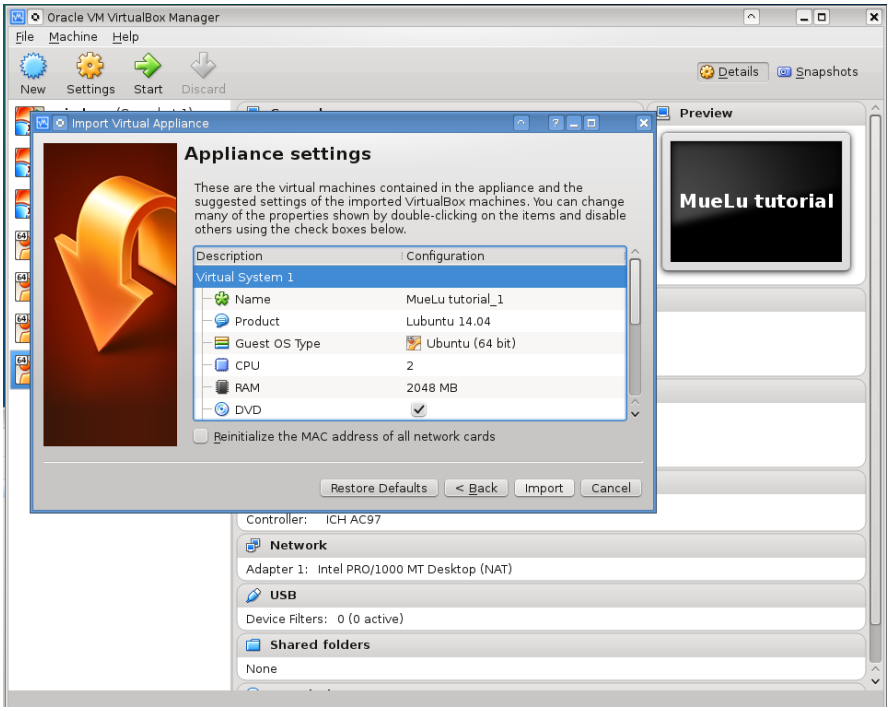
A.1 Preparations

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.



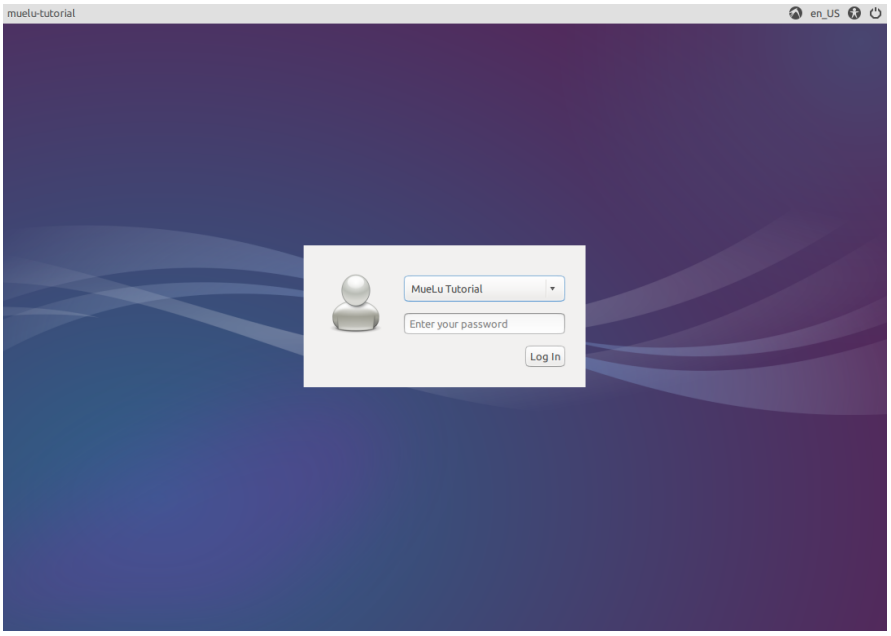
In general, one processor should be enough. But if you want to make some more reasonable tests with parallel multigrid you should increase the number of processors accordingly. Click import, to import the virtual machine.

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

A.2 First steps

A.2.1 Login and setup

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



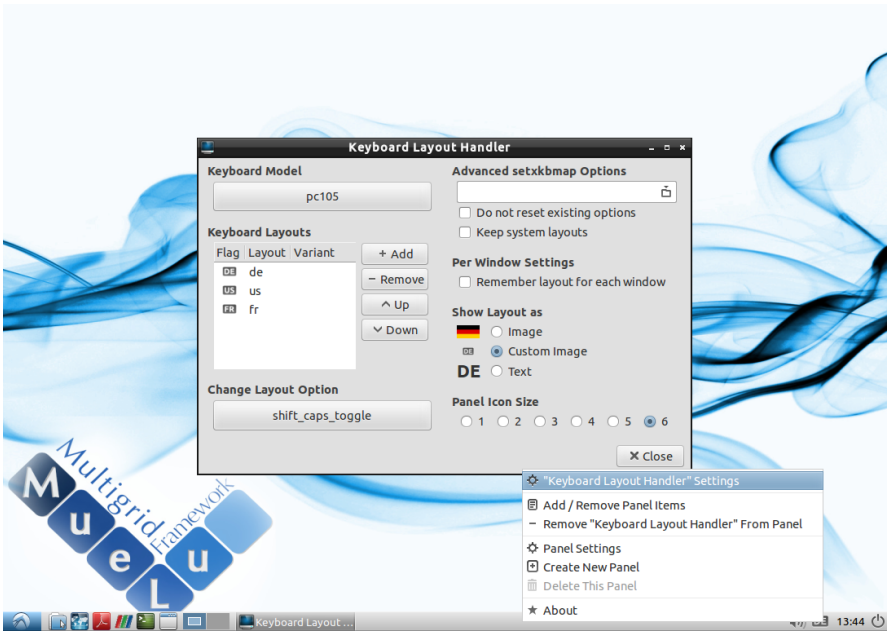
The login data is:

Username: muelu
Password: sandia

☛ 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

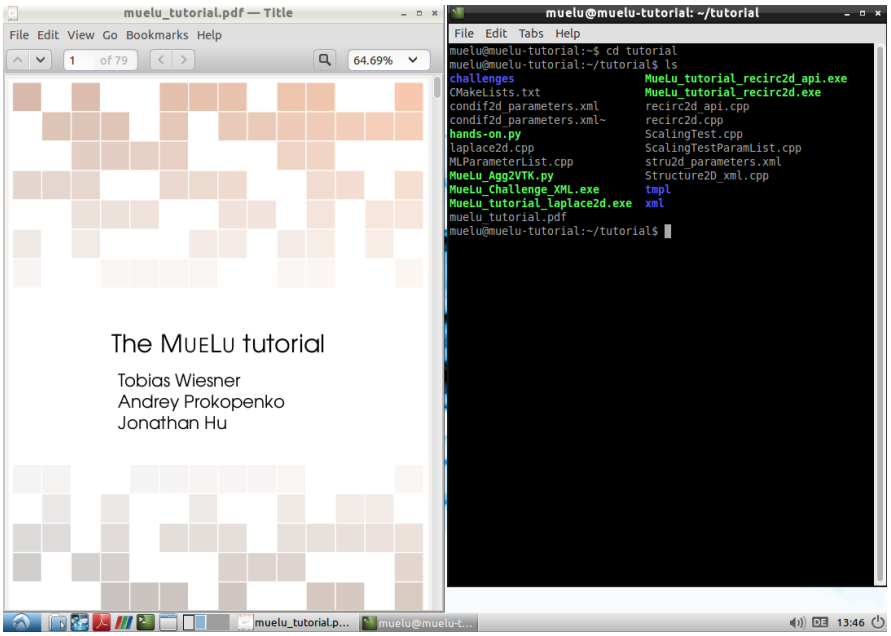


Then you are ready to go with the tutorial.

A.2.2 MueLu tutorial

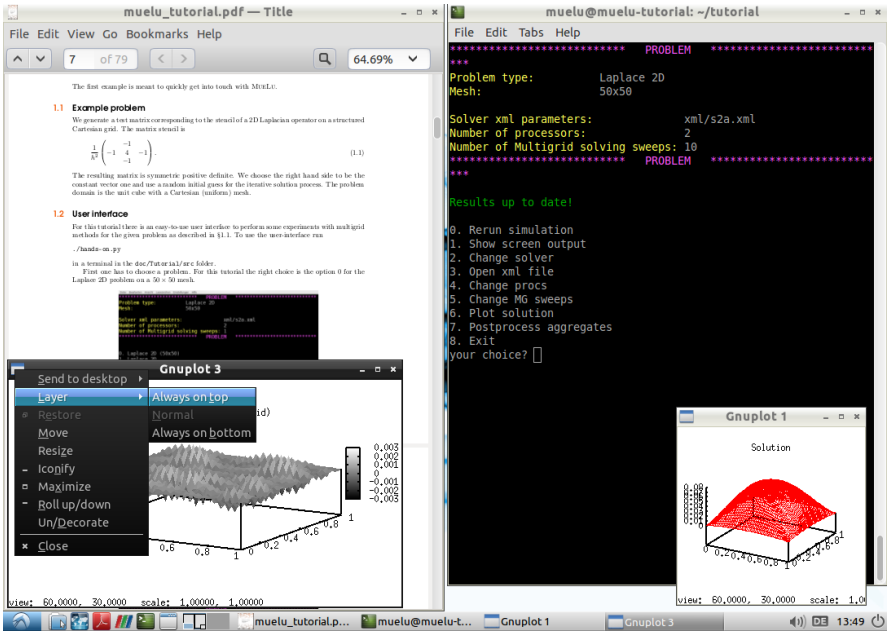
Open the tutorial with `evince` as pdf viewer. To open `evince` you can either use the shortcut in the lower left corner of your desktop or press Alt + F2 to open the `Run` dialog and enter `evince`. Load the `muelu_tutorial.pdf` file in the `tutorial` folder of your home directory.

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 + 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.

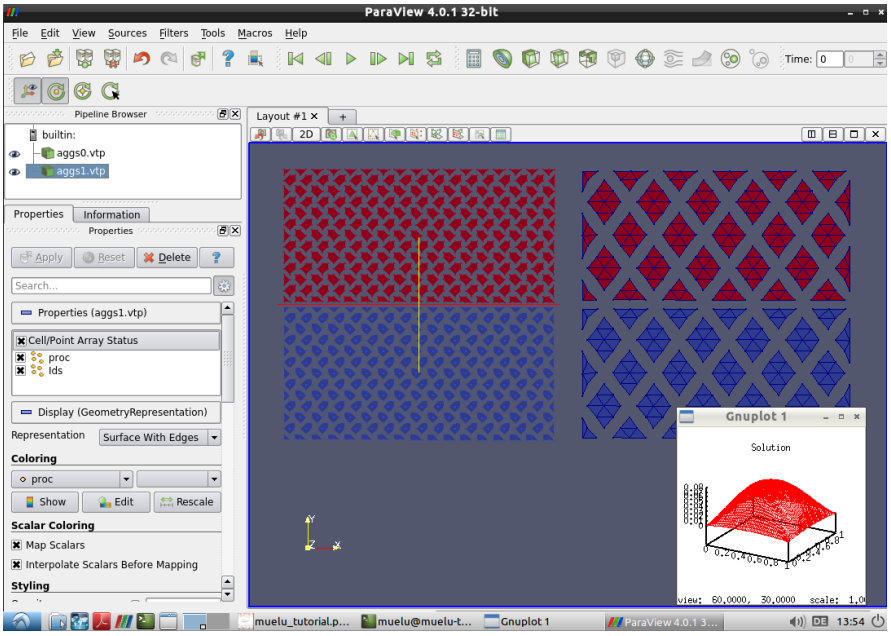


☛ Use the Win + ← and Win + → keys to arrange the windows in a split view as shown above. There are other useful keyboard shortcuts such as Win + R to open the `Run` dialog or Win + E to open the file manager.

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`)



A.3 Software

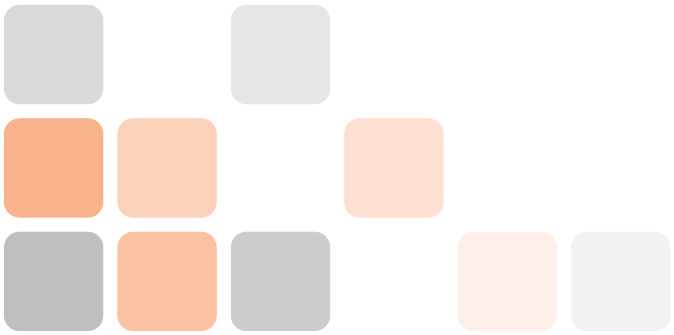
The virtual machine is based on a minimal installation of **Lubuntu 14.04**. The image file has 4 GB with about 250 MB free for the user.

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



B. Error messages

B.1 Syntax errors

B.1.1 Parser errors

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

Forgot to close the `</ParameterList>` section that is opened in line 1 of the xml file.

```
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.

B.1.2 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.

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"/>
```

B.2 MueLu errors

B.2.1 General errors

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

Smoother for Epetra 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`).

```
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 (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 smallter than the `aggregation: max agg size` parameter for the aggregation routine.

B.2.2 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 usually one 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=high` 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 §10.2.

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>
```

Looking at the error output it seems to be a problem with aggregates. However, in the XML file no special aggregation factory has been declared. 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.

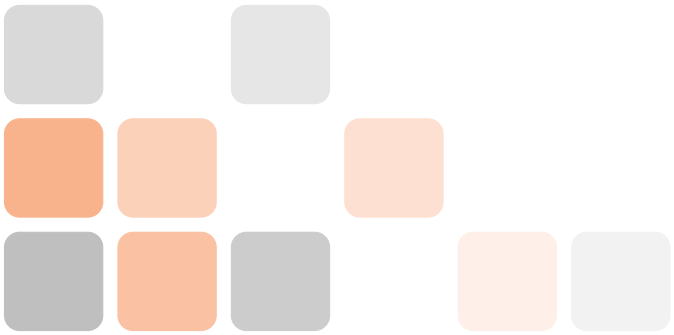
When looking at the XML file one can see that the `myTentativePFact` factory has been registered to be responsible for generating the prolongator P, but the generating factory for the variable `Nullspace` is not declared. MueLu tries to generate the default null space, but since it does not know about `myTentativePFact` to be a `TentativePFactory` which would already produce the needed information the calling 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.

☛ 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.



Bibliography

[1] W.L. Briggs, S.F. McCormick and others, A multigrid tutorial, SIAM, 2000.

[2] A. Prokopenko, J.J. Hu, T.A. Wiesner, C.M. Siefert and R.S. Tuminaro *MueLu User's Guide 1.0 (Trilinos Version 11.12)*, SAND2014-18874, 2014

[3] Vanek, P. and Mandel, J. and Brezina, M. Algebraic Multigrid by Smoothed Aggregation for Second and Fourth Order Elliptic Problems, Computing, 1996, 56, p. 179–196

[4] 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

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

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

[7] 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