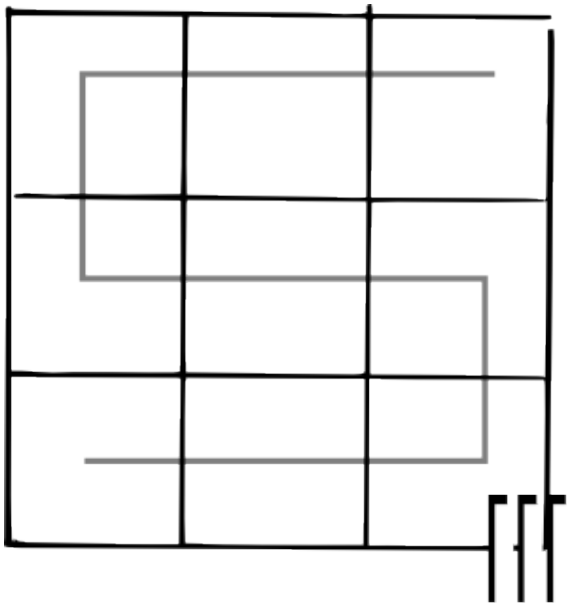


Peano Cookbook

www.peano-framework.org

Dr. rer. nat. Tobias Weinzierl

July 23, 2016

1 Preamble

Peano is an open source C++ solver framework. It is based upon the fact that spacetrees, a generalisation of the classical octree concept, yield a cascade of adaptive Cartesian grids. Consequently, any spacetree traversal is equivalent to an element-wise traversal of the hierarchy of the adaptive Cartesian grids. The software Peano realises such a grid traversal and storage algorithm, and it provides hook-in points for applications performing per-element, per-vertex, and so forth operations on the grid. It also provides interfaces for dynamic load balancing, sophisticated geometry representations, and other features. Some properties are enlisted below.

Peano is currently available in its third generation. The development of the original set of Peano codes started around 2002. 2005-2009, we merged these codes into one Peano kernel (2nd generation). In 2009, I started a complete reimplementation of the kernel with special emphasis on reusability, application-independent design and the support for rapid prototyping. This third generation of the code is subject of the present cookbook.

Dependencies and prerequisites

Peano is plain C++ code and depends only on MPI and Intel's TBB or OpenMP if you want to run it with distributed or shared memory support. There are no further dependencies or libraries required. C++ 11 is used. GCC 4.2 and Intel 12 should be sufficient to follow all examples presented in this document. If you intend to use Peano, we provide a small Java tool to facilitate rapid prototyping and to get rid of writing glue code. This Peano Development Toolkit (PDT) is pure Java and uses DaStGen. While we provide the PDT's sources, there's also a jar file available that comprises all required Java libraries and runs stand alone. To be able to use DaStGen—we use this tool frequently throughout the cookbook—you need a recently new Java version.

We recommend to use Peano in combination with

- Paraview (`www.paraview.org`) or VisIt (`https://wci.llnl.gov/simulation/computer-codes/visit/`) as our default toolboxes create vtk files.
- make and `awk`.

But these software tools are not mandatory.

The whole cookbook assumes that you use a Linux system. It all should work on Windows and Mac as well, but we haven't tested it in detail.

Who should read this document

This cookbook is written similar to a tutorial in a hands-on style. Therefore, it also contains lots of source code snippets. If you read through a chapter, you should immediately be able to re-program the presented details in your code and use the ideas.

Therefore, this cookbook is written for people that have a decent programming background as well as scientific computing knowledge. Some background in the particular application area's algorithms for some chapters also is required. If you read about the particle handling in Peano, e.g., the text requires you to know at least some basics such as linked-cell methods. The text does not discuss mathematical, numerical or algorithmic background. It is a cookbook after all.

What is contained in this document

This book covers a variety of problems I have tackled with Peano when I wrote scientific papers. There is no overall read thread through the document. I recommend to start reading some chapters and then jump into chapters that are of particular interest. Whenever something comes to my mind that should be added, I will add it. If you feel something is urgently missing and deserves a chapter or things remain unclear, please write me an email and I'll see whether I can provide some additional text or extend the cookbook.

July 23, 2016
Tobias Weinzierl

Contents

2 Quickstart

Time: Should take you around 15 minutes to get the code up and running. Then another 15 minutes to have the first static adaptive Cartesian grid.
Required: No previous knowledge, but some experience with the Linux command line and Paraview is advantageous.

2.1 Download and install

To start work with Peano, you need at least two things.

1. The Peano source code. Today, the source code consists of two important directories. The `peano` directory holds the actual Peano code. An additional `tarch` directory holds Peano's technical architecture.
2. The Peano Development Toolkit (PDT). The PDT is a small Java archive. It takes away the cumbersome work to write lots of glue code, i.e. empty interface implementations, default routines, ..., so we use it quite frequently.

For advanced features, you might want to use some **toolboxes**. A toolbox in Peano is a small collection of files that you store in a directory and adopt all pathes accordingly. From a user's point of view, when we use the term toolbox we actually mean this directory with all its content.

> **Remark:** Originally, we hoped that Peano's technical architecture (`tarch`) might become of value for several projects, i.e. projects appreciate that they do not have to re-develop things such as logging, writing of output files, writing support for OpenMP and TBB, and so forth. To the best of our knowledge, the tarch however is not really used by someone else, so we cannot really claim that it is independent of Peano. Nevertheless, we try to keep it separate and not to add anyting AMR or grid-specific to the tarch.

There are two ways to get hold of Peano's sources and tools. You either *download the archives from the website* or you *access the repository directly*. Both variants are fine. We recommend to access the respository directly.

2.1.1 Download the archives from the website

If you don't want to download Peano's whole archive, change to Peano's webpage `http://www.peano-framework.org` and grab the files

- `peano.tar.gz` and
- `pdt.jar`

from there. If you do so, please skip the first two lines from the script before. Otherwise, load down the important files with `wget`. Independent of which variant you follow, please unpack the `peano.tar.gz` archive. It holds all required C++ sources.

```
> wget http://sourceforge.net/projects/peano/files/peano.tar.gz
> wget http://sourceforge.net/projects/peano/files/pdt.jar
> tar -xzvf peano.tar.gz
```

There's a couple of helper files that we use IN the cookbook. They are not necessarily required for each Peano project, but for our examples here they are very useful. So, please create an additional directory `usrtemplates` and grap these files

```
> mkdir usrtemplates
> cd usrtemplates
> wget http://sourceforge.net/projects/peano/files/ \
  usrtemplates/VTKMultilevelGridVisualiserImplementation.template
> wget http://sourceforge.net/projects/peano/files/ \
  usrtemplates/VTKMultilevelGridVisualiserHeader.template
> wget http://sourceforge.net/projects/peano/files/ \
  usrtemplates/VTKGridVisualiserImplementation.template
> wget http://sourceforge.net/projects/peano/files/ \
  usrtemplates/VTKGridVisualiserHeader.template
> wget http://sourceforge.net/projects/peano/files/ \
  usrtemplates/VTK2dTreeVisualiserImplementation.template
> wget http://sourceforge.net/projects/peano/files/ \
  usrtemplates/VTK2dTreeVisualiserHeader.template
```

Remark: Many features of Peano are archives into toolboxes, i.e. small extensions of the kernel that simplify your life and provide certain features (such as default load balancing or support of patches or particles in the grid). These toolboxes are stored in Peano's repository as tar.gz files. Alternatively, you can download them from the webpage with `wget http://sourceforge.net/projects/peano/files/toolboxes`.

2.1.2 Access the repository directly

Instead of a manual download, you might also decide to download a copy of the whole Peano repository. This also has the advantage that you can do a simple `svn update` anytime later throughout your development to immediately obtain all kernel modifications.

```
> svn checkout http://svn.code.sf.net/p/peano/code/trunk peano
```

Your directory structure will be slightly different than in the example above, but this way you can be sure you grabbed everything that has been released for Peano through the webpage ever.

The archive `pdt.jar` will be contained in `pdt`, while the two source folders will be held by `src`. The directory `usrtemplates` is contained in `pdt`.

2.1.3 Prepare your own project

From hereon, we recommend that you do not make any changes within Peano repositories but use your own directory `peano-projects` for your own projects. We refer to one of these projects

generically from hereon as `myproject`. Within `peano-projects`, we will need to access the directories `peano` and `tarch`. It is most convenient to create symbolic links to these files. Alternatively, you also might want to copy files around or adopt makefiles, scripts, and so forth. I'm too lazy to do so and rely on OS links.

```
> mkdir peano-projects
> cd peano-projects
> ln -s <mypath>/peano peano
> ln -s <mypath>/tarch tarch
> ls
  peano tarch
```

2.2 Create an empty Peano project

Peano projects require four files from the very beginning:

- A **specification** file is kind of the central point of contact. It defines which data models are used and which operations (algorithmic phases) do exist in your project. And it also specifies the project name, namespace, and so forth.
- A **vertex definition** file specifies which data is assigned to vertices in your grid.
- A **cell definition** file specifies which data is assigned to cells in your grid.
- A **state definition** file specifies which data is held in your solver globally.

We will use these files and modify them all the time. For our first step, they are basically empty. As mentioned before, we suggest to have one directory per project. Rather than creating the files as well as the directory manually, we can use the PDT for this:

```
> java -jar <mypath>/pdt.jar --create-project myproject myproject
> ls
  myproject peano tarch
```

If you are interested in the semantics of the magic arguments, call jar file without any argument and you will obtain a brief description. A quick check shows that the aforementioned four files now have been created:

```
> ls -al myproject
drwxr-xr-x 2 ... .
drwxr-xr-x 5 ... ..
-rw-r--r-- 1 ... Cell.def
-rw-r--r-- 1 ... project.peano-specification
-rw-r--r-- 1 ... State.def
-rw-r--r-- 1 ... Vertex.def
```

The PDT typically is used only once with the `--create-project` argument. From hereon, it serves different purposes. That is ...

2.3 A first spacetree code

... it helps us to write all the type of code parts that we don't want to write: **glue code** that does nothing besides gluing the different parts of Peano together.

We postpone a discussion of the content of the generated files to Chapter 3 and continue to run a first AMR example. For this, we call the PDT again. However, this time, we use the generated specification file as input and tell the tool to create all glue code.

```
> java -jar <mypath>/pdt.jar --generate-gluecode \
  myproject/project.peano-specification myproject \
  <mypath>/usrtemplates
```

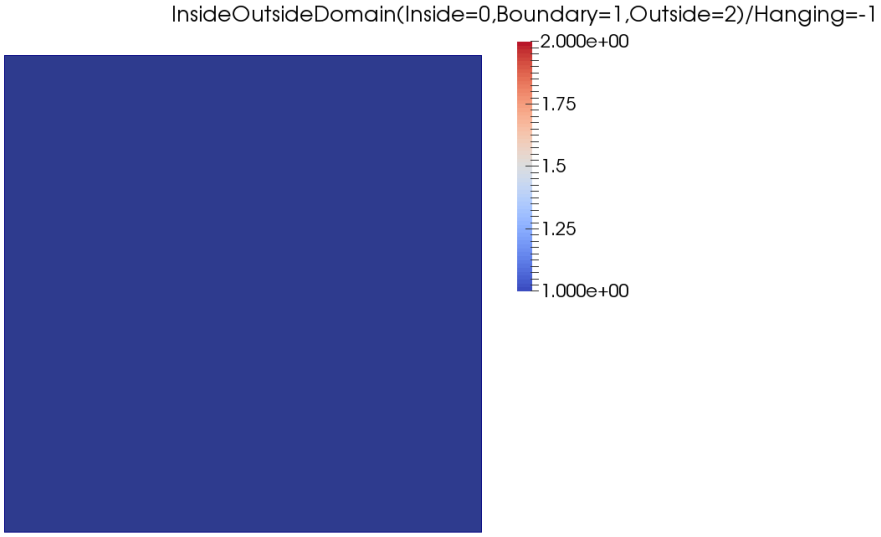
By default, the autogenerated, (almost) empty four files require the `usrtemplates`. We reiterate that many projects later won't need them. If we again study the content of our directory, we see that lots of files have been generated. For the time being, the `makefile` is subject of our interest. Depending on your compiler, you should be able to call `make` straight away. If it doesn't work, open your favourite text editor and adopt the makefile accordingly.

```
> ls myproject
  adapters Cell.cpp Cell.def
  Cell.h dastgen main.cpp
  makefile mappings project.peano-specification
  records repositories runners
  State.cpp State.def State.h
  tests Vertex.cpp Vertex.def
  Vertex.h VertexOperations.cpp VertexOperations.h
> make -f myproject/makefile
> ls
  files.mk myproject peano peano-YourProjectName-debug tarch
```

There it is: the first Peano executable. We can run it straight away:

```
> ./peano-YourProjectName-debug
> ls
  files.mk grid-0.vtk myproject peano
  peano-YourProjectName-debug tarch
```

We see that it has produced a vtk file. So it is time to startup Paraview or VisIt and see what is inside.



Congratulations: We have created the simplest adaptive Cartesian grid in 2d that does exist. A single square!

2.4 Some real AMR

We now set up something slightly more complicated. First of all, we switch to a 3d setup rather than 2d. For this, open the makefile (`myproject/makefile`) and alter the content of the `DIM` variable.

```
# Set Dimension
# -------------
#DIM=-DDim2
DIM=-DDim3
#DIM=-DDim4
```

If you clean your project (`make -f myproject/makefile clean`) and rebuild your code, you see that the individual files are translated with the compile switch

```
g++ ... -DDim3 ....
```

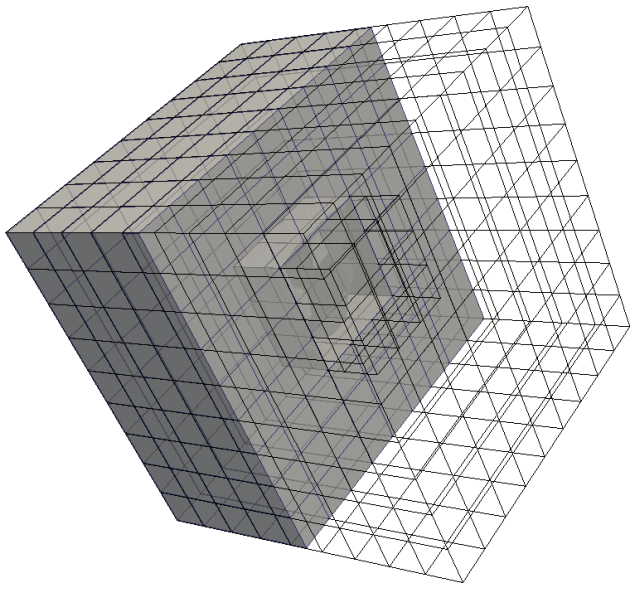
Indeed, this is all that's required for Peano to run a 3d experiment rather than a 2d setup.

> **Remark:** We do support currently up to 10-dimensional setups. If you require higher dimensions, you might even be able to extend Peano accordingly by changing solely the file `peano/utils/Dimensions.h`. But have fun with your memory requirements exploding.

Next, we will edit the file `myproject/mappings/CreateGrid.cpp`. Open it with your favourite text editor and search for the operation `createBoundaryVertex`. Change it into the code below:

```
void myproject::mappings::CreateGrid::createBoundaryVertex(
 myproject::Vertex& fineGridVertex,
 const tarch::la::Vector<DIMENSIONS,double>& fineGridX,
 const tarch::la::Vector<DIMENSIONS,double>& fineGridH,
 myproject::Vertex *const coarseGridVertices,
 const peano::grid::VertexEnumerator& coarseGridVerticesEnumerator,
 myproject::Cell& coarseGridCell,
 const tarch::la::Vector<DIMENSIONS,int>& fineGridPositionOfVertex
) {
  logTraceInWith6Arguments("createBoundaryVertex(...)", ...);
    // leave this first line as it is

  if (coarseGridVerticesEnumerator.getLevel()<2) {
    fineGridVertex.refine();
  }

  logTraceOutWith1Argument("createBoundaryVertex(...)",fineGridVertex);
}
```

If you compile this code and run the executable, you will (besides lots of debug output) obtain a way bigger vtk file. If you visualise it this time, we observe that the code refines towards the cube's boundary. You may want to play around with magic `2` in the operation above. Or you might want to continue to our final example.



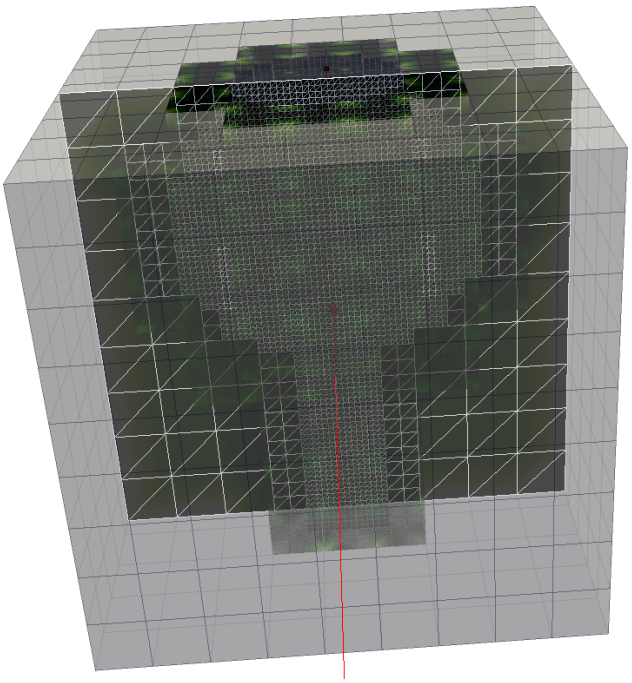
2.5 A tree within the spacetree

In the final example we create a slightly more interesting setup. We solely edit the operation `createInnerVertex` within the file `myproject/mappings/CreateGrid.cpp`, recompile it and have a look at the result. When you study source code, please note the similarity to Matlab when we work with vectors in Peano; as well as that the indices start with 0. If your want to get rid of all the debug statements and are sick of long waiting times, remove the `-DDebug` statement in the line `PROJECT_CFLAGS = -DDebug -DAsserts` within the makefile. There are more elegant ways to filter out log statements that we will discuss later.

```
void myproject::mappings::CreateGrid::createInnerVertex(
 myproject::Vertex& fineGridVertex,
 const tarch::la::Vector<DIMENSIONS,double>& fineGridX,
 const tarch::la::Vector<DIMENSIONS,double>& fineGridH,
 myproject::Vertex *const coarseGridVertices,
 const peano::grid::VertexEnumerator& coarseGridVerticesEnumerator,
 myproject::Cell& coarseGridCell,
 const tarch::la::Vector<DIMENSIONS,int>& fineGridPositionOfVertex
) {
 logTraceInWith6Arguments("createInnerVertex(...)",fineGridVertex,...);

 if (
   fineGridVertex.getRefinementControl()==Vertex::Records::Unrefined
   &&
   coarseGridVerticesEnumerator.getLevel()<4
 ) {
   bool trunk = (fineGridX(0)-0.5)*(fineGridX(0)-0.5)
              + (fineGridX(2)-0.5)*(fineGridX(2)-0.5)<0.008;
   bool treeTop = (fineGridX(0)-0.5)*(fineGridX(0)-0.5)
                + (fineGridX(1)-0.7)*(fineGridX(1)-0.7)
                + (fineGridX(2)-0.5)*(fineGridX(2)-0.5)<0.3*0.3;
   if (trunk | treeTop) {
     fineGridVertex.refine();
```

```
    }
  }

  logTraceOutWith1Argument("createInnerVertex(...)",fineGridVertex);
}
```

So here's what I get. Feel free to create better pics:



Further reading

- Weinzierl, Tobias and Mehl, Miriam (2011). *Peano—A Traversal and Storage Scheme for Octree-Like Adaptive Cartesian Multiscale Grids.* SIAM Journal on Scientific Computing 33(5): 2732-2760.
- Bungartz, Hans-Joachim, Eckhardt, Wolfgang, Weinzierl, Tobias and Zenger, Christoph (2010). *A Precompiler to Reduce the Memory Footprint of Multiscale PDE Solvers in C++.* Future Generation Computer Systems 26(1): 175-182.
- Bungartz, Hans-Joachim, Mehl, Miriam, Neckel, Tobias and Weinzierl, Tobias (2010). *The PDE framework Peano applied to fluid dynamics: an efficient implementation of a parallel multiscale fluid dynamics solver on octree-like adaptive Cartesian grids.* Computational Mechanics 46(1): 103-114.
- Weinzierl, Tobias (2009). *A Framework for Parallel PDE Solvers on Multiscale Adaptive Cartesian Grids.* München: Verlag Dr. Hut.
- Bungartz, Hans-Joachim, Mehl, Miriam, Weinzierl, Tobias and Eckhardt, Wolfgang (2008). *DaStGen—A Data Structure Generator for Parallel C++ HPC Software.* In ICCS 2008: Advancing Science through Computation, Part III. Bubak, van Albada, Sloot and Dongarra, Heidelberg, Berlin: Springer-Verlag. 5103: 213-222.
- Brenk, Markus, Bungartz, Hans-Joachim, Mehl, Miriam, Muntean, Ioan Lucian, Neckel, Tobias and Weinzierl, Tobias (2008). *Numerical Simulation of Particle Transport in a Drift Ratchet.* SIAM Journal of Scientific Computing 30(6): 2777-2798.

3 Basic Programming Course

3.1 Grid creation

Time: 15 minutes for the programming but perhaps around 30 minutes for the visualisation.
Required: Chapter 2.

In this section, we study a 2d example. Please adopt your makefile accordingly. Furthermore, we use the files `VTKMultilevelGridVisualiserHeader` and `...Implementation` as well as `VTK2dTreeVisualiser...`. If you have downloaded the whole Peano repository, these files can be found in `pdt/usrtemplates`. If not, you have to download them manually from the webpage. Please set up an empty project as discussed in Chapter 2 and implement one operation as follows (all other operations can remain empty/only filled with log statements):

```
void myproject::mappings::CreateGrid::touchVertexLastTime(
      myproject::Vertex& fineGridVertex,
      const tarch::la::Vector<DIMENSIONS,double>& fineGridX,
      const tarch::la::Vector<DIMENSIONS,double>& fineGridH,
      myproject::Vertex *const coarseGridVertices,
      const peano::grid::VertexEnumerator& coarseGridVerticesEnumerator,
      myproject::Cell& coarseGridCell,
      const tarch::la::Vector<DIMENSIONS,int>& fineGridPositionOfVertex
) {
  logTraceInWith6Arguments( "touchVertexFirstTime(...)", fineGridVertex, fineGridX, ...

  if (
    coarseGridVerticesEnumerator.getLevel()<5
    &&
    tarch::la::equals( fineGridX, 0.0 )
    &&
    fineGridVertex.getRefinementControl()==Vertex::Records::Unrefined
  ) {
    fineGridVertex.refine();
  }

  logTraceOutWith1Argument( "touchVertexFirstTime(...)", fineGridVertex );
}
```

This source fragment requires some additional explanation. We neglect the enumerator stuff for the time being. That will become clear later throughout the present chapter. The refinement control check says 'well, refine, but do it only on unrefined vertices'. It's just a matter of good style, not to call `refine` on a refined vertex. The middle line uses a function from the tarch's linear algebra namespace. It takes the `fineGridX` vector (the position of the vertex in space) and checks whether all entries equal zero. As we are working with floating point numbers, it is not a bit-wise check. Instead, it uses an interval of machine precision around zero. You may want to

change this notion of machine precision in Peano (file `Scalar` within the `tarch::la` namespace). In general, it would be a good idea to study the content of the `la` component soon—there's lots of useful stuff in there to work with tiny, dense vectors[1].

> **Remark:** Peano realises a **vertex-based**, **logical-or** refinement: You can invoke `refine` on any unrefined vertex. Peano then refines all cells around a vertex in the present or next traversal (it basically tries to do it asap, but sometimes data consistency constraints require it to postpone the actual refinement by one iteration). The other way round, you may read it as follows: A cell is refined if the refinement flag is set for any adjacent vertex.

Whenever you use Peano, you have to do three things:

1. Decide which algorithmic phases do exist and in which order they are called. Examples for algorithmic phases could be: set up grid, initialise all variables, refine regions of interest, perform an iterative solve step, plot some data, compute metrics on the solution, ...
2. Model the data, i.e. decide which data is assigned to the vertices and cells of the grid.
3. Implement the different actions on this data model that are used by the algorithmic phases.

This scheme lacks the bullet point 'run through the grid'. Indeed, Peano applications do never run themselves through the spacetree. They specify which set of operations is to be called throughout a run through the grid, i.e. they say what is done on which data. Afterward, they invoke the iteration and leave it to Peano to run through the grid and invoke these operations in the right order on the right ranks using all the cores you have on your machine[2]. This scheme realises something people call 'The Hollywood Principle':Don't call us, we call you!

> **Remark:** The **inversion of control** is the fundamental difference of Peano to other spacetree-based codes offered as a library. And typically it is the property many users first struggle with. Often, people claim 'I have to run through the grid this and that way'. Often, they are wrong. It can become quite comfortable to leave it to someone else to decide how grid traversals are realised. And it allows the grid traversal in turn to optimise the code under the hook without an application developer to bother.

3.1.1 On the power of loosing control

The algorithmic phases, i.e. what can be done on a grid, are specified in the specification file. Open your project's file. There are two different parts of the document that are of interest to us: An *event mapping* is an algorithmic step that you have to implement yourself. In this chapter's example, we want to do two things: create a grid and count all the vertices. Furthermore, we want to plot our grid, but let's keep in mind that Peano has some predefined actions as well. So we augment our mapping set as follows:

```
// Creates the grid
event-mapping:
```

[1] Peano someday should perhaps be rewritten to use boost linear algebra or some fancy template library. Feel free to do so. Right at the moment, it is all plain hand-craftet routines.

[2] This statement requires explanation, and indeed it is not *that* straightforward. But the idea is phrased correctly: the application codes specifies what is to be done and then outsources the scheduling and the responsibility to use a multicore machine to Peano.

```
  name: CreateGrid

// Counts all the vertices within the grid
event-mapping:
  name: CountVertices
```

Event mappings cannot be used directly. Instead, we have to specify adapters. Adapters take the tree traversal and invoke for each grid part a set of events. As we distinguish adapters which basically just glue together (multiple) events from the events themselves, we will be able to do the following later: we write a fancy visualisation routine, a routine that adopts the grid to a new data set and some compute routines. As we have done this in three different event sets, we can then combine these events in various ways: compute something and at the same time plot, compute only, plot and afterward adopt the grid, and so forth. For the time being, we use the following adapters:

```
adapter:
  name: CreateGrid
  merge-with-user-defined-mapping: CreateGrid

adapter:
  name: CountVertices
  merge-with-user-defined-mapping: CountVertices

adapter:
  name: CreateGridAndPlot
  merge-with-user-defined-mapping: CreateGrid
  merge-with-predefined-mapping: VTKGridVisualiser(finegrid)
  merge-with-predefined-mapping: VTKMultilevelGridVisualiser(grid)

adapter:
  name: CountVerticesAndPlot
  merge-with-user-defined-mapping: CountVertices
  merge-with-predefined-mapping: VTKMultilevelGridVisualiser(grid)

adapter:
  name: Plot
  merge-with-predefined-mapping: VTKGridVisualiser(finalgrid)
```

The first two adapters are trivial: They basically delegate to one event set. The next two take one event set each and invoke it. Furthermore, they also use a predefined event set. They will call `CreateGrid` or `CountVertices`, respectively, and at the same time plot. If you create all code with

```
java -jar <mypath>/pdt.jar --generate-gluecode
myproject/project.peano-specification myproject <mypath>/usrtemplates
```

it is the directory `usrtemplate` where the PDT searches for the predefined event sets. The last adapter by the way is a trivil one, too: It invokes only one of the events that ship with Peano.

Remark: You may mix predefined and user-defined mappings in your adapters in arbitrary order. They are always ran in the order specified per event. If you have to call different mappings in different order for different events (i.e. one way in `createHangingNode` but the other way round for destruction), you have to split up the events into a preamble and epilogue event and merge them together in the spec file.

Next, please create all glue code and have a quick look into the file `runners/Runner.cpp`. This file is the starting point of Peano. The C++ main routine does some setup steps and then creates an instance of the Runner (see the source code yourself if you don't believe). It then invokes `run()` which in turn continues to `runAsMaster` or `runAsWorker()`. The latter will play a role once we use MPI. For the time being, let's focus on the master's routine. Here, we see the following:

```
int myproject::runners::Runner::runAsMaster(myproject::repositories::Repository& repository) {
  peano::utils::UserInterface userInterface;
  userInterface.writeHeader();

  // @todo Insert your code here

  // Start of dummy implementation

  repository.switchToCreateGrid(); repository.iterate();
  repository.switchToCountVertices(); repository.iterate();
  repository.switchToCreateGridAndPlot(); repository.iterate();
  repository.switchToCountVerticesAndPlot(); repository.iterate();
  repository.switchToPlot(); repository.iterate();

  repository.logIterationStatistics();
  repository.terminate();
  // End of dummy implementation

  return 0;
}
```

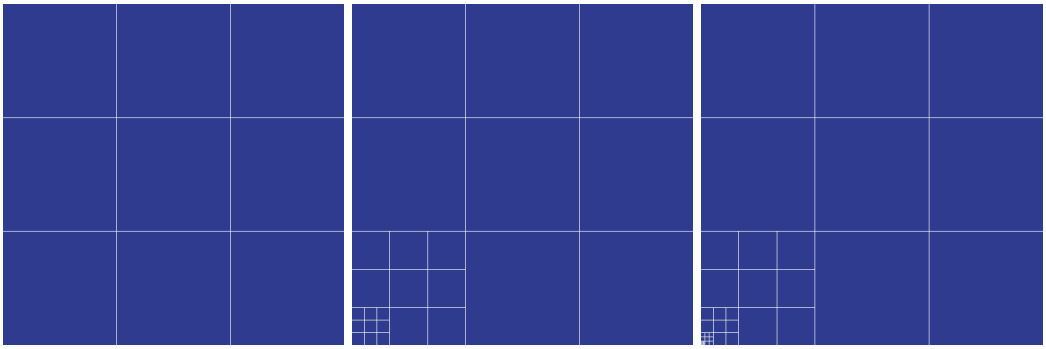
The PDT cannot know what exactly we do, so it basically runs all the adapters we have specified. Once there is a runner implementation in place, the PDT never overwrites this file. Whenever the PDT overwrites files, it places a `readme.txt` file in the corresponding directory and highlights this explicitly. There are very few directories whose files are overwritten. Most files are never overwritten, i.e. if you want PDT to regenerate something, you have to delete the directory explicitly before. In the example above, it might be that the runner has been generated by your first PDT run. As such, some lines might be missing, some might be slightly different. This is the place where we implement our overall algorithm, i.e. the big picture. So we have to modify it anyway. Lets change the function body as follows:

```
  peano::utils::UserInterface userInterface;
  userInterface.writeHeader();

  repository.switchToCreateGridAndPlot();
  for (int i=0; i<10; i++) repository.iterate();
  repository.switchToCountVertices(); repository.iterate();

  repository.logIterationStatistics();
  repository.terminate();
```

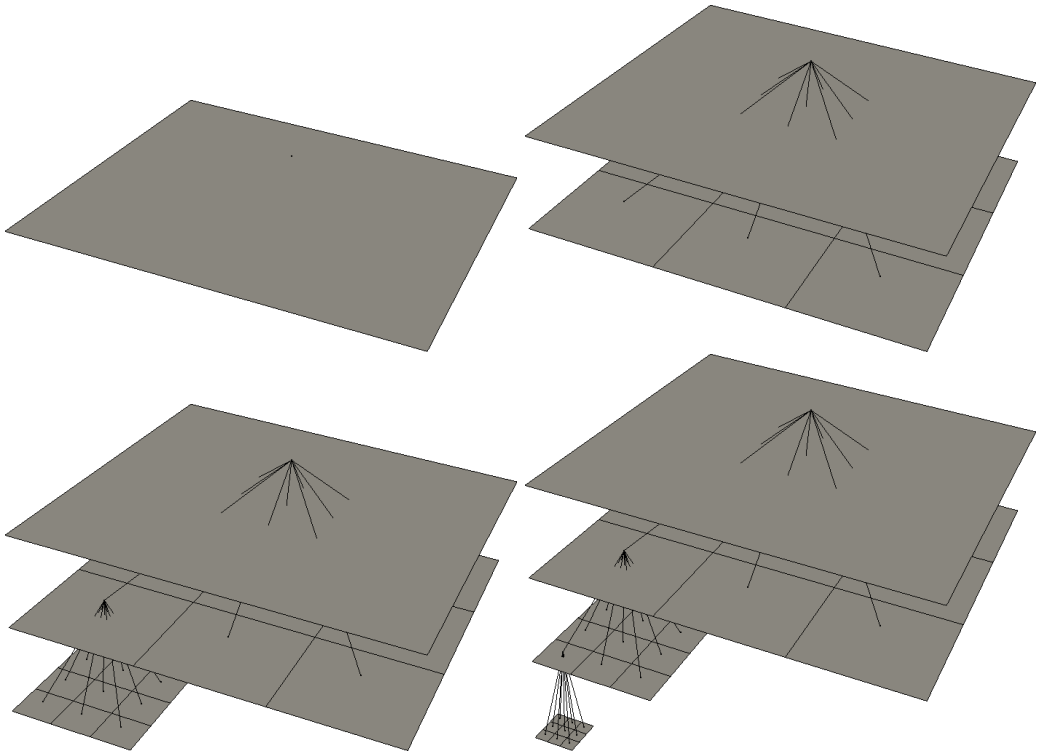
```
  return 0;
```

This algorithm says that we want to create a grid and at the same time plot it. We want to do this ten times in a row. Afterward, we switch to our vertex counting and want to run through the grid once more. This time, nothing shall be plotted. We just want to know how many vertices there are.

We really do not care about how the code runs through the grid. We also do not really care how all vertices, cells, whatever are processed. We say what is done in which order from a bird eye's perspective. If you compile the code and run it now, you should end up with a sequence of vtk files. You might want to make a video (take the files that are called `finegrid`-something). Below are some screenshots:



3.1.2 What happens



To understand what is happening, we can read all the outputs written to the terminal by Peano (see the next chapter how to remove them/filter them). Or we can just give a quick sketch:

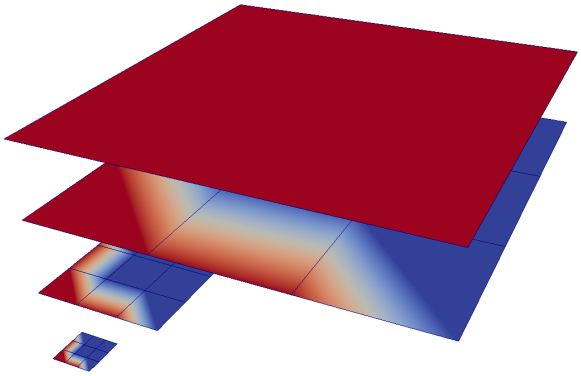
1. We first create a repository. A repository is basically the spacetree, i.e. the grid, plus all the adapters we've defined. It also provides some statistics the can read out.
2. The code runs into the Runner's `runAsMaster()` routine and selects which adapter to use. This is the switch statement. In a parallel code, all involved ranks would now immediately active this adapter.
3. In the runner, we then call the `iterate` operation on the repository. The repository now starts to run through the whole grid. Actually, it runs through the spacetree in a kind of top-down way.
4. Whenever it encounters an interesting situation (it loads a vertex for the very first time, e.g.), it triggers an event. Event means that it calls an operation on the adapter.
5. An adapter may fuse multiple events; both predefined and user-defined. Per event, it calls all these mappings' implementations one after another. The order is the same you have used in your specification file.

> **Remark:** This document does not run through the list of available events, in which order they are called and so forth. You may want to have a look into any event's header. The PDT augments the header with a quite verbose explanation what event is called when.

It might now be the right time to look into one of these mapping headers to get a first impression what is available. Basically, they describe all important plug-in points that you might want to use in any element-wise multiscale grid traversal.

3.1.3 Multiscale data

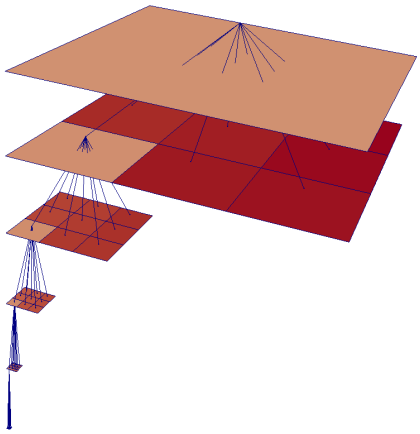
Prior to a discussion of Peano's data model, it might make sense to load all the files `grid-level-?-0.vtk` into your favourite visualisation software. Dilate them according to their level encoded in the name. In Paraview, you have to select each file, apply Filters/Alphabetical/Transform, and dilate each file by -0.2 along the z-axis per level. You might end up with something alike:



Again, feel free to create a video that shows how additional levels are added in each step. There's a more elegant way to end up with a similar picture. I once created a graph visualiser that is today also available as predefined mapping. Just modify your adapters as follows:

```
adapter:
  name: CreateGridAndPlot
  merge-with-user-defined-mapping: CreateGrid
  merge-with-predefined-mapping: VTK2dTreeVisualiser(tree,getLevel)
```

This time, creating a video should be straightforward.



We see that speaking of Peano in terms of a spacetree software is only one way to go. We also could speak of it of a software managing Cartesian grids embedded into each other. The latter point of view reveals an important fact: Peano handles vertices and cells. Cells are embedded into each other as they form the tree and there is a clear parent-child relation. Vertices connect cells. A vertex is unique due to its position in space plus its level, i.e. there might be some coordinates that host multiple vertices. In math, we would call this generating system. There are two types of vertices: the standard ones are adjacent to 2^d cells on the same level with d being the dimension of the problem. All other vertices are hanging nodes.

> **Remark:** Hanging nodes are not stored persistently. They are created and destroyed in each traversal upon request and never stored in-between two iterations. You can never be sure how often a hanging nodes is constructed per traversal. It could be up to $2^d - 1$ times. You only know it is created once at least.

Cells and vertices hold data. This data is described in the files `Vertex.def` and `Cell.def`. The .def files feed into our tool DaStGen that translated them internally into a standard C++ class (though it does some more stuff: it autogenerates all MPI data types that we need later on for parallel codes, and it in particular compresses the data such that a bool field is really mapped onto a single bit, e.g.). There's an alternative way to hold data on the grid that is called heap: in this case, we do not assign a fixed number of properties to vertices or cells. Both storage variants are discussed in my tech report *The Peano software - parallel, automaton-based, dynamically adaptive grid traversals* which can be obtained from arXiv, e.g.

3.2 Logging, statistics, assertions

Time: There are no real examples coming along with this chapter, but you might want to use its topics for any project.
Required: Chapter 2.

3.2.1 Logging

Peano relies on a plain interface to write out user information. All constituents of this interface are collected in the package `tarch::logging`. Documentation on this package can be found in the corresponding header files, files ending with `.doxys`, or the Peano webpage (section on sources). The usage paradigm is simple:

1. Each class that wants to write logging information requires an instance of class `tarch::logging::Log`. As such an instance is required per class, it makes sense to make this field a static one.

```
#include "tarch/logging/Log.h"
...
class MyClass {
  private:
    static tarch::logging::Log _log;
};

tarch::logging::Log MyClass::_log( "MyClass" );
```

   For most auto-generated classes, the PDT already creates the `_log` instance. Please keep to the nomenclature of the class field to make all macros work. Please use as string argument in the constructor the fully qualified class name.

2. Whenever you want to log, you should use Log's' operations to write the messages. Alternatively, you may want to use the log macros from `Log.h`. They work with stringstreams internally, i.e. you may write things along the lines

```
logInfo( "runAsMaster(...)", "time step " << i << ": dt=" << dt );
```

   where you concatenate the stream with data.

Peano offers three levels of logging:

- **Info**. Should be used to inform your user about the application's state.
- **Warning**. Should be used to inform your user about bothering behaviour. The MPI code uses it, e.g., if many messages arrive in a different order than expected. Messages written to the warning level are piped to `cerr`.
- **Error**. Should be used for errors. Is piped to `cerr` as well.
- **Debug**. Should be used to write debug data. It goes to `cout` and all debug data is removed if you do not translate with the compile flag `-DDebug`. Notably, use the `logDebug` macros when you write to the debug level, as all required log operations then are removed by the compiler once you create a release version of your code.

Logging device: CommandLineLogger

The `Log` instance forwards the information to a logger. By default, this is the `tarch::logging::CommandLineLogger` which writes information in a table-like format. You may want to write your own alternative implementation of the logger if you require formats such as XML.

Alternatively, you can configure the command line logger to plot only those fields that are of relevance to you. For this, the logger provides a particular setter. Please consult the header or the webpage for details on the semantics of the arguments:

```
tarch::logging::CommandLineLogger::getInstance().setLogFormat(
  "␣", true, false, false, true, true, "my-fancy.log-file" );
```

This interface also allows you to pipe the output into a file rather than to the terminal. This is particular useful for MPI applications, as each rank is assigned a file of its own and messages are not screwed up. Typically, the logger is configured in the `main` of the application.

If you run Peano for long simulations and, notably, if you run Peano with debug information switched on (`-DDebug`), log files soon become massive. To ease the pain, Peano's command line logger offers an operation

```
tarch::logging::CommandLineLogger::getInstance().closeOutputStreamAndReopenNewOne();
```

that works if you have specified an output file before (see `setLogFormat` above). Per close, you close all output files and start to stream the output into a new file. Typically, developers use this operation in their iterative schemes to stream each iteration to a different file. The output files are enumerated automatically.

Log filter

The amount of log information often becomes hard to track; notably if you run in debug mode. Often, you are interested only in a subset of all log messages. For this, Peano offers log filters which provide a blacklist and whitelist mechanism to filter messages before they are written. A log filter entry is created by

```
tarch::logging::CommandLineLogger::getInstance().addFilterListEntry(
  ::tarch::logging::CommandLineLogger::FilterListEntry(
    "debug", -1, "myproject", false ) );
```

and again this is something that is typically done in the `main`. See the `CommandLineLogger` header for details on the log filters.

Configuring log filters in your source code is a convenient option when you start a new project. On the long run, it is cumbersome if you have to recompile every time you want different log information. Therefore, the `CommandLineLogger` also offers a routine that allows you to load log filter entries from a text file. This facilitates work with log filters. The usage is straightforward

```
tarch::logging::LogFilterFileReader::parsePlainTextFile( "my.log-filter" );
```

and the format is very simple:

```
# Level  Trace    Rank   Black or white list entry
# (info or debug) (-1 means all ranks)
debug    tarch    -1     black
debug    peano    -1     black
info     tarch    -1     black
info     peano    -1     black
```

Tracing

Peano uses tracing command in several places. Consult the mapping classes generated by the PDT, e.g. Tracing commands are basically debug statements, and once you compile your code without `-DDebug` or with log filters on the debug level, tracing messages are removed from the output. In Peano, trace messages are used to track when a method is entered and when the code leaves a routine. They can be found all over the code. The additional benefit of the trace routines compared to pure debug statements is that the tracings also apply a well-suited indentation, i.e. when you enter a routine, all messages afterwards are indented by two spaces (default; can be reconfigured) afterwards until you leave this operation again.

> **Remark:** If you want to get familiar with the program workflow, you might want to use a debugger to step to your program. This is time consuming. Another option is to configure your log filters such that only the trace messages (debug messages) from the mapping are printed to the terminal/a file. You can then run through this output and see which operations from the mapping are called at which time.

4 Applications

4.1 The heat equation with an explicit Euler

Time: 60 minutes.
Required: Chapter 2.

In this section, we sketch how to realise a heat equation solver for

$$\partial_t u - \nabla(\epsilon\nabla)u = 0$$

that is based upon an explicit Euler. It uses the spacetree as computational grid.

4.1.1 Preparation

We create an empty project with the PDT (`--create-project myproject myproject`) and first adopt our cell and vertex data structure such that each cell holds an ϵ value and each vertex holds the current and previous solution.

```
Packed-Type: short int;
class myproject::dastgen::Vertex {
  parallelise persistent double u;
  discard double oldU;
};
```

```
Packed-Type: short int;
class myproject::dastgen::Cell {
  persistent double epsilon;
};
```

Furthermore, we make the code's state hold the solver's time step size. We will write the code such that it works in 2d and 3d. All the pictures are done for a 3d setup. To run with other dimensions, you have to adopt your makefile accordingly.

```
Packed-Type: short int;
class myproject::dastgen::State {
  persistent parallelise double dt;
};
```

We do not use any sophisticated plotting routines and thus rely in some plotters that are available out-of-the-box for Peano. We also require only two mappings: one for the setup, one for the time stepping. Depending on our personal choices, we combine them with plotting features or not[1].

[1] There is a known issue with the coding standards in Peano/PDT which can be avoided a priori if all read and write attributes start with an uppercase—even though they might be defined with lowercase in the def file.

```
omponent: ExplicitEulerForHeatEquation
namespace: ::myproject
vertex:
  dastgen-file: Vertex.def
  read scalar(double): U
  read scalar(double): OldU
  write scalar(double): U
cell:
  dastgen-file: Cell.def
state:
  dastgen-file: State.def
event-mapping:
  name: CreateGrid
event-mapping:
  name: TimeStep
adapter:
  name: CreateGrid
  merge-with-user-defined-mapping: CreateGrid
adapter:
  name: TimeStep
  merge-with-user-defined-mapping: TimeStep
adapter:
  name: CreateGridAndPlot
  merge-with-user-defined-mapping: CreateGrid
  merge-with-predefined-mapping: VTKPlotCellValue(epsilon,getEpsilon,eps)
  merge-with-predefined-mapping: VTKPlotVertexValue(initialSetup,getU,u)
adapter:
  name: TimeStepAndPlot
  merge-with-user-defined-mapping: TimeStep
  merge-with-predefined-mapping: VTKPlotVertexValue(result,getU,u)
```

To translate this file, you need the corresponding predefined mappings that are held in the repository or on the webpage. To find out what the arguments of the predefined mappings mean, please have a look into the corresponding template header files. We note that we write out epsilon only throughout the setup phase. It does not make sense to plot each each iteration, as this material parameter does not change in time. We furthermore note that we add some `read` and `write` statements. They make the PDT generate helper methods that allow us within each cell to access all u values of a cell as one vector.

4.1.2 Making the plotter work

This code so far does not compile. It complains with

```
> make -f myproject/makefile
---This is Peano 3 ---
g++ -DDim3 [...] -c myproject/adapters/CreateGridAndPlot2VTKPlotCellValue_0.cpp -o \
myproject/adapters/CreateGridAndPlot2VTKPlotCellValue_0.o
[...] In member function void [...]::CreateGridAndPlot2VTKPlotCellValue_0::enterCell([...]):
[...] error: class myproject::Cell has no member named getEpsilon
      _cellValueWriter->plotCell(cellIndex,fineGridCell.getEpsilon() );
                                                        ^
make: ***[myproject/adapters/CreateGridAndPlot2VTKPlotCellValue_0.o] Error 1
```

This is correct. We have told the predefined mapping that there would be an opteration `getEpsilon` to print cell data, but we have not provided one yet. A similar reasoning holds for the plotting of the actual solution. Therefore, we add

```
void myproject::Cell::init() {
  _cellData.setEpsilon( 1.0 + static_cast<double>(rand() % 100)/100.0 );
}

double myproject::Cell::getEpsilon() const {
  return _cellData.getEpsilon();
}
```

This snippet also allows us to initialise a cell with a random value from $(1, 2)$.

> **Remark:** An initialisation of the cell's ϵ in the default constructor does not work because of the flyweight pattern (see next remark). Instead, we have to provide an explicit initialisation routine, and we have to call this routine within the mapping's `createCell` event.

We reiterate our code extension for the vertex,

```
double myproject::Vertex::getU() const {
  return _vertexData.getU();
}
```

Finally, we plug into `CreateGrid`'s `createInnerVertex` and `createBoundaryVertex` and add some refinement statements:

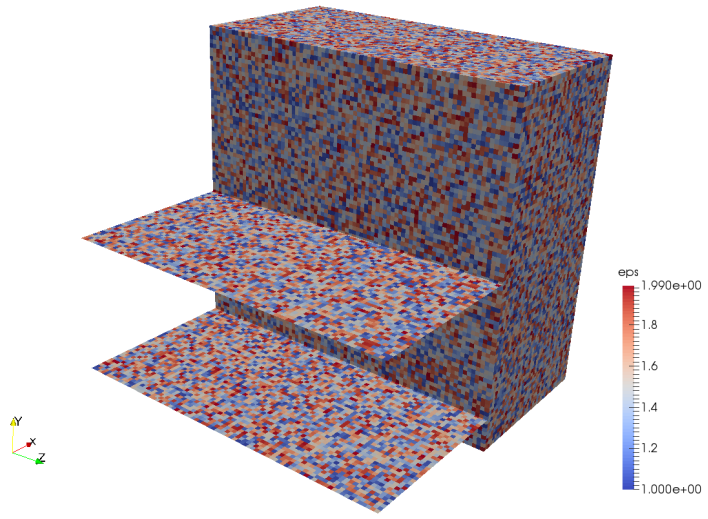
```
void myproject::mappings::CreateGrid::createInnerVertex(...) {
 if (coarseGridVerticesEnumerator.getLevel()<3) {
  fineGridVertex.refine();
 }
}

void myproject::mappings::CreateGrid::createBoundaryVertex(...) {
 if (coarseGridVerticesEnumerator.getLevel()<3) {
  fineGridVertex.refine();
 }
}

void myproject::mappings::CreateGrid::createCell(...) {
 logTraceInWith4Arguments( "createCell(...)", fineGridCell, ... );

 fineGridCell.init();

 logTraceOutWith1Argument( "createCell(...)", fineGridCell );
}
```

As soon as this first plot is available, as add a time stepping loop in the `runners::Runner`:

```
int myproject::runners::Runner::runAsMaster(myproject::repositories::Repository& repository) {
  peano::utils::UserInterface userInterface;
  userInterface.writeHeader();
```

```
  repository.switchToCreateGridAndPlot();
  repository.iterate();

  repository.getState().setTimeStepSize( 0.5e-7 );
  for (int i=0; i<10000; i++) {
    if (i%100==0) {
      repository.switchToTimeStepAndPlot();
    }
    else {
     repository.switchToTimeStep();
    }
    repository.iterate();
  }

  repository.logIterationStatistics();
  repository.terminate();

  return 0;
}
```

The implementation of the `State`'s `void setTimeStepSize(double dt)` operation is left to the reader. Please add the corresponding `double getTimeStepSize() const`, too.



Remark: It is a 'original' decision to model states, vertices and cells as classes that actually aggregate their data objects (attribute `_stateData`, `_vertexData` or `_cellData`, respectively). The reason for this is two-fold: On the one hand, this allows us to separate user-defined code (the aggregating class) from the data model generated by DaStGen. The latter also comprises complex technical details such as the MPI data types or bit compression. If application-specific code is rewritten, the data model is not affected and the other way round. On the other hand, the pattern allows us to apply the flyweight pattern. A vertex object is not actually stored and loaded from input data. It exists only a few time in total, and in each step its change is exchanged underneath by the traversal.

4.1.3 A stencil code

In this example, we stick to a finite differences formulation and vertex-centred unknown assignment to do the time stepping. Our strategy (within the mapping) is simple:

1. In `beginIteration`, we grab the time step size from the state. This way, the user might alter the time step size in the outer control loop (adaptive time stepping). The mapping then always works with the right parameter.
2. In `touchVertexFirstTime`, we take the current solution and back it up in the vertex's property `_oldU`. This property is marked as discard, i.e. the additional helper variable per vertex is not held in-between two iterations (actually it is only held for a small number of vertices that are still in use).
3. In `enterCell`, we element-wisely accumulate the new solution in the vertices.

```
class myproject::mappings::TimeStep {
 private:
  /**
   *Logging device for the trace macros.
   */
  static tarch::logging::Log _log;

  double _timeStepSize;
  ...
};


void myproject::mappings::TimeStep::beginIteration(
  myproject::State& solverState
) {
  logTraceInWith1Argument( "beginIteration(State)", solverState );

  _timeStepSize = solverState.getTimeStepSize();

  logTraceOutWith1Argument( "beginIteration(State)", solverState);
}
```

We reiterate that the state is not available to a mapping by default. If you need the state (or one of its properties), you explicitly have to grab this data in `beginIteration`. As an alternative to the double above, it also would be possible to copy the whole state. If you want to modify the solver's state, you have to alter it in `endIteration`. As Peano requires the user to explicitly move state data around when required, we ensure that the data remains consistent in the parallel code variants.

```
void myproject::mappings::TimeStep::touchVertexFirstTime(...) {
 ...
 fineGridVertex.copyCurrentSolutionIntoOldSolution();
 ...
}

void myproject::Vertex::copyCurrentSolutionIntoOldSolution() {
  _vertexData.setOldU( _vertexData.getU() );
}
```

The interesting stuff happens in the mapping's `enterCell`, where we first of all take all 2^d vertices and write their old solution into one double vector. For this, there's a predefined operation in `VertexOperations` as we have asked the PDE that we read the scalar. This 2^d vector then is multiplied with the local assembly matrix subject of an ϵ scaling. The result finally is added to the new value. Again, we us the generated read and write methods.

```
#include "myproject/VertexOperations.h"
#include "tarch/la/Matrix.h"

void myproject::mappings::TimeStep::enterCell(...) {
  logTraceInWith4Arguments( "enterCell(...)", fineGridCell, ... );

  tarch::la::Matrix<TWO_POWER_D,TWO_POWER_D,double> A;

  A = 6.0/8.0, -1.0/4.0, -1.0/4.0, 0.0, -1.0/4.0, 0.0, 0.0, 0.0,
      -1.0/4.0, 6.0/8.0, 0.0, -1.0/4.0, 0.0, -1.0/4.0, 0.0, 0.0,
      -1.0/4.0, 0.0, 6.0/8.0, -1.0/4.0, 0.0, 0.0, -1.0/4.0, 0.0,
            0.0, -1.0/4.0, -1.0/4.0, 6.0/8.0, 0.0, 0.0, 0.0, -1.0/4.0,
      -1.0/4.0, 0.0, 0.0, 0.0, 6.0/8.0, -1.0/4.0, -1.0/4.0, 0.0,
            0.0, -1.0/4.0, 0.0, 0.0, -1.0/4.0, 6.0/8.0, 0.0, -1.0/4.0,
            0.0, 0.0, -1.0/4.0, 0.0, -1.0/4.0, 0.0, 6.0/8.0, -1.0/4.0,
            0.0, 0.0, 0.0, -1.0/4.0, 0.0, -1.0/4.0, -1.0/4.0, 6.0/8.0;

  tarch::la::Vector<TWO_POWER_D,double> uOld =
    VertexOperations::readOldU(fineGridVerticesEnumerator,fineGridVertices);

  const double h = fineGridVerticesEnumerator.getCellSize()(0);

  tarch::la::Vector<TWO_POWER_D,double> uUpdate =
    -_timeStepSize *fineGridCell.getEpsilon() *A *uOld / h / h ;

  VertexOperations::writeU(
    fineGridVerticesEnumerator,fineGridVertices,
    VertexOperations::readU(fineGridVerticesEnumerator,fineGridVertices) + uUpdate
  );

  logTraceOutWith1Argument( "enterCell(...)", fineGridCell );
}
```

In this example, I wrote down the local assembly matrix for $d = 3$ explicitly. If you want to support other dimensions, you have to adopt this part accordingly.

We use Peano's linear algebra routines here. They are held in the `tarch` component, and provide all basic functionality we typically need. Obviously, it might make sense to switch to other linear algebra packages (BLAS, e.g.) for more demanding setups.

For a better understanding, it might make sense to have a look into `readOldU`. We note that `fineGridVertices` is a pointer to an array of vertices, but the layout of the vertices within this array remains open. Therefore, Peano hands over an enumerator object. The enumerator object has a functor to allow us to select the right vertices. `fineGridVertices[ fineGridVerticesEnumerator(0) ]` for example returns the bottom left vertex of the cell. See the enumerator's documentation in the header for detailed information. The read and write that are generated by the PDT run through the vertices and collect the `u` or `oldU` value, respectively, in one vector. They are scatter/gather operations. Some codes might prefer not to rely on the temporary vectors and instead work with the data associated to the vertices directly. Both options are fine, both options might have different performance characteristics.

Obviously, the above code is not very elaborate. The stiffness matrix is set up multiple times. The most basic optimisation would be to make the matrix an attribute of the mapping and to initialise it only once.

> **Remark:** On the Peano webpage and in the repository, you find a `matrixfree` toolbox. It contains all kind of primitive helper operations that allow you to work with stencil codes. It is not as powerful as a real stencil compiler or other matrix-free/PDE toolboxes, but it a small suite to build up at least the simpler PDE operators from a finite element/tensor-product formalism and it also provides operations fitted to PDT's generated read and write routines. The toolbox also provides helper functions that decompose stencils in element-wise assembly matrices as the one above.

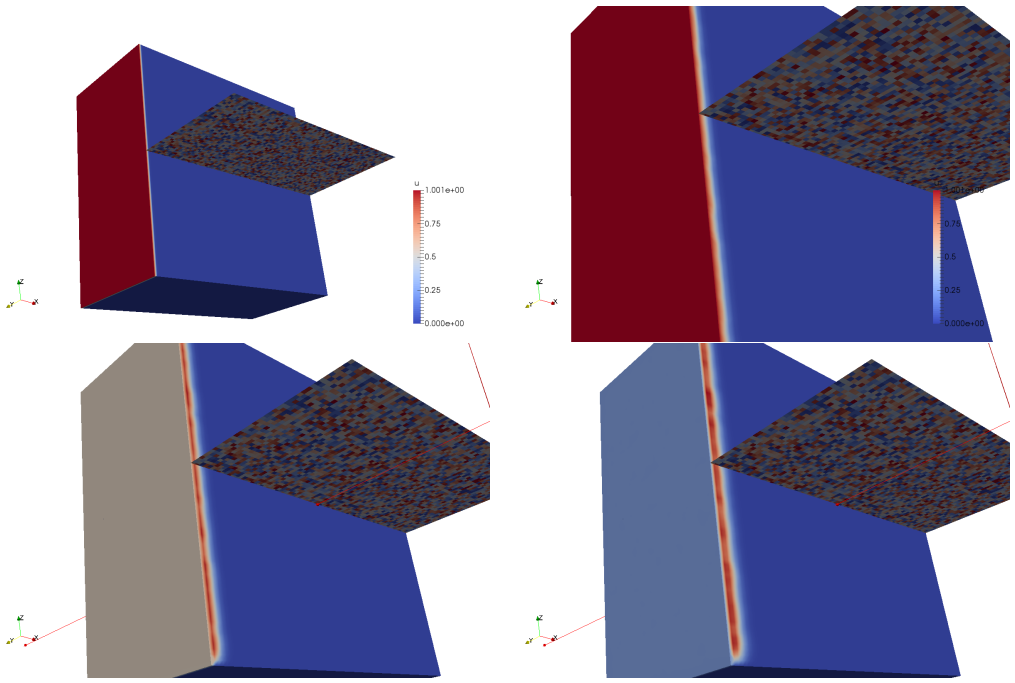
Obviously, our code does not do anything as we do not set any nonzero boundary conditions. Lets do this upon a vertex's first use. As we have specified a read, the PDT provides write operations that break up the object encapsulation. We make use of this now:

```
void myproject::mappings::TimeStep::touchVertexFirstTime(...) {
  ...
  if (fineGridVertex.isBoundary() && fineGridX(0)<1e-8) {
    VertexOperations::writeU( fineGridVertex, 1.0 );
  }
  else if (fineGridVertex.isBoundary()) {
    VertexOperations::writeU( fineGridVertex, 0.0 );
  }
  ...
}
```

> **Remark:** In this example, we do not ensure that we do not update boundary vertices. Indeed, we update them, but in the subsequent traversal we overwrite them again with the prescribed Dirichlet values (code snippet above). Most codes rather add an additional if statement checking `fineGridVertex.isBoundary()`.

@todo Update boundary types

> **Remark:** It is obvious that the Jacobi update scheme may only update inner unrefined vertices as well as Neumann boundary points. Peano does not offer vertex types. It distinguishes only inner and outer vertices. If you need more flags than inside and outside (usually, you need at least a boundary flag), you have to implement this on your own[a].
>
> [a] Up to early 2016, Peano had a three-valued logic with inner, outer and boundary vertices. I removed this from the kernel as most applications need way more vertex types anyway.



All figures cut through the domain in the middle and use an orthogonal slice to visualise the permeability ϵ. Top, left: We start from a zero condition where only the face at $x = 0$ is set to 1. Top, right: Very slowly, the temperature propagates through the domain. As ϵ is fuzzy, the propagation profile does not exhibit symmetry but is ragged as well. Bottom: The temperature propagates further into the medium.

4.1.4 Multiscale data representation

We continue the solver development with a technical extension that proofs to be of great value. Rather than working on the actual fine grid, i.e. our compute grid, we inject the finest solutions to the coarser grids all the time: whenever a vertex coincides with a vertex on a coarser level, we take its value and copy it to the coarser grid. To realise this injection, we typically propose to introduce a new mapping in the specification, and to realise the injection in this new mapping—it has nothing to do directly with the solver, so the separation of mappings reflects a separation of concerns.

```
...
event-mapping:
  name: Inject
...
adapter:
  name: TimeStep
  merge-with-user-defined-mapping: TimeStep
  merge-with-user-defined-mapping: Inject
...
adapter:
  name: TimeStepAndPlot
  merge-with-user-defined-mapping: TimeStep
  merge-with-user-defined-mapping: Inject
  merge-with-predefined-mapping: VTKPlotVertexValue(result,getU,u)
```

As a result, we have a multiscale representation of the solution on each individual grid level. It is (almost) for free, as Peano's vertices are unique due to their combination of level and spatial position, i.e. these coarse vertices do exist anyway.

```
void myproject::Vertex::inject(const Vertex& fromVertex) {
  _vertexData.setU( fromVertex._vertexData.getU() );
}

void myproject::mappings::Inject::touchVertexLastTime(...) {
 logTraceInWith6Arguments( "touchVertexLastTime(...)", fineGridVertex, fineGridX, ... );

 if ( peano::grid::SingleLevelEnumerator::isVertexPositionAlsoACoarseVertexPosition(
  fineGridPositionOfVertex) ) {
  const peano::grid::SingleLevelEnumerator::LocalVertexIntegerIndex coarseGridPosition =
   peano::grid::SingleLevelEnumerator::getVertexPositionOnCoarserLevel
   (fineGridPositionOfVertex);

  coarseGridVertices[ coarseGridVerticesEnumerator(coarseGridPosition) ].inject(fineGridVertex);
 }

 logTraceOutWith1Argument( "touchVertexLastTime(...)", fineGridVertex );
}
```

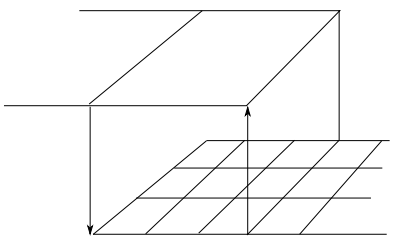
In our implementation, we use helper functions of the enumerator classes. Alternatively, we could have checked the integer array `fineGridPosiitionOfVertex` manually whether all entries are either 0 or 3.

> **Remark:** Several projects have exploited this multiscale representation for visualisation and data postprocessing. Whenever data in a reduced accuracy is sufficient (to create fancy pictures, e.g.), one can directly extract this data from the corresponding spacetree level rather than using the finest grid representation and postprocess these data.

4.1.5 Static adaptivity

An elegant way to realise adaptive solvers is to pick up the concept of FAC/MLAT from the multigrid community. The idea is very simple:

- We calculate an update of the solution for each and every cell in the sapcetree. This includes the refined ones.
- We interpolate hanging nodes from the next coarser levels. This can be done recursively, i.e. a 3:1 balancing of the tree is not required. Rippling does not happen.
- We inject the solution of a fine grid vertex to the coarser levels and thus overwrite the impact of the unknown update there.



One can read such a scheme as an overlapping domain decomposition: The coarser levels slightly overlap finer levels. Those coarse grid vertices that fall into a fine grid partition are treated as Dirichlet values (we may update them, but the update then is overwritten immediately). Their value is injected from finer grids—an ingredient we already have realised. Fine grid problems are handled as Dirichlet problems as well. Their boundary points either are real boundary points or hanging nodes. The value of hanging nodes in turn stems from coarser levels which closes the coupling of the coarse and fine grid problems.

We propose to introduce—mirroring `Inject`—a new mapping `InterpolateHangingNodes` that solely plugs into the generation of hanging nodes. This time, we use Peano's d-dimensional loops, as well as PDT's generated write operations that break up the OO encapsulation.

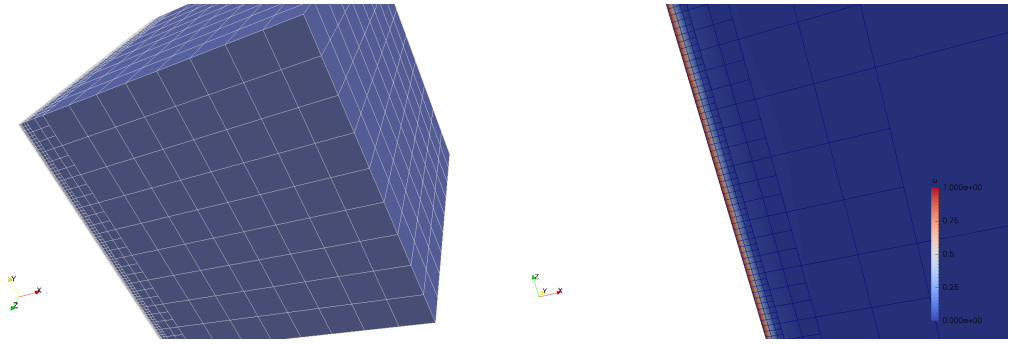
```
#include "myproject/VertexOperations.h"
#include "peano/utils/Loop.h"

void myproject::mappings::InterpolateHangingNodes::createHangingVertex(...) {
  logTraceInWith6Arguments( "createHangingVertex(...)", fineGridVertex, fineGridX, ... );

  double interpolatedValue = 0.0;
  dfor2(k)
    double weight = 1.0;
    for (int d=0; d<DIMENSIONS; d++) {
      if (k(d)==0) {
        weight *= 1.0 -(fineGridPositionOfVertex(d))/3.0;
      }
      else {
        weight *= (fineGridPositionOfVertex(d))/3.0;
      }
    }
    interpolatedValue = weight *coarseGridVertices[ coarseGridVerticesEnumerator(k)].getU();
  enddforx

  VertexOperations::writeU( fineGridVertex, interpolatedValue );
  fineGridVertex.copyCurrentSolutionIntoOldSolution();

  logTraceOutWith1Argument( "createHangingVertex(...)", fineGridVertex );
}
```

We reiterate that Peano's `matrixfree` toolbox provides helpers realising such typical interpolation tasks.

Finally, we have to setup an adaptive grid. We stick to a static, simple setup for the time being where the mesh is made very fine along the $x = 0$ plane and reasonably coarse everywhere else. Please note that you might have to adopt your time step size as well if you reduce the resolution along the heated up plate. Furthermore, if you visualise, please note that the default visualiser we used so far does not know anything about the hanging nodes' interpolation and thus plots them as zero values.

```
void myproject::mappings::CreateGrid::createInnerVertex(...) {
 if (coarseGridVerticesEnumerator.getLevel()<1) {
  fineGridVertex.refine();
 }
}

void myproject::mappings::CreateGrid::createBoundaryVertex(...) {
 if (coarseGridVerticesEnumerator.getLevel()<3 && fineGridX(0)<1e-8) {
  fineGridVertex.refine();
```

```
  }
  else if (coarseGridVerticesEnumerator.getLevel()<1) {
    fineGridVertex.refine();
  }
}
```



In the present example, it does not matter that some cells compute stencil evaluations that are never used: These are the cells where all adjacent vertices are refined. Due to the injection, the stencil evalution here is not necessary. As the stencils are very small and cheap to evaluate, this does not make a difference. For more complicated setups, it however might make a difference.

Obviously, it is not possible in the present case to use the cells' `isRefined()` operation to determine whether a stencil evaluation should be skipped or not. Refined cells next to an unrefined one have to be taken into account. Instead, we have to run over all adjacent vertices of a cell and determine their refinement state. Peano realises an or-based refinement—if any vertex adjacent to a cell carries the refinement bit, the cell is refined—so the required check is an or combination followed by a negation. In Peano, there's a class `peano::grid::aspects::VertexStateAnalysis`. It is worth to study this class; it offers most of such analysis routines already and makes the programming of multiscale algorithms more convenient.

4.1.6 Dynamic adaptivity

Given a statically adaptive grid, dynamic adaptivity is (technically) a minor improvement. We realise it in two steps:

1. We write the refinement criterion, and
2. we ensure that the a posteriori refinement initialises the correct data.

The first step is strongly application dependent. A simple criterion that seems to work sufficiently is to assign each vertex a new (non-persistent) value that holds the average value of the surrounding vertices. If this value differs significantly from the computed value in the point (which is kind of a smoothness analysis), we refine. For this, we change the vertex definition

```
Packed-Type: short int;


class myproject::dastgen::Vertex {
  parallelise persistent double u;
  discard double oldU;
  discard double averagedU;
};
```

and we also add additional read statements to the specification. The specification remains more or less unchanged, but we keep all the dynamic refinement in an additional refinement mapping.

```
vertex:
  dastgen-file: Vertex.def
  read scalar(double): U
  read scalar(double): OldU
  read scalar(double): AveragedU
  write scalar(double): U
  write scalar(double): AveragedU

...

event-mapping:
  name: RefineDynamically

...

adapter:
  name: TimeStep
  merge-with-user-defined-mapping: TimeStep
  merge-with-user-defined-mapping: Inject
  merge-with-user-defined-mapping: InterpolateHangingNodes
  merge-with-user-defined-mapping: RefineDynamically
```

Before we implement this additional mapping, we create a slightly more appealing problem setup in create grid:

```
void myproject::mappings::CreateGrid::createCell(...) {
  logTraceInWith4Arguments( "createCell(...)", fineGridCell, ... );

  fineGridCell.init( fineGridVerticesEnumerator.getCellCenter() );

  logTraceOutWith1Argument( "createCell(...)", fineGridCell );
}


void myproject::Cell::init(const tarch::la::Vector<DIMENSIONS,double>& x) {
  _cellData.setEpsilon( x(1) *x(2) + static_cast<double>(rand() % 100)/100.0 );
}
```

Furthermore, we make the start grid one level coarser along the $x = 0$ axis compared to the previous setup.

Our refinement criterion materialises in a very simple mapping relying on the simple stencil (here given for two dimensions)

$$\begin{bmatrix} 0 & 1 & 0 \\ -1 & 0 & 1 \\ 0 & -1 & 0 \end{bmatrix}$$

that we apply on the solution. This stencil determines the derivative in a point. For a first try, it often yields very reasonable refinement patterns—we refine where we observe a steep gradient.

```
void myproject::mappings::AdaptiveRefinementCriterion::touchVertexFirstTime(...) {
  logTraceInWith6Arguments( "touchVertexFirstTime(...)", fineGridVertex, ... );

  VertexOperations::writeAveragedU( fineGridVertex, 0.0 );
```

```
  logTraceOutWith1Argument( "touchVertexFirstTime(...)", fineGridVertex );
}


void myproject::mappings::AdaptiveRefinementCriterion::enterCell(...) {
  logTraceInWith4Arguments( "enterCell(...)", fineGridCell, ... );

  tarch::la::Matrix<TWO_POWER_D,TWO_POWER_D,double> A;

  A = 0.0, 1.0/4.0, 1.0/4.0, 0.0, 1.0/4.0, 0.0, 0.0, 0.0,
      -1.0/4.0, 0.0, 0.0, 1.0/4.0, 0.0, 1.0/4.0, 0.0, 0.0,
      -1.0/4.0, 0.0, 0.0, 1.0/4.0, 0.0, 0.0, 1.0/4.0, 0.0,
       0.0, -1.0/4.0, -1.0/4.0, 0.0, 0.0, 0.0, 0.0, 1.0/4.0,
      -1.0/4.0, 0.0, 0.0, 0.0, 0.0, 1.0/4.0, 1.0/4.0, 0.0,
       0.0, -1.0/4.0, 0.0, 0.0, -1.0/4.0, 0.0, 0.0, 1.0/4.0,
       0.0, 0.0, -1.0/4.0, 0.0, -1.0/4.0, 0.0, 0.0, 1.0/4.0,
       0.0, 0.0, 0.0, -1.0/4.0, 0.0, -1.0/4.0, -1.0/4.0, 0.0;

  tarch::la::Vector<TWO_POWER_D,double> uOld =
    VertexOperations::readOldU(fineGridVerticesEnumerator,fineGridVertices);

  const double h = fineGridVerticesEnumerator.getCellSize()(0);
  const double scaling = 1.0/h;
  tarch::la::Vector<TWO_POWER_D,double> averageUpdate = scaling *A *uOld;

  VertexOperations::writeAveragedU(
    fineGridVerticesEnumerator,fineGridVertices,
    VertexOperations::readAveragedU(fineGridVerticesEnumerator,fineGridVertices)
    + averageUpdate
  );

  logTraceOutWith1Argument( "enterCell(...)", fineGridCell );
}


void myproject::mappings::AdaptiveRefinementCriterion::touchVertexLastTime(...) {
  logTraceInWith6Arguments( "touchVertexLastTime(...)", fineGridVertex, fineGridX, ... );

  if (coarseGridVerticesEnumerator.getLevel()<6) {
    fineGridVertex.evaluateRefinementCiterion();
  }

  logTraceOutWith1Argument( "touchVertexLastTime(...)", fineGridVertex );
}
```

The magic final operation is kept very simple in this guide book:

```
void myproject::Vertex::evaluateRefinementCiterion() {
 if (
  getRefinementControl()==Records::Unrefined
  &&
  std::abs( _vertexData.getAveragedU() )>1.0
 ) {
   refine();
 }
```

```
}
```

> **Remark:** Almost all sophisticated refinement criteria are based upon a marker concept such that only a given ratio of the cells are refined per step. Such markers are non-trivial to implement as they require global sorting. It is thus convenient to use some binning approach. The `matrixfree` toolbox of Peano has a reference implementation that also works in an MPI environment.

For the second step, we have to ensure that all newly created cells and vertices are intialised correctly. For the cells, we might simply merge the `CreateGrid` mapping into our time stepping adapter. For the vertices, we have to plug into `createInnerVertex` where we initialse the vertex with the interpolated value of its coarse grid parents. Again, Peano offers toolboxes with routines that do so. However, you may also simply reprogram it using the dimension-specific for-loops of `Loop.h`. The routines are then very similar to the interpolation for hanging nodes. In this example, we do not reuse `CreateGrid`, but implement everything within the adaptivity criterion's mapping.

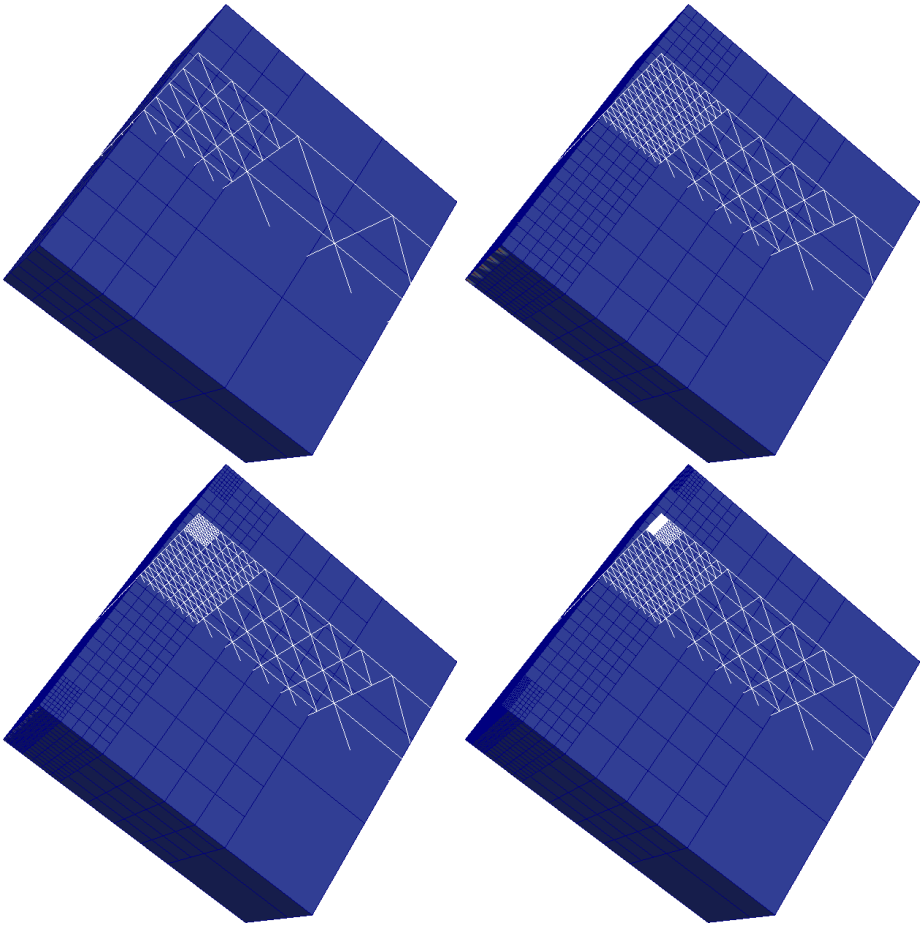
```
void myproject::mappings::AdaptiveRefinementCriterion::createCell( ... ) {
  logTraceInWith4Arguments( "createCell(...)", fineGridCell, ... );

  fineGridCell.init( fineGridVerticesEnumerator.getCellCenter() );

  logTraceOutWith1Argument( "createCell(...)", fineGridCell );
}


void myproject::mappings::AdaptiveRefinementCriterion::createInnerVertex( ... ) {
  logTraceInWith6Arguments( "createInnerVertex(...)", fineGridVertex, fineGridX, fineGridH, ... );

  double interpolatedValue = 0.0;
  dfor2(k)
    double weight = 1.0;
    for (int d=0; d<DIMENSIONS; d++) {
      if (k(d)==0) {
        weight *= 1.0 -(fineGridPositionOfVertex(d))/3.0;
      }
      else {
        weight *= (fineGridPositionOfVertex(d))/3.0;
      }
    }
    interpolatedValue = weight *coarseGridVertices[ coarseGridVerticesEnumerator(k)].getU();
  enddforx

  VertexOperations::writeU( fineGridVertex, interpolatedValue );
  fineGridVertex.copyCurrentSolutionIntoOldSolution();

  logTraceOutWith1Argument( "createInnerVertex(...)", fineGridVertex );
}
```

We finally modifiy our runner in two ways:

1. We make the runner also plot if the grid changes in a particular time step, and
2. we ensure that the time step size adopts to the grid structure, i.e. we implement adaptive time stepping.

```
int myproject::runners::Runner::runAsMaster(myproject::repositories::Repository& repository) {
 peano::utils::UserInterface userInterface;
 userInterface.writeHeader();

 repository.switchToCreateGridAndPlot();
 repository.iterate();

 const double initialDt = 1e-4;
 repository.getState().setTimeStepSize( initialDt );
 for (int i=0; i<10000; i++) {
   if (i%100==0 || !repository.getState().isGridStationary()) {
     repository.switchToTimeStepAndPlot();
   }
   else {
    repository.switchToTimeStep();
   }
   repository.iterate();
   double dt = initialDt *repository.getState().getMinimumMeshWidth()
               *repository.getState().getMinimumMeshWidth();
   repository.getState().setTimeStepSize( dt );
   logInfo(
    "runAsMaster(...)",
    "time step " << i << ": dt=" << dt << ", h_min=" <<
    repository.getState().getMinimumMeshWidth() );
 }

 repository.logIterationStatistics();
 repository.terminate();

 return 0;
}
```

The screenshots below illustrate how the grid evolves in the first four time steps of the simulation:



4.1.7 Global data

For most solvers, some kind of global information is required. Such data could be

- a global time step size,
- a residual norm,
- a counter for file plots,
- and so forth.

We could hold such information as static/global data in our code or as an attribute in a mapping. Both options are not particular appealing. Global variables contradict our notion of nice decomposed programming. Furthermore, I have to anticipate that Peano automatically replicates mappings on a multithreaded system. So we have to be very carefully if we access global data within the mapping's routines to avoid race conditions. Attributes of mappings suffer from the same issues. Furthermore, if mappings are replicated, they are replicated as well: instead of data races we have to ensure that data is correctly replicated and then fused again. Finally, if we run on a distributed memory machine, multiple instances of our application will execute because of MPI's SPMD paradigm, and we have to somehow manually ensure that global data is kept consistent among all the ranks.

Peano's concept is to model all global data within the `State.def` file. Again, we can here also anticipate which data later on is to be distributed between different ranks. We have discussed this issue before. We also have clarified that the standard pattern is to set global data in the runner through the State object and then to use this data alter on.

```
Packed-Type: short int;


class myproject::dastgen::State {
  persistent parallelise double dt;
};
```

To use a state, I recommend to make each mapping extract all required data from the state in `beginIteration()`. The idea is as follows (for parallel runs later on): a global algorithm control routing sets global data in the state. This state is distributed among all ranks. All ranks run a particular set of mappings through one adapter. Each mapping get the required data in `beginIteration()`. Later on, we will can tune exactly when the state data is exchanged.

As copying attributes from the `State` object is laborious, a standard pattern is to make each mapping hold a copy of the whole state:

```
class myproject::mappings::TimeStep {
  private:
    /**
     *Logging device for the trace macros.
     */
    static tarch::logging::Log _log;

    State _localState;
    ...
};
```

It is important that this state is a local one and is overwritten in each run, as the runner outside might have changed it.

```
void myproject::mappings::TimeStep::beginIteration(
  myproject::State& solverState
) {
  logTraceInWith1Argument( "beginIteration(State)", solverState );

  _localState = solverState;
  _localState._clearAttributes(); // see remark below

  assertion2( _timeStepSize>0.0, _timeStepSize, solverState.toString() );

  logTraceOutWith1Argument( "beginIteration(State)", solverState);
}
```

Once you need data from the state, you can extract this data within the mapping via `solverState.getTimeStepSize()` in the present example.

We finally have to clarify how to collect global data, i.e. how to realise the data flow the other way round. For this, we also use the copy of the state (this is only one implementation pattern and again we could fill back data manually attribute-by-attribute. It is just more convenient this way). Usually, we first introduce an operation `State::clearAttributes()` that resets all data, and we offer operations alike `State::updateResidual(double myParam)` that allow us to collect information within a `State` instance.

Within the mapping, we then use this operation to collect the data from all grid entities:

```
  ...
  _localState.updateResidual(myValue);
```

```
  ...
```

Obviously, the solution so far is not worth a penny, as the mappings work with their own local copy of state. The solution now is to introduce a new operation

```
void State::merge( const State& otherState ) {
  _stateData.setMyLocalAttribute(
    _stateData.getMyLocalAttribute() +
    otherState._stateData.getMyLocalAttribute()
  );
}
```

We call this operation in `endIteration()` to basically backplay the data from the local copy into the global state:

```
void myproject::mappings::TimeStep::endIteration(
  myproject::State& solverState
) {
  solverState.merge( _localState );
}
```

Finally, we can read the state in the global runner and plot some data to the terminal, e.g.:

```
int myproject::runners::Runner::runAsMaster(myproject::repositories::Repository& repository) {
  ...

  const double initialDt = 1e-4;
  repository.getState().setTimeStepSize( initialDt );
  for (int i=0; i<10000; i++) {
    if (i%100==0 || !repository.getState().isGridStationary()) {
      repository.switchToTimeStepAndPlot();
    }
    else {
     repository.switchToTimeStep();
    }
    repository.getState()._clearAttributes();
    repository.iterate();
    logInfo( "runAsMaster(...)", "my_fancy_global_attribute=" <<
    repository.getState().getMyFancyGloalAttribute() );

    ...
  }

  ...
}
```

Please note that you might have to clear also the global state from time to time, which is done in the snippet above right before we call `iterate`.

> **Remark:** The whole process seems to be a little bit of overengineering just to get a few global variables right. The elegance of the solution becomes obvious when we use multiple MPI ranks and multiple threads. There, we rely on exactly the same `merge` operation to distribute a global state first to all ranks and threads before we fuse all distributed states together again into one object.

Further reading

- Muntean, Ioan Lucian, Mehl, Miriam, Neckel, Tobias and Weinzierl, Tobias (2008). *Concepts for Efficient Flow Solvers Based on Adaptive Cartesian Grids*. In High Performance Computing in Science and Engineering, Garching 2007. Wagner, Siegfried, Steinmetz, Matthias, Bode, Arndt Brehm, Matthias Berlin Heidelberg New York: Springer.
- Weinzierl, Tobias and Köppl, Tobias (2012). *A Geometric Space-time Multigrid Algorithm for the Heat Equation*. Numerical Mathematics: Theory, Methods and Applications 5(1): 110-130.
- Mehl, Miriam, Weinzierl, Tobias and Zenger, Christoph (2006). *A cache-oblivious self-adaptive full multigrid method*. Numerical Linear Algebra with Applications 13(2-3): 275-291.

4.2 Matrix-free multigrid

Time: 1–2 days.
Required: Chapter 2. It is advantageous if the reader has studied the implementation of the heat equation before.

In this section, we sketch how to solve the convection-diffusion equation

$$-\nabla(\epsilon\nabla)u + \nabla(v\ u) = f \qquad \text{with } v \in R^d, \epsilon \in R^{d\times d}$$

with various geometric multigrid solvers. *epsilon* is a diagonal matrix with entries ϵ_1, ϵ_2 or $\epsilon_1, \epsilon_2, \epsilon_3$, respectively. Our realisation is based upon a few design decisions:

1. The solvers use the spacetree as computational grid.
2. We use a finite element formalism with d-linear shape functions.
3. The material parameters ϵ and v are given per cell.

For the implementation, we use Peano's `matrixfree` toolbox. This is a tiny little collection of helper classes to work with stencils and local assembly matrix. It is neither fast, i.e. computationally mature, nor can it cope with real stencil libraries, but it does the job.

4.2.1 Setup

We start with Peano's PDT and generate a project. We also link Peano's sources into the project and unzip the `matrixfree` toolbox.

```
java -jar pdt.jar --create-project multigrid multigrid
ln -s mypath/src/peano
ln -s mypath/src/tarch
cp mypath/tarballs/toolboxes/matrixfree.tar.gz .
tar -xzvf matrixfree.tar.gz
```

I recommend to hold `matrixfree` parallel to the `multigrid`, `peano` and `tarch` directory. As soon as we use such a toolbox, we also might have to adopt our makefile accordingly: we have to add the matrixfree directory to the find pathes when we build up a list of source codes. Furthermore, you might have to add an additional search directory. If you place your toolbox parallel to `peano` and `tarch`, this however should not be necessary. Here's the corresponding excerp from the makefile:

```
files.mk:
    touch files.mk
    echo -n SOURCES= > files.mk
    find -H $(PEANO_HOME)/peano -name *.cpp | awk {␣printf␣"%s␣",␣$$0␣} >> files.mk
    find -H $(PEANO_HOME)/tarch -name *.cpp | awk {␣printf␣"%s␣",␣$$0␣} >> files.mk
    find -H $(PEANO_HOME)/matrixfree -name *.cpp | awk {␣printf␣"%s␣",␣$$0␣} >> files.mk
    find $(PROJECT_HOME) -name *.cpp | awk {␣printf␣"%s␣",␣$$0␣} >> files.mk
```

We next add our material parameters to the `Cell.def` file

```
Packed-Type: short int;

Constant: DIMENSIONS;
```

```
class multigrid::records::Cell {
  persistent parallelise double epsilon[DIMENSIONS];
  persistent parallelise double v[DIMENSIONS];
};
```

and create a simple first specification file:

```
component: Multigrid

namespace: ::multigrid

vertex:
  dastgen-file: Vertex.def

cell:
  dastgen-file: Cell.def

state:
  dastgen-file: State.def

event-mapping:
  name: CreateGrid

event-mapping:
  name: PlotCells

adapter:
  name: CreateGrid
  merge-with-user-defined-mapping: CreateGrid
  merge-with-user-defined-mapping: PlotCells
```

We run this specification file through the PDT

```
java -jar <mypath>/pdt.jar --generate-gluecode multigrid/project.peano-specification multigrid
```

and implement both the plotter and the creational mapping such that we have a few characteristic setups. It might however make sense to validate that make passes before we start any PDE-specific coding:

```
make -f multigrid/makefile
```

We next introduce an operation

```
matrixfree::stencil::ElementWiseAssemblyMatrix multigrid::Cell::getElementsAssemblyMatrix(
  const tarch::la::Vector<DIMENSIONS,double>& h
) const {
  matrixfree::stencil::ElementWiseAssemblyMatrix result;

  const matrixfree::stencil::Stencil laplacianStencil =
    matrixfree::stencil::getLaplacian(_cellData.getEpsilon(), h);

  return matrixfree::stencil::getElementWiseAssemblyMatrix(laplacianStencil);
}
```

which returns the $\mathbf{R}^{2^d \times 2^d}$ local system matrix. The method sets up the stencils that correspond to a regular Cartesian system given the mesh size h and the material parameters. Here, also the convective term has to be handled. Finally, it uses `getElementWiseAssemblyMatrix` to extract the actual matrix from this stencil.

> **Remark:** Peano supports all stencil/linear algebra operations for complex values.

4.2.2 Jacobi smoother

The basic building block of all of our solvers is a simple Jacobi smoother working on adaptive grids as well as on multiple scales. Its realisation is an extension of the solver in 4.1. To make it work without the assembly of any global matrix, we associate each vertex a residual value as well as the actual value. There are a few other features such as boundary properties or some level analysis that we either use later on or we pass to the used toolboxes. Their exact semantics and rationale have to be taken from the source code.

```
Packed-Type: short int;


class multigrid::records::Vertex {
  /**
   *Solution
   */
  persistent parallelise double u;

  /**
   *Rhs
   */
  persistent parallelise double f;

  /**
   *Residual
   */
  persistent parallelise double r;

  /**
   *Diagonal element
   */
  persistent parallelise double d;

  enum VertexType {
    Unknown, Dirichlet, Neumann
  };

  persistent VertexType vertexType;

  // some other attributes
};
```

Besides a proper initialisation of the vertices, we extend the specification similar to the heat equation.

```
component: Multigrid
```

```
namespace: ::multigrid

vertex:
  dastgen-file: Vertex.def
  read scalar(double): U
  read scalar(double): R
  read scalar(double): D
  read scalar(double): F
  write scalar(double): U
  write scalar(double): R
  write scalar(double): D

...

adapter:
  name: CreateGrid
  merge-with-user-defined-mapping: CreateGrid
  merge-with-user-defined-mapping: PlotCells
  merge-with-predefined-mapping: VTKPlotVertexValue(u,getU,u)
```

Once this code framework passes (only minor technical helper routines have to be implemented, but by now this should be straightforward to any user), we can introduce a new mapping/adapter `JacobiSmoother`, call this one a couple of hundred times in the runner and plot the result file then. For the latter, it makes sense to use a predefined plotter. The interesting new aspects can be found in three routines of the smoother. The design of the code realises matrix-free element-wise mat-vecs 1:1:

1. Each vertex carries a residual and a diagonal value attribute. They are cleared whenever the vertex is read the very first time in a traversal.

   ```
   void multigrid::mappings::JacobiSmoother::touchVertexFirstTime(...) {
     logTraceInWith6Arguments( "touchVertexFirstTime(...)", ... );

     fineGridVertex.clearAccumulatedAttributes();

     logTraceOutWith1Argument( "touchVertexFirstTime(...)", fineGridVertex );
   }

   // in Vertex files

   void multigrid::Vertex::clearAccumulatedAttributes() {
     _vertexData.setR(0.0);
     _vertexData.setD(0.0);
   }
   ```

   Our concept is that we accumulate the diagonal element d and the residual r within these (temporary) attributes per vertex throughout the traversal.

2. When we use a vertex for the very last time, we may thus update the unknown according to the values of the residual and the diagonal value. This is the actual Jacobi smoothing step:

$$u^{(new)} \leftarrow u^{(old)} + \omega \frac{1}{d} r$$

```
void multigrid::mappings::JacobiSmoother::touchVertexLastTime(...) {
  const bool hasUpdated = fineGridVertex.performJacobiSmoothingStep( omega );

  ...
}

// in Vertex files

double multigrid::Vertex::getResidual() const {
  return _vertexData.getF() + _vertexData.getR();
}


void multigrid::Vertex::performJacobiSmoothingStep( double omega ) {
  if (_vertexData.getVertexType()== Records::Unknown) {
    assertion1( _vertexData.getD()>0.0, toString() );
    assertion2( omega>0.0, toString(), omega );
    _vertexData.setU( _vertexData.getU() + omega / _vertexData.getD() *getResidual() );
  }
}
```

Please note that the residual here is modelled as sum of the right-hand side and the accumulated value (cf. helper operation `getResidual()`). We anticipate the minus from the definition

$$r = f - Au$$

already in the accumulation, i.e. sum up $-Au$ in the vertex attribute r. An additional pitfal is this context stems from the usage of a finite element method. It implies that we have to ensure that f is scaled with h^d (h being the local mesh width), which is something we typically do already in the initialisation.

> **Remark:** It is obvious that the Jacobi update scheme may only update inner unrefined vertices as well as Neumann boundary points. Peano does not offer vertex types. It distinguishes only inner and outer vertices. As we need more flags than inside and outside (at least two boundary flags), we have to offer these on our own[2].

In the present implementation, we make the Jacobi update return a flag that indicates whether a fine grid update has been done or not. Most codes will like to track global data such as a global residual or the maximum value of the solution and can use this flag to decide whether a vertex contributes to global data or not. Please study the accompanying source code for details.

3. The most complicated part is obviously the evaluation of the local mat-vec contributions. Here, we rely on the cell's `getElementsAssemblyMatrix` as well as operations from `VertexOperations`. All operations in this class are generated by the PDT and extract from `enterCell`'s vertices vectors: you hand in all fine grid data and extract a vector of all u values, e.g. The other way round is supported as well. The operations within this helper class are all generated because of the read and write statements in the specification.

```
#include "multigrid/VertexOperations.h"
```

[2]Up to early 2016, Peano had a three-valued logic with inner, outer and boundary vertices. I removed this from the kernel as most applications need way more vertex types anyway.

```
void multigrid::mappings::JacobiSmoother::enterCell(...) {
  logTraceInWith4Arguments( "enterCell(...)", fineGridCell, ... );

  const tarch::la::Vector<TWO_POWER_D,double> u =
    VertexOperations::readU( fineGridVerticesEnumerator, fineGridVertices );
  const tarch::la::Vector<TWO_POWER_D,double> dOld =
    VertexOperations::readD( fineGridVerticesEnumerator, fineGridVertices );
  const tarch::la::Vector<TWO_POWER_D,double> rOld =
    VertexOperations::readR( fineGridVerticesEnumerator, fineGridVertices );
  const matrixfree::stencil::ElementWiseAssemblyMatrix A =
    fineGridCell.getElementsAssemblyMatrix( fineGridVerticesEnumerator.getCellSize() );

  tarch::la::Vector<TWO_POWER_D,double> r = rOld -A *u;
  tarch::la::Vector<TWO_POWER_D,double> d = dOld + tarch::la::diag(A);

  VertexOperations::writeR( fineGridVerticesEnumerator, fineGridVertices, r );
  VertexOperations::writeD( fineGridVerticesEnumerator, fineGridVertices, d );

  logTraceOutWith1Argument( "enterCell(...)", fineGridCell );
}
```

4.2.3 Environment

The changes in the environment are straightforward once the smoother is in place:

1. We extend the runner such that it switches to the Jacobi smoother once the grid is set up and triggers a fixed number of iterations then.
2. We add a logging device to the runner

```
class multigrid::runners::Runner {
  private:
    static tarch::logging::Log _log;
    ...
};
```

and make the innermost loop plot residual and other statistics after each grid traversal:

```
repository.switchToJacobiAndPlot();
for (int i=0; i<100; i++) {
  repository.iterate();

  logInfo(
    "runAsMaster(...)",
    "#vertices=" << repository.getState().getNumberOfInnerLeafVertices() <<
    ",|res|_2=" << repository.getState().getResidualIn2Norm() <<
    ",|res|_max=" << repository.getState().getResidualInMaxNorm() <<
    ",|u|_L2=" << repository.getState().getSolutionInL2Norm() <<
    ",|u|_max=" << repository.getState().getSolutionInMaxNorm() <<
    ",#stencil-updates=" << repository.getState().getNumberOfStencilUpdates()
  );

  repository.getState().clearAccumulatedAttributes();
}
```

3. To make the code work, we augment the state with the corresponding fields

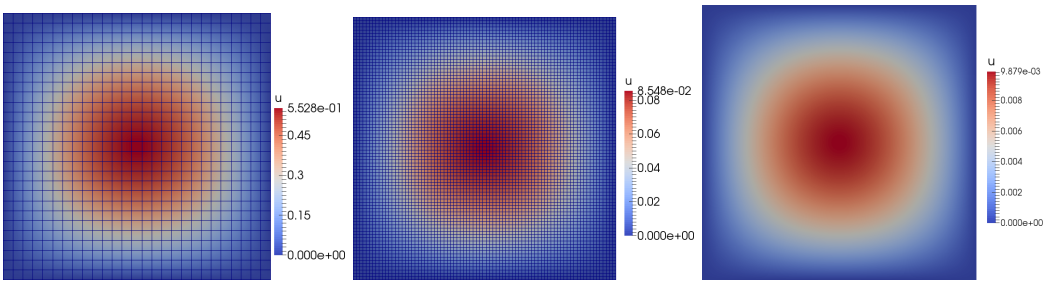
```
Packed-Type: short int;

class multigrid::records::State {
  // Stores squared value, i.e. apply sqrt before returning it
  persistent parallelise double residual2Norm;
  persistent parallelise double residualMaxNorm;
  // Stores squared value, i.e. apply sqrt before returning it
  persistent parallelise double solutionL2Norm;
  persistent parallelise double solutionMaxNorm;
  persistent parallelise double numberOfStencilUpdates;
};
```

   and realise the corresponding setters and getters. The design of the state methods (notably a method `clearAccumulatedAttributes()` in combination with `merge`) follows recommendations motivated in Section 5.1. For the time being, we do not discuss them further.

4. Finally, we extend the `main` such that it can read in a well-suited relaxation parameter from the command line. It then sets this relaxation parameter (the static field) in the mapping:

```
multigrid::mappings::JacobiSmoother::omega = atof( argv[2] );
```

We may run this code for example the well-known Poisson benchmark ($\epsilon = 1, v = 0, f = d\ \pi^2 \prod_i \sin(x_i \pi)$), and observe the well-known dependency of Jacobi on the mesh width (below grids with two, three or four compute grid levels):



4.2.4 Dynamically adaptive Jacobi mit FAC

We next implement a dynamically adaptive Jacobi solver that implements the FAC scheme. Its fundamental idea is that the Jacobi smootherr is applied on each and every grid level in parallel. Any hanging node's value is interpolated from coarser grids. While we update all grid levels, we do overwrite coarse vertices with fine grid values for all vertices that do exist on finer grid resolutions as well. We inject the solution from the fine grids onto coarser grids. This first solver is capable to handle dynamically adaptive grids with arbitrary refinement pattern. It also is a preliminary exercise how to implement multigrid full approximation storage (FAS).

We extend the grid setup slightly such that it creates a very coarse grid even if we prescribe a fine minimum mesh size. Next, we validate that `enterCell` evaluates the stencil on each grid level. This leaves two tasks: interpolatation and injection. The injection is basically the same we have used in the heat equation before.

```
void multigrid::mappings::JacobiSmoother::touchVertexLastTime(...) {
 // see code snippets introduced before

 if (
```

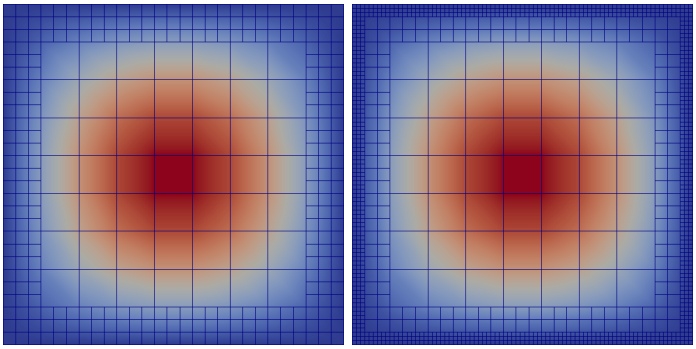
```
    peano::grid::SingleLevelEnumerator::isVertexPositionAlsoACoarseVertexPosition(
      fineGridPositionOfVertex
    )
  ) {
    const peano::grid::SingleLevelEnumerator::LocalVertexIntegerIndex coarseGridPosition =
      peano::grid::SingleLevelEnumerator::getVertexPositionOnCoarserLevel
      (fineGridPositionOfVertex);
    coarseGridVertices[ coarseGridVerticesEnumerator(coarseGridPosition) ].inject(fineGridVertex);
  }
}


// in the vertex

void multigrid::Vertex::inject(const Vertex& fineGridVertex) {
  _vertexData.setU( fineGridVertex._vertexData.getU() );
}
```

> **Remark:** The `matrixfree` toolbox offers a type `solver::Smoother` that realises a Jacobi smoother that automatically tracks different residual and solution norms. In the present example we do not use this smoother while we use the toolbox's multigrid class. This is kind of inconsistent. It would probably be better to use the smoother as well.

The interpolation follows the idea of the heat equation solver, too. However, we propose to rely on a premanufactured interpolation operation from the `matrixfree` toolbox. For this, we make our smoother mapping hold an instance of `matrixfree::solver::Multigrid _multigrid`. This object offers us an operation `getDLinearInterpolatedValue`:

```
void multigrid::mappings::JacobiSmoother::createHangingVertex(...) {
  logTraceInWith6Arguments( "createHangingVertex(...)", ... );

  fineGridVertex.setU(
    _multigrid.getDLinearInterpolatedValue(
      VertexOperations::readU( coarseGridVerticesEnumerator, coarseGridVertices ),
      fineGridPositionOfVertex
    )
  );

  logTraceOutWith1Argument( "createHangingVertex(...)", fineGridVertex );
}
```

Obviously, the vertex requires an additional `setU( double )` operation to make this snippet work. Its implementation is trivial. This is the only additional extension required. If we adopt the setup, we might solve the equation on grids alike below where the mesh is resolved up to the finest level along the boundaries. The latter is a proper choice for many problems, though inadequate and thus only a proof of concept for the present case:



Remark: On some machines, the resulting code will fail in `Assert` mode as the plotters complain about nan for the right-hand side. In the release mode, i.e. without assertions, no complaint is raised, but some visualisation software might complain about the nans. To fix this issue with the plotting, it is important to set the right-hand side f to zero for hanging nodes and for boundary nodes. We propose to introduce a `clearF()` operation and to call it prior to any plotting (in the `CreateGrid` mapping for example) for hanging and boundary vertices.

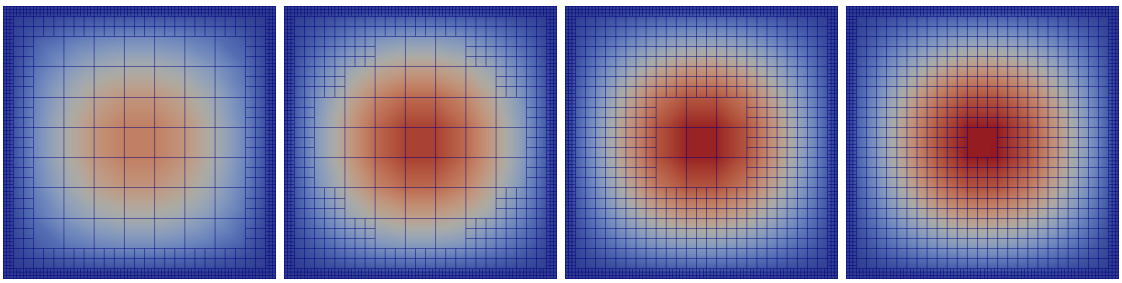
We close our discussion on the Jacobi smoother with the introducton of a dynamic refinement criterion. For this, we rely on a predefined refinement criterion offered with the matrixfree toolbox. We use `LinearSurplusRefinementCriterionWithFixedMeshSizes` from `matrixfree::adaptivitycriteria`. There is an extensive documentation how to use it in its superclass' header, so we do not reiterate this here. Instead we just show some of the added code:

```
multigrid::mappings::RefinementCriterion::RefinementCriterion():
  _refinementCriterion(
    0.1, // refinementPercentage,
    0.0, // deletePercentage,
    0.5, // minimumMeshSize,
    0.5 // maximumMeshSize
  ) {
}

void multigrid::mappings::RefinementCriterion::touchVertexFirstTime(...) {
  VertexOperations::writeLinearSurplus(fineGridVertex,0.0);
}

void multigrid::mappings::RefinementCriterion::enterCell(...) {
  VertexOperations::writeLinearSurplus(
    fineGridVerticesEnumerator,
    fineGridVertices,
    _refinementCriterion.getNewLinearSurplus(
      VertexOperations::readU(fineGridVerticesEnumerator,fineGridVertices),
      VertexOperations::readLinearSurplus(fineGridVerticesEnumerator,fineGridVertices)
    )
  );
}

void multigrid::mappings::RefinementCriterion::touchVertexLastTime(...) {
  if ( fineGridVertex.isInside() ) {
    const tarch::la::Vector<TWO_POWER_D_TIMES_D,double > coarseGridLinearSurplus =
```

```
      VertexOperations::readLinearSurplus(coarseGridVerticesEnumerator, coarseGridVertices)
      +
      _refinementCriterion.getLinearSurplusContributionFromFineGrid(
        VertexOperations::readLinearSurplus( fineGridVertex ),
        fineGridVertex.getRefinementControl()==Vertex::Records::Unrefined,
        fineGridPositionOfVertex
      );

    VertexOperations::writeLinearSurplus( coarseGridVerticesEnumerator,
      coarseGridVertices, coarseGridLinearSurplus );

    switch (
      _refinementCriterion.analyse(
        VertexOperations::readLinearSurplus(fineGridVertex),
        fineGridVertex.getRefinementControl()==Vertex::Records::Refined,
        fineGridVertex.getRefinementControl()==Vertex::Records::Unrefined,
        fineGridH
      )
    ) {
      case matrixfree::adaptivitycriteria::LinearSurplusRefinementCriterion::Refine:
        fineGridVertex.refine();
        break;
      case matrixfree::adaptivitycriteria::LinearSurplusRefinementCriterion::Delete:
      case matrixfree::adaptivitycriteria::LinearSurplusRefinementCriterion::NoAction:
        break;
    }
  }
}
```

The criterion's idea is to evaluate a stencil that allows us to compare the linear interpoland of the solution within a vertex to the actual vertex's solution value. We anticipate what would happen if we could remove a particular vertex alone. The other way round, we assume that vertices with a big difference are critical, and we would benefit a lot if we would refine around this vertex.

The offered criterion class bucket sorts the linear surplus values. This is not exact, but it does not require a real sort being in $\mathcal{O}(n \log n)$—actually nothing is sorted, but each bucket is flagged (refine, coarse, do nothing). If a vertex falls into a particular bucket, the criterion returns the corresponding flag. More sophisticated criteria might yield better approximation patterns, but this generic one works surprisingly well.



This code marks up to ten percent of the vertices for refinemenet and consequently refines all the surrounding cells given that the residual in the vertices underruns the given threshold `_convergenceThreshold`. The latter magic constant is set to 10^{-2} for the plots above.

Now, one simple feature is worth trying: We have so far always used a predefined visualisation routine that visualises the finest grid of a simulation. Among the set of standard visualisation routines also is a plotter that plots the individual levels of a grid. Given that we inject the solution

to coarse levels anyway, this allows us to visualise all data in a multilevel fashion. To use it, we again extend our specification, regenerate the glue code and recompile.

```
adapter:
  name: JacobiAndPlot
  merge-with-user-defined-mapping: CreateGrid
  merge-with-user-defined-mapping: JacobiSmoother
  merge-with-user-defined-mapping: PlotCells
  merge-with-predefined-mapping: VTKPlotVertexValue(u,getU,u)
  merge-with-predefined-mapping: VTKPlotVertexMultilevelValue(multiscaleU,getU,u)
```

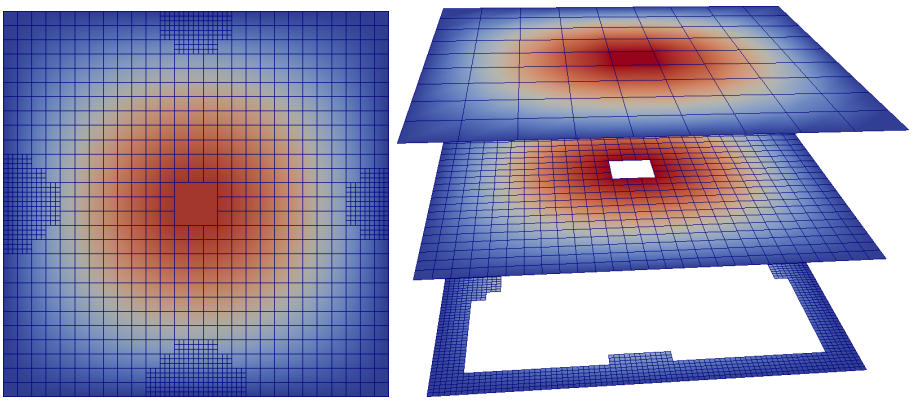
Some minor remarks on proper initialisation that are often encountered shall close the discussion. First, the presented refinement criterion does not properly refine along the domain's boundary, as it does not distinguish whether a stencil is applied along the boundary. In practice, it is reasonable to ensure that the boundary is always refined at least as fine as the vertices next to the boundary. In practive, it is reasonable to avoid hanging vertices along the boundary. This can be achieved if we check within each cell whether there is a boundary vertex and whether one vertex is refined. If both properties hold, all unrefined boundary vertices have to be refined. The class `peano::grid::aspects::VertexStateAnalysis` provides generic helper routines to realise this behaviour with a few lines:

```
#include "peano/grid/aspects/VertexStateAnalysis.h"
#include "peano/utils/Loop.h"

void multigrid::mappings::RefinementCriterion::enterCell(...) {
  ...
  if (
    fineGridCell.isRefined()
    &&
    peano::grid::aspects::VertexStateAnalysis::doesOneVertexCarryRefinementFlag(
      fineGridVertices, fineGridVerticesEnumerator, Vertex::Records::Unrefined
    )
  ) {
    bool isOneVertexABoundaryVertex = false;
    dfor2(k)
      isOneVertexABoundaryVertex |=
        fineGridVertices[ fineGridVerticesEnumerator(k) ].isBoundary();
    enddforx
    if (isOneVertexABoundaryVertex) {
      dfor2(k)
        if (fineGridVertices[ fineGridVerticesEnumerator(k) ].getRefinementControl()
          ==Vertex::Records::Unrefined) {
          fineGridVertices[ fineGridVerticesEnumerator(k) ].refine();
        }
      enddforx
    }
  }

}
```

Second, many codes require a proper balancing. As Peano relies on three-partitioning, a 3:1 balancing is something many codes would like to have: No two neighbouring cells differ in their maximum refinement level by more than one. This can be realised manually within a mapping. Or you may decide to merge the predefined mapping `GridBalancer3to1` into your adapters.

Finally, any dynamically adaptive code should properly initialise new vertices. In the present example, a proper initialisation could rely on d-linear interpolation again. For multigrid, higher order schemes are desireable (though in practice I again observed that linear interpolation is sufficient), but higher order interpolation then has to be realised manually with helper variables or some additional code applying the stencil to newly generated vertices. We stick to the linear case here:

```
void ...::createInnerVertex(...) {
  fineGridVertex.setU(
    _multigrid.getDLinearInterpolatedValue(
      VertexOperations::readU( coarseGridVerticesEnumerator, coarseGridVertices ),
      fineGridPositionOfVertex
    )
  );
}
```



The right image above presents the result of the multilevel plotter: Each grid resolution level is written to a separate file. It is obvious that finer grid solution approximations overwrite coarser vertex values, while the coarse values determine the hanging node values. We also see that the grid is refined towards the boundary here.

4.2.5 Multigrid

We start with some metaphilosophical considerations. Multigrid algorithms work in two types of spaces: the ansatz (discretisation) space holding the solution and a set of correction spaces. The latter either are updated all at once (additive multigrid) or one after another (multiplicative) before their contribution is added back to the actual representation. In the present discussion, we restrict ourselves to geometrically inspired multigrid where both the correction and the discretisation space are spanned by the spacetree's grid levels. On the previous pages, we have realised a Jacobi directly within the discretisation space without assembling any bigger matrix. Obviously, such an approach also works for the correction equations.

At first glance, we then however need two variable sets per vertex: one set for the actual discretisation and one for the correction equations. As long as we work on regular grids only, there is no need to store different types of variables within the vertices: refined vertices hold correction equations, fine grid vertices hold the discretisation. This approach becomes problematic for adaptive grids. Still, unrefined vertices of the fine grid hold the discretised PDE. On the same level of an arbitrary unrefined level, there might however also be an area that is refined further. Here, we solve a correction equation. If we realised both the correction equation and the discretisation with the same set of variables, what would the right vertex values at the transition

between unrefined and correction areas be? We have to distinguish carefully in each cell which type of equation we are solving, unless ...

... we solve also the correction equation in a discretisation space. This is the idea of full approximation storage (FAS) which fits perfectly to our injection that we realised anyway. Instead of solving

$$A_{3h} e = \hat{f}$$

in areas that are refined further and thus hold a correction to the actual solution iterate, we solve

$$\begin{aligned} A_{3h}\left(Iu_h + e\right) &= \hat{f} + A_{3h} I u_h \qquad \text{added an additional term on both sides} \\ &= Rr_h + RA_h PIu_h \\ &= R\left(r_h + A_h PIu_h\right) \\ &= R\left(f_h - A_h u_h + A_h PIu_h\right) \\ &= R\left(f_h - A_h\left(id - PI\right)u_h\right) \\ &=: R\left(f_h - A_h \hat{u}_h\right) =: R\hat{r}_h \end{aligned}$$

i.e. work with a coarsened solution as input to the correction solve which is a formalism introduced by Griebel as HTMG.

So we can stick to our injections of the solution. In each traversal, we have to determine the hierarchical surplus in each vertex $\hat{u} = u - PIu_h$ which is simple, as Iu_h is available anyway. This hierarchical transform is a temporary helper. We can throw it away after the traversal again. We can model it as `discard` in the `Vertex.def`.

Once we have $\hat{u}$, we can simultaneously determine the residual r as well as its hierarchical counterpart $\hat{r}$. With the residual r, we update the solution if we have to—a detail subject of discussion in a minute when we finalise whether to realise a multiplicative or additive scheme. The hierarchical residual $\hat{r}$ in turn is not used to update any solution directly. It is restricted to the next coarser level to yield a right-hand side there.

We finally observe that we may not just update the solution with the residual. If we did that we would loose the knowledge which correction to prolong to the next finer level. We therefore store the updates in another helper variable and use this update to prolong it to a finer level later on.

> **Remark:** The update helper variable is not the only valid choice to distinguish coarse grid updates from fine grid representation. It is obviously that we also might store the hierarchical surplus persistently and then reconstruct the nodal value with the relation $u_h = \hat{u}_h + Pu_{3h}$. This is the implementation variant discussed in (Weinzierl:09), while a more detailed analysis of required helper variables for additive multigrid can be found in (Reps:16).

```
class multigrid::records::Vertex {
  persistent parallelise double u; // Solution
  persistent parallelise double f; // Rhs
  discard parallelise double r; // Residual
  discard parallelise double d; // Diagonal element
  discard parallelise double hierarchicalU; // Hierarchical solution
  discard parallelise double hierarchicalR; // Hierarchical residual
  persistent parallelise double uUpdate; // Update of solution
  discard parallelise double linearSurplus[DIMENSIONS];

  enum VertexType {
    Unknown, Dirichlet, Neumann
  };
  persistent VertexType vertexType;
};
```

The hierarchical surplus has to be determined in `touchVertexFirstTime` where we also clear the temporary variables. I decided to realise the generic multigrid operations in a mapping of its own. It is called `HierarchicalTransformAndRHSRestriction`. Once more, we rely on an instance of the Multigrid class.

```
void
multigrid::mappings::HierarchicalTransformAndRHSRestriction::createHangingVertex(...) {
 fineGridVertex.clearHierarchicalValues();
}

void multigrid::mappings::HierarchicalTransformAndRHSRestriction::touchVertexFirstTime(...) {
 fineGridVertex.clearHierarchicalValues();

 if ( fineGridVertex.isInside() ) {
   const tarch::la::Vector<TWO_POWER_D,double > u_3h =
     VertexOperations::readU(coarseGridVerticesEnumerator,coarseGridVertices);
   const tarch::la::Vector<TWO_POWER_D,double > e_3h =
     VertexOperations::readUUpdate(coarseGridVerticesEnumerator,coarseGridVertices);
   const double Pu_3h =
     _multigrid.getDLinearInterpolatedValue(u_3h,fineGridPositionOfVertex);
   const double Pe_3h =
     _multigrid.getDLinearInterpolatedValue(e_3h,fineGridPositionOfVertex);

   fineGridVertex.correctU(Pe_3h);
   fineGridVertex.determineUHierarchical(Pu_3h);

   if (fineGridVertex.getRefinementControl()!=Vertex::Records::Unrefined) {
     fineGridVertex.clearF();
   }
 }

 fineGridVertex.clearAccumulatedAttributes();
}

void multigrid::Vertex::determineUHierarchical(double Pu_3h) {
  _vertexData.setHierarchicalU( _vertexData.getU()-Pu_3h );
}
```

> **Remark:** In this presentation, we make the solvers restrict residuals on all levels all the time and prolong all updates all the time. The actual distinction between additive and multiplicative multigrid stems from the decision which level is smoothed, i.e. where we find an update. This is highly inefficient. Good solvers would compute only those residuals and right-hand sides that have changed and are required. But the purpose of this presentation is to introduce a quick, brief prototype. Speed is a different topic.

The accumulation of the hierarchical surplus itself is more or less a cut-n-paste from the Jacobi smoother:

```
void multigrid::mappings::HierarchicalTransformAndRHSRestriction::enterCell(...) {
  const tarch::la::Vector<TWO_POWER_D,double> hierarchicalU =
      VertexOperations::readHierarchicalU( fineGridVerticesEnumerator, fineGridVertices );
```

```
    const tarch::la::Vector<TWO_POWER_D,double> hierarchicalROld =
      VertexOperations::readHierarchicalR( fineGridVerticesEnumerator, fineGridVertices );
    const matrixfree::stencil::ElementWiseAssemblyMatrix A =
      fineGridCell.getElementsAssemblyMatrix( fineGridVerticesEnumerator.getCellSize() );

    tarch::la::Vector<TWO_POWER_D,double> hierarchicalR =
      hierarchicalROld -A *hierarchicalU;

    VertexOperations::writeHierarchicalR
      ( fineGridVerticesEnumerator, fineGridVertices, hierarchicalR );
}
```

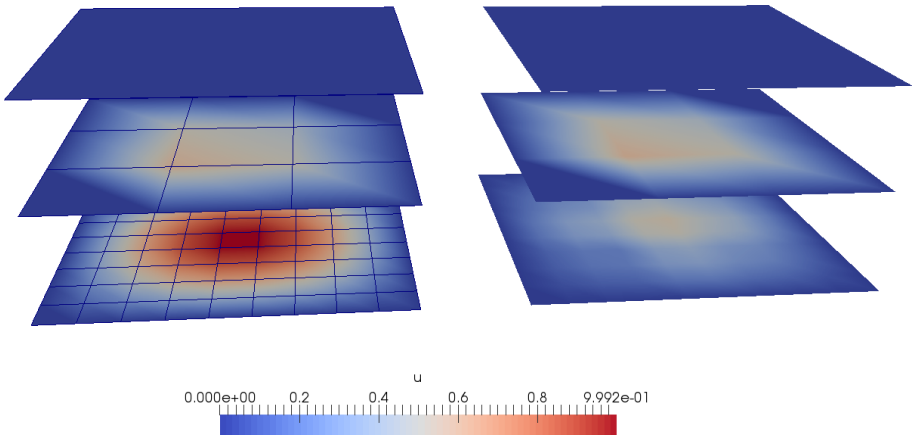
For the restriction, we again rely on the multigrid object, where a specialised prolongation function does exist that uses d-linear interpolation. Furthermore, we stick to the convention $R = P^T$ and thus can write:

```
void
multigrid::mappings::HierarchicalTransformAndRHSRestriction::touchVertexLastTime(...) {
  if ( fineGridVertex.isInside() ) {
    const tarch::la::Vector<TWO_POWER_D, double > P =
      _multigrid.calculateP(fineGridPositionOfVertex);

    dfor2(k)
      // There is no need to exclude boundary points here (the rhs does not play
      // there a role anyway), but it makes the visualisation nicer.
      if (
        coarseGridVertices[ coarseGridVerticesEnumerator(k) ].getRefinementControl()==
          Vertex::Records::Refined
        &&
        coarseGridVertices[ coarseGridVerticesEnumerator(k) ].isInside()
      ) {
        coarseGridVertices[ coarseGridVerticesEnumerator(k) ].incF(
          P(kScalar) *fineGridVertex.getHierarchicalResidual()
        );
      }
    enddforx
  }
}

void multigrid::Vertex::incF(double value) {
  _vertexData.setF( _vertexData.getF()+value );
}

double multigrid::Vertex::getHierarchicalResidual() const {
  return _vertexData.getF() + _vertexData.getHierarchicalR();
}
```

We close the preparatory work with the remark that the specification has to be augmented by additional readers and writers to make our code snippets work:

```
component: Multigrid

namespace: ::multigrid

vertex:
```

```
  dastgen-file: Vertex.def
  read scalar(double): U
  read scalar(double): F
  read scalar(double): R
  read scalar(double): D
  read scalar(double): HierarchicalU
  read scalar(double): HierarchicalR
  write scalar(double): U
  write scalar(double): R
  write scalar(double): D
  write scalar(double): HierarchicalU
  write scalar(double): HierarchicalR

...
```



We see above a Jacobi smoother (100 iterations) for the Poisson equation on the unit square (left) as well as the hierarchical representation (right). We also may augment the adapters with plots of the temporary data

```
adapter:
  name: AnyAdapter
  ...
  merge-with-predefined-mapping: VTKPlotVertexValue(u,getU,u)
  merge-with-predefined-mapping: VTKPlotVertexMultilevelValue(multiscaleU,getU,u)
  merge-with-predefined-mapping: VTKPlotVertexMultilevelValue
    (multiscaleHierarchicalU,getHierarchicalU,u)
  merge-with-predefined-mapping: VTKPlotVertexMultilevelPointCloud
    (multiscaleResidual,getResidual,res)
  merge-with-predefined-mapping: VTKPlotVertexMultilevelPointCloud
    (multiscaleHierarchicalResidual,getHierarchicalResidual,res)
  merge-with-predefined-mapping: VTKPlotVertexMultilevelPointCloud(f,getF,f)
```

and emphasise that we basically have almost all multigrid ingredients at hands. Missing is a projection from coarse grid updates to fine grid values. So far, we update the coarse grid values according to the right-hand side. These updates however are not projected. Instead, the updated approximation in a FAS sense are overwritten at the end of the subsequent iteration with fine grid solution data.

Remark: Most default VTK plotters plot the grid as well as values and (for consistency reasons imposed by the VTK data file format) have to plug into `touchVertexFirstTime`. For residuals and other temporary data, we typically however are interested to plot the value at the end of the iteration. A quick solution here is to plot the grid with one mapping and to use a second mapping that solely plots a point cloud with the vertex values. We then can visualise both data sets simultaneously.

4.2.6 Additive Geometric Multigrid

In additive multigrid, we may not use the same relaxation factor on each and every level. Instead, it is important that the coarser the level the smaller the contribution to the solution update. Otherwise, the solver tends to overshoot and become unstable. We propose to use relaxation factors ω^k with $k = 1$ on the fine grid level and increasing by one for each level up the hierarchy. ω obviously has to be adopted if the grid changes. It is thus straightforward to compute it on-the-fly. For this, we add an additional variable to the vertex

```
class multigrid::records::Vertex {
  persistent parallelise int numberOfFinerLevelsAtSamePosition;
};
```

and make the grid traversal determine the correct value of this attribute on the fly:

void multigrid::Vertex::clearAccumulatedAttributes() $_v ertexData.setR(0.0);_v\, ertexData.setD(0.0);$ $_v ertexData.setNumberOfFinerLevelsAtSamePosition(1);$

void multigrid::mappings::AdditiveMGProlongation::touchVertexFirstTime(multigrid::Vertex fineGridVertex, const tarch::la::Vector¡DIMENSIONS,double¿ fineGridX, const tarch::la::Vector¡DIMENSIONS,double¿ fineGridH, multigrid::Vertex * const coarseGridVertices, const peano::grid::VertexEnumerator coarseGridVerticesEnumerator, multigrid::Cell coarseGridCell, const tarch::la::Vector¡DIMENSIONS,int¿ fineGridPositionOfVertex) logTraceInWith6Arguments("touchVertexFirstTime(...)", fineGridVertex, fineGridX, fineGridH, coarseGridVerticesEnumerator.toString(), coarseGridCell, fineGridPositionOfVertex);

if (fineGridVertex.isInside()) const tarch::la::Vector¡TWO$_P OWER_D, double > e_3h = VertexOperations$ :: $readUUpdate(coarseGridVerticesEnumerator, coarseGridVertices); constdoublePe_3h =_m$ $ultigrid.getDLinearInterpo$

fineGridVertex.correctU(Pe$_3h * fineGridVertex.getDampingFactorForAdditiveCoarseGridCorrection(multigrid$ :: $mappings$ :: $JacobiSmoother$ :: $omega$));

// hier muss ich auch noch die Level beruecksichtigen. Das kann aber innerhalb der Vertex erfolgen.

logTraceOutWith1Argument("touchVertexFirstTime(...)", fineGridVertex);

Update doch spezifisch

inject ein bischen falsch. Eigentlich besser R.

@todo Instabil. Relaxation mit aufnehmen

@todo Kleiner Fehler mit der Injektion. Auf Bram verweisen, aber so lassen

4.2.7 Multiplicative Geometric Multigrid

V2,2?

Further reading

- Reps, Bram and Weinzierl, Tobias: *Complex additive geometric multilevel solvers for Helmholtz equations on spacetrees*, tech report, arXiv:1508.03954

- Weinzierl, Marion: *Hybrid Geometric-Algebraic Matrix-Free Multigrid on Spacetrees*, Dissertation, Technische Universität München, 2013
- Muntean, Ioan Lucian, Mehl, Miriam, Neckel, Tobias and Weinzierl, Tobias (2008). *Concepts for Efficient Flow Solvers Based on Adaptive Cartesian Grids.* In High Performance Computing in Science and Engineering, Garching 2007. Wagner, Siegfried, Steinmetz, Matthias, Bode, Arndt Brehm, Matthias Berlin Heidelberg New York: Springer.
- Weinzierl, Tobias and Köppl, Tobias (2012). *A Geometric Space-time Multigrid Algorithm for the Heat Equation.* Numerical Mathematics: Theory, Methods and Applications 5(1): 110-130.
- Mehl, Miriam, Weinzierl, Tobias and Zenger, Christoph (2006). *A cache-oblivious self-adaptive full multigrid method.* Numerical Linear Algebra with Applications 13(2-3): 275-291.

4.3 Diffusion-convection with PETSc

Time: 60 minutes.
Required: Chapter 2 and a working PETSc installation.

I wrote Peano with matrix-free methods from CFD in mind. It turns out that it is not too complicated to use it for other application areas or to established linear algebra packages. We assume that you have PETSc plus BLAS, Lapack and MPI (if required) ready installed on your system. I have tested all the present concepts with PETSc 3.7.2 but everything discussed here is pretty basic, so it should work with older versions as well. Please note that this is a feasibility study and we do not optimise at all.

4.3.1 Preparation of Peano

If we use explicit assembly, we need an enumeration of our grid. So we do start with a grid

```
> java -jar <mypath>/pdt.jar --create-project petsc chapter-4.3
```

and add each vertex an index. We do restrict to a discretisation with one degree of freedom per vertex for the time being:

```
Packed-Type: short int;

class petsc::records::Vertex {
  parallelise persistent int index;
};
```

For the present exercise, we will have four different things to do: create the grid, create and assemble the matrix, solve the problem and plot the result. The solve is a pure PETSc operation. For each of the remaining three phases, we need a mapping/adapter in Peano and we also want access to `index`. For the latter, we could write setters and getters and stick to an object-oriented paradigm. But this way, it is easier (though not as beautiful):

```
component: MyFancyPETScProject

namespace: ::petsc

vertex:
  dastgen-file: Vertex.def
  read scalar(int): Index // mind the uppercase
  write scalar(int): Index

cell:
  dastgen-file: Cell.def

state:
  dastgen-file: State.def

event-mapping:
  name: CreateGrid

event-mapping:
  name: Assemble
```

```
adapter:
  name: CreateGrid
  merge-with-user-defined-mapping: CreateGrid

adapter:
  name: Assemble
  merge-with-user-defined-mapping: Assemble

adapter:
  name: Plot
  merge-with-predefined-mapping: VTKPlotVertexValue(result,getU,u)
```

We generate all glue code

```
> java -jar <mypath>/pdt.jar --generate-gluecode \
  petsc/project.peano-specification petsc <mypath>/pdt/usrtemplates
> make -f petsc/makefile
```

We recognise that this code does not compile yet as we have not yet realised `double Vertex::getU() const`. For the time being, we make the routine return 0. Finally, we change into the mapping the creational events yield a regular grid:

```
void petsc::mappings::CreateGrid::createInnerVertex(...) {
  logTraceInWith6Arguments( "createInnerVertex(...)", ... );

  if (coarseGridVerticesEnumerator.getLevel()<3) {
        fineGridVertex.refine();
  }

  logTraceOutWith1Argument( "createInnerVertex(...)", fineGridVertex );
}


void petsc::mappings::CreateGrid::createBoundaryVertex(...) {
  logTraceInWith6Arguments( "createBoundaryVertex(...)", ... );

  if (coarseGridVerticesEnumerator.getLevel()<3) {
        fineGridVertex.refine();
  }

  logTraceOutWith1Argument( "createBoundaryVertex(...)", fineGridVertex );
}
```

Compile it, run it, visualise the output.

4.3.2 Connecting to PETSc

As PETSc's philosophy is pretty straightforward and minimalistic (they do, e.g., not hijack the main), the initialisation and integration are straightforward. Please note that we stick to serial jobs in the first part of this section. Nevertheless, we use the MPI compiler wrapper to translate the code as I faced issues when I tried to use PETSc without MPI[3].

[3]Please consult `http://www.mcs.anl.gov/petsc/petsc-current/docs/installation.html#i-dont-want-to-use-mpi` for the usage with OpenMPI as the `LD_LIBRARY_PATH` has to be

1. Before we start to code with PETSc, we have to make the makefile know where PETSc is installed. If you use your own build environment, adopt all pathes accordingly. If you use the pre-generated makefile, open it and add your PETSc include directory to the search path. Also switch to `mpiCC`.

```
PROJECT_CFLAGS = -DDebug -DAsserts -DParallel \
  -DMPICH_IGNORE_CXX_SEEK -I/opt/petsc/include
PROJECT_LFLAGS = -L/opt/petsc/lib -lpetsc
PROJECT_LFLAGS = -L/opt/petsc/lib -lpetsc

[...]

CC=/opt/openmpi/bin/mpiCC
```

   Please adopt all pathes accordingly. Depending on your MPI version, you also might run into issues with the compiler's error management. I recommend to remove `-Werror -pedantic-errors` from your makefile in this case.

2. As the present PETSc solutions use MPI (through they run serially), we have to make a few modifications in the runner. These modifications are detailed in Chapter 5.2, so we simply show what has to be changed in `runners/Runner.cpp`. This solution should work out-of-the-box. For a description of its semantics, please see the aforementioned chapter.

```
#include "tarch/parallel/FCFSNodePoolStrategy.h"
#include "peano/parallel/loadbalancing/OracleForOnePhaseWithGreedyPartitioning.h"

[...]

int petsc::runners::Runner::run() {

  [...]

  petsc::repositories::Repository* repository =
    petsc::repositories::RepositoryFactory::getInstance().createWithSTDStackImplementation(
      geometry,
      tarch::la::Vector<DIMENSIONS,double>(1.0), // domainSize,
      tarch::la::Vector<DIMENSIONS,double>(0.0) // computationalDomainOffset
    );


  // This is new because of PETSc:
  if (tarch::parallel::Node::getInstance().isGlobalMaster()) {
    tarch::parallel::NodePool::getInstance().setStrategy(
      new tarch::parallel::FCFSNodePoolStrategy()
    );
  }
  tarch::parallel::NodePool::getInstance().restart();
  tarch::parallel::NodePool::getInstance().waitForAllNodesToBecomeIdle();
  peano::parallel::loadbalancing::Oracle::getInstance().setOracle(
    new peano::parallel::loadbalancing::OracleForOnePhaseWithGreedyPartitioning(false)
  );

  [...]
```

set properly.

```
}
```

3. Finally, we have to initialise and shut down PETSc properly in `main.cpp`:

```
#include "petscsys.h"

[...]

int main(int argc, char** argv) {
  [...]
  PetscInitialize(&argc,&argv,(char*)0,(char*)0);

  int programExitCode = 0;

  if (programExitCode==0) {
    [...]
    petsc::runners::Runner runner;
    programExitCode = runner.run();
  }

  PetscFinalize();
  [...]
}
```

According to PETSc's documentation, we could use PETSc's initialisation to set up MPI instead of the Peano initialisation. However, Peano's initialisation does not check whether MPI has been set up before where PETSc does. So it makes sense to initialise Peano first.

4.3.3 Letting PETSc do the work

Once we have created the grid (and perhaps visualised it), we can start to set up the PETSc data structures corresponding to the grid, we can call its solver, and we can finally free everything afterwards. We stick to the regular case for the time being.

Every fine grid vertex in the grid has a right-hand side and a solution entry. Different to previous sections, we do not store this data in the vertex. Instead, we will make PETSc hold two vectors rhs and x and make each vertex hold the index, i.e. each vertex knows which entry in the two vertices is associated with the vertex. So this means that the runner has to create these vectors and destroy them afterwards as well:

```
  thisIsATest++;
```

```
#include "petscvec.h"
#include "petscmat.h"

[...]

int petsc::runners::Runner::runAsMaster(petsc::repositories::Repository& repository) {
  peano::utils::UserInterface::writeHeader();

  // Build up the grid. Afterwards, we know how many vertices exist.
  repository.switchToCreateGrid(); repository.iterate();
```

```
// These are the global vectors that we use to make the adapter communicate with PETSc:
Vec x;
Vec rhs;
Mat A;

// Create the global vector required for our work with PETSc.
PetscErrorCode errorX =
  VecCreate(tarch::parallel::Node::getInstance().getCommunicator(), &x);
PetscErrorCode errorRhs =
  VecCreate(tarch::parallel::Node::getInstance().getCommunicator(), &rhs);
PetscErrorCode errorA =
  MatCreate(tarch::parallel::Node::getInstance().getCommunicator(), &A);

if (errorX!=0) {
  PetscError( tarch::parallel::Node::getInstance().getCommunicator(),
    __LINE__, __FUNCT__, __FILE__, errorX, PETSC_ERROR_INITIAL,
    "creating global solution vector failed" );
}
if (errorRhs!=0) {
  PetscError( tarch::parallel::Node::getInstance().getCommunicator(),
    __LINE__, __FUNCT__, __FILE__, errorRhs, PETSC_ERROR_INITIAL,
    "creating global rhs vector failed" );
}
if (errorA!=0) {
  PetscError( tarch::parallel::Node::getInstance().getCommunicator(),
    __LINE__, __FUNCT__, __FILE__, errorA, PETSC_ERROR_INITIAL,
    "creating system matrix failed" );
}


// Local size (first parameter) and global size are the same for the time
// being. Peano uses doubles to count vertices, as the number of vertices
// sometimes exceed int. So we have to use an ugly cast here.
const PetscInt numberOfUnknowns = static_cast<int>(
  repository.getState().getNumberOfInnerLeafVertices());

assertion1( numberOfUnknowns>0, numberOfUnknowns );

VecSetSizes(x, PETSC_DECIDE,numberOfUnknowns);
VecSetSizes(rhs,PETSC_DECIDE,numberOfUnknowns);
VecSetType(x,VECMPI);
VecSetType(rhs,VECSEQ);
MatSetSizes(A,PETSC_DECIDE,PETSC_DECIDE,numberOfUnknowns,numberOfUnknowns);
MatSetType(A,MATSEQAIJ);
MatSetUp(A);

repository.switchToAssemble(); repository.iterate();

// @todo Do the real solve here

repository.switchToPlot(); repository.iterate();


// Clean up PETSc data structures
```

```
  VecDestroy(&x);
  VecDestroy(&rhs);
  MatDestroy(&A);

  repository.logIterationStatistics( true );
  repository.terminate();

  return 0;
}
```

That's basically the structure of our whole solver. What is missing is to set up the system matrix and to befill the *rhs* vector. For this, we have to ensure that the assembly process (and also the plot later on) know the respective data structures. For convenience, I thus recommend to move the three important data variables into `Vertex.h`:

```
#include "petscvec.h"
#include "petscmat.h"


namespace petsc {
  class Vertex;

  // Forward declaration
  class VertexOperations;

  // These are the global vectors that we use to make the adapter communicate
  // with PETSc:
  extern Vec x;
  extern Vec rhs;
  extern Mat A;
}
```

Please note that we move these variables, i.e. their declaration has to be removed from `runAsMaster`. Furthermore, we have to add their definition to `Vertex.cpp`:

```
Vec petsc::x;
Vec petsc::rhs;
Mat petsc::A;
```

We finally have to solve the actual equation system in the runner. For this, we plug into the `@todo` from the snippet above and furthermore include `#include "petscksp.h"`. We use a simple Krylov CG solver here with a diagonal value preconditioner.

```
KSP krylovSolver;
PC preconditioner;

KSPCreate(tarch::parallel::Node::getInstance().getCommunicator(),&krylovSolver);
KSPSetOperators(krylovSolver,A,A); // use matrix from Vector.h

// connect/get preconditioner
KSPGetPC(krylovSolver,&preconditioner);

// set Jacobi as preconditioner
PCSetType(preconditioner,PCBJACOBI);
```

```
KSPSolve(krylovSolver,rhs,x);
KSPDestroy(&krylovSolver);
```

This is not a particular sophisticated code, but it does the job. No need to rewrite parts of the excellent PETSc documentation.

4.3.4 Connecting the vertices to PETSc

We next have to couple the vertices from our grid to the PETSc unknowns. For this, we add a field

```
int _unknownCounter;
```

to the class `Assemble` and make each fine grid vertex hold a unique index which is basically the corresponding entry in the x and rhs vector. If we do so, we can also set rhs already.

```
#include "petsc/VertexOperations.h"

void petsc::mappings::Assemble::beginIteration(
  petsc::State& solverState
) {
  logTraceInWith1Argument( "beginIteration(State)", solverState );

  _unknownCounter = 0;

  logTraceOutWith1Argument( "beginIteration(State)", solverState);
}

void petsc::mappings::Assemble::touchVertexFirstTime(
  ...
) {
  logTraceInWith6Arguments( "touchVertexFirstTime(...)", ... );

  if (
    fineGridVertex.getRefinementControl()==Vertex::Records::Unrefined
    &&
    fineGridVertex.isInside() // it is not a boundary point
  ) {
    VertexOperations::writeIndex(fineGridVertex,_unknownCounter);
    _unknownCounter++;

    fineGridVertex.setRhs( 1.0 ); // See routines below
  }

  logTraceOutWith1Argument( "touchVertexFirstTime(...)", fineGridVertex );
}
```

There are many technical ways to couple the vertex to PETSc. We do it via the vertices:

```
double petsc::Vertex::getU() const {
  double result = 0.0;
  if (isInside()) {
    PetscInt indices[] = {_vertexData.getIndex()};
    PetscScalar values[] = {0.0};
```

```
    VecGetValues(x,1,indices,values);

    result = values[0];
  }
  return result;
}


void petsc::Vertex::setRhs(double value) {
  PetscInt indices[] = {_vertexData.getIndex()};
  PetscScalar values[] = {value};
  VecSetValues(rhs,1,indices,values, INSERT_VALUES);
}
```

4.3.5 Assembling and retrieving data from PETSc

We finally assemble our systems. The key activity is on the one hand to tell PETSc that we have finished the assembly

```
void petsc::mappings::Assemble::endIteration(
  petsc::State& solverState
) {
  logTraceInWith1Argument( "endIteration(State)", solverState );

  logInfo( "beginIteration(State)", "finalise_assembly" );

  VecAssemblyBegin( x );
  VecAssemblyBegin( rhs );
  VecAssemblyEnd( x );
  VecAssemblyEnd( rhs );

  MatAssemblyBegin(A, MAT_FINAL_ASSEMBLY);
  MatAssemblyEnd(A, MAT_FINAL_ASSEMBLY);

  logInfo( "beginIteration(State)", "assembly_finished" );

  logTraceOutWith1Argument( "endIteration(State)", solverState);
}
```

and on the other hand to fill entries into the matrix:

```
#include "tarch/la/Matrix.h"


void petsc::mappings::Assemble::enterCell(
  ...
) {
 logTraceInWith4Arguments( "enterCell(...)", ... );

 if ( not fineGridCell.isRefined() ) {
  tarch::la::Matrix<4,4,double> localA;
  localA = 0.5, -0.25, -0.25, 0,
           -0.25, 0.5, 0, -0.25,
           -0.25, 0, 0.5, -0.25,
```

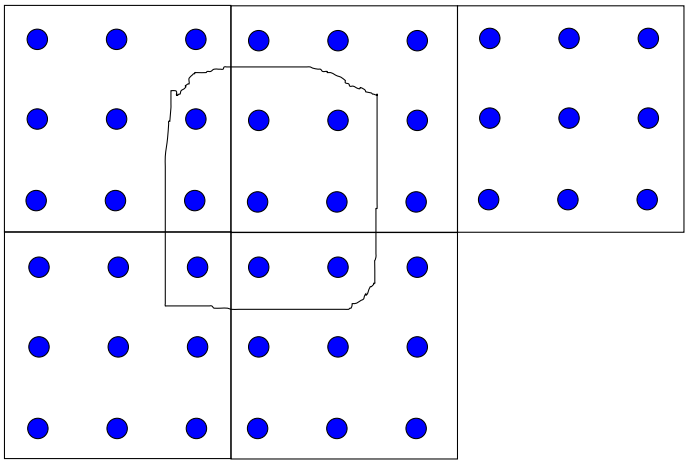


Figure 4.1: One possible data layout for DG with PETSc. All indices are stored within vertices.

```
            0, -0.25, -0.25, 0.5;

  for(int ii=0; ii<4; ii++)
  for(int jj=0; jj<4; jj++) {
   if (
    fineGridVertices[fineGridVerticesEnumerator(ii)].isInside()
    &&
    fineGridVertices[fineGridVerticesEnumerator(jj)].isInside()
   ) {
    PetscInt row = VertexOperations::readIndex( fineGridVertices[fineGridVerticesEnumerator(ii)] );
    PetscInt col = VertexOperations::readIndex( fineGridVertices[fineGridVerticesEnumerator(jj)] );
    MatSetValue(A,row,col,localA(ii,jj),ADD_VALUES);
   }
  }
 }

 logTraceOutWith1Argument( "enterCell(...)", fineGridCell );
}
```

Needless to say, this is a 2d example. 3d requires more indices. You might want to have a look into the Peano utilities for dimension-generic routines. And it is also obvious that this is not a very elegant and fast solution as we write into the matrix entry by entry rather than in batches.

Once the matrix is assembled, it makes sense to add statements alike

```
KSPCreate(tarch::parallel::Node::getInstance().getCommunicator(),&krylovSolver);
KSPSetOperators(krylovSolver,A,A); // use matrix from Vector.h
```

into the runner to see the matrix pattern, e.g.

4.3.6 Some discontinuous Galerkin

There are various ways to implement discontinuous Galerkin in Peano. We sketch a vertex-based variant at hands of bi-quadratic shape functions. For such an approach, we have to store nine unknown per cell. However, we propose not to store indices within a cell but within vertices. This

is advantageous as we have, without additional effort, only vertices within a cell at hand. We do not know neighbour cells.

So we propose to hold nine integer indices within each individual vertex. Basically, every unknown is stored within its closest neighbouring vertex (Figure 4.1). For those where multiple options do exist, we store it to the left bottom. The corresponding vertex data structure reads as follows:

```
Packed-Type: short int;

class petsc::records::Vertex {
  parallelise persistent int index[9];
};
```

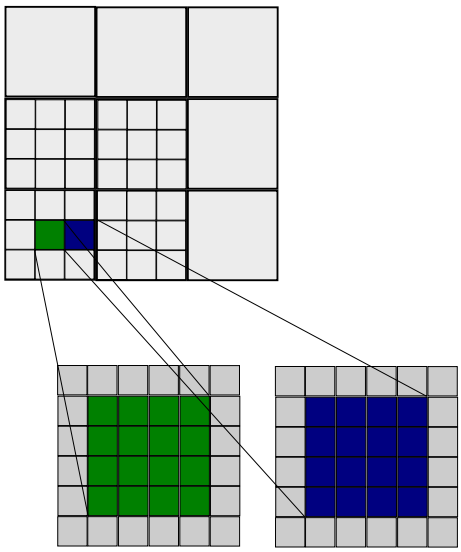
4.3.7 Serial assembly with a parallel PETSc

4.3.8 Using PETSc and MPI

4.4 A patch-based heat equation solver

Time: 60 minutes.
Required: Chapter 2.

In this section, we sketch how to realise a simple explicit heat equation solver that is based upon patches. The idea is that we embed a small regular Cartesian grid into each individual spacetree leaf. This patch is surrounded by a halo/ghost cell layer holding copies from neighbouring patches.



In the sketch, we enbed 4×4 patches into the spacetree cells. Two cells (blue and green) are illustrated. Each patch is surrounded by a halo layer of size one. We will write the code to work within the patches, while the layers around the patches will hold copies of the neighbouring patches and thus couple them.

For this endeavour, we need the toolbox `multiscalelinkedcell` that you can download from the Peano webpage. We assume that the whole toolbox is unzipped into a directory `multiscalelinkedcell` which is held by your source directory.

4.4.1 Preparation

We start with the creation of a file `PatchDescription.def` in your project's root directory. In our example, each patch solely shall hold one array of unknowns u that are associated to the vertices of the patch. So each patch will have exactly $(4 + 2 + 1)^2$ unknowns (the four is the size, there's two halo cells along each coordinate axis, and then there's finally one more vertex than there are cells). Besides the unknowns called u, we also store the position and size as well as the level with each patch—well-aware that level and size are kind of redundant.

```
#include "peano/utils/Globals.h"

Packed-Type: int;

Constant: DIMENSIONS;

/**
 *A cell description describes one individual patch of the overall grid, i.e.
```

```
 *it holds pointers to the actual data of the patch (arrays) and its meta data
 *such as time stamps. Each unrefined node of the spacetree, i.e. each leaf,
 *holds exactly one instance of this class.
 */
class myprojectname::records::PatchDescription {
  /**
   *Two pointers to float arrays.
   */
  parallelise persistent int u;
  /**
   *I need level and offset to be able to determine the source and image in
   *the adaptive case.
   */
  parallelise persistent int level;
  parallelise persistent double offset[DIMENSIONS];
  parallelise persistent double size[DIMENSIONS];
};
```

Please note that u is modelled as integer. Actually, we do not hold the data directly within the patch description but we make the patch description hold a pointer to the actual data. The data will be managed by Peano on the heap. The heap uses integers as pointers. They are actually hash map indices.

Our system design is as follows:

- Each cell holds a pointer to one `PatchDescription`.
- The `PatchDescription` holds a pointer to the actual patch data and comprises some additional meta data (such as the level).
- Each vertex holds 2^d pointers to the `PatchDescription` instances belonging to the adjacent cells.

Whenever we enter a cell, we can thus take its patch description, and get the actual data from this description. Alternatively, we can use the cell's 2^d adjacent vertices. As they know the adjacent patch descriptions, we can also get the data associated to cell neighbours and thus befil the ghost layers, e.g.

Take the Peano description file of our project ensurte that it contains the following lines:

```
heap-dastgen-file: PatchDescription.def

[...]

vertex:
  dastgen-file: Vertex.def
  read vector2PowD(int): PatchIndex
  write vector2PowD(int): PatchIndex

[...]

event-mapping:
  name: Mapping1


event-mapping:
  name: Mapping2

[...]
```

```
adapter:
  name: Adapter1
  merge-with-user-defined-mapping: Mapping1
  merge-with-predefined-mapping: MultiscaleLinkedCell(PatchIndex)

adapter:
  name: Adapter2
  merge-with-user-defined-mapping: Mapping2
  merge-with-predefined-mapping: MultiscaleLinkedCell(PatchIndex)
```

Managing all the adjaceny data (making each vertex point to the right patch) obviously is a tedious task. The `multiscalelinkedcell` toolbox fortunately does most of the stuff for us, if we augment each adapter with a predefined mapping, tell this mapping what the attribute for the patch handling will be (`PatchIndex`), and augment the vertex accordingly. Finally, open `Vertex.def` and augment it accordingly:

```
#include "peano/utils/Globals.h"

Packed-Type: int;

Constant: TWO_POWER_D;

class myprojectname::dastgen::Vertex {
  /**
   *These guys are pointers to the adjacent cells. Actually, they do not point
   *to the neighbouring cells but to the heap indices associated to these cells.
   *These heap indices reference one or several instances of PatchDescription.
   */
  expose persistent int patchIndex[TWO_POWER_D];

  [...]
};
```

We run the translation process and add the toolbox directory to the PDT call:

```
java -jar <mypath>/pdt.jar <mypath>/project.peano-specification <mypath> \
<mypath>/usrtemplates:<mypath>/multiscalelinkedcell
```

The PDT in collaboration with the toolbox will now create code that makes each vertex track the `patchIndex` value of the adjacent cells. If you change your grid, the indices are updated automatically, as long as you merge `MultiscaleLinkedCell` into your adapters. To make the code compile, you finally have to add a routine

```
  int getPatchIndex() const;
```

to your Cell class. Make the routine return the value of an attribute `persistent int patchIndex` that you add to your `Cell.def`. Set this field to -1 in the default constructor.

4.4.2 Setting up the patches

Before we start any coding, we have to specify which heaps we want to use to administer the patch description objects and the actual u data. One option is to define this centrally in the `Cell.h` file that is generated by the PDT:

```
#include "peano/heap/Heap.h"
#include "<mypath>/records/PatchDescription.h"

namespace myprojectsnamespace {
  class Cell;

  typedef peano::heap::PlainHeap< myprojectsnamespace::records::PatchDescription >
    PatchDescriptionHeap;
  typedef peano::heap::PlainDoubleHeap DataHeap;
}
```

In this setup, we use the plain heap from Peano's heap directory to administer both the data and the patch descriptions. There are several other, more sophisticated, heap implementations available. While they allow you to tune your code for special purposes, the plain heap typically is a good starting point.

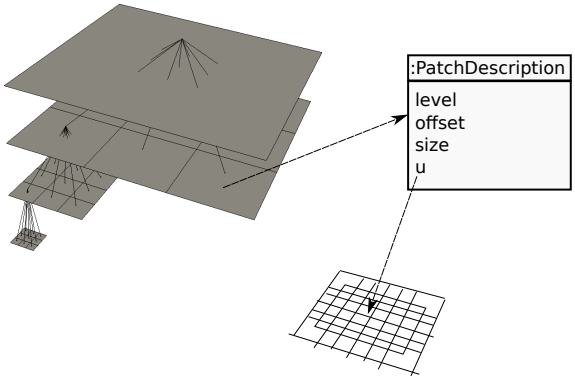
To set up the patches, we create plug into the mapping creating our grid. Alternatively, we can first create the grid and then outsource the patch initialisation into an additional mapping. In any case, I strongly encourage you to initialise the heap as a first step. This is however optional:

```
void myprojectsnamespace::mappings::InitPatches::beginIteration(
  ...
) {
  logTraceInWith1Argument( "beginIteration(State)", solverState );

  PatchDescriptionHeap::getInstance().setName( "patch-description-heap" );
  DataHeap::getInstance().setName( "data-heap" );

  logTraceOutWith1Argument( "beginIteration(State)", solverState);
}
```

So far, each cell points to index -1 as patch description index, and each vertex knows that all adjacent cells point to -1. We change this now as we plug into `enterCell` and introduce a new operation in Cell:

```
void myprojectsnamespace::mappings::InitPatches::enterCell(
  ...
) {
  fineGridCell.init( ... ); // please pass through the level, the offset and the size
}


void myprojectsnamespace::Cell::initCellInComputeTree( ... ) {
  const int newPatchIndex = PatchDescriptionHeap::getInstance().createData(1);
  _cellData.setPatchIndex( newPatchIndex );
  assertion( newPatchIndex>=0 );
  ...
}
```

We finally befill the patch data, i.e. replace the dots in `initCellInComputeTree`. This is a three-fold process. First, we initialise all the meta data. Second, we create the real patch. Finally, we make the meta data record point to this data, while the cell itself points to the `PatchDescription` instance.



It is up to you to specificy the semantics of the data arrays used. They can represent overlapping or non-overlapping patches. It just has paid off not to make any overlap exceed any neighbouring cell on the same level. The introductory sketch of this chapter illustrates 4×4 patches with a ghost layer/overlap of one. A code for this setup might read as follows:

```
void myprojectsnamespace::Cell::initCellInComputeTree( ... ) {
  const int numberOfPatchDescriptionsPerCell = 1;
  const int newPatchIndex = PatchDescriptionHeap::getInstance().createData
    (numberOfPatchDescriptionsPerCell);
  _cellData.setPatchIndex( newPatchIndex );
  assertion( newPatchIndex>=0 );

  records::PatchDescription& patchDescription =
    PatchDescriptionHeap::getInstance().getData(newPatchIndex)[0];

  patchDescription.setU( DataHeap::getInstance().createData(7*7) );
  patchDescription.setLevel(...);
  patchDescription.setOffset(...);
  patchDescription.setSize(...);
}
```

> **Remark:** There is absolutely no reason to restrict a code to use only the finest level of the spacetree. Furthermore, it might make sense to use different patch sizes in different cells of the same level. Please note that all the patch techniques also work if you store higher order DG shape functions in your cells, e.g.

4.4.3 Working with patches on regular grids

As each cell points to patch description through its field `PatchIndex`, it is a natural choice to work with the associated patch data in `enterCell`. Before we actually do the work, we use the data from the `PatchIndex` objects of the neighbouring cells to initialise our ghost cells. In our case, a simple copying does the job. In other situations, you might have to implement more sophisticated projection operators.

```
#include "multiscalelinkedcell/HangingVertexBookkeeper.h"
#include "multiscalelinkedcell/SAMRTools.h"
#include "peano/utils/Loop.h"
...
void pesplines::mappings::JacobiUpdate::enterCell( ... ) {
```

```
 if (
  !fineGridCell.isRefined()
  &&
  multiscalelinkedcell::HangingVertexBookkeeper::allAdjacencyInformationIsAvailable(
   VertexOperations::readPatchIndex(fineGridVerticesEnumerator,fineGridVertices)
  )
 ) {
  intialiseGhostLayerOfPatch(
    fineGridCell,
    fineGridVertices,
    fineGridVerticesEnumerator
  );

  solve(
    PatchDescriptionHeap::getInstance().getData( fineGridCell.getPatchIndex() )[0],
    fineGridVerticesEnumerator.getCellSize()
  );
 }
}
```

The use of the predicate `allAdjacencyInformationIsAvailable` is too careful here: it should always return `true`. In dynamically adaptive settings, it can happen that adjacency information in the vertices (i.e. which patch descriptions are held by the adjacent cells) it not up-to-date immediately. In such a case, the branching would skip a cell with incomplete data and wait for the next traversal where all information is available.

The initialisation of the ghost layer uses Peano's d-dimensional loops (the code then should work for $d = 3$ as well), it rather technical, but not too difficult to understand as it realises plain copying at the end of the day. Again, we use operations provided by the `multiscalelinkedcell` package:

```
void pesplines::mappings::MatVec::intialiseGhostLayerOfCell(
 Cell& fineGridCell,
 Vertex *const fineGridVertices,
 const peano::grid::VertexEnumerator& fineGridVerticesEnumerator
) {
 const tarch::la::Vector<THREE_POWER_D,int> neighbourCellIndices =
  multiscalelinkedcell::getIndicesAroundCell(
   VertexOperations::readPatchIndex(fineGridVerticesEnumerator,fineGridVertices)
 );

 assertion3(
   multiscalelinkedcell::HangingVertexBookkeeper::allAdjacencyInformationIsAvailable(
    VertexOperations::readPatchIndex(fineGridVerticesEnumerator,fineGridVertices)
   ),
   fineGridVerticesEnumerator.toString(),
   VertexOperations::readPatchIndex(fineGridVerticesEnumerator,fineGridVertices),
   neighbourCellIndices
 ); // no enty of neighbourCellIndices points to a patch description on the
    // heap that does not exist

 // we take data from surrounding patches and always write into the
 // patch in the center (destPatchDescription)
 records::PatchDescription& destPatchDescription =
  PatchDescriptionHeap::getInstance().getData( fineGridCell.getPatchIndex())[0];
```

```
 dfor3(i)
  // i does not point to the central patch (all entries 1) and is valid
  // i.e. we are not at the domain boundary, e.g.
  if (
   i!=tarch::la::Vector<DIMENSIONS,int>(1) &&
   neighbourCellIndices(iScalar) >
    multiscalelinkedcell::HangingVertexBookkeeper::InvalidAdjacencyIndex
  ) {
   assertion(neighbourCellIndices(iScalar)>=0);
   records::PatchDescription& srcPatchDescription
    = PatchDescriptionHeap::getInstance().getData( neighbourCellIndices(iScalar) )[0];

   // this if is always true as long as we work with a regular grid
   if (srcPatchDescription.getLevel()==destPatchDescription.getLevel()) {
    // so now write your well-suited for loop here that befill all entries
    // of the ghost layer of your patch. If the for loop determines the
    // indices destVertexIndex and srcVertexIndex specifying vertices within
    // the patch, then the loop body copying the data around reads as
    DataHeap::getInstance().getData(destPatchDescription.getU())
      [destVertexIndex]._persistentRecords._u =
     DataHeap::getInstance().getData(srcPatchDescription.getU())
      [srcVertexIndex]._persistentRecords._u;
   }
  }
 enddforx // counterpart of dfor3 (see documentation in source code)
}
```

The textttdfor3(i) runs over a $\{0,1,2\}^d$ domain. In the loop body (that has to be terminated by a `enddforx` pragma), it provides two loop counters: `i` is a d-dimensional integer vector where each entry is from $\{0,1,2\}$. Furthermore, it gives us another loop counter `iScalar` which runs from 0 through 3^d-1, i.e. is a linearisation of `i`. All the macros are defined in `Loop.h`.

`getIndicesAroundCell` is a helper function that takes the 2^d adjacent vertices of a cell. It actually requires their `PatchIndex` entries which are automatically set by the predefined mapping. The helper tool returns an array with 3^d entries to be read as a 3^d integer field. The first entry holds the patch index of the left bottom neighbour. The second entry holds the patch index of the bottom neighbour. The fourth entry holds the patch index of the left neighbour. The fifth entry hold the patch index of the cell itself. And so forth. The operation works for any dimension. Study its source code documentation for details.

How the actual copying is done depends on the semantics of your data. It also depends on column-major or row-major storage formats for the patch data. The implementation of the actual `solve` operation then finally is straightforward: take the u-array and run through it. Again, a `dfor` loop might simplify your code. If you struggle with the `solve` operation, it might make sense to study the plotting in the next section first. It uses exactly the sketched loop of the patch entries.

Remark: It is not clear a priori whether it is better to have overlapping patches or non-overlapping data structures. With overlaps, there's an additional memory overhead for redundant data and data has to be moved around in the preamble of `enterCell`, e.g. In return, the actual work on the patches can be realised on a continuous array; and thus benefit from vectorisation and parallel fors. Alternatively, one can not use any redundant data and instead access the neighbouring data indirectly. This makes the actual computation often more complicated (one has to analyse whether data is held within the patch of comes from an adjacent patch), but induces no memory overhead and perhaps reduces the stress on the memory subsystem.

4.4.4 Plotting

We briefly sketch what rapid coding of the plotting of a patch-based solver might look like. For this, we rely on a mapping `Plot` that holds plotter classes from Peano's plotter component.

```
#include "tarch/plotter/griddata/blockstructured/PatchWriterUnstructured.h"

namespace pesplines {
  namespace mappings {
    class Plot;
  }
}

class pesplines::mappings::Plot {
  private:
    ...
    static int _snapshotCounter;

    tarch::plotter::griddata::blockstructured::PatchWriter* _writer;
    tarch::plotter::griddata::blockstructured::PatchWriter::SinglePatchWriter* _patchWriter;
    tarch::plotter::griddata::blockstructured::PatchWriter::VertexDataWriter* _uWriter;
};
```

The implementation plugs into `beginIteration` and `endIteration` to open the plotter or to write its data into a file, respectively. Depending on the build type, we either plot binary data or we plot a plain text file that is easier to debug.

```
int ...::mappings::Plot::_snapshotCounter(0);

void ...::mappings::Plot::beginIteration(
  ...::State& solverState
) {
  #if defined(Asserts) || defined(Debug)
  _writer = new tarch::plotter::griddata::blockstructured::PatchWriterUnstructured(
    new tarch::plotter::griddata::unstructured::vtk::VTKTextFileWriter() );
  #else
  _writer = new tarch::plotter::griddata::blockstructured::PatchWriterUnstructured(
    new tarch::plotter::griddata::unstructured::vtk::VTKBinaryFileWriter() );
  #endif
  _patchWriter = _writer->createSinglePatchWriter();

  _uWriter = _writer->createVertexDataWriter("u",3);
```

```
}


void ...::mappings::Plot::endIteration(
  ...::State& solverState
) {
  _patchWriter->close();
  _uWriter->close();

  delete _uWriter;
  delete _patchWriter;

  _uWriter = 0;
  _patchWriter = 0;

  std::ostringstream snapshotFileName;
  snapshotFileName << "solution"
                   #ifdef Parallel
                   << "-rank-" << tarch::parallel::Node::getInstance().getRank()
                   #endif
                   << "-" << _snapshotCounter
                   << ".vtk";
  _writer->writeToFile( snapshotFileName.str() );

  _snapshotCounter++;

  delete _writer;
  _writer = 0;
}
```

The heart of the plotting can be found in `enterCell` where the actual patch data it piped into the solution plotter:

```
void ...::mappings::Plot::leaveCell(...) {
 if ( !fineGridCell.isRefined() ) {
  assertion(PatchDescriptionHeap::getInstance().isValidIndex(fineGridCell.getPatchIndex()));
  const records::PatchDescription& patchDescription
   = PatchDescriptionHeap::getInstance().getData( fineGridCell.getPatchIndex() )[0];

  const std::pair<int,int> indexPair = _patchWriter->plotPatch(
    fineGridVerticesEnumerator.getVertexPosition(),
    fineGridVerticesEnumerator.getCellSize(),
    4+1 // number of inner cells per spacetree leaf
  );

  int unknownVertexIndex = indexPair.first;
  //int unknownCellIndex = indexPair.second;

  dfor(i,4+1) {
   const tarch::la::Vector<DIMENSIONS,int> currentVertex = i + 1;
   const int linearisedCurrentVertex
    = peano::utils::dLinearisedWithoutLookup(currentVertex,4+1);
   const double u = DataHeap::getInstance().getData(patchDescription.getU())
    [linearisedCurrentVertex]._persistentRecords._u;
```

```
      _uWriter->plotVertex( unknownVertexIndex,u );
      unknownVertexIndex++;
    }
  }
}
```

4.4.5 Adaptive grids

Further reading

- Unterweger, K., Wittmann, R., Neumann, P., Weinzierl, T. and Bungartz, H.-J. (2015), *Integration of FULLSWOF2D and PeanoClaw: Adaptivity and Local Time-stepping for Complex Overland Flows*, in Mehl, M., Bischoff, M. Schfer, M. eds, Lecture notes in computational science and engineering, 105 Part II: 3rd International Workshop on Computational Engineering CE 2014. Stuttgart, Germany, Springer, 181-195.
- Weinzierl, Tobias, Wittmann, Roland, Unterweger, Kristof, Bader, Michael, Breuer, Alexander and Rettenberger, Sebastian (2014), *Hardware-aware block size tailoring on adaptive spacetree grids for shallow water waves*, in Größlinger, Armin and Köstler, Harald eds, HiPEAC HiStencils 2014 - 1st International Workshop on High-Performance Stencil Computations. Vienna, Austria.
- Weinzierl, Tobias, Bader, Michael, Unterweger, Kristof and Wittmann, Roland (2014). *Block Fusion on Dynamically Adaptive Spacetree Grids for Shallow Water Waves*. Parallel Processing Letters 24(3): 1441006.

5 Parallel Computing

5.1 Shared memory parallelisation

Time: 30 minutes.
Required: A working simulation code and a compiler that supports either OpenMP or Intel's Threading Building Blocks (TBB) [a].

[a] At the moment, Peano's OpenMP support is slightly outdated. All examples should work straightforwardly with TBB however

In this section, we discuss how to parallelise a simulation code on a shared memory architecture. Hereby, we focus on Peano's parallelisation features. In mature, big applications, they are typically supplemented by further parallelisation that is application-specific: Peano can run lots of routines in parallel if it is correctly used. This way, we are able to exploit several cores. To exploit modern multicore and manycore architectures, codes however also have to use parallelised routines, linear algebra, and so forth. This is an additional level of parallelism that cannot be tackled here.

5.1.1 Preparation

Peano compiles in parallel out-of-the-box if you use the PDT to setup a project blueprint. To facilitate a shared memory parallel build, you have to translate your code with the compile flag `-DSharedTBB`. Alternative variants are `-DSharedOMP`, e.g. Once you edit your makefile and add this compile flag, please also provide the correct include and link paths to the makefile. For modern Intel compilers, no changes should be required, as Intel's TBB come along with the compiler suite.

By default, the auto-generated `main` configures the parallel environment. While OpenMP relies on the setup of a well-suited thread count via environment variables, TBB requires/allows the user to select a thread count manually. If you want to configure the thread count this way, please add the corresponding instructions to your `main`. To make your code portable (and to preserve a serial version), I recommend to embed all shared memory-specific routines into ifdefs. Besides the aforementioned `SharedXXX` defines, Peano also provides a flag `SharedMemoryParallelisation` that is set as soon as OpenMP or TBB is selected. To make use of it, you have to include `MulticoreDefinitions.h`.

```
#ifdef SharedMemoryParallelisation
#include "tarch/multicore/Core.h"
#endif

#include "tarch/multicore/MulticoreDefinitions.h"

  // should be generated by the PDT
  int sharedMemorySetup = peano::initSharedMemoryEnvironment();
  ...
```

```
// manual configuration of threads (optional)
#ifdef SharedMemoryParallelisation
const int numberOfCores = 16;
tarch::multicore::Core::getInstance().configure(numberOfCores);
#endif
```

Peano's kernel uses multiple threads in several places. However, most of these concurrent fragments are really very short-running. As a result, it is not clear whether it pays off to use multithreading or not. Therefore Peano uses an oracle—an object that returns per grid traversal phase whether multitasking should be used or not. This oracle can be found in `peano/datatraversal/autotuning`. Details on this oracle are discussed later. For the time being, insert a commands into your runner.

```
int mynamespace::runners::Runner::run() {
  #ifdef SharedMemoryParallelisation
  peano::datatraversal::autotuning::Oracle::getInstance().setOracle(
    new peano::datatraversal::autotuning::OracleForOnePhaseDummy(true)
  );
  #endif
  ...
}
```

The `true` parameters enables multicore support. It might be reasonable to study the other parameters (see either the header file or Peano's webpages) later. Right now, it should be possible to recompile the code and to run a first version on a shared memory machine. It probably won't yield that much parallel speedup though ...

5.1.2 Specifying concurrency levels

Peano's fundamental idea is that users use events to say what is to be done. But codes leave it to the kernel to decide when and—anticipating some constraints—in which order it is done. This property is exploited by the multicore variant: Peano mappings specify whether multiple calls to one event (such as `enterCell`) may run in parallel. The kernel then decides autonomously whether to run in parallel and on which cores.

To make this work, we have to revise the concept of an adapter. An adapter invokes multiple events. Therefore, the concurrency level of an event has to be the most pessimistic combination of the concurrency levels of all mappings realising this event. This combination is automatically determined by the adapters generated by the PDT.

Concurrency levels are specified within the events. Per event there is one concurrency specification. `touchVertexFirstTime`'s concurrency level for example is specified by the routine `touchVertexFirstTimeSpecification`. If you want to run `touchVertexFirstTime` in parallel, open all mappings and edit `touchVertexFirstTimeSpecification` in each individual one.

A specification returns an instance of `peano::MappingSpecification`. Such an instance accepts parameters:

- The first flag specifies whether the corresponding event works on the whole tree, only on its leaves, or whether it actually does not implement anything at all.
- The second flag specifies whether a particular event may run in parallel.
- Further flags specify whether events support resiliency and other experimental features. For most Peano kernel variants, such further flags are not supported. We maintain these variants in experimental Peano kernels only.

> **Remark:** Even if you run your code without any shared memory parallelisation, it makes sense to tailor all specifications. If your code does work on the finest tree levels only, e.g., you can obtain significant speedup if you change the `WholeTree` flag into `OnlyLeaves`. The most significant (serial) speedups are obtained if you mark all specs as `Nop` where the actual mapping does not do anything in the corresponding events. In this case, the Peano kernel can skip whole function calls completely.

> **Remark:** If you tune/parallelise particular events and if you, at the same time, use predefined mappings, you may have to study the specification objects constructed there. Adapters always have to work pessimistically. If a predefined mapping requires a particular event to be called sequentially (for plotters, e.g.), you can specify any concurrency level you want—the kernel always will run the whole event serially.

For a quick start, I recommend to pick one particular event that is not empty and where you do know exactly that it can run in parallel with other events. Select the correct concurrency level then:

- Serial specifies that this event may not run in parallel and
- All other variants allow the kernel to issue events in parallel. However, they anticipate certain data dependencies—you may for example decide that entering a cell may be issued in parallel as long as no two events can access two adjacent vertices at the same time. This way, you anticipate data races.

5.1.3 Ensuring inter-thread data consistency

Once a parallel event is identified, the Peano kernel may run it on multiple threads in parallel. For this, the whole mapping object is replicated. The replication triggers the mapping's copy constructor. Once the parallel phase is processed—all cells have been entered in parallel, e.g.—all mapping replica are destroyed again and merged into one mapping that is held by the adapter. The mappings provide routines to plug into this life cycle.

In the copy constructor, you have to copy all mapping attributes that you need in your parallel routines. Two scenarios appear most often: Globally read properties such as a state object are copied to each thread instance of the mapping. Globally written attributes such as a global residual are set to zero in the copy constructor:

```
#if defined(SharedMemoryParallelisation)
mynamespace::MyMapping::MyMapping(const Collision& masterThread):
  _localState( masterThread._localState ) {
  // alternatively, we could call
  // _localState = masterThread._localState;

  // now we clear all reduced/accumulated data:
  _localState.clearAttributes();
}
```

The counterpart of the copy constructor is the routine `mergeWithWorkerThread` that is invoked every time a mapping has been replicated to run in parallel on multiple cores and this parallel phase is about to terminate. Peano does not use the destructor of the mapping to merge data to obtain a finer control of thread replica and to be able to reuse mapping instances.

Usually, the merger only reduces globally accumulated data in this routine. If you have followed the recommendation in Section 4.1 to make mappings hold copies of the `State` object and to offer a merge routine, the mapping's shared memory code resembles

```
void mynamespace::MyMapping::::mergeWithWorkerThread(
  const mynamespace::MyMapping& workerThread
) {
  logTraceIn( "mergeWithWorkerThread(Collision)" );

  _localState.merge( workerThread._localState );

  logTraceOut( "mergeWithWorkerThread(Collision)" );
}
#endif
```

5.1.4 Tailoring the oracle

The oracle is the central point of control to decide whether event should be invoked in parallel for given problem sizes. The mapping specifications decide whether events may run in parallel. The oracle specifies whether concurrent events should run in parallel for a given problem size.

Whenever the kernel runs into a particular set of events and decides that it would like to invoke those events in parallel, it realises the following workflow:

- Ask the adapter whether its combination of events allows the kernel to run events in parallel. If the result is a yes:
- Tell the oracle which adapter currently is active.
- Pass the oracle the problem size and ask which grain size (minimal problem chunk size) might be used.
- If the adapter returns 0, nothing is ran in parallel. 0 should be returned by the oracle if the problem overall is too small to benefit from any shared memory parallelisation.
- Otherwise, split the problem into sizes of size at least grain size, replicate the mapping as often as required, and start using multiple threads.

As clarified, each adapter is associated to one oracle, i.e. you are able to use oracles specifying grain and minimal problem sizes on a per-adapter base. Furthermore, oracles are not static. They have a state and thus can for example 'learn' which grain sizes are reasonable[1]. For a quick start, some trial and error with `peano::datatraversal::autotuning::OracleForOnePhaseDummy` are usually sufficient. Just modify some of the predefined variables and study how the actual CPU usage and the time-to-solution change. Once you have detailed knowledge about your application's behaviour, it might be reasonable to replace the Dummy with a tailored implementation of `peano::datatraversal::autotuning::OracleForOnePhase`. Peano's toolbox collection also contains generic, autotuning oracle implementations.

When you study performance, it might be particular interesting that all oracles can fed with real-time data about their performance and provide statistics. To obtain the statistics, call

```
peano::datatraversal::autotuning::Oracle::getInstance().plotStatistics( "" );
```

If you do not need real-time measurements, please construct the oracle (see `new` call in the snippet before) accordingly and disable the clocking. It is expensive.

The routine expects an filename, as most applications that do real-time measurements want to have this data in a file. Furthermore, many oracles (the classes that decide when and how to split up data into concurrent chunks) provide routines to reload these statistics. This way, statistical data is not always created from scratch but incrementally updated. You may pass an empty string to get the data piped directly to the terminal.

[1] We've written a paper on this a few years ago.

5.1.5 Working with Peano's tasks, semaphores, locks and loops

All shared memory parallelisation standards today provide tasks, semaphores, locks, and so forth. Peano provides a wrapper around those which allows you to create one implementation that then runs with OpenMP, TBBs and so forth. To use them, I recommend to study all classes stored in `tarch::multicore`. Notably `BooleanSemaphore` and `Lock` are of interest:

```
static tarch::multicore::BooleanSemaphore mysemaphore;


tarch::multicore::Lock myLock(mysemaphore);

... // critical section

myLock.free(); // destructor would free semaphore automatically
```

The code above is removed automatically if you do not use OpenMP or TBB. Otherwise, it introduces a critical section that is processed by at most one thread at a time.

Furthermore, the header `Loop.h` might be of interest. It provides very useful macros:

- `pfor` is a macro resembling a simple for-loop. If you compile your code with any shared memory compile flag, it makes the loop run in parallel.
- `pdfor` is a `pfor` as well. However, it does not traverse a linear sequence but runs over a d-dimensional index set. Very useful within a cell, e.g., where you have to do something with all vertices, while all vertices can be processed in parallel.

Peano's task concept basically mirrors TBB's task concept. However, if you implement everything within Peano's context, you can also switch to OpenMP later on. The idea is very simple: You create a new class that does you job and you create an instance of this class in your code. Ensure that the class has a functor:

```
void MyTaskClass::operator()() {
 ... // do whatever you wanna do in your task
}
```

and then pass it over to the `TaskSet` class:

```
MyTaskClass myTask(...); // pass in the constructor whatever dataa your task needs
peano::datatraversal::TaskSet( myTask ); // the task now runs in the background
```

This constructor sends `myTask` to the background and continues. It does not wait for the task to finish. A classic pattern is that you introduce a boolean set to false before you launch the task. Modifications to the boolean are protected by a semaphore. Once the task's `operator()` function finishes, it sets the bool. The main thread meanwhile continues, but then there's a while loop that continues to poll the boolean

```
bool terminated = false;
while (!terminated) {
  tarch::multicore::Lock myLock( _myStaticSemphore );
  terminated = _myGlobalBool;
}
```

until it is set.

Finally, there are two useful static operations in `BooleanSemaphore`. The operation `tarch::multicore::BooleanSemaphore::sendTaskToBack()` which is useful to realise low-priority tasks: Other tasks in the system (see section below) are activated if you call this operation.

Please note that there are other constructors of `TaskSet` that run multiple tasks in parallel and wait until both have terminated. See the class description for details.

Further reading

- Weinzierl, Tobias, Bader, Michael, Unterweger, Kristof and Wittmann, Roland (2014). *Block Fusion on Dynamically Adaptive Spacetree Grids for Shallow Water Waves.* Parallel Processing Letters 24(3): 1441006.
- Schreiber, Martin, Weinzierl, Tobias and Bungartz, Hans-Joachim (2013). *Cluster Optimization and Parallelization of Simulations with Dynamically Adaptive Grids.* Euro-Par 2013, Berlin Heidelberg, Springer-Verlag.
- Schreiber, Martin, Weinzierl, Tobias and Bungartz, Hans-Joachim (2013). *SFC-based Communication Metadata Encoding for Adaptive Mesh.* Proceedings of the International Conference on Parallel Computing (ParCo), IOS Press.
- Nogina, Svetlana, Unterweger, Kristof and Weinzierl, Tobias (2012). *Autotuning of Adaptive Mesh Refinement PDE Solvers on Shared Memory Architectures.* PPAM 2011, Heidelberg, Berlin, Springer-Verlag.
- Eckhardt, Wolfgang and Weinzierl, Tobias (2010). *A Blocking Strategy on Multicore Architectures for Dynamically Adaptive PDE Solvers.* Parallel Processing and Applied Mathematics, PPAM 2009, Springer-Verlag.

5.2 MPI parallelisation

Time: 180 minutes.
Required: A working simulation code and working MPI environment.

In this section, we discuss how to parallelise a simulation code with message passing.

5.2.1 Preparation

Peano relies on MPI with its SPMD paradigm, i.e. all ranks run exactly the same code. Internally, it however creates a logical tree topology on all ranks. There is one rank that is the *global master*. By convention, this is rank 0, i.e. the very first rank launched by MPI. All other ranks are *workers*. Each rank besides the global master has one unique *master*. Each worker can either be active, i.e. participate in the computation, or it can be idle. At the begin of the simulation, all workers are idle and only the global master is working (as it is working always).

The global master runs the actual Peano simulation code, holds the simulation state and triggers the grid traversals. Whenever it triggers a grid traversal, it tells all non-idle workers which adapter is currently used. It determines which algorithmic phase to run. Then, it starts to run through the grid. If parts of the grid are deployed to other ranks, they are informed to start a traversal as well. The traversal kick-off propagates through the ranks along the master-worker topology. All of this is done automatically. Most codes require the programmer to think only about the global master.

Re-translating the code. We expect that there is a command `mpicxx` available in your path. This command often also is called `mpiCC`. Please ensure it calls the right compiler backend. If you are unsure, check with argument `-V` and adopt the backend manually if required.

Change your compile command to `mpicxx` and add the two options

```
-DParallel -DMPICH_IGNORE_CXX_SEEK
```

to your compiler call. While Peano is written in C++11, it relies only on C bindings of MPI. Some MPI versions (mpich) have issues with this combination (give you tons of warnings) unless you pass `MPICH_IGNORE_CXX_SEEK`.

Setting up the code. The auto-generated `main` calls Peano's operation `peano::initParallelEnvironment` which takes care of a proper MPI initialisation. As soon as the initialisation has terminated without an error code, you may use the predicate

```
if (tarch::parallel::Node::getInstance().isGlobalMaster()) {
  ...
}
```

to find out whether a particular piece of code runs on the global master. One of the first steps in many codes might be to perform some tests only on this global master. The counterpart `peano::shutdownParallelEnvironment()` typically is invoked directly within `main` as well and is also called by the autogenerated templates already.

As we follow SPMD, `main` has to create an instance of the runner and invoke its `run` operation. The `run` then distinguishes between the global master and all the other workers that blindfolded follow their masters.

```
int result = 0;
if (tarch::parallel::Node::getInstance().isGlobalMaster()) {
  result = runAsMaster( *repository );
}
#ifdef Parallel
else {
  result = runAsWorker( *repository );
}
...
```

This code is generated and it is most of the time sufficient to focus on the operation `runAsMaster`, i.e. to focus on the serial code version. There are however a few steps that have to be done on each individual worker separately. You may implement this within the `main`. However, we typically realise it within `run` just before or after the operation splits up into the function for the global master and all other ranks.

Choose a load balancing request answering strategy on the global master. One for the first steps on the global master is to configure a proper load balancing strategy. Peano realises a hybrid centralised-decentralised load balancing by default, where load balancing decisions are, whenever possible, made centrally. Once load balancing decisions are made (along the lines 'I would like more ranks to help me with my work'), a central point is contacted that decides which ranks are involved in the rebalancing. This central point is called *node pool*. Peano realises the whole node pool, but allows the user to plug into the decisions which ranks are assigned to help which other ranks, e.g. We have to configure the node pool on the global master only

```
if (tarch::parallel::Node::getInstance().isGlobalMaster()) {
  tarch::parallel::NodePool::getInstance().setStrategy(
    new tarch::parallel::FCFSNodePoolStrategy()
  );
}
```

where we select here one of the default strategies. It answers to any incoming load balancing request FCFS (while other strategies might decide to bundle requests to get an overview who asks for resources first before any rank assignment is performed) and hands out MPI ranks as long as there are still idle workers available.

Restart the load balancing on all ranks. Once the node pool is initialised, we have to restart it. This has to be done on all ranks. On the global master, the operation really restarts the node pool. On all other ranks, it establishes the connection to the central node pool and informs the latter how many ranks are available.

```
tarch::parallel::NodePool::getInstance().restart();
tarch::parallel::NodePool::getInstance().waitForAllNodesToBecomeIdle();
```

> **Remark:** Many sophisticated load balancing schemes can deal with MPI ranks signing in and out throughout the computation dynamically. Peano per se can deal with dynamic rank allocation. However, the default FCFS strategy is not that elaborate. It requires that all ranks are well-known when it starts up and that they do not change. Therefore, it is mandatory to add the `waitForAllNodesToBecomeIdle()` instruction here which ensures that all nodes register at the central node pool and tell the central node pool strategy that they are idle and free for new jobs. Though not mandatory, you might want to add this statement anyway at runtime to ensure your whole system is up properly.

Configure the load balancing strategy on all ranks. In a next step, we configure the actual load balancing. Again, we rely on a default balancing that simply yields a static partitioning. This partitioning is established by a greedy grid decomposition while the grid is built up.

```
peano::parallel::loadbalancing::Oracle::getInstance().setOracle(
 new peano::parallel::loadbalancing::OracleForOnePhaseWithGreedyPartitioning(false)
 );
```

This trivial load balancing runs embarrassingly parallel on all ranks. The `false` switches off joins, i.e. we decompose the grid in a greedy fashion.

Peano's general idea is the these oracles implement the load balancing and you can create your own oracle tailored to your needs if you wish. There are also some default oracles available from the homepage (though not included in the kernel—the above one is the only one in the kernel to allow you a quick start) that might suit you. In principle, you could use different oracles on different ranks and make them communicate with each other to globally optimise the load balancing though, in general, it seems to be a good idea to make all ranks run the same oracle. The default oracles don't communicate. Each rank is autonomous.

> **Remark:** Peano's MPI routines are written with the idea in mind that `ifdef`s pollute your code and basically introduce many different branches of your source code that are more difficult to maintain than one single strand. The high level MPI-related function calls thus all work in a serial code, too. If you compile without `-DParallel`, they degenerate to nop (no operation), but you always can be sure that everything compiles, is consistent and lots of assertions are built in in non-release mode.

Please note that the oracles typically should be set *after* you have created the repository. The oracles have to know how many adapters do exist. This information is encoded in the repository.

Set buffer sizes on all ranks. MPI code tends to be very sensitive to proper buffer sizes. In Peano, you have to ensure that all ranks work with the same buffer sizes (otherwise the exchange of buffers will fail) and that you set them before you actually exchange data. Please note that it is not recommended to change these buffer sizes throughout any computation.

```
peano::parallel::SendReceiveBufferPool::getInstance().setBufferSize( bufferSize );
peano::parallel::JoinDataBufferPool::getInstance().setBufferSize( bufferSize );
```

Configure deadlock time outs on all ranks (optional). Peano's MPI communication routines all have some built-in deadlock detection. To enable it, you have to quantify what MPI waiting time is to be considered to be a deadlock. The deadlock identification is split into two phases. A certain timeout interval first has to pass. After it, a warning is launched. After a second timeout passes, the code is terminated. It is reasonable to set these timeouts rather high if you run your code with assertions or debug information. To do all the checks and create debug data is time-consuming and thus might lead into a deadlock identification though all nodes are busy creating the additional information. The other way round, many applications do not allow to run production runs with assertions and debugs. As deadlocks for small problems occur sooner than for large problems where already MPI ill-balancing might yield long waiting times, they might realise the waiting strategy exactly the other way round.

```
#if defined(Debug) || defined(Asserts)
tarch::parallel::Node::getInstance().setDeadlockTimeOut(120*4);
tarch::parallel::Node::getInstance().setTimeOutWarning(60*4);
#else
tarch::parallel::Node::getInstance().setDeadlockTimeOut(120);
tarch::parallel::Node::getInstance().setTimeOutWarning(60);
#endif
```

Clean up all the ranks. It is finally good practice to make the node pool terminate just before `run` has terminated. You might also want to release all MPI datatypes Peano has created, as some tools do complain if you don't so:

```
tarch::parallel::NodePool::getInstance().terminate();
myprojectnamespace::repositories::RepositoryFactory::getInstance().
  shutdownAllParallelDatatypes();
```

Inside `runAsMaster`. Peano's `runAsMaster` for many application is MPI-agnostic. Some applications run into problems if they use a grid setup in one sweep. If Peano tries to decompose/load balance the initial grid, it might be forces to postpone some `refine` calls on the grid. As a result, your grid initialisation should look similar to

```
repository.switchToInitialGrid();
do {
  repository.iterate();
} while (!repository.getState().isGridBalanced());

logInfo(
  "runAsMaster()",
  "number_of_working_ranks=" <<
  tarch::parallel::NodePool::getInstance().getNumberOfWorkingNodes()
);
logInfo(
  "runAsMaster()",
  "number_of_idle_ranks=" << tarch::parallel::NodePool::getInstance().getNumberOfIdleNodes()
);
```

A serial code should continue to quit the `do` loop after one sweep. A parallel code might run through the grid several times.

Two first, almost-parallel runs. Before you continue, I recommend to run your code with `mpirun` and only one active process. Just ensure that your code continues to behave as the serial code does even though MPI features are compiled into the sources. Next, I recommmend you to run the code with two ranks and check whether everything continues to work.

> **Remark:** Peano by default implements a special policy for rank 0. Rank 0 is the global master responsible for global load balancing decisions. Those decisions are critical. Thus, rank 0 always deploys all work to other ranks and focuses exclusively on load balancing and on running the overall algorithm control, i.e. `runAsMaster()`. If you run the code with two MPI ranks, you should thus get similar runtime results as with one rank.

> **Remark:** For fair and proper runtime comparisons, it might make sense to place the first two MPI ranks on one CPU and make them share the resources. Alternatively, you may want to reduce the number of actual ranks in your measurements by one.

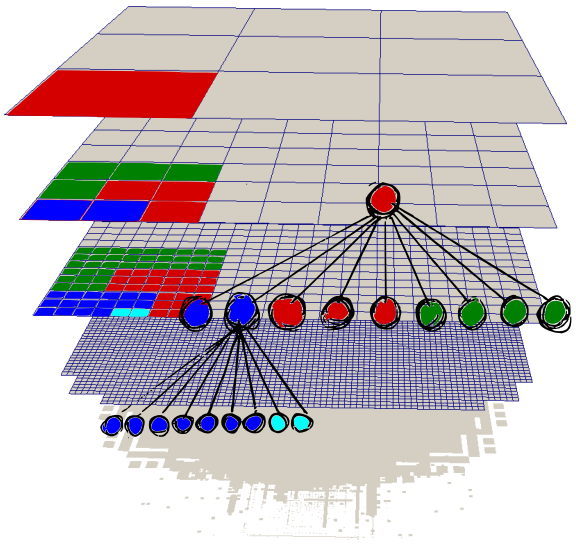
5.2.2 Exchanging the global state

If Peano runs with MPI, the global master holds a repository which in turn holds a solver `State`. This state is the globally valid state object. If the computational domain is split up, each `iterate`

call within the global master is subsequently distributed among all non-idle workers. Together with this run information, the global master also distributes its state object. The state object is broadcasted to all ranks automatically (unless not switched off explicitly as detailed below).

The `State`, the `Vertex` and the `Cell` objects are modelled with DaStGen. DaStGen also is responsible to configure proper MPI calls. Every "DaStGen" object that is exchanged might be exchanged only partially: indeed, DaStGen makes MPI exchange only those attributes that are explicitly marked with `parallelise`. The broadcast remark above thus has to be narrowed: Whenever you call `iterate`, the global master exchanges those properties of its `State` instance that are marked to be exchanged. The object you receive in `beginIteration()` as state argument it exactly this received object.

As soon as a worker terminates its traversal, it informs its master about this termination. This works recursively through the logical master-worker tree. Again, Peano cares for the state transmission. However, any state object received from a worker is *not* merged into the master. If you want to plug into the merge process, i.e. if you want the global master state to hold data from the other states in the end, you have to realise the operation `mergeWithMaster` in any of your mappings.



The picture above details Peano's MPI concept: The spacetree is top-down split up among the ranks. In the present fragment, red is the "highest" rank. When it receives a `iterate` message from the global master, it also receives a copy of the master's state as well as a copy of the coarsest cell belonging to red and its adjacent vertices. All these data is redundantly stored on the master. The received state is automatically made the local state. If you want to do something with the coarsest red cell or the four vertices belonging to the coarsest level, you can realise

```
void mergeWithWorker(
  exahype::Cell& localCell,
  const exahype::Cell& receivedMasterCell,
  const tarch::la::Vector<DIMENSIONS,double>& cellCentre,
  const tarch::la::Vector<DIMENSIONS,double>& cellSize,
  int level
);

void mergeWithWorker(
  exahype::Vertex& localVertex,
  const exahype::Vertex& receivedMasterVertex,
  const tarch::la::Vector<DIMENSIONS,double>& x,
```

```
  const tarch::la::Vector<DIMENSIONS,double>& h,
  int level
);
```

in one of your mappings. Multigrid codes, e.g., might transfer residual through the individual levels.

With all data received, the red node starts to traverse its own tree. It recognises that is in turn is the master to two other ranks (green and blue) and forwards its received state to these guys—as well as the corresponding cells and vertices on the coarsest common level that the master and its workers hold redundantly.

Afterwards, red continues to run through its local tree. When it starts to ascend through its local tree, it waits for its two workers (green and dark blue) to terminate and receives from them a state as well as copies of their coarsest cell plus the adjacent vertices. This time, the state is not merged into anything automatically. However, you can realise the corresponding merge operations to restrict a global residual value for example.

5.2.3 Exchanging boundary data

Peano realises a non-overlapping domain decomposition on each individual resolution level. This means that no cells are held redundantly between two neighbouring ranks on the same level (only the coarsest cell of any rank is held redundantly by its master). Boundary data exchange thus exclusively affects vertices. It is important to note that all vertices on all levels that are adjacent to cells of different ranks are exchanged per iteration (unless not explicitly switched off).

Again, the code does not literally exchange all data. It exchanges only those attributes of `Vertex` that are explicitly marked with `parallelise` in the definition file.

There are plug-in points where you can influence how data is exchanged along the boundaries. The first plug-in point is the operation

```
void prepareSendToNeighbour(
  exahype::Vertex& vertex,
  int toRank,
  const tarch::la::Vector<DIMENSIONS,double>& x,
  const tarch::la::Vector<DIMENSIONS,double>& h,
  int level
);
```

Whenever a rank finds out that it has just used a vertex for the very last time throughout a traversal—this happens right after `touchVertexLastTime` has been called—and that this vertex is adjacent to multiple ranks and thus stored on multiple ranks, it calls `prepareSendToNeighbour` for this vertex per communication partner `toRank`. This operation gives every mapping the opportunity to plug into the send mechanism.

Peano exchanges its data similar to the Jacobi scheme in linear algebra: at the end of an iteration, it sends away copies of vertices. Prior to the subsequent traversal, it receives this data and allows the user to merge it into the local data structures. For the latter, there is another plug in point:

```
void mergeWithNeighbour(
  exahype::Vertex& vertex,
  const exahype::Vertex& neighbour,
  int fromRank,
  const tarch::la::Vector<DIMENSIONS,double>& x,
  const tarch::la::Vector<DIMENSIONS,double>& h,
  int level
);
```

This operation hands you over the local vertex copy as well as the received data from every other adjacent rank. While Peano ensures that all grid refinement data, e.g., is kept consistently, it is your job to ensure that all application-specific data is kept consistent.

A simple matrix-free Jacobi iteration on the vertices thus reads as follows:

1. When a vertex is loaded for the very first time throughout a traversal, we set its residual to 0. This happens in `touchVertexFirstTime()`.
2. In `enterCell`, the local residual of this vertex is accumulated.
3. As soon as we receive `touchVertexLastTime()` of a particular vertex, we know that the code has run through all adjacent cells before. The residual hence is complete and we can update the vertex solution according to the Jacobi update scheme.

This workflow transfers as follows to a distributed memory environment:

1. When a vertex is loaded for the very first time throughout a traversal, we set its residual to 0. This happens in `touchVertexFirstTime()`.
2. In `enterCell`, the local residual of this vertex is accumulated.
3. As soon as we receive `touchVertexLastTime()` of a particular vertex, we know that the code has run through all adjacent cells before *if the vertex is a local one*. We can check this with the operation `isAdjacentToRemoteRank()`. The residual hence is complete and we can update the vertex solution according to the Jacobi update scheme.
4. Of a vertex is adjacent to any other rank, we may not update its value. We know that this vertex is sent away at the end of the iteration through `prepareSendToNeighbour`. We do not change any data of the vertex herein.
5. Prior to the next `touchVertexFirstTime()` on any other rank, we know that this rank will receive our local copy of the vertex in `mergeWithNeighbour`. We thus make `mergeWithNeighbour` take any incoming vertex copy, take the residual from there and merge it into the local residual.
6. If we run into `touchVertexFirstTime()` for a vertex with `isAdjacentToRemoteRank()`, we know that we didn't update its value yet because it had bee incomplete in the previous `touchVertexLastTime()` call. We however know that its residual is accumulated between domain boundaries now. We thus update and continue with the standard `touchVertexFirstTime()`.

5.2.4 Exchanging boundary data on the heap

Different to the vertex boundaries, heap data is not automatically exchanged by Peano's MPI. There are different variants how to manage heaps. The simplest one is to rely on the communication specification. Please ensure that one of your mappings defines a communication specification that sets the third flag to `true`. This instructs the Peano kernel that you want to use heaps for boundary data exchange as well.

```
peano::CommunicationSpecification
myMapping::communicationSpecification() {
  return peano::CommunicationSpecification( ...,...,true );
}
```

Boundary data is also not transferred automatically. If you want to exchange data through a boundary vertex, you have to do so explicitly in `prepareSendToNeighbour`. Please note that Peano's heap has specialised operations for this:

```
void myMapping::prepareSendToNeighbour(
  particles::pidt::Vertex& vertex,
  int toRank,
```

```
  const tarch::la::Vector<DIMENSIONS,double>& x,
  const tarch::la::Vector<DIMENSIONS,double>& h,
  int level
) {
  MyHeap::getInstance().sendData(
    index-of-heap-entry,
    toRank,
    x,
    level,
    peano::heap::NeighbourCommunication
  );
}
```

The counterpart has to be realised in `mergeWithNeighbour`. Peano takes care that all data is exchanged in the right order as efficiently as possible. You are however responsible to ensure that each heap send operation has a corresponding receive on the other side for exactly the same vertex. As the data exchange is distributed between two iterations, you might decide to implement the receive in a different mapping. However, you have to ensure that each send is mapped by exactly one receive on the vertex counterpart.

5.2.5 Running global steps on all ranks

5.2.6 Specifying the communication pattern

5.2.7 Doing something special on a worker

5.2.8 Other MPI components

- If you couple Peano with another component using MPI, and the two codes shall not interfer with collectives, you can change Peano's **communicator**. For this, have a look into `tarch::parallel::Node`. All Peano kernel routines ask the `Node` class which communicator to use. So change it there and you switch Peano to a different communicator.
- Peano uses various MPI **tags** to exchange data. Most Peano components grab a tag from a central data base and then work with this tag, i.e. the tags are hidden from the user. If you want to exchange your own data with MPI commands and want to ensure not to interfer with any Peano messages, use the static operation `reserveFreeTag` from `tarch::parallel::Node` in the system setup to reserve a tag for your own purposes that then exclusive to you.
- If you want to exchange data along the master-worker **topology** or with the **global rank**, it is best to ask the texttttarch::parallel::NodePool for the right ranks. Usually, rank 0 is the global master. However, Peano can be reconfigured to change this behaviour—which is particularly useful if you couple the AMR framework with some other codes.

6 Tuning

6.1 Performance analysis

Time: Less than 10 minutes unless you postprocess a big file.
Required: You may work with the plain output that Peano writes to the terminal. If you use log filters (cmp. Chapter **??**), it is important that you know how to switch particular logging infos on. You also need a working Python installation.

Prior to any parallelisation or tuning discussion, I want to emphasise that it usually makes sense first of all to have a how Peano is performing from a grid point of view. For this, the framework comes along with a rather useful script.

- Recompile your code with `-DPerformanceAnalysis`. For the performance analysis, it usually makes sense to compile with the highest optimisation level and to disable `-DDebug` and `-DAsserts`.
- Run your code and ensure that `info` outputs from the `peano::performanceanalysis` component are enabled.
- Pipe the output into a file:

```
> ./myExecutable myArguments > outputfile.txt
```

  We call this file `outputfile.txt` from hereon.

- Pass the output file to Peano's performance analysis script written in Python. Besides the script (name), you also have to tell the script how many MPI ranks you have used and how many threads have been enabled. Skip the arguments if you haven't used MPI.

```
> python <mypath>/peano/performanceanalysis/performanceanalysis.py outputfile.txt
```

- Open the web browser of your choice and open the file `outputfile.txt.html`

Remark: Besides the output written by Peano through the component `peano::performanceanalysis`, you also have to use the `CommandLineLogger` (the default), and you have to make this one write out time stamps as well as trace information. If you use your own logger or a modified log format, the Python script will fail.

Remark: The present document suggests to pipe all output data from the terminal into a file and to process this file. For many applications with low rank count this works fine. The more ranks you have however the higher the probability that concurrent writes to the terminal mess up your piped file. In most cases, the performance analysis still succeeds. However, there are cases where the postprocessing fails. In this case, it is better to use `setLogFormat` of the `CommandLineLogger` to make the logger pipe output from different ranks into different files. There's an additional script in the `performanceanalysis` directory that you can then use to fuse the various output files into one file before you start the actual postprocessing.

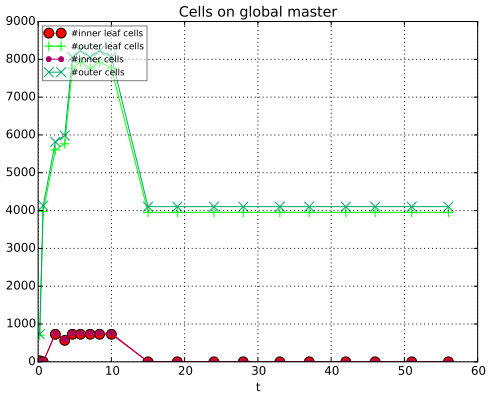
If you browse through your directory, you will notice that all graphs are written both as png and as pdf. You can thus integrate them directly into your LaTeX reports.

6.2 Reducing the MPI grid setup and initial load balancing overhead

Time: Around 30 minutes.
Required: A working MPI code.

In this section, we assume that you've a reasonable load balancing and that you were able to postprocess your performance analysis outputs. We discuss issues that arise over and over again for parallel applications.

6.2.1 Massive grid on rank 0 with long redistribution phase afterwards



Smell: There are ranks (notably rank 0) that are assigned lots of cells and vertices and then this number decreases though the grid should be more or less static.

We observe this behaviour either through a performance analysis (see plot above) or by outputs from the state where the grid depth immediately goes up to the maximum depth while the load balancing still splits up things. The iterations already are very expensive; obviously as the grid is already in place but the ranks are not all employed.

The reason for this behaviour can be found in the semantics of `createVertex` and `touchVertexFirstTime`. Both operations try to refine the grid around the respective vertex immediately. Only if circumstances such as a parallel partitioning running through this vertex—the refinement instruction then first has be distributed to all ranks holding a copy of this vertex—do not allow Peano to realise the refinement immediately, the refinement is postponed to the next iteration. In many parallel codes, all the refinement calls pass through immediately on rank 0 before it can spawn any rank. This leads to the situation that the whole grid is in one sweep built up on the global master and afterwards successively distributed among the ranks.

Such a behaviour is problematic: the global rank might run out of memory, lots of data is transferred, and the sweeps over the whole grid on rank 0 are typically pretty expensive. A distributed grid setup is advantageous.

Smell: Switch from an aggressive refinement into an iterative grid refinement strategy to allow the ranks to deploy work throughout the grid construction and thus build up the grid in parallel and avoid the transfer of whole grid blocks due to rebalancing.

The simplest materialisation of this idea is to move your `refine()` calls from the creational or touch first events into `touchVertexLastTime()`: As a consequence, setting up a (rather regular) grid of depth k requires at least k iterations.

Remark: I typically extract the intial grid construction decision into a function of its own. If `-DParallel` is used, I invoke grid constructions only from `touchVertexLastTime`. Otherwise, I use it in the creational routines for inner and boundary vertices. For this strategy, we have to ensure that touch last is not set to `Nop` in the specification attributes if we compile with MPI.

In a second step, you might consider to extend your grid only every second traversal. Everytime you rebalance your grid, Peano disables dynamic load balancing for a couple of iterations (three or four). Throughout these iterations, it can recover all adjacency information if the grid itself changes as well. Consequently, it does make sense to add a couple of adapter runs after each grid modification that to not change the grid structure: When you know that you have an adapter that changes the grid, apply afterwards an adapter that does not change the grid for a couple of times. This way, you ensure that no mpi rank runs out of memory. The grid generation does not overtake the rebalancing. I often do not use an additional adapter but ensure that refines are only called if the tree traversal direction is not inverted: Peano runs forth and back through the grid, so this effectively switches off refinement every second iteration.

The deluxe version is a code that refines only if the previous grid traversal did not change the mesh, if no load balancing is going anymore and, for example, `isTraversalInverted()` does not hold. This postpones any refinement further. However, such an approach is a bad idea if no idle nodes are available anymore. In this case, it is better not to veto the refinement anymore. This situation is discussed in the following bad smell, where we propose a sophisticated solution to both smells.

6.2.2 Incremental, slow grid setup though detailed grid structure is known/all nodes are already busy

Smell: The grid is static, but, nevertheless, Peano needs lots of iterations to finally build it up. Even worse, there are no idle nodes left and we know analytically what the grid should look like. The grid construction thus should rush through in one iteration.

Peano exchanges the vertices along a domain boundary after each traversal. At the same time, it tries to refine a vertex with a refinement flag as soon as possible: if a code sets a refine command upon creation of a vertex or when a vertex is loaded for the very first time, it immediately refined. If refine is called later throughout the traversal, Peano has to memorise the refinement request and wait for the subsequent traversal as all adjacent cells have to anticipate an ongoing refinement but some have already been processed. To ensure that the grid is always consistent, vertices at a domain boundary never can be refined immediately. These guys are exchanged after each traversal and merged with their counterpart on other ranks prior to the next usage. Refinement triggers thus never are available immediately on other ranks—these vertices are replicated not held synchronous.

This behaviour implies that the grid can be built up at most by one level per iteration along domain boundaries. Therefore, we often see codes running many iterations until a regular grid is built up completely. Often, this is not necessary as all ranks would know without any data exchange where to refine (a certain mesh size might be prescribed, e.g.). For this case, Peano provides a function `enforceRefine` in the parallel mode that tells the code not to bother about data consistency and to refine everywhere. Basically, the user tells the grid that he accepts responsibility to call `enforceRefine` on all replicants of a vertex.

We have to be very careful to use the enforced refinement. We may use it if and only if there are no more idle nodes available and if all previous forks and joins have successfully passed through. This is furthermore information that is available only on the global master.

We thus propose to augment the `State.def` by a new flag:

```
#ifdef Parallel
enum GridConstructionState {
  Default, Veto, Aggressive
};

parallelise persistent GridConstructionState gridConstructionState;
#endif
```

Per rank, we now create a copy of the global state that is propagated through the tree (the state is used to determine the refinement strategy)

```
_localState = solverState;
```

and we furthermore plug into the mapping's `endIteration` on modify this global state on the global master. It has to be the global master as only the global master may ask the node pool about idle nodes. Furthermore, it knows the (fork/join) state of all other ranks. It is important to plug into `endIteration`, as this is the earliest point where we have all information about the grid state at hands. The underlying status variables all are accumulated, i.e. they might be already cleared in the subsequent `beginIteration`:

```
if ( tarch::parallel::Node::getInstance().isGlobalMaster() ) {
  solverState.updateRegularInitialGridRefinementStrategy();
}
```

The state transitions then are

```
void exahype::State::updateRegularInitialGridRefinementStrategy() {
  assertion( tarch::parallel::Node::getInstance().isGlobalMaster() );

  #ifdef Parallel
  if (
    tarch::parallel::Node::getInstance().getNumberOfNodes()==1
    ||
    _stateData.getGridConstructionState()==exahype::records::State::Aggressive
  ) {
    _stateData.setGridConstructionState( exahype::records::State::Aggressive );
  }
  else if (
    tarch::parallel::NodePool::getInstance().getNumberOfIdleNodes()==0
    &&
    isGridStationary()
  ) {
    _stateData.setGridConstructionState( exahype::records::State::Aggressive );
```

```
  }
  else if (
    isInvolvedInJoinOrFork()
    ||
    !isTraversalInverted()
    ||
    !isGridStationary()
  ) {
    _stateData.setGridConstructionState( exahype::records::State::Veto );
  }
  else {
    _stateData.setGridConstructionState( exahype::records::State::Default );
  }
  #endif
}

bool myproject::State::refineInitialGridInCreationalEvents() const {
  #ifdef Parallel
  return _stateData.getGridConstructionState() == myproject::records::State::Aggressive;
  #else
  return true;
  #endif
}

bool myproject::State::refineInitialGridInTouchVertexLastTime() const {
  #ifdef Parallel
  return _stateData.getGridConstructionState() != myproject::records::State::Veto;
  #else
  return false;
  #endif
}
```

and we invoke our refinement from the creational vertex events as well as `touchVertexLastTime` if the corresponding state predicates hold. Finally, we make the refinement fall back to an enforced refinement if the state permits us to do so:

```
void myproject::mappings::RegularMesh::refineVertexIfNecessary(
  exahype::Vertex& fineGridVertex,
  const tarch::la::Vector<DIMENSIONS, double>& fineGridH,
  bool isCalledByCreationalEvent
) const {
  if (
    fineGridVertex.getRefinementControl() == Vertex::Records::Unrefined
    &&
    I want to refine and know this a priori
  ) {
    #ifdef Parallel
    if (isCalledByCreationalEvent) {
      fineGridVertex.enforceRefine();
    }
    else {
      fineGridVertex.refine();
    }
    #else
    fineGridVertex.refine();
```

```
    #endif
  }
}
```

6.2.3 The load balancing kicks in immediately while I build up my grid but it yields non-reasonable partitions

> **Smell:** The code runs over the grid a couple of times and starts to distribute the domain level-by-level as we rely on an incremental build-up. This is however not clever as it turns out later throughout the grid construction that some regions are heavily refined while others remain coarse.

If this is the case, it might be reasonable to build up the grid up to a certain level where grid characteristics become visible, and to switch off the load balancing while you do so. Afterwards, load balancing can be enabled and should yield better results. Peano allows you to switch off the load balancing, but you have to do this on each individual rank.

```
peano::parallel::loadbalancing::Oracle::getInstance().activateLoadBalancing(false);
```

You might want to use

```
void myproject::runners::Runner::runGlobalStep() {
  // assertion( !peano::parallel::loadbalancing::Oracle::getInstance().
  // isLoadBalancingActivated() );

  peano::parallel::loadbalancing::Oracle::getInstance().activateLoadBalancing(false);
}
```

to switch off load balancing globally.

6.3 MPI quick tuning

Time: Around 15 minutes.
Required: A working MPI code.

This section collects a couple of really primitive measurements to make your code faster.

6.3.1 Filter out log statements

It is probably to simple to mention, but all our teams from time to time forget this. One of the major things slowing down codes is writing to the terminal. So adding a few additional log filters can significantly speed up your code.

6.3.2 Switch off load balancing

Most of Peano's load balancing algorithms (at least the ones coming along with the standard package) rely on a central node pool. If a rank decides that it would be advantageous to split up its domain, it sends a request to the first rank whether there are any idle nodes available. If your code already uses all ranks, this is a time consuming process that suffers from latency. If you know a prior that the load balancing is static and no further splits of subdomains are possible, it does make sense to switch the load balancing off. There is a routine `activateLoadBalancing` operation on the load balancing oracle to do so.

This operation has to be called on each individual rank, i.e. you can switch the load balancing on and off on a rank-per-rank basis. There are basically two variants/patterns to disable the load balancing:

1. You may introduce a new mapping that does nothing besides switching the load balancing off (typically in `beginIteration`). You then merge this mapping into your other adapters.
2. You add a new bool to your state. In the global runner you set this boolean flag once you want to switch the load balancing off. The state then is successively propagated to the workers. In `beginIteration`, you analyse this bool (in any mapping) and you switch off the load balancing if the flag is set.

Peano also offers the opportunity to invoke a global step on all ranks prior to an `iterate` call. This feature can be used to switch off the load balancing, too:

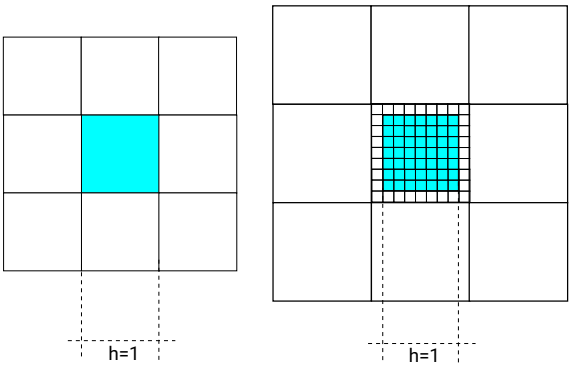
```
void picard::runners::Runner::runGlobalStep() {
  peano::parallel::loadbalancing::Oracle::getInstance().activateLoadBalancing(false);
}


int picard::runners::Runner::runAsMaster(...) {
  ...

  repository.runGlobalStep(); // on all other ranks
  runGlobalStep(); // and locally, too
}
```

As clarified in the documentation of the operations (see the autogenerated header files of your repository, e.g.), you have to be careful if you follow this variant: You are never allowed to run a global step if any rank is involved in a join or fork.

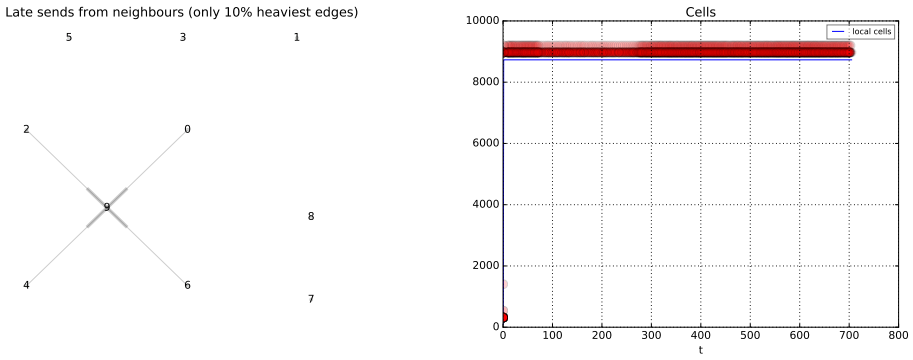
6.3.3 Reduce data exchange with global master

> **Smell:** It seems that rank 0 is involved in lots of data exchange (the performance analysis says that it is a neighbour data exchange bottleneck), and we see that rank 0 holds a significant number of total vertices and cells.

Peano takes the computational Domain (the unit square, e.g.) and embeds it into a 3^d patch. This surrounding patch is held by rank 0 being the global master. This rank deploys the central element, i.e. the whole domain, immediately to rank 1 and sticks himself with administrative duties (the node pool server realising domain decomposition decisions, e.g.) only. We have a situation as sketched below on the left:



The global master 0 deploys the real domain (filled) to rank 1 and then rank 1 continues to split up its domain further. Though rank 0 has deployed all cells to other ranks, still many workers of rank 1 (up to eight) are adjacent to rank 0. If they refine (and they most probably will do as most PDE solvers refine along the domain boundary), there is a pretty huge refined surface that connects each of the eight workers of rank 1 with rank 0. And now rank 0 becomes a bottleneck though rank 0 does no computation at all.



Smells on the global master in the performance analysis output. Standard run on unit square, regular grid, $d = 2$ with 10 ranks. Left: Rank 0 is a boundary data exchange bottleneck though it has deployed all of its work to rank 1. Right: Rank 0 holds a significant number of cells (solid line) cmp. to the cells on the other ranks (red dots) though it has deployed the domain completely.

One solution is to extend the bounding box around the computational domain by a halo region. For a unit square, using an offset of $-1/7 \times -1/7$ and a bounding box size of $9/7 \times 9/7$ has proven

of value. This way, all halo cells of rank 0 are sufficiently away from the domain's real boundary. So, if a worker of rank 1 refines, it does not share additional Vertices with rank 0. 0 is not a bottleneck anymore. The drawback of this approach is that the first few coarser levels on the ranks are not involved in any computation (makes a difference for multigrid, e.g.) as they overlap the whole computational domain. Only the third or fourth level of the spacetree actually holds valid compute data.

To identify whether you can benefit from this technique, try a simple run with a regular grid and only two ranks. In this case, you should not see any speedup, as all work is deployed by rank 0 to rank 1. However, you should also not observe a significant runtime penalty. If you do observe a penalty, try to realise this fix:

```
// the actual computational domain remains the same -we are not altering the
// physics
peano::geometry::Hexahedron geometry(
  tarch::la::Vector<DIMENSIONS,double>(1.0),
  tarch::la::Vector<DIMENSIONS,double>(0.0) );
myproject::repositories::Repository* repository =
  myproject::repositories::RepositoryFactory::getInstance().createWithSTDStackImplementation(
  geometry,
  tarch::la::Vector<DIMENSIONS,double>(9.0/7.0), // has been 1.0 before
  tarch::la::Vector<DIMENSIONS,double>(-1.0/7.0) // has been 0.0 before
  computationalDomainOffset );
```

6.4 Reduce MPI Synchronisation

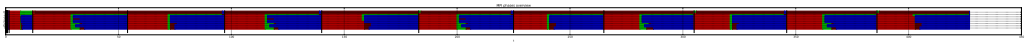
Time: Around 60 minutes.
Required: A working MPI code.

Peano has very strong constraints on the master-worker and worker-master communication as the data exchange between these two is synchronous. It imposes a partial order. If that slows down your application (you see this from the mpianalysis reports), you can kind of weaken the communication constraints. Often, some data is not required immediately, not required globally all the time, or doesn't have to be 100% correct at all algorithmic stages. This chapter discusses some things that you can do then.

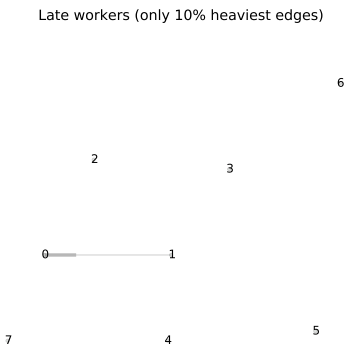
On the following pages, we assume that you have proper load balancing.

6.4.1 The smell

Strong synchronisation materialises in very regular patterns where each rank waits for rank 0 to start up a new traversal.



It also becomes obvious if you study how often a master has to work for its workers. In the picture below, only rank 0 synchronises the other ranks. In this case, you have to weaken the global synchronisation. If multiple of these edges pop up, it is time to weaken all the worker-master synchronisations—unless you can identify that you have a load balancing issue.



6.4.2 Weaken synchronisation with global master

The global master (rank 0) is kind of a pulse generator for the whole code. Whenever the `runAsMaster` operation triggers `iterate`, it tells each rank that handles a partition which adapter to use and to start its traversal or wait for its master to trigger the traversal, respectively. This is a very strong synchronisation. Notably, no rank can continue to work with the next iteration unless rank 0 runs into the next `iterate` as well. There are basically two variants to improve this situation:

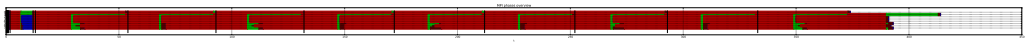
1. Perform more than one time step with the same adapter and settings in a row. For this, use the integer argument of `iterate()`. Note that running multiple time steps switches off load balancing for this phase of the program. Obviously, this version works if and only if you run the same adapter several times.

2. You may alternatively find out that you don't need the rank 0 (that doesn't hold any data anyway) to wait for all the other ranks in each iteration. Often, you run for example a sequence of adapters and you require global data (such as global residual) only after the last run. If you want to realise the second variant, you have to ensure that all mappings you use (also the predefined ones) return false in `prepareSendToWorker(...)`.

> **Remark:** If you want to validate that reductions have been skipped, switch on the log info of `peano::grid::nodes::Node::updateCellsParallel StateBeforeStoreForRootOfDeployedSubtree`.

Please be aware that reduction are skipped by the kernel if and only if all mappings allow the kernel to switch off the reduction. Furthermore, load balancing has to be disabled. If you want to load balance, master and worker ranks have to communicate with each other and may not skip any data/status exchange.

We also observe that the reduction skips often only change the communication profile but do not speed up the computation. Often it is only a preparatory step to switch off boundary data exchange afterwards. Once this is done, you should get a profile as below. It is more or less completely asynchronous, and all data exchange (blue) is hidden in the background, i.e. not visible anymore:



6.4.3 Postpone master-worker and worker-master data exchange

Take the communication specification of each mapping. By default, they are set to the most general case. Adopt it to your algorithmic needs (see documentation of the communication specification class)/

Introduce eager send and late receive

By default, Peano send away data from a local node if and only if it has traversed the whole local tree. In return, it requires all input data before it starts to traverse anything. You may want to tailor this to your needs and send data earlier and receive data later which allows you to overlap computations more aggressively. To do so, you have to adopt the communication specification fields of your mappings. See the documentation of the underlying class for more details. Avoid communication with rank 0

6.4.4 Skip worker-master data transfer locally/sporadically

6.5 Other ideas

If it doesn't, you can tune your load balancing. Prior to this, I however recommend that you write down a cost model - how expensive should one cell be to solve? Then you can use the load per cell individually in your mappings and thus guide the load balancing (actually any load balancing you intend later on to use) how costly different subtrees are. Doublecheck the multiscale concurrency

Peano relies on a modified depth-first (dfs) traversal. The parallel variant also is a dfs, but whenever the dfs traversal encounters a remote node, it makes another mpi rank traverse the corresponding spacetree, while it continues itself with the local subtree. Before is ascends again, it checks whether the remote subtree traversals have terminates as well. As a result, it is important to split up the tree on an as coarse level as possible to obtain a high concurrency Level. Let's study a toy problem in 1d:

foobar

In the upper picture, we have forked 2,4,7,8,10 and 12 to remote ranks while we stop the forking on level 1. As a result, our dfs descends into node 1, then forks 3 and 4, waits until they are done, continues with node 5, forks 7 and 8, does ist local 6, waits for 7 and 8 to finish, and continues with 9 forking 11 and 12. Obviously, the maximum concurrency level is two. If we change the decomposition into the lower splitting, the concurrency level is 7 (mind that 8 should have a different colour than 0 and 9, but that's only a visualisation relict).

Now, one has to Keep in mind that Peano forks only subtrees that have a certain regularity: Only nodes (and hence their children) can be forked where all $2^d adjacent vertices are refined. So, if we argue the other way r$
$to refine all vertices up to a certain level independent of your application needs) to allow the load balancer to fork a way sub$

Often, enforcing this kind of regularity is not possible within the mappings, as the mappings work basically inside the computational domain. Given the sketched situations, it can be advantageous however also to refine within obstacles or along complicated boundaries regularly. Therefore, peano::parallel::loadbalancing::OracleForOnePhase holds another attribute that you can uuse to enforce a certain grid regularity and to enable your code to fork more aggressively. Introduce administrative ranks and reduce algorithmic latency

Throughout the bottom-up traversal, each mpi traversal first receives data from all its children, i.e. data deployed to remote traversals, and afterward sends data to its master in turn. Unfortunately, Peano has to do quite some algorithmic work after the last children record has been received if and only if some subtrees are also to be traversed locally. It hence might make sense to introduce pure administrative ranks that do not take over any computation on the finest grid level. Again, we do a brief 1d toy case study:

foobar

In the upper case, the blue rank triggers the red one to traverse its subtree. The red one in turn tiggers 3 and 4. Afterward, it continues with 2 and then waits for 3 and 4 to finish. After the records from 3 and 4 have been received, it has to send its data to 0 to allow 0 to terminate the global traversal. However, between the last receive and the send, some administrative work has to be done, as the red node also holds local work (it has to run through the embedding cells to get the ordering of the boundary data exchange right, but that's irrelevant from a user point of view). This way, we've introduced an algorithmic latency: Some time elaps between 3 and 4 sending their data and the red one continuing with the data flow up the tree. This latency becomes severe for deep Splittings.

In such a case, it is a better idea to make the red one fork all of its work. See the lower part of the Illustration. In this case, (almost) no local administration is required, i.e. 1 accepts the finished Messages from 2,3 and 4 and almost immediately passes on the token to 0. Now, 1 basically does no work and you introduce a bad balancing here. But you have mpi rank overloading to compensate for this. And a latency reduction usually is more important. Exploit overloading

Peano's parallelisation is based upon tree-splits, i.e. the code can 'only' deploy whole subtrees to other ranks. Imbalances thus are always built-in. They become the more severe the fewer mpi ranks one uses. Thus, one has to check carefully whether mpi overbooking pays off. A general rule of thumb is that the smaller the computational workload the higher the overbooking should be. For codes with an extremely low compute load (just moving data, e.g.), overbooking by a factor

of four on SandyBridge seems to be the method of choice. In that case, you start 64 mpi ranks per node. On SuperMUC, you have to restrict yourself to 32 ranks per node due to a load leveler constraint.

Peano provides both mpi and shared memory parallelisation. I currently recommand to use the TBB variant of the latter. Following the overbooking discussion above, it does make sense to equip each mpi rank with t threads though the total number of threads then outnumbers the number of cores by far. This way, some mpi ranks might become idle throughout the computation, but their cores are grapped by other mpi ranks due to their many tbb threads eventually. Pays off in most cases ... as long as the shared memory scales for reasonably fine grids and as long as your grid has such regular subregions. Optimise worker-master commmunication

Given the output of the component mpianalysis (either via text file or the postprocessing script), you can identify whether the masters had to wait for their workers a significant time. If that is the case, i.e. if you observe late workers,

try to balance your workload better, or even try to undersubscribe the workers.

In the latter case, you try to assign nodes acting both as compute nodes and as masters a bigger workload than those without any workers to administer. The rationale is that nodes far away from the global master in the call hierarchy may not delay the time per traversal as any delay there has a huge impact. Consequently, it is better to assign them a smaller workload and to give them time to send away their finished messages (that also have to run through the network). One avoids algorithmic latency. The price to pay is a non-optimal workload balancing.

If a node delays ist master and you cannot change its workload, study its individual runtime profile. If the code spends a signficant time within its boundary exchange, this means that it needs this significant time to wait until mpi has released its last send and receive request and other nodes have delivered their data. Now, if all workload is reasonable balanced, you cannot do anything about the latter fact. However, it might make sense to try a different buffer size. As the code spends all this time to wait for the last peace of data to arrive and to release its current buffer, a smaller buffer size might lead to a situation where the nodes can exchange more data in the background. Splitting up the buffer into smaller chunks then is an option, i.e. to reduce the buffer size. Be Aware that smaller buffer sizes increase the administrative overhead. A too small buffer size hence slows down the code.

If nodes delay their masters but are not suffering from data exchange, you have to reduce their workload. If that is not possible and if your code runs correctly (read: no deadlocks), it makes sense to try to switch Peano's communication protocols to blocking send. For this, you have to switch the corresponding flag in your compiler specific settings. Send them to sleep

If several ranks are booked to one node, they compete for the network facilities. This competition can slow down the overall system: If one rank is done, it listens for a new message from its master. The other ranks however might still need the network to exchange data and thus are throtteled. For avoid this, you might think either to switch to a real blocking call (given that your MPI implementation realises this internally via an interrupt) or send threads waiting for a synchronous message to sleep.

To do so, you have to go the compiler-specific settings and introduce a sleep penalty. The compiler flags are called ReceiveMasterMessagesBlocking or similar. The allowed values are documented. Switch off reduction

The reduction of data along the spacetree often harms Peano's performance significantly. Check whether you can live (at least for some iterations) without the reduction. Often, e.g., load balancing and reduction are important in time stepping, but one can always do few linear algebra traversals without reducing any global data.

Peano's iterate method can be passed false as argument. Then, the reduction is avoided. And your code should run faster and not suffer from latency and ill-balancing. Not that much at least.

Please note that there are two different types of reduction: Peano by default transers vertices and cells bottom-up and thus, e.g., allows for load balancing. This is the behaviour you can switch off due to the flag. However, note that load balancing relies on reduced data, i.e. if you switch this off, you also disable load balancing. A different story is the user-defined reduction. Many applications reduce data in their services (if they have such global object instances per node) or

send data to the master in their events. This is a reduction you have to handle correctly. Diving into implementation details

The file peano::utils::PeanoOptimisation holds a number of different defines that influence Peano's runtime behaviour. With respect to MPI the compile arguments

-DnoParallelExchangePackedRecordsAtBoundary -DnoParallelExchangePackedRecordsBetweenMasterAndWork
-DnoParallelExchangePackedRecordsInHeaps -DnoParallelExchangePackedRecordsThroughoutJoinsAndForks

do have impact on the communication behaviour. They tell Peano not to reduce the memory footprint of the messages prior to sending them away. If you switch them off, you increase the bandwidth required (perhaps only slightly) but you skip the marshalling and unmarshalling steps. This might yield significant speedup but depends strongly on the PDE-specific data exchanged.

Besides the four flags from above, there are some more settings that might interplay with the scaling of your application. But here, everything is trial-and-error. Become topology-aware

Peano uses a node pool strategy to decide which rank assists which other rank if a rank asks for additional workers. By default, this strategy is FCFS and the ranks are just handed out without additional considerations. It thus can happen that all big partitions are assigned to the first p' ranks whereas the remaining p-p' ranks become responsible for rather small subgrids only.

In such a case it often does make sense to implement topology-awareness into your node pool server (it also might make sense to add problem-awareness, i.e. knowledge about your grid, to the oracle, but that is a different story). A simple example for such a generic server can be found in the toolboxes. It assumes that there are k ranks assigned to each compute node. As a result, it assigns work to the ranks in the order 1, k, 2k, 3k, ..., 2, k+1, k+2, and so forth. It realises a modulo work assignment. This reduces the memory required per compute node and it also distributes the communication data footprint evenly.

7 Troubleshooting

7.1 PDT

- **The PDT does not pass as the jar file is built with the wrong Java version.** Download the whole Peano project (from sourceforge via subversion), change into the directory `pdt`, and run

```
ant createParser
ant compile
ant dist
```

  Use the `pdt.jar` from the `pdt` directory now.

7.2 Programming

- **How can I find out the location of a cell**. See the documentation of the `VertexEnumerator`. The object has routines to query cell position, size and level in the spacetree.