ExaHyPE Guidebook

http://www.exahype.eu

August 31, 2016

The project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 671698 (ExaHyPE).

Preamble

ExaHyPE stands for an "Exascale Hyperbolic PDE Engine", where an "PDE Engine" represents the novel idea of an software suite comparable to a game engine. However, instead of offering all the neccessary functions to render 3D scenes, manage the gameplay and all that (as a game engine would do), our PDE Engine offers similar ways to solve particular systems of Partial Differential Equations.

This document is very much about the technical aspects how the program suite is working and meant to be used. To read more about the project, we refer to the official website http://www.exahype.eu as well as the objectives at the cordis page.

ExaHyPE is merely a collection of open source tools and mainly written in C, C++, FORTARN, Java, Matlab and Python.

Who should read this document

This guidebook is written similar to a tutorial in a hands-on style. It addresses users of ExaHyPE. It is not meant to help developers or replace code documentation. Using the guidebook requires a decent knowledge of the application domain modelled via a hyperbolic equation system. It does not require deep programming knowledge.

Everything in this guidebook runs out-of-the-box by following the written text only. There are however few additional steps that advanced users might want to try out. Those do not come along with very detailed step-by-step descriptions. They are marked with an (optional) postfix in the title. Standard or new users should just skip these sections.

August 31, 2016
The ExaHyPE team

Contents

1. Setup and Installation

1.1. Dependencies and prerequisites

ExaHyPE's core routines can be used if you have a recently new C++ compiler (C++11 has to be supported) and Java runtime environment (JRE) installed.

- `ExaHyPE` source code 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. All examples from this guidebook run and have been tested with GCC 4.2 and Intel 12 or later. Earlier compiler versions might work.
- ExaHyPE's setup environment relies on Java, as parts of it come along as jar files. Java 1.6 or newer is needed.
- ExaHyPE's default build environment uses GNU Make among with standard Unix commands (POSIX).

The guidebook 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 and during development it is not supported.

1.2. Obtaining ExaHyPE

`ExaHyPE` does not require any real installation. It is basically just a download.

1.2.1. Downloading an official ExaHyPE distribution

In the future, you will be able to just visit www.exahype.eu to get particular `ExaHyPE` source code variants. All remarks here refer to the latest `ExaHyPE` release.

The following text assumes you live in the future, where there are official well open-sourced distributions at download servers available. This will be sometimes in 2017, but it is not the presence. Please read the next section in order to learn how to get ExaHyPE already today.

Each release consists of three major source code packages:

1. The actual `ExaHyPE` sources is a `tar.gz` archive. It contains the actual engine with all mesh organisation, computational kernels, load balancing components, and so forth.

2. ExaHyPE is based upon the PDE solver framework Peano (peano-framework.org) which provides the underlying adaptive mesh refinement (AMR) data structures as well as mesh traversal algorithms.

3. The engine is used through a Java toolbox that provides the opportunity to configure the engine to a particular setup. The sources for the latter typically are not required, but they are available from the ExaHyPE home page.

A complete download/setup reads as follows:

```
> cd mywellsuiteddirectory
> wget http://www.exahype.eu/exahype.tar.gz
> wget http://sourceforge.net/projects/peano/files/peano.tar.gz
> wget http://www.exahype.eu/exahype.jar
> tar -xzf exahype.tar.gz
> ls
Applications ExaHyPE exahype.jar exahype.tar.gz peano peano.tar.gz
> tar -xzf peano.tar.gz -C peano
```

You might choose to maintain a different directory structure or rely on a previous Peano or ExaHyPE installation. In this case, you have to adopt pathes in your ExaHyPE scripts.

1.2.2. Setting up an Developer ExaHyPE machine

Currently, in order to install ExaHyPE, you have to work with the revision control system git. To start, copy and paste this mini installation guide into your Linux terminal:

```
git clone git@gitlab.lrz.de:gi26det/ExaHyPE.git
cd ExaHyPE/Code/Peano
tar xvfz peano.tar.gz
git checkout .gitignore
cd ..
ln -s Toolkit/dist/ExaHyPE.jar
```

This is the current real-world pendant to what is mentioned in the previous section. As you see, the central ressource where the code is managed is located at https://gitlab.lrz.de/gi26det/ExaHyPE. This location is currently closed sourced and you have to get in touch to the ExaHyPE team to get access.

1.3. Dry run

To check whether you are ready to use ExaHyPE, type in

```
> java -jar ExaHyPE.jar
```

This should give you the following welcome message:

```
================================
 ___           _  _      ___ ___
/ __|_ ____ _| || |_  _/ _ \ __|
| _|\ \ / _` | __ | || |  _/ _|
\___/_\_\__,_|_||_|\_, |_| \___|
                   |__/
================================

 www.exahype.eu

================================

The project has received funding from the European Unions
Horizon 2020 research and innovation programme under grant
agreement No 671698 (ExaHyPE). It is based upon the PDE
framework Peano (www.peano-framework.org).


ERROR: Please provide input file as argument
```

If you don't see this message, there's a problem with the java based ExaHyPE toolbox.

1.4. Optional installation variants

1.4.1. Using a newer/other **Peano** version

ExaHyPE is based upon the PDE solver framework Peano (peano-framework.org). With the present scripts, you use the newest stable Peano release. As Peano is still improved, it might make sense to download a new archive from time to time.

ExaHyPE uses a couple of Peano toolboxes. These toolboxes are part of Peano's framework, but they are not by default included in the release. While normal Peano users have to download them manually, we provide a snapshot with ExaHyPE. These snapshots are help in ExaHyPE's Peano directory. If you want to try newer versions, download the corresponding toolboxes from Peano's webpage and extract them manually into ExaHyPE's Peano subdirectory.

If you use Peano in several projects, it might make sense to skip the download of the archive into ExaHyPE's Peano directory and instead add two symbolic links with `ln -s` in the Peano directory to Peano's peano and tarch directory.

2. Tutorial: A first experiment

ExaHyPE is written for high performance computing experiments. Our philosophy is *write one code specification per experiment/machine/setup configuration* and sacrifice flexibility in exchange for performance. Because of this, ExaHyPE specification files act on the one hand as configuration files that are passed into the executable for a run. On the other hand, a specification file tailors the executable, i.e. is used to generate some source code. The compiler can, at hands of the specification, optimise aggressively in several places. ExaHyPE prefers to run the compiler more often rather than working with one or few generic executables.

An ExaHyPE specification is a text file where you specify exactly what you want to solve in which way. This file[1] is handed over to our ExaHyPE toolkit, a small Java application generating all required code. This code (also linking to all other required resources such as parameter sets or input files) then is handed over to a compiler. You end up with an executable that may run without the Java toolkit or any additional sources from ExaHyPE. It however rereads the specification file again for input files or some parameters, e.g. We could have worked with two different files, a specification file used to generate code and a config file, but decided to have everything in one place.

2.1. Setting up trivial demonstration project

A trivial project in ExaHyPE is described by the following specification file:

```
exahype-project TrivialProject
  peano-path = ./Peano/peano
  tarch-path = ./Peano/tarch
  multiscalelinkedcell-path = ./Peano/multiscalelinkedcell
  sharedmemoryoracles-path = ./Peano/sharedmemoryoracles
  exahype-path = ./ExaHyPE

  output-directory = ./Applications/trivialproject

  architecture = noarch

  computational-domain
    dimension = 2
    width = 1.0, 1.0
```

[1] Some examples can be found in the Toolkit directory or on the ExaHyPE webpage. They have the extension exahype.

```
    offset = 0.0, 0.0
    end-time = 10.0
  end computational-domain
end exahype-project
```

Most parameters should be rather self-explanatory. You might have to adopt some of the paths. Note ExaHyPE supports both two- and three-dimensional setups. We hand the file over to the toolkit

```
java -jar ExaHyPE.jar examples/trivial-project-2d.exahype
```

and observe that the toolkit has created a new directory `Applications/trivialproject`. We change into this directory and type in

```
make
```

which gives us an executable. The toolkit plots additional information how to switch into debug versions of the code or how to switch to another compiler through environment variables. For most systems, the defaults should do. We finally run this first ExaHyPE application with

```
./ExaHyPE <yourpath>/trivial-project-2d.exahype
```

Within the section `computational-domain`, all entries except the `dimension` can be changed before application startup: If you change their value, there is no need to recompile or to rerun the toolkit again.

2.2. Introducing physics: 2d Euler flow

So far, the simulation run neither computes anything nor does it give you any output. We close this section with a brief demonstration how to realise a 2d Euler flow within ExaHyPE. ExaHyPE's fundamental idea is that the user specifies all PDEs she wants to solve within the specification file. We start with a solve of the compressible Euler equations via the ADER-DG method. For this type of solver, the engine needs to know how many unknowns the solved PDE has and which polynomial order shall be used. We do not have any material involved, so we set the field parameters to 0.

For the actual computations, there are three types of kernels (PDE solver realisations) that can be used within ExaHyPE. You can either write the whole solver yourself, you can make ExaHyPE realise the solver while you specify the flux update operations, or you can use a pre-defined ExaHyPE kernel which gives you the best performance. Available options are described and detailed in Chapter D. For the time being, we use `kernel = generic::fluxes::nonlinear`, the generic kernels for non-linear PDEs, where PDE-specific functions, e.g. the flux function, yet have to be specified by us.

In the present example, we rely on the latter variant. We also stick to a regular grid simulation for the time being. The specification file then is

```
exahype-project MyEuler
```

```
  [...]

  output-directory = ./Applications/guidebook-chapter-2
  architecture = noarch

  solver ADER-DG MyEulerSolver
    unknowns = 5
    parameters= 0
    order = 3
    kernel = generic::fluxes::nonlinear
    language = C
  end solver

end exahype-project
```

We run the toolkit and end up with a new directory `Applications/guidebook-chapter-2`. Within this directory, there is a file `MyEulerSolver.cpp` that we have to complete as follows:

```
#include "MyEulerSolver.h"

MyEuler::MyEulerSolver::MyEulerSolver(int nodesPerCoordinateAxis, double maximumMeshSize, e
  exahype::solvers::ADERDGSolver("MyEulerSolver", 5, 0, nodesPerCoordinateAxis, maximumMesh
}

int MyEuler::MyEulerSolver::getMinimumTreeDepth() const {
  return 3;
}

// ...

void Euler::MyEulerSolver::adjustedSolutionValues(...) {
  if (tarch::la::equals(t, 0.0)) {
    const double GAMMA = 1.4;

    Q[0] = 1.;
    Q[1] = 0.;
    Q[2] = 0.;
    Q[3] = 0.;
    Q[4] =
        1. / (GAMMA -1) +
        std::exp(-((x[0] -0.5) *(x[0] -0.5) + (x[1] -0.5) *(x[1] -0.5)) /
                (0.05 *0.05)) *
            1.0e-3;
  }
}

bool MyEuler::MyEulerSolver::hasToAdjustSolution(...) {
  if (tarch::la::equals(t, 0.0)) {
```

```
    return true;
  }
  return false;
}
```

Actually, only proper initial conditions have to be provided. The remainder is pre-generated.

To set the initial conditions, we need to implement two of the pregenerated functions: We specify that we want to adjust the solution values at time $t = 0$ in function `hasToAdjustSolutionValues` and then set initial conditions via the function `adjustedSolutionValues`. We clarify the semantics of the arguments of `hasToAdjustSolutionValues` and `adjustedSolutionValues` in Chapter 4. For the time being, we compile and run the code.

2.3. A remark on material parameters

In ExaHyPE, we do not really distinguish unknowns that are subject to a PDE from parameters on the user side. If you have 4 unknowns and 3 (material) parameters, all functions just seem to handle 4+3=7 variables. We only distinguish the values in the script file as we obviously do not want to run any PDE kernel (unknown update) on top of the material parameters. For them, we restrict to a subset of all the variables—namely the first 4 in the example above. To the user, we do not distinguish material parameters from unknowns. When you set up the initial condition, e.g., you will be passed 7 unknowns.

2.4. Introducing Plots

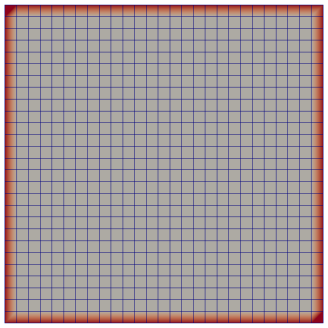
Still, the simulation does not give any output besides status messages on the terminal. We have to add appropriate plotters. `ExaHyPE` offers a couple of predefined export mechanisms. Within each `solver` section, we may use an arbitrary number of plotters to pipe the result at certain time steps into files. We start with

```
...
solver ADER-DG MyEulerSolver
  unknowns = 5
  parameters= 0
  order = 3
  kernel = generic::fluxes::nonlinear
  language = C

  plot vtk::binary
    unknowns = 5
    time = 0.0
    repeat = 0.02
    output = ./solution
    select = {}
```

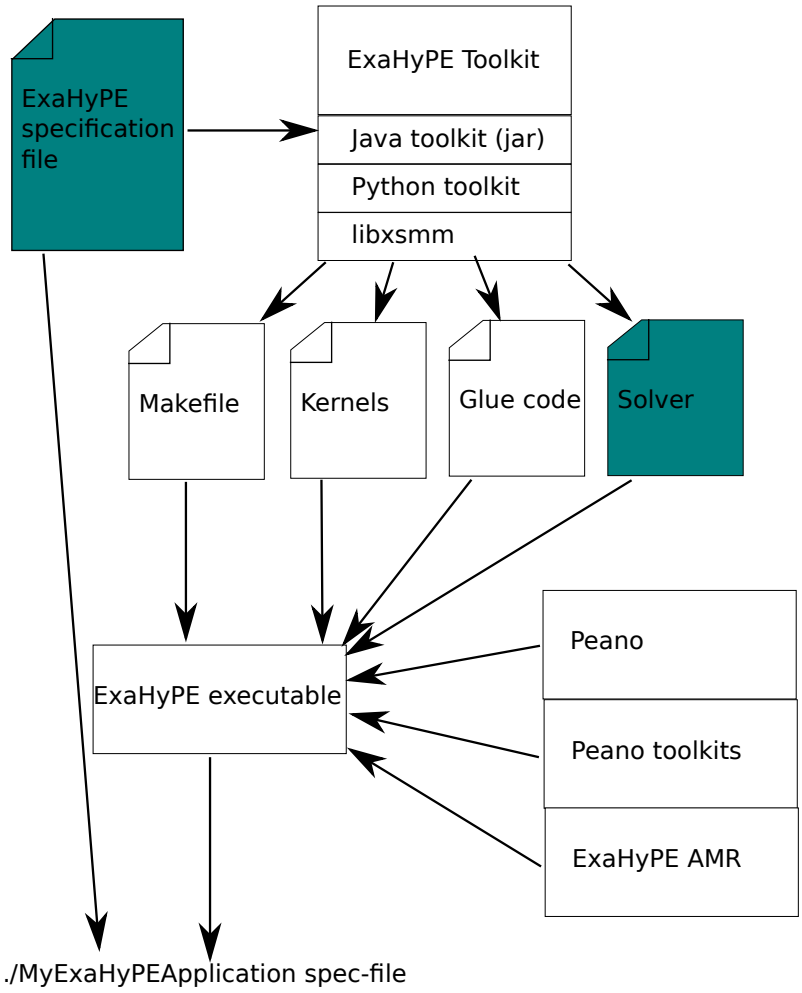
```
  end plot
end solver
...
```

and postpone a detailed explanation to later chapters. For the time being, it is sufficient to note that this operation writes VTK binary files (they can be read in by Paraview or VisIt, e.g.) into files named `solution...  .vtk`. The first file is written at time 0, i.e. at startup time. From hereon, we obtain a new snapshot every 0.02 simulation intervals.

Once you have inserted the plotter into your specification file, invoke the toolkit again, compile the code and run the simulation. You obtain a sequence of files yielding images similar to



3. **ExaHyPE** architecture



1. The user writes a specification file (text format) that holds all required data ranging from paths, which parallelisation to use, computational domain up to a specification which kind of solvers will exist and which numerical schemes they realise.

2. This specification file is handed over to the ExaHyPE toolkit which is a Java tool. It internally relies on Python scripts and invokes the libxsmm generator driver as well. A local build of the libxsmm's code generation driver is therefore a prerequisite for using optimised kernels.

3. The toolkit yields a couple of files (Makefile, glue code, helper files, ...). Among them is also one C++ implementation class per solver that was spec-

ified in the specification file. The output directory in the specification file defines where all these generated files go to.

4. a) Within each C++ implementation file, the user can code solver behaviour such as initial conditions, mesh refinement control, and so forth.

   b) The whole build environment is generated. A simple `make` thus at any time should create the `ExaHyPE` executable.

5. If you run your `ExaHyPE` executable, you have to hand over the specification file again. Obviously, many entries in there (simulation time or number of threads to be used) are not evaluated at compile time but at startup. You don't have to recompile your whole application if you change the number of threads to be used.

To summarise, the blueish text files are the only files you typically have to change yourself. All the remainder is generated and assembled according to the specification file.

3.1. Switching between Generic and Optimised Kernels

Driven by the choice in the specification file, `ExaHyPE` chooses either generic kernels or optimised kernels. If you switch between the generic and the optimised kernels, you have to use the toolkit to regenerate the glue code.

4. ADER-DG solvers with user-defined kernels

In this chapter, we introduce ExaHyPE's kernels that allow the user to specify flux and eigenvalue functions for the kernels (for linear problems), but leaves everything else to ExaHyPE. The resulting code can run in parallel and scale, but its single node performance might remain poor, as we do not rely on ExaHyPE's high performance kernel. Therefore, the present coding strategy is well-suited for rapid prototyping of new solvers. To outline the steps a user has to perform to define his own fluxes and state-depending initial values, we again consider the compressible Euler equations with $d = 2$. This is the example we also used in Chapter 2.

For the Euler equations

$$\frac{\partial}{\partial t}\mathbf{Q} + \boldsymbol{\nabla} \cdot \mathbf{F}(\mathbf{Q}) = 0 \quad \text{with} \quad \mathbf{Q} = \begin{pmatrix} \rho \\ \mathbf{v} \\ E \end{pmatrix} \quad \text{and} \quad \mathbf{F} = \begin{pmatrix} \mathbf{v} \\ \frac{1}{\rho}\mathbf{v} \otimes \mathbf{v} + pI \\ \frac{1}{\rho}\mathbf{v}\,(E+p) \end{pmatrix}$$

supplemented by initial values $\mathbf{Q}(0) = \mathbf{Q}_0$ and appropriate boundary conditions, we rewrite the conserved quantities $\mathbf{Q}$ and fluxes $\mathbf{F}$ into an equation over five state unknowns

$$\mathbf{Q} = \begin{pmatrix} Q_0 \\ Q_1 \\ Q_2 \\ Q_3 \\ Q_4 \end{pmatrix} = \begin{pmatrix} \rho \\ v_x \\ v_y \\ v_z \\ E \end{pmatrix}$$

which are discretised over Ω.

Here, ρ denotes the mass density, $\mathbf{v} \in \mathbb{R}^d$ denotes the velocity vector, E denotes the energy, and p denotes the fluid pressure. For our 2d setup, the velocity in z-direction v_z is set to zero.

Introducing the adiabatic index γ, the fluid pressure is here defined as

$$p = (\gamma - 1)\left(E - \frac{1}{2}\mathbf{v}^2/\rho\right).$$

A corresponding specification file `euler-2d.exahype` for this setup is

```
exahype-project Euler2d

  peano-path = ./Peano/peano
  tarch-path = ./Peano/tarch
```

```
  multiscalelinkedcell-path = ./Peano/multiscalelinkedcell
  sharedmemoryoracles-path = ./Peano/sharedmemoryoracles
  exahype-path = ./ExaHyPE
  output-directory = ./ApplicationExamples/EulerFlow
  architecture = noarch

  computational-domain
    dimension = 2
    width = 1.0, 1.0
    offset = 0.0, 0.0
    end-time = 0.4
  end computational-domain


  solver ADER-DG MyEulerSolver
    variables = 5
    parameters = 0
    order = 3
    maximum-mesh-size = 0.1
    time-stepping = global
    kernel = generic::fluxes::nonlinear
    language = C

    plot vtk::binary
      variables = 5
      time = 0.0
      repeat = 0.05
      output = ./solution
      select = {}
    end plot
  end solver
end exahype-project
```

The script sets some paths in the preamble before it specifies the computational domain and the simulation time frame in the environment `computational-domain`.

In the next lines, a solver of type `ADER-DG` is specified and assigned the name `MyEulerSolver`. The kernel type of the solver is set to `generic::fluxes::nonlinear`. This tells the `ExaHyPE` engine that we do not provide it with problem specific and highly optimized ADER-DG kernels. Instead we tell it to use some generic partially optimized ADER-DG kernels that can be applied to virtually any hyperbolic problem. In this case, the user is only required to provide the `ExaHyPE` engine with problem specific flux (and eigenvalue) definitions.

Within the solver environment, there is also a plotter specified. The specified plotter is configured to write binary VTK output by setting its identifier to `vtk::binary`. The plotter is further configured to write out a snapshot of the solution of the associated solver every 0.05 time intervals. The first snapshot is set to be written at time $t = 0$.

Once we are satisfied with the parameters in our ExaHyPE specification file, we hand it over to the ExaHyPE toolkit:

```
java -jar <mypath>/ExaHyPE.jar Euler2d.exahype
```

4.1. Study the generated code

We obtain a new directory (equalling `output-directory` from the specification file) and change into this file. It contains a makefile, some helper files, and a new class `MyEulerSolver`. This new class was generated according to the `solver` entry `ADER-DG MyEulerSolver` that we have specified in our script.

Class `MyEulerSolver` is divided into three source code files: First, there is a header that we might want to extend later on. For the time being, it suits our needs. There is further an implementation file with an addendum `_generated`. This is a file we should never edit since it is overwritten every time we use the toolkit to generate code for our problem. Finally, there is another implementation file that we will use to realise the actual solver behaviour. Here, we will specify the aforementioned user defined flux and eigenvalue definitions that are necessary if we want to use the generic ADER-DG kernels.

Before we start to do so, it might be reasonable to compile the overall code once and to doublecheck whether we can visualise the resulting outputs. This should all be straightforward. Please verify that any output you visualise holds only zero values since the initial values are set to zero at the moment.

If you translate in debug mode or with assertions, ExaHyPE might even stop with an assertion.

4.2. Setting up the experiment

To set up the experiment, we open the implementation file `MyEulerSolver.cpp` of the class `MyEulerSolver`. There is one routine `adjustedSolutionValues` that allow us to setup the initial grid. The implementation of the initial values might look as follows[1]:

```
void Euler2d::MyEulerSolver::adjustedSolutionValues(
  const double *const x,
  const double w,
  const double t,
  const double dt,
  double *Q
) {
 if (tarch::la::equals( t,0.0,1e-15 )) {
   const double GAMMA = 1.4;
```

[1]The exact names of the parameters might change with newer ExaHyPE versions and additional parameters might drop in, but the principle always remains the same.

```
  Q[0] = 1.;
  Q[1] = 0.;
  Q[2] = 0.;
  Q[3] = 0.;
  Q[4] = 1./(GAMMA-1) + std::exp(
           -((x[0]-0.5)*(x[0]-0.5) + (x[1]-0.5)*(x[1]-0.5))/(0.05*0.05)
         ) *1.0e-3;
  }
}
```

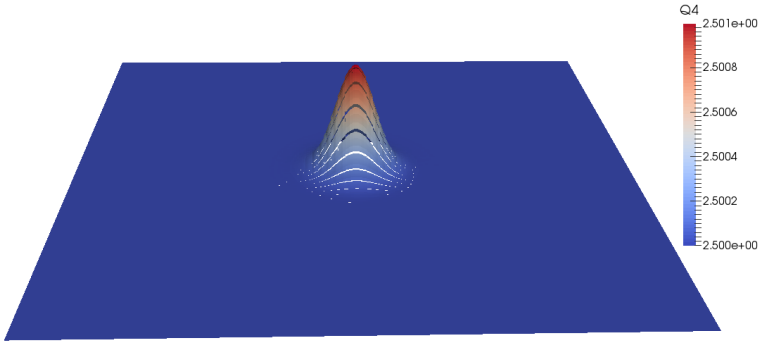
The above routine enables us to specify time dependent solution values. It further enables us to add source term contributions to the solution values. While the routine itself is more or less self-explaining, we nevertheless highlight a few details:

- We only want to impose initial conditions so we check if the time `t` is zero with respect to some tolerance in the first line of the routine.
- While the unknown pointer `Q` itself does not give any hint about the actual size of the array it points at, we know by construction that we have five unknowns per grid point and thus choose the iteration space accordingly. Note that as usual for `C` type languages indexing of the array starts with zero.
- The spatial position `x` is given as a pointer to a double array of size `dimension` which you have previously specified in the script `euler-2d.exahype`. Note that the indexing of the arrays starts here of course also with zero.

The routine `adjustedSolutionValues` must always be paired with the routine `hasToAdjustSolut` that specifies when and in which grid cell we want to adjust solution values:

```
bool Euler2d::MyEulerSolver::hasToAdjustSolution(
  const tarch::la::Vector<DIMENSIONS,double>& center,
  const tarch::la::Vector<DIMENSIONS,double>& dx,
  double t
) {
  if (tarch::la::equals( t, 0.0 ,1e-15 )) { // @todo precision
    return true;
  }
  return false;
}
```

The routine's arguments `center` and `dx` refer to the center point of the cell and to its extent in each coordinate direction. The simulation time is here again denoted by `t`.
We conclude our experimental setup with a compile run and a first test run. This time we get a meaningful image and definitely not a program stop for an assertion, as we have set all initial values properly. However, nothing happens so far; which is not surprising given that we haven't specified any fluxes yet.



4.3. Realising the fluxes

To realise the fluxes, we have to fill the functions `flux` and `eigenvalues` in file `MyEulerSolver.cpp` with code. As we rely on the kernel mode `generic::fluxes::nonlinear`, ExaHyPE autonomously realises all steps of ADER-DG and only calls back to the solver for the flux and eigenvalue evaluations.

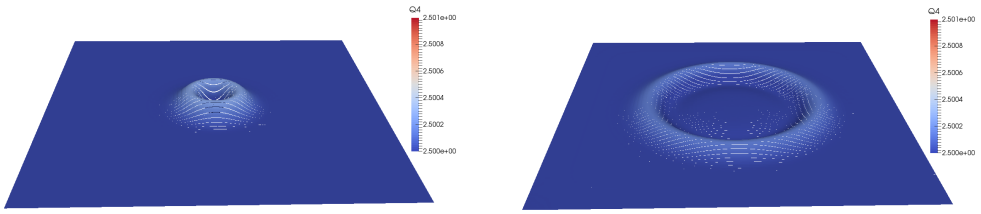
```
void Euler2d::MyEulerSolver::flux(
  const double* const Q,
  double** F
) {
  double* f = F[0];
  double* g = F[1];

  const double GAMMA = 1.4;

  const double irho = 1.0/Q[0];
  const double p = (GAMMA-1)*( Q[4] -0.5* (Q[1]*Q[1] + Q[2]*Q[2]) *irho );

  f[0] = Q[1];
  f[1] = irho*Q[1]*Q[1] + p;
  f[2] = irho*Q[1]*Q[2];
  f[3] = irho*Q[1]*Q[3];
  f[4] = irho*Q[1]*(Q[4]+p);

  g[0] = Q[2];
  g[1] = irho*Q[2]*Q[1];
  g[2] = irho*Q[2]*Q[2] + p;
  g[3] = irho*Q[2]*Q[3];
  g[4] = irho*Q[2]*(Q[4]+p);
}
```

```
void Euler2d::MyEulerSolver::eigenvalues(
  const double* const Q,
  const int normalNonZeroIndex,
  double* lambda
```

```
) {
  const double GAMMA = 1.4;

  double irho = 1.0/Q[0];
  double p = (GAMMA-1)*( Q[4] -0.5* (Q[1]*Q[1] + Q[2]*Q[2]) *irho );

  double u_n = Q[normalNonZeroIndex+1] *irho;

  double c = std::sqrt(GAMMA *p *irho);

  lambda[0] = u_n-c;

  lambda[1] = u_n;
  lambda[2] = u_n;
  lambda[3] = u_n;

  lambda[4] = u_n+c;
}
```

The implementation of function `flux` is again very straightforward. The code basically mimics the definition of the flux tensor $\mathbf{F(Q)} = (\mathbf{f}(\mathbf{Q}), \mathbf{g}(\mathbf{Q}))^{\mathrm{T}}$. As the creators of the `Euler2d` project, we naturally know the size of the parameters `Q` ,f, and g which is identical to the number of state unknowns. The same holds for the size of the array of eigenvalues the pointer `lambda` appearing in the signature and body of function `eigenvalues` refers to which is also identical to the number of state unknowns.

The normal vectors that are involved in ExaHyPE's ADER-DG kernels always coincide with the (positively signed) Cartesian unit vectors. Thus, the `eigenvalues` function is only supplied with the index of the single component that is non-zero within these normal vectors. In function `eigenvalues`, the aforementioned index corresponds to the parameter `normalNonZeroIndex`. A rebuild and rerun should yield results similar to



4.4. Adding some runtime constants

There are various ways to add application-specific constants to your code. If you want to have them in the specification file, ExaHyPE allows you to add a constants section to your solver. Please note that our parser is not particular powerful, i.e. the parameters in the constants' brackets may not have any whitespaces.

```
  solver ADER-DG MyParameterisedEulerSolver
    variables = 5
    parameters = 0
    order = 3
    maximum-mesh-size = 0.5
    time-stepping = global
    kernel = generic::fluxes::nonlinear
    language = C
    constants = {rho:0.4567,gamma:-4,alpha:4.04e-5,file:output}
  end solver
```

If you rerun the toolkit now, you'll get a modified constructor that accepts a `ParserView` object which you can query for the constants.

```
Euler::MyParameterisedEulerSolver::MyParameterisedEulerSolver(
  ...,
  exahype::Parser::ParserView constants):
  exahype::solvers::ADERDGSolver(...) {

  if (constants.isValueValidDouble( "rho" )) {
    double rho = constants.getValueAsDouble( "rho" );
    // do some magic with rho
  }
}
```

It is now a convenient exercise to make your simulation code run several solvers simultanously in the code. For this, add multiple solver sections to the specification file. If all solvers solve the same PDE, it is a natural choice to make one solver invoke fluxes, initial conditions, ...from the other solvers. `ExaHyPE` does not support (yet, i.e. in this chapter) a feature to reuse one solver multiple times. As you have different solver sections in your specification file, you can make them tackle different constants. This way, you save grid construction overhead (the actual grid holding the solvers is maintained only once), and you typically increase the arithmetic intensity of your code. It thus should scale better.

4.5. Time stepping strategies

`ExaHyPE` provides various time stepping schemes for both the ADER-DG solvers and the Finite Volume solvers. You can freely combine them for the different solvers in your specification file. Please note that some schemes require you to recompile your code with the flag `-DSpaceTimeDataStructures` as they need lots of additional memory for their realisation. At the same time, it would be stupid to invest that much memory, i.e. to increase the memory footprint, if these flags were defined all the time.

If you combine various time stepping schemes, please note that the most restrictive scheme determines your performance. `ExaHyPE` has time stepping schemes that try to fuse multiple time steps, e.g., and thus is particulary fast on distributed

memory machines where synchronisation between time steps can be skippted. However, if you also have a solver with a tight synchronisation in your spec file, these optimisations automatically have to be switched off.

`global` The `global` time stepping scheme makes each cell advance in time per iteration with a given time step Δt irrespective of its spatial resolution. If one cell in the domain finds out that Δt harms its CFL condition, all cells in the domain are rolled back to their last time step, Δt is reduced, and the time step is ran again. If a time step finds out that it has been too restrictive, the code carefully increases Δt for the next time step. So the scheme is a global scheme that does not anticipate any adaptive pattern (small cells usually are subject to more restrictive CFL conditions than coarse cells). However, the time step size is adaptively chosen. This implies that all ranks have to communicate once per time step, i.e. the scheme induces a tight synchronisation.

`globalfixed` The `globalfixed` time stepping scheme runs one time step of type `global` which determines a global Δt. From hereon, it uses this Δt for all subsequent time steps. If it turns out that the CFL condition is harmed through Δt later on throughout the simulation, no action is taken. In return, the scheme allows `ExaHyPE` to desynchronise all the ranks. This is among the fastest schemes available in the code. Furthermore, as the code knows the time step size and knows the time intervals between two plots a priori, it typically runs multiple time steps in one rush, i.e. you will not receive one terminal plot per time step, but most terminal outputs will summarise multiple time steps.

5. Finite Volume solvers with user-defined kernels

In this chapter, we work with ExaHyPE's Finite Volume code where we again specify the flux and eigenvalue functions ourselves. For the actual Finite Volume scheme, we rely on MUSCL and leave this implementation to ExaHyPE. While such a scheme is clearly inferior to ADER-DG (at least in the ExaHyPE philosophy), it offers an excellent limiter. However, for the time being, we realise it as stand-alone solver.

For a Finite Volume solver, we add a new `Finite-Volumes` solver section:

```
solver Finite-Volumes MyFVEulerSolver
  variables = 5
  parameters = 0
  patch-size = 3
  maximum-mesh-size = 0.1
  time-stepping = global
  kernel = generic::MUSCL
  language = C
end solver
```

Different to an ADER-DG solver, we have a `patch-size` here instead of an order flag. All other parameters are basically the same. We support most plotters for both ADER-DG and finite volumes.

If you rerun the toolkit, this addendum will yield a `MyFVEulerSolver` class with routines that have to be implemented by the user. We again stick to the Euler equations here as an examnple and realise them as

@todo Please add code

```
@todo Can someone please insert the C++ code snippets here?
```

Please note that the Finite Volume solvers are not realised via cells but via patches: ExaHyPE's cells act as meta data structure into whose we embed patches of `patch-size`×`patch-size` or `patch-size`×`patch-size`×`patch-size`, respectively. This fact is hidden from the user at the moment, but it gains importance once we make the finite volume solver act as limiter.

5.1. Localising a solver

The `maximum-mesh-size` attribute of a solver does allow the user to specify a maximum mesh resolution in the ADER-DG context. As we work with patches here, it specifies the size of a patch in the Finite Volume context. The actual size of a finite volume thus results from the maximum mesh size divided by `patch-size`.

@todo This is not working yet

Other than a constraint on the mesh layout, the `maximum-mesh-size` can also be used to localise a solver. If you hand in 0 as size, `ExaHyPE` will set up your simulation successfully, but it will not activate the solver. Its computational domain is $\Omega = \emptyset$.

@todo We have to provide the feature to insert the solver which is basically an expansion of the computational domain.

6. Limiters: coupling an ADER-DG scheme to Finite Volumes

In the ExaHyPE philosphy we provide a general strategy to couple various solvers. Our idea is that you can always have multiple solvers and that there are generic routines to couple these solvers patch-wisely. In the present chapter, we use exactly this feature to realise a limiter for ADER-DG. We start from a specification file that uses ADER-DG with order p and a Finite Volume scheme with a patch size of $2p + 1$. Both solvers shall be implemented completely, but the Finite Volume solver has a `maximum-mesh-size` of 0, i.e. it is inactive initially. Furthermore, we do not need any sophisticated boundary conditions for the Finite Volume solver. Just leave the corresponding routines blank.

Once we have more than one solver, the coupling of these solvers in the specification file is activated via a new block:

```
couple-solvers cellwise
  time = 0.0
  repeat = 0.0
end couple-solvers
```

In the present chapter, we couple two solvers if and only if they reside on the same spacetree cell (`cellwisee`), i.e. one Finite Volume patch coincides with an ADER-DG cell. We couple all solvers right from the start (`time`) and we couple after each time step (`repeat` equals zero); we could also couple after certain time spans.

Once we have added the `couple-solvers` tag, we may rerun the toolkit and we will obtain new routines that allow us to realise the actual interaction between the Finite Volume and the ADER-DG code.

7. Shared memory parallelisation

ExaHyPE currently supports shared memory parallelisation through Intel's Threading Building Blocks (TBB) and OpenMP. We recommend using the TBB variant that is typically one step ahead of the OpenMP support. To make an ExaHyPE project use multi- and manycore architectures, please add a `shared-memory` configuration block to your specification file:

```
shared-memory
  strategy = dummy
  cores = 4
  properties-file = sharedmemory.properties
end shared-memory
```

Rerun the ExaHyPE toolkit afterwards, and recompile your code. For the compilation, we assume that the environment variables `TBB_INC` and `TBB_SHLIB` are set[1]. If you use OpenMP, your compiler has to support OpenMP. The toolkit checks whether the environment variables are properly set and gives advise if this is not the case.

Whenever you configure your project with shared memory support, the **default** build variant is a shared memory build (with TBB or OpenMP, respectively). However, you always are able to rebuild without shared memory support or a different shared memory model just by manually redefining environment variables and by rerunning the makefile. There is no need to rerun the toolkit again. Also note that all arguments within the `shared-memory` environment are read at startup time. If you change the core count, there's no need to recompile. The same statements holds for the other parameters.

For users familiar with OpenMP, we emphasise that we do set the thread count manually within the code. OpenMP requires, per default, all users to control the thread count via environment variables. In ExaHyPE, the value of the environment variable and the value in the config file thus should match. You'll obtain a warning otherwise.

For most users, the dummy strategy should be sufficient. While the field `cores` is self-explaining, the properties file is neglected for the dummy strategy. If you use a more sophisticated strategy, it however makes a difference. More sophisticated strategies are subject of discussion next.

[1] `TBB_INC=-I/mypath/include` and `TBB_SHLIB="-L/mypath/lib64/intel64/gcc4.4 -ltbb"` are typical environment values.

Table 7.1.: Parameter choices for the multicore configuration. All values are read at program startup. There is no need to rerun the toolkit if you change any of them.

Parameter	Options	Description

strategy	`dummy`
	`autotuning`
	`sampling`
		The shared memory efficiency is very sensitive to a multitude of parameters that threshold the size of minimal problems deployed to one task, decide which code fragments shall run in parallel, and so forth. Three shared memory strategies for these parameters are provided at the moment. `dummy` uses some default values that have proven to be reasonably robust and yield acceptable speed. `sampling` tests different choices and plots information on well-suited variants to the terminal. `autotuning` tries to find the best choice on-the-fly.

cores	> 0	Number of cores that shall be used.
properties-file	filename	The `sampling` and the `autotuning` strategy can write their results into files and reuse these results in follow-up runs. If no (valid) file is provided, they both start from scratch and without any history. If the file name is empty or invalid, no output data is dumped. The entry is ignored if you use the `dummy` strategy.

Autotuning and tailoring

Shared memory performance can be very sensitive to machine-specific tuning parameters. We thus encourage high performance applications to use ExaHyPE's autotuning strategy. This strategy loads a set of tuning parameters from the properties file—if a file does exist—incrementally tries to improve these architecture-specific properties, and then dumps the values again into the property file.

This means that first runs might be really slow, but the runtime improves throughout the development. For production runs, you then have a well-suited parameter file at hands for the autotuning. The autotuning switches off automatically as soon as it has to believe that best-case parameters are found. However, it could be that these parameters are local runtime minima.

If you prefer to get the whole picture, you might want to run the sampling. However, the parameter space is huge, i.e. getting a valid picture might require several days. The output of the sampling is written into the specified properties file and then allows you to identify global best case parameters.

Once the autotuning strategy has found a (local) best-case configuration, it marks these configurations as valid and does not continue to search. As a result, also timing overheads are reduced. As all strategies write text configuration files, it is possible to configure the autotuning manually with proper parameter choices and to switch off the actual autotuning at the same time. You might for example search for optimal choices with the sampling and then feed these values into the autotuning.

It is obvious that proper property files depend on the machine you are using. They reflect your computer's properties. Please note that very good property files also depend on the

- choice of cores to be used,
- MPI usage and balancing,
- application type, and even
- input data/simulation scenario.

It thus might make sense to work with various property files for your experiments.

Hybrid parallelisation

This section discusses the interplay of the shared memory parallelisation with MPI. It is written for HPC experts. For most users, no changes have to be made to the shared memory parallelisation when they turn on MPI as well.

This subsection yet has to be written.

8. Distributed memory parallelisation

ExaHyPE's distributed memory parallelisation is done with plain MPI. We rely on the 1.3 standard with the pure C bindings. To make an ExaHyPE project use mpi, please add a `distributed-memory` configuration block to your specification file:

```
distributed-memory
  identifier = static_load_balancing
  configure = {greedy,FCFS}
  buffer-size = 64
  timeout = 120
end distributed-memory
```

Rerun the ExaHyPE toolkit afterwards, and recompile your code. For the compilation, we assume that a proper MPI compiler is available.

Whenever you configure your project with distributed memory support, the **default** build variant is a distributed memory build. However, you always are able to rebuild without distributed memory support by manually redefining environment variables and by rerunning the makefile. There is no need to rerun the toolkit again. Also note that all arguments within the `distributed-memory` environment are read at startup time.

The parameter `timeout` specifies how long a node shall wait (in seconds) before it triggers a time out if no message has arrived. If half of the time has elapsed, it furthermore writes a warning. This is a very useful variable to identifiy deadlocks as well as communication inefficiencies. Set the value to zero if you don't want to use the time out detection.

The parameter `buffer-size` specifies how many messages Peano shall internally bundle into one MPI message. This operation allows you to tailor your application behaviour w.r.t. latency and bandwidth requirements. The fewer messages you bundle, i.e. the small this value, the faster messages are sent out and other ranks might not have to wait for them. The more messages you bundle the lower the total communication overhead. Please note that some Infiniband implementations tend to deadlock, if this value is too small as they are swamped with lots of tiny messages. We found 64 to be a reasonable first try.

The required/permitted values in the `configure` field depend on the `identifier` chosen and thus are enlisted below in the respective tables.

MPI variant `static_load_balancing`

Argument	Values	Semantics
`greedy`	none	Uses a greedy split strategy to decompose the underlying spacetree. Subpartitions do not join again. The individual ranks do not synchronise with each other. What might be a hot spot on a rank A might not matter overall as rank B has so much more work to do. So this scheme is fast and stupid but might yield non-optimal partitions right from the start.
`hotspot`	none	The ranks do synchronise with each other. This variant is the most conservative one and introduces quite some grid construction overhead as the load balancing has to plug into the grid construction. It should however yield better partitions than the greedy partitioning scheme.
`FCFS`	none	If an MPI rank want to split its domain, it sends a request for an additional MPI rank to rank 0. All requests on rank 0 are answered in FCFS fashion which minimises the answer latency but might yield unfair decompositions: ranks with a very low latency towards rank 0 are more likely to be served than others. You may not combine `FCFS` and `fair`.
`fair`	none	All requests sent to rank 0 are collected for a couple of ms and then answered such that those ranks with the lowest number of workers so far get new workers first. The answering latency is slightly higher than for `FCFS` but the distributions tend to be fairer. This variant expects the user to specify `ranks_per_node`, too. You may not combine `FCFS` and `fair`.
`ranks_per_node`	≥ 1	Ignored if `fair` is not specified. It instructs the load balancing how many ranks are placed on one node.

On `ranks_per_node`

This flag is used by the fair/hotspot load balancing strategies. It instructs the load balancing how many ranks are placed on one node. The balancing uses this information in two ways:

1. Out of the first ranks_per_node, only the very first rank is used for the global master and the global load balancing code. The rationale is as follow: The load balancing is a single point of contact. Every node that wants to fork, has to ask at the global master for free ranks. So it is critical that the global master has a low latency. I thus sacrifice one node and run only the global master there to allow it to have the whole MPI IO exclusively. We accept that ranks_per_node-1 ranks are wasted.

2. If n ranks request for one worker each for a level ℓ, the load balancing tries to make each node, i.e. every ranks_per_nodeth rank, serve this request. Notably, the ranks responsible for the coarests tree levels are assigned to the ranks ranks_per_node, $2\cdot$ ranks_per_node, $3\cdot$ ranks_per_node, and so forth. The idea is that jobs are homogeneously distributed among the nodes and no node runs risk to run out of memory. Furthermore, if the grid is reasonably regular, also the load should be reasonably distributed among the nodes.

If you plot scalability results, you have to rescale your x-axis corresponding to ranks_per_node. If you want to use all cores on the first node, please make MPI deploy ranks_per_node + ranks_per_node-1 ranks to this node.

Hybrid parallelisation

Please see Section 7 for details how MPI and the shared memory parallelisation interplay.

9. Optimisation

9.1. High-level tuning

ExaHyPE realises few high level optimisations that you can switch on and off at code startup through the specification file. To gain access to these optimisations, add a paragraph

```
optimisation
  ...
end optimisation
```

at the end of your specification file. It requires a sequence of parameters in a fixed order as they are detailed below.

Step fusion. Each time step of an ExaHyPE solver consists of three phases: computation of local ADER-DG predictor (and projection of the prediction onto the cell faces), solve of the Riemann problems at the cell faces, and update of the predicted solution. We may speed up the code if we fuse these four steps into one grid traversal. To do so, we set `fuse-algorithmic-steps`. Permitted values are `on` and `off`.

The fusion is using a moving average kind time step size estimate. If we detect a-posteriori that this estimate has violated the CFL condition, we have to rerun the predictor phase of the ADER-DG scheme with a time step size that does not violate the CFL condition. In our implementation, such a CFL-stable time step size is available after the ADER-DG time step has been performed.

We offer to rerun the predictor phase of the ADER-DG scheme with this CFL-stable time step size weighted by a factor `fuse-algorithmic-steps-factor`. This problem-dependent factor must be chosen greater zero and smaller to/equal to one. A value close to one might lead to more predictor reruns during a simulation but to less additional numerical diffusion.

```
optimisation
  fuse-algorithmic-steps = off
  fuse-algorithmic-steps-factor = 0.99
  ...
end optimisation
```

Batching of time steps. If you use fixed, adaptive or anarchic time stepping, Peano can guess from an evaluation of the CFL condition how many grid sweeps (global time steps) are required to advanced all patches such that the next plotter becomes

active or we meet the simulation's termination time. Each grid sweep/global time step might update only a few patches and their permitted time step size again might change throughout the simulation run. So we can only predict. For MPI's speed is it advantageous if multiple time steps are triggered in one rush—this way, ranks that have already finished its traversal know whether they can immediately continue with the subsequent time step. To allow so, you have to set `fuse-algorithmic-steps-factor` to a value from $(0, 1($. Setting it to zero switches off this feature. The semantics is as follows: The code computes how many time steps it expects to be required to reach the next plotter or final simulation time, respectively. This value is scaled with `fuse-algorithmic-steps-factor` and the resulting number of time steps then are ran. A factor of around `0.5` has to be proven to be good starting point.

```
optimisation
  ...
  timestep-batch-factor = 0.5
end optimisation
```

Skip reduction of global data. If you allow the code to run multiple iterations in one batch, it makes sense to tell ExaHyPE that there is no need to restrict any data (such as minimal time steps) while a batch is processed. It is usually sufficient (unless you need this data explicitly) to restrict global data after the last grid update sweep has terminated. The flag `skip-reduction-in-batched-time-steps` switches the reduction on or off. Permitted values are `on` and `off`. The value is always requried even though you might not allow the code to batch time steps. In this case, it is ignored.

```
optimisation
  ...
  timestep-batch-factor = 0.5
  skip-reduction-in-batched-time-steps = on
end optimisation
```

9.2. Optimised Kernels

ExaHyPEoffers optimised compute kernels. Given a specification file, the toolkit triggers the code generator to output optimised compute kernels for this specific setup.

Prerequisites The code generator exhibits two dependencies. First, the code generator requires Python 3. Second, it relies on Intel's libxsmm generator driver. You therefore have to clone the libxsmm repository https://github.com/hfp/libxsmm and build your local generator driver

```
make generator
```

to obtain the executable in `path/to/libxsmm/bin/libxsmm_gemm_generator`.

Microarchitecture The optimised kernels explicitly use the instruction set available on the target processor. You therefore have to define its microarchitecture in the specification file. The specification file terms the processor architecture just architecture. Here are the supported options for this flag.

Architecture	Meaning
noarch	unspecified
wsm	Westmere
snb	Sandy Bridge
hsw	Haswell
knc	Knights Corner (Xeon Phi)
knl	Knights Landing (Xeon Phi)

You can identify the microarchitecture of your processor through amplxe-gui, an analysis tool part of Intel VTune Amplifier. Alternatively, you can obtain the 'model name' via

```
cat /proc/cpuinfo
```

and search for it on http://ark.intel.com.

Set the architecture flag to 'noarch' if the microarchitecture of your processor is not supported. You will nevertheless obtain semi-optimised kernels. The default makefile utilises the flags `march=native` (gcc) respectively `xHost` (icc). The compiler queries what hardware features are available and optimises the kernels to some degree.

10. Postprocessing and plotting

10.1. DG polynomials in **ExaHyPE**'s output format

As ExaHyPE involves the Discontinous Galerkin method, it aims to output not only cell-averaged data for each cell, but instead the data on all the polynomial supporting points. For instance, for a second order DG approximation in 1D, your output contains three data points per cell and thus holds almost three times more data than with a cell-averaging output method. The chosen output format allows you to exactly reconstruct the polynomials as they are available during an ExaHyPE run.

In shock-free simulations, this output format introduces a certain redundancy, as values at the cell boundaries are given out ("plotted") two times. However, these points provides additional information as soon as discontinuities appear, cf. figure 10.1.

While a visualization software might use the data to reconstruct the polynomial, we don't expect this to happen in every-day visualisations. This has two reasons: First, it is an expensive computation. Second, it needs information about the grid structure which is present in the VTK files that is not so easily extractable. We provide a VTK hands on in the next pages to show how to access these information.

10.2. Data Writers (Plotters)

The text below lists all available plotter types. They are distinguished by their identifier in the specification file. While the number of plotters and their type have to be fixed at compile time—if you want to introduce a new plotter, you have to rerun the ExaHyPE toolkit and rebuild your code—all other parameters are read at runtime. A plotter always has to specify how many variables are written through the unknowns statement. By default, these are the first unknowns unknowns from your solver. If unknowns is greater than the unknowns of the solver, you also obtain the material parameters if there are material parameters.

10.2.1. Plotter types

vtk::Cartesian::vertices::ascii Dumps the whole simulation domain into one VTK file. If you use multiple MPI ranks, each ranks writes into a file of its own. The plotter uses ASCII, i.e. text format, so these files can be large. output has to be

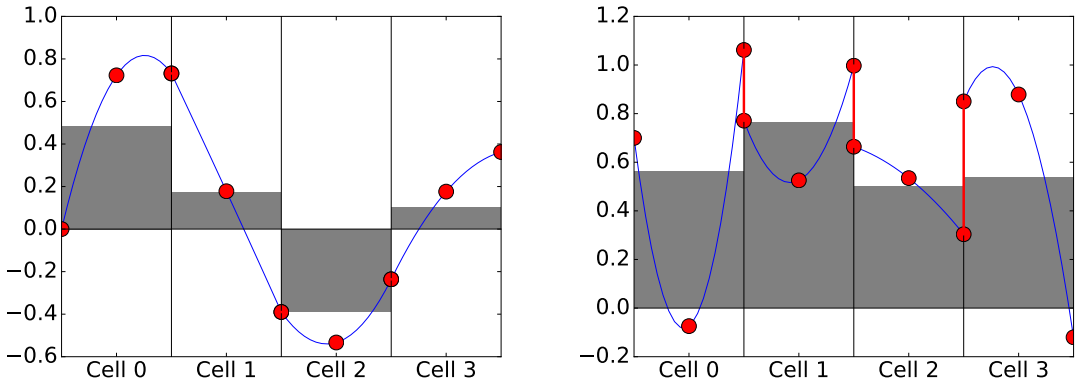


Figure 10.1.: A sketch of ExaHyPE's output point structure in a 1D example with four cells and a polynomial degree of 2. In each example, the polynomial degree is plotted. The shaded regions indicate the cell average. In the regular case of a continous solution, there are two data points at the cell boundaries with exactly the same value. In the vicinity of shocks (discontinuities), as sketched in an exaggerated manner on the right, the different values for two points at the same grid value becomes obvious. The output is thus a double-valued function.

a valid file name where a rank and time step identifier can be appended. The plotter always adds a `.vtk` postfix. Through the `select` statement, you can use spatial filters. if you add values such as `select = {left:0.4,right:0.6,bottom:0.2, top:0.8,front:0.2,back:0.5}`, you get only data overlapping into this box. As VTK does not natively support higher order polynomials, the solution is projected onto a grid, and the solution values are sampled on this grid. In this case, we use a Cartesian grid and the values are sampled at the grid vertices.

`vtk::Cartesian::vertices::binary` Same as `vtk::Cartesian::vertices ::ascii`, but the files internally hold binary data rather than ASCII data. All written files should thus be slightly smaller.

`vtk::Cartesian::cells::ascii` See `vtk::Cartesian::vertices::ascii`. Solution values are sampled as cell values instead of vertex values.

`vtk::Cartesian::cells::binary` Binary variant of plotter with identifier `vtk::Cartesian::cells::ascii`.

`vtk::Legendre::vertices::ascii` See counterpart with `Cartesian` instead of `Legendre`. Mesh values are not sampled at Cartesian mesh points but at Legendre points.

`vtk::Legendre::vertices::binary` See counterpart with `Cartesian` instead of `Legendre`. Mesh values are not sampled at Cartesian mesh points but at Legendre

points.

vtk::Legendre::cells::ascii See counterpart with Cartesian instead of Legendre. Mesh values are not sampled at Cartesian mesh points but at Legendre points.

vtk::Legendre::cells::binary See counterpart with Cartesian instead of Legendre. Mesh values are not sampled at Cartesian mesh points but at Legendre points.

probe::ascii This option probes the solution over time, i.e. you end up with files specified by the field output that hold a series of samples in one particular point of the solution. The code adds a .probe postfix to the file name. This option should be used to plot seismograms, e.g. For this data format is ascii, the file holds one floating point value per line per snapshot in text format. To make the plotter know where you want to probe, please add a line alike

```
select = {x:0.3,y:0.3,z:0.3}
```

to your code. If you run your code with MPI, the probe output files get a postfix -rank-x that indicates which rank has written the probe. Usually, only one rank writes a probe at a time. However, you might have placed a probe directly at the boundary of two ranks. In this case, both ranks do write data. If dynamic load balancing is activated, the responsibility for a probe can change over time. If this happens, you get multiple output files (one per rank) and you have to merge them manually. Please note that output files are created lazily on-demand, i.e. as long as no probe data is available, no output file is written. The probe file contains data from all variables specified in the plotter. It furthermore gives you the time when the snapshot had started and the actual simulation time at the snapshot location when data had been written. As ExaHyPE typically runs adaptive or local time stepping, a snapshot is triggered as soon as all patches in a simulation domain have reached the next snapshot time. At this point, some patches might already have advanced in time. This is why we record both snapshot trigger time and real time at the data point.

10.2.2. Plotting only a few parameters and on-the-fly postprocessing

If you want to plot only a few parameters of your overall simulation, you have to invest a little bit of extra work. First, adopt your plotter statements such that the unknowns statement tells the toolkit exactly how many variables you want to write. Typically, you plot fewer variables than the unknowns of your actual PDE. However, there might be situations where you determine additional postprocessing data besides your actual data and you then might plot more quantities than the original unknowns.

Once you have tailored the unknowns quantity, you rerun the toolkit and you obtain classes alike MySolver_Plotter0. These classes materialise the individual

solvers as written down in your specification file and they are enumerated starting from 0. Open your plotter's implementation file and adopt the function there that maps quantities from the PDE onto your output quantities. By default, this mapping is the identity (that's what the toolkit generates). However, you might prefer to do some conversations such as a conversion of conservative to primitive variables. Or you might want to select the few variables from the PDE solver that you are actually interested in.

Below is an example:

```
  solver ADER-DG MyEulerSolver
    variables = 5
    parameters = 0
    order = 3
    maximum-mesh-size = 0.1
    time-stepping = global
    kernel = generic::fluxes::nonlinear
    language = C

    plot vtk::Cartesian::ascii
      variables = 2
      time = 0.0
      repeat = 0.05
      output = ./solution
      select = {}
    end plot

  ...
```

Originally, we did plot all five variales. Lets assume we are only interested in $Q0$ and $Q4$. We therefore set `variables` in the plotter to 2 and modify the corresponding generated plotter:

```
void MyEulerSolver_Plotter0::mapQuantities(
    const tarch::la::Vector<DIMENSIONS, double>& offsetOfPatch,
    const tarch::la::Vector<DIMENSIONS, double>& sizeOfPatch,
    const tarch::la::Vector<DIMENSIONS, double>& x,
    double* Q,
    double* outputQuantities,
    double timeStamp
) {
  outputQuantities[0] = Q[0];
  outputQuantities[1] = Q[4];
}
```

10.2.3. Plotting/computing global data such as integrals

The plotter sections in the specification file allow you to write `unknowns=0`. For this exercise use for example In this case, the toolkit creates a plotter and hands

over to this plotter all discretisation points. However, it neglects all return data, i.e. nothing is plotted.

To compute the global integral of a quantity, you might want to use the `vtk::Cartesian::ascii` plotter but set the unknowns to zero. Whenever your plotter becomes active, `ExaHyPE` calls your `startPlotting` operation. Add a local attribute m to your class and set it to zero in the start function. In your conversation routines, you can now accumulate the L2 integral in m, while you use `finishPlotting` write the result to the terminal or into a file of your choice, e.g. If you alter the start and finish routine, please continue to call the parent operation, too.

We illustrate the realisation at hands of a simple computation of the global L_2 norm over all quantities of a simulation. For this, we first add a new variable to the class:

```
class MyEulerSolver_Plotter0: public ...
  private:
    // We add a new attribute to the plotter.
    double _globalL2Accumulated;
  public:
    MyEulerSolver_Plotter0();
    virtual ~MyEulerSolver_Plotter0();
...
```

Next, we set this quantity to zero in the plotter initialisation, we accumulate it in the mapping operation and we write the result to the terminal when the plotting finishes.

```
void MyEulerSolver_Plotter0::startPlotting(double time) {
  // Reset global measurements
  _globalL2Accumulated = 0.0;
}


void MyEulerSolver_Plotter0::finishPlotting() {
  _globalL2Accumulated = std::sqrt(_globalL2Accumulated);
  std::cout << "my global value = " << _globalL2Accumulated << std::endl;
}


void MyEulerSolver_Plotter0::mapQuantities(
    const tarch::la::Vector<DIMENSIONS, double>& offsetOfPatch,
    const tarch::la::Vector<DIMENSIONS, double>& sizeOfPatch,
    const tarch::la::Vector<DIMENSIONS, double>& x,
    double* Q,
    double* outputQuantities,
    double timeStamp
) {
  // There are no output quantities
  assertion( outputQuantities==nullptr );
```

```
  // Now we do the global computation on Q but we do not write anything
  // into outputQuantities
  const NumberOfLagrangePointsPerAxis = 4; // Please adopt w.r.t. order
  const NumberOfUnknownsPerGridPoint = 5; // Please adopt

  double scaling = tarch::la::volume(
    sizeOfPatch* (1.0/NumberOfLagrangePointsPerAxis)
  );
  for (int iQ=0; iQ<NumberOfUnknownsPerGridPoint; iQ++) {
    _globalL2Accumulated += Q[iQ] *Q[iQ] *scaling;
  }
}
```

As we have set the number of plotted unknowns to zero, no output files are written at all. However, all routines of `MyEulerSolver_Plotter0` are invoked, i.e. we can compute global quantities, e.g. Some remarks on the implementation:

- It would be nicer to use the plotter routines of Peano when we write data to the terminal. This way, we can filter outputs via a filter file, i.e. at startup, and Peano takes care that data from different ranks is piped into different log file and does not mess up the terminal through concurrent writes.
- Most L_2 computations do scale the accumulated quantity with $\frac{1}{N}$ where N is the number of data points. Such an approach however fails for adaptive grids. If we scale each point with the mesh size, we automatically get a discrete equivalent to the L_2 norm that works for any adaptivity pattern. The `volume` function computes h^d for a vectorial h. See the documentation in `tarch::la`.
- The abovementioned version works if you use a Cartesian plotter where ExaHyPE's solution already is projected onto regular patches within each cell (subsampling). There are other plotters that allow you to evaluated the unknowns directly in the Lagrange points. The scaling of the weights then however has to be chosen differently.
- Plotting is expensive as ExaHyPE switches off multithreading for plotting always. It thus makes sense not to invoke a plotter too often—even if you can handle the produced data of a simulation.

10.3. Visualisation and postprocessing

There are a number of Open-Source tools available to interactively watch, inspect and render VTK files. Popular choices are *ParaView* and *Visit*. These programs are available for many operating systems and also typically present on vizualization nodes in scientific clusters.

However, in many applications it is neccessary to programmatically access the data. As VTK defines not only a file format but also a programming standard

(class interface), there is a straight forward way to process the data generated by ExaHyPE. It is that the VTK files produced by ExaHyPEcontain an *Unstructured Grid* which is basically a set of points with scalar fields on each point. Each conserved quantity (numberOfVariables and numberOfParameters in ExaHyPE) corresponds to one scalar field in the output. It is thus possible to display the output as a table with a huge number of row entries, with columns

```
t x y z Q0 Q1 Q2 Q3 Q4 ...
```

The next section will contain a primer how to access and inspect the VTK files.

10.4. Postprocessing hands on

This section shall contain a small C++ or Python hands on how to read VTK files and plot a 1D slice with some trivial postprocessing, as applying a constant shift and applying a time rescaling, using Matplotlib or gnuplot. This can be done in less than 20 lines of code.

10.5. Logging

ExaHyPE creates a large number of log files; notably if you build in debug or assert mode. Which data actually is written to the terminal can be controlled via a file exahype.log-filter. This file has to be held in your executable's directory. It specifies exactly which class of ExaHyPE is allowed to plot or not. The specification works hierarchically, i.e. we start from the most specific class identifier and then work backwards to find an appropriate policy.

```
# 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

info peano::utils::UserInterface -1 white
info exahype -1 white
```

If no file exahype.log-filter is found in the working directory, a default configuration is chosen by the code. Please note that a sound performance analysis requires several log statements to be switched on. For a detailed description, please study Peano's handbook or the produced report pages as well as the report scripts.

A. The **ExaHyPE** toolbox

The `ExaHyPE` toolbox is a small Java toolkit. It acts as sole interface for users, i.e. non-developers. We decided to realise it in Java for several reasons:

1. We can realise lots of syntax checks without blowing up the C++ code.
2. We have to generate code fragments in `ExaHyPE`. The order of the methods chosen for example has an impact on kernel calls and mapping variants. This is easy in Java.
3. We prefer not to have several interfaces. With a Java front-end, we can directly generate the configuration rather than parsing configurations again.

A.1. Rebuilding the toolbox

To build the toolbox, we rely on ant.

```
ant target
```

where `target` is from

- `createParser`. Recreates the Java parser. `ExaHyPE`'s Java parser relies on SableCC. The parser has to be regenerated if and only if you change the grammar or you have to rebuild the tool without a previous build.
- `compile`. Translate the toolkit.
- `dist`. Pack the translated files into one jar file.
- `clean`. Clean up the compiled and temporary files.

A.2. Troubleshooting

Our toolkit is based upon ant as build environment. It seems that newer Linux/-Java versions are often incompatible with precompiled ant versions (OpenSUSE, e.g., has problems). In this case, you might have to switch to superuser and to select an older Java version manually:

```
update-alternatives --config java
```

B. Frequently asked developer questions & problems

- **MPI plus TBB deadlocks or is very slow**
  On several systems, we have encountered problems when we tried to use multithreaded MPI. As a result, we disable multithreaded MPI usage by default in the makefile (switch `-DnoMultipleThreadsMayTriggerMPICalls`). You might want to try to enable multithreaded MPI again with `-DMultipleThreadsMayTriggerMPICalls`, but we cannot guarantee that it works properly. It is trial & error depending on your compiler/MPI/cluster choice.

- **How can I use ExaHyPE's logging**
  ExaHyPE uses Peano's technical architecture (`tarch`) and the logging therein. For a detailed documentation of this technical architecture (besides logging, there's also a small linear algebra library in there that allows you to write down C++ code in a MATLAB-like notation) please consult the Peano cookbook. For the simple logging, please add a static field to year header

  ```
  static tarch::logging::Log _log;
  ```

  initialise it and then use the `logInfo` macro:

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

  [...]

  void MyParticularClass::myRoutine() {
    double myA = 1.2345;

    logInfo( "myRoutine()", "variable myA has the value " << myA );

    [...]
  }
  ```

  From hereon, you can also filter your own log messages with the filter file.

- **To be continued ...**

C. Frequently asked user & problems

- **How do I modify the computational domain?**
  Open the specification file and alter the `width`. The entry specifies the dimensions of the computational domain. Please note that ExaHyPE only supports square/cube domains as it works on cube/square cells only. You might have to extend your problem's domain accordingly. `width` is read at startup time, i.e. there's no need to rerun any script or recompile once you have changed the entry. Just relaunch your simulation.

- **How do I alter the time step size?**
  ExaHyPE implements three time stepping schemes (global time stepping, local time stepping, anarchic time stepping). You can configure them within your generated solver class in your application directory. Once you have chosen the time stepping scheme, the code autonomously determines the biggest stable time step size that is permitted. Nothing has to be done.

  Only global currently. You can't configure yet.

- **How do I alter the time step size (advanced)?**
  You may however choose the time step size yourself if you do not rely on the generated kernels but write a whole solver yourself (see solver variant `user::defined` in Section 4) from scratch.

- **How do I change the spatial resolution statically?** The spatial resolution (mesh width) is tied to a particular solver/problem. ExaHyPE thus wants to tell the kernel/grid per solver which resolution you require. For this, have a look into `getMinimumTreeDepth()`. Here, you tell the code how fine the start grid has to be.

- **How do I change the resolution dynamically?** The dynamic AMR is realised along the same lines as the static refinement. You have to implement `refinementCriterion` in your solver which tells ExaHyPE where you want to refine. How this is realised—notably how all the ranks and cores synchronise—is hidden away from the user code and decided in the kernel.

  We have to make this routine speak in terms of mesh sizes, and we have to make it domain dependent once we have AMR.

- **To be continued ...**

D. Solver Kernels

The kernel variants below are offered for the `ADER-DG` solver type. The last column indicates whether we offer FORTRAN support for these.

Kernel	Description	F
user::defined	Only function signatures are generated. All responsibility which and how kernels are realised is up to the user. We basically only hand over large double arrays. This is the lowest level of abstraction.	yes
generic::fluxes::nonlinear	Default ADER-DG routines are used, but they make no assumptions on the fluxes. The toolkit will ask you to provide fluxes, eigenvalues, and so forth which are then used within nested for loops generically. This is the second level of abstraction and well-suited for fast prototyping.	yes
generic::fluxes::linear	Similar to generic::fluxes::nonlinear. Given the explicit notion of a linear flux, the underlying generic source code however is simpler. This variant is faster.	yes
optimised::fluxes::nonlinear	Equals generic::fluxes::nonlinear from a user's point of view. However, the underlying ADER-DG routines/loops all are optimised through libxsmm.	no
optimised::fluxes::linear	Optimised version of optimised::fluxes::nonlinear that removes any non-linear iteration.	no
kernel::euler2d	These are optimised kernels that solves the Euler equations in two or three dimensions. They mirror the example from Chapter 4. As we fix the PDE, no fluxes, eigenvalues, ... have to be written by the user and the resulting kernel is very aggressively optimised.	t.b.a.

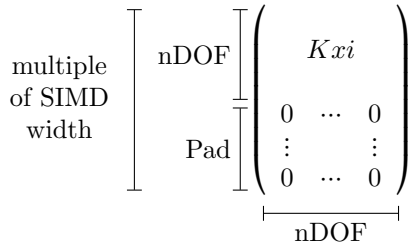
The following kernel variants are offered for the `Finite-Volume` solver type:

Kernel	Effect
user::defined	Only function signatures are generated. All responsibility which and how kernels are realised is up to the user. We basically only hand over large double arrays. This is the lowest level of abstraction.
t.b.d.	

E. Datastructures in the Optimised Kernels

This chapter provides an up-to-date documentation of the memory layout employed in the optimised solver kernels.

System Matrices The optimised kernels use padded structures and thereby differ from the generic code base. Padding, of course, depends on the target microarchitecture. It guarantees that the height of the matrices is always a multiple of the SIMD width.



All matrices are stored in column major order. Table E.1 lists the memory layout. The corresponding CodeGen files are `DGMatrixGenerator.py` and `WeightsGenerator.py`. The matrix `dudx` is only generated in the case of a linear PDE.

Matrix	Memory Layout	Size
Kxi	[[Kxi];0]	(nDOF+Pad) × nDOF
iK1	analogous to Kxi	
dudx	analogous to Kxi	
FRCoeff	[FRCoeff]	1 × nDOF
FLCoeff	analogous to FRCoeff	
F0	[F0;0]	1 × (nDOF+Pad)
equidistantGridProjector1d	analogous to generic version	
fineGridProjector1d	analogous to generic version	

Table E.1.: Structure of the system matrices in the optimised kernels.

Quadrature Weights The solver kernels exhibit three distinct patterns of weights. Either occurrence is mapped onto a separate data structure.

- `wGPN`. Quadrature weights.
- `aux = (/ 1.0, wGPN(j), wGPN(k) /)`. Combination of two weights, stored in weights2.
- `aux = (/ wGPN(i), wGPN(j), wGPN(k) /)`. Combination of three weights, stored in weights3.

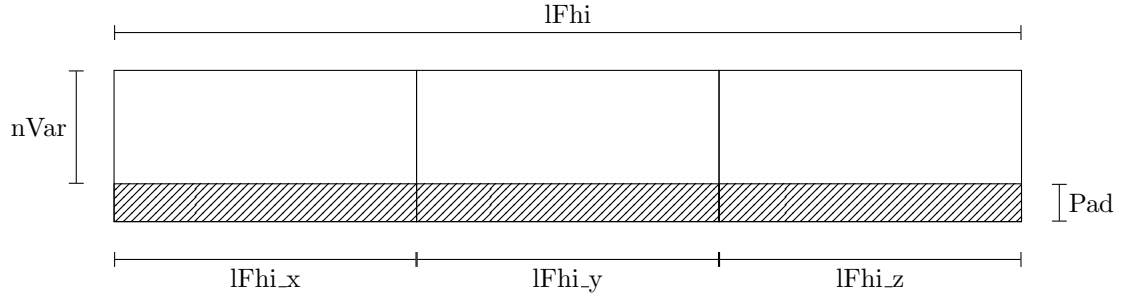
Table E.2 illustrates the memory layout for a 2D and 3D setup, respectively. In any case, the weights vectors are interpreted as linear, padded arrays. The pad width depends on the microarchitecture and ensures that the total length of the vectors is a multiple of the SIMD width. In the 2D case, the vectors `weights1` and `weights2` coincide because their computational pattern coincides.

Name	Memory Layout	Meaning
	2D	
weights1	$[w_i, 0]$	quadrature weights + Pad
weights2	$[w_i, 0]$	quadrature weights + Pad
weights3	$[w_i w_j, 0]$	outer product of quadrature weights + Pad
	3D	
weights1	$[w_i, 0]$	quadrature weights + Pad
weights2	$[w_i w_j, 0]$	outer product of quadrature weights + Pad
weights3	$[w_i w_j w_k, 0]$	all combinations of quadrature weights + Pad

Table E.2.: Variables holding combinations of quadrature weights employed in the optimised kernels.

Parameters and Local Variables Table E.3 lists the structure of the input/output parameters of the solver kernels and local variables used within the solver kernels. The parameters nDOFx, nDOFy and nDOFz denote the number of degrees of freedom in x-, y-, and z-direction, respectively. Analogously, nDOFt gives the number of degrees of freedom in time. All variables are linearised, i.e. one-dimensional arrays.

The parameter lFhi is decomposed into three tensors lFhi_x, lFhi_y, lFhi_z, all sized (nVar+Pad)·nDOF^3



Name	Ordering	Size
rhs, rhs0	(nVar,nDOFx,nDOFy,nDOFz,nDOFt)	nVar · $nDOF^4$
lFbnd	(nDOFx, nDOFy, nVar)	((nDOFx · nDOFy)+Pad)· nVar
lQbnd	(nDOFx, nDOFy, nVar)	((nDOFx · nDOFy)+Pad)· nVar
lqh (nonlinear)	(nVar, nDOFt, nDOFx, nDOFy, nDOFz)	(nVar+Pad)·$nDOF^4$
lqh (linear)	(nVar, nDOFx, nDOFy, nDOFz,nDOFt+1)	(nVar+Pad)·$nDOF^3$·(nDOF+1)
lFh	(nVar, nDOFx, nDOFy, nDOFz,nDOFt,d)	(nVar+Pad)·$nDOF^4$·d
lFhi	irregular	(nVar+Pad)·$nDOF^3$·d
luh, lduh	(nVar, nDOFx, nDOFy, nDOFz)	nVar · $nDOF^3$
	temporary variables	
QavL, QavR	(nVar)	(nVar+Pad)
lambdaL, lambdaR	(nVar)	(nVar+Pad)
	auxiliary variables	
tmp_bnd	(nDOFx, nDOFy, nVar)	((nDOFx · nDOFy)+Pad)· (nVar+Pad)
s_m (“scaled *m*atrix”)	analogous to Kxi	
s_v (“scaled *v*ector”)	analogous to F0	

Table E.3.: Data layout (Fortran notation) used in the optimised kernels.

Their internal ordering of the degrees of freedom depends on the direction.

- `lFhi_x`. (nVar, nDOFx, nDOFy, nDOFz)
- `lFhi_y`. (nVar, nDOFy, nDOFx, nDOFz)
- `lFhi_z`. (nVar, nDOFz, nDOFx, nDOFy)

Note that the structure of luh respectively lduh is chosen for compatibility with the generic kernels. TODO: the data layout (nDOFx, nDOFy, nDOFz, nVar) looks favourable due to the structure present in the extrapolation and in the volume integral.