
# ExaHyPE Guidebook

http://www.exahype.eu

Version 0.9 (pre-Release)

February 13, 2017

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 the term Engine circumscribes our vision to offer a software suite comparable to a game engine. Instead of offering all the neccessary functions to render 3D scenes, manage the gameplay, add sounds or make units walk over your game board (as a game engine would do), our PDE Engine offers ingredients to solve hyperbolic systems of Partial Differential Equations written in first order form either via explicit ADER-DG or Finite Volume schemes. Users have to stick to these methodological/strategic ingredients, but can freely tailor and adopt them, write new plug ins or assemble them in various ways. This document details

- how some "simple" demonstrators are ran,
- how a new application can be written from scratch,
- how ExaHyPE applications are scaled up,
- and it also details and documents how and why the engine has been designed in certain ways.

To read more about the project, we refer to the official website www.exahype.eu as well as the objectives at the EU's cordis page. ExaHyPE is completely open source.

## Who should read this document

This guidebook is written in a hands-on style. It addresses users of ExaHyPE and developers building new applications within ExaHyPE. Yet, it is not meant to replace the code documentation. Using the guidebook requires a decent background in your application domain modelled via a hyperbolic equation system. It does not require deep programming knowledge. Some C knowledge is advantageous.

Most examples 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. We want to encourage users to help us to improve ExaHyPE through feedback via ExaHyPE's forum/issue tracking system. Questions beyond the application domain—notably the used gridding paradigms—can be sent directly to Tobias Weinzierl, e.g.

February 13, 2017
The ExaHyPE team

# Contents

# Part I.

# **ExaHyPE** installation and demonstrator applications

# 1. Setup and Installation

## 1.1. Dependencies and prerequisites

ExaHyPE comes along with the following dependencies:

- `ExaHyPE` source code is C++ code. For serial codes, plain C++ does the job. All examples from this guidebook run and have been tested with GCC 4.2 and Intel 12 or later. Earlier compiler versions might work. There are no further dependencies or libraries required.
- You may prefer to write parts of your `ExaHyPE` code in Fortran. In this case, you need a Fortran compiler. `ExaHyPE` itself does not require Fortran.
- If you want `ExaHyPE` to exploit your multi- or manycore computer, you have to have Intel's TBB or OpenMP. Both come are open-source and work with GCC and Intel compilers.
- If you want to run `ExaHyPE` on a distributed memory cluster, you have to have MPI installed. `ExaHyPE` uses only very basic MPI routines. In our tests, basically any MPI version works.
- ExaHyPE's development environment relies on Java (Java Runtime Environment JRE). We provide some binaries (a standalone JAR file) typically built against Java 1.6 or newer. You can always build your own version of all development tools as long as you have a Java compiler (Java Development Kit JDK). Again, `ExaHyPE` uses only very basic Java, so most versions should do.
- ExaHyPE's default build environment uses GNU Make.
- If you want to use ExaHyPE's optimised compute kernels, you have to install Intel's `libxsmm`[1]. Furthermore, you have to have Python 3 available on your development system.

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. `ExaHyPE` also can not provide any support for these platforms. The code itself is minimalistic, i.e. in `ExaHyPE`'s basic form no further libraries are required.

[1] Libxsmm is open source: https://github.com/hfp/libxsmm

## 1.2. Obtaining ExaHyPE

ExaHyPE is available as source code only. We discuss several variants how to obtain the code below[2]. ExaHyPE is built on top of the AMR framework Peano. If you download a complete snapshot of ExaHyPE (the `tar.gz` files), a snapshot of Peano is included. If you clone the repository (Variant 2), you have to add Peano manually.

### Variant 1: Download an **ExaHyPE** release

Open a browser and go to https://github.com/exahype. Switch to the Code tab where you should find a subtag `release`. Find the release of your choice and download it. A minimalistic ExaHyPE version requires `ExaHyPE.tar.gz` and `ExaHyPE.jar`. Unzip them.

The files offered under the label Source code are snapshots of the repository. Different to the tar.gz from above, they do not comprise the Peano code which is the meshing base of ExaHyPE, i.e. if you work with these snapshots, you have to install Peano separately as detailed below.

If the jar file provided is incompatible with your Java version, you have to download `ExaHyPE-toolkit.tar.gz`. It comprises a makefile to rebuild the toolkit.

### Variant 2: Clone the **ExaHyPE** release repository

Alternatively, you might also want to clone the ExaHyPE release repository which has the advantage that you can quickly get new releases. For details, please consult our git page. Usually something along the lines

```
git clone https://github.com/exahype/exahype.git
```

or

```
git clone git@github.com:exahype/exahype.git
```

should do. The repository snapshot in git holds all the archives, but it also holds the sources from the archives, i.e. the tar.gz files are redundant. There is consequently no need to unzip them manually if you follow this variant.

While we do provide snapshots of Peano, ExaHyPE's underlying meshing component, with ExaHyPE snapshots (the tar.gz files we offer), Peano itself is not mirrored in the repository. You therefore have to download Peano separately: Visit www.peano-framework.org and follow the descriptions there. Afterwards, add two symbolic links from ExaHyPE's Peano directory to your Peano installation:

```
> cd Peano
> ls
mpibalancing multiscalelinkedcell sharedmemoryoracles
> ln -s yourPeanodirectory/src/peano
> ln -s yourPeanodirectory/src/tarch
```

[2] Please note that the ExaHyPE consortium uses internally a private repository, i.e. all remarks here refer to the public ExaHyPE release.

```
> ls
mpibalancing multiscalelinkedcell peano sharedmemoryoracles tarch
```

## Finish the setup

Once you have unzipped all the archives into a directory of your choice, you should see something like

```
> cd mywellsuiteddirectory
> ls
ExaHyPE ExaHyPE.jar ExaHyPE.tar.gz ExaHyPE-toolkit.tar.gz LICENSE.txt 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 all of your ExaHyPE scripts from hereon.

Please also check that your Peano subdirectory holds the Peano AMR directories:

```
> cd mywellsuiteddirectory
> ls Peano
mpibalancing multiscalelinkedcell peano sharedmemoryoracles tarch
```

There might be more subdirectories. However, these five are absolutely mandatory.

## 1.3. Dry run of development tools

**Remark:** The jar file deployed with ExaHyPE (in its root folder) might not work on your system. If this is the case, you have to rebuild the jar first. See the next Section 1.4 for instructions. If you rebuild the jar file, it typically is located at xxxx

To check whether you are ready to program new applications with 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 ExaHyPE development toolkit. In this case, retranslate it.

## 1.4. Retranslating the **ExaHyPE** development toolkit

ExaHyPE relies on a Java toolkit to free the developers from writing most of the glue code. We find many users experience problems with various Java versions and thus recommend to build your own toolkit for your target system. For this, unzip the toolkit sources (if you do not work with a cloned git repository)

```
mkdir MyToolkit
mv ExaHyPE-toolkit.tar.gz MyToolkit
cd MyToolkit
tar -xzvf ExaHyPE-toolkit.tar.gz
```

change into the directory and type in

```
make compile
make jar
```

You obtain a new jar file `ExaHyPE.jar` in the local directory that you should use from hereon.

ExaHyPE's toolkit uses the compiler compiler frontend SableCC (sablecc.org). If you also want to regenerate the front-end, you have provide SableCC to the makefile. Usually, this should not be necessary.

## 1.5. 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. Demonstrator applications

We provide a small number of out-of-the-box ExaHyPE applications. They suit three purposes:

1. They allow users to assess whether their installation is working, and which size/characteristics of ExaHyPE codes their system is able to run. They act as technical assessment exercise.
2. They allow newbies to study particular technical concepts. As some demonstrator codes are minimalistic, it is easier for users to find out how certain things are realised than searching for features in large-scale applications.
3. Within the underlying EU H2020 ExaHyPE project, the team had committed to publish some grand challenge software. Our demonstrators cover (parts of) these grand challenge codes.

## 2.1. Minimalistic Finite Volumes for the Euler equations

We provide a complete Finite Volume implementation of a plain Euler solver realised with ExaHyPE. This solver relies on a Rusanov solver and can, with only a few lines, be changed into an ADER-DG scheme later. Indeed, it can be used by an ADER-DG Euler as a limiter.

### 2.1.1. Download

The demonstrator is available from ExaHyPE's release web pages as tar archive. This archive has to be unzipped. If you work with a clone of the repository, you find it in the Demonstrators subdirectory.

### 2.1.2. Preparation

The archive contains solely files modified to realise the solver. All glue code are to be generated with the toolkit. Before we do so, we open the unzipped file EulerFV.exahype. This simple text file is the centerpiece of our solver. It specifies the solver properties, the numerical schemes, and it also holds all the mandatory pathes:

```
exahype-project EulerFVM

  peano-kernel-path const = ./Peano
  exahype-path const = ./ExaHyPE
  output-directory const = ./Demonstrators/EulerFV

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

  solver Finite-Volumes MyEulerSolver
    variables const = rho:1,j:3,E:1
    patch-size const = 10
    maximum-mesh-size = 5e-2
    time-stepping = global
    kernel const = generic::Godunov
    language const = C
    plot vtk::Cartesian::cells::ascii EulerWriter
      variables const = 5
      time = 0.0
      repeat = 0.5E-1
      output = ./variables
    end plot
```

```
  end solver
end exahype-project
```

To prepare this example for the simulation, we have to generate all glue code

```
> java -jar Toolkit/dist/ExaHyPE.jar --not-interactive \
    Demonstrators/EulerFV/EulerFV.exahype
```

and afterwards change into the application's directory (`Demonstrators/EulerFV`) and type in `make`. Depending on your system, you might have to set/change some environment variables or adopt pathes, but both the toolkit and the makefile are very chatty. For the present demonstrator, it is usually sufficient to set either

```
export COMPILER=Intel
```

or

```
export COMPILER=gnu
```

In both modes, the makefile defaults to `icpc` or `g++`, respectively. To change the compiler used, export the variable `CC` explicitly.

> **Design philosophy:** Most modifications to the specification file do not require you to rerun the `ExaHyPE` toolkit. A rerun is only required for changes to parameters with a trailing `const` or if you add more solvers or plotters to the specification file.

### 2.1.3. Run

Once the compile has been successful, you obtain an executable `ExaHyPE-EulerFVM` in your application-specific directory. `ExaHyPE`'s specification files always act as both specification and configuration file, so you start up the code by handling it over the spec file again:

```
./ExaHyPE-EulerFV ./EulerFV.exahype
```

A successful run yields a sequence of `vtk` files that you can open with Paraview or VisIt, e.g. There are two quantites plotted: The time encodes per vertex and quantity at which time step the data has been written. For these primitive tests with global time stepping, this information is of limited value. It however later makes a big difference if parts of the grid are allowed to advance in time asynchronous to other grid parts. The second quantity $Q$ is a vector of the real unknowns. The underlying semantics of the unknowns is detailed in Section 5. For the time being, you may want to select first entry which is the density or the last quantity representing the local energy. The plot should look similar to figure 2.1.

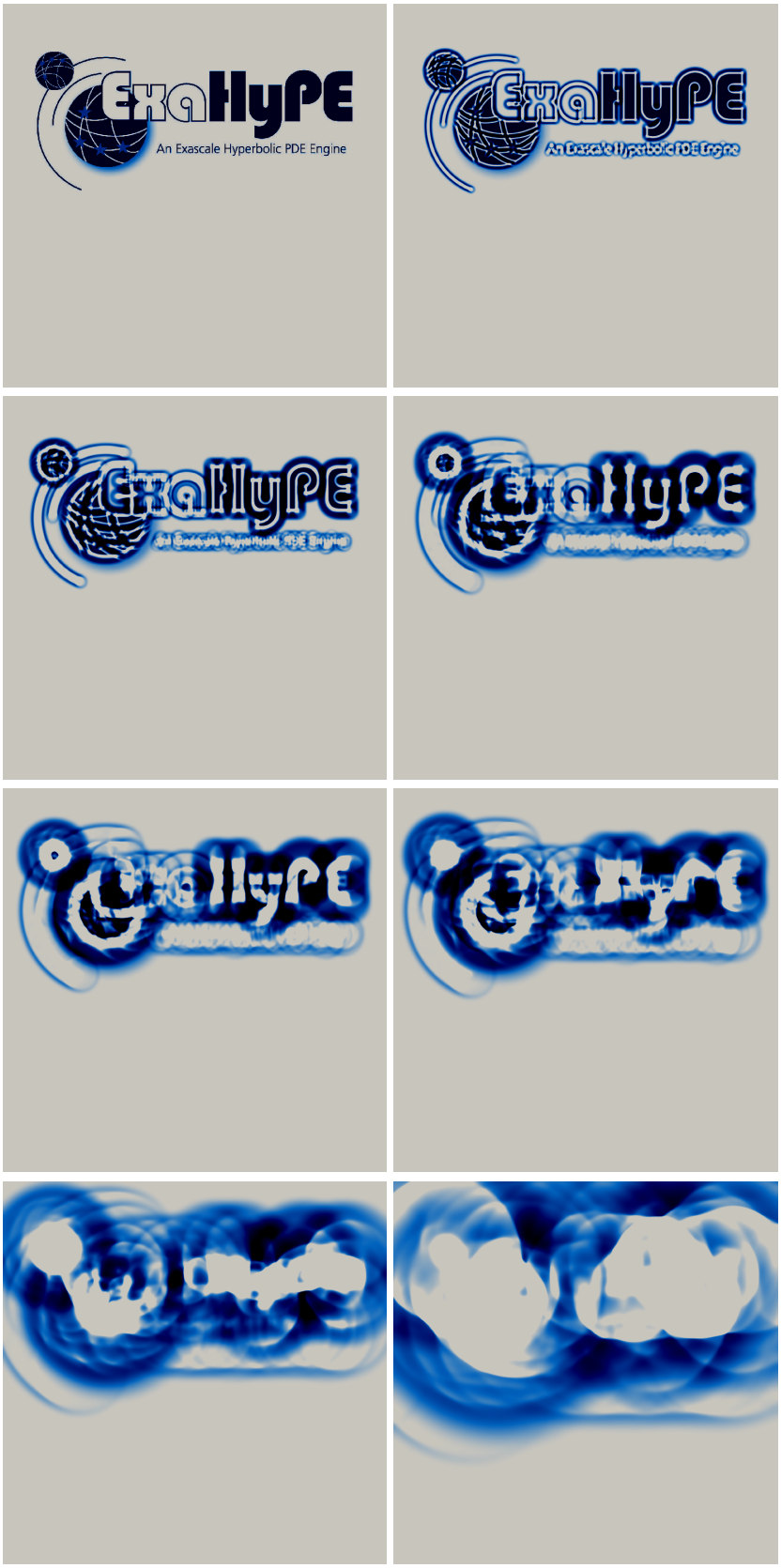



Figure 2.1.: Snapshots of the time evolution of $Q_4$, ie. the energy distribution in the EulerFlow Finite Volume demonstrator code.

# 3. Source code documentation

ExaHyPE realises an "everything is in the code" philosophy. This guidebook give a high level overview and some references as well as recipes that do not align directly with a particular code snippet. Our papers detail the mathematical ideas, scientific outcomes and algorithmic concepts. Everything else is documented right inside the code.

Our paradigm is to document our code in the header files in JavaDoc syntax, i.e. documentation reads as

```
/**
 *
 *This is an ExaHyPE comment.
 *
 */
```

No important code documentation is found inside the actual source code. It is always in the header. Typical header documentation comprises information alike

- what does a function do?
- how is a function to be used?
- how is the function's semantics realised?
- which alternatives have been considered (and probably discarded throughout the development process)?
- what bugs did arise, how have they been fixed, and why did the drop in in the first place?

## Accessing the documentation

Modern integrated development environments (IDEs) such as Eclipse parse header files and extract our documentation automatically. They then are able to display documentation on-the-fly within the editor. This is our major motivation to stick to the `/** ... */` syntax and to the "everything is documented in the header" convention.

If you prefer not to use an IDE, the ExaHyPE project extracts all documentation from all source code parts automatically each and every night and creates a webpage (Figure 3.1) from this information. The webpage is available through

exahype.eu[1] and provides, besides hyperlinks and sketches of algorithmic concepts, a search engine.

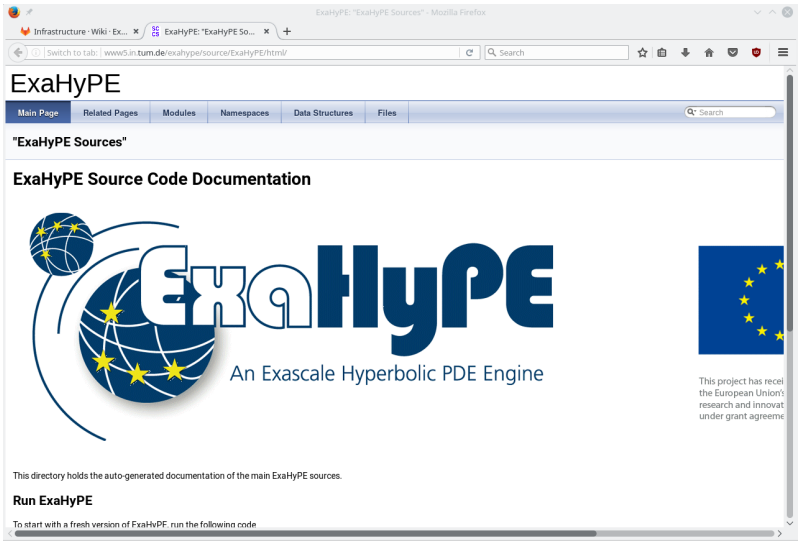



Figure 3.1.: The ExaHyPE source code documentation webpage that is updated every night.

ExaHyPE's adaptive meshing is based upon the framework Peano. You find both Peano and its source code documentation at peano-framework.org.

## Create all documentation locally

The webpage generation is based upon the open source tool Doxygen (doxygen.org). If you prefer to have all source code documentation in HTML form on your local computer, you may prefer to change into ExaHyPE's ExaHyPE source directory and type in

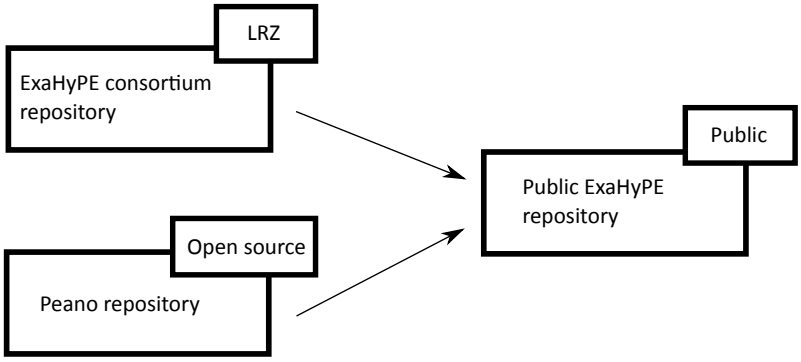
```
doxygen exahype.doxygen-configuration
```

By default, no particular application is made part of the source code documentation. All HTML pages refer to the pure engine. Also, we do not document any toolbox/feature from Peano. It is however easy to augment your local documentation by these details. For this, you just extend the source/parsed files in the configuration file `exahype.doxygen-configuration`.

[1]Should the official page be down, you may access the documentation via its direct link `http://www5.in.tum.de/exahype/source/ExaHyPE/html`.

# 4. Development and release process

ExaHyPE relies on two repositories managed via git. A private repository is used by the consortium members for day-to-day development. A public repository is befilled with source code snapshots in regular intervals. We also maintain a limited number of demonstrator applications in the public repository. If you wish to contribute, the consortium is happy to integrate any changes back into the internal ExaHyPE repository and from thereon publish it on the public mirror.




ExaHyPE relies on the several third party components. The most prominent one is Peano which acts as our adaptive mesh refinement code base. As Peano is the only part of the project that is mandatory—all ExaHyPE applications need it to run—we include the source code in our ExaHyPE archives and we also mirror the sources on the public repository.

Our community support and interaction is based upon four pillars.

1. We manage our code releases through the public repository, i.e. code releases including code deltas are available to every ExaHyPE user.
2. We maintain a public issue tracking platform together with the repository. For questions, suggestions and bug reports, please use this platform.
3. We run internally a nightly build service. It compiles ExaHyPE with various compiler and feature combinations, it checks whether a large number of unit tests pass, and it also runs a couple of integration tests (convergence runs) and performance analyses.
4. We maintain a guidebook that we continually improve and augment by frequently asked questions.

While we do not follow a formalised development process, but work internally test-driven with small ad-hoc teaming that liase for sprints.

# Part II.

# Developing new **ExaHyPE** applications

# 5. Preparing a new **ExaHyPE** development project

> **Design philosophy:** ExaHyPE users typically write one code specification per file experiment plus machine plus setup combination.

An ExaHyPE specification file acts on the one hand as classic configuration files passed into the executable for a run. It holds all the constants (otherwise often passed via command lines), output file names, core numbers, and so forth. On the other hand, a specification file defines the solver's characteristics such as numerical scheme used, polynomial order, used postprocessing steps, and so forth. Specification files translate the ExaHyPE engine into an ExaHyPE application. As they realise a single point-of-contact policy, ExaHyPE fosters reproducability. We minimise the number of files required to uniquely define and launch an application. As the specification files also describe the used machine (configurations), they enable ExaHyPE to work with machine-tailored engine instances. The engine can, at hands of the specification, optimise aggressively. If in doubt, ExaHyPE prefers to run the compiler rather than to work with 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.

> **Design philosophy:** The specification file parameters that are interpreted by the toolkit are marked as `const`. If constant parameters or the structure of the file (all regions terminated by an `end`) change, you have to rerun the toolkit. All other variables are read at runtime, i.e. you may alter them without a rerun of the toolkit or a re-compile.

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

If you run various experiments, you start from one specification file and build up your application from hereon. Once the executable is available, it can either be passed the original specification file or variants of it, i.e. you may work with sets of specification files all feeding into one executable. However, the specification files have to match each other in terms of const variables and their structure. ExaHyPE itself checks for consistent files in many places and, for many invalid combinations, complains.

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

```
exahype-project TrivialProject
  peano-kernel-path const = ./Peano
  exahype-path const = ./ExaHyPE
  output-directory const = ./Demonstrators/TrivialProject

  computational-domain
    dimension const = 2
    width = 1.0, 1.0
    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 the paths[2]. Note that ExaHyPE supports both two- and three-dimensional setups.

We hand the file over to the toolkit

```
java -jar ExaHyPE.jar Demonstrators/TrivialProject.exahype
```

Finally, we change into this project's directory and type in

```
make
```

which gives us an executable. Depending on your system, you might have to change some environment variables. ExaHyPE by default for examples wants to use an Intel compiler and builds a release mode, i.e. a highly optimised code variant. To change this, type in

```
export MODE=Asserts
export COMPILER=GNU
```

All these environment variables are enlisted by the toolkit, together with recommended settings for your system. We finally run this first ExaHyPE application with

```
./ExaHyPE-TrivialProject TrivialProject.exahype
```

As clarified before, the specification file co-determines which variant of the ExaHyPE engine is actually built. You can always check/reconstruct these settings by passing in `--version` instead of the configuration file:

```
./ExaHyPE-TrivialProject --version
```

[2] You may specify pathes of individual components in more detail (cmp. Section **??**), but for most applications working with the three pathes above should be sufficient.

# 6. ADER-DG and Finite Volume solvers with generic kernels

In our prevous chapter, the simulation run neither computes anything nor does it yield any output. In this chapter, we introduce ExaHyPE generic kernels: they allow the user to specify flux and eigenvalue functions for the kernels, but delegate the actual solve completely to ExaHyPE. Furthermore, we quickly run through some visualisation and the handling of material parameters. 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 kernels. Therefore, the present coding strategy is particulary well-suited for rapid prototyping of new solvers.

**Design philosophy:** ExaHyPE realises the Hollywood principle: "Don't call us, we call you!" The user does *not* write code that runs through some data structures, she does *not* write code that determines how operations are invoked in parallel and she does *not* write down an algorithmic outline in which order operations are performed.
ExaHyPE is the commander in chief: It organises all the data run throughs, determines which core and node at which time invokes which operation and how the operations are internally enumerated. Only for the application-specific routines, it calls back user code. Application-specific means

- how do we compute the flux function,
- how are the eigenvalues to be estimated,
- how do we convert the unknowns into records that can be plotted,
- how and where do we have to set non-homogeneous right-hand sides,
- ...

We have adopted this design pattern/paradigm from the Peano software that equipts ExaHyPE with adaptive mesh refinement. Our guideline illustrates all of these steps at hands of a solver for the Euler equations.

## 6.1. Establishing the data structures

We follow the Euler equations in the form

$$\frac{\partial}{\partial t}\mathbf{Q} + \boldsymbol{\nabla}\cdot\mathbf{F}(\mathbf{Q}) = 0 \quad \text{with} \quad \mathbf{Q} = \begin{pmatrix} \rho \\ \mathbf{j} \\ E \end{pmatrix} \quad \text{and} \quad \mathbf{F} = \begin{pmatrix} \mathbf{j} \\ \frac{1}{\rho}\mathbf{j}\otimes\mathbf{j} + pI \\ \frac{1}{\rho}\mathbf{j}\,(E+p) \end{pmatrix} \tag{6.1}$$

supplemented by initial values $\mathbf{Q}(0) = \mathbf{Q}_0$ and appropriate boundary conditions. $\rho$ denotes the mass density, $\mathbf{j} \in \mathbb{R}^d$ denotes the momentum density, $E$ denotes the energy density, 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 $\gamma$, the fluid pressure is defined as

$$p = (\gamma - 1)\left(E - \frac{1}{2}\mathbf{j}^2/\rho\right). \tag{6.2}$$

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

```
exahype-project Euler2d

  peano-path const = ./Peano/peano
  tarch-path const = ./Peano/tarch
  exahype-path const = ./ExaHyPE
  output-directory const = ./ApplicationExamples/EulerFlow

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

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

    plot vtk::binary MyPlotter
      variables const = 5
      time = 0.0
      repeat = 0.05
      output = ./solution
    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 optimised ADER-DG kernels. Instead we tell it to use some generic partially optimised 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 and configured. This 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 $t = 0$. The above plotter statement creates a plotter file `MyPlotter` that allows you to alter the plotter's behaviour.

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
```

## 6.2. Study the generated code

We obtain a new directory equalling `output-directory` from the specification file if such a directory hasn't existed before. Lets change to this path. The directory contains a makefile, and it contains a bunch of generated files:

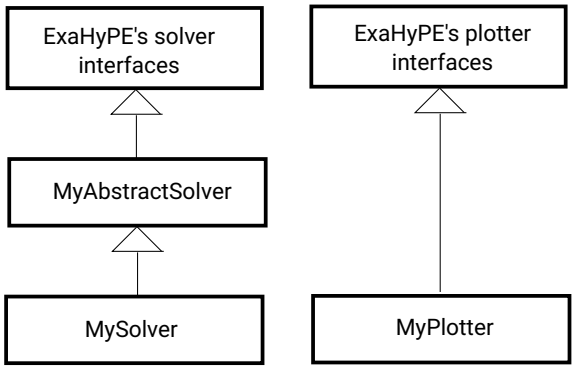



Figure 6.1.: Simplest class architecture for ExaHyPE solvers.

- `MyAbstractEulerSolver` is a class holding solely glue code, i.e. this class is not to be altered. It invokes for example the generic non-linear kernels. Once you rerun ExaHyPE's toolkit, it will overwrite this class. The class implements some interfaces/abstract classes from ExaHyPE's kernel. There is usually no need to study those at all.

- `MyEulerSolver` is the actual solver. This is where users implement the solver's behaviour, i.e. this class is to be befilled with actual code. Some methods are pregenerated (empty) to make your life easier. Have a close look into the header file—by default, the toolkit places all documentation in the headers (cmp. Section 3)—which contains hint what behaviour can be added through a particular method.

- `MyPlotter` is a class that allows us to tailor `ExaHyPE`'s output. It implements an interface and can quite freely be adopted. For the time being, the default variant dropped by the toolkit does the job. This class implements an interface from the `ExaHyPE` kernel. There's usually no need to study this interface—again, the toolkit generates quite some comments to the generated headers that you may redefine and implement.

Before we continue our tour de `ExaHyPE`, 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 garbage since no initial values are set so far.

## 6.3. Working with the arrays

> **Design philosophy:** `ExaHyPE` relies on plain arrays to encode all unknowns and fluxes. We however leave it to the user to decide whether she prefers to work with these plain double arrays (similar to Fortran) or with some symbolic identifiers. `ExaHyPE` also provides support for the latter.

### Variant A: Sticking to the arrays

If you prefer to work with plain double arrays always, it is sufficient to tell the solver how many unknowns your solution value contains:

```
solver [...]
  variables const = 5
  [...]
end solver
```

It is up to you to keep track which entry in any double array has which semantics. In the Euler equations, you have for example to keep in mind that the fifth entry always is the energy $E$. All routines of relevance are plain double pointers to arrays. So an operation altering the energy typically reads `Q[4]` (we work with C and thus start to count with 0). All fluxes are two-dimensional double arrays, i.e. are represented as matrices.

### Variant B: Working with symbolic identifiers

As alternative to plain arrays, you may instruct ExaHyPE about the unknowns you work with:

```
  solver [...]
    variables const = rho:1,j:3,E:1
    [...]
  end solver
```

This specification is exactly equivalent to the previous declaration. It tells ExaHyPE that there are five unkonwns held in total. Two of them are plain scalars, the middle one is a vector with three entries. Also, all operation signatures remain exactly the same. Yet, the toolkit now creates a couple of additional classes that can be wrapped around the array. These classes do not copy any stuff around or themselves actually alter the array. They provide however routines to access the array entries through their unknown names instead of plain array indices:

```
void Euler2d::MyEulerSolver::anyRoutine(..., double *Q) {
  Variables vars(Q); // we wrap the array

  vars.rho() = 1.0; // accesses Q[0]
  vars.j(2) = 1.0; // accesses Q[3]
  vars.E() = 1.0; // accesses Q[4]
}
```

The wrapper allows us to "forget about the mapping of the unknown onto array indices. We may use variable names instead. Besides the `Variables` wrapper, the toolkit also generates wrappers around the matrices as well as read only variants of the wrappers that are to be used if you obtain a `const double const*` argument.

We note that there are subtle syntacic differences between the plain array access style and the wrappers. The latter typically rely on `()` brackets similar to Matlab. Without going into details, we want to link to two aspects w.r.t. symbolic accessors:

1. The wrappers are plain wrappers around arrays. You may intermix both access variants—though you have to decide whether you consider this to be good style. Furthermore, all vectors offered through the wrapper provide a method `data()` that again returns a pointer to the respective subarray. The latter is useful if you prefer to work with symbolic identifiers but still have to connect to array-based subroutines.

2. All wrapper vector and matrix results rely no the linear algebra subpackage of Peano's technical architecture (`tarch::la`). We refer to Peano's documentation for details, but there are plenty of frequently used operations (norms, scalar products, dense matrix inversion, ...) shipped along with Peano that are all available to ExaHyPE.

From hereon, most of our examples rely on Variant B, i.e. on the symbolic names. This makes a comparison to text books easier. The cooking down to a plain array-based implementation is straightforward.

> **Design philosophy:** All of our signatures rely on plain arrays. Symbolic access is offered via a wrapper and always is optional. Please note that you might also prefer to warp coordinates, e.g., with `tarch::la::Vector` classes to have Matlab-style syntax.

We do not apply such a sophisticated syntactic connvention here to keep the guidebook as simple to understand as possible.

## 6.4. Setting up the experiment

We return to the Euler flow solver: 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 )) {
   Variables vars(Q);
   const double GAMMA = 1.4;

   vars.j( 0.0, 0.0, 0.0 );
   vars.rho() = 1.0;
   vars.E() =
       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.

As we only want to impose initial conditions so we check if the time `t` is zero. As these values are floating point values, standard bitwise C comparison would be inappropriate. We rely here on a routine coming along with the linear algebra subroutines of Peano to check for "almost equal".

[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.

The routine `adjustedSolutionValues` must always be paired with the routine `hasToAdjustSolution` 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 )) {
    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`. Such semantic information on coordinates can be found in the generated header files (as long as you don't overwrite them).

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 (a plot should be similar to fig 6.2).

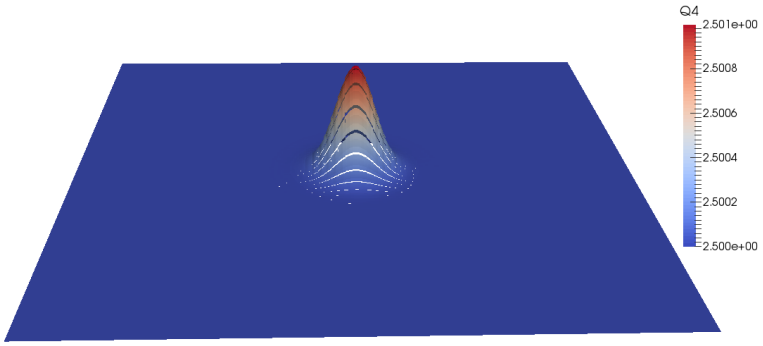



Figure 6.2.: The rest mass density $Q_0$ in the EulerFlow setup stays a constant Gaussian distribution during the time evolution in our similar setup.

**Design philosophy:** The adoption of the solution (to initial values) as well as source terms are protected by an additional query `hasToAdjustSolution`. This allows `ExaHyPE` to optimise the code aggressively: The user's routines are invoked only for regions where `hasToAdjustSolution`'s result holds. For all the remaining computational domain, `ExaHyPE` uses numerical subroutines that are optimised to work without solution modifications or source terms.

## 6.5. Realising the fluxes

To realise the fluxes, we have to fill the functions `flux` and `eigenvalues` in file `MyEulerSolver.cpp` with code.

### Variant A

In this first part, we present the realisation based upon plain arrays:

```
void Euler::FirstEulerSolver::flux(const double* const Q, double** F) {
  // Dimensions = 2
  // Number of variables = 5 (#unknowns + #parameters)
  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);

  double* f = F[0];
  double* g = F[1];

  // f is the flux on faces with a normal along x direction
  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 is the flux on faces with a normal along y direction
  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 Euler::FirstEulerSolver::eigenvalues(const double* const Q,
```

```
  const int normalNonZeroIndex, double* lambda) {
  // Dimensions = 2
  // Number of variables = 5 (#unknowns + #parameters)
  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;
}
```

### Variant B

Alternatively, we can work with symbolic identifiers if we have specified the unknowns via `rho:1,j:3,E:1`:

```
void Euler2d::MyEulerSolver::flux(
  const double* const Q,
  double** F
) {
  ReadOnlyVariables vars(Q);
  Fluxes f(F);

  tarch::la::Matrix<3,3,double> I;
  I = 1, 0, 0,
      0, 1, 0,
      0, 0, 1;

  const double GAMMA = 1.4;
  const double irho = 1./vars.rho();
  const double p = (GAMMA-1) *(vars.E() -0.5 *irho *vars.j()*vars.j() );

  f.rho ( vars.j() );
  f.j ( irho *outerDot(vars.j(),vars.j()) + p*I );
  f.E ( irho *(vars.E() + p) *vars.j() );
}
```

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

```
) {
  ReadOnlyVariables vars(Q);
  Variables eigs(lambda);

  const double GAMMA = 1.4;
  const double irho = 1./vars.rho();
  const double p = (GAMMA-1) *(vars.E() -0.5 *irho *vars.j()*vars.j() );

  double u_n = vars.j(normalNonZeroIndex) *irho;
  double c = std::sqrt(GAMMA *p *irho);

  eigs.rho()=u_n -c;
  eigs.E() =u_n + c;
  eigs.j(u_n,u_n,u_n);
}
```

The implementation of function `flux` is very straightforward. Again, the details are only subtle: We wrap up the arrays `Q` and `F` in wrappers of type `ReadOnlyVariables` and `Fluxes`. Similar to `Variables`, the definitions of `ReadOnlyVariables` and `Fluxes` were generated according to the variable list we have specified in the specification file. While `Fluxes` indeed is a 1:1 equivalent to `Variables`, we have to use a read-only variant of `Variables` here as the input array `Q` is protected. The read-only symbolic wrapper equals exactly its standard counterpart but lacks all setters.

The pointer `lambda` appearing in the signature and body of function `eigenvalues` has the size as the vector of conserved variables. It thus makes sense to wrap it in an object of type `Variables`, too.

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 figure 6.3.

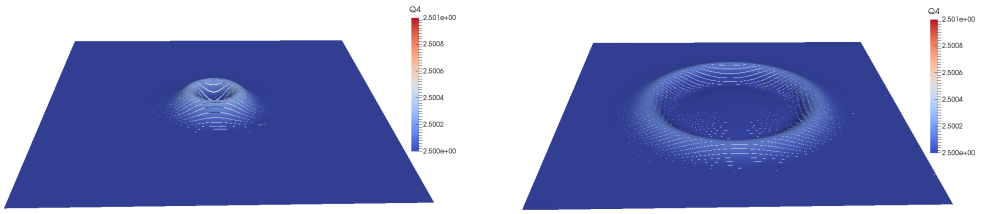


Figure 6.3.: Time evolution of $Q_0$, now with the Hydrodynamics PDEs implemented. The left figure shows an early time while the right figure shows a later time.

## 6.6. Supplementing boundary conditions

ExaHyPEoffers a generic API to implement any kind of *local* boundary conditions, ie. conditions to solve the Riemann problem at the simulation boundary which do not need to communicate with other cells. That is, you can straightforwardly[2] implement

- outflow boundary conditions (also sometimes refered to as ”null boundary conditions” or ”none boundary conditions”),
- exact boundary conditions or
- reflection symmetries.

The signature to implement your boundary conditions reads

```
void Euler::MyEulerSolver::boundaryValues(
    const double* const x, const double t, const double dt,
    const int faceIndex, const int normalNonZero,
    const double* const fluxIn,const double* const stateIn,
    double* fluxOut, double* stateOut)
    {
    ...
    }
```

The input arguments (marked with the C `const` modifier) offer the position of a cell's boundary and time as well as the local timestep of the cell, and the cell's state and flux variables. Cell hereby always is inside the computational domain, i.e. ExaHyPE queries the boundary values from the inner cell's point of view.

The two variables `faceIndex` and `normalNonzero` to answer the questions at which boundary side we currently are. The face index decodes as

```
0-left, 1-right, 2-front, 3-back, 4-bottom, 5-top
```

or in other terms

```
0 x=xmin 1 x=xmax, 2 y=ymin 3 y=ymax 4 z=zmin 5 z=zmax
```

This is also encoded in figure 6.4.

[2] In this guidebook, this implies that we stick to those guys only.

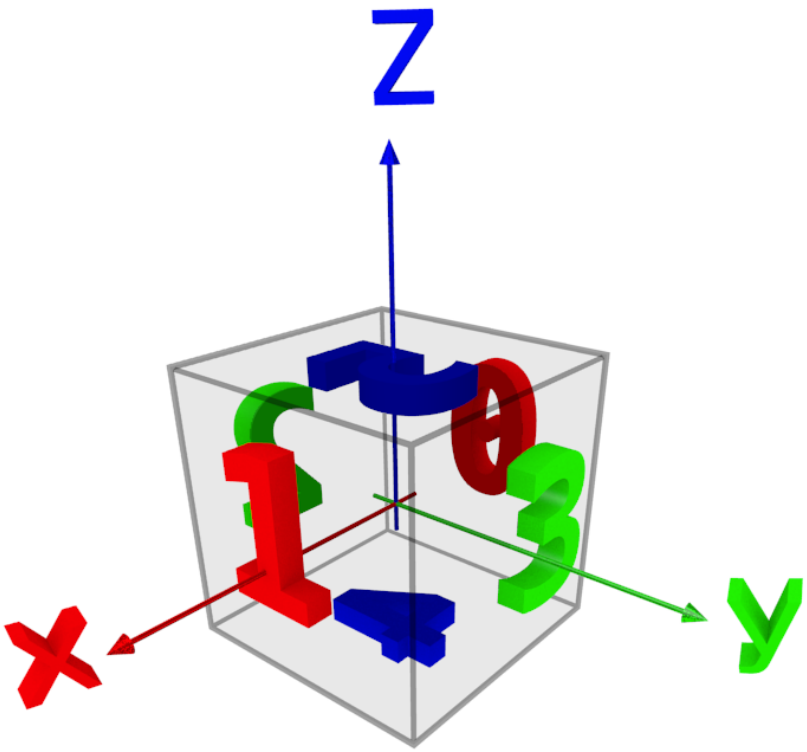



Figure 6.4.: The face indexes as named in the ExaHyPE code

## 6.7. Finite Volumes

If you prefer your code to run with Finite Volumes, ExaHyPE sticks to all paradigms introduced so far. The user has to provide basically fluxes, eigenvalues and boundary/initial conditions. All such information is already available here. Consequently, switching from ADER-DG to Finite Volumes is a slight modification of the configuration file and a rerun of the toolkit:

```
solver Finite-Volumes MyFVEulerSolver
  variables = rho:1,j:3,E:1
  patch-size = 3
  maximum-mesh-size = 0.1
  time-stepping = global
  kernel = generic::Godunov
  language = C
end solver
```

> **Design philosophy:** With ADER-DG, ExaHyPE embeds a higher order polynomial into each grid cell. With Finite Volumes, ExaHyPE embeds a small patch (a regular Cartesian grid) into each grid cell.

We switch the solver type to `Finite-Volumes` and fix a patch resolution, before we recompile the ExaHyPE application and run a Finite Volume solver instead of the ADER-DG variant.

# 7. Some advanced solver features

## 7.1. Multiple solvers in one specification file

ExaHyPE specification files can host multiple solvers. In this case, multiple solvers are simultanously hosted within one compute grid. This feature can be advantageous for parameter studies, e.g., as all grid managements, parallelisation, load balancing, ...overhead is amortised between the various solvers ran simultaneously.

## 7.2. Runtime constants/configuration parameters

There are various ways to add application-specific constants to your code. While it is reasonable to extend/tailor the code to make it accept additional runtime parameters on the command line, e.g., our original design philosophy was to have everything in one file. This means, that many application need application-specific settings in this file. Therefore, ExaHyPE allows you to add a `constants` section to your solver:

```
  solver ADER-DG MyEulerSolverWithConstantsFromSpecFile
    variables const = rho:1,j:3,E:1
    order = 3
    maximum-mesh-size = 0.5
    time-stepping = global
    kernel const = generic::fluxes::nonlinear
    language const = 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::MyEulerSolverWithConstantsFromSpecFile::
  MyEulerSolverWithConstantsFromSpecFile(
  ...,
  exahype::Parser::ParserView constants):
  exahype::solvers::ADERDGSolver(...) {
  if (constants.isValueValidDouble( "rho" )) {
    double rho = constants.getValueAsDouble( "rho" );
    // do some magic with rho
  }
}
```

**Design philosophy:** ExaHyPE is an engine that focuses on simplicity, capability and speed. We do not support the reuse of one solver type with different constants in one file or the reuse of a solver between various projects (though both features should be straightforward to implement within an application). However, you can create multiple specification files differing in their constants section to feed them into one solver, i.e. constants are not built into the executable.
Lastly remember that you can always specify file paths in the `constants` section and then use your own application-specific file parser.

## 7.3. 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 $\Delta t$ irrespective of its spatial resolution. If one cell in the domain finds out that $\Delta t$ harms its CFL condition, all cells in the domain are rolled back to their last time step, $\Delta t$ 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 $\Delta t$ 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 $\Delta t$. From hereon, it uses this $\Delta t$ for all subsequent time steps. If it turns out that the CFL condition is harmed through

$\Delta t$ 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.

## 7.4. Material parameters

ExaHyPE does not "natively" distinguish unknowns that are subject to a PDE from parameters on the user side. It however provides a `parameter` keyword. Syntax and semantics of this keyword equal the `variables`, i.e. you may either just add a parameter quantity or give the parameters symbolic names.

```
solver ADER-DG MyEulerSolverWithParameters
  variables const = rho:1,j:3,E:1
  parameters const = MaterialA:1,MaterialB:2
  order const = 3
  maximum-mesh-size = 0.5
  time-stepping = global
  kernel const = generic::fluxes::nonlinear
  language const = C
end solver
```

Technically, parameters are simply appended to the actual unknows: If you specify 4 unknowns and 3 (material) parameters, all ExaHyPE functions just seem to handle 4+3=7 variables. Notably, one set of parameters is associated to each integration point. Parameters hence are directly accessed in `Q` via the indices 4,5 and 6 (in this example) or via the generated `Variables` array wrappers.

Obviously, material parameters may not change over time—unless an application code manually resets them in `adjustSolution`.

> **Design philosophy:** ExaHyPE models material parameters as additional unknowns $Q_p$in the PDE that are subject to $\partial_t Q_p = 0$.

This paradigm is not revealed to the user code that always receives a native flux and eigenvalue function with all variable plus parameter entries. However, the ExaHyPE internally sets all fluxes and eigenvalues in the corresponding routines to zero, i.e. they are a posteriori eliminated from any equation system. This way, parameters do not diffuse and are not transported.

## 7.5. Alternative symbolic naming schemes

ExaHyPE allows you to specify an arbitrary number of additional symbolic naming schemes other than `variables` and `parameters` per solver. These naming schemes must be listed below the field `variables` or below `parameters` if the latter is present.

To give an example: We might prefer to compute the fluxes in primitive variables instead of the conserved ones. We thus add a symbolic naming scheme "`primitives`" to our solver:

```
solver ADER-DG MyEulerSolverWithParametersAndPrimitives
  variables const = rho:1,j:3,E:1
  parameters const = MaterialA:1,MaterialB:2
  primitives const = rho:1,u:3,E:1
  order const = 3
  [..]
end solver
```

The ExaHyPE toolkit will then generate another array wrapper named `Primitives` which can be accessed in similar ways as `Variables` and `Parameters`.

# Part III.

# Upscaling and tuning **ExaHyPE**

# 8. 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.

The dummy strategy as displayed above is a reasonable starting point to assess whether your code is correctly translated and started. 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 8.1.: 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`<br>`autotuning`<br>`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. |

## 8.1. Tailoring the shared memory configuration

Shared memory performance can be very sensitive to machine-specific tuning parameters: Peano, ExaHyPE's AMR code base, relies on static problem subpartitioning and thus needs precise instructions which problem sizes for particular parts of the code are convenient. We thus encourage high performance applications to tailor ExaHyPE's shared memory parallelisation to get the most out of their machine.

Key ingredient to do so is the usage of a different strategy than the `dummy` plus the configuration/properties files read by these strategies. All non-dummy strategies load tuning parameters from the properties file. If a file is specified but does not exist, they fall back to default/empty settings as detailed below. If your application terminates correctly, all strategies pipe updated information back into the properties file. This way, knowledge can be reused in the next run or runtime characteristics can be analysed.

It is obvious that proper property files depend on the machine you are using. They reflect your computer's properties. Yet, 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 that depend on your simulation characteristics.

### Autotuning

Our most convenient shared-memory tailoring relies on the `autotuning` strategy. Autotuning starts with serial configurations for all program parts. Once it has obtained a reasonable accurate estimate of how long each program part runs, it tries to deploy various parts of the code among multiple cores. We typically start with two cores, four cores, and so forth unless subproblems consist of large arrays that immediately can be broken up into more chunks. If a problem decomposition improves the runtime, the oracle continues to break it up into even smaller chunks to exploit more cores until the decomposition either does not improve the runtime anymore or even makes the performance worse. In this case, we roll back to the last reasonable configuration. This means that first runs might be really slow, but the runtime improves throughout the development.

The autotuning switches off per code segment automatically as soon as it has to believe that best-case parameters are found. As a result, timing overheads are reduced. As any configuration might correspond to a local runtime minimum, the autotuning restarts the search from time to time. This restart is randomised.

Peano provides a Python script to translate all measurements into a big HTML table. The script is located within `Peano/sharedmemoryoracles`. If you invoke it without parameters, you obtain a detailed usage message.

If a properties file does exist at startup already, `ExaHyPE` loads this propery file, i.e., the autotuning starts from the previous properties dump. This way, long-term learning can be split among various program invocations.

- As the autotuning reads 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. Switching off is also done within the text files. You might for example search for optimal choices with the sampling and then feed these values into the autotuning. The format of the properties file is documented as comment within the file itself.

- The autotuning requires your OS to offer real-time timers. We have encountered various situations, notably on Intel KNL, where timers seem not to work properly. In this case, the autotuning works if and only if a properties file from another machine is handed in, i.e., a configuration from a different machine is used. Please note that disfunctional autotuning might mess up your properties files. In this case, they have to be replaced with copies from a different machine prior to each run.

- Large simulation runs seem to yield runtime data that varies strongly. It thus requires the autotuning to measure quite some time before code regions are identified that scale. We recommend to run autotuning first on small problem setups. The dumped properties files then can be reused by larger runs. If `ExaHyPE` is passed an autotuning configuration from a small run, it extrapolates measurements therein as initial data for the autotuning and then continues to optimise further.

### Configuration sampling and manual tuning

To get the whole picture which code fragments perform in which way on your machine, you might want to run the `sampling` strategy. 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.

Peano provides a Python script to translate all measurements into graphs. The script is located within `Peano/sharedmemoryoracles`. If you invoke it without parameters, you obtain a detailed usage message.

If a properties file does exist at startup already, `ExaHyPE` loads this propery file, i.e., the statistics are incrementely improved. This way, a long-term statistical analysis can be split among various program invocations.

## 8.2. Hybrid parallelisation

If you want to combine shared memory parallelisation with MPI, nothing has to be done in most cases. However, the tailoring of a reasonable number of ranks per node plus a efficient number of threads per rank can be tedious. You cannot expect that a hybrid code shows the same shared memory speedup as a code that is parallelised with shared memory only.

ExaHyPE supports multithreaded MPI and most MPI implementations nowadays support multithreaded calls. However, we found that most applications do not benefit from hyperthreaded MPI or even are slowed down. It thus is disabled by default in ExaHyPE. If you want to use hyperthreaded MPI in your code, please comment the line

```
PROJECT_CFLAGS+=-DnoMultipleThreadsMayTriggerMPICalls
```

in your autogenerated makefile out.

If you use autotuning with MPI on $p$ ranks, please note that ExaHyPE disables the learning on all ranks besides rank $p-1$. While rank $p-1$ reads in the properties file and tries to improve parameters specified therein, all remaining ranks do read the properties file, too. They however do not alter the file's settings.

Once your code terminates, rank $p-1$ dumps any (improved) parameter choice. If you use grain size sampling, rank $p-1$ dumps its statistics. The statistics from all other ranks are not dumped persistently. If your rank $p-1$ successively improves the setting in the properties file, better parameter findings automatically will be available to the other ranks in the next program run.

# 9. 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 (cf. table 9.1).

| 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` | $\geq 1$ | Ignored if `fair` is not specified. It instructs the load balancing how many ranks are placed on one node. |

Table 9.1.: MPI variant `static_load_balancing`

## 9.1. 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 $\ell$, the load balancing tries to make each node, i.e. every `ranks_per_node`th 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.

## 9.2. Hybrid parallelisation

Please see Section 8.2 for details how MPI and the shared memory parallelisation interplay.

# 10. Optimisation

## 10.1. 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
```

Restrict adaptivity pattern. For many setups, it makes sense to disable the dynamic AMR from time to time; when multiple iterartions are batched and the code runs a fixed number of iterations or when the previous grid iteration did not identify any refinement and thus we have to expect that the probability of a refinement in the subsequent iteration is low, e.g. Often, the CFL condition in local time stepping requires the code to perform $N$ grid sweeps and it is sufficient if the grid is adopted afterwards. This can speed up the code in many cases. To tell ExaHyPE to throttle the dynamic adaptivity, set the flag to `on`.

```
optimisation
  ...
  disable-amr-if-grid-has-been-stationary-in-previous-iteration = on
end optimisation
```

Modify storage precision. ExaHyPE internally can store data in less-than-IEEE precision. This can reduce the memory footprint of the code signficantly. To allow ExaHyPE to do so, the user has to specify through `double-compression` which accuracy actually is required. If the field is set to zero, ExaHyPE works with full double precision and no compression is activated.

While the data is compressed, ExaHyPE nevertheless computes with double precision always. Consequently, data has to be converted forth and back: Any compressed data is converted into IEEE precision before any arithmetic operation and later on compressed before it is written back to the main memory. This process is costly, though we can spawn it into background threads. This is particularly convenient if you have idle cores 'left' and controlled via the flag `spawn-double-compression-as-background-thread`.

```
optimisation
  ...
  double-compression = 0.0
  spawn-double-compression-as-background-thread = off
end optimisation
```

```
optimisation
  ...
  disable-amr-if-grid-has-been-stationary-in-previous-iteration = on
end optimisation
```

## 10.2. 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`.

Specification file The path to libxsmm have to be given in the specification file under the path to ExaHyPE:

```
exahype-project MyEulerFlow
  ...
  exahype-path = ./ExaHyPE
  libxsmm-path = ./Libxsmm
  output-directory = ./ApplicationExamples/EulerFlow
  ...
```

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

Table 10.1.: Supported flags

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. The supported options for this flag are given in table 10.1. 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.

# Part IV.

# Working with **ExaHyPE**'s output

# 11. **ExaHyPE** program output

In this section, we discuss how to tailor the ExaHyPE program output to your needs. We focus on direct output such as terminal messages. Information on supported output file formats can be found in Chapter 12 while real on-the-fly postprocessing such as the computation of global integrals is picked up in Chapter 13.

## 11.1. Logging

ExaHyPE creates a large number of log files and log messages; 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 some performance analyses requires several log statements to be switched on. We refer to Peano's handbook, the usage descriptions of the analysis postprocessing scripts and the respective guidebook sections for details.

By default, Peano pipes all output messages to the terminal, and many developers thus pipe such output into a report file. Alternatively, you can add a

```
log-file = mylogfile.log
```

statement to your specification file right after the `architecture` flag. If such a statement exists ExaHyPE pipes the output into the respective log files. Furthermore, one log file is used per grid sweep if you translate with assertions or in debug mode where lots of information is dumped. This enables you to search for particular information of a particular grid sweep. Once you translate your code with

MPI support, each rank writes a log file of its own. This way, we avoid garbage in the output files if multiple MPI ranks write concurrently as well as congestion on the IO terminal.

## 11.2. Grid statistics

By default, ExaHyPE does not plot statistics about the grid such as used mesh widths. There are multiple reasons why we refrain from this:

- ExaHyPE makes the compute grid host the solution but often chooses on purpose finer grids for efficiency reasons.
- Most applications can derive information about the actual compute grids used from the output files and a generic plot thus would be redundant.
- In MPI, ExaHyPE tries to avoid global communication whereever possible, i.e. any global grid statistics written out can be corrupted. If you use plot routines, all data is consistent, but this comes at the price of some additional synchronisation.

For many codes it makes, at least throughout the development phase, however sense to track grid statistics such as used mesh sizes and vertex counts. To enable it, please translate your code with the option `-DTrackGridStatistics`. For this, we propose to add the line

```
PROJECT_CFLAGS+=-DTrackGridStatistics
```

to you makefile. Following a recompile, you obtain data similar to

```
48.5407 info step 13 t_min =0.0229621
48.5407 info dt_min =0.00176632
48.5408 info memoryUsage =639 MB
48.5408 info inner cells/inner unrefined cells=125479/59049
48.5408 info inner max/min mesh width=[5,5]/[0.0617284,0.0617284]
48.5408 info max level=6
```

# 12. Plotting

In this chapter, we start with an overview over various output formats supported by ExaHyPE. In a second part, we discuss how to pass selections over to the simulation engine. Often, you are not interested in the whole domain or all variables, so notifying the code base about the regions of interests allows the code to stream out only the data actually required which makes it faster. We wrap up with some remarks how to modify/postprocess variables on-the-fly before they are written. An all time classic of such an in-situ postprocessing is the conversion of conservative into primitive variables of the elimination of helper variables from the output.

> **Design philosophy:** ExaHyPE's IO is per se very simplistic as we do ackwnowledge that most applications have very application-specific plotting demands. We thus decided on purpose that we support only very few output formats, do not support sophisticated plotting (such as symbolic names onto output fields) or complex filtering. However, we do offer the infrastructure to realise sophisticated plots on the application side that do scale up.

Most output formats of ExaHyPE are standardised output formats such as VTK. There are a number of Open-Source tools available to interactively watch, inspect and render these 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.

## 12.1. DG polynomials in the **ExaHyPE** output

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 12.1.

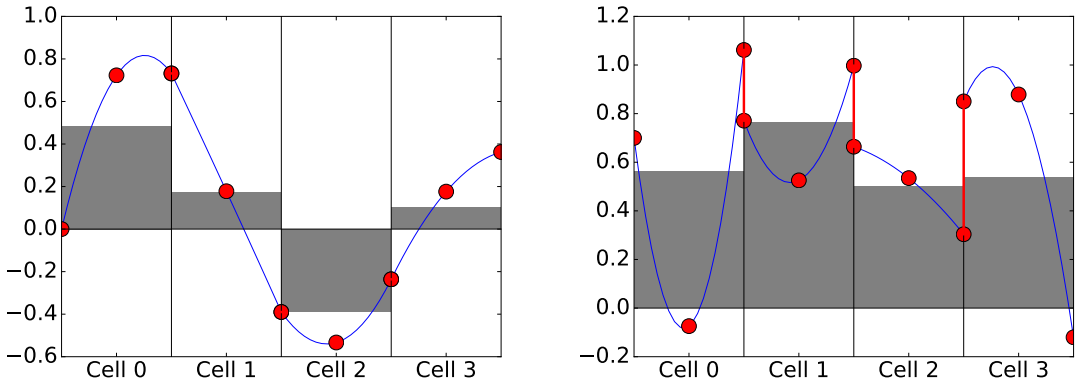



Figure 12.1.: A sketch of ExaHyPE's output point structure in a 1D example with four cells and a polynomial degree of 2. In each example, the polynomial degree is plotted. The shaded regions indicate the cell average. In the regular case of a continous solution, there are two data points at the cell 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.

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.

## 12.2. Overview of supported data output formats

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.

### Volumetric plotters

We currently support the following combinations for volumetric data (snapshots):

$$\texttt{vtk} \times \left\{ \begin{matrix} \texttt{Legendre} \\ \texttt{Cartesian} \end{matrix} \right\} \times \left\{ \begin{matrix} \texttt{Vertices} \\ \texttt{Cells} \end{matrix} \right\} \times \left\{ \begin{matrix} \texttt{Ascii} \\ \texttt{Binary} \end{matrix} \right\}$$

**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 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.

### Non-volumetric plotters

**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.

## 12.3. Filtering

By default, ExaHyPE offers two types of built-in filtering: You can use the `select` statement to impose spatial constraints on the output data, and you can constrain the number of variables written.

The spatial selection mechanism is described above. We built it into the ExaHyPE kernel as IO has a severe performance impact. It synchronises the involved MPI ranks. If we know that only variables of a certain region are of interest, only ranks responsible for these regions have to be synchronised, i.e. the resulting code is faster.

With a restriction on only a few output parameters, you can reduce the output data file and thus reduce the pressure on the memory system. By default, the first $k$ unknowns are streamed if $k$ unknows are to be written. However, you can use the postprocessing techniques discussed below to plot other unknowns from your unknown set.

## 12.4. Parameter selection and conversion

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 const = 5
    order const = 3
    maximum-mesh-size = 0.1
    time-stepping = global
    kernel const = generic::fluxes::nonlinear
    language const = C

    plot vtk::Cartesian::ascii MySolutionPlotter
      variables const = 2
      time = 0.0
      repeat = 0.05
      output = ./solution
    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];
}
```

# 13. Postprocessing

## 13.1. On-the-fly computation of global metrics 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.

## 13.2. Reduction of global quantities over all MPI ranks

The code snippets so far are unaware of any MPI parallelisation. ExaHyPE does disable any shared memory parallelisation if you decide to plot—so ensure that there is a reasonable time in-between any two plots, even if they do only derive a single scalar—but it does not automatically synchronise the plotters between any two MPI ranks at any time.

We try to make ExaHyPE minimalistic and easy to handle, maintain and learn. Predefined reduction routines—reduction means all MPI ranks combine their results into one final piece of data on one rank—would contradict this objective and require us to come up with a comprehensive list of possibly required reductions. So we decided to make the programming of reductions simple rather than offering as many as possible reductions out-of-the-box.

Reducing data from all MPI ranks is a popular task in distributed memory programming. Therefore, MPI offers a collective operation to do this. The term collective here means that all MPI ranks are involved. The term collective also reasons why we may not use MPI collectives in ExaHyPE. Our dynamic load balancing, notably in combination with dynamic mesh refinement, may remove MPI ranks from the computation at any time; just to use them for other tasks later on that are urgent. You can never rely that really all MPI ranks do work (though most of the time, all of them will do). As a result, we have to program the reduction manually. We discuss how to do this by means of the summation of values from all plotters running on MPI ranks at a certain time.

- If you have code parts that you want not to be translated if no MPI is activated, protect it via

```
#ifdef Parallel
...
#endif
```

- ExaHyPE runs plotters only on "active" MPI ranks. If an MPI rank currently is not used by the (dynamic) load balancing, no plotter ever is started on this rank. At the same time, ExaHyPE's Peano code base identifies a *global master* rank. This rank is always working. If you have to reduce data, it is convenient always to reduce all data on the global master. At any point, you can find out whether you are on the global master via

```
#include "tarch/parallel/Node.h"
...
if (tarch::parallel::Node::getInstance().isGlobalMaster()) {
  ...
}
```

Please note that `isGlobalMaster()` is always defined. If you run without MPI, it always returns `true` as there is only one instance of ExaHyPE running.

- Peano, which is ExaHyPE's grid management engine, uses literally hundreds of different message types running through the network at any time simultaneously. It is important that we do not interfer with any ExaHyPE core messages if we postprocess data. To ensure this, we propose to *tag* messages, i.e. to give them a label to say "these are for my postprocessing" only. Peano offers a tagging mechanism that we use here.

Our proposed reduction now realises the following three ideas:

1. We do all the reduction at the end of `finishPlotting` where we can assume that the reduced data per rank is available.
2. If we run on any rank that is not the global master, `finishPlotting` sends its data to the global master.
3. The global master's `finishPlotting` runs over all ranks that are not idling around and collects the data from them.

```
#include "tarch/parallel/Node.h"
#include "tarch/parallel/NodePool.h"

...

void MyEulerSolver_Plotter0::finishPlotting() {
 // All the stuff we have done before comes here
 // ...
 const int myMessageTagUseForTheReduction =
  tarch::parallel::Node::getInstance().reserveFreeTag(
   "MyEulerSolver_Plotter0::finishPlotting()" );

 double myValue = ...;

 if (tarch::parallel::Node::getInstance().isGlobalMaster()) {
  double receivedValue;
  for (
   int rank=1;
   rank<tarch::parallel::Node::getInstance().getNumberOfNodes(); rank++
  ) {
   if ( !tarch::parallel::NodePool::getInstance().isIdleNode(rank) ) {
    MPI_Recv( &receivedValue, 1, MPI_DOUBLE, rank,
     myMessageTagUseForTheReduction,
     tarch::parallel::Node::getInstance().getCommunicator(),
     MPI_STATUS_IGNORE );
    myValue += receivedValue;
   }
```

```
  }
 }
 else {
  MPI_Send( &myValue, 1, MPI_DOUBLE,
   tarch::parallel::Node::getGlobalMasterRank(),
   myMessageTagUseForTheReduction,
   tarch::parallel::Node::getInstance().getCommunicator()
  );
 }

 tarch::parallel::Node::getInstance().releaseTag(
  myMessageTagUseForTheReduction);
}
```

The `reserveFreeTag` operation stems from Peano. We are supposed to tell the function which operation has invoked it, as this is a quite useful piece of information for MPI debugging. The last line of the code snippet returns this tag again, so it might be reused later on. Our snippet demonstrates the reduction at hands of the double variable `double`. An extension to non-double and vector types should be straightforward for developers having rudimentary knowledge of MPI.

The class `Node` is a singleton—it exists only once—representing an MPI rank. The class `NodePool` is a singleton, too, and holds all of Peano's load balancing information. Notably is knows which ranks are idling. We use those two classes here.

Our code snippet sends the value of `myValue` to the global master if we run into the routine on a rank which is not the global master itself. If the plotter is invoked on the global master, we run over all the ranks that do exist. Per rank, we ask the node pool whether the rank is idling. If it is not, we know that it will send data to the global master and we receive these data.

# Part V.

# **ExaHyPE** design & rationale

# 14. **ExaHyPE** architecture

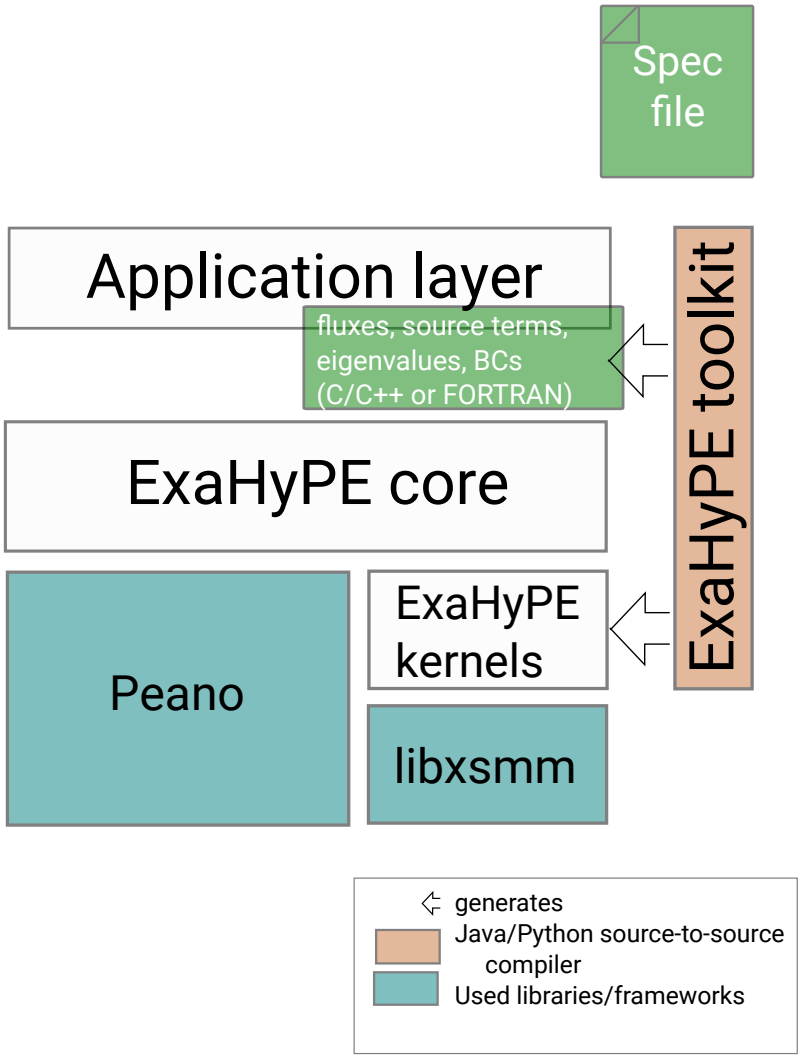



Figure 14.1.: The generic ExaHyPE architecture at a glance

ExaHyPE is a solver engine (Figure 14.1), i.e. the sole code can not solve any problem. Domain-specific code has to integrate into ExaHyPE to make it a working simulation code. All these code fragments that are PDE-specific are summarised in ExaHyPE as *application layer*.

To write an ExaHyPE code, users typically start from a specification file. We use a green colour in our cartoon to highlight that this file is to be written completely by the ExaHyPE user. The specification file is passed to the ExaHyPE toolkit that

creates lots of glue code and empty application-specific classes where the user has to fill in flux functions, eigenvalue computations, and so forth (green again). The generated glue code and these empty templates that are to be befilled make up the aforementioned *application layer*. Usually, there is no need to modify any glue, but the user is free to do so.

The core ExaHyPE code is now an application based upon the software Peano that brings together the application-specific code fragments and the dynamically adaptive Cartesian mesh. It also controls Peano's parallelisation. Peano itself is a 3rd party component and not part of the original ExaHyPE project. We thus colour it here. ExaHyPE tailors its generic features towards ADER-DG.

In the simplest version, the sketched architecture now is complete. One of ExaHyPE's key ideas however is to use tailored, extremely optimised code snippets whenever it comes to the evaluation of fluxes, eigenvalues, space-time predictors, and so forth. If the user decides to run for these optimised variant in the specification file, the toolkit skips the creation some application-specific emtpy templates that are to be befilled. Instead, it creates domain- and architecture-specific code fragments (*kernels*) that are now invoked whenever we run into computationally demanding program phases.

These generated fragments are typically not to be edited and are an essential part of ExaHyPE (and therefore white) and of ExaHyPE's identity. While the ExaHyPE core relies on Peano, the generated kernels use Intel's libxsmm which is the second 3rd party building block in the basic ExaHyPE architecture.

Obviously, our architecture as described so far still is simplistic and rudimentary; too rudimentary for most real-world applications. We hence expect most applications to extend and connect it to futher software fragments. An example is given in Figure 14.2. However, such links to other products are not part of the core ExaHyPE architecture and concept, as we try to keep it as simple and free of dependencies as possible.

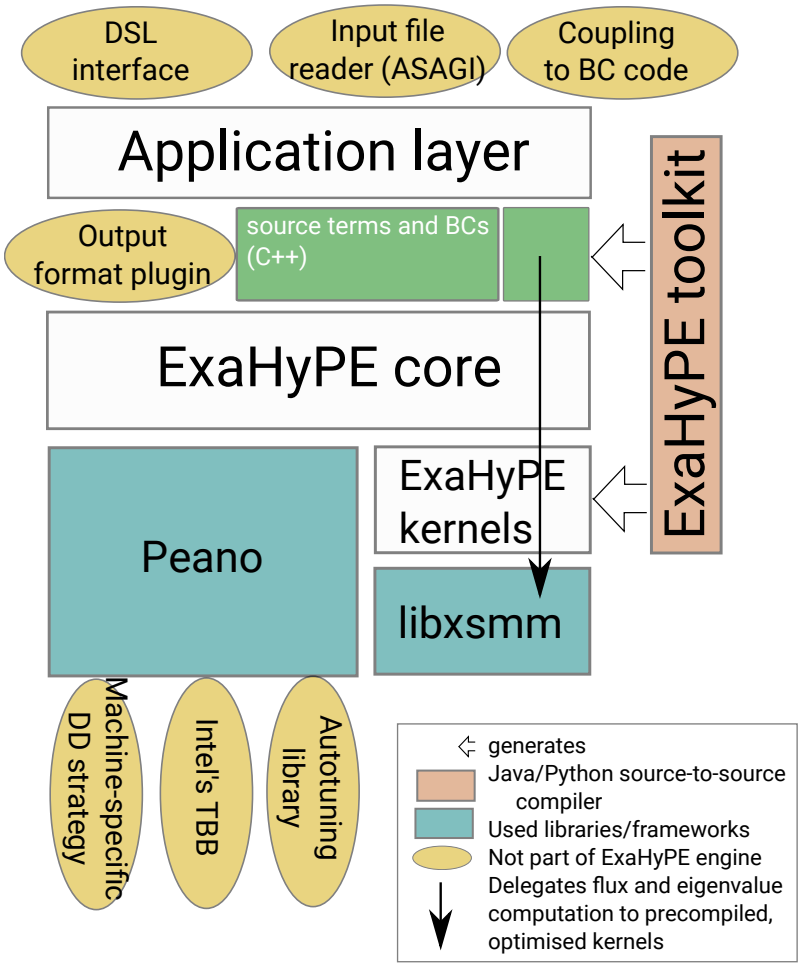



Figure 14.2.: Any ExaHyPE application tailors/uses/adopts the generic architecture to its needs. The present snapshot shows a sophisticated seismology ExaHyPE code that couples the simulation engine to 3rd party software and provides a domain specific language (DSL) in the top layer, delegates all flux and eigenvalue computations to predefined, optimised kernels generated by the toolkit and using libxsmm, and internally relies on Intel's TBBs, Peano's autotuning and Peano's generic domain decomposition to speed the code up.

# 15. **ExaHyPE** workflow

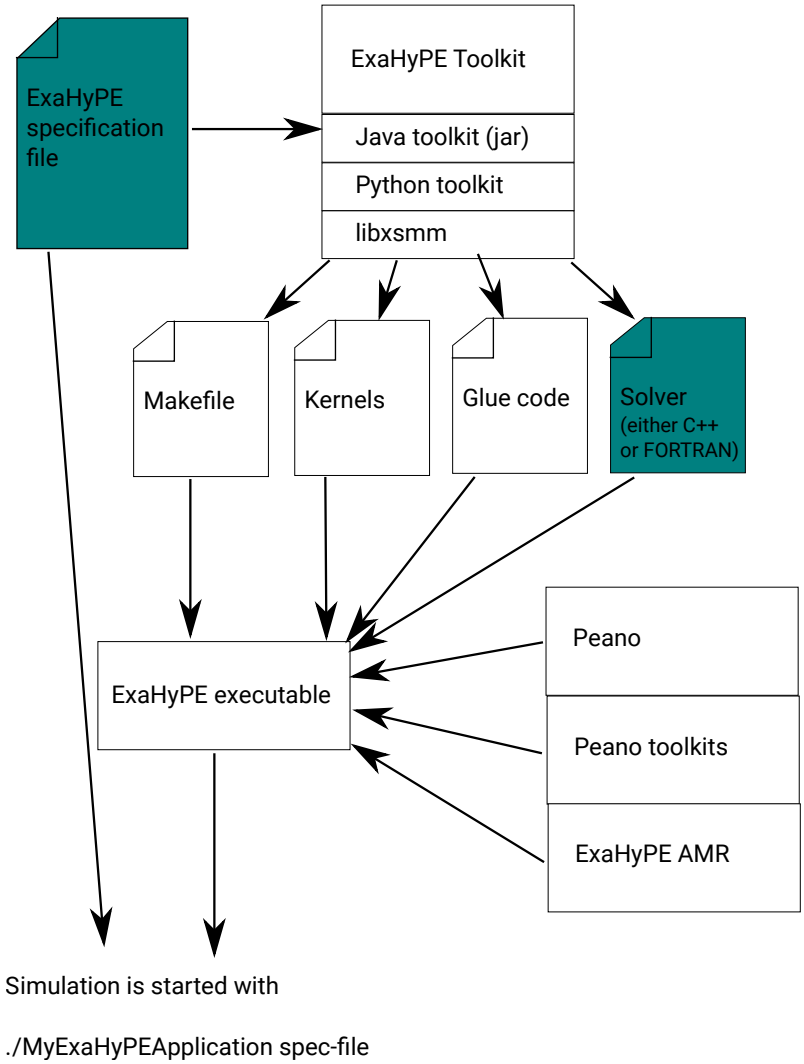



Figure 15.1.: A typical ExaHyPE workflow.

The typical workflow when working with the ExaHyPE engine is graphically displayed in Figure 15.1 and described in the following:

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 specified 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 in Figure 14.1 are the only files you typically have to change yourself. All the remainder is generated and assembled according to the specification file.

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.

# Part VI.

# Appendix

# 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 (re-)build the toolbox, change into the source directory of the toolkit and type in

```
make all
```

Alternatively, you can invoke the makefile with `make 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
```

In most cases, we however found it more convenient to recompile the toolkit.

# B. Frequently asked developer questions

- **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.

- **I get tons of warnings in the MPI part of the code that irritate me.**
  Please add the statement `-Wno-deprecated-declarations` to the compile arguments in your makefile.

- **How do I deal with vectors $\vec{v}$ in the code?** Peano, the underlying grid code for ExaHyPE, contains a small linear algebra library called `tarch::la`. It is used inside the ExaHyPE core all the place and also visible to the user-defined kernels (chapter 6). You may want to dig into the Peano Doygen

documentation to learn about the vector classes offered. You can, however, also use the method `tarch::la::Vector::data()` to access the raw C array, ie. to obtain a `data*` pointer. This allows you to easily connect any C++ vector library, ie. LAPACK, Eigen, Boost, GLM, Armadillo or you own one to your kernels. It also allows you to freely use Fortran functions in your kernels. This is frequently used in the project where tensorial computations are done.

- **To be continued ...**

# C. Frequently asked user questions

- **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 spatial resolution $\Delta x$?**
  ExaHyPE asks you to specify the `maximum-mesh-size` which determines the coarsest resolution in your grid. The number of cells in each direction on a unigrid layout is always $3^i$ where $i$ is the depth. That is, the possible values are given by table C.1 with depth $d$, number of cells in one spatial direction $N = 3^d$, real grid spacing $\Delta x = 3^{-d}$ and `maximum-mesh-size` $\Delta x_{\text{max}} > \Delta x$. It may help to quickly choose the right value for $\Delta x_{\text{max}}$.

| $d$ | $N$ | $\Delta x$ | $\Delta x_{\text{max}}$ |
|---|---|---|---|
| 1 | 3 | 0.333 | 0.50 |
| 2 | 9 | 0.111 | 0.20 |
| 3 | 27 | 0.037 | 0.10 |
| 4 | 81 | 0.012 | 0.02 |
| 5 | 243 | 0.004 | 0.01 |
| ... | | | |

Table C.1.: Possible unigrid layouts with ExaHyPE.

- **How do I alter the time step size?**
  ExaHyPE implements two time stepping schemes (global time stepping and globally fixed time stepping). Per solver, you can choose a strategy. All schemes run a trial time step in the first iteration to evaluate the CFL condition and determine a reasonable time step size. All schemes besides `globalfixed` then continue to study the CFL constraints and adopt the time step size automatically. Only `globalfixed` fixes the time step size for the whole simulation period. All schemes use a security factor on the CFL condition, i.e. the admissible time step size is scaled before it is applied. This scaling is set by `fuse-algorithmic-steps-factor` in the `optimisation` block. Its value has to be small or equal to one. Besides this security factor, nothing has to be done by the user.

- **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 6) from scratch.

- **Why do the time stamps in the plot output differ from the ones I have set in the specification file?**
  Plotting at specific times would require ExaHyPE to perform shorter time steps than prescribed by the CFL condition. It is well-known that this leads to long-term pollution of the numerical solution.

  We thus plot as soon as the solver time (minimum over all cells) is greater or equal to the next plotting time.

- **ExaHyPE did not create all the plots I requested. Why?** It's very likely that more than one of the plotting times you specified did fall into a single time step interval of the corresponding solver.

  In this case, see bullet point **Why do the time stamps in the plot output differ from the ones I have set in the specification file?**.

- **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.

- **To be continued ...**

# D. Solver Kernels

Table D.1 shows the kernel variants 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 6. 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. |

Table D.1.: ADER-DG solver types

Table D.2 displays the kernel variants offered for the the `Finite-Volume` solver type.

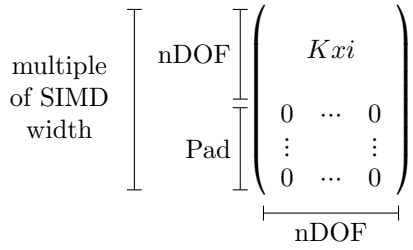
| Kernel | Effect |
|---|---|
| user::defined | Only function signatures are generated. All responsibility which and how kernels are realised is up to the user. We basically only hand over large double arrays. This is the lowest level of abstraction. |
| t.b.d. | |

Table D.2.: Finite-Volume solver types

# 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.

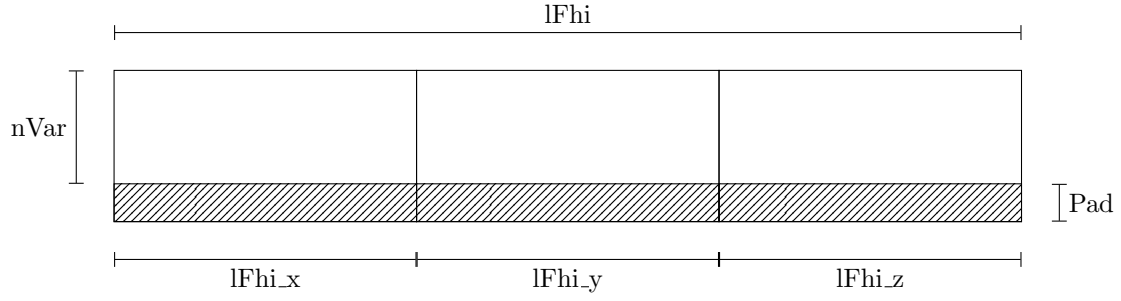
Table E.2 illustrates the memory layout for a 2D and 3D setup, respectively. In any case, the weights vectors are interpreted as linear, padded arrays. The pad width depends on the microarchitecture and ensures that the total length of the vectors is a multiple of the SIMD width. In the 2D case, the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

# F. Remarks on output file formats

## F.1. VTK

Our VTK support sticks to the VTK legacy format. This section is not supposed to replace the VTK file format description as it can be found in The VTK User's Guide available from Kitware free of charge. An excerpt of the relevant pages can be found at http://www.vtk.org/wp-content/uploads/2015/04/file-formats.pdf, e.g. This section complements the technical aspects with a user's point of view.

- **File format.** As VTK has no real support for block-structured adaptive meshes, ExaHyPE writes out all data as an unstructured mesh: the output file describes a graph. The graph is specified via three sections.
- **Section 1: Point coordinates.** In the first section of the VTK file, all points used by the mesh are specified. There is no particular on these points. Please see the introductory remarks in Chapter 12 on DG polynomials that clarify why some points can be found redundantly in the file. The only way the order plays a role is through the fact that it imposes an enumeration of the vertices starting with 0.
- **Section 2: Cells.** In a second part of the file, we specify all cells. Again, there is no particular order but the sequence in which the cells are enlisted defines an order. Each line in the file specifies on cell. VTK requires each cell definition to start with the number of vertices used to span the cell. As ExaHyPE restricts to block-Cartesian grids, this initial number always is either 4 or 8.
- **Section 3: Cell types.** The VTK file format requires a block where one enlists the cell type per cell entry. As we stick to cubes/squares in ExaHyPE, this section contains the same identifier (number) per cell and can be skipped if you want to postprocess the data yourself.
- **Point time step data.** After the cell types, you find the time steps of the point data. Per vertex, we track at which time stamp its data is written if your plotter has to write data. Otherwise, the section is omitted. Per vertex, we write one scalar. For global time stepping, all values are equal. For adaptive time stepping, they might differ as the data gives the real time stamp of the data, not the simulation time when the data has been written.
- **Point data.** After the time stamp, you find per vertex one vector holding all the vertex data.

- **Cell time step data.** Analogous to vertices.
- **Cell data.** Analogous to vertices.

If you have chosen a VTK binary format, all the section identifiers are still written in plain text, i.e. you can continue to open the output files as text files. The real data, i.e. the coordinates, the indices spanning a cell, the unknowns the are written binary and this binary stream is embedded into the text file.

requires texts about the file format. A first attemp is made in `https://gitlab.lrz.de/exahype/ExaHyPE-Engine/wikis/fileformats` and shall be resumed here.

Here we follow the display of cell and point data as defined in the VTK file format.

### F.1.1. VTK cell data

Table F.1 describes the data present in a VTK file which was created with the `vtk::*::cells::*` stanza. A note on the ordering of the data rows: There is no obvious order one should assume. Instead, it is helpful to sort the data on own criteria after reading in.

| **Point ID** | **Points** | | | **Time** |
|---|---|---|---|---|
| | x | y | z | |
| 0 | 0.00 | 0.00 | 0.00 | 0.01 |
| 1 | 0.01 | 0.00 | 0.00 | 0.01 |
| 2 | 0.01 | 0.01 | 0.00 | 0.01 |
| … | | | | |

| **Cell ID** | **Cell Type** | **Q** | | |
|---|---|---|---|---|
| | | $Q_0$ | $Q_1$ | … |
| 0 | Pixel | 1.23 | 4.56 | … |
| 1 | Pixel | 3.41 | 0.07 | … |
| 2 | Pixel | 0.81 | 5.47 | … |
| … | | | | |

Table F.1.: The point and cell data tables in ExaHyPE's VTK cell data format

### F.1.2. VTK point data

Table F.2 describes the data present in a VTK file which was created with the `vtk::*::vertices::*` stanza. A note on the ordering of the data rows: There is no obvious order one should assume. Instead, it is helpful to sort the data on own criteria after reading in.

| Point ID | Points | | | Time | Q | | |
|---|---|---|---|---|---|---|---|
| | x | y | z | | $Q_0$ | $Q_1$ | … |
| 0 | 0.00 | 0.00 | 0.00 | 0.01 | 1.23 | 4.56 | … |
| 1 | 0.01 | 0.00 | 0.00 | 0.01 | 3.14 | 0.07 | … |
| 2 | 0.01 | 0.01 | 0.00 | 0.01 | 0.81 | 5.47 | … |
| … | | | | | | | |

| Cell ID | Cell Type |
|---|---|
| 0 | Pixel |
| 1 | Pixel |
| 2 | Pixel |
| … | |

Table F.2.: The point and cell data tables in ExaHyPE's VTK point data format