



ExaHyPE Guidebook

User manual



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

This particular document was generated on Friday 23rd June, 2017 at 14:17.

Copyright © 2017 The ExaHyPE consortium

PUBLISHED BY THE EXAHYPE DEVELOPERS

EXAHYPE.EU

Licensed under the Creative Commons Attribution-NonCommercial 3.0 Unported License (the “License”). You may not use this file except in compliance with the License. You may obtain a copy of the License at <http://creativecommons.org/licenses/by-nc/3.0>. Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an “AS IS” BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

First printing, October 2015



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 necessary 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 adhere to these methodological/strategic ingredients, but can freely tailor and adopt them, write new plugins or assemble them in various ways. This document details

- how to execute some “simple” demonstrators,
- 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 knowledge of the C programming language is advantageous.

Most examples in this guidebook will run 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., or the ExaHyPE mailing list exahype@lists.lrz.de.



Contents

I ExaHyPE installation and demonstrator applications

1	Setup and Installation	13
1.1	Dependencies and prerequisites	13
1.2	Obtaining ExaHyPE	14
1.3	Dry run of development tools	15
1.4	Retranslating the ExaHyPE development toolkit	16
1.5	Using a newer/other Peano version	16
2	Demonstrator applications	19
2.1	Minimalistic Finite Volumes for the Euler equations	20
2.1.1	Download	20
2.1.2	Preparation	20
2.1.3	Run	21
3	Source code documentation	23
4	Development and release process	25

II Developing new ExaHyPE applications

5	Preparing a new ExaHyPE development project	29
----------	--	-----------

6	ADER-DG and Finite Volume solvers with generic kernels	33
6.1	Establishing the data structures	34
6.2	Study the generated code	35
6.3	Working with the arrays	36
6.4	Setting up the experiment	37
6.5	Realising the fluxes	39
6.6	Supplementing boundary conditions	42
6.6.1	Integration of Exact Boundary Conditions	43
6.6.2	Outflow Boundary Conditions	44
6.7	Finite Volumes	44
6.8	Localised, specialised solver features	45
7	Shallow Water Equations with ADER-DG	49
7.1	Setting up the experiment	49
7.2	Writing out the data	50
7.3	The (non-bathymetric) fluxes	51
7.4	Injecting the bathymetry	52
7.5	Transcribing the algorithm into Finite Volumes	54
8	ADER-DG with Finite Volume limiting	55
8.1	Effects of the limiter	57
9	Adaptive mesh-refinement	59
9.1	Prerequisites	59
9.2	Static mesh adaptivity: A geometry-based refinement criterion	60
9.3	Dynamic adaptive mesh refinement: A density-based refinement criterion	61
9.4	Limiter-based static adaptive mesh refinement	61
10	Some advanced solver features	63
10.1	Multiple solvers in one specification file	63
10.2	Runtime constants/configuration parameters	63
10.3	The init function	64
10.4	Time stepping strategies	64
10.5	Material parameters	65
10.6	Alternative symbolic naming schemes	66
10.7	Non-conservative formulations	66
10.8	Point sources	67
10.9	Custom Riemann Solvers	67
11	ExaHyPE FORTRAN	69
11.1	An example FORTRAN binding	69

11.2	The Fortran code	71
11.3	Known limitations	71
11.4	Further reading	71

12	Non-trivial computational domains	73
12.1	Real and computational domains	73

III

Upscaling and tuning ExaHyPE

13	Shared memory parallelisation	77
13.1	Tailoring the shared memory configuration	78
13.1.1	Autotuning	79
13.1.2	Configuration sampling and manual tuning	80
13.2	Hybrid parallelisation	80
14	Distributed memory parallelisation	83
14.1	An hitchhiker's guide through MPI	83
14.2	Buffer and timeout settings	85
14.3	identifier and configure settings	85
14.4	Hybrid parallelisation	85
14.5	MPI grid modifications	86
14.6	MPI troubleshooting and inefficiency patterns	87
15	Optimisation	89
15.1	High-level tuning	89
15.2	Optimised Kernels	91

IV

Working with ExaHyPE's output

16	ExaHyPE program output	95
16.1	Logging	95
16.2	Grid statistics	96
17	Plotting	97
17.1	Cell and subcell structure in the ExaHyPE output	97
17.2	Overview of supported data output formats	98
17.2.1	Volumetric plotters with generic file formats (VTK)	99
17.2.2	Tailored volumetric output formats	102
17.2.3	Non-volumetric plotters	103
17.3	Filtering	104
17.4	Parameter selection and conversion	104
17.5	User-defined plotters: Understanding the Plotting API	105

18	Postprocessing	107
18.1	On-the-fly computation of global metrics such as integrals	107
18.2	Reduction of global quantities over all MPI ranks	109

V ExaHyPE design & rationale

19	ExaHyPE architecture	115
19.1	A solver engine	116
19.2	A user's perspective	118
20	ExaHyPE workflow	119
20.1	Toolkit usage	119
20.2	ExaHyPEs build system	120
20.3	Build management	121
20.3.1	The Exa toolbox	121
20.3.2	Out-of-Tree compilation and code generation	121
20.4	Simulation management	122
20.4.1	Templated specification files	122

VI Appendix

A	The ExaHyPE toolbox	125
A.1	Rebuilding the toolbox	125
A.2	Troubleshooting	125
B	Frequently asked developer questions	127
C	Frequently asked user questions	129
D	Bugs, limitations and compilation problems	131
E	Solver Kernels	135
F	Datastructures in the Optimised Kernels	137
G	ExaHyPE Output File format specifications	141
G.1	ExaHyPEs VTK Unstructured Mesh output	141
G.1.1	VTK cell data	142
G.1.2	VTK point data	142
G.2	Peano block regular fileformat	142
G.3	The CarpetHDF5 output format in ExaHyPE	143
G.3.1	CarpethHDF5 conversion postprocessing conversion utilities	143
G.3.2	Postprocessing and Visualization tools understanding CarpetHDF5	143

G.4	The FlashHDF5 output format in ExaHyPE	144
H	Running ExaHyPE on some supercomputers	145
H.1	Hamilton (Durham's local supercomputer)	145
H.2	SuperMUC (Munich's petascale machine)	146
H.3	Tornado KNL (RSC group prototype)	147
H.4	RWTH Aachen Cluster	147



ExaHyPE installation and demonstrator applications

1	Setup and Installation	13
1.1	Dependencies and prerequisites	
1.2	Obtaining ExaHyPE	
1.3	Dry run of development tools	
1.4	Retranslating the ExaHyPE development toolkit	
1.5	Using a newer/other Peano version	
2	Demonstrator applications	19
2.1	Minimalistic Finite Volumes for the Euler equations	
3	Source code documentation	23
4	Development and release process ...	25



1. Setup and Installation

1.1 Dependencies and prerequisites

ExaHyPE comes along with the following dependencies:

- ExaHyPE source code is C++ code. For sequential simulations, only a C++ compiler is required. All examples from this guidebook run and have been tested with newish GCC and Intel compilers. The code uses few C++14 features, but for many older versions enabling those features through `-std=c++0x` made the code pass. There are no further dependencies or libraries required.
- You may prefer to write parts of your ExaHyPE code in Fortran. In this case, you obviously need a Fortran besides a C++ compiler. ExaHyPE itself does not depend on 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 (as provided with MPI 1.3, e.g.).
- ExaHyPE's development environment relies on Java (Java Runtime Environment JRE). We provide some standalone JAR files typically built against Java 1.6 or newer. You can however always build your own version of all development tools (recommended) as long as you have a Java compiler (Java Development Kit JDK).
- 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. Members of the ExaHyPE consortium and other users successfully use Windows and Mac systems. ExaHyPE however focuses not on these platforms and thus no typical pitfalls are discussed here. The code itself is minimalistic, i.e. in ExaHyPE's basic form no further libraries are required.

¹Libxsmm 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 below². 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 page³. 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 peano-framework.org and follow the descriptions there. Afterwards, add two symbolic links from ExaHyPE's Peano directory to your Peano installation⁴:

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

³<https://github.com/exahype/exahype>

⁴If you grab the Peano repository, the sources are held in a subdirectory `src` as the checkout also comes along with many other things such as development tools or documentation. If you download the tarball from the webpage, it usually holds only Peano's sources. In this case, you might have to omit the `src` subdirectory in the symbolic links.

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

We do trigger that Peano is updated regularly and ExaHyPE's build environment does validate your Peano version against its on requirements. Yet, as Peano is an independent package with a different release model/schedule, we cannot ensure that a tarball of Peano obtained from SourceForge or peano-framework.org is fully compatible with the current ExaHyPE release. If you encounter problems, we therefore recommend that you use the snapshot of Peano provided in `ExaHyPE.tar.gz`. It is guaranteed to be compatible with the ExaHyPE.

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. The consortium can not provide jar files for various Java versions as the one you might find on your system. Instead of messing around with Java compatibilities, we 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 -xzf ExaHyPE-toolkit.tar.gz

```

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 to 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 flux and can, with only a few lines, be changed into an ADER-DG scheme later. Indeed, it can be used by an ADER-DG scheme 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 EulerFV

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 vtu::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 2.1 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-EulerFV 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 Paraview¹ or VisIt², e.g. There are two quantities 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 energy. The plot should look similar to Figure 2.1.

Please note that this minimalistic ExaHyPE application will yield some errors/warnings that no exahype.log-filter file has been found and thus no filter is applied on the output messages. You will obtain tons of messages. The tailoring of the log outputs towards your needs is subject of discussion in Chapter 16. For the present experiment, you should be fine without diving into these details.

¹www.paraview.org

²wci.llnl.gov/simulation/computer-codes/visit



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.eu¹ and provides, besides hyperlinks and sketches of algorithmic concepts, a search engine.

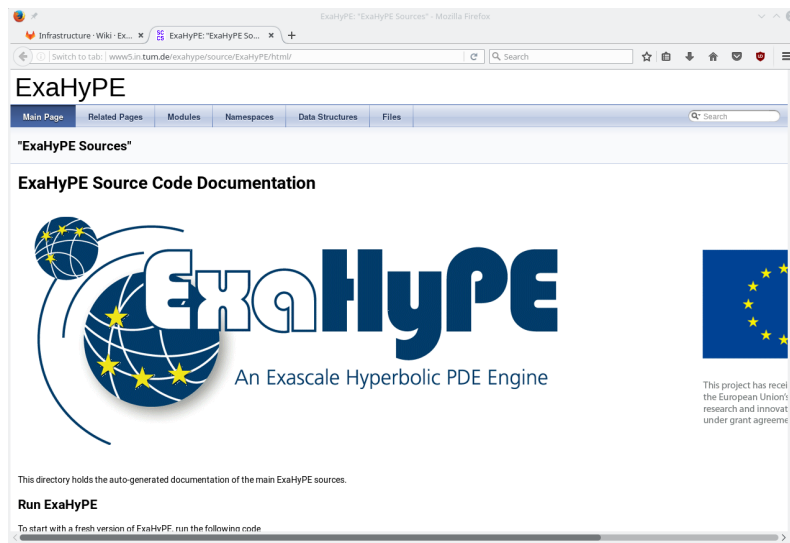


Figure 3.1: The ExaHyPE source code documentation webpage that is updated every night.

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

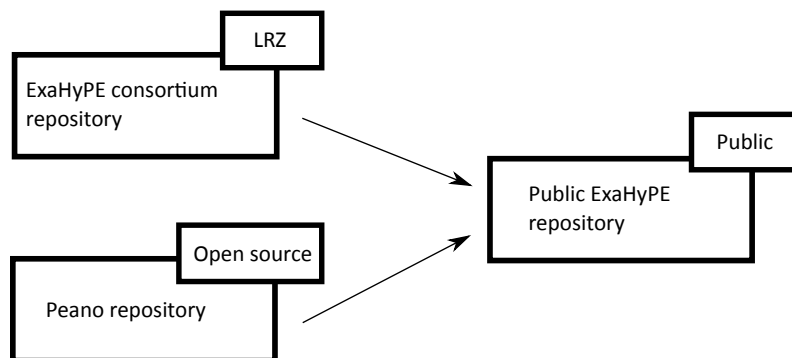
```
doxygen exahype.doxygen-configuration
```

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`.

¹Should 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.

Developing new ExaHyPE applications

5	Preparing a new ExaHyPE development project	29
6	ADER-DG and Finite Volume solvers with generic kernels	33
6.1	Establishing the data structures	
6.2	Study the generated code	
6.3	Working with the arrays	
6.4	Setting up the experiment	
6.5	Realising the fluxes	
6.6	Supplementing boundary conditions	
6.7	Finite Volumes	
6.8	Localised, specialised solver features	
7	Shallow Water Equations with ADER-DG ..	49
7.1	Setting up the experiment	
7.2	Writing out the data	
7.3	The (non-bathymetric) fluxes	
7.4	Injecting the bathymetry	
7.5	Transcribing the algorithm into Finite Volumes	
8	ADER-DG with Finite Volume limiting ..	55
8.1	Effects of the limiter	
9	Adaptive mesh-refinement	59
9.1	Prerequisites	
9.2	Static mesh adaptivity: A geometry-based refinement criterion	
9.3	Dynamic adaptive mesh refinement: A density-based refinement criterion	
9.4	Limiter-based static adaptive mesh refinement	
10	Some advanced solver features	63
10.1	Multiple solvers in one specification file	
10.2	Runtime constants/configuration parameters	
10.3	The init function	
10.4	Time stepping strategies	
10.5	Material parameters	
10.6	Alternative symbolic naming schemes	
10.7	Non-conservative formulations	
10.8	Point sources	
10.9	Custom Riemann Solvers	
11	ExaHyPE FORTRAN	69
11.1	An example FORTRAN binding	
11.2	The Fortran code	
11.3	Known limitations	
11.4	Further reading	
12	Non-trivial computational domains ...	73
12.1	Real and computational domains	



5. Preparing a new ExaHyPE development project

Design philosophy 5.1 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 reproducibility. 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 file¹ 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 5.2 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.

¹Some 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 paths². 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 before make:

```
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:

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.

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




6. ADER-DG and Finite Volume solvers with generic

In our previous 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 particularly well-suited for rapid prototyping of new solvers.

Design philosophy 6.1 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 equips 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

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

on a domain Ω supplemented by initial values $\mathbf{Q}(0) = \mathbf{Q}_0$ and appropriate boundary conditions. ρ 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 γ , the fluid pressure is defined as

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

Our example restricts to

$$\Omega = (0, 1)^d$$

as complicated domains are subject of discussion in a separate Chapter 12. 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 spec. file 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
```

Design philosophy 6.2 The formulation (6.1) lacks a dependency on the spatial position x . As such, it seems that ExaHyPE does not support spatially varying fluxes/equations. This impression is wrong. Our design philosophy is that spatial variations actually are material parameters, i.e., we recommend that you introduce material parameters for your spatial positions x or quantities derived from there. The Section 10.5 details the usage of material parameters.

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:

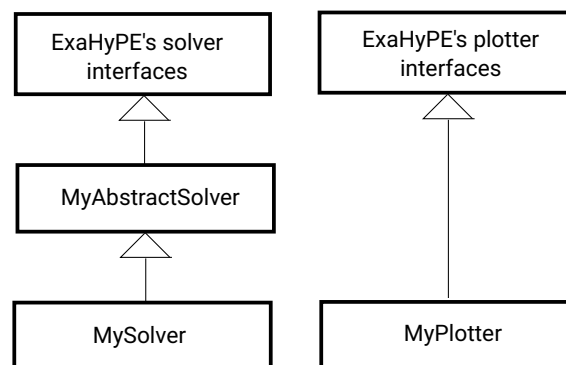


Figure 6.1: Simplest class architecture for ExaHyPE solvers.

- `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 6.3 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. Note that symbolic identifiers may degrade performances.

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 in the C/C++ language and thus start to count with 0). All fluxes are two-dimensional double arrays, i.e. are represented as matrices (that is, they are not guaranteed to be continous storage).

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 unknowns 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 syntactic 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 classes stem from 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, but be aware that it be less efficient than a direct implementation with arrays. The cooking down to a plain array-based implementation is straightforward.

Design philosophy 6.4 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 wrap coordinates, e.g., with `tarch::la::Vector` classes to have Matlab-style syntax.

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 follows¹:

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

```

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” up to machine precision.

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).

Whenever your `hasToAdjustSolution` returns `true`, the `adjustedSolutionValues` is called. Thus you don’t need further checks on conditions there and the example here is slightly overengineered.

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).

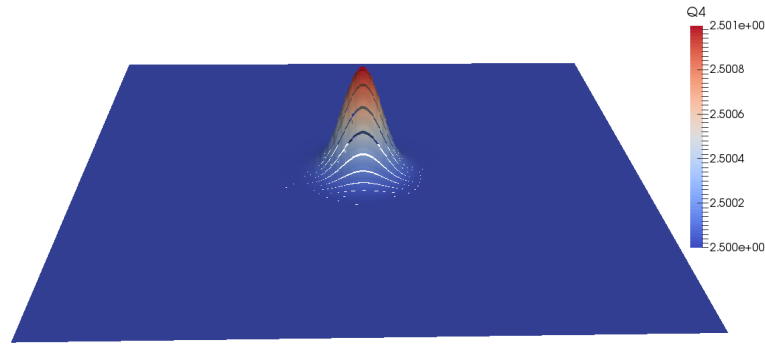


Figure 6.2: The rest mass density Q_0 in the EulerFlow 2D setup. The scalar value is encoded both in height and color. So far, the PDE has not been specified so it stays constant during the time evolution.

Design philosophy 6.5 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 actually implement the Euler equations, we have to realise the fluxes into the code. We do so by filling 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.

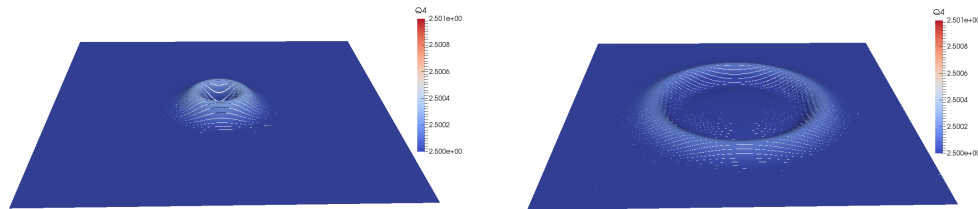


Figure 6.3: Time evolution of Q_0 , now with the Hydrodynamics PDEs implemented. The left figure shows an early time while the right figure shows a later time.

6.6 Supplementing boundary conditions

ExaHyPE offers a generic API to implement any kind of *local* boundary conditions. Local here refers to element-wise boundary conditions where the solver in one cell does not have to communicate with other cells. That is, you can straightforwardly² implement

- outflow boundary conditions (also sometimes referred 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)
{
    ...
}
```

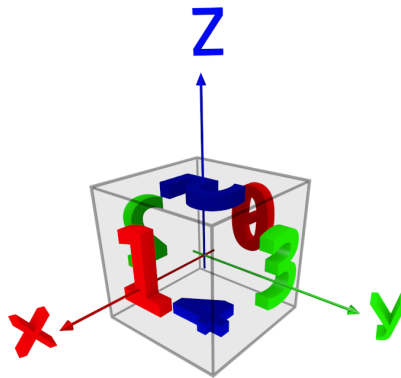


Figure 6.4: The face indexes as named in the ExaHyPE code

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
```

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

This is also encoded in Figure 6.4.

There is a big difference between Finite Volume boundary conditions as typically found in literature and our ADER-DG boundary conditions. For a DG method in general it is not sufficient to just prescribe the `stateOut`, i.e. the values, along the boundary that are just outside of the domain. We further need to prescribe the normal fluxes (`fluxOut`; Figure 6.5); unless if use lowest order DG, i.e. Finite Volumes. For Finite Volumes, the fluxes are not required as they directly result from the values “outside” of the domain.

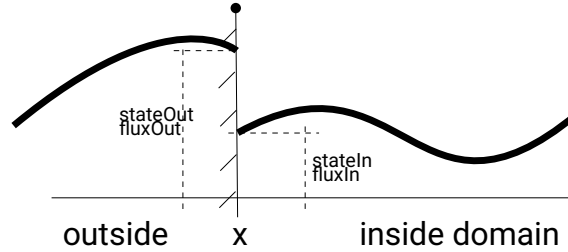


Figure 6.5: 1-dimensional sketch of boundary conditions in ExaHyPE’s ADER-DG. The state inside the domain and the fluxes from this side are passed to the boundary treatment by the solver. They are therefore const. A boundary condition is imposed through prescribing the state outside of the domain (thus describing the solution’s jump) and the associated flux.

Furthermore, ADER-DG methods require us to prescribe the state just outside of the domain (`stateOut`) and the corresponding fluxes (`fluxOut`) at all the temporal interpolation points. Within each cell, ADER-DG relies on a local space-time formulation: it derives all operators from integrals $\int_t^{t+\Delta t} \dots dt$. If you use a type of global time stepping where all cells march with exactly the same time step size, then the integral formulation translates into an arithmetic exercise. At the end of the day, the integrals over all state and flux variables along the face over the time span $(t, t + \Delta t)$ are required to set proper boundary conditions.

The story is slightly different if you use some local time stepping. In this case, ExaHyPE has to know the boundary conditions over time. It has to know the polynomial description of the state and flux solution over the whole time span.

6.6.1 Integration of Exact Boundary Conditions

We supply here a small example how to correctly integrate imposed Boundary Conditions in the ADER-DG scheme.

```
#include "kernels/GaussLegendreQuadrature.h"

void DemoADERDG::DemoSolver::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) {

    // Defining your constants
    const int nVar = DemoADERDG::AbstractSolver::NumberOfVariables;
    const int order = DemoADERDG::AbstractSolver::Order;
    const int basisSize = order + 1;
    const int nDim = DIMENSIONS;

    double Qgp[nVar];
```

```

std::memset(stateOut, 0, nVar *sizeof(double));
std::memset(fluxOut, 0, nVar *sizeof(double));

double F[nDim][nVar];

// Integrate solution in gauss points (Qgp) in time
for(int i=0; i < basisSize; i++) { // i == time
    const double weight = kernels::gaussLegendreWeights[order][i];
    const double xi = kernels::gaussLegendreNodes[order][i];
    double ti = t + xi *dt;

    setYourExactData(x, Qgp, &ti);
    flux(Qgp, F); // calling your Solvers flux function
    for(int m=0; m < nVar; m++) {
        stateOut[m] += weight *Qgp[m];
        fluxOut[m] += weight *F[normalNonZero][m];
    }
}
}

```

This code snippet assumes you have a function `setYourExactData` which give exact boundary conditions for a point `x` at time `ti`.

6.6.2 Outflow Boundary Conditions

As another example, outflow boundary conditions are achieved by just copying the fluxes and states.

```

void DemoADERDG::DemoSolver::boundaryValues(...) {
    for(int i=0; i < DemoADERDG::AbstractSolver::NumberOfVariables; i++) {
        fluxOut[i] = fluxIn[i];
        stateOut[i] = stateIn[i];
    }
}

```

Design philosophy 6.6 We stick to $\Omega = (0,1)^d$ without MPI here. Please read carefully through Chapter 12 if you need more sophisticated boundary conditions or if precise boundary conditions form an essential part of your simulation.

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 6.7 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. There are only subtle differences to take into account:

- If you (as discussed later in this document) fuse the ADER-DG solver with a Finite Volumes solver, patch-size typically should be chosen as $2 \cdot \text{order} + 1$. This ensures that the admissible time step sizes of the DG scheme matches the time step sizes of the Finite Volumes formulation.
- Boundary treatment of Finite Volume solvers simpler than for Finite Volumes. It only requires us to prescribe the state variables (unknowns) outside of the domain. All fluxes then are determined from hereon. As a consequence, the `boundaryValues` routine is a briefer signature. While you can share flux/eigenvalue computations, e.g., between an ADER-DG and FV solver, the boundary treatment has to be realised independently.

6.8 Localised, specialised solver features

Many features of an ExaHyPE application are not enabled by default:

- Source terms within the PDE.
- Admissibility checks on the solution (such as positivity checks) plus their a posteriori processing.
- Non-conservative terms required to handle material parameters, e.g.
- ...

In general, ExaHyPE solves equations of the form

$$\frac{\partial}{\partial t} \mathbf{Q} + \nabla \cdot \mathbf{F}(\mathbf{Q}) + \underbrace{\sum_i \mathcal{B}_i \frac{\partial \mathbf{Q}}{\partial x_i}}_{\text{useNonConservativeProduct}} = \underbrace{\mathbf{S}}_{\text{useAlgebraicSource}} + \underbrace{\sum \delta}_{\text{usePointSource}}, \quad (6.3)$$

and it allows the user to overwrite any solution (locally) and to invalidate any solution (locally) after each time step. As most users do not require all terms of the equation, ExaHyPE allows you to switch on/off certain terms. This also allows the engine to optimise more aggressively. In the generic formulation above, all terms with an underbrace are optional.

Design philosophy 6.8 ExaHyPE disables many terms of (6.3) by default. They are realised as additional function calls per cell and the engine tries to minimise such calls wherever possible. As a result, all optional function calls come along as a pair of operations: a `use...` operation tells the engine on a *per-cell per time step* base whether a feature is to be switched on or off. Only if the predicate holds, the corresponding functions realising the actual behaviour are invoked.

This design pattern can be studied for all solvers through `adjustSolution`. It allows us to prescribe values of the solution manually (read as an a posteriori adjustment of the simulation values) and, thus, notably allows us to insert initial conditions. This feature is not always enabled. Instead, we can, for initial conditions, switch it on only for time $t = 0$.

Table 6.1: The following per-cell routines of a solver are not evaluated for a cell unless the corresponding guard returns true.

Feature	Guard	Semantics
<code>adjustSolution</code>	<code>useAdjustSolution</code>	The adjustment allows you to alter the solution, i.e. overwrite it, manually after it has been updated. Used by all solvers to set boundary conditions, e.g. In this case, <code>useAdjustSolution</code> has to hold for time zero. However, you can manually set values at any time if you want. You can decide via the return value of the guard function whether you want to adjust the solution not at all, point-wisely or you prefer to be handed over the whole space polynomial.
<code>pointSource</code>	<code>usePointSource</code>	Add a point source to your solution.
<code>algebraicSource</code>	<code>useAlgebraicSource</code>	Add an algebraic source to your solution.
<code>nonConservativePr.</code> and <code>coefficientMatrix</code>	<code>useNonConservativePr.</code>	Work with a non-conservative product term in your equation. If you switch on this feature via the guard, you have to implement both the product and the matrix operation
<code>isPhysicallyAdmissible</code>		This routine does not literally belong into the class of these routines—there is no guard, e.g.—but it allows you to tell the integration kernels that a solution is physically not admissible. It is then up to the kernel to follow up, i.e., to reduce the time step size or switch to a limiter.

A comprehensive list of features is summarised in Table 6.1. Having an opportunity to switch features on or off allows ExaHyPE’s kernels to offer optimised variants for particular feature combinations. Notably, it is thus important that you study which variants are supported by a particular optimisation if you use a non-generic kernel. The default generic kernels, e.g., assume that all products are conservative. If you want to have non-conservative terms, you have to choose a different kernel type and then make the corresponding predicate return `true`.

If you switch to Finite Volumes, not all of the optional features might be available. Please consult the generated code. Please note that the actual routines are not constant: users might want to couple to other codes in these routines, e.g., and so we can’t make them constant as we may not assume anything about other codes.

Design philosophy 6.9 ExaHyPE’s toolkit creates an abstract superclass per solver. In this abstract superclass, all optional routines are declared and they all are switched off. If you want to use a feature

1. Add both routine plus its corresponding guard (`use . . .`) to your solver’s header^a and mark them with `override`.
2. Implement them in your solver’s class.

This holds solely for the generic, non-optimised kernels.

^aPlease note that ExaHyPE's toolkit never modifies our solver's implementation or header file. It however defaults off of these functions (switches them off) in the abstract solver, so you can study their syntax there.

Two remarks:

- To the best of our knowledge, there's not a single solver in the world that does not at least plug into `adjustSolution` to set up initial values.
- Some routines such as the nonconservative product map one guard expression onto multiple semantical operations. There is one guard (useNonconservativeProduct). If you activate it, you have to implement both the scalar product and the actual matrix realisation³.
- Some routines allow the guard not only to switch a feature on or off but also allows the guard to decide which realisation variant is chosen. The adjust solution feature for example can be activated point-wisely or patch-wisely.

³ In the Riemann solvers (both for Finite Volumes and ADER-DG), we need the application of matrix \mathcal{B} along one direction. This contribution however is required multiple times—at least once per dimension. Contrary, ADER-DG's space-time predictor applies the matrix literally.



7. Shallow Water Equations with ADER-DG

In this chapter, we quickly detail how to write a Shallow Water solver with ADER-DG. It is very hands on, application-focused and brief, so it might be reasonable to study Chapter 6 first. Different to Chapter 6 we focus on a few particular challenges tied to the shallow water equations:

1. How we quickly write down a plotter to visualise the water height though we are given the bathymetry and water depth in the solver only.
2. How to handle bathymetry, i.e. material parameters, elegantly within ExaHyPE's ADER-DG framework.

Our shallow water equations are written down as

$$\frac{\partial}{\partial t} \begin{pmatrix} h \\ hu \\ hv \\ b \end{pmatrix} + \frac{\partial}{\partial x} \begin{pmatrix} hu \\ hu^2 + 0.5gh^2 \\ huv \\ 0 \end{pmatrix} + \frac{\partial}{\partial y} \begin{pmatrix} hv \\ huv \\ hv^2 + 0.5gh^2 \\ 0 \end{pmatrix} + \begin{pmatrix} 0 \\ hg \cdot b_x \\ hg \cdot b_y \\ 0 \end{pmatrix} = 0. \quad (7.1)$$

h denotes the height of the water column, (u, v) the horizontal flow velocity, g the gravity and b the bathymetry. The subscripts x, y symbolize partial differentiation. Following ExaHyPE's philosophy, distinguish our unknown vector (h, hu, hv) from the material parameter b , but we summarise them where appropriate as one vector or quantities of interest $\mathbf{Q} = (h, hu, hv, b)$.

In the first part of this discussion, we neglect the bathymetry. A separate subsection is dedicated to this material parameter. As the bathymetry does not directly have an impact on the solution—it is the gradient that has an impact—this means we assume a constant sub-ocean profile.

7.1 Setting up the experiment

The specification for this example is close to trivial

```
exahype-project SWE
peano-kernel-path const = ./Peano
```

```

exahype-path const = ./ExaHyPE
output-directory const = ./ApplicationExamples/SWE_ADERDG
architecture const = noarch
log-file = mylogfile.log

computational-domain
  dimension const = 2
  width = 10.0, 10.0
  offset = 0.0, 0.0
  end-time = 5.0
end computational-domain

solver ADER-DG MySWESolver
  variables const = h:1,hu:1,hv:1
  parameters const = b:1
  order const = 3
  maximum-mesh-size = 0.15
  time-stepping = global
  kernel const = generic::fluxes::nonlinear
  language const = C
end solver
end exahype-project

```

and we set some initial data as we adjust our solution manually and point-wisely if the simulation time equals $t = 0$:

```

bool SWE::MySWESolver::useAdjustSolution(...,const double t) const {
  return tarch::la::equals(t,0.0);
}

void SWE::MySWESolver::adjustSolution(const double* const x,...,
  const double t,...,double* Q) {
  assertion(tarch::la::equals(t, 0.0);
  initialData(x,Q);
}

```

We may add this `initialData` as function to `MySWESolver.h` or we may have it separately; it does not really matter. Before we continue, we open `MySWESolver.h` and ensure that

```
bool useAdjustSolution(...) const override;
```

is already marked there as `override`. By default, ExaHyPE does not adjust any solution and the `AbstractSWESolver` indeed will by default return `false`. We explicitly have to switch this “feature” on. This is a pattern that will arise later for the bathymetry again.

7.2 Writing out the data

While we could plot out \mathbf{Q} as it is, it is convenient, for the shallow water equations, to see the water height rather than the entries in \mathbf{Q} . We also want to get the bathymetry (though this is kind of an overhead as it does not change in time). The water height is the sum of the bathymetry plus h . We therefore make the plotter give us two more unknowns than we actually have in the specification file:

```

plot vtu::Cartesian::cells::ascii ConservedWriter
  variables const = 5
  time = 0.0
  repeat = 0.1
  output = ./conserved
  select = x:0.0,y:0.0
end plot

```

A rerun of the ExaHyPE toolkit yields a writer class, and we modify one function therein:

```

void SWE::ConservedWriter::mapQuantities(...) {
  for (int i=0; i<4; i++){
    outputQuantities[i] = Q[i];
  }
  outputQuantities[4] = Q[3] + Q[0];
}

```

This snippet uses two properties of ExaHyPE:

1. The material parameters in terms of data are just modelled as additional quantities attached to the unknown vector (this is one of the reasons why we have defined \mathbf{Q} as above). We exploit this factor here by accessing $Q[3]$ which otherwise would not make any sense.
2. We have told the toolkit that we want to write out five quantities, so it already prepares for us a reasonably large array `outputQuantities`.

7.3 The (non-bathymetric) fluxes

The realisation of the eigenvalues and the fluxes here is straightforward once we note that we basically just set all contributions from the bathymetry to zero:

```

void SWE::MySWESolver::eigenvalues(const double* const Q,
  const int normalNonZeroIndex, double* lambda) {
  // Dimensions = 2
  // Number of variables = 3 (#unknowns + #parameters)
  ReadOnlyVariables vars(Q);
  Variables eigs(lambda);

  const double c= std::sqrt(grav*vars.h());
  const double ih = 1./vars.h();

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

  eigs.h() = u_n + c ;
  eigs.hu()= u_n -c;
  eigs.hv()= u_n ;
}

void SWE::MySWESolver::flux(const double* const Q, double** F) {
  ReadOnlyVariables vars(Q);

  const double ih = 1./vars.h();

```

```

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

f[0] = vars.hu();
f[1] = vars.hu()*vars.hu()*ih + 0.5*grav*vars.h()*vars.h();
f[2] = vars.hu()*vars.hv()*ih;

g[0] = vars.hv();
g[1] = vars.hu()*vars.hv()*ih;
g[2] = vars.hv()*vars.hv()*ih + 0.5*grav*vars.h()*vars.h();
}

```

7.4 Injecting the bathymetry

So far, we did assume $\nabla b = 0$ and thus neglected any subocean variations. To get in the bathymetry, we follow the following ExaHyPE conventions:

- The bathymetry b is subject to the trivial PDE $\partial_t b = 0$.
- We can thus rewrite (7.1) as a PDE over four unknowns.
- Following exactly the notation from (7.1), the bathymetry gradient then enters the PDE as additional derivative entry. This follows the notion of path-conservative integration, but yields non-conservative terms. We thus have to switch on ExaHyPE's non-conservative formulation explicitly.

Design philosophy 7.1 Material parameters in ExaHyPE are simply appended to the unknowns. From a data structure point of view, it does not make a difference whether we write

```

variables const = h:1,hu:1,hv:1
parameters const = b:1

```

or

```

variables const = h:1,hu:1,hv:1,b:1

```

in the present application. ExaHyPE's implementation however ensures that parameters are never updated even though some Riemann schemes might come up with updates to the “parameter PDE” $\partial_t b = 0$. If you require time-dependent material parameters, we recommend to plug into `adjustSolution` to set the corresponding “solution” entries for particular time steps.

We formalise our approach by rewriting (7.1) into

$$\frac{\partial}{\partial t} \mathbf{Q} + \frac{\partial}{\partial x} \mathbf{F}(\mathbf{Q}) + \frac{\partial}{\partial y} \mathbf{G}(\mathbf{Q}) + B_1(Q) \frac{\partial Q}{\partial x} + B_2(Q) \frac{\partial Q}{\partial y} = 0 \quad (7.2)$$

where

$$\mathbf{Q} = \begin{pmatrix} h \\ hu \\ hv \\ b \end{pmatrix} \quad \text{and} \quad \mathbf{F} = \begin{pmatrix} hu \\ hu^2 + 0.5gh^2 \\ huv \\ 0 \end{pmatrix} \quad \text{and} \quad \mathbf{G} = \begin{pmatrix} hv \\ huv \\ hv^2 + 0.5gh^2 \\ 0 \end{pmatrix}.$$

h denotes the height of the water column, (u, v) the horizontal flow velocity, g the gravity and b the bathymetry. The subscripts x, y symbolize partial differentiation. By defining

$$\mathbf{B}_1 := \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & hg \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \quad \text{and} \quad \mathbf{B}_2 := \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & hg \\ 0 & 0 & 0 & 0 \end{pmatrix},$$

the bathymetry gradient is taken into account through the non-conservative products $\mathbf{B}_1 \frac{\partial \mathbf{Q}}{\partial x}$ and $\mathbf{B}_2 \frac{\partial \mathbf{Q}}{\partial y}$. This way of mapping the equations into ExaHyPE allows for a well-balanced numerical solver and automatic differentiation of b .

We realise non-conservative product in five steps:

1. We add a line

```
virtual bool useNonConservativeProduct() const { return true; }
```

By default, non-conservative products are disabled in ExaHyPE, so we have to enable them explicitly.

2. As we inform ExaHyPE that we want to use the non-conservative products, we have to clarify in the header that we will offer a realisation of those routines:

```
void nonConservativeProduct(const double* const Q,
    const double* const gradQ, double* BgradQ) override;

void coefficientMatrix(const double* const Q,
    const int d, double* Bn) override;
```

3. We realise the non conservative product:

```
void SWE::MySWEsolver::nonConservativeProduct(const double* const Q,
    const double* const gradQ, double* BgradQ) {
    idx2 idx_gradQ(DIMENSIONS, NumberOfVariables+NumberOfParameters);
    BgradQ[0]=0.0;
    BgradQ[1]=g*Q[0]*gradQ[idx_gradQ(0,3)];
    BgradQ[2]=g*Q[0]*gradQ[idx_gradQ(1,3)];
    BgradQ[3]=0.0;
}
```

In the above code snippet, we used the `idx2` struct in `kernels/KernelUtils.h` to access the derivatives.

4. The matrices \mathbf{B}_1 and \mathbf{B}_2 are finally parameterised over the normal direction and can be implemented as follows:

```
void SWE::MySWEsolver::coefficientMatrix(const double* const Q,
    const int d, double* Bn) {
    idx2 idx_Bn(NumberOfVariables+NumberOfParameters,
        NumberOfVariables+NumberOfParameters);

    Bn[0] = 0.0;
    Bn[1] = 0.0;
    ...
}
```

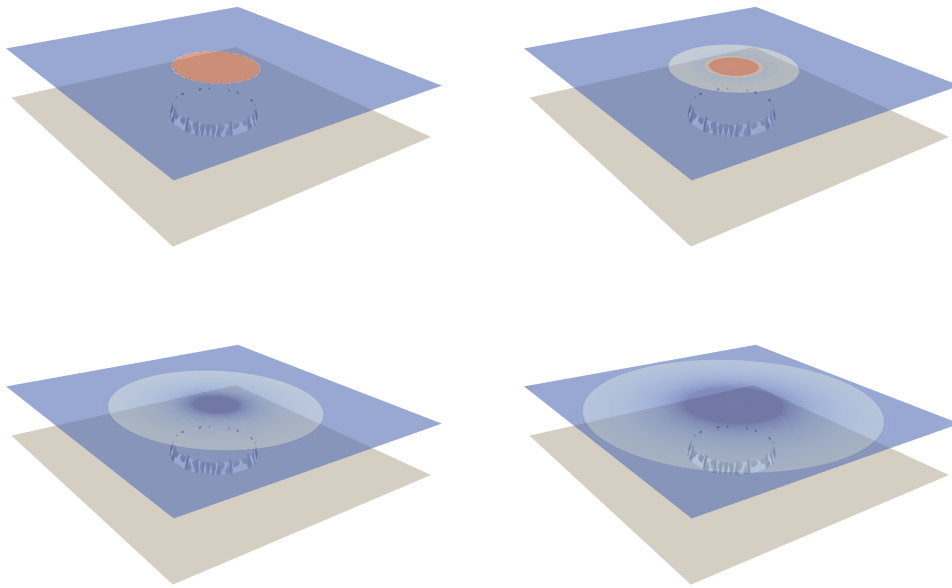
```

Bn[15]= 0.0;

Bn[idx_Bn(3,d+1)]=g*Q[0]; //g*h
}

```

It is important to note that the solver expects the matrix to be in Fortran storage order. To abstract from this fact, we use ExaHyPE's `idx_Bn` array. Alternatively, you can access the data directly. Again, the above snippet exploits that material parameter physically are treated as additional unknowns attached to the original unknown vector.



Left: A ocean with a plain subsurface hosting a circular plateau is covered by constant constant water height, i.e. the water mirrors the subocean topology. To the right: Once we “release” this initial condition, we see waves spreading.

7.5 Transcribing the algorithm into Finite Volumes

The finite volumes realisation of the shallow water equations follows the workflow sketched in this chapter. There are subtle differences:

- The non-conservative product is not required for our 1st order Godunov schemes. The corresponding routine has to be removed.
- In the routine `boundaryValues` are no fluxes, so we can eliminate any flux sets along the boundary.



8. ADER-DG with Finite Volume limiting

The ADER-DG scheme with Finite Volume limiting combines the high-order approximation properties of the ADER-DG method in areas with a sufficiently regular solution with robustness of the Finite Volumes method in areas where shocks and contact discontinuities dominate the solution.

In ExaHyPE, empty solver code for the ADER-DG method with Finite Volumes limiting can be generated by specifying a `LimitingADERDG` solver environment in the ExaHyPE specification file. This could e.g. look as follows:

```
[..]
solver Limiting-ADER-DG MySolver
  variables = rho:1,j:3,E:1
  primitives = 0
  parameters = 0
  order = 3
  maximum-mesh-size = 0.05
  time-stepping = global
  kernel = generic::fluxes::nonlinear
  language = C
  limiter-kernel = generic::Godunov
  limiter-language = C
  dmp-relaxation-parameter = 1e-4
  dmp-difference-scaling = 1e-3
[..]
end solver
[..]
```

Unlike to the pure ADER-DG and Finite Volumes method, we have to specify two kernels for the limiting ADER-DG solver – one for the ADER-DG method (`kernel`, `language`) and one for the Finite Volumes method employed for the limiting (`limiter-kernel`, `limiter-language`). Of course, we can choose any of the kernels that are available for the pure ADER-DG or pure Finite Volumes method.


```

}
return true;
}
[...]
```

8.1 Effects of the limiter

After each timestep, the limiter will examine all cells and mark the troubled cells and their neighbours. A cell is troubled if for any conserved quantity, its maximum is greater than the maximum of the cell or its neighbors before the steps, likewise for the minimum. The extrema are approximated by examining the quantities at the Gauss-Legendre and the Gauss-Lobatto ones to include the boundary values.

Additionally, a cell might be marked as troubled if the solution in the cell is not physically admissible.

If any cell was marked, a Finite Volume steps takes place on them using the saved values from before the ADER-DG steps or saved healthy values from a previous limiter intervention. Once the Finite Volumes solutions are computed, they are used to recompute and update a new ADER-DG solution on the marked cells. If the cells was troubled or the cell was a neighbors of a troubled cell and becomes troubled after this limiter step, the Finite Volume healthy value are saved for the next time to be used if required.

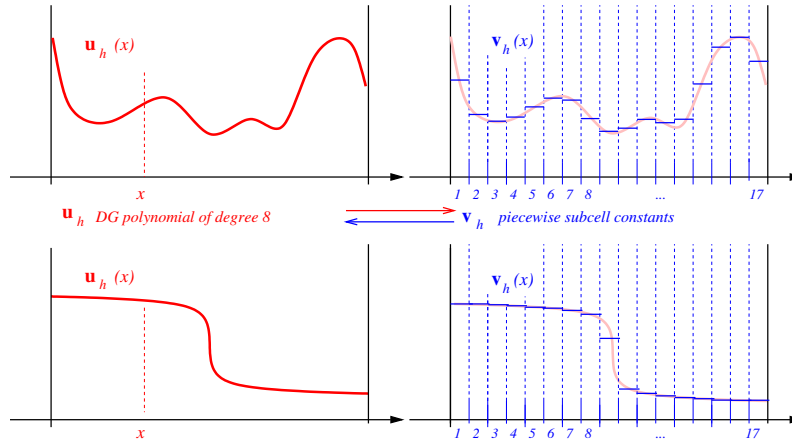


Figure 8.1: Projection to and recovery from the finite volume solver with $2N + 1$ subcells for an polynomial order N ADER-DG polynomial. The figure shows one cell with $N = 8$. Taken from Dumbser, Zanotti, Loubere, Diot 2015.



9. Adaptive mesh-refinement

In this chapter, we demonstrate the steps to perform for using ExaHyPE's adaptive mesh refinement. To this end, we consider an ADER-DG solver for the compressible Euler equations (Ch. 6). Note that ExaHyPE does currently only support adaptive mesh refinement for the ADER-DG solver and the limiting ADER-DG solver. We currently do not support adaptive mesh refinement for the FV solver.

9.1 Prerequisites

In the following sections, we assume that the reader has performed all steps in Ch. 6 and is able to run the example given there on a uniform mesh.

In order to enable adaptive mesh refinement up to a certain maximum depth, we have to add a parameter `maximum-mesh-depth` just below the mandatory parameter `maximum-mesh-size`:

```

exahype-project Euler2d

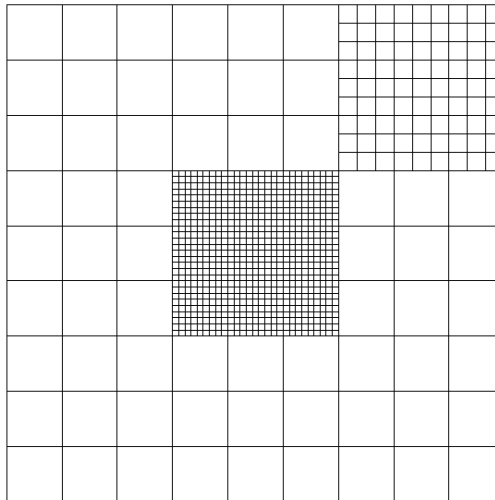
..
..

solver ADER-DG MyEulerSolver
  variables const = 5
  order const = 3
  maximum-mesh-size = 0.1
  maximum-mesh-depth = 2
  time-stepping = global
  kernel const = generic::fluxes::nonlinear
  language const = C
  ..
  ..
end solver
end exahype-project

```

9.2 Static mesh adaptivity: A geometry-based refinement criterion

Let us start with a simple geometry-based refinement criterion. In our example, we consider a uniform mesh of 9×9 cells on the unit cube (use e.g. `maximum-mesh-size=0.1`). We additionally want to perform one level of adaptive refinement in the region $[0.666, 1] \times [0.666, 1]$ and two levels of adaptive refinements in region $[0.333, 0.666] \times [0.333, 0.666]$:



To facilitate the desired adaptive mesh refinement, we augment the function `refinementCriterion(...)` in our `MyEulerSolver.cpp` file as given below: (No modification of the project's specification file is necessary.)

```

exahype::solvers::Solver::RefinementControl
Euler::MyEulerSolver::refinementCriterion(
  const double* luh, const tarch::la::Vector<DIMENSIONS, double>& center,
  const tarch::la::Vector<DIMENSIONS, double>& dx, double t,
  const int level) {
  if (level < getMaximumAdaptiveMeshLevel())

```

```

    if (center[0] > 0.666)
        if (center[1] > 0.666)
            return exahype::solvers::Solver::RefinementControl::Refine;

    if (level > getCoarsestMeshLevel())
        if (center[0] > 0.333 && center[0] < 0.666)
            if (center[1] > 0.333 && center[1] < 0.666)
                return exahype::solvers::Solver::RefinementControl::Refine;

    return exahype::solvers::Solver::RefinementControl::Keep;
}

```

Please be aware that we do not enforce a 2:1 balancing of the mesh in ExaHyPE. The generic kernels we provide are able to restrict and prolongate over an arbitrary number of levels.

If a 2:1 balance is important for your application, you have to enforce it on your own by carefully choosing the mesh refinement regions.

9.3 Dynamic adaptive mesh refinement: A density-based refinement criterion

ExaHyPE also allows you to dynamically refine and recoarsen the mesh. A simple refinement criterion that takes the density amplitude into account might look as follows:

```

exahype::solvers::Solver::RefinementControl
Euler::MyEulerSolver::refinementCriterion(
    const double* luh, const tarch::la::Vector<DIMENSIONS, double>& center,
    const tarch::la::Vector<DIMENSIONS, double>& dx, double t,
    const int level) {
    double largestRho = -std::numeric_limits<double>::max();

    kernels::idx3 idx_luh(Order+1, Order+1, NumberOfVariables);
    dfor(i, Order+1) {
        ReadOnlyVariables vars(luh + idx_luh(i(1), i(0), 0));

        largestRho = std::max (largestRho, vars.rho());
    }

    if (largestRho > 0.65) {
        return exahype::solvers::Solver::RefinementControl::Refine;
    }

    if (level > getCoarsestMeshLevel())
        return exahype::solvers::Solver::RefinementControl::Erase;

    return exahype::solvers::Solver::RefinementControl::Keep;
}

```

In order to use the above d -dimensional for loop, make sure to include the file `peano/utils/Loop.h`. The index function `kernels::idx3` is included via `kernels/KernelUtils.h`

9.4 Limiter-based static adaptive mesh refinement



10. Some advanced solver features

10.1 Multiple solvers in one specification file

ExaHyPE specification files can host multiple solvers. In this case, multiple solvers are simultaneously 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.

10.2 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 10.1 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.

10.3 The init function

The ExaHyPE toolkit equips every solver with a function `init` which is called immediately after the solver construction, i.e. at startup time. It is comparable to a `main` function in regular C programs.

The `init` method is the right place to do any startup preparation, for instance to prepare loading heavy initial data. There is no parallelization done when calling `main`, so it might want to make use of TBB and MPI to realize the preparation of initial data. Note that the actual initial data are fed by the `adjustedSolutionValues(x,Q)` function (cf. section 6.4). Users may want to extend their solver class to store prepared initial data, if necessary.

ExaHyPE also supports passing command line parameters down to every solver. This allows developers to quickly setup a workflow for passing extensive own parameter files or command line parameters. To do so, they can exploit the (exemplary) signature

```

void Euler::MyEulerSlover::init(std::vector<std::string>& cmdlineargs) {
    std::cout << "Command_line_options:\n";
    for(auto arg& : cmdlineargs) {
        std::cout << "-\n" << arg << "\n";
    }
}

```

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

global

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

globalfixed

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

10.5 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 unknowns: 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 10.2 ExaHyPE models material parameters as additional unknowns Q_p in the PDE that are subject to $\partial_t Q_p = 0$ on an analytical level (no dissipation).

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.

10.6 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`.

10.7 Non-conservative formulations

So far, we only considered how to implement strongly hyperbolic PDEs in ExaHyPE, ie. PDEs which can be casted to the form

$$\frac{\partial}{\partial t} \mathbf{Q} + \nabla \cdot \mathbf{F}(\mathbf{Q}) = S(\mathbf{Q}). \quad (10.1)$$

However, the generic kernels in ExaHyPE also support hyperbolic PDEs with non-conservative terms which can be written in a quasi-linear form as

$$\frac{\partial}{\partial t} \mathbf{Q} + \nabla \cdot \mathbf{F}(\mathbf{Q}) + \sum_i B_i(Q) \frac{\partial Q}{\partial x_i} = S(Q). \quad (10.2)$$

with matrices $B_i(Q)$ per space dimension i . The B_i are used in the ADER-DG and FV schemes in the Riemann solver and in the space-time predictor. We thus have two signatures:

```
NCP(BgradQ, Q, gradQ): BgradQ = B(Q)*gradQ
matrixB(Bn, Q, nv): B[n] = B(Q)
```

The symbol `BgradQ` has different meanings in the linear and nonlinear kernels:

- `BgradQ` is a vector of size `NumberOfVariables` if you use ExaHyPE’s ADER-DG kernels for nonlinear PDEs.

- `BgradQ` is a tensor of size `Dimensions × NumberOfVariables` if you use ADER-DG kernels for linear PDEs,

10.8 Point sources

While we support a generic way of expressing Source terms, this approach is not ideal in the vicinity of Dirac shaped point sources. Such sources typically occur in Seismology applications. In order to exploit the high order features of the ADER-DG scheme, we let users instead specify the actual positions of such point sources.

10.9 Custom Riemann Solvers

ExaHyPE recognizes the fact that there's not a single numerical scheme to rule all kind of problems. Therefore, the user API is quite flexible and allows users to engage parts of the generic kernels while implementing other parts on their own. For instance, in both the ADERDG and Finite Volume schemes, users can overwrite the Riemann Solver with their own implementation.

In the ADERDG scheme, the Riemann Solver is called once per patch/cell. In the Finite Volume schemes, the Riemann Solver is called on each point.



11. ExaHyPE FORTRAN

ExaHyPE is a pure C++ application. However, it allows developers a seamless integration of FORTRAN code. There are two places where FORTRAN code may be used:

- Everywhere inside the users solver. Modern compilers support binding FORTRAN functions to C and thus passing actually *all* PDE signatures, definition of boundary values, initial data, etc. to FORTRAN code managed by the user.
- The 3D ADERDG FORTRAN kernels. They descend from the original ADER-DG formulation by M. Dumbser and may be used by user applications. They are identified with the token `kernels::aderdg::generic::fortran::3d` in contrast to the generic C kernels. However, there is in principal no advantage in using the fortran kernels over the C kernels. In any case, when using the Fortran kernels, applications *must* implement their PDE functions in FORTRAN.

While the use of FORTRAN in ExaHyPEs kernels is more historical than officially supported, users are encouraged to extensively use FORTRAN in their application if needed. The ExaHyPE Makesystem detects, compiles and links FORTRAN modules and files (.f90 files).

11.1 An example FORTRAN binding

When passing arguments to FORTRAN, we stick here to the convenience rules to

- pass all parameters by reference (ie. with pointers)
- call Fortran functions named `FortranFunction` from C as `fortranfunction_`, ie. lower-cased and with a trailing underscore.

First, we present some signatures how to pass variables for an exemplary Demo project with solver named `FortranSolver` to FORTRAN:

```
void Demo::FortranSolver::flux(const double* const Q, double** F) {
    pdeflux_(F[0], F[1], (DIMENSIONS==3) ? F[2] : nullptr, Q);
}

void Demo::FortranSolver::eigenvalues(const double* const Q,
```

```

    const int normalNonZeroIndex, double* lambda) {
    double nv[3] = {0.};
    nv[normalNonZeroIndex] = 1;
    pdeeigenvalues_(lambda, Q, nv);
}

void Demo::FortranSolver::adjustedSolutionValues(const double* const x,
    const double w, const double t,
    const double dt, double* Q) {
    adjustedsolutionvalues_(x, &w, &t, &dt, Q);
}

void Demo::FortranSolver::source(const double* const Q, double* S) {
    pdesource_(S, Q);
}

void Demo::FortranSolver::ncp(const double* const Q,
    const double* const gradQ, double* BgradQ) {
    pdencp_(BgradQ, Q, gradQ);
}

void Demo::FortranSolver::matrixb(const double* const Q,
    const int normalNonZero, double* Bn) {
    pdematrixb_(Bn, Q, nv);
}

```

It is evident how simple these function calls are. For the C++ side, one might define the signatures now in an appropriate header file as

```

extern "C" {
void adjustedsolutionvalues_(const double* const x, const double* w,
    const double* t, const double* dt, double* Q);
void pdeflux_(double* Fx, double* Fy, double* Fz, const double* const Q);
void pdesource_(double* S, const double* const Q);
void pdeeigenvalues_(double* lambda, const double* const Q, double* nv);
void pdencp_(double* BgradQ, const double* const Q, const double* const gradQ);
void pdematrixb_(double* Bn, const double* const Q, double* nv);
}/* extern "C" */

```

Quantity	Size	Meaning
x	DOUBLE(nDim)	spatial positions
Q	DOUBLE(nVar)	Unknowns/Degrees of Freedom
Fi	DOUBLE(nVar)	Flux for each unknown in direction i
S	DOUBLE(nVar)	Source term for each unknown
lambda	DOUBLE(nVar)	Eigenvalue for each unknown
nv	DOUBLE(nDim)	Normal vector for computation
BgradQ	DOUBLE(nVar)	Vector $(B^k \otimes (\nabla Q)_k)_i$
gradQ	DOUBLE(nVar, nDim)	Matrix $(\nabla Q_i)_j$

Table 11.1: Parameters for the PDE solvers, in Fortran notation, cf. also appx. F.

11.2 The Fortran code

In Fortran, one might now implement the functions anywhere, for instance with

```
SUBROUTINE PDEEigenvalues(Lambda,Q,nv)
  USE Parameters, ONLY : nVar, nDim
  USE, INTRINSIC :: ISO_C_BINDING
  IMPLICIT NONE
  ! Argument list
  REAL, INTENT(IN) :: Q(nVar), nv(nDim)
  REAL, INTENT(OUT) :: Lambda(nVar)
  ! Local variables
  REAL :: foo(nVar), bar

  Lambda = 1
END SUBROUTINE PDEEigenvalues
```

Two comments about this code: First, we specify the size of the individual parameter arrays, in contrast to C. Table 11.1 gives an overview about the actual extends. Second, we make use of a module `Parameters` which is introduced for convenience in order to store the information about the number of variables in the PDE system and the number of dimensions. There is no connection at all to ExaHyPE, it's in the users duty to maintain this code:

```
MODULE Parameters
  IMPLICIT NONE
  PUBLIC

  INTEGER, PARAMETER :: nDim = 2
  INTEGER, PARAMETER :: nVar = 9
  ! etc.
END MODULE Parameters
```

11.3 Known limitations

- Note that ExaHyPE currently provides no further support for Fortran code generation in terms of providing consistency between the Fortran `nDim`, `nVar` as introduced here and the C++ counterparts (which can actually be found in the `Abstract*Solver.h` header).
- ExaHyPE's build system is quite rudimentary: It basically slurps all C++ and f90 files it can find, compiles them in no well defined order and links everything together. However, especially Fortran requires modules to be compiled before files depending on them. Thus you might have to invoke `make` several times if you run into this problem. You also might want to replace the `Makesystem` of your application with an more advanced one.
- In any case, we can compile Fortran code with both Intel and GCC compilers within ExaHyPEs build system.

11.4 Further reading

- <https://computing.llnl.gov/tutorials/bgq/mixedProgramming1.pdf>



12. Non-trivial computational domains

ExaHyPE is written with some very simple assumptions w.r.t. boundaries in mind:

- Design philosophy 12.1**
1. If you do not realise a sophisticated domain handling—the ExaHyPE consortium favours immersed boundary techniques—domains are hexahedrons or rectangles, respectively.
 2. ExaHyPE mesh elements are squares or cubes. Only and always.
 3. ExaHyPE applications may prescribe any mesh size or hexahedron dimensions. The gridding tries to meet these requirements as accurate as possible. It ensures that the mesh is at least as fine as requested. It may be finer however.

The last item is important to keep in mind: ExaHyPE may refine if it deems to be advantageous in terms of performance, e.g.!

12.1 Real and computational domains

Given ExaHyPE's design decisions, the computer assumes that

$$\Omega_{\text{exahype}} \subseteq \Omega$$

holds. If ExaHyPE can not tessellate the computational domain Ω with its cubes, it will “shrink” the domain. It tries to minimise the shrinks though, and you can ensure a further minimisation by adapting the grid towards the boundary. Furthermore, our `boundaryValues` routine is passed the position in space, i.e. your code can identify how far it is away from the actual boundary.



Upscaling and tuning ExaHyPE

13	Shared memory parallelisation	77
13.1	Tailoring the shared memory configuration	
13.2	Hybrid parallelisation	
14	Distributed memory parallelisation . . .	83
14.1	An hitchhiker's guide through MPI	
14.2	Buffer and timeout settings	
14.3	identifier and configure settings	
14.4	Hybrid parallelisation	
14.5	MPI grid modifications	
14.6	MPI troubleshooting and inefficiency patterns	
15	Optimisation	89
15.1	High-level tuning	
15.2	Optimised Kernels	



13. 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 before the solvers:

```
shared-memory
  identifier = 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 set¹. The toolkit checks whether the environment variables are properly set and gives advice 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. See the messages plotted by the makefile upon startup for the variables. Also note that all arguments within the shared-memory environment are read at startup time as none of them is marked with `const`, i.e. you can change them without rerunning the toolkit, too.

For users with an OpenMP background, please note that we do set the thread count manually within the code. OpenMP environment variables are neglected.

Using the identifier `dummy` 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 identifier. If you use a more sophisticated shared memory

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

strategy through another identifier, `properties-file` is important. More sophisticated strategies are subject of discussion next.

ExaHyPE's shared memory parallelisation yields reasonable speedups if and only if your problem is sufficiently compute intense and regular. If you work with low polynomial order, very coarse meshes or meshes that are extremely (dynamically) adaptive, the shared memory parallelisation will not yield a massive speedup. You might decide to use MPI to exploit cores instead.

Table 13.1: Parameter choices for the multicore configuration.

Parameter	Options	Description
identifier	dummy autotuning sampling	Three shared memory parallelisation strategies are provided by ExaHyPE at the moment. <code>dummy</code> uses some default values that have proven to be reasonably robust and yield acceptable speed. <code>sampling</code> tests different choices and plots information on well-suited variants to the terminal. <code>autotuning</code> tries to find the best machine parameter choices on-the-fly via a machine learning algorithm. Variants of the autotuning strategy are discussed in Table 13.2.
cores	> 0	Number of cores that shall be used.
properties-file	filename	The <code>sampling</code> and the <code>autotuning</code> 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 <code>dummy</code> strategy.

13.1 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 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 parallelisation strategy than our `dummy` plus the configuration/properties files read by these strategies. Parallelisation strategies are selected through the `identifier` option.

Design philosophy 13.1 All non-dummy strategies load tuning parameters from the properties file. We keep this file separate from the ExaHyPE specification file as some shared memory strategies automatically update/autotune the settings therein, i.e. the file is altered by the code.

Through a dump of information, knowledge can be reused in the next run, runtime characteristics can be analysed, or a knowledge base can be built up over multiple simulation runs.

Table 13.2: Variants of shared memory autotuning.

Options	Description
<code>autotuning</code>	<p>The basic variant <code>autotuning</code> tries to find the best machine parameter choices on-the-fly. When your code terminates, the found configuration is piped into the properties file. The next time you run your code, the autotuning does not start its search from scratch. Instead, it uses the state documented by the file. If no file exists, it starts from scratch.</p> <p>Part of the autotuning strategy is the opportunity to restart the search from time to time, i.e. if the strategy thinks it has found a reasonable configuration for some parameter over quite some time, it randomly decides to try out a few other options nevertheless. This way, we try to avoid that the machine learning runs into local minima.</p>
<code>autotuning_without_learning</code>	<p>This variant reads in the properties file of the autotuning, but it does not continue to learn. The variant is reasonable for production runs where you already have found a valid shared memory configuration. If no file is found, the strategy however has to switch off multicore parallelisation.</p>
<code>autotuning_without_restarts</code>	<p>Variant of the autotuning that does never restart any search, i.e., once a working parameter configuration is found it is kept. As all searches successively are shut down, the machine learning switches itself gradually off. As the timings underlying the machine learning are expensive OS calls themselves, this might be reasonable but bear in mind that you might find good parameter combinations that represent a local minimum and are not the globally optimal parameter choice.</p>

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 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.

13.1.1 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. The whole learning process is documented via terminal outputs. It might be reasonable to disable them via a log filter entry (cmp. Chapter 16.1).

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 property file, i.e., the autotuning starts from the previous properties dump. This way, long-term learning can be split among various program invocations.

- 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 performance as ExaHyPE is forced to switch off shared memory parallelisation if the properties file is malformed.
- 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.
- As the autotuning reads text configuration files, it is possible to configure the autotuning manually. The format of the properties file is documented as comment within the source code.

13.1.2 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 property file, i.e., the statistics are incrementally improved. This way, a long-term statistical analysis can be split among various program invocations.

13.2 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.



14. 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 through there are MPI-2 variants available (cmp. Peano's documentation). This chapter provides an MPI hitchhiker's guide through MPI first, as we acknowledge that MPI upscaling often is not straightforward, before we detail ExaHyPE's MPI specifics.

14.1 An hitchhiker's guide through MPI

Prepare build

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-naive,FCFS}
  buffer-size = 64
  timeout = 120
end distributed-memory
```

The above file is minimal and thus well-suited for a first try. 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.

Prepare a benchmark simulation setup

Before you start to do any MPI experiments, we strongly recommend that

1. you switch off any output. Adding output is a second thing you can do once your application scales to a certain number of nodes/cores. But starting with IO enabled right from the start makes, in our experience, things way more complicated—notably the identification and isolation of issues. The straightforward way to disable any output is to set the value of the `time` attribute, i.e. the first time a file should be written, to something bigger than the total simulation time.
2. you change your grid resolutions such that your simulation fits onto exactly one node of your cluster without exceeding the memory. It is very handy to have a reasonable baseline that still runs serially while it is as large as possible to see scalability issues early. To obtain information about your code's memory consumption, recompile with `-DTrackGridStatistics`, i.e. add it to `PROJECT_CFLAGS` of your application's makefile. Furthermore, ensure that your log filter file (if you use one) does not filter out the messages and contains

```
info exahype::runners::Runner::runAsWorker -1 white
info exahype::runners::Runner::createRepository -1 white
```

3. you start with a regular grid setup. This way, you can first eliminate technical difficulties before adaptivity and load balancing kick in.
4. you stick to two dimensions if possible. It just makes many things (including runtimes) easier to handle.

Fixing MPI codes can be tedious. It is thus advantageous to have one minimal setup available to study ExaHyPE's behaviour.

First run & quick checks

I strongly recommend to do three runs first before you start any MPI experiments:

1. **Run with** `mpirun -n 1`. Just to ensure that the sole compilation has not ruined anything in your code. Please ensure: Does the simulation fit into the memory? MPI will blow up your memory footprint already in this stage.
2. **Run with** `mpirun -n 2`. ExaHyPE follows Peano's MPI paradigm where the first rank is exclusively reserved for algorithm control and load balancing. This implies that a two-rank run should behave exactly as a run with only one rank: the first rank does all the admin work, the second rank does the real computation. If you don't want to sacrifice a complete rank, almost all MPI installations give you the opportunity to overbook one node, i.e. to launch more than one rank on the first node. If your code fails already in this stage, often people have written bugs into their global reduction.
3. **Run with** `mpirun -n K` where K is either ten for two-dimensional setups or 28 for three-dimensional problems. ExaHyPE is based upon three-partitioning of grids. With one rank reserved for load balancing and administration $1 + 3^d$ is a core count that makes it very easy for any ExaHyPE configuration to distribute the grid already at startup. If the simulation hangs, doublecheck the memory requirements of the individual ranks, i.e. whether they can be accommodated by your cluster. If possible, it makes sense to issue the first run with 10 or 28 ranks on one node. In this case, MPI falls back to shared memory data exchange, i.e. buffer locking effects are avoided.

From hereon, ExaHyPE is prepared to scale up the problem size as well as the grid resolution.

The upscaling cycle

We recommend to scaleup ExaHyPE in an iterative scheme that reads as follows:

1. Increase the problem size **or** increase the core count **or** change the ExaHyPE specification settings/problem characteristics.

2. Conduct a first test. If the scalability is satisfying, continue with 1. Otherwise proceed as follows.
3. Recompile your code in the mode `PeanoProfile` and rerun the experiment. Pipe all output into a text file but ensure that your log filters enable logs from `peano::performanceanalysis`.
4. Pass the output file to the Python script `peano/performanceanalysis/performanceanalysis.py`. If you call the script without arguments, it yields a usage message. You obtain a html page that you can open with any browser.
5. Search for inefficiency patterns in the output (cmp. Section 14.6), alter your code settings and return to 1.

14.2 Buffer and timeout settings

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. Set the value to zero if you don't want to use the time out detection.

Design philosophy 14.1 Peano, ExaHyPE's AMR code base, introduces timeouts to allow you to identify deadlocks without you burning too many CPUs as well as communication inefficiencies. We recommend to start with rather restrictive timeout settings and to scale the timeout with the grid size.

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.

14.3 identifier and configure settings

Through `identifier`, ExaHyPE users set the load balancing paradigm used. We support

identifier	Semantics
<code>static_load_balancing</code>	A static load balancing that decomposes ExaHyPE's underlying spacetime at construction time.

The required/permitted values in the `configure` field depend on the `identifier` chosen and thus are enlisted in the respective tables. A configuration is simply a comma-separated list of identifiers that control how the chosen load balancing strategy is behaving.

- We offer various metrics—analysis types that guide the load balancing paradigm how to split up the grid (Table 14.1).
- ExaHyPE implements a centralised rank administration: rank 0 has a list of all ranks available and decides which rank is used next whenever some load balancing is triggered. We offer different ways how rank 0 (we call it global node pool) answers requests (Table 14.2).
- Multiple additional settings can configure further behaviour. They are discussed next.

14.4 Hybrid parallelisation

Please see Section 13.2 for details how MPI and the shared memory parallelisation interplay.

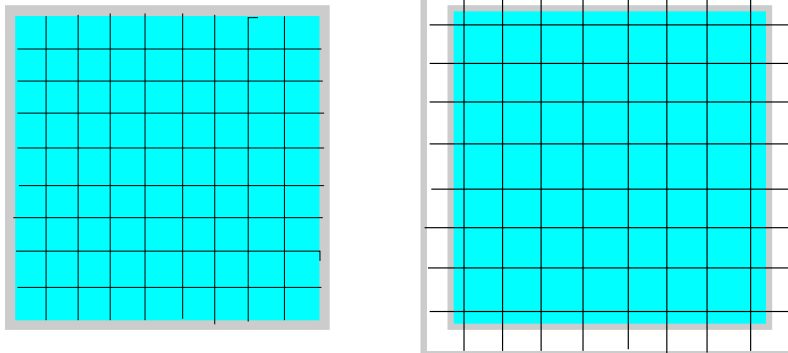
Argument	Semantics
greedy-naive	Uses a greedy split strategy to decompose the underlying spacetree. The individual ranks do not synchronise with each other, i.e. one rank is unaware of load imbalanced on another rank. This scheme is fast and naive but might yield non-optimal partitions right from the start.
greedy-regular	Extension of greedy-naive that takes into account that grids are (rather) regular. These regular grid levels then are distributed aggressively. Very fast.
hotspot	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 as it searches for heavy parts within the grid and splits those up first.

Table 14.1: ExaHyPE supports various metrics, i.e. ways the load balancing is guided. We recommend to start with greedy-naive always.

14.5 MPI grid modifications

MPI communication is very sensitive to the grid layout. ExaHyPE can minimise data exchange with the critical first rank which controls the whole simulation workflow (and thus quickly becomes a bottleneck) by slightly stretching and realigning the grid. To switch on this feature, add `virtually-expand-domain` to your configuration:

```
configure = {...,virtually-expand-domain}
```



If you have a grid that is very regular along the bounding box (sketch above, left), all ranks synchronise with rank 0 each and every time step as rank 0 holds knowledge about changes along the global domain boundary. This renders rank 0, who otherwise holds no compute data at all, a bottleneck. With the additional flag, the compute grid is slightly enlarged and thus the ranks responsible for boundary cells have all boundary knowledge local—if a cell overlaps the boundary, then all boundary conditions are evaluated locally; problems arise, if the computational boundary coincides with the grid boundaries.

While a virtual expansion speeds up some codes dramatically, its impact on the chosen mesh size and load balancing has to be studied carefully. Mesh sizes are modified, global cell counts may

Argument	Semantics
FCFS	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	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 <code>ranks_per_node:X</code> , too, such that the load balancing knows how the ranks are distributed first. If n ranks request for one worker each for a level ℓ , the load balancing tries to make each node, i.e. every <code>ranks_per_node</code> th rank, serve this request. Notably, the ranks responsible for the coarsest tree levels are assigned to the ranks <code>ranks_per_node</code> , $2 \cdot \text{ranks_per_node}$, $3 \cdot \text{ranks_per_node}$, 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.

Table 14.2: Configuration of the global node pool, i.e. how ExaHyPE decides which ranks to use next.

change as refinement criteria yield slightly different grids, and the load balancing becomes different and harder: Grids that did fit to a certain rank count brilliantly before might now be grids that are very hard to decompose reasonably. In general, we thus recommend to use this feature for large MPI rank counts only where the high number of ranks gives ExaHyPE the freedom to find proper load balancings. Furthermore, we recommend that you use the a non-greedy load metric if you switch this feature on (hotspot, e.g.).

14.6 MPI troubleshooting and inefficiency patterns

For many of the smells below, it is quite useful to translate the code with `-DTrackGridStatistics`. The flag makes your code slightly slower helps to identify many smells quickly.

- **I have increased my problem size and my code crashes now.** Often, scheduling systems return signal 9 but the outcome might be different. Validate that you are not running out of memory! ExaHyPE does not add, for performance reasons, catches around all memory allocations. It might be that you have spread your simulation over more cores or that you have to decrease your mesh resolution.

If you see, for a smaller-scale run, each rank require around $160MB$ and if you run your code with dimensions $d \in \{2, 3\}$ and if you have a rather regular (startup) grid and if you run 6 MPI ranks per node, then a mesh of one additional refinement level (decrease your mesh to $1/3h$) already will require at least $6 \cdot 3^d \cdot 160MB$ per node.

Please note furthermore that ExaHyPE first builds up the grid up to a certain (regular) level before it befills the grid with content, i.e. once this initial grid is set up, you will require in

one rush a massive amount of memory.

- **My code crashes.** Compile with `MODE=asserts` and rerun. Noone working with ExaHyPE will be able to give you useful support without an assertion run. Please increase your timeouts significantly before you launch a simulation with assertions. Your code will be significantly slower.
- **I have increased my problem size and now run into a time out.** Increase time out values. See Section 14.2.
- **I have an (almost) regular grid but the load balancing seems not to yield very good decompositions.** This smell manifests typically in unbalanced logical topology trees if you run Peano's Python scripts. They also produce a basic visualisation of your domain splitting. The load balancing configuration greedy-regular (Table 14.1) might be better suited.
- **I have a adaptive grid and the load balancing seems not to yield very good decompositions.** This smell manifests typically in unbalanced logical topology trees if you run Peano's Python scripts. They also produce a basic visualisation of your domain splitting. The load balancing configuration hotspot (Table 14.1) might be better suited.
- **ExaHyPE successively puts load on all nodes but as it fills them one after another and as I have multiple ranks per node, my first few nodes are heavily booked while the other ranks are assigned a smaller amount of work.** Switch the node pool strategy to fair (Table 14.2).
- **The load balancing seems to be ok, but I permanently run into time outs and the performance analysis reports late sender w.r.t. rank 0.** Add the load balancing flag `virtually-expand-domain` to your configuration but carefully study the implications (Section 14.5).
- **I do not have success at all to parallelize my application.** You might want to start with a simpler PDE with less unknowns. The lowest common denominator is an advection equation $\partial_t U + \partial_x U = 0$ with one unknown or the classical hydrodynamics as in section 6.1 (5 unknowns). Also you should start with the simplest scheme possible, i.e. start with a Finite Volume scheme (section 6.7) and try it to parallelize with MPI. If this works, use instead the ADER-DG scheme and try again. Only if both schemes work on their own, you are ready to try to parallelize the coupled ADER-DG scheme with limiting (section 8). Also, only change one thing at a time (ie. do not change the PDE and the scheme at the same time) in order to determine what scenario can be run with your setup and which not.
- **To be continued ...**

For further helpful hints to get MPI running, take a look into the developer FAQ section in appendix B, the user FAQ section in appendix C and the known bugs and limitations in appendix D. If you try to get MPI working on a well-known supercomputer, probably appendix ?? can give you an advice.



15. Optimisation

15.1 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 required 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 iterations 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 significantly. 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

```

15.2 Optimised Kernels

ExaHyPE offers 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 with `numpy` and `jinja2`. 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:

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

Table 15.1: Supported flags

```

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

```

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 15.1. 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.

Working with ExaHyPE's output

16 ExaHyPE program output 95

- 16.1 Logging
- 16.2 Grid statistics

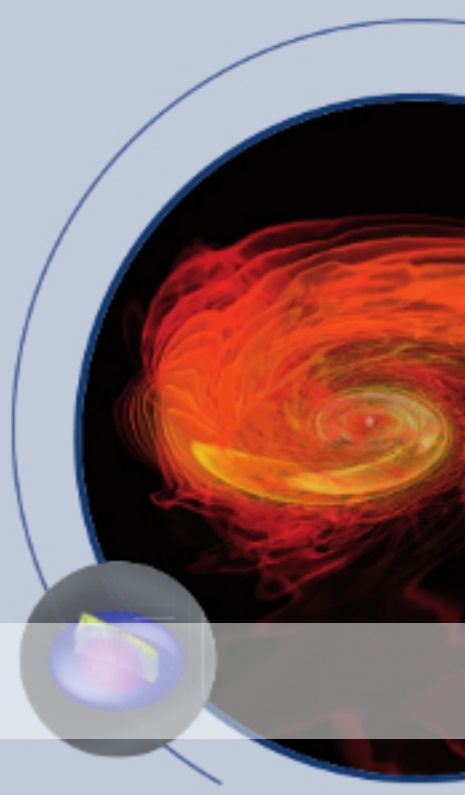
17 Plotting 97

- 17.1 Cell and subcell structure in the ExaHyPE output
- 17.2 Overview of supported data output formats
- 17.3 Filtering
- 17.4 Parameter selection and conversion
- 17.5 User-defined plotters: Understanding the Plotting API

18 Postprocessing 107

- 18.1 On-the-fly computation of global metrics such as integrals
- 18.2 Reduction of global quantities over all MPI ranks

16. 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 17 while real on-the-fly postprocessing such as the computation of global integrals is picked up in Chapter 18.

16.1 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.

16.2 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 wherever 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
```




17. 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 or the elimination of helper variables from the output.

Design philosophy 17.1 ExaHyPE's IO is per se very simplistic as we do acknowledge 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 visualization nodes in scientific clusters.

17.1 Cell and subcell structure in the ExaHyPE output

All standard kernels shipped with ExaHyPE make use of subcell grids. In the case of the ADER-DG kernel, this are the polynomial supporting points. For instance, for a second order DG approximation in 1D, the output contains three data points per cell and thus holds almost three times more data than with a cell-averaging output method. This can be translated to the FV kernels where there is a patch of subgrid points in each cell.

Depending on the output format (cf. next section), you may be able to exactly reconstruct e.g. the polynomials from a run.

In shock-free simulations, such a subcell output format introduces a certain redundancy, as

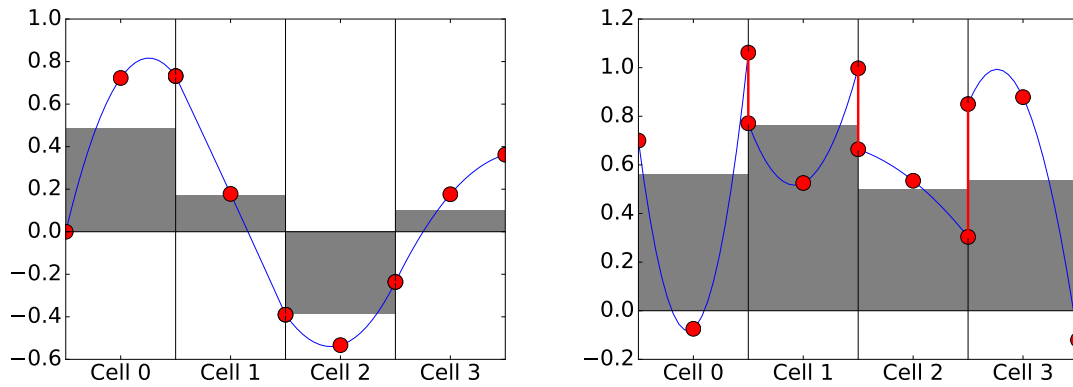


Figure 17.1: A sketch of ExaHyPE’s output point structure in a 1D example with four cells and a polynomial degree of 2. In each example, the polynomial degree is plotted. The shaded regions indicate the cell average. In the regular case of a continuous solution, there are two data points at the cell 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 a double-valued function. Note that this sketch is simplifying the ADER-DG scheme: In reality, the supporting Legendre points are not even equally spaced.

values at the cell boundaries are given out (“plotted”) two times. However, these points provides additional information as soon as discontinuities appear, cf. figure 17.1.

While a visualization software might use the data to reconstruct the polynomial, we don’t expect this to happen in every-day visualisations. This has two reasons: First, it is an expensive computation. Second, it needs information about the subgrid structure. In principle, this information is present in the output files. However, in general, it is not easy to extract (ie. from VTK files).

17.2 Overview of supported data output formats

The text below lists all available plotter types. They are distinguished by their identifier in the specification file. Note that due to limitations of the glue code builder, the number and order of the plotters have to be fixed at toolkit time. You can read them off by calling

```
> ./ExaHyPE-YourApplication --version
....
Toolkit static registry info
=====
projectName: YourApplication
....
Kernel[0].Plotter[0]: YourApplication::ConservedWriter(variables=19)
Kernel[0].Plotter[1]: YourApplication::ConservedWriter(variables=19)
Kernel[0].Plotter[2]: YourApplication::SphereIntegrals(variables=0)
Kernel[0].Plotter[3]: YourApplication::IntegralsWriter(variables=0)
...
```

This example corresponds to a specification file where a solver is equipped with at least four plotters where the first two use the same mapping class (“ConservedWriter”) while the third and fourth use a different one, cf. section 17.4 for further details.

If you want to change the number of plotters or their type, ie. if you want to introduce a new

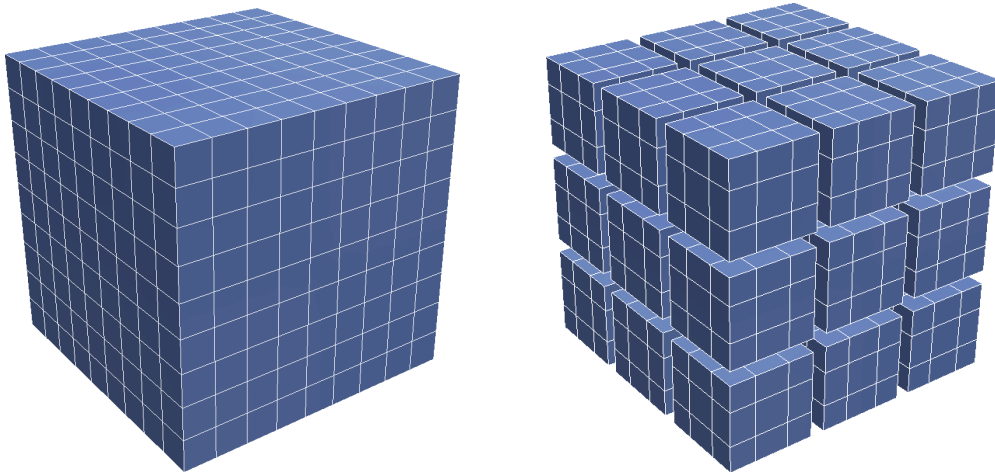


Figure 17.2: Cartesian vs. Legendre volumetric plotters: Both cubes show a one-time refined cube with $p=3$ ADERDG subgrid, left equally spaced interpolated, right on the Legendre points.

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` from your solver. If `unknowns` is greater than the `unknowns` of the solver, you also obtain the material parameters if there are material parameters. Otherwise you do not obtain useful data but could map your `unknowns` to locally derived quantities.

17.2.1 Volumetric plotters with generic file formats (VTK)

All volumetric plotters are by construction able to plot the full simulation domain. To do so, the VTK file format is used. As a first rule of thumb, ExaHyPE currently supports the following combinations of plotter parameters. Note that this only covers the VTK plotters and also not all special cases:

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

For an in-detail difference about the storage of cell or vertex data, cf. section G.1. Figures 17.2 and 17.3 visualize the difference between Cartesian and Legendre plotters. Figure 17.4 visualizes the difference between Vertices and Cells sampling. The variant `vtk` is a legacy format supported by most postprocessing tools. `vtu` is a newer format¹. If you use `vtu`, ExaHyPE does generate meta data formats such as `pvtu` (only with MPI) or `pvd` for the time series that allow you to load a whole set of `vtk` files in one rush. The following sections refer only to the prefix `vtk`. You may always replace `vtk` with `vtu`.

Available VTK plotter parameter combinations

`vtk::Cartesian::vertices::ascii` Dumps the whole simulation domain into one VTK file.

If you use multiple MPI ranks, each rank 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

¹We currently do not fully support binary `vtu`, i.e. we recommend to use the text variant.

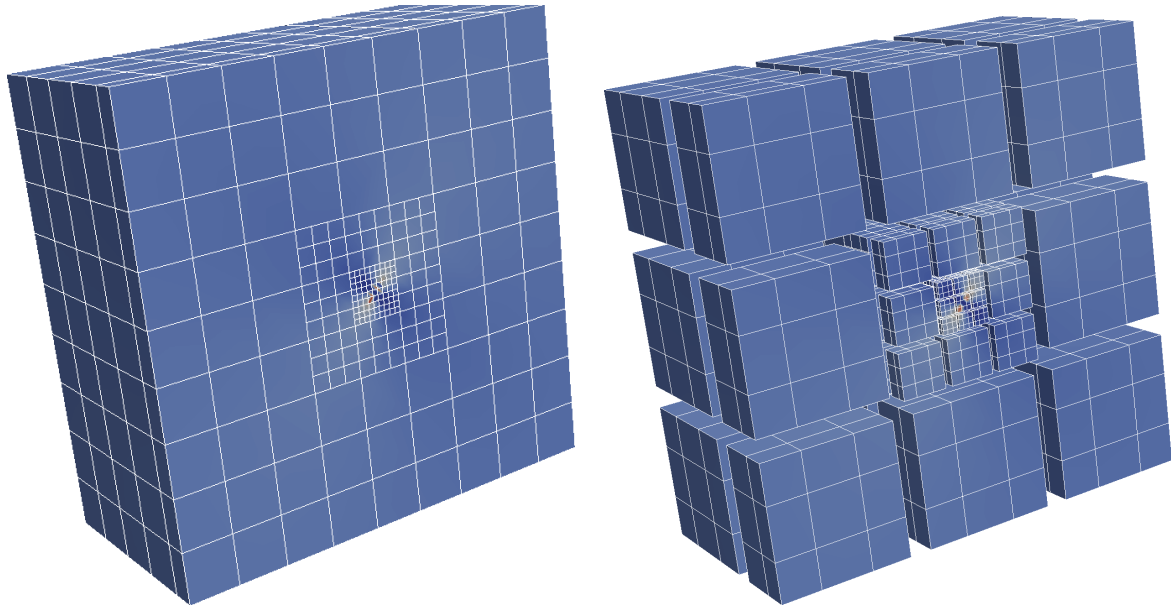


Figure 17.3: Cartesian vs. Legendre volumetric plotters, with focus on mesh refinement, in this example around the center. The plot shows a color encoded scalar field.

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.

`vtk::Cartesian::vertices::limited::ascii` This plotter is for the LimitingADERDG solver and not suitable for any other solver/scheme. It does everything the ordinary plotter (without "limited" in the name) also does. On top of that, it includes another field "LimiterStatus" which encodes the limiting status of cells in discrete numbers $n \in \{0, 1, 2, 3\}$ where 3 indicates a troubled (limited) cell, 2 indicates a next-to-troubled (NT) cell, 1 indicates a next-to-next-to-troubled cell (NNT) and 0 indicates an ordinary untroubled cell. The limiting status of all points inside a cell/patch is similar, so this method is not efficient at all but mainly suitable for debugging.

`vtk::Cartesian::vertices::limited::binary` as above.

`vtk::Cartesian::subcells::limited::ascii` We don't know what this plotter is for.

`vtk::Cartesian::subcells::limited::binary` as above.

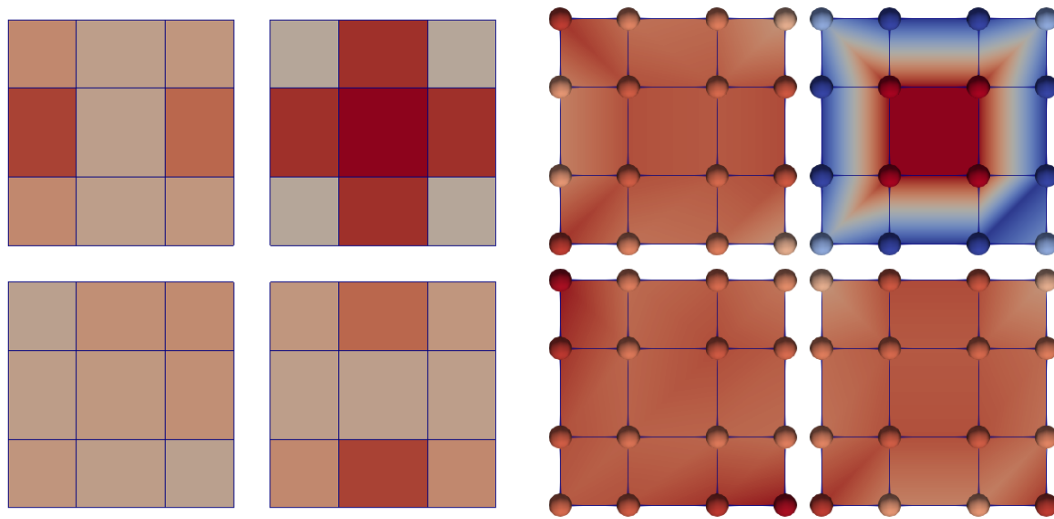


Figure 17.4: Cell-centered plotting (left) vs. plotting at vertices (right) at the example of the Legendre plotter. For the cell centered plotting, values are sampled in the centers of each plotting cell, in the case of the Legendre plotter this is in each subcell center. In contrast, plotting on the Legendre vertices shows exactly the degrees of freedom as in the computation. In order to highlight the values at the vertices, they are represented by coloured balls. Typically one only sees a linearly interpolated cell colouring in such a display.

Solver-specific peculiarities

Note that not all plotter types are available with each solver. Instead, only some combinations make sense. We should come up here with some flags or table in order to omit this particular chapter. However, instead, we provide this list here to constraint the description list given in the previous section to a particular solver.

In ExaHyPE, we currently have three solvers:

ADERDGSolver This is the most versatile solver in terms of plotting as it allows a lossless interpolation of the polynomials on any other basis. The internal representation (“where the data live”) is the vertex-centered Legendre basis. Thus, when plotting with the combination `Legendre::vertices`, you obtain the real position of the data how they are treated in the simulation. This can be very helpful in order to understand how the computational grid really looks like.

However, as we are working with polynomials, we can interpolate them on any other basis making the intrinsic error of the method (\sim polynomial order). We also allow to plot values cell-centered or vertex-centered – it is only a matter of a small position shift for the interpolation of a polynomial.

FVSolver The internal representation of the Finite Volume solver data is the cell-centered Cartesian basis. Thus, you only obtain the real unchanged data when using a `Cartesian::cells` combination. Indeed, for the VTK plotters, we don’t even allow a different combination, ie. we don’t do interpolation².

LimitingADERDGSolver When it comes to plotting, one can always treat a coupled ADERDG Solver to a FV scheme (ie. the LimitingADERDG scheme as described in chapter 8), ie. at every plotter invocation a recovery takes place in the Finite Volume cells. Thus you can treat the LimitingADERDGSolver in a similar way to the uncoupled ADERDGSolver.

²Note that the CarpetHDF5 plotter allows you to interpolate Finite Volume Solver data also vertex-centered. This could be easily extended to cover VTK plotters, too

However, it is also possible to export the further information about a cell hold by the LimitingADERDGSolver. For instance, one can think of plotting the subcell structure in a clearly troubled cell. This operation is not necessarily straightforward as one could with a similar argument plot also the subcell structure in the neighboured and next-to-neighboured cells. Thus, as a limitation, we currently don't have any (VTK) plotter in ExaHyPE plotting the subcell structure.

In contrast, we do have the `vtk::Cartesian::vertices::limited::...` plotters which are supported only for the LimitingADERDGSolver for obvious reasons.

17.2.2 Tailored volumetric output formats

The VTK file formats from the previous chapters are a generic output format for 2D and 3D data which is supported straightforwardly by many visualisation tools. However, there is a large overhead of metadata describing the position and size of cells tied to these formats as they basically project ExaHyPE's block-structured mesh onto an unstructured mesh. We thus support a number of different more specific file formats, too:

`Peano::Legendre::cells::hdf5` Dumps data in the Peano block regular fileformat, as proposed in section G.2. It resembles closely the array-of-structure in-memory storage layout of ExaHyPE and saves the majority of the metadata overhead by essentially storing $\vec{x}, d\vec{x}$ of each cell instead the positions of each data point in the subcell structure. At the moment, Peano's block regular file format supports ASCII text and HDF5.

`Carpet::Cartesian::Vertices::HDF5` The CarpetHDF5 block regular fileformat is the file format used by the Cactus/Carpet/EinsteinToolkit codes. It has the advantage of wide support in common tools (such as a reader included in Visit and Amira as well as desktop players for 1D and 2D movies). The CarpetHDF5 file format supports real 1D/2D/3D slicing, ie. when taking a lower-dimensional slice in an ADERDG solver, we evaluate the polynomials on the submanifold.

The CarpetHDF5 file format is not suitable for large ExaHyPE runs due to the enormous meta data overhead, mainly coming from the structure-of-array data layout (each written unknown goes into an own table) while ExaHyPE uses an array-of-structures approach (all written unknowns at one physical point are written together in one table). Also note that the CarpetHDF5 file format can only represent cartesian blocks, ie. each block must have a cartesian subgrid with fixed point/lattice spacing $d\vec{x}$. The data points in the CarpetHDF5 file format live on the vertices.

In ExaHyPE, we currently support plotting the CarpetHDF5 file format straightforward from an ADERDG scheme by interpolating on a Cartesian subgrid. We also support dumping data in the CarpetHDF5 file format from the Finite Volume scheme, however this yields in data loss in the outputted files as the interpolation loses exactness of the cell centered simulation data. The output will look blurred on coarse grids.

For details about the CarpetHDF5 files as well as an overview about the many postprocessing tools, see section G.3 in the appendix. In order to use the CarpetHDF5 file format, you have to build ExaHyPE with HDF5 support.

`hdf5::flash` The experimental FlashHDF5 block regular file format tries to resemble the file format used by the FLASH code. At the current stage, a compatibility is not yet given. Once this is finished, the file format will have a wide support in common tools such as Visit and standalone postprocessing and visualization tools. For more details see section G.4.

Enable HDF5

All proposed tailored volumetric output formats have the option or require the HDF5 container format to properly work with. To do so, they rely on the HDF5 bindings.

Design philosophy 17.2 As users shall be able to use ExaHyPE without any external libraries, we make the HDF5 support optional for all provided plotters. If you enable a plotter which requires HDF5 during runtime on an ExaHyPE build without HDF5, the plotter will not do anything but print warnings at every plotter step. This allows you to get started or continue working with ExaHyPE without worrying about dependencies.

If you want to have ExaHyPE crash in case of missing HDF5 support, for instance when you run a large job and cannot afford an accidental wrong compile setting which would result in wasting large amounts of CPU hours, you can set the environmental variable `EXAHYPE_STRICT` before invoking ExaHyPE built in with these plotters, i.e.

```
export EXAHYPE_STRICT="True"
```

If you want ExaHyPE to run with HDF5, you have to compile with `-DHDF5`. The simplest way to achieve this is to overload the environment variable `EXAHYPE_CC`:

```
export EXAHYPE_CC="mpiCC-DHDF5"
```

Furthermore, you have to link against both the `hdf` library and `hdf5_cpp`:

```
export EXAHYPE_CC="mpiCC-DHDF5-lhdf5-lhdf5_cpp"
```

17.2.3 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.

n.a. We integrated the possibility to plot only integrals/sums of values into the ExaHyPE core. This technique is proposed in section 18.1. However, currently we did not expose this yet with a

plotter device.

17.3 Filtering

Filtering allows you to reduce the dimensionality of your output or to plot only certain regions of the computational domain. Note that by default and without filtering, ExaHyPEs plotting is always full $3D^3$. For big simulations, this is certainly not always useful.

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 for the individual plotters. 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 unknowns are to be written. However, you can use the postprocessing techniques discussed below to plot other unknowns from your unknown set.

17.4 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 conversions 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
```

³in case of a 2D simulation, the plotting is of course always full 2D

...

Originally, we did plot all five variables. Lets assume we are only interested in Q_0 and Q_4 . 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];
}
```

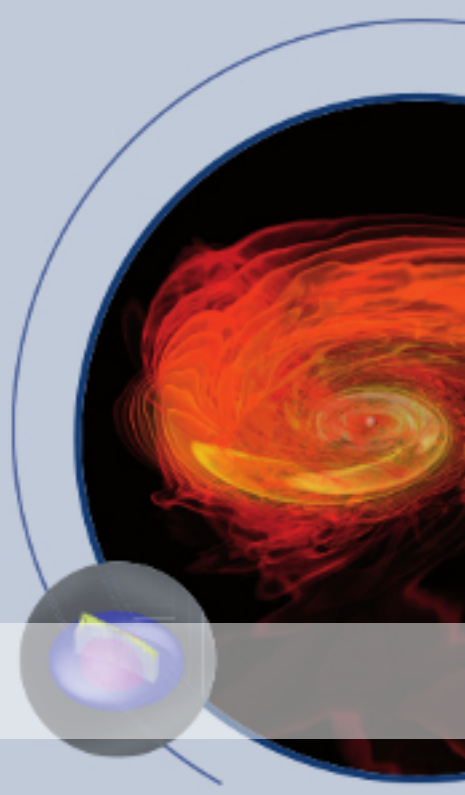
Alternatively, with all the information like the space x and time t at hand, you can easily inject other quantities into `outputQuantities`, for instance exact initial conditions. You can also use the information to compute *locally* a derived quantity from Q .

17.5 User-defined plotters: Understanding the Plotting API

ExaHyPE already proposes a number of plotters, but certainly this will not cover all users needs. Thus, our plotting API (C++ class API) is written in a way that should allow users to come up with their own plotters.

Users can start to implement their own plotters when providing the string `user::defined` as plotter description. The toolkit then generates another method in the user plotter. Instead of allowing to map quantities, users now can decide what to do when an individual patch is visited by Peano. One can then access all unknowns on a patch, or, more generic, the cell description.

18. Postprocessing



18.1 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 L_2 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

    // This is the correct scaling for FV, but not DG.
    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.

18.2 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 interfere 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.



ExaHyPE design & rationale

19	ExaHyPE architecture	115
19.1	A solver engine	
19.2	A user's perspective	
20	ExaHyPE workflow	119
20.1	Toolkit usage	
20.2	ExaHyPEs build system	
20.3	Build management	
20.4	Simulation management	

19. ExaHyPE architecture

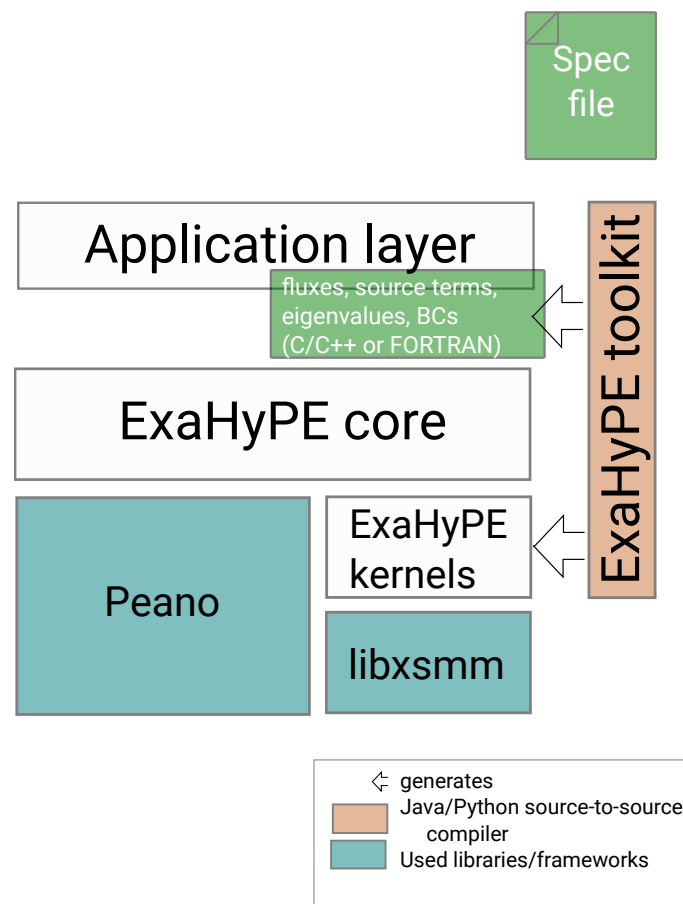


Figure 19.1: The generic ExaHyPE architecture at a glance

19.1 A solver engine

ExaHyPE is a solver engine (Figure 19.1), 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 empty 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 further software fragments. An example is given in Figure 19.2. 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.

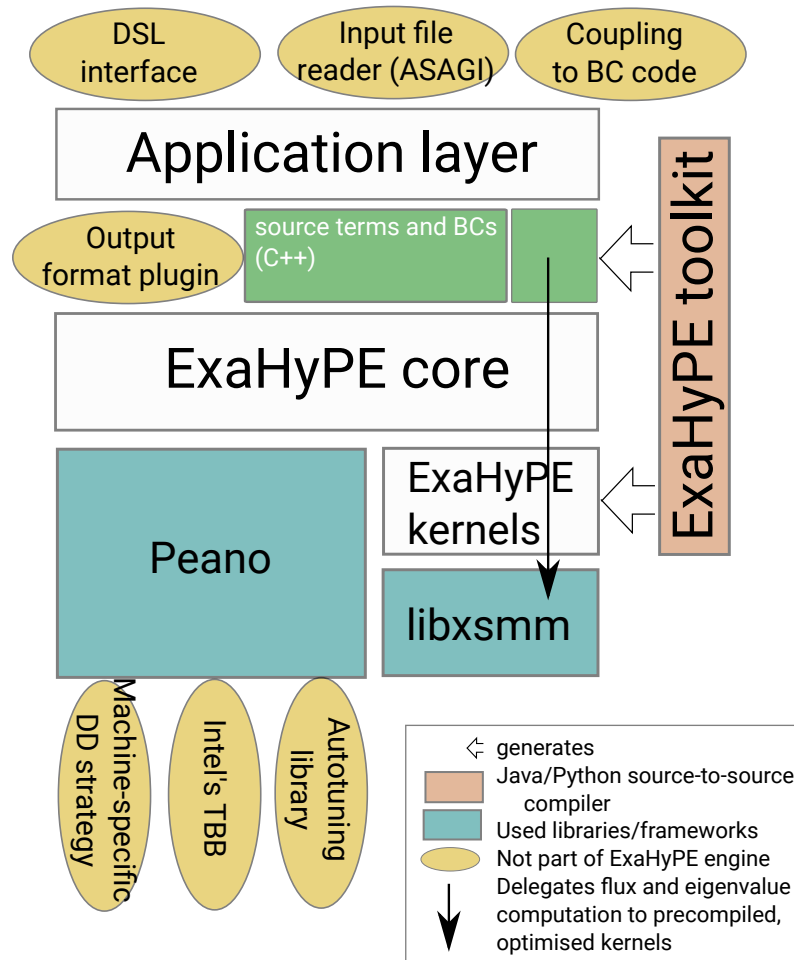


Figure 19.2: Any ExaHyPE application tailors/uses/adopts the generic architecture to its needs. The present snapshot shows a sophisticated seismology ExaHyPE code that couples the simulation engine to 3rd party software and provides a domain specific language (DSL) in the top layer, delegates all flux 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.

19.2 A user's perspective

Typically, a user makes his first contact with the ExaHyPE software when trying to setup a new application, solving a specific physical problem with a PDE. Doing so, she might go down the rabbit hole, exploring the infrastructure and class hierarchy/dependency. A typical simplified picture one might end with might be figure 19.3 which outlines the stack trace of a typical invocation of the user's PDE.

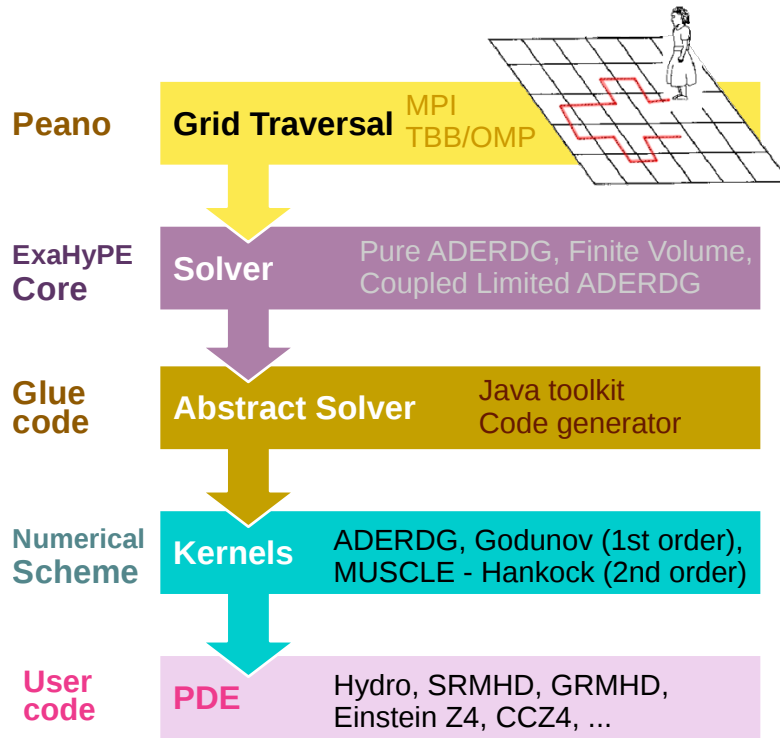


Figure 19.3: The different layers of ExaHyPE, a C++-centric way of understanding the architecture of ExaHyPE (compare with figure 19.2.)

It turns out that every part of figure 19.3 performs an individual, mostly modular and separable task. It is strongly supported to employ an IDE (integrated development environment) instead a simple text editor to enable browsing around in the rich class structure of an ExaHyPE application.



20. ExaHyPE workflow

20.1 Toolkit usage

The typical workflow when working with the ExaHyPE engine is graphically displayed in Figure 20.1 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 19.1 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.

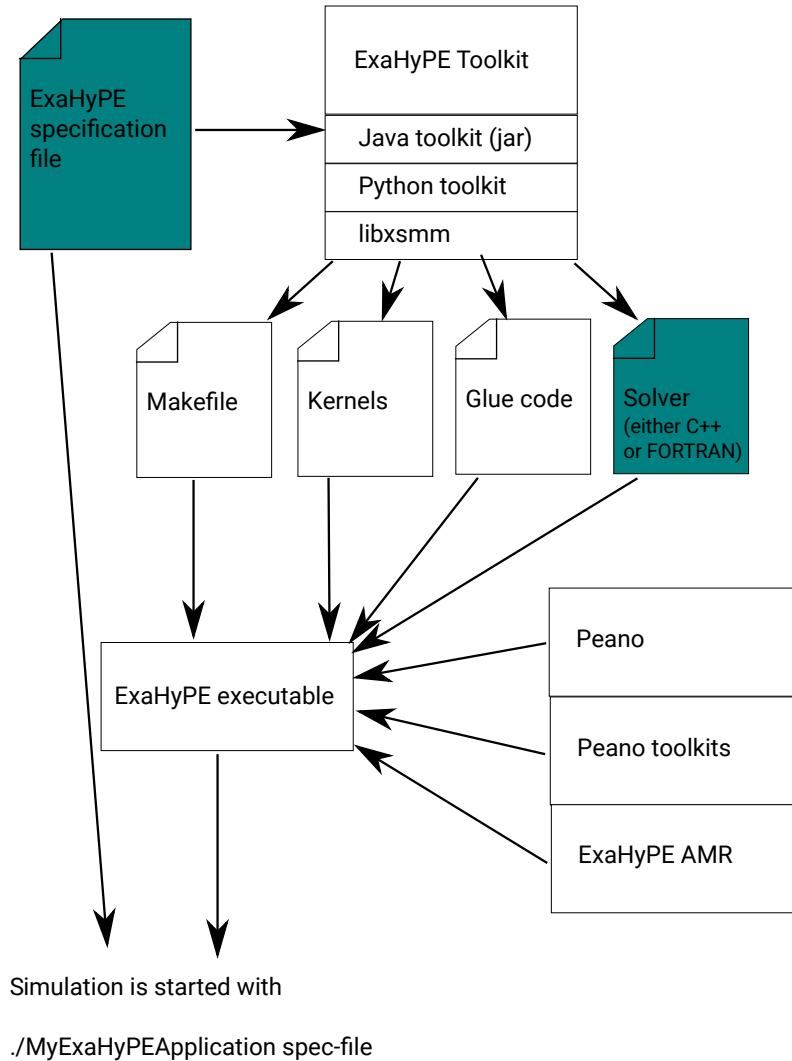


Figure 20.1: A typical ExaHyPE workflow.

20.2 ExaHyPEs build system

The build system of ExaHyPE is driven by environment variables. The major variables to drive a build are given in the following:

The ExaHyPE build cheat sheet
=====

```
export COMPILER=GNU
export COMPILER=Intel
```

Select GNU compiler
Select Intel compiler (default)

```
export MODE=Debug
export MODE=Asserts
export MODE=Profile
export MODE=Release
```

Build debug version of code
Build release version of code that is augmented with assertions
Build release version of code that produces profiling information
Build release version of code (default)

```
export SHAREDMEM=TBB
```

Use Intels Threading Building Blocks (TBB) for shared memory parallelisation

```
export SHAREDMEM=OMP
export SHAREDMEM=None
```

Use OpenMP for shared memory parallelisation
Do not use shared memory (default if not indicated)

otherwise by "shared memory ..." message above)

```
export DISTRIBUTEDMEM=MPI
export DISTRIBUTEDMEM=None
```

```
Use MPI
Do not use MPI (default)
```

Design philosophy 20.1 ExaHyPEs build system is minimal, it just compiles all C++/Fortran files it can find. Since there are no external dependencies in ExaHyPE, it is easy to exchange the build system with a tool suitable to your code and libraries.

20.3 Build management

In a high performance production system, typically a number of different combinations of builds (in terms of optimizations, included features, etc.) arises. In order to organize the compilation process in a structured manner, big software packages often come up with words like *configurations* or *builds* which refer to a way how an executable (a binary) can be built reproducably, fast and in parallel. Typically, these systems are out-of-tree build systems. *Out-of-tree* means that the source code files (C, Fortran, Header files) and generated build files (object files and modules) are kept in separate directory trees.

In ExaHyPE, there is a lot of contributed code which results from endeavours to master the ExaHyPE production workflow in a quick and cross platform portable manner. All of this is optional code which you may want to use of but don't have to in order to use ExaHyPE. The main outcome of these products can be found in the repository directory `Miscellaneous/BuildScripts/`. It is:

20.3.1 The Exa toolbox

The *exa* command line utility collects all kind of standard tasks which frequently arise in the every day life of ExaHyPE simulations. Most of these tasks are accomplished by individual standalone command line executables. Such tasks are

- Manage the repository: Updating external dependencies and providing a quick way to upgrade an ExaHyPE developer installation (Updating Peano and recompiling the toolkit).
- Location and identification of the vast amount of installed applications
- Running the toolkit from any directory
- Compiling an application after running the toolkit on it, also taking into account computer-specific settings, paths, files overwritten by the toolkit (which can be restored from git), cleaning before the build.
- Quickly running an application in a dedicated directory
- Compiling an application for several polynomial orders serially
- Managing out of tree builds (setup, synchronization, compilation, execution)
- Checking and looking up of build-specific environment variables
- Simulation management
- Starting postprocessing tools from any directory
- Starting analysis tools (such as for Peano) from any directory
- ... expandable: Users can add their own commands

20.3.2 Out-of-Tree compilation and code generation

In principle, one can understand the Build process of ExaHyPE as a chain of individual processing units:

1. A code generator (ie. the toolkit or the Python3 optimized kernel generator) *compiles* C++ code from some abstract instructions. That is, the code generator is a classical compiler compiling some high-level language to C++.

2. The C compiler, creating machine code from the C code. This includes the turing complete machinery of C++ templating which is heavily used in Peano as well as in the kernels to create optimized code with the minimal amount of instructions on a certain path.
3. Actually running ExaHyPE, which is without doubt the most challenging task of the chain.

The fact that code generation is widely employed in ExaHyPE makes it difficult to apply an ordinary out-of-tree build concept which solely relies on the idea to put the build files at some other place. This approach fails as soon one wants to build in parallel different parts of the program which rely on different generated code which basically would go at the same place.

Thus, the out-of-tree concept logically starts even before the very first step of the compilation chain outlined above: Literally all the crucial code directories are mirrored (ie. copied) to an out-of-tree location and executed there, including the toolkit and all code generation.

20.4 Simulation managment

Following the logic of section 20.3.2, running a simulation is the pretty same thing as compiling a binary. In real-world applications users want to do a lot of them, typically in parallel, executed on a fast parallel file system in a computer cluster. Performing large three-dimensional time-dependent simulations is a complex numerical task¹. Thus there is an urgent need for tools managing the directory layout and queue managment. In principle, this is beyond the scope of ExaHyPE. However, similar to the build tools, there has been some contributed code to for a basic managment of generic simulations.

20.4.1 Templated specification files

As the ExaHyPE specification files are described by an unique grammar which can in principle only be *exactly* parsed by the SableCC Java toolkit, it is useful to understand specification files as common text files when it comes to batch processing. Typical attempts are therefore replacement of striking patterns such as @VARIABLENAME@ in a key-value manner. Such a mechanism is broadly employed in all kind of runscripts available at `Miscellaneous/RunScripts`, ie. the templated runners for ExaHyPE. Following the logic of the build system, they are all driven by environment variables.

¹as can be read at the website of <http://simfactory.org/>.

Appendix

A	The ExaHyPE toolbox	125
A.1	Rebuilding the toolbox	
A.2	Troubleshooting	
B	Frequently asked developer questions	127
C	Frequently asked user questions	129
D	Bugs, limitations and compilation problems	131
E	Solver Kernels	135
F	Datastructures in the Optimised Kernels	137
G	ExaHyPE Output File format specifications	141
G.1	ExaHyPEs VTK Unstructured Mesh output	
G.2	Peano block regular fileformat	
G.3	The CarpetHDF5 output format in ExaHyPE	
G.4	The FlashHDF5 output format in ExaHyPE	
H	Running ExaHyPE on some supercomputers	145
H.1	Hamilton (Durham's local supercomputer)	
H.2	SuperMUC (Munich's petascale machine)	
H.3	Tornado KNL (RSC group prototype)	
H.4	RWTH Aachen Cluster	



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.

Can I use TBB for ExaHyPE and OMP for my Initial Data? You might want to use the `init` function of your solver (cf. section 10.3) to prepare Initial Data for your simulation at runtime. If you want to take advantage of shared memory parallelization but your code is actually using OMP while you stick to TBB for ExaHyPE, you can do so. The compile flag `-DSharedTBB` ensures that ExaHyPE uses solely TBB, so if you add additional OpenMP pragmas, they go on top. Intel claims that the combination is straightforward and that there's nothing special to care about. So you can just proceed and compile with `-fopenmp`.

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 your 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.

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.

How can I get the exact nodal value at a certain position within a computational cell? ExaHyPE's ADER-DG is based upon high order polynomial, i.e., once you are in a cell, you can reconstruct down the exact nodal value spanned by the polynomials via a tensor product. ExaHyPE offers a Cartesian plotter. In this plotter, the polynomial defined through Gauss Legendre integration points is projected onto a Cartesian grid. Have thus a look into `exahype::plotters::ADERDG2CartesianVTK::plotVertexData()` for a working example. A standard workflow then reads as follows:

1. You either create a plotter (recommended) or you use `adjustSolution` to plug into a traversal. Both can be switched on, either in regular time intervals (plotter) or in each and every sweep (adjust), and then are called on every cell of the grid in the respective grid sweep.
2. Both come along with the offset of the cell or its centre, respectively (please consult the functions' documentations) as well as the cell size. So you can quickly check whether a cell overlaps with your region of interest. If not, you quit the function.
3. If a cell overlaps with your point of interest, use the interpolation routine to reconstruct the exact value at your coordinate of interest as sketched above.

How can I obtain performance analysis data? See the user FAQ. See also the Peano guidebook for more detailed information.

MPI does not work for me at all. We experienced problems with certain MPI installations at some computers. For instance, on recent Ubuntu machines, OpenMPI does not work well beyond 4 ranks while we did not experience the same issues with MPICH. Try to switch to MPICH if possible.

TBB does not work for me at all. We experienced problems with older TBB versions. For instance, recent Ubuntu versions ship quite old versions of `libtbb-dev` (version 4.2). For these versions we see failures beyond 4 cores. Try to upgrade to a recent version of TBB which you can obtain freely from the web¹. Version 4.4 should be fine.

To be continued ...

¹<https://www.threadingbuildingblocks.org/download>

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 Δx ? 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 $\Delta x_{\max} > \Delta x$. It may help to quickly choose the right value for Δx_{\max} .

d	N	Δx	Δx_{\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 (regular grid) layouts with ExaHyPE. All quantities refer to the unit cube/square.

If the exact mesh size is important to you and you don't want to extract it from plots (which is the recommended way for AMR production runs), you might want to retranslate your code with `-DTrackGridStatistics` which plots the grid resolution after each step. Typically, you this option to the makefile in your application's folder:

```
PROJECT_CFLAGS+="-DTrackGridStatistics"
```

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 plots differ from the ones I set in the specification? Plotting at specific times would require ExaHyPE to perform shorter time steps than prescribed by the CFL condition. This leads to long-term pollution of the numerical solution due to artificial diffusion. We thus track when the next plot would be due and plot as soon as the solver time (minimum over all cells) is greater or equal to the next plotting time stamp.

ExaHyPE did not create all the plots I requested. Why? It is 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. Bugs, limitations and compilation problems

ExaHyPE's MPI ping pong test fails if I compile for `MODE=asserts`. We experience memory alignment/layout issues if you translate the code with `-DPackedRecords` (cmp. papers on the DaStGen tool). While we are aware of issues here, we have, so far, not seen these layout problems cause the code to crash. We decided to leave the test in there however to document that there might be issues. Please comment the call to `pingPoingTest()`; out in the main if this causes issues for your code, i.e. if you want to run the assertion mode with MPI.

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.

I run into linker errors on macOS. Open the Makefile in the directory ExaHyPE and remove the lines

```
SYSTEM_LFLAGS += -lrt
```

If you use macOS with CLANG, you might want to download the Peano guidebook¹ that discusses how to tailor ExaHyPE's AMR code base to the MAC system.

I obtain a linker error undefined reference to `'std::regex_iterator...'`. ExaHyPE relies on C++11 standard library components realised not before gcc 4.9.0. Unfortunately, GNU did already provide some C++ headers in earlier version of upcoming C++ without implementing them. As a result, your code compiles but cannot link. You have to update to a newer GCC version.

When I try to compile with MPI, I get errors mentioning undefined `MPI_CXX` datatypes. You most likely have an MPI installation that does not implement the MPI-2.0 standard. In this case, ExaHyPE might still compile and work properly if you add

```
PROJECT_FLAGS += -DCMAKE_CXX_BOOL=CMAKE_C_BOOL
```

¹www.peano-framework.org

to the `PROJECT_CFLAGS` in your project's `Makefile`.

In general however we recommend you to switch to an MPI installation that implements the MPI-2.0 standard.

My code deadlocks right from the start once I increase the rank count. We have seen this error a couple of times on Omnipath systems driven by Intel MPI. Please consult the Peano guidebook² and your supercomputer documentation for fixes to this problem. It is an issue with the implementation and the fabric. Intel provides additional information on this at <https://software.intel.com/en-us/node/535584>.

My code seg faults in the TBB initialisation. We have seen this issue a couple of times on Ubuntu if TBBs had been installed via `apt`. In our cases, downloading TBBs directly from Intel and working against these downloads did fix the problem.

On IBM systems, my code does obviously not use all cores although I use TBBs. Please consult our remarks on SuperMUC in Chapter H.2. It is an IBM-based system where we could identify this issue and resolve it.

My code runs in Debug/Assertion mode but not in Release mode Apparently you are a victim of a compiler bug which appears at too aggressive optimizations turned on. You can do the following: Either you disable the optimizations at certain C++ files to track down the bugs, you can do so very easily for GCC/Intel by including

```
#ifdef __GNUC__ /* GCC */
#pragma GCC optimize ("O0")
#endif

#ifdef __INTEL_COMPILER
#pragma optimize("", off)
#endif
```

If you want to turn off optimizations for certain functions, use in GCC

```
int __attribute__((optimize("O0"))) yourFunc(...) { ... }
```

and in Intel compiler,

```
#pragma optimize("", off)
int yourFunc(...) { ... }
```

In GCC, you can use a pragma locally by enclosing the scope with

```
#pragma GCC push_options
#pragma GCC pop_options
```

Also, to understand optimizations, you can make use of Intels

```
-opt-report=6
```

flag to obtain a report about the optimizations³.

²www.peano-framework.org

³<https://software.intel.com/en-us/articles/vectorization-and-optimization-reports>

I haven't found the option to enable periodic boundary conditions. This is because we don't have nonlocal boundary conditions available in ExaHyPE and probably never will have. When implementing your boundary conditions (ie. when giving a solution to the Riemann problem at the simulation boundary), you only can make use of the last patch (cell) inside the simulation and not access any other patch in the simulation.

To be continued ...

E. Solver Kernels

Table E.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 E.1: ADER-DG solver types

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

Kernel	Effect
<code>user::defined</code>	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.
<code>t.b.d.</code>	

Table E.2: Finite-Volume solver types

F. 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.

$$\begin{array}{c} \text{multiple} \\ \text{of SIMD} \\ \text{width} \end{array} \left[\begin{array}{c} \text{nDOF} \\ \text{Pad} \end{array} \right] \left(\begin{array}{ccc} Kxi & & \\ 0 & \dots & 0 \\ \vdots & & \vdots \\ 0 & \dots & 0 \end{array} \right) \begin{array}{c} \\ \\ \text{nDOF} \end{array}$$

All matrices are stored in column major order. Table F.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]	$(nDOF+Pad) \times nDOF$
iK1	analogous to Kxi	
dudx	analogous to Kxi	
FRCoeff	[FRCoeff]	$1 \times nDOF$
FLCoeff	analogous to FRCoeff	
F0	[F0;0]	$1 \times (nDOF+Pad)$
equidistantGridProjector1d	analogous to generic version	
fineGridProjector1d	analogous to generic version	

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

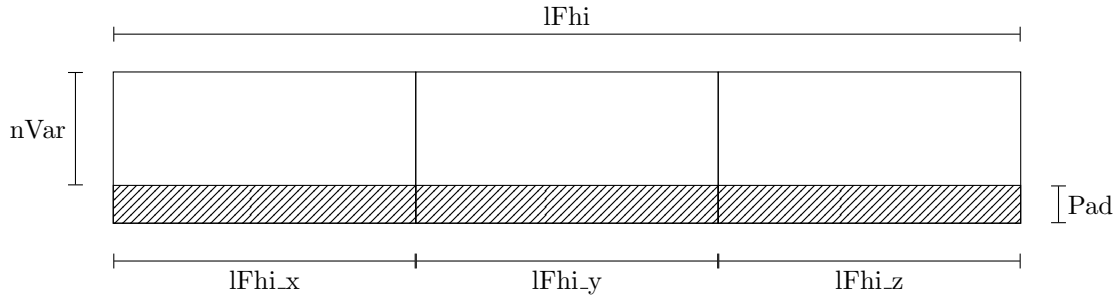
Parameters and Local Variables

Table F.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 $(nVar+Pad) \cdot nDOF^3$

Name	Ordering	Size
rhs, rhs0	(nVar,nDOFx,nDOFy,nDOFz,nDOFt)	$nVar \cdot nDOF^4$
lFhbnd	(nVar, nDOFx, nDOFy, nFace)	$(nVar+Pad) \cdot nDOF^2 \cdot 6$
lQhbnd	(nVar, nDOFx, nDOFy, nFace)	$(nVar+Pad) \cdot nDOF^2 \cdot 6$
lqh (nonlinear)	(nVar, nDOFt, nDOFx, nDOFy, nDOFz)	$(nVar+Pad) \cdot nDOF^4$
lqh (linear)	(nVar, nDOFx, nDOFy, nDOFz, nDOFt+1)	$(nVar+Pad) \cdot nDOF^3 \cdot (nDOF+1)$
lFh	(nVar, nDOFx, nDOFy, nDOFz, nDOFt, d)	$(nVar+Pad) \cdot nDOF^4 \cdot d$
lFhi	irregular	$(nVar+Pad) \cdot nDOF^3 \cdot d$
luh, lduh	(nVar, nDOFx, nDOFy, nDOFz)	$nVar \cdot nDOF^3$
temporary variables		
QavL, QavR	(nVar)	(nVar+Pad)
lambdaL, lambdaR	(nVar)	(nVar+Pad)
auxiliary variables		
tmp_bnd	(nVar, nDOFx, nDOFy)	$(nVar+Pad) \cdot nDOF^2$
s_m ("scaled matrix")	analogous to Kxi	
s_v ("scaled vector")	analogous to F0	

Table F.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.



G. ExaHyPE Output File format specifications

This chapter contains a technical specification for the file formats described in section 17.2. These are sophisticated complex file formats to hold the ExaHyPE cells/patches and their subcell structure which is either composed by the ADERDG Legendre subgrid or the Finite Volume Cartesian subgrid.

G.1 ExaHyPEs VTK Unstructured Mesh output

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 17 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 that are written binary and this binary stream is embedded into the text file.

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

G.1.1 VTK cell data

Table G.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	Cell ID	Cell Type	Q		
	x	y	z				Q_0	Q_1	...
0	0.00	0.00	0.00	0.01	0	Pixel	1.23	4.56	...
1	0.01	0.00	0.00	0.01	1	Pixel	3.41	0.07	...
2	0.01	0.01	0.00	0.01	2	Pixel	0.81	5.47	...
...					...				

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

G.1.2 VTK point data

Table G.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			Cell ID	Cell Type
	x	y	z		Q_0	Q_1	...		
0	0.00	0.00	0.00	0.01	1.23	4.56	...	0	Pixel
1	0.01	0.00	0.00	0.01	3.14	0.07	...	1	Pixel
2	0.01	0.01	0.00	0.01	0.81	5.47	...	2	Pixel
...								...	

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

G.2 Peano block regular fileformat

This file format is still subject to change, thus it is not documented in this current version of the guidebook, c.f. Removal.

G.3 The CarpetHDF5 output format in ExaHyPE

When writing CarpetHDF5 files, in ExaHyPE we generate a number of superfluous data within each dataset. The file format within HDF5 itself is very easy, it contains one dataset (table) for each cell/patch holding the values for a single unknown projected onto a Cartesian subcell grid. There are no other subgrids then Cartesian spacing (constant dx) possible. Each dataset holds a number of attributes which indicate the position \vec{x} and point spacing $d\vec{x}$ of the cell/patch. Using these two vectors, one can uniquely locate the cell in the two- or threedimensional domain.

Our implementation of CarpetHDF5 allows writing 2D or 3D data. We have can do real slicing from 3D to 2D.

Additionally to the tables, there is indeed one group called “Parameters and Global Attributes”. It must hold a Cactus-specific set of global variables which are however completely meaningless for ExaHyPE.

G.3.1 CarpetHDF5 conversion postprocessing conversion utilities

We collected standalone C++ binaries for mangling CarpetHDF5 files from various places of Cactus/the Einstein toolkit. These tools are

hdf5_merge Merge multiple CarpetHDF5 files into one file. This is straightforward as it just means that tables are copied into one file.

hdf5_slicer Slice data to 3D cubes, 2D planes or 1D lines. This is a postprocessing slicing which is probably helpful to reduce the size of an enormously large file. Note that we can do the same already online.

hdf5_extract Allows extracting (filtering) certain data sets to another file. The filter can be given by a list of requested blocks. This is another kind of postprocessing filtering which allows for instance to filter certain fields in a file with multiple fields.

hdf5toascii_slicer The same as the slicer stated before, but outputting in the CarpetASCII format. It is a suitable format for 1D data but not for higher dimensional data.

hdf5_double_to_single Reduces the IEEE754 floating point number representation from double precision (8 bytes) to single precision (4 bytes), thus saving half of the disk space for that file.

G.3.2 Postprocessing and Visualization tools understanding CarpetHDF5

This is an incomplete list of tools which understand the CarpetHDF5 file format. As a rule of thumb, all the plugins for GPU-accelerated interactive software are written in C++. The Pythonic libraries or tools all use the official Python HDF5 bindings to read HDF5 files and do not require any graphics card to work. Therefore, they are better suited to offload the visualization (HDF5 to any picture or movie format) in a postprocessing step.

Visit There is a plugin officially included in the open-source VTK-ish parallel visualization and graphical analysis software Visit¹. You can use it to display 3D and 2D data conveniently and exploit the full power of Visit to analyze and postprocess the data.

Amira There is also a plugin for the proprietary visualization software Amira².

pygraph, gnuplot can process the CarpetASCII files which are strictly speaking not equivalent to the CarpetHDF5 file format but can be generated with the utility conversion tools presented in the previous section. There are older tools called *xgraph*, *ygraph* and newer ones such as *pygraph*³ to do this plotting interactively.

¹<https://wci.llnl.gov/simulation/computer-codes/visit>

²<http://www.amira.com/>

³<https://bitbucket.org/dradice/pygraph>

scivis, **scidata** are two standalone Python libraries written by D. Radice to parse CarpetHDF5 files⁴ and to generate pictures and movies in a batch step⁵.

PostCactus is a standalone Python code written by W. Kastaun⁶ to read CactusHDF5 files. It contains a whole class library and an *OmniReader* which tries to read any kind of fields from a Cactus data directory. It also contains routines to extract gravitational waves in a postprocessing step, as well as plotting.

rugutils is a standalone Python code written by F. Guercilena⁷ to quickly plot CarpetHDF5 files using Matplotlib. It can plot time series data, still 1D data, space-time diagrams, 2D colourmaps and movies of 1D data and 2D data.

Inspect the Cactus page on Visualization for further references.

G.4 The FlashHDF5 output format in ExaHyPE

The FlashHDF5 output file format is inspired by the file format written out by the FLASH code⁸. There are a number of analysis tools for this file format, ie. QuickFLASH⁹ or plugins for software like VisIt.

Note that our FlashHDF5 file format is *not yet fully compatible to the real FLASH format*. It is work in progress.

⁴<https://bitbucket.org/dradice/scidata>

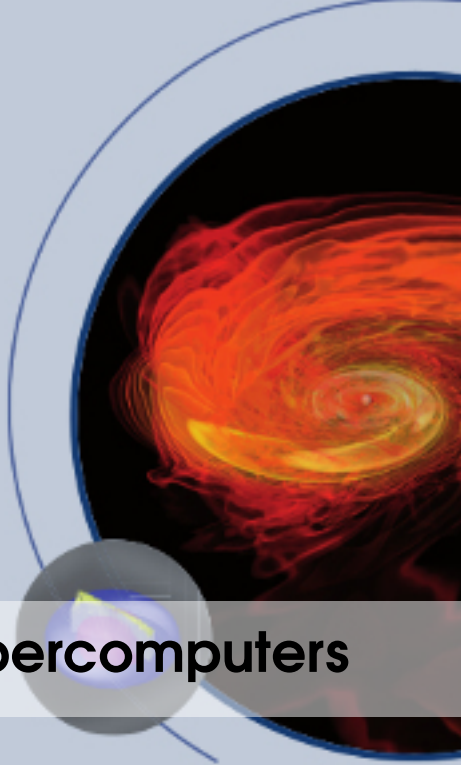
⁵<https://bitbucket.org/dradice/scivis>

⁶<https://bitbucket.org/DrWhat/pycactuset>

⁷<https://bitbucket.org/fguercilena/rugutils>

⁸<http://flash.uchicago.edu/site/flashcode/>

⁹<http://quickflash.sourceforge.net/>



H. Running ExaHyPE on some supercomputers

In this section, we collect some remarks on experiences how to use ExaHyPE on particular supercomputers.

H.1 Hamilton (Durham's local supercomputer)

We have successfully tested ExaHyPE with the following modules on Hamilton 7:

```
module load intel/xe_2017.2
module load intelmpi/intel/2017.2
module load gcc
```

Given ExaHyPE's size, it is reasonable to use `/ddn/data/username` as work directory instead of the home. SLURM is used as batch system and appropriate SLURM scripts resemble

```
#!/bin/bash
#SBATCH --job-name="ExaHyPE"
#SBATCH -o ExaHyPE.%A.out
#SBATCH -e ExaHyPE.%A.err
#SBATCH -t 01:00:00
#SBATCH --exclusive
#SBATCH -p par7.q
#SBATCH --nodes=24
#SBATCH --cpus-per-task=6
#SBATCH --mail-user=tobias.weinzierl@durham.ac.uk
#SBATCH --mail-type=ALL
source /etc/profile.d/modules.sh

module load intel/xe_2017.2
module load intelmpi/intel/2017.2
module load gcc
```

```
export I_MPI_FABRICS="shm:dapl"

mpirun ./ExaHyPE-Euler EulerFlow.exahype
```

For the Euler equations (five unknowns) on the unit square with polynomial order $p = 3$, $h = 0.001$ is a reasonable start grid as it yields a tree of depth 8. Please note that Hamilton relies on Omnipath. Once you use slightly higher rank counts and you place multiple ranks on one node, you have to specify how the Intel MPI has to use the fabric. Otherwise, your code will deadlock right from the start.

H.2 SuperMUC (Munich's petascale machine)

There are very few pitfalls on SuperMUC that mainly arise from the interplay of IBM's MPI with Intel TBBs as well as new compiler versions. Please load a recently new GCC version (Intel by default uses a too old version) as well as TBBs manually before you compile

```
module load gcc/4.9
module load tbb
```

and remember to do so in your job scripts, too:

```
#!/bin/bash
#@ job_type = parallel
##@ job_type = MPICH
#@ class = micro
#@ node = 1
#@ tasks_per_node = 1
#@ island_count = 1
#@ wall_clock_limit = 24:00:00
#@ energy_policy_tag = ExaHyPE_rulez
#@ minimize_time_to_solution = yes
#@ job_name = LRZ-test
#@ network.MPI = sn_all,not_shared,us
#@ output = LRZ-test.out
#@ error = LRZ-test.err
#@ notification=complete
#@ notify_user=tobias.weinzierl@durham.ac.uk
#@ queue
. /etc/profile
. /etc/profile.d/modules.sh
module load gcc/4.9
module load tbb
```

If you use Intel's TBB in combination with poe or MPI, please ensure that you set

```
export OMP_NUM_THREADS=28
export MP_TASK_AFFINITY=core:28
```

manually, explicitly and correctly before you launch your application. If you forget to do so, ExaHyPE's TBB launches the correct number of TBB threads as specified in your ExaHyPE

specification file, but it pins all of these threads to one single core. You will get at most a speedup of two (from the core plus its hyperthread) in this case¹.

H.3 Tornado KNL (RSC group prototype)

H.4 RWTH Aachen Cluster

We have successfully tested ExaHyPE on RWTH Aachen's RZ clusters using MUST. Here, it is important to switch the GCC implementation before you compile, as GCC is by default version 4.8.5 which does not fully implement C++11.

```
module load UNITE must

#module unload gcc
module unload openmpi
module switch intel gcc/5
module load intel openmpi

export SHAREDMEM=none
export COMPILER=manual
export EXAHYPE_CC="mpiCC-std=c++11-g3"
export COMPILER_CFLAGS="$FLAGS_FAST"
```

The above setups use the compiler variant `manual` as RWTH has installed MUST such that `mustrun` automatically throws the executable onto the right cluster. To create a binary that is compatible with this cluster, the flags from `FLAST_FAST` are to be used.

¹Thanks to Nicolay Hammer from LRZ for identifying this issue.