ExaHyPE Guidebook

User manual

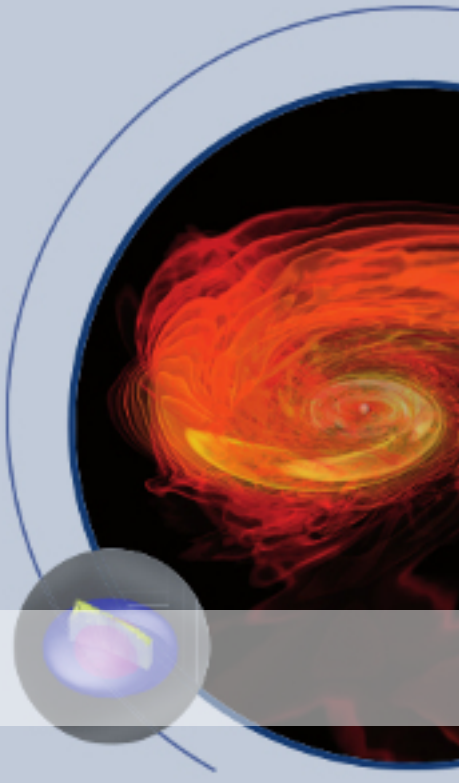
This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 671698.

This particular document was generated on Thursday 14th March, 2019 at 14:27.

PUBLISHED BY THE EXAHYPE DEVELOPERS

EXAHYPE.EU

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 neccessary functions to render 3D scenes, manage the gameplay, add sounds or make units walk over your game board (as a game engine would do), our PDE Engine offers ingredients to solve hyperbolic systems of Partial Differential Equations written in first order form either via explicit ADER-DG or Finite Volume schemes. Users have to 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

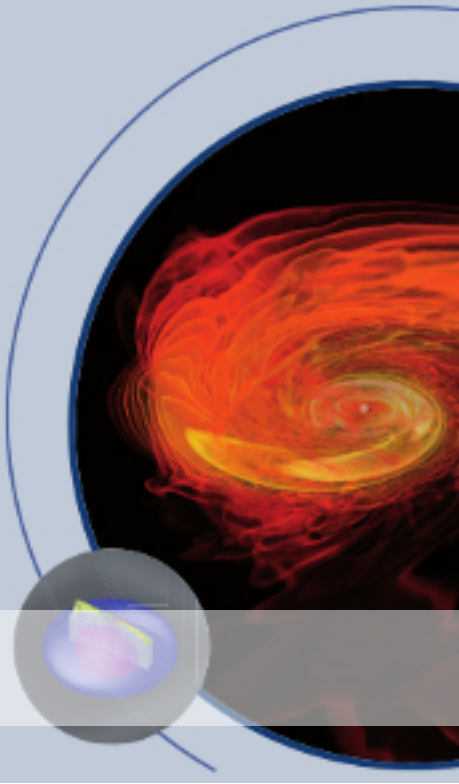
- how to execute some "simple" demonstrators,
- how a new application can be written from scratch,
- how ExaHyPE applications are scaled up,
- and it also details and documents how and why the engine has been designed in certain ways.

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

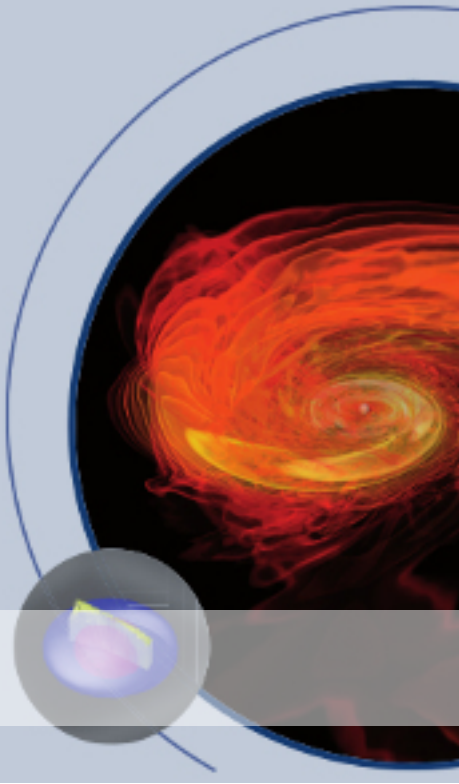
This guidebook is written in a hands-on style. It addresses users of ExaHyPE and developers building new applications within ExaHyPE. Yet, it is not meant to replace the code documentation. Using the guidebook requires a decent background in your application domain modelled via a hyperbolic equation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

ExaHyPE installation and demonstrator applications



1. Setup and Installation

1.1 Dependencies and prerequisites

ExaHyPE comes along with the following dependencies:

- ExaHyPE source code is C++ code. For 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. it is open source and works with GCC and Intel compilers. At least TBB 2017 is required.
- 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 default build environment uses GNU Make.
- ExaHyPE's development environment relies on Python 3. All additional Python 3 dependencies can be downloaded using the git submodules and the provided script.

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.

1.2 Obtaining ExaHyPE

ExaHyPE is available as source code only. We discuss several variants how to obtain the code below. ExaHyPE is built on top of the AMR framework Peano .

Variant 1: Download an ExaHyPE release

Open a browser and go to `http://www.peano-framework.org`. Here, click on `Request Repository Access` and fill out the application form. Click on `Register`. The application will trigger a creation of a LRZ account linked to the Peano and ExaHyPE projects. Eventually, you will get reporter access to the software repositories via `https://gitlab.lrz.de`. You will get notified by mail as soon as this is the case. The process can take a few days.

The ExaHyPE repository can then be cloned via:

```
> git clone https://gitlab.lrz.de/exahype/ExaHyPE-Engine.git
```

or

```
> git@gitlab.lrz.de:exahype/ExaHyPE-Engine.git
```

You can then install all dependencies using

```
 > ./Submodules/updateSubmodules.sh
```

Variant 2: Download an ExaHyPE snapshot (not recommended)

Go to `http://www.peano-framework.org/index.php/exahype/` and download a tar.gz file of a recent release. A snapshot of Peano is included.

If this option is chosen all dependencies must be installed manually.

Obtaining the dependencies

ExaHyPE's development environment relies on a few Python 3 dependencies. They are all registered as git submodule to the project. Peano is also registered as a git submodule

To obtain them, run the update script:

```
> ./Submodules/updateSubmodules.sh
```

Different options to change the default download procedure are listed if you call the script with option "`-h`":

```
> ./Submodules/updateSubmodules.sh -h
-h prints this message
-s set submodules url to ssh
-t use ssh tunnel (port: 12345) and git protocol (works on SuperMUC)
-w set submodules url to https
-p only update the Peano submodule
-o only update submodules other than Peano
```

We ensured that ExaHyPE can be setup on systems where you are required to create an SSH tunnel to `github.com` or `gitlab.lrz.de` in order to clone repositories. If the default port 12345 does not work for you, just change it in the script.

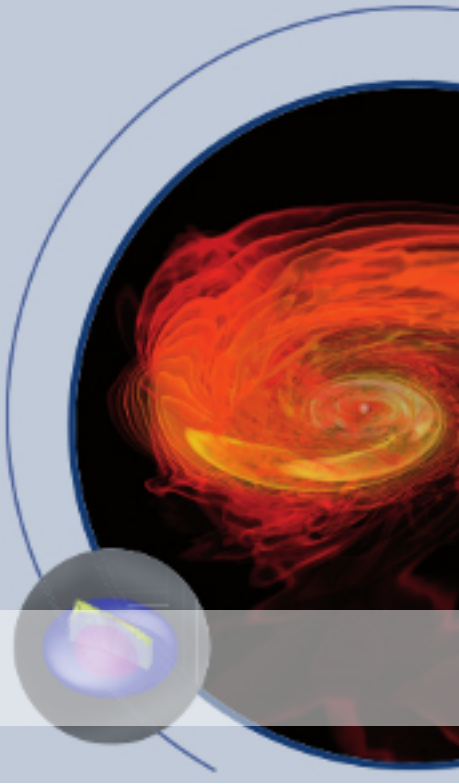
In case you use Peano in several projects, you might want to skip the download of the Peano submodule and instead add two symbolic links to Peano's `peano` and `tarch` directory in the `Peano` subdirectory of ExaHyPE. You further need to create symbolic links to Peano's toolboxes `mpibalancing`, `multiscalelinkedcell`, `sharedmemoryoracles`

1.3 Dry run of development tools

To check whether you are ready to program new applications with ExaHyPE, type in

```
> ./Toolkit/toolkit.sh -h
```

This should give you a description of the various toolkit options. If you encounter an error, please make sure the submodules are downloaded.



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. You find the `EulerFV` in the `Demonstrators` subdirectory.

2.1.1 Preparation

The demonstrator folder 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 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 paths:

```
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
    type const = godunov
    terms const = flux
    optimisation const = generic
    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

```
> ./Toolkit/toolkit.sh \
    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`, if you change the number or type of solvers, or if you change the number of plotters in the specification file.

2.1.2 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:

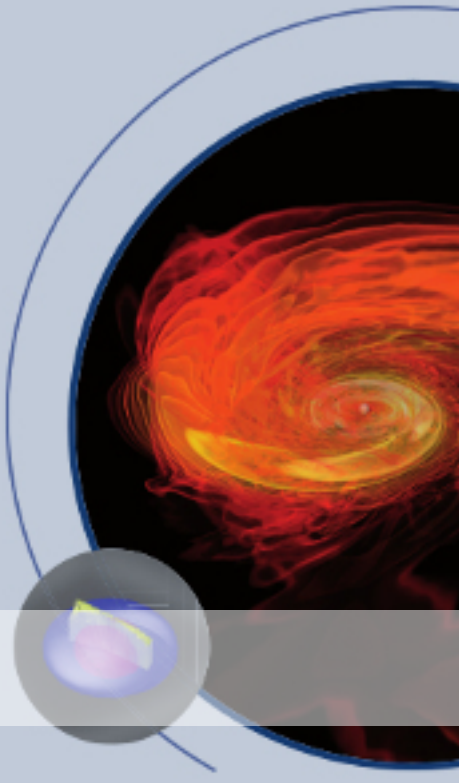
```
> ./ExaHyPE-EulerFV ./EulerFV.exahype
```

A successful run yields a `*.pvd` file that you can open with Paraview[1] or VisIt[2], e.g. There are two quantites plotted: The time encodes per vertex and quantity at which time step the data has been written. For these primitive tests with global time stepping, this information is of limited value. It however later makes a big difference if parts of the grid are allowed to advance in time asynchronous to other grid parts. The second quantity Q is a vector of the real unknowns. The underlying semantics of the unknowns is detailed in Section 4. 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.

[1] `www.paraview.org`
[2] `wci.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/Doxygen syntax, i.e. documentation in multiline C++ comments reads as

```
/**
 *This is an ExaHyPE comment head line.
 *
 *This is the documentation more in detail. You can typically use <em>HTML
 *tags</em> or other kind of text markup like references to other classes.
 *
 */
void someFunction(int a, int b, ...);
```

No important code documentation is found inside the actual source code (`.cpp` files). It is always in the header (`.h` or `.hpp` files). 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?

3.0.1 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 or are no able to access one, using the Doxygen software one can generate/come up with standalone webpages (*HTML* documents which can be opened with any web browser and render as in Figure 3.1) from the sources. We host the documentation online browsable at exahype.eu[1]. Note that doxygen provides, besides hyperlinks and sketches of algorithmic concepts, a search engine.

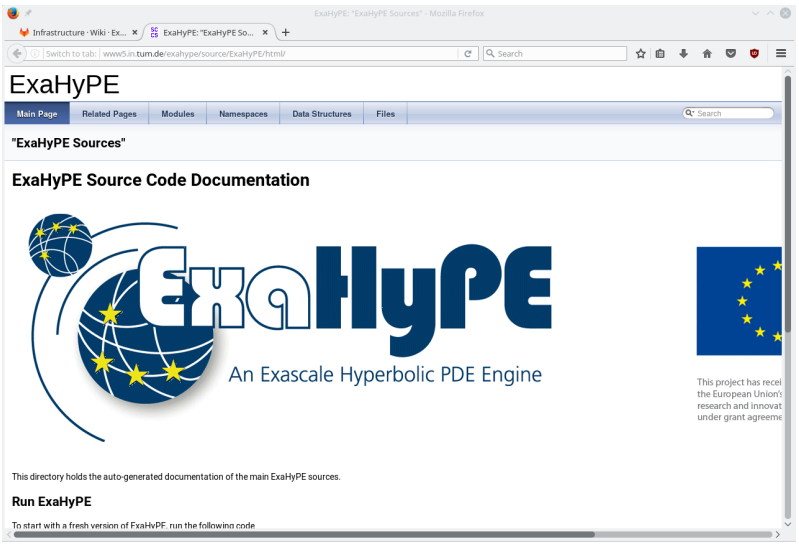


Figure 3.1: The ExaHyPE source code documentation webpage that is updated every night.

ExaHyPE's adaptive meshing is based upon the framework Peano. You find both Peano and its source code documentation (which is also based on Doxygen) at peano-framework.org. The API documentation is especially helpful to understand the `tarch` utilties framework of Peano.

3.0.2 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
```

3.0.3 Seperated parts yield in seperate documentation

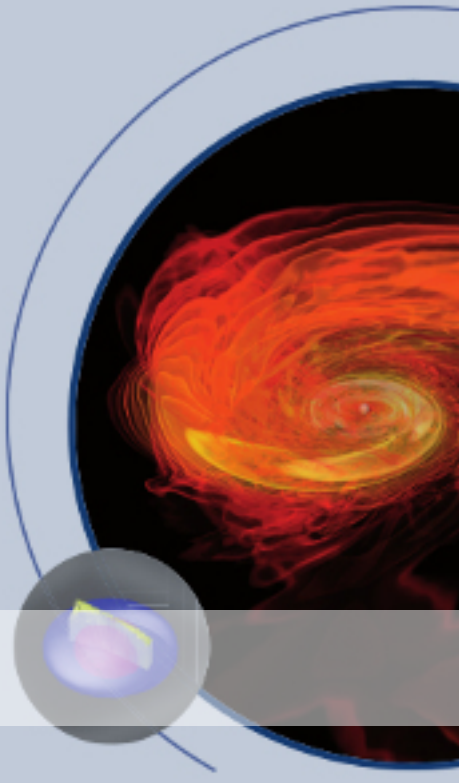
When inspecting the Doxygen documentation, you might notice that by default, no particular application (in sense of some particular PDE such as *EulerFlow*) is included into the documentation. The HTML contains only the pure engine as well as the kernels. It also does not document anything from Peano. 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`.

To recapitulate, you can read off the project structure (c.f. chapter 19) from the documentation structure. It is (c.f. figure 19.1)

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

- The Peano code which seperates into two parts: The actual code `peano` and a utilities library `tarch`. Classes from the `tarch` package/C++ namespace are used all over the place so it is useful if you have a Doxygen build explaining these classes.
- The ExaHyPE engine (also refered to as ExaHyPE *core*) which relies on Peano. This typically goes into your Doxygen but may be quite useless without Peano documentation.
- The ExaHyPE kernels which are rather standalone, they only rely on `tarch`. Again, you want Peano documentations to understand them.
- Your application. You might want to document this if it has grown big or if you want to understand the generated C++ code class hierarchy.
- The ExaHyPE toolkit, explained in detail in chapters 20. The toolkit is written in Python 3. It's code is also documented. However, this documentation will not help you when programming ExaHyPE in C++. Thus we don't go into detail about this part of ExaHyPE.
- Further ExaHyPE code generators such as the Python3 code which generates the optimized kernels, cf. section 15.2. This is one of the many non-mandatory (in terms that you can avoid running them if you don't want to) parts in ExaHyPE which are written in Python and not taken into account at all when it comes to documentation.

II Developing new ExaHyPE applications



4. A new ExaHyPE project

Design philosophy 4.1 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 file[1] is handed over to our ExaHyPE toolkit, a small python 3 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. Depending on the option used, see A, you end up with an executable that may run without the python 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 4.2 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.

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

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

4.1 A trivial project

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

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

  computational-domain
    dimension const = 2
    width = 1.0, 1.0
    offset = 0.0, 0.0
    end-time = 10.0
  end computational-domain
end exahype-project
```

Most parameters should be rather self-explanatory. You might have to adopt the paths[2]. Note that ExaHyPE supports both two- and three-dimensional setups.

We hand the file over to the toolkit

```
./Toolkit/toolkit.sh 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
```

[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.

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

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

4.2 Solver configuration—an overview

Before we run through particular solvers as life examples, we quickly summarise ExaHyPE's main solver concepts. ExaHyPE solves equations of the form

$$\underbrace{\mathbf{P}}_{\texttt{materialmatrix}} \frac{\partial}{\partial t}\mathbf{Q} + \nabla \cdot \underbrace{\mathbf{F}}_{\texttt{fluxes}}(\mathbf{Q}) + \underbrace{\sum_{i=1}^{d} \mathbf{B}_i(\mathbf{Q}) \frac{\partial \mathbf{Q}}{\partial x_i}}_{\texttt{ncp}} = \underbrace{\mathbf{S}(\mathbf{Q})}_{\texttt{sources}} + \underbrace{\sum \delta}_{\texttt{pointsources}}, \tag{4.1}$$

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. Before you start to write your own solver, you have to clarify and specify in the specification file

1. which terms will be used in your formulation? By default, ExaHyPE uses $P = id$ and removes all other terms, i.e. it solves the trivial PDE

   $$\frac{\partial}{\partial t}\mathbf{Q} = 0.$$

   This is, most likely, not what you want.
2. which type of solver and integration scheme do you need? At the moment, we offer a solver for linear equations and a non-linear one based upon Picard iterations. Furthermore, we do support Gauß-Legendre and Gauß-Lobatto integration points for the ADER-DG scheme. Gauß-Legendre allow exact integration in our code and are the default, while Gauß-Lobatto nodes are subject to mass lumping yet are useful sometimes for imposing boundary conditions. The choice of the shape functions is not of interest for Finite Volume schemes.
3. whether you want to employ particular optimisation methods or you prefer a `generic` solver.

While we detail particular solvers in the subsequent chapters, any solver is embedded into a `solver ...end solver` environment and has a unique name. Both ADER-DG and Finite Volume solvers always have three main properties

```
type const = nonlinear
terms const = flux
optimisation const = generic
```

which give answers to the questions raised above.

4.2.1 Solver type

For `type`, the supported values are given in table 4.1.
The order of the optional features is irrelevant. Technically, ExaHyPE allows you to pass non-linear PDEs over to a linear solver, as you define used PDE terms independent of the solver type, though this does not really make sense.

ADER-DG	
`linear`	Is a shortcut for `linear, Legendre`.
`nonlinear`	Is a shortcut for `nonlinear, Legendre`.
`linear, Legendre`	Kernel for linear PDEs solved on Gauss-Legendre-Nodes
`nonlinear, Legendre`	Kernel for nonlinear PDEs, solved on Gauss-Legendre-Nodes
`linear, Lobatto`	Kernel for linear PDEs solved on Gauss-Lobatto-Nodes
`nonlinear, Lobatto`	Kernel for nonlinear PDEs solved on Gauss-Lobatto-Nodes
Finite Volumes	
`musclhancock`	Use a MUSCL-Hancock Riemann solver.
`robustmusclhancock`	Use a slightly more robust version of the MUSCL-Hancock Riemann solver.
`godunov`	Use a standard Godunov Riemann solver.

Table 4.1: Solver types. See also figure 21.2 for a comparison of the different nodal basis grids.

4.2.2 Solver PDE terms

For `terms`, the supported values are given in table 4.2.

`materialparameters`	If this PDE term is present, ExaHyPE allows the user to specify a spd matrix P. If it is not present, ExaHyPE uses $P = id$.
`fluxes`	Informs ExaHyPE that you want to use a standard, conservative first-order flux formulation.
`ncp`	Informs ExaHyPE that you plan to use non-conservative formulations.
`sources`	Informs ExaHyPE that you plan to use algebraic sources, i.e. right-hand side terms.
`pointSources`	Informs ExaHyPE that you plan to use point sources, i.e. Dirac distribution right-hand side terms.

Table 4.2: Supported solver PDE terms

The names correspond to the identifiers in (4.1). The order of the optional features is irrelevant. Depending on the chosen terms, the ExaHyPE toolkit will create C++/FORTRAN user classes that have to provide implementations for these terms.

R The $\mathbf{B}_i$, δ and S term all may depend on the solution Q, i.e. we do not restrict to particular linear term flavours.

R The `ncp` signatures of ExaHyPE allows users to write code that technically adds a term $\mathbf{B} \cdot \nabla \mathbf{Q}$, i.e. a source term to the PDE as part of the non-conservative term. While this formally seems to be possible, it is usually not a good idea to incorporate sources into the ncp formulation. Sources are volume terms, i.e. do influence ADER-DG's space-time predictors but not any Riemann solvers. The ncp term however enters the Riemann solve. If you incorporate algebraic sources into ncp, they thus enter the Riemann solve and thus yield (usually) invalid solution jump contributions.

Note that we discuss advanced solver features in section 9, notably there is a subsection about non-conservative systems (section 9.7) and point sources (section 9.7).

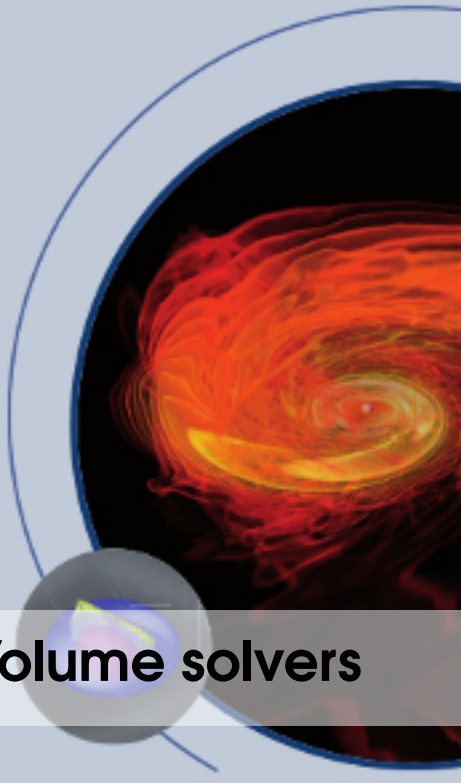
4.2.3 Solver optimisations

For `optimisation`, the supported values are given in table 4.3.

ADER-DG	
`generic`	This is our generic implementation of Finite Volumes. This is the default if no optimisation is specified.
`optimised`	todo JM
`usestack`	Put temporary variables such as the space-time predictor array on the stack. This might require you to adjust the stack size per thread; see Table 13.3 for more details.
`maxpicarditer:<int>`	Fix the number of Picard iterations the space-time predictor kernel performs.
Finite Volumes	
`generic`	This is our generic implementation of Finite Volumes. This is the default if no optimisation is specified.
`optimised`	todo JM
`usestack`	Put temporary variables such as arrays required for imposing boundary conditions on the stack. array on the stack. This might require you to adjust the stack size per thread; see Table 13.3 for more details.

Table 4.3: Solver optimisations

If you switch to Finite Volumes, not all of the optional features might be available.



5. Generic ADER-DG and Finite Volume solvers

In our prevous chapter, the simulation run neither computes anything nor does it yield any output. In this chapter, we introduce ExaHyPE generic kernels: they allow the user to specify flux and eigenvalue functions for the kernels, but delegate the actual solve completely to ExaHyPE. Furthermore, we quickly run through some visualisation and the handling of material parameters. The resulting code can run in parallel and scale, but its single node performance might remain poor, as we do not rely on ExaHyPE's high performance kernels. Therefore, the present coding strategy is particulary well-suited for rapid prototyping of new solvers.

Design philosophy 5.1 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 equipts ExaHyPE with adaptive mesh refinement. Our guideline illustrates all of these steps at hands of a solver for the Euler equations.

5.1 Establishing the data structures

We follow the Euler equations in the form

$$\frac{\partial}{\partial t}\mathbf{Q} + \nabla \cdot \mathbf{F}(\mathbf{Q}) = 0 \quad \text{with} \quad \mathbf{Q} = \begin{pmatrix} \rho \\ \mathbf{j} \\ E \end{pmatrix} \quad \text{and} \quad \mathbf{F} = \begin{pmatrix} \mathbf{j} \\ \frac{1}{\rho}\mathbf{j} \otimes \mathbf{j} + pI \\ \frac{1}{\rho}\mathbf{j}(E+p) \end{pmatrix} \tag{5.1}$$

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

$$p = (\gamma - 1)\left(E - \frac{1}{2}\mathbf{j}^2/\rho\right). \tag{5.2}$$

Our example restricts to

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

as complicated domains are subject of discussion in a separate Chapter 11. A corresponding specification file `euler-2d.exahype` for this setup is

```
exahype-project Euler2d

  peano-kernel-path const = ./Peano/
  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
    type const = nonlinear
    terms const = flux
    optimisation const = generic
    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
```

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 `nonlinear`. We will not employ any `optimisation`. Instead we use some generic 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 eigenvalues) definitions. Here, we tell the toolkit that we want to specify a conservative flux via parameter `terms`. Other options are possible (Sec. 4.2).

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:

```
./Toolkit/toolkit.sh Euler2d.exahype
```

Design philosophy 5.2 The formulation (5.1) 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 9.5 details the usage of material parameters.

5.2 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:

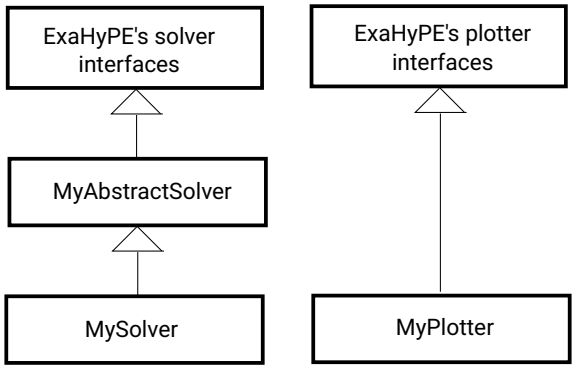


Figure 5.1: Simplest class architecture for ExaHyPE solvers.

- `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.

5.3 Working with the arrays

Design philosophy 5.3 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 performance.

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

Warning 5.1 Working with symbolic identifiers may degrade performance!

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

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

This specification is exactly equivalent to the previous declaration. It tells ExaHyPE that there are five unkonwns held in total. Two of them are plain scalars, the middle one is a vector with

three entries. Also, all operation signatures remain exactly the same. Yet, the toolkit now creates a couple of additional classes that can be wrapped around the array. These classes do not copy any stuff around or themselves actually alter the array. They provide however routines to access the array entries through their unknown names instead of plain array indices:

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

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

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

We note that there are subtle 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 5.4 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.

5.4 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 a routine `adjustPointSolution` that allow us to setup the initial grid. Alternatively, we can also use `adjustSolution`. One routine works point-wisely, the other one hands over a whole patch. The implementation of the initial values might look as follows[1]:

```
void Euler2d::MyEulerSolver::adjustPointSolution(
  const double *const x,
```

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

```
  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.

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 5.2).

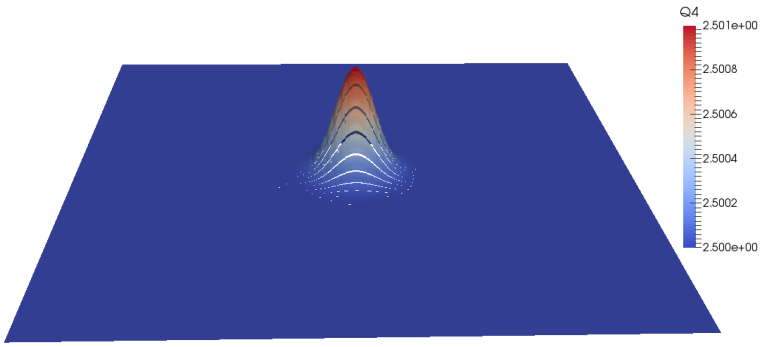


Figure 5.2: The rest mass density Q_0 in the EulerFlow 2D setup. The scalar value is encoded both in height and color. So far, the PDE has not been specified so it stays constant during the time evolution.

Design philosophy 5.5 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.

5.5 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

Warning 5.2 Working with symbolic identifiers may degrade performance!

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 sub-

tle: 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`.

5.6 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 straightforwardly[2] implement

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

The signature to implement your boundary conditions reads

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

For now we simply set stateOut = stateIn and fluxOut = fluxIn. A rebuild and rerun should yield results similar to figure 5.3.

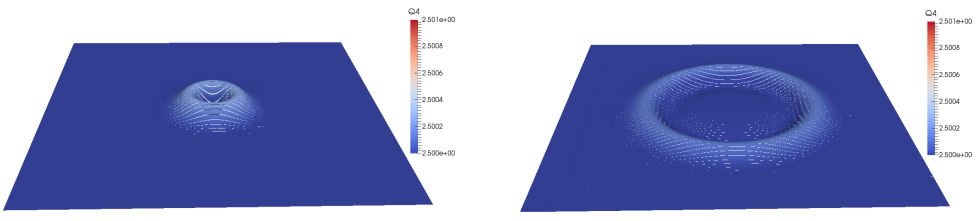


Figure 5.3: Time evolution of Q_0, now with the Hydrodynamics PDEs implemented. The left figure shows an early time while the right figure shows a later time.

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

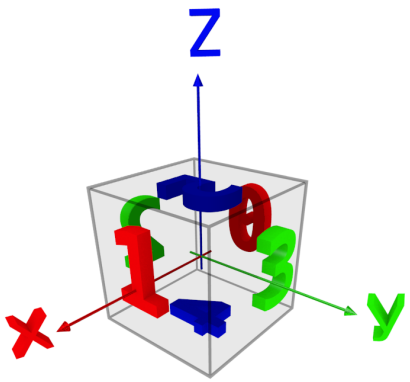


Figure 5.4: The face indexes as named in the ExaHyPE code

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
```

This is also encoded in Figure 5.4.

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 5.5); 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.

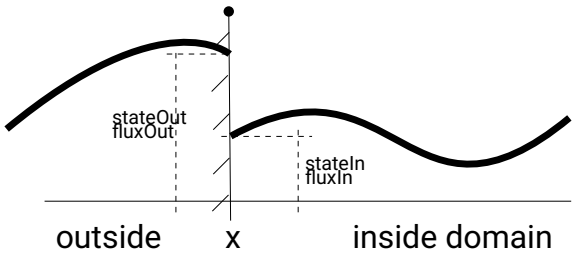


Figure 5.5: 1-dimensional sketch of boundary conditions in ExaHyPE's ADER-DG. The state inside the domain and the fluxes from this side are passed to the boundary treatment by the solver. They are therefore const. A boundary condition is imposed through prescribing the state outside of the domain (thus 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.

5.6.1 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 Solver's 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`.

5.6.2 Outflow Boundary Conditions

As another example, outflow boundary conditions are archieved 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 5.6 We stick to $\Omega = (0,1)^d$ without MPI here. Please read carefully through Chapter 11 if you need more sophisticated boundary conditions or if precise boundary conditions form an essential part of your simulation.

5.7 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:

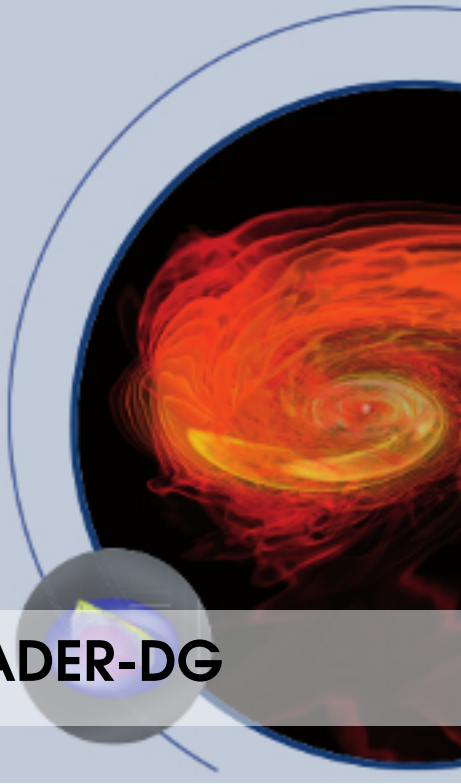
```
solver Finite-Volumes MyFVEulerSolver
  variables = rho:1,j:3,E:1
  patch-size = 3
  maximum-mesh-size = 0.1
  time-stepping = global
  kernel = generic::godunov
  language = C
end solver
```

This needs an update.

Design philosophy 5.7 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 $2 \cdot$ `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. 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 5 first. Different to Chapter 5 we focus on a few particular challenges tied to the shallow water equations:

1. How we quickly write down 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

$$\frac{\partial}{\partial t}\begin{pmatrix} h \\ hu \\ hv \\ b \end{pmatrix} + \frac{\partial}{\partial x}\begin{pmatrix} hu \\ hu^2 + 0.5gh^2 \\ huv \\ 0 \end{pmatrix} + \frac{\partial}{\partial y}\begin{pmatrix} hv \\ huv \\ hv^2 + 0.5gh^2 \\ 0 \end{pmatrix} + \begin{pmatrix} 0 \\ hg \cdot b_x \\ hg \cdot b_y \\ 0 \end{pmatrix} = 0. \tag{6.1}$$

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.

6.1 Setting up the experiment

Fluxes and non-conservative products will be required and therefore the solver's kernel is `generic::fluxes::ncp::nonlinear`. 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
    type const = nonlinear
    terms const = flux,ncp
    optimisation const = generic
    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.

6.2 Writing out the data

While we could plot out **Q** as it is, it is convenient, for the shallow water equations, to see the water height rather than the entries in **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 **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`.

6.3 The (non-bathymetric) fluxes

The realisation of the eigenvales 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();
}
```

6.4 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 (6.1) as a PDE over four unknowns.
- Following exactly the notation from (6.1), 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 6.1 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 (6.1) into

$$\frac{\partial}{\partial t}\mathbf{Q} + \frac{\partial}{\partial x}\mathbf{F}(\mathbf{Q}) + \frac{\partial}{\partial y}\mathbf{G}(\mathbf{Q}) + \boldsymbol{B}_1(\boldsymbol{Q})\frac{\partial \boldsymbol{Q}}{\partial x} + \boldsymbol{B}_2(\boldsymbol{Q})\frac{\partial \boldsymbol{Q}}{\partial y} = 0 \tag{6.2}$$

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 the non-conservative product:

1. As we informed ExaHyPE that we wanted to use the non-conservative products, the header should already 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;
```

2. 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;
}
```

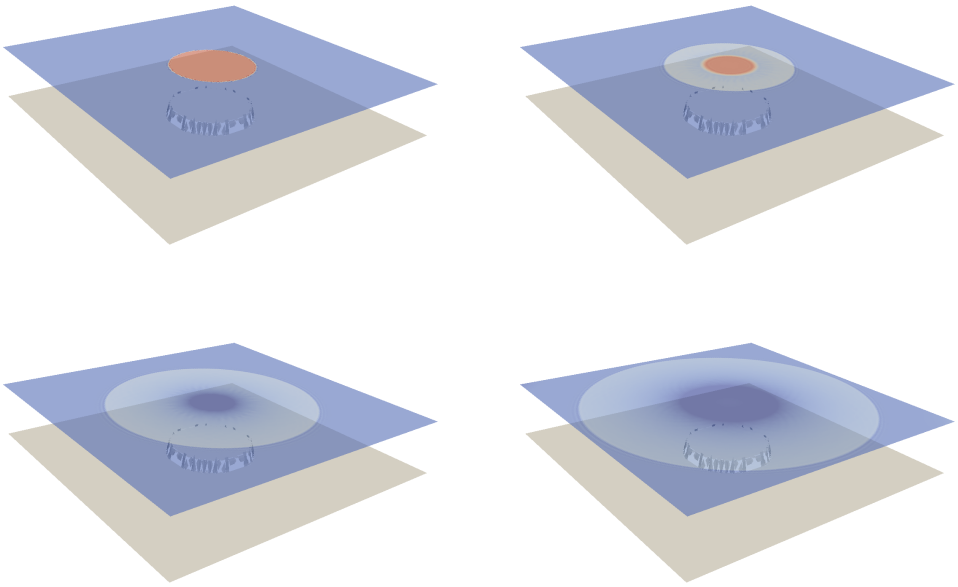
   In the above code snippet, we used the `idx2` struct in `kernels/KernelUtils.h` to access the derivatives.

3. 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.



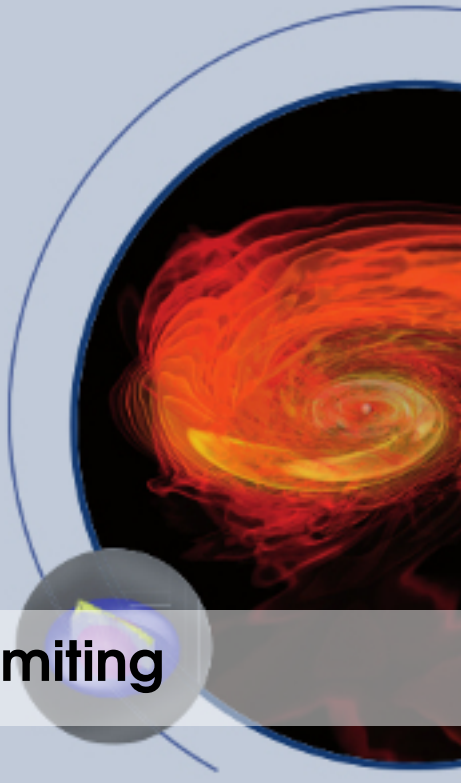
Left: A ocean with a plain subsurface hosting a circular plateau is covered by constant constant water height, i.e. the water mirrors the subocean topology. To the right: Once we "release" this initial condition, we see waves spreading.

6.5 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.

Note 6.1 The current Rusanov solver is not well-balanced which results in spurious waves for some scenarios. A simple fix to make it well-balanced for the SWE adds $0.5 * s_{max} * (b_L - b_R)$ to the first component of the Rusanov flux.



7. ADER-DG with Finite Volumes limiting

The ADER-DG scheme with Finite Volumes 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 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 const = rho:1,j:3,E:1
  order const = 3
  maximum-mesh-size = 0.03704
  time-stepping = global
  type const = nonlinear
  terms const = flux
  optimisation const = generic
  language const = C
  limiter-type const = musclhancock
  limiter-optimisation const = generic
  limiter-language const = C
  dmp-observables = 5
  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 (`type`, `optimisation`, `language`) and one for the Finite Volumes method employed for the limiting (`limiter-typpe`, `optimisation`,

`limiter-language`). Both kernels will use thes same `terms`. Of couse, we can choose any of the kernels that are available for the pure ADER-DG or pure Finite Volumes method.

The last two parameters, `dmp-relaxation-parameter` and `dmp-difference-scaling`, are tuning parameters to relax a discrete maximum principle (DMP). The DMP plays a key role role in detecting troubled cells – cells that need limiting.

The toolkit will then generate implementation files for one ADER-DG solver and for one Finite Volumes solver. No additional implementation files for the limiting ADER-DG solver itself are generated. Instead, the ADER-DG and Finite Volumes solver are coupled with each other in the `initSolvers(...)` routine in file `KernelCalls.cpp`

7.1 Identifying troubled cells

The values that must satisfy the discrete maximum principle are selected via a function `mapDiscreteMaximumPrincipleObservables`. We extend `MySolver_ADERDG.h` file by the following function declaration:

```
MySolver\_ADERDG.h:
[..]
void mapDiscreteMaximumPrincipleObservables(
    double* observables,const int numberOfObservables,
    const double* const Q) const override;
[..]
```

An implementation that picks all variables out of the nodal solution `Q` might then look as follows:

```
MySolver\_ADERDG.cpp:
[..]
void MyProject::MySolver_ADERDG::mapDiscreteMaximumPrincipleObservables(
    double* observables,const int numberOfObservables,
    const double* const Q) const {
  for (int i=0; i<NumberOfVariables; ++i) {
    observables[i] = Q[i];
  }
}
[..]
```

Furthermore, it is important to test the numerical solution for physical admissibility, e.g., if a density is positive or if it is an actual finite number, e.g. not "NaN" or ± "Inf". The troubled cell indicator of ExaHyPE's ADER-DG scheme with Finite Volumes limiting thus does not only rely on the DMP but makes use of an physical admissbility detection (PAD). In contrast to the DMP, the PAD is also applied to the initial solution to build up the initial domains of the coupled FV and ADER-DG solvers. The troubled cell detection will consider the PAD if you extend your `MySolver_ADERDG.h` and `MySolver_ADERDG.cpp` files by the following function declaration and a proper definition:

```
MySolver_ADERDG.h:
[..]
bool isPhysicallyAdmissible(
    const double* const solution,
```

```
    const double* const observablesMin,const double* const observablesMax,
    const int numberOfObservables,
    const tarch::la::Vector<DIMENSIONS,double>& center,
    const tarch::la::Vector<DIMENSIONS,double>& dx,
    const double t, const double dt) const override;
[..]
```

```
MySolver_ADERDG.cpp:
[..]
bool MyProject::MySolver_ADERDG::isPhysicallyAdmissible(
    const double* const solution,
    const double* const observablesMin,const double* const observablesMax,
    const int numberOfObservables,
    const tarch::la::Vector<DIMENSIONS,double>& center,
    const tarch::la::Vector<DIMENSIONS,double>& dx,
    const double t, const double dt) const {
  // This is an example for the compressible Euler equations.
  // Modify it according to your needs.
  if( !std::isfinite(observablesMin[0] ) // always check for NaNs!
     return false;
  if( !std::isfinite(observablesMin[4] )
     return false;
  if( !std::isfinite(observablesMax[0] )
     return false;
  if( !std::isfinite(observablesMax[4] )
      return false;
  if (observablesMin[0] <= 0.0) return false; // (min) density should not be
                                              // negative
  if (observablesMin[4] < 0.0) return false; return true;
                                              // (min) energy should
                                              // not be negative
}
[..]
```

7.2 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.

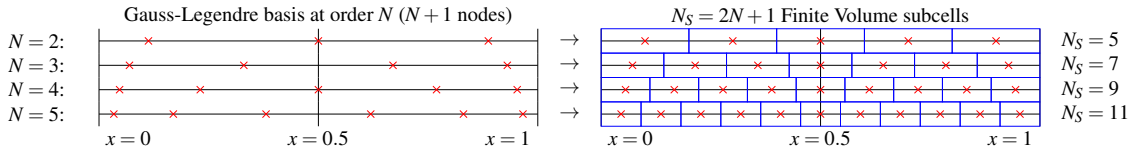


Figure 7.1: *Left*: Node locations in the reference one-dimensional cell of the ADER-DG scheme. The coordinate $x \in \{0,1\}$ covers the computational cell holding N degrees of freedom, where $N+1$ is the order of the method. *Right*: representation of the finite-volume subcell structure, with each cell being divided in a set of $2N+1$ subcells. Taken from [1].

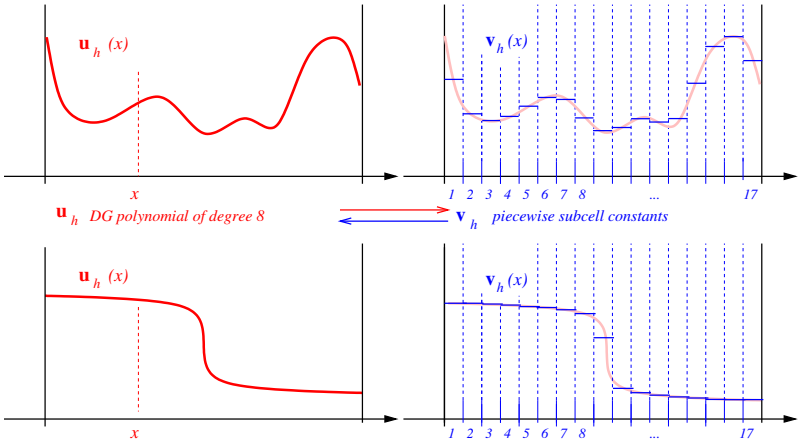


Figure 7.2: Projection to and recovery from the finite volume solver with $2N+1$ subcells for an polynomial order N ADER-DG polynomial. The figure shows one cell with $N = 8$. Taken from [2].

7.3 Understanding the subcell structure

In some applications, the positions where the actual field values are stored or requested are important, for instance if singularities in fields have to be taken into account or special geometries shall be implemented. We refer to chapter 21 for details on the cell structure. For details how cells are plotted, see section 17.1.

Figure 7.1 demonstrates the grid setting for a single reference cell but several polynomial orders N. One can clearly see the vertex-centered DG basis which translates to a completely different grid in the finite volume regular subgrid. There, the finite volume cells stand out in blue and the cell centers are indicated once again.

The actual projection and prolongation between the “coarse” DG cell and the “finer” FV cell is depicted in figure 7.2. It takes place in LimitingADERDG solver in the kernels.

7.4 Hands on: Implementing a limited solver

This section shall give a few tips how to solve a PDE with the LimitedADERDG framework of ExaHyPE. We refer to the FAQ chapter B in the appendix for general tips.

Implement your actual PDE in a seperate file. You have to implement two solvers when you want to use ExaHyPE’s LimitingADERDG framework. Do not repeat yourself: Write the actual PDE in functions which you store in a seperate (C or Fortran) file and call them by the solver files. This attemp degrades your actual solver as an API between your PDE code and ExaHyPE.

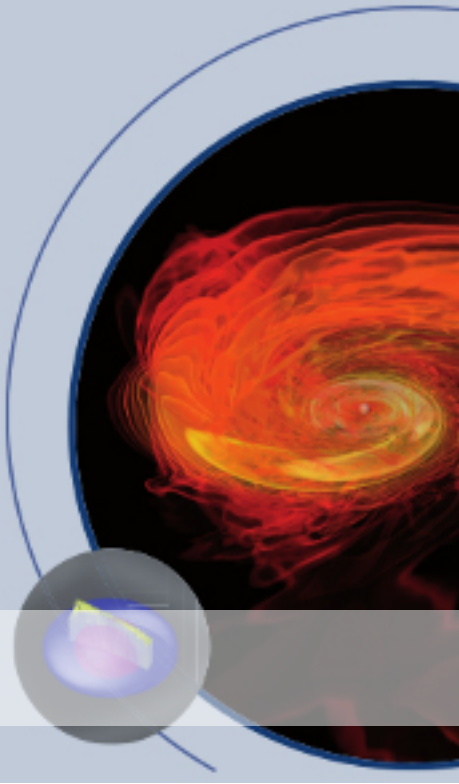
Start with an application and solver name `YourSolver_ADERDG`**.** This is helpful if you have

smooth benchmarks for your PDE. If you are more experienced with FV, start instead with an application named `YourSolver_FV`. Test both FV and ADERDG seperatedly.

Label your solvers `YourSolver_FV`, `YourSolver_ADERDG` and `YourSolver` for the LimitedADERDG. By doing so, you don't have to copy the whole application three times but instead can work in a single directory. Once you know that your individual solvers work, you can start activating the coupling.

Develop always with GCC, without any optimization or parallelization. This makes debugging easy. You should make sure you detect NANs and problems early and be able to use your debugger.

Only change one thing at a time. Especially, do not change your PDE properties (such as variable names, size of variables, usage of constants) while trying to activate further solvers. Otherwise, it will be hard to understand the compiler errors you obtain. In case of trouble, it can be helpful to start with a specification file from the scratch and compare the generated code with the code in your application directory (for instance with a graphical `diff` tool).



8. 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. 5). 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.

8.1 Prerequisites

In the following sections, we assume that the reader has performed all steps in Ch. 5 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
    type const = nonlinear
    terms const = flux
    optimisation const = generic
    language const = C
    ..
```

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

8.2 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 $[0.333, 0.666] \times [0.333, 0.666]$, cf. figure 8.1. To facilitate the desired adaptive mesh refinement,

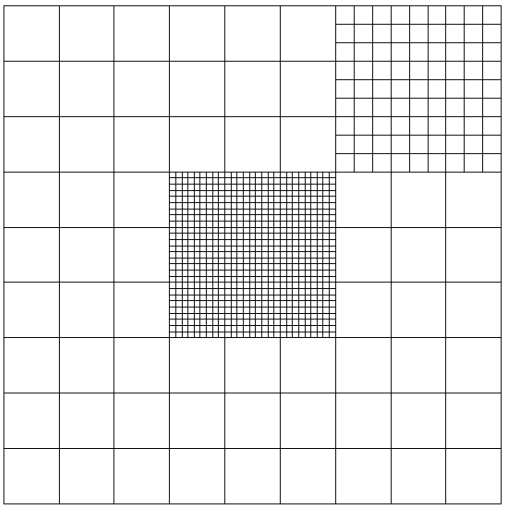


Figure 8.1: Exemplary static mesh refinement

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

  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.

8.3 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`

8.4 Limiter-guided static adaptive mesh refinement

Note 8.1 Limiter-based refinement works only for solvers of type `Limiting-ADER-DG` (see section 7). They utilise the troubled cell indicator routines which are ignored for the pure `ADER-DG` solver.

We start here again with prescribing a static refinement criterion. However, this time we will not utilise the `refinementCriterion` routine but we will (ab)use the physical admissibility detection (PAD)criterion `isPhysicallyAdmissible`.

Design philosophy 8.1 In ExaHyPE, we enforce the constraint that limiting is only performed on the finest level of the adaptive mesh. Should a cell be marked as troubled on the coarsest mesh level, it will be refined down to the finest user-defined mesh level. However, not just this cell is refined. Also a certain number of neighbouring cells. This is necessary since we need to place also the FV and DG helper cells on the finest level of the mesh.

If we now mark a certain area as troubled by means of the PAD criterion, this will result in mesh refinement as well as in placing FV volume patches in the marked areas.

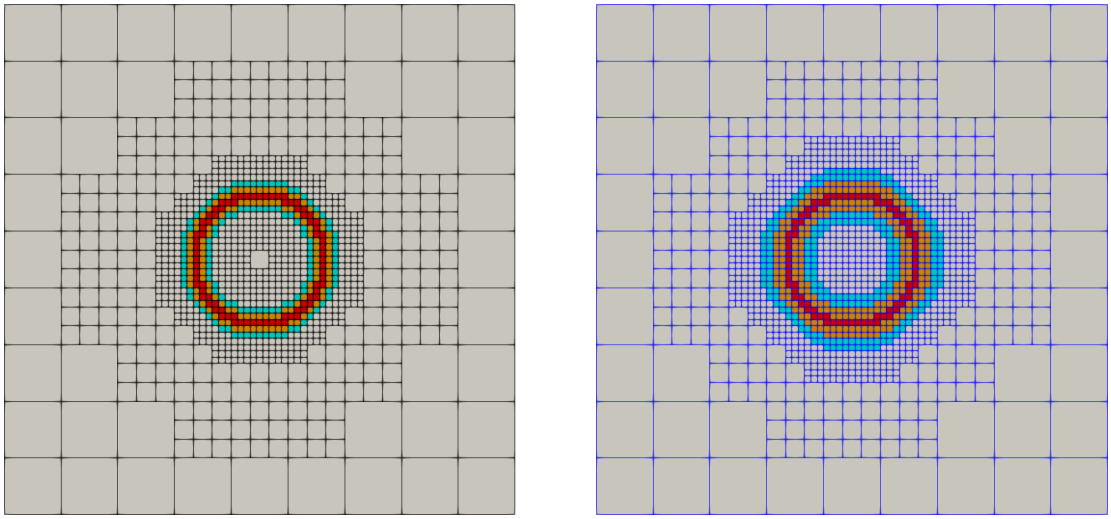
In our example, we mark all cells on the circle or sphere $\|\mathbf{x}-0.5\|_2 = 0.25$ as troubled. We detect these cells by counting vertices inside and outside of the circle. Cells which are cut by the circle have inside (and outside) vertex counts which are larger than zero but smaller than 2^d.

```
MySolver_ADERDG.cpp:
[..]
bool MyProject::MySolver_ADERDG::isPhysicallyAdmissible(
    const double* const solution,
    const double* const observablesMin,const double* const observablesMax,
    const int numberOfObservables,
    const tarch::la::Vector<DIMENSIONS,double>& center,
    const tarch::la::Vector<DIMENSIONS,double>& dx,
    const double t, const double dt) const {
  int outsidePoints = 0;

  tarch::la::Vector<DIMENSIONS,double> offset = center-0.5*dx;
  dfor2(p)
    tarch::la::Vector<DIMENSIONS,double> corner=offset;
    #if DIMENSIONS==3
    corner[2]+=p[2]*dx[2];
    #endif
    corner[1]+=p[1]*dx[1];
    corner[0]+=p[0]*dx[0];

    outsidePoints += (tarch::la::norm2(corner)>rad) ? 1 : 0;
  enddforx

  if (outsidePoints>0 && outsidePoints<TWO_POWER_D) {
    return false;
  }
  else {
  return true;
  }
}
[..]
```

This yields the mesh and limiter domain depicted in figure 8.2.

Design philosophy 8.2 We distinguish between limiter-based refinement during the imposition of the initial solution and limiter-based refinement during the time stepping. For the former, we evaluate again after each refinement if the new cells' ADER-DG (initial) solution is troubled. For the latter, this check is not performed. Here, all children of a refined cell are marked as troubled and thus compute with FV.

The selection of a value of the optional `Limiting-ADER-DG` solver parameter `helper-layers` larger than 1 results in more halo refinement around the marked troubled cells. This is necessary for placing the additional helper cells around the troubled cells. In order to produce the picture below, we have set `helper-layers` to 2, cf. figure 8.2b.



(a) One helper layer (b) Two helper layers

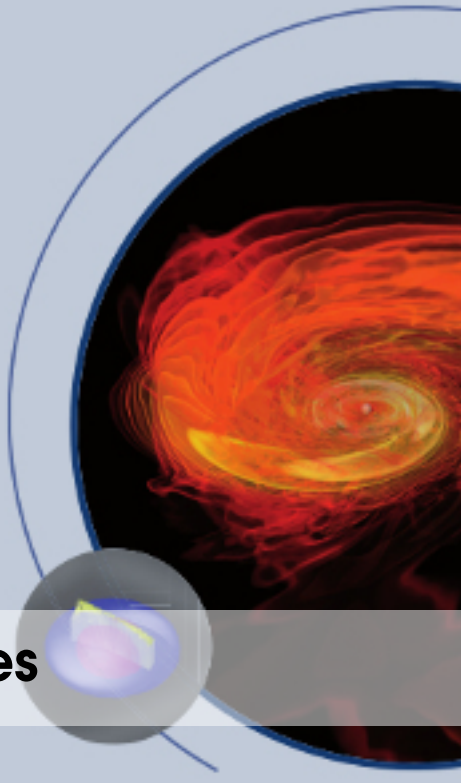
Figure 8.2: Actual troubled cells are marked in red, pure ADER-DG cells are marked in gray, the DG helper cells are marked in light-blue, and the FV helper cells are marked in orange.

8.5 Limiter-guided dynamic adaptive mesh refinement

We start again with the `Limiting-ADER-DG` solver application from Ch. 7. Here, we have already implemented the routines `isPhysicallyAdmissible` and `mapDiscreteMaximumPrincipleObservables`. Both routines have a say in the process of indicating troubled cells.

For running the simulation with dynamic AMR, the only thing that is left is to switch on ExaHyPE's adaptive mesh refinement by providing a `maxium-mesh-depth` > 0. We thus augment the original solver definition 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
  maximum-mesh-depth = 2 // maximum depth of the adaptive mesh
  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
[..]
```



9. Some advanced solver features

9.1 Multiple solvers in one specification file

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

9.2 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
  type const = nonlinear
  terms const = flux
  optimisation const = generic
  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 9.1 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.

9.3 The init function

The ExaHyPE toolkit equipps every solver with a function `init` which is called immediately after the solver construction, ie. 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 5.4). Users may want to extend their solver class to store prepared initial data, if neccessary.

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 << "-␣" << arg << "\n";
  }
}
```

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

global

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

globalfixed

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

9.5 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
  type const = nonlinear
  terms const = flux
  optimisation const = generic
  language const = C
end solver
```

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

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

Design philosophy 9.2 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.

9.6 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`.

9.7 Non-conservative formulations

Most of this document considered only how to implement strongly hyperbolic conservation laws, ie. PDEs which can be casted to the form

$$\frac{\partial}{\partial t}\mathbf{Q} + \nabla \cdot \mathbf{F}(\mathbf{Q}) = \boldsymbol{S}(\boldsymbol{Q}). \tag{9.1}$$

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

$$\frac{\partial}{\partial t}\mathbf{Q} + \nabla \cdot \mathbf{F}(\mathbf{Q}) + \sum_i \boldsymbol{B}_i(\boldsymbol{Q}) \frac{\partial \boldsymbol{Q}}{\partial x_i} + \boldsymbol{B}_0(\boldsymbol{Q})\,\boldsymbol{Q} = \boldsymbol{S}(\boldsymbol{Q}). \tag{9.2}$$

with matrices $\boldsymbol{B}_i(\boldsymbol{Q})$ per space dimension i. The $\boldsymbol{B}_i$ are used in the ADER-DG and FV schemes in the Riemann solver and in the space-time predictor. Pleae compare these formulations also with the generic solver structure as proposed in section 4.2.

Design philosophy 9.3 By default, we adopt a Riemann solver which anticipates the non-conservative fluxes. Therefore, we do not allow source terms $\boldsymbol{S}(\boldsymbol{Q}, \nabla \boldsymbol{Q})$ but split such a source term into the *algebraic source* $\boldsymbol{S}(\boldsymbol{Q})$ and the *non-conservative* source (or non-conservative product) $\boldsymbol{B} \cdot \nabla \boldsymbol{Q}$.

9.7.1 Signatures for the non-conservative product

The special treatment of the non-conservative product in the ADER-DG scheme implemented in ExaHyPE requires to specify the non-conservative product explicitely. We experimented so far with 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,

However, note that in the generic nonlinear kernels, we decided to use only the `NCP` variant. Users therefore are asked to provide the result vector $\boldsymbol{B} \cdot \nabla \boldsymbol{Q}$.

9.7.2 The fused Source

In many PDE systems, the computation of the non-conservative product as well as the algebraic source is expensive. In the scheme, at one point both quantities are just added to a *right hand side* source term $\boldsymbol{S} - \boldsymbol{B} \cdot \nabla \boldsymbol{Q}$ which we refer to as the *fused source*.

Design philosophy 9.4 In ExaHyPE, the user PDE application interface shall be *minimalistic*. We just do not support to implement a fused source with the generic kernels. It is subject to optimised kernels to exploit the PDE structure and skip unneccessary PDE calls.

9.8 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.

This section is still under construction. Kduru or the Seismic people might add some text to it.

9.9 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.

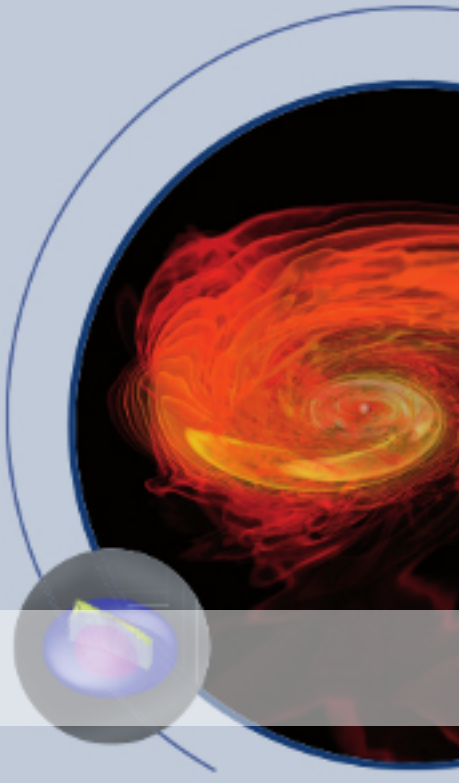
Start with a spec file that asks for a standard Riemann solver, such as `type=godunov` and then overwrite the Riemann solvers implementation in the abstract solver interface in the user solver.

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.

As an example we provide here an implementation of the Rusanov flux (as it is currently implemented in ExaHyPE) for a simple system without non-conservative products or source terms.

```
double riemannSolver(double* fL, double *fR,
                     const double* qL, const double* qR,
                     int direction) {
 constexpr int numberOfVariables = SolverType::NumberOfVariables;
 constexpr int numberOfParameters = SolverType::NumberOfParameters;
 constexpr int numberOfData = numberOfVariables + numberOfParameters;

 double sL[numberOfVariables];
 double sR[numberOfVariables];
 solver.eigenvalues(qL, normalNonZero, sL);
 solver.eigenvalues(qR, normalNonZero, sR);

 double s_max = -1.0;
 for (int i = 0; i < numberOfVariables; i++) {
   const double abs_sL_i = std::abs(sL[i]);
   s_max = std::max( abs_sL_i, s_max );
 }
 for (int i = 0; i < numberOfVariables; i++) {
   const double abs_sR_i = std::abs(sR[i]);
   s_max = std::max( abs_sR_i, s_max );
 }

 double FL2[DIMENSIONS][numberOfVariables] = {0.0};
 double FR2[DIMENSIONS][numberOfVariables] = {0.0};
 double* FL[DIMENSIONS]={FL2[0], FL2[1]};
 double* FR[DIMENSIONS]={FR2[0], FR2[1]};
 solver.flux(qL, FL);
 solver.flux(qR, FR);

 for (int i = 0; i < numberOfVariables; i++) {
   fL[i] = 0.5 *s_max *(qL[i] -qR[i])
               + 0.5 *(FL2[normalNonZero][i] + FR2[normalNonZero][i]);
   fR[i] = fL[i];
 }
 return s_max;
}
```

Provide a better example, for instance an Osher-type Riemann Solver.



10. 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).

10.1 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. lowercased 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 unkown 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 10.1: Parameters for the PDE solvers, in Fortran notation, cf. also appx. **??**.

10.2 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 10.1 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:

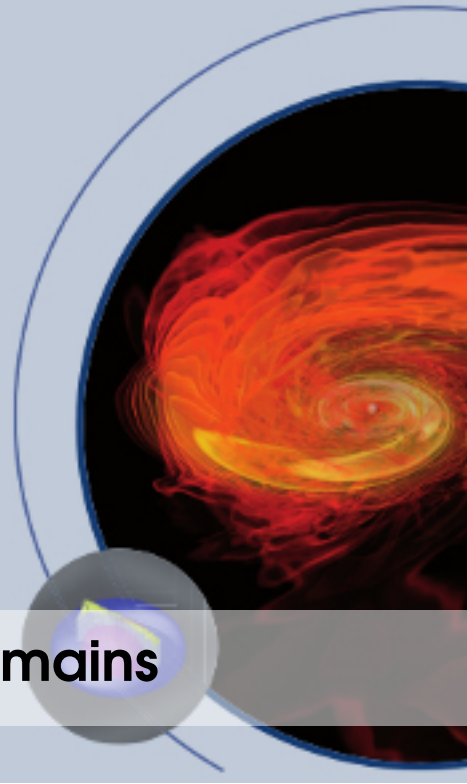
```
MODULE Parameters
    IMPLICIT NONE
    PUBLIC

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

10.3 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.

10.4 Further reading

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



11. Non-trivial computational domains

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

Design philosophy 11.1

1. If you do not realise a sophisticated domain handling—the ExaHyPE consortium favours immersed boundary techiques—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.!

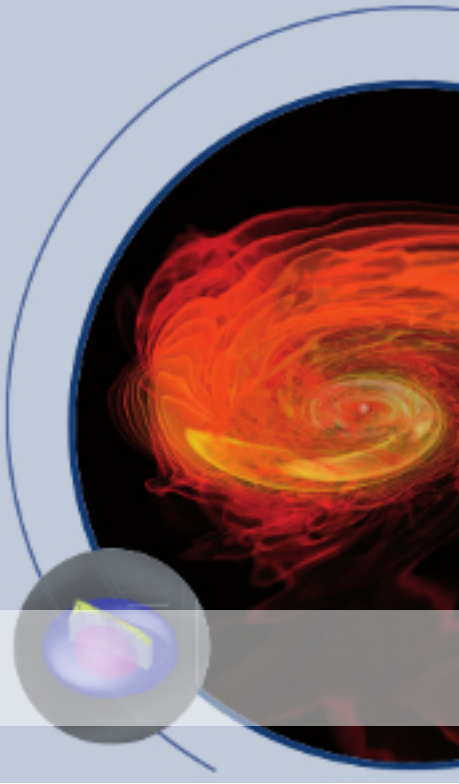
11.1 Real and computational domains

Given ExaHyPE's design decisions, the computer assumes that

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

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.

III Upscaling and tuning ExaHyPE



12. Profiling

While we use ExaHyPE extensively with profiling and analysis tools such as Intel's VTune Amplifier and Likwid, we equip the engine with a couple of profile options and features.

12.1 Build targets

We currently support two build targets for profiling:

- `MODE=Profile` Creates a release executable which is augmented with symbol information and instrumented to output gprof performance data. While the gprof output can be used directly, we also recommend that you use this mode with 3rd party performance analysis tools such as VTune.
- `MODE=PeanoProfile` Creates a release executable which outputs tailored Peano performance data. See section below on further information.

12.2 Plain performance analysis with Peano's scripts

ExaHyPE is based upon Peano and thus offers a few lightweight analysis tools that can be used *without* any particular compiles. Please ensure however that your log filters enable outputs from `peano::performancenalaysis`, i.e. set these entries as whitelist entries. Pipe the executable output into a file and pass it over to the Python scripts `domaindecompositionanalysis.py` which you find in the Peano's directory `peano/performanceanalysis`. Invoking the script without parameters displays a usage message. If the script terminates successfully, you obtain a HTML page which you can study with any browser.

12.3 Advanced Peano performance analysis

Once you translate your code with `MODE=PeanoProfile` and you allow log outputs from `peano::performancenalaysis`, you obtain lots of data that you can postprocess with the Python script `performanceanalysis.py` from the directory `peano/performanceanalysis`.

If you use log files, each MPI rank pipes its performance analysis data into a file of its own. These files have to be merged before you hand them over to the performance analysis. There is a Python script `merge-log-files.py` coming along with Peano that allows you to do this.

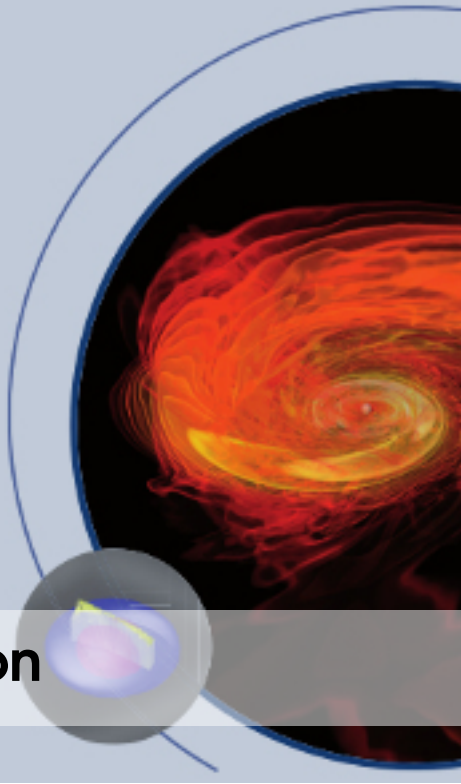
12.4 Scalasca

ExaHyPE supports profiling through Scalasca[1] via a variety of Score-P macro annotations in the Peano kernel and within the ExaHyPE core routines. To translate the code with Scalasca, change the compiler to something alike

```
export EXAHYPE_CC="scorep␣--noonline-access␣--nocompiler␣--mpp=none␣\
␣␣--thread=none␣--user␣icpc␣-DUseScoreP"
export EXAHYPE_CC="scorep␣--noonline-access␣--nocompiler␣--mpp=none␣\
␣␣--thread=none␣--user␣g++␣-DUseScoreP"
export EXAHYPE_CC="scorep␣--noonline-access␣--nocompiler␣--mpp=mpi␣\
␣␣--thread=none␣--user␣mpicxx␣-DUseScoreP"
```

All the arguments instruct Score-P, the instrumenter used by Scalsca. The very last argument is passed through to ExaHyPE/Peano and makes it use the Score-P macros. Please note that this compiler variant works with any build variant, but we strongly encourage users to use solely `MODE=Release` and `MODE=PeanoProfile`.

[1] `http://www.scalasca.org`.



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
  configure = {}
  properties-file = sharedmemory.properties
end shared-memory
```

Rerun the ExaHyPE toolkit afterwards, and recompile your code. Whenever you configure your project with shared memory support, the *default* build variant becomes a shared memory build. Whether it is TBB or OpenMP depends on the environment variable `SHAREDMEM`. Invoke `make` without arguments for details or study the toolkit's output. 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.

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.

TBB

For the compilation, we assume that the environment variables `TBB_INC` and `TBB_SHLIB` are set[1]. The toolkit checks whether the environment variables are properly set and gives advise if this is not the case. ExaHyPE sets an appropriate core number internally reading in the value of `cores`. Consult Table 13.1 for further details.

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

OpenMP

For users with an OpenMP background, please note that we do set the thread count manually within the code. OpenMP environment variables, notably `OMP_NUM_THREADS`, are neglected. Instead, ExaHyPE sets the core count internally reading in the value of `cores`. Consult Table 13.1 for further details.

Using the `identifier` `dummy` as displayed above is a reasonable starting point to assess whether your code is correctly translated and started. The properties file is neglected for the dummy identifier. If you use a more sophisticated shared memory strategy through another identifier, `properties-file` is important. More sophisticated strategies are subject of discussion next. Values within `configure` are specific to the `identifier` selected. The only generic value you can specify is the number of background tasks to be used by ExaHyPE: Our engine deploys tasks to the background from time to time. These are typically low-priority ones. Any idle thread then can automatically process these tasks. The number of the background tasks is set via `background-tasks:x`. Please consult Table 13.3 for further details.

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.

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.

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

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. `dummy` uses some default values that have proven to be reasonably robust and yield acceptable speed. `sampling` tests different choices and plots information on well-suited variants to the terminal. `autotuning` tries to find the best 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. ExaHyPE ignores thread/core counts set externally (via environment variables, e.g.) and tries to book an appropriate number of cores itself. Please consult Table 13.2 if you combine shared memory runs with MPI.
configure		Some specialised shared memory setups require additional configuration data. For plain TBB and OpenMP, an empty clause does the job.
properties-file	filename	The `sampling` and the `autotuning` strategy can write their results into files and reuse these results in follow-up runs. If no (valid) file is provided, they both start from scratch and without any history. If the file name is empty or invalid, no output data is dumped. The entry is ignored if you use the `dummy` strategy.

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.

Table 13.2: Variants of shared memory autotuning.

Options	Description
`autotuning`	The basic variant `autotuning` 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. 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 configuation 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.
`autotuning_without_learning`	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.
`autotuning_without_restarts`	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.

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

- 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

Table 13.3: Supported flags that can be used in the `configure` section.

Options	Description
`background-tasks:x`	ExaHyPE uses background tasks in several places to deploy long-lasting, low priority jobs to free cores. By default, the number of these background tasks is restricted. This avoids that the system is swamped with too many of them and the "main" code starves. You can alter this value. By default it is 1. We notably encourage users to play around with this quantity if they have many cores.
`no-invade` | `invade-between-time-steps` | `invade-throughout-computation` | `invade-at-time-step-startup-plus-throughout-computation`	Can be used if you compile with `TBBInvade` to control the invasion level. Only taken into account if you compile with the invasive TBB extension (cmp. Section 13.3).
`manual-pinning`	Can be used if you compiled with `TBB`. Enforces a strict one to one thread to cpu affinity map. This setting can decrease the applications run-time as cache-efficiency is increased. If the flag is present, ExaHyPE also reports on the exact pinning via terminal outputs. This is useful to validate whether the job scheduler assigns the right cores—even if you rely on the scheduler to take care of all pinning.
`thread-stack-size:X`	Can be used if you compiled with `TBB`. Ensures that each thread at least uses a certain stack size. Consult `https://www.threadingbuildingblocks.org/docs/help/reference/appendices/preview_features/tbb_global_control.html` for details.

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 propery file, i.e., the statistics are incremently 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.

As a rule of thumb, it makes sense first to choose a core count that anticipates the number of MPI ranks you deploy per node. That is, if you have c cores per node and you launch $p < c$ MPI ranks per node, it makes sense to set `cores` to c/p. Please note that some scheduling systems require you to tell the scheduler (Slurm, e.g.) as well if you wanna use more than one thread per MPI rank.

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.

13.3 Invasive TBB

Invasive TBB is a concept to weaken the fixed association of cores to MPI ranks: The ranks with most of the load per node grab more cores than the other ranks. To enable this feature, ExaHyPE requires you to rebuild the code and to re-configure your scheduler scripts. The standard settings of SLURM, e.g., do not allow MPI ranks to shuffle around cores—each rank has a fixed number of cores to administer over the whole runtime, and typically this is the same number of cores per rank.

13.3.1 Use InvasiveTBB version shipped with ExaHyPE

We ship a version of our invasive TBB component with ExaHyPE as the majority of the development on the second generation of the invasive TBB routines has been done within ExaHyPE. To use this extension, first change ExaHyPE's build mode into

```
export MODE=TBBInvade
```

Furthermore, please extend you pathes for the shared memory oracles (`PEANO_TOOLBOX_SHAREDMEMORY_ORACLES_PATH`) to include the `shminvade` directory, too.

```
PEANO_TOOLBOX_SHAREDMEMORY_ORACLES_PATH=mypath/ExaHyPE-Engine/Peano/\
  sharedmemoryoracles:mypath/ExaHyPE-Engine/Peano/shminvade
```

13.3.2 Use SHMInvade 1.0

This section refers to the first version of SHMInvade and thus might not be up-to-date anymore.

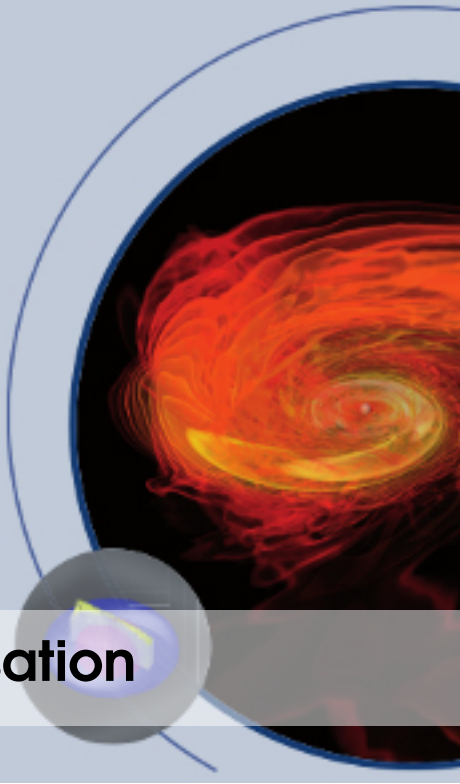
Get the Invasive TBB from `bitbucket.org` through git. It comes along as a header-only distribution. Change ExaHyPE's build mode into

```
export MODE=TBBInvade
```

A build now probably fails due to lacking header search directories. We recommend to extend `TBB_INC` such that they link to the correct invasive libraries, too. Furthermore, we found that most Linux distributions require us to explicitly include `pthread`:

```
export TBB_INC=$TBB_INC"␣-I/directory-holding-shminvade"
export TBB_SHLIB=$TBB_SHLIB"␣-lpthread"
```

13.3.3 Configure SLURM



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, before we detail ExaHyPE's MPI specifica.

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 fragment is minimal and thus well-suited for a first try. Now either rerun the ExaHyPE toolkit and recompile your code, or

```
export DISTRIBUTEDMEM=mpi
```

and recompile. There's always the opportunity to switch MPI on/off through `DISTRIBUTEDMEM`. Depending on whether the `distributed-memory` segment in the specification file is available, MPI on or off is the default. For the compilation, we assume that a proper MPI compiler is available. You can reconfigure it through `EXAHYPE_CC`.

Prepare a benchmark simulation setup

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

1. you switch off any output unless it is the very primitive VTK/VTU output stuff we ship with ExaHyPE. 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. It 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. Follow Chapter 12 and notably Sections 12.2 and 12.3.
4. 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 identifiy deadlocks without you burning too many CPUhs 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. To bundle all messages into one MPI message use a comparably high buffer size (e.g. 16000).

14.3 Configuring the load balancing

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

`identifier`	**Semantics**
`static_load_balancing`	A static load balancing that decomposes ExaHyPE's underlying spacetree at construction time. The grid decomposition is not altered afterwards, even if the grid's structure changes.

Besides the identifier, ExaHyPE needs entries in the `configure` field. A configuration is simply a comma-separated set of identifiers—written down with curly brackets—that control how the chosen load balancing strategy is behaving. There are to be two mandatory entries in the configuration: the load balancing analysis (Table 14.1) and the load assignment strategy (Table 14.2). So you need exactly one entry from each of the two tables. Further optional configuration entries might complete the `configure` set.

[1] If the fair or SFC-based load balancing does not yield proper domain decompositions, please try first to increase the `max-node-pool-answering-time:x` value. Also, please note that all flags are written with a `-` rather than an underscore.

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

Argument	Semantics
`greedy-naive`	Uses a greedy split strategy to decompose the underlying space-tree. 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`	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 however yields better partitions than the greedy partitioning scheme if you use AMR or your rank count is not an exact match to the regular grid (something alike 3^{3kd}, $k \in \mathbb{N}^+$), as it searches for refined mesh regions within the grid and splits those up more aggressively than the remainder. Please note that this strategy works best if combined with `fair` or the SFC-based diffusion plus additional information how the ranks are distributed among the nodes (cmp. Table 14.2).

14.4 Hybrid parallelisation

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

If you use the strategy `sfc-diffusion`, we recommend that you use p primary ranks per node. If a node has c cores, ensure $c \mod p = 0$. For $c = 24$ (and our first KNL tests), we have made good experience with $p = \frac{c}{t}$ and $t = 4$. In this case, we will obtain $p \leq r \leq c$ working MPI ranks r. Some of these will, through the meander pattern of the SFC, become responsible for larger cubic regions of the domain, while others only are responsible for tiny tails of the curve. If thus makes sense to configure the code to use t threads per rank.

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

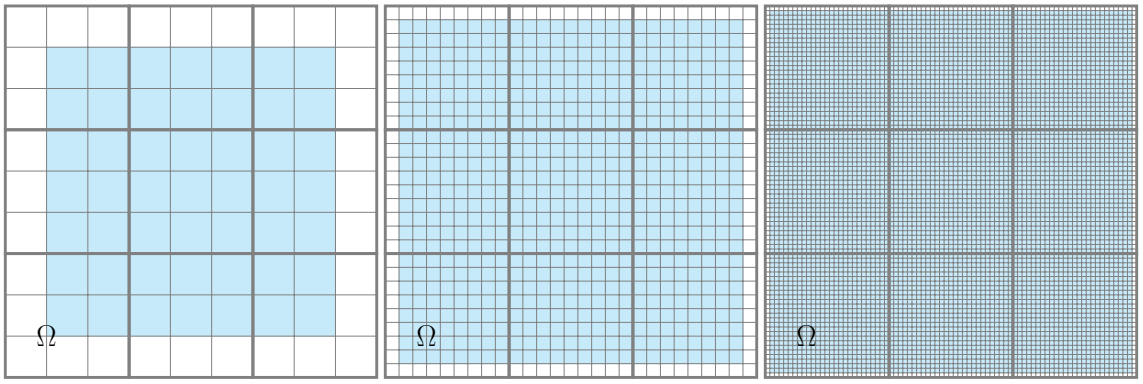


Figure 14.1: The bounding box is scaled such that one cell lies outside of the computational domain Ω. The bounding box is partitioned into 3×3 equal parts. Left to Right: The subdomain sizes become more equal for increasing mesh resolution.

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 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. Furthermore, please ensure that you run the unit tests and the ping pong test before you continue. To run these tests, invoke ExaHyPE without arguments and read how you run the tests instead of the simulation.

I have increased my problem size and my code crashes now Often, scheduling systems return signal 9 but the reason might be different from case to case. Validate that you are not running out of memory! ExaHyPE does not add, for performance reasons, checks 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.

The ping pong test fails This is a known bug with MPI/MPI-2 and ExaHyPE's user-defined data types. Please consult Chapter B.3.

I have increased my problem size and now run into a time out Increase time out values. See Section 14.2. Before you do so, please ensure: If the load balancing seems to be ok, add the load balancing flag `virtually-expand-domain` to your configuration but carefully study the implications (Section 14.5).

I have an (almost) regular grid but the load balancing partitions unfavourable This smell manifests typically in unbalanced logical topology trees if you run Peano's Python scripts. Please consult Section 12.2. Consult the section on load balancing and try different variants. Usually, the `hotspot` strategy from Table 14.1 is our method of choice. If your grid is pretty regular, `greedy-regular` might be the better choice. It is definitely faster throughout the grid construction.

One of my nodes runs out of memory `sfc-diffusion` yields close-to-perfect load distribution among the nodes (not ranks) which might fix your problem. Also ensure that you supplement your load balancing strategy with appropriate information on the number of ranks per node. Please consult the tables in the respective sections.

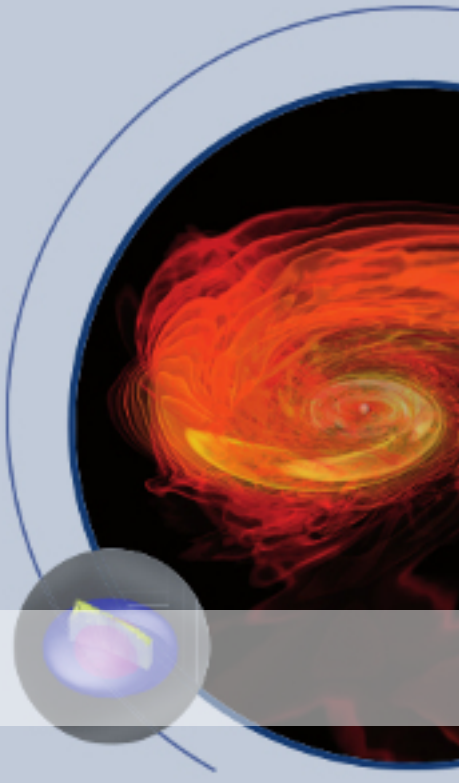
I run out of memory on rank 0 or into a timeout because of rank 0 Add the load balancing flag `virtually-expand-domain` to your configuration but carefully study the implications (Section 14.5).

To be continued ...

For further helpful hints to get MPI running, take a look into the developer FAQ section in appendix B.1, the user FAQ section in appendix B.2 and the known bugs and limitations in appendix B.3. If you try to get MPI working on a well-known supercomputer, probably appendix D can give you an advice.

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

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 `ranks-per-node:X`, 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 `ranks-per-node`th rank, serve this request. Notably, the ranks responsible for the coarsest tree levels are assigned to the ranks `ranks-per-node`, 2· `ranks-per-node`, 3· `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. The fair strategy can be tailored further by a flag `max-node-pool-answering-time:x` which tells the strategy how many seconds to wait until any answer is given, i.e. how long to wait for someone else to raise demands, too. If it is not specified, ExaHyPE falls back to a default value.[1]
`sfc-diffusion`	This is an extension of the `fair` strategy where we try to align all ranks along the Peano space-filling curve. To allow us to do this, we have to know the `ranks-per-node:X` settings. `ranks-per-node` and the total rank count have to match, i.e. the total count modulo ranks per node has to be zero. Further, the strategy also needs `primary-ranks-per-node:Y` with $X \geq Y$. All requests sent to rank 0 are collected for a couple of ms whenever they drop in and then answered such that the load is distributed evenly among the nodes. Hereby, only `primary-ranks-per-node` ranks on each single node are used. Once all primary ranks are distributed, the strategy switches into another state and now uses the remaining ranks so make the SFC be split such that each node receives roughly the same amount of work. The strategy has been developed to be used in combination with shared memory parallelisation and the `hotspot` balancing. The diffusion strategy can be tailored further by a flag `max-node-pool-answering-time:x`, too. See the comments for the `fair` strategy.



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

```
global-optimisation
  ...
end global-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.

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

Restrict adaptivity during time stepping iterations

For many setups, it makes sense to disable the dynamic AMR from time to time, e.g. 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.

Switching dynamic AMR off during time steps removes some additional neighbour communication which might have an impact on the overall performance.

```
global-optimisation
  ...
  disable-vertex-exchange-in-time-steps = on
end global-optimisation
```

ExaHyPE will additionally turn the exchange of metadata completely off in all algorithmic phases where no mesh refinement or limiter status spreading is performed.

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 `time-step-batch-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 `time-step-batch-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.

If you allow the code to run multiple iterations in one batch, ExaHyPE assumes that there is no need to broadcast and restrict any data (such as minimal time steps) during intermediate batch iterations. A broadcast and reduction is still performed in the first and last batch iteration, respectively.

```
global-optimisation
  ...
  time-step-batch-factor = 0.5
end global-optimisation
```

Turning metadata sends off (in intermediate batch iterations)

If you allow the code to run multiple iterations in one batch and you do not employ dynamic limiting, it makes sense to turn off the exchange of ExaHyPE's solver metadata during intermediate batch iterations.

```
global-optimisation
  ...
  disable-metadata-exchange-in-batched-time-steps = 0.5
end global-optimisation
```

Spawn prediction and/or fused time step as background jobs

Certain space-time-predictor kernel computations can be spawned as a background job. In particular, this is the case for cells where we do not need to prolongate boundary fluxes to finer grids or where we do not need to restrict boundary fluxes up to more than one parent cell.

Additionally, we do not spawn a predictor background job if a cell is adjacent to the boundary of an MPI rank's mesh partition as we need to send out boundary fluxes to neighbours directly in the current iteration. This cannot be delayed to the beginning of the next iteration.

This feature can thus establishes an overlap of MPI communication and computation as it prioritises cells at the boundary and postpones work in the interior of an MPI rank's partition.

If we fuse the algorithmic phases of our solver and batching is switched on, we can further spawn a whole fused time step instead of the predictor kernel as a background job during intermediate batch iterations. Of course, the above constraints apply here as well.

To enable the feature, set the flag `spawn-predictor-as-background-thread` to `on`.

```
global-optimisation
  ...
  spawn-predictor-as-background-thread = on
end global-optimisation
```

Spawn AMR operations as background job

Costly AMR operations such as the imposition of initial conditions and evaluation of refinement criteria can also be performed as a background job. This usually increases the concurrency of the mesh refinement iterations however might add a few exta iterations as work is done in the next iteration.

To enable the feature, set the flag `spawn-amr-background-threads` to `on`.

```
global-optimisation
  ...
  spawn-amr-background-threads = on
end global-optimisation
```

Modify storage precision.

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

While the data is compressed, ExaHyPE nevertheless computes with double precision always. Consequently, data has to be converted forth and back: Any compressed data is converted into IEEE precision before any arithmetic operation and later on compressed before it is written

back to the main memory. This process is costly, though we can spawn it into background threads. This is particularly convenient if you have idle cores 'left' and controlled via the flag `spawn-double-compression-as-background-thread`.

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

If you compile with assertions, you can ask ExaHyPE explicitly to validate that all of your compressed data still is close (up to prescribed accuracy) to the actual data. For this, you compile with `-DValidateCompressedVsUncompressedData`.

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:

```
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

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

```
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.

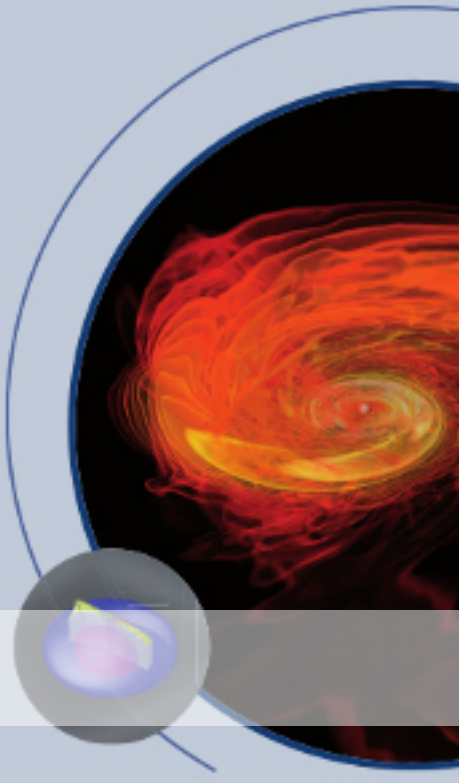
15.3 An Example Optimised Specification File

This section collects together the various optimisations discussed so far into a single production ready specification file.

```
exahype-project Euler

  peano-kernel-path const = ./Peano
  exahype-path const = ./ExaHyPE
  output-directory const = ApplicationExamples/Euler
  // See Optimised Kernels/microarchitecture
  architecture const = knl

  computational-domain
    dimension const = 3
    width = 1.0, 1.0, 1.0
    offset = 0.0, 0.0, 0.0
    end-time = 1.0
  end computational-domain

  shared-memory
    identifier = dummy
    // Set background-tasks and cores to max hardware threads
    configure = {background-tasks:32,manual-pinning}
    cores = 32
    properties-file = sharedmemory.properties
  end shared-memory

  distributed-memory
    identifier = static_load_balancing
```

```
    configure = {hotspot,fair,ranks-per-node:1}
    buffer-size = 64000
    timeout = 600
  end distributed-memory

  global-optimisation
    fuse-algorithmic-steps = on
    fuse-algorithmic-steps-factor = 0.99
    spawn-predictor-as-background-thread = on
    spawn-amr-background-threads = on
    disable-vertex-exchange-in-time-steps = on
    time-step-batch-factor = 1.0
    // Turn on if you do not use dynamic limiting
    disable-metadata-exchange-in-batched-time-steps = on
    double-compression = 0.0
    spawn-double-compression-as-background-thread = on
  end global-optimisation

  solver ADER-DG EulerSolver_ADERDG
    variables const = rho:1,j:3,E:1
    order const = 3
    maximum-mesh-size = 0.05
    time-stepping = globalfixed
    type const = nonlinear
    terms const = flux
    // Use the optimised kernels
    optimisation const = optimised,usestack
    language const = C
    constants = reference:entropywave

  end solver
end exahype-project
```

IV Working with ExaHyPE's output



16. ExaHyPE program output

In this Chapter, 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:

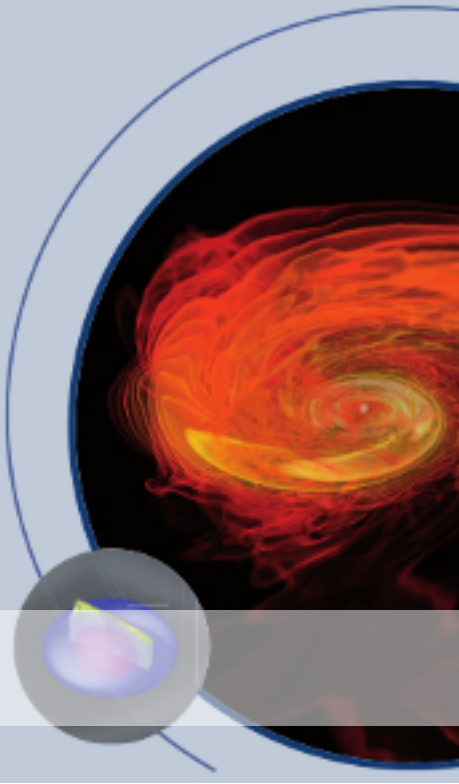
- ExaHyPE makes the compute grid host the solution but often chooses on purpose finer grids for efficiency reasons.
- Most applications can derive information about the actual compute grids used from the output files and a generic plot thus would be redundant.
- In MPI, ExaHyPE tries to avoid global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

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

```
PROJECT_CFLAGS+=-DTrackGridStatistics
```

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

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



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 of the elimination of helper variables from the output.

> **Design philosophy 17.1** ExaHyPE's IO is per se very simplistic as we do ackwnowledge that most applications have very application-specific plotting demands. We thus decided on purpose that we support only very few output formats, do not support sophisticated plotting (such as symbolic names onto output fields) or complex filtering. However, we do offer the infrastructure to realise sophisticated plots on the application side that do scale up.

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

17.1 A short introduction of plotting paradigms

Due to the complex subcell structure in ExaHyPE (for details, see chapter 21), there are many design decisions when it comes to plotting the degrees of freedom of a simulation. Some of these decisions are classical for any fluid dynamics code and shall reviewed here briefly.

17.1.1 Exact representations of degrees of freedom

ADERDGSolver The subgrid are the polynomial supporting points living on the Legrende basis points. In general, an p order ADERDG method has $p+1$ supporting points (vertices) on one axis. For instance, for a second order DG approximation in 1D, an exact output (in

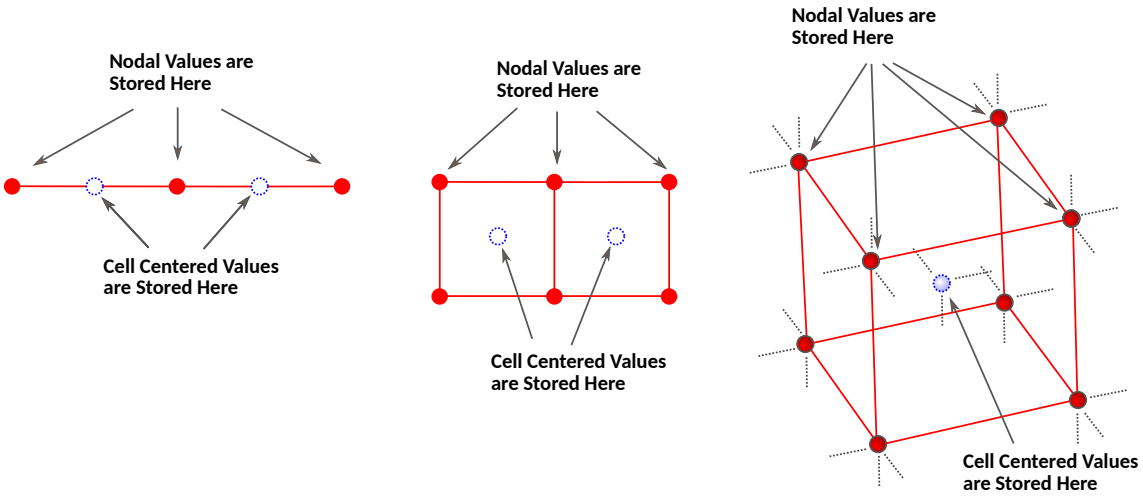


Figure 17.1: Storage location of data in node- vs cell-centered formats in 1D, 2D and 3D. The cell-centered location is similarly determined for non-cubic cells. Adopted from [3].

terms of a dump of the simulation) contains three data points per cell and thus holds three times more data than with a cell-averaging output method. As long as these $p+1$ points are represented in an output file format, one is able to exactly reconstruct the polynomials in a postprocessing step.

FVSolver In case of the Finite Volume kernels, in ExaHyPE they are always realized with regular block based grids embedded into cells/patches. The size of these blocks can be controlled via the specification file (parameter `basis-size`). For a basis size N, there are N (cell centered) points on one axis. As a matter of principle, the number of points is independent from the (convergence/computational) order of the finite volume scheme working on this grid. Anyway the basis size is one of the primarily visible things in any ExaHyPE output format. Note that in the Finite Volume schemes the plotters we provide in ExaHyPE *never* print ghost zones.

LimitingADERDGSolver The same as ADERDGSolver.

17.1.2 Cell-centered and nodal values: Native solver representation

Many file formats support the representation of physical fields both cell-centered and vertex-centered (or nodal). The concept is briefly sketched in figure 17.1. The following overview lists the way how this concept goes along with specific solvers:

ADERDGSolver This is the most versatile solver in terms of plotting as it allows an easy evaluation (sampling) of polynomials on any basis. The internal representation ("where the data live") is the *nodal Gauss-Legendre* basis. Thus, the `Legendre::vertices` plotters display the degrees of freedom as they are used in the simulation without any change. In any other representation, there is at least an intrinsic error of the evaluation (~ polynomial order). When plotting DG solutions cell-centered, it is only a matter of a small position shift but potentially a different meaning for the output data.

FVSolver The internal representation of the Finite Volume solver data is the regular *cell-centered Cartesian* basis. Thus, you only obtain the real unchanged data when using a `Cartesian::cells` combination. Indeed, for the VTK plotters, we (currently) don't even implement a different combination, ie. we don't do interpolation[1].

[1] Note 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

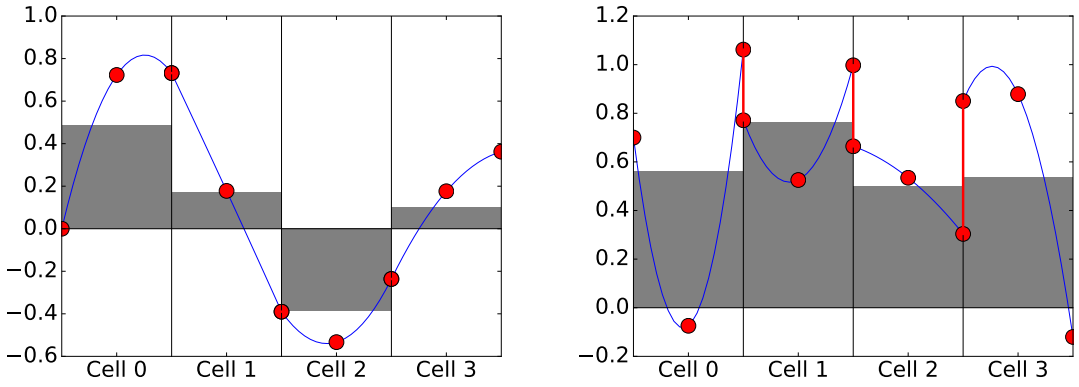


Figure 17.2: A sketch of four 1D cells holding a polynomial of degree $p = 2$, supported by $N = 3$ nodal points in an equidistand basis (which could be a Gauss-Lobotto basis). Since the nodal points are on the cell boundary, there are data doublings (multiplicities) which only stand out visually when there are discontinuities (right panel). The output is a double-valued function. The gray bars indicate the cell average.

LimitingADERDGSolver The limiting ADER-DG scheme (as described in chapter 7) always holds the DG degrees of freedom when it comes to plotting. Therefore any statement about the ADERDGSolver also applies for the LimitingADERDGSolver.
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 neccessarily 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.

17.1.3 Subcell structure: Discontinuities and ghost cells

> **Design philosophy 17.2** Plotting shall represent the degrees of freedom in a simulation. In this spirit, we *never* plot ghost zones in a FV scheme. However, we *always* plot double values in an DG scheme.

All kernels in ExaHyPE have their own cell description where there is a certain overlap of information with neighbouring cells. When it comes to plotting, one has to choose how redundancy is handled. Principle 17.2 applies. In detail, the following rules apply:

ADERDGSolver In a nodal representation, in particular bases nodes lay on the cell boundary and thus the discontinous nature of the DG scheme is exposed by the plotter. In continous ("shock-free") fields, this introduces redundancy, i.e. the same values $\Phi(\vec{x})$ seem to occur multiple times in the plot. However, as soon as the field values are discontinous along multiple cells, these points provide additional information, cf. figure 17.2. In any case, in a recovarable plot, the discontinuity can be recovered by computing the polynomial from the plot data.

FVSolver Finite volume schemes always have *ghost layers* which are however usually never plotted in ExaHyPE. If the DOF are plotted cell-centered, they never touch the cell boundary. Note that any nodal presentation either offsets the data (staggering), introduces data doubling or interpolates into the ghost layer. In such a nodal plot, figure 17.2 applies

again.

LimitingADERDGSolver The same comments as for the ADERDGSolver apply.

While a visualization software might use available ADER-DG data (in combination with knowledge of the polynomial order) in order to reconstruct the polynomial we don't expect this to happen in every-day visualisations. Common visualization software (such as ParaView, Visit, Amira) only does linear interpolation between data points at all and users prefer even to turn off this in favour of an neirest-neighbour visualization. Depending on the file formats, it may be also difficult to extract the neccessary information for high order visualization, for instance in the VTK file formats it had to be reconstructed.

17.2 Overview of supported data output formats

This section 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 your compiled application with the argument `--version` and you should obtain an output similar to

```
> ./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 plotter, you have to rerun the ExaHyPE toolkit and rebuild your code—all other parameters are read at runtime. A plotter always has to specify how many variables are written through the `unknowns` statement. By default, these are the first `unknowns` unknowns from your solver. If `unknowns` is greater than the unknowns of the solver, you also obtain the material parameters if there are material parameters. Otherwise you do not obtain useful data but could map your unknowns to locally derived quantities.

Note 17.1 Plotters in ExaHyPE have an identifier which reads like `featureA::featureB::featureC::...`. The identifier is not case sensitive.

17.2.1 Volumetric plotters with generic file formats (VTK)

All volumetric plotters are by construction able to plot the full simulation domain. The VTK file format is adopted as one of the most widespread file formats. For details about the file format, we refer to appendix C.1.

As a first rule of thumb, ExaHyPE currently supports the following combinations of plotter parameters:

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

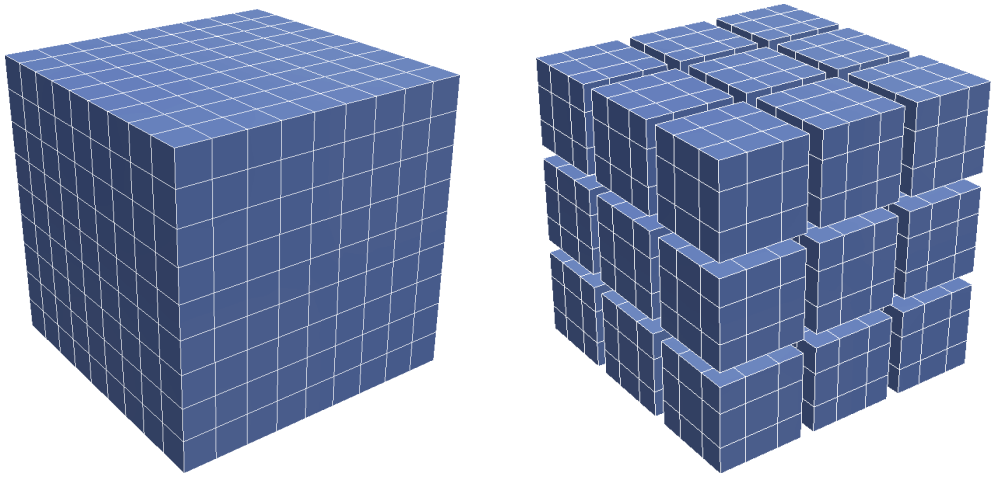


Figure 17.3: Cartesian vs. Legendre volumetric plotters: Both cubes show a one-time refined cube with p =3 ADERDG subgrid, left equally spaced interpolated, right on the Legendre points.

Figures 17.3 and 17.4 visualize the difference between Cartesian and Legendre plotters. Figure 17.5 visualizes the difference between Vertices and Cells sampling. The variant `vtk` is a legacy format supported by most postprocessing tools. `vtu` is a newer format wich can represent the same data, but encoded as XML and with substantially improved handling in case MPI is used. 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`. As a restriction, in the moment we only support `vtu` in combination with `ascii`, i.e. binary vtu is not (yet) implemented. Note that the `vtu` plotters create substantially more files. However, as in general with the VTK family, these files are well compressible (for instance with *zip*).

Design philosophy 17.3 ExaHyPE uses Peano routines for block structured output. Peano draws a line between the semantic file format (*what* is represented in a file) and the file format syntax (*how* information is encoded in a file). This seperation reflects in the possibility to dump either human-readable text files (ASCII) or special binary files (for instance VTK's binary representation or the HDF5 binary file format).

Basic VTK plotter parameter combinations

The basic VTK plotters are given by the combination

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

Note that you cannot choose `Legendre` plotters on FV solvers, while you can choose `Cartesian` plotters on DG solvers.

`vtk::Cartesian::vertices::ascii` Dumps the whole simulation domain into one VTK file. If you use multiple MPI ranks, each ranks writes into a file of its own. The plotter uses ASCII, i.e. text format, so these files can be large. `output` has to be a valid file name where a rank and time step identifier can be appended. The plotter always adds a `.vtk` postfix. Through the `select` statement, you can use spatial filters. if you add values such as `select = left:0.4,right:0.6,bottom:0.2, top:0.8,front:0.2,back:0.5`, you get only data overlapping into this box. As VTK does not natively support higher order polynomials, the solution is projected onto a grid, and the solution values are sampled

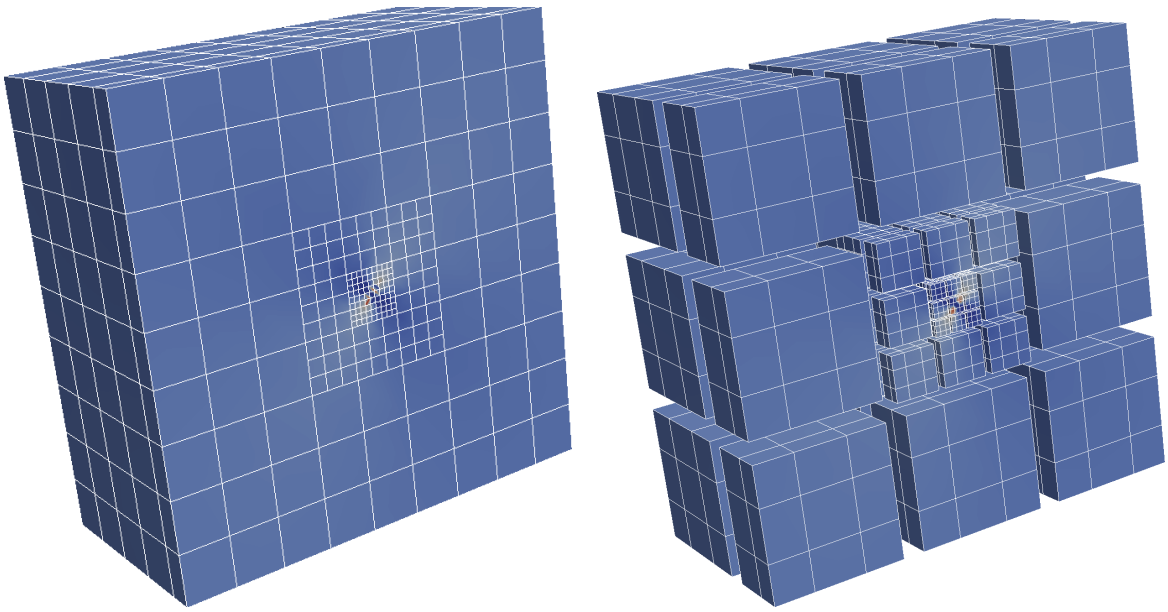


Figure 17.4: Cartesian vs. Legendre volumetric plotters, with focus on mesh refinement, in this example around the center. The plot shows a color encoded scalar field.

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.

Plotters for limited solvers

For the limited solvers, additionally the following family of VTK-based plotters may be used:

$$\begin{Bmatrix}\texttt{vtk}\\ \texttt{vtu}\end{Bmatrix} \times \begin{Bmatrix}\texttt{Legendre}\\ \texttt{Cartesian}\end{Bmatrix} \times \begin{Bmatrix}\texttt{Vertices}\\ \texttt{subcells}\end{Bmatrix} \times \texttt{limited} \times \begin{Bmatrix}\texttt{Ascii}\\ \texttt{Binary}\end{Bmatrix}$$

`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

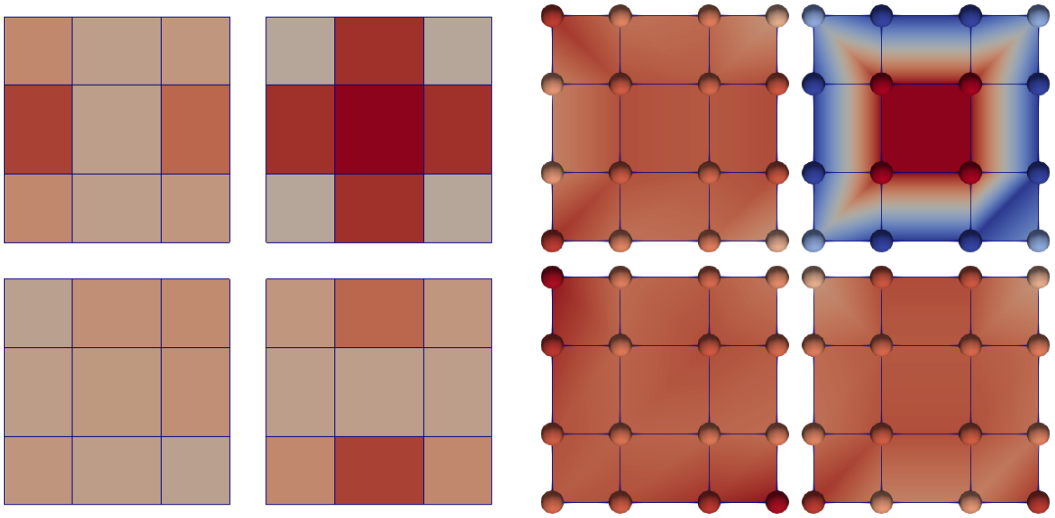


Figure 17.5: Cell-centered plotting (left) vs. plotting at vertices (right) at the example of the Legendre plotter. For the cell centered plotting, values are sampled in the centers of each plotting cell, in the case of the Legendre plotter this is in each subcell center. In contrast, plotting on the 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.

cell. The limiting status of all points inside a cell/patch is similiar, 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` This plotter generates a VTK unstructured mesh which shows the real position of the DOF (degrees of freedom) for a limited ADERDG solver. That is, for any unlimited cell it is equal to the `vtk::Legendre::cells` plotter while for any limited cell it is basically equal to a `vtk::Cartesian::cells` plotter which shows the real $2N+1$ subcells.

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

Plotters for cell averages

In addition, there is a family for plotting only *one* point per cell (or patch), i.e. disregarding the complete subcell structure. It follows the naming

$$\left\{\begin{matrix}\texttt{vtk}\\\texttt{vtu}\end{matrix}\right\} \times \texttt{patches} \times \left\{\begin{matrix}\texttt{gaps}\\\texttt{boxes}\end{matrix}\right\} \times \left\{\begin{matrix}\texttt{Ascii}\\\texttt{Binary}\end{matrix}\right\}$$

where data are always stored cell-centered in the patches. Users can map values of a complete patch (for instance cell averages or some flags) to a single number. Data is enriched in any case with ExaHyPE-internal information. For instance, in the vicinity of a limiting solver, limiting information is written out. In the vicinity of MPI, rank information is written out.

`vtk::patches::boxes::ascii` Displays the cells as touching boxes (resulting in a regular Cartesian grid).

`vtk::patches::gaps::ascii` Mimics the look of the `Legendre` plotter, i.e. an artificial gap between each cell is inserted.

17.2.2 ASCII CSV human-readable file plotters

Certainly even more straightforward then VTK is the usage of CSV plotters. CSV stands for *Comma Seperated Values* and is an expensive but widespread and accepted file format for tables of numbers. Virtually any postprocessing tool will be able to read in such ASCII tables (ASCII is used to indicate a *plain text* file), such as *gnuplot*, *numpy* or *Microsoft Excel*.

ExaHyPE already ships a basic CSV writer which is useful in practice when combined with lower-dimensional slicing (cf. section 17.3), for instance to quickly write out a 1D axis of data in a commonly understood fileformat.

In ExaHyPE, we currently have these CSV file formats:

`csv::Legendre::vertices::ascii` Writes out the original Legendre nodes (vertices) in ASCII. Suitable for the Legendre basis.

17.2.3 Tailored volumetric output formats: Peano and Carpet

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 unstructed mesh.

Peano block regular file format

The following combination of the peano block regular files is supported:

$$\texttt{Peano} \times \left\{\begin{matrix}\texttt{Legendre}\\ \texttt{Cartesian}\end{matrix}\right\} \times \left\{\begin{matrix}\texttt{vertices}\\ \texttt{cells}\end{matrix}\right\} \times \left\{\begin{matrix}\texttt{ascii}\\ \texttt{hdf5}\end{matrix}\right\}$$

`Peano::Legendre::cells::hdf5` Dumps data in the Peano block-regular file format, as proposed in section C.3. It resembles closely the array-of-structure in-memory storage layout of ExaHyPE and saves the majority of the metadata overhead by essentially storing $\vec{x}, \mathrm{d}\vec{x}$ of each cell instead the positions of each data point in the subcell structure. This plotter is not supported for Finite Volume solvers.

`Peano::Legendre::vertices::hdf5` Vertex-based variant of the above plotter. This plotter is not supported for Finite Volume solvers.

`Peano::Legendre::cells::ascii` ASCII, i.e. text, output of Peano's block-regular file format. This plotter is not supported for Finite Volume solvers.

`Peano::Legendre::vertices::ascii` Vertex-based variant of the above plotter. This plotter is not supported for Finite Volume solvers.

`Peano::Cartesian::vertices::ascii` Similar to Legendre variant. The ADER-DG polynomials are mapped onto Cartesian coordinates, i.e. onto a block-regular Cartesian mesh per cell. If you use pth order polynomials, the plotter dumps a $p \times p \times p$ grid. The plotter projects the unknowns onto the vertices of this image grid. This plotter is not supported for Finite Volume solvers.

`Peano::Cartesian::vertices::hdf5` Similar to variant above with HDF5 output file.

`Peano::Cartesian::cells::ascii` If this plotter is used with an ADER-DG solver, the integration points are mapped onto Cartesian coordinates, i.e. onto a block-regular Cartesian mesh per cell. See remarks above. The unknowns are mapped onto the cell centres. If the plotter is used with Finite Volumes, the Finite Volume submesh per cell is dumped.

`Peano::Cartesian::cells::hdf5` HDF5 variant of the plotter as described above.

The Carpet output file format

The following combinations for the Carpet output file format are supported:

$$\texttt{Carpet} \times \texttt{Cartesian} \times \texttt{vertices} \times \left\{\begin{matrix}\texttt{ascii}\\ \texttt{hdf5}\end{matrix}\right\}$$

That is, by design, the Carpet file format always stores vertex data, sampled on a Cartesian grid. The Carpet plotter can be used in ASCII mode (creating CSV tables) and in HDF5 mode (creating compact H5 files). Furthermore, the plotter can be used in a Finite Volume, Discontinous Galerkin or Hybrid (Limiting) solver application. The two options in detail are:

`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 $\mathrm{d}\vec{x}$. 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 quality loss in the outputted files as the interpolation from cell to vertex data is not ideal, also given the limited amount of ghost cell information available. 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 C.4 in the appendix. In order to use the CarpetHDF5 file format, you have to build ExaHyPE with HDF5 support.

`Carpet::Cartesian::Vertices::ASCII` This flavour of the Carpet file format does *not* represent exactly the same data structure just in another container (as with the Peano format) but instead really writes very lengthy CSV tables. They can be directly read in by Gnuplot or similiar tools. However, in practice they are only useful for lower dimensional data because they quickly grow big.

The Carpet writer is especially known for some distinct features:

- Actual dimension reducing slicing of the data (i.e. writing a 1D cut, or a 2D plane from 3D data). As the file format is so far in use only in highly symmetric astrophysical simulations, we concentrated on easy Cartesian slicing only (i.e. planes and lines aligned with the Cartesian coordinate system).
- The option to write all the written output vector to one file or to multiple files, as well as to open one file per timestep or to write several timesteps in a single file.
- So far, every MPI rank writes their own file. This is similar to all other ExaHyPE plotters.

Add some scree of how Carpet f look like

The Flash output file format

`Flash::hdf5` 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 C.5.

Design philosophy 17.4 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.

Note 17.2 Please refer to section C.2.1 to learn how to enable HDF5 in ExaHyPE's build system.

17.2.4 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 possiblity 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 and Slicing

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[2]. 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

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

if k unknows 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 conversations such as a conversion of conservative to primitive variables. Or you might want to select the few variables from the PDE solver that you are actually interested in.

Below is an example:

```
solver ADER-DG MyEulerSolver
  variables const = 5
  order const = 3
  maximum-mesh-size = 0.1
  time-stepping = global
  type const = nonlinear
  terms const = flux,ncp
  optimisation const = generic
  language const = C

  plot vtk::Cartesian::ascii MySolutionPlotter
    variables const = 2
    time = 0.0
    repeat = 0.05
    output = ./solution
  end plot

...
```

Originally, we did plot all five variales. Lets assume we are only interested in 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];
}
```

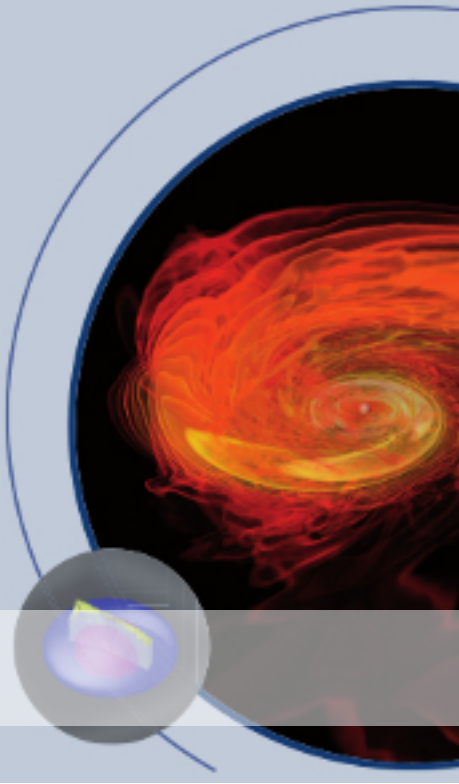
Alternatively, with all the information like the space x and time t at hand, you can easily inject other quantities into `outputQuantities`, for instance exact initial conditions. You can also use the information to compute *locally* a derived quantity from Q.

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.

Describe better the plotting classes and where to start to implement an own plotter.



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 L2 integral in m, while you use `finishPlotting` write the result to the terminal or into a file of your choice, e.g. If you alter the start and finish routine, please continue to call the parent operation, too.

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

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

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

```
void MyEulerSolver_Plotter0::startPlotting(double time) {
  // Reset global measurements
```

```
  _globalL2Accumulated = 0.0;
}


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


void MyEulerSolver_Plotter0::mapQuantities(
    const tarch::la::Vector<DIMENSIONS, double>& offsetOfPatch,
    const tarch::la::Vector<DIMENSIONS, double>& sizeOfPatch,
    const tarch::la::Vector<DIMENSIONS, double>& x,
    double* Q,
    double* outputQuantities,
    double timeStamp
) {
  // There are no output quantities
  assertion( outputQuantities==nullptr );
  // Now we do the global computation on Q but we do not write anything
  // into outputQuantities
  const NumberOfLagrangePointsPerAxis = 4; // Please adopt w.r.t. order
  const NumberOfUnknownsPerGridPoint = 5; // Please adopt

  // 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 interfer with any ExaHyPE core messages if we postprocess data. To ensure this, we propose to *tag* messages, i.e. to give them a label to say "these are for my postprocessing" only. Peano offers a tagging mechanism that we use here.

Our proposed reduction now realises the following three ideas:

1. We do all the reduction at the end of `finishPlotting` where we can assume that the reduced data per rank is available.
2. If we run on any rank that is not the global master, `finishPlotting` sends its data to the global master.

3. The global master's `finishPlotting` runs over all ranks that are not idling around and collects the data from them.

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

...

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

 double myValue = ...;

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

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

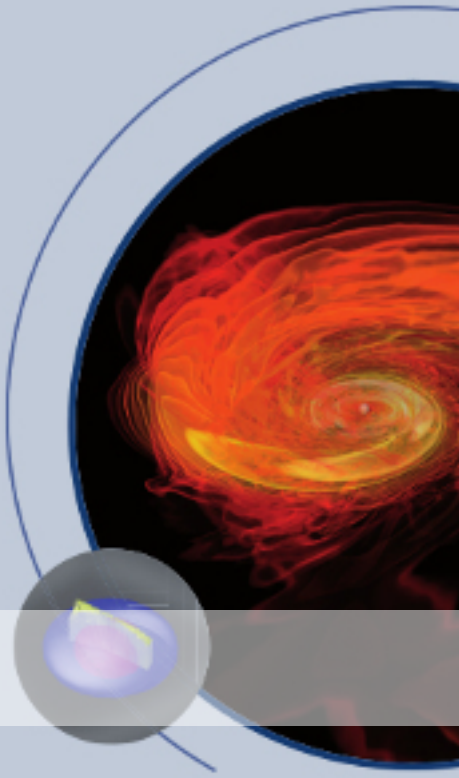
The `reserveFreeTag` operation stems from Peano. We are supposed to tell the function which operation has invoked it, as this is a quite useful piece of information for MPI debugging. The last line of the code snippet returns this tag again, so it might be reused later on. Our snippet demonstrates the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

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

Our code snippet sends the value of `myValue` to the global master if we run into the routine

on a rank which is not the global master itself. If the plotter is invoked on the global master, we run over all the ranks that do exist. Per rank, we ask the node pool whether the rank is idling. If it is not, we know that it will send data to the global master and we receive these data.

V ExaHyPE design & rationale



19. ExaHyPE architecture

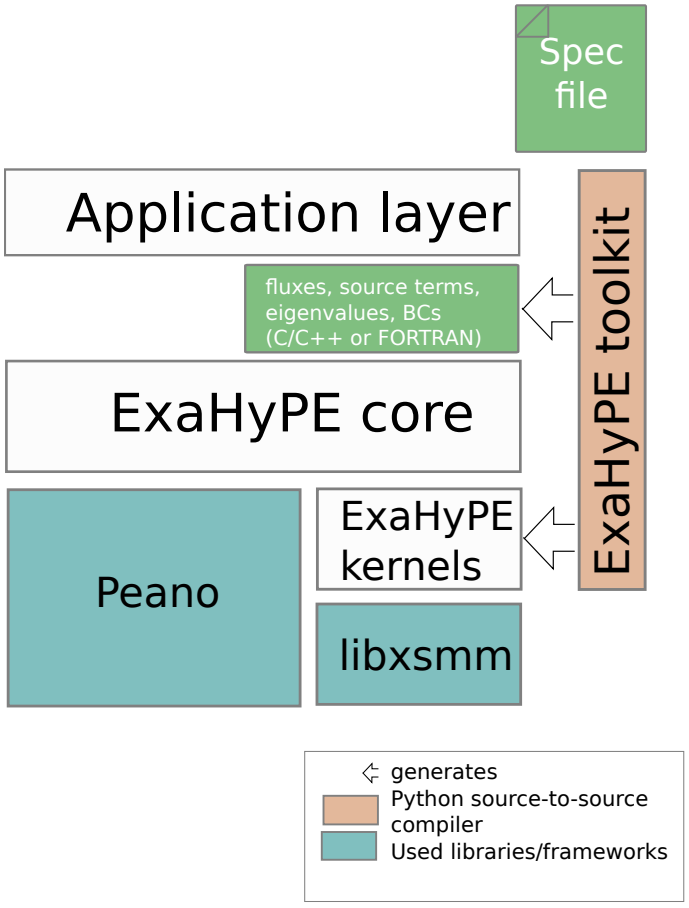


Figure 19.1: The generic ExaHyPE architecture at a glance

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 emtpy templates that are to be befilled. Instead, it creates domain- and architecture-specific code fragments (*kernels*) that are now invoked whenever we run into computationally demanding program phases.

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

Obviously, our architecture as described so far still is simplistic and rudimentary; too rudimentary for most real-world applications. We hence expect most applications to extend and connect it to futher software fragments. An example is given in Figure 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.

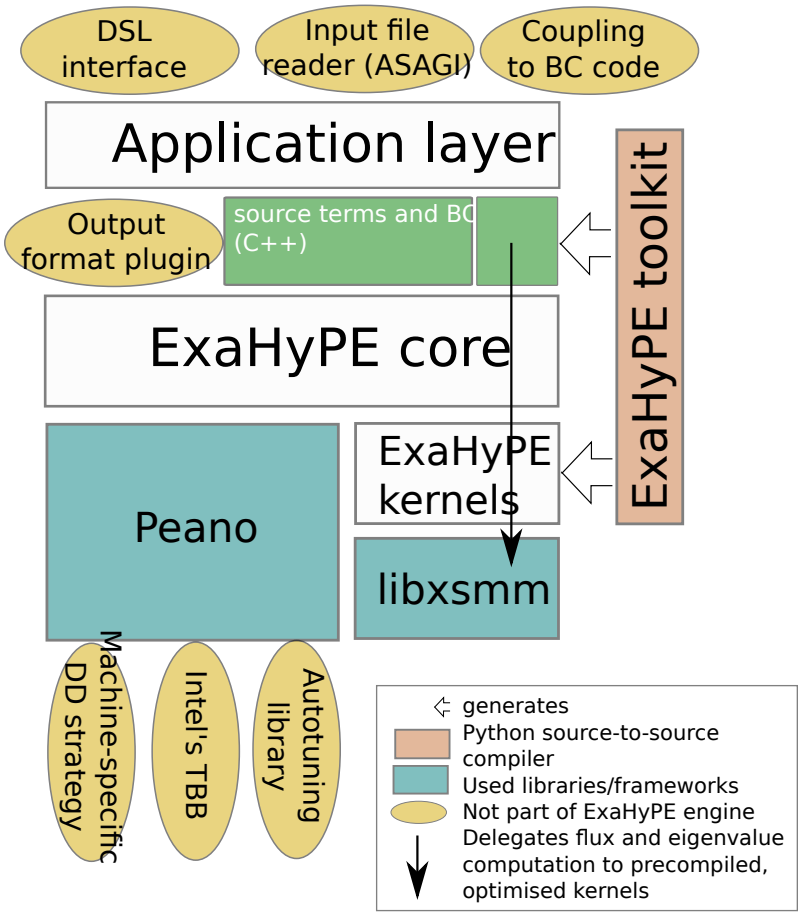


Figure 19.2: Any ExaHyPE application tailors/uses/adopts the generic architecture to its needs. The present snapshot shows a sophisticated seismology ExaHyPE code that couples the simulation engine to 3rd party software and provides a domain specific language (DSL) in the top layer, delegates all flux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

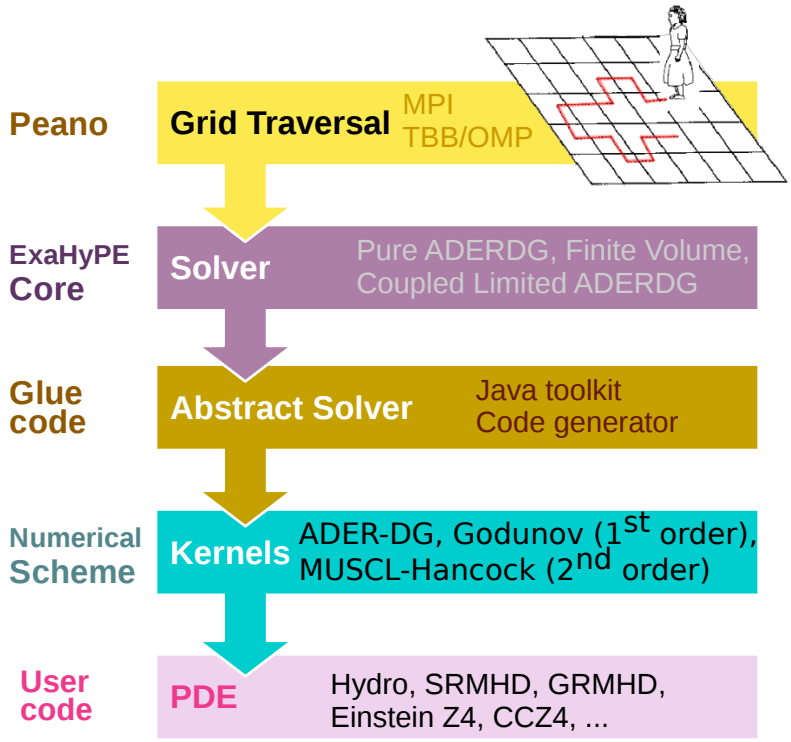
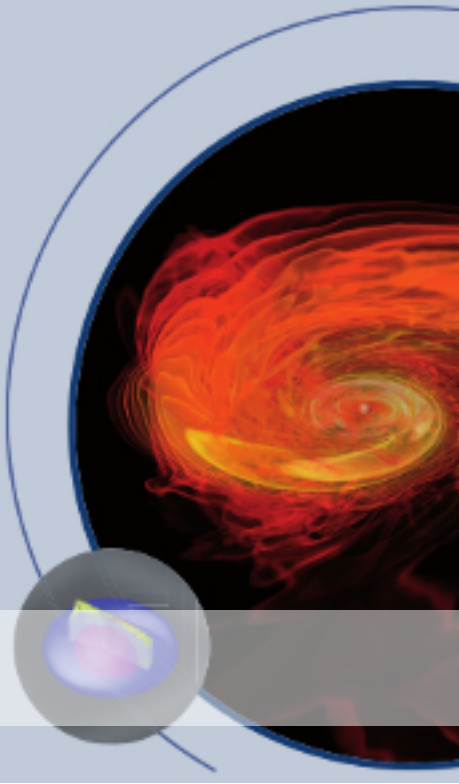


Figure 19.3: The different layers of ExaHyPE, a C++-centric way of understanding the architecture of ExaHyPE (compare with figure 19.2.)

It turns out that every part of figure 19.3 performs an individual, mostly modular and seperable 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 Python 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.

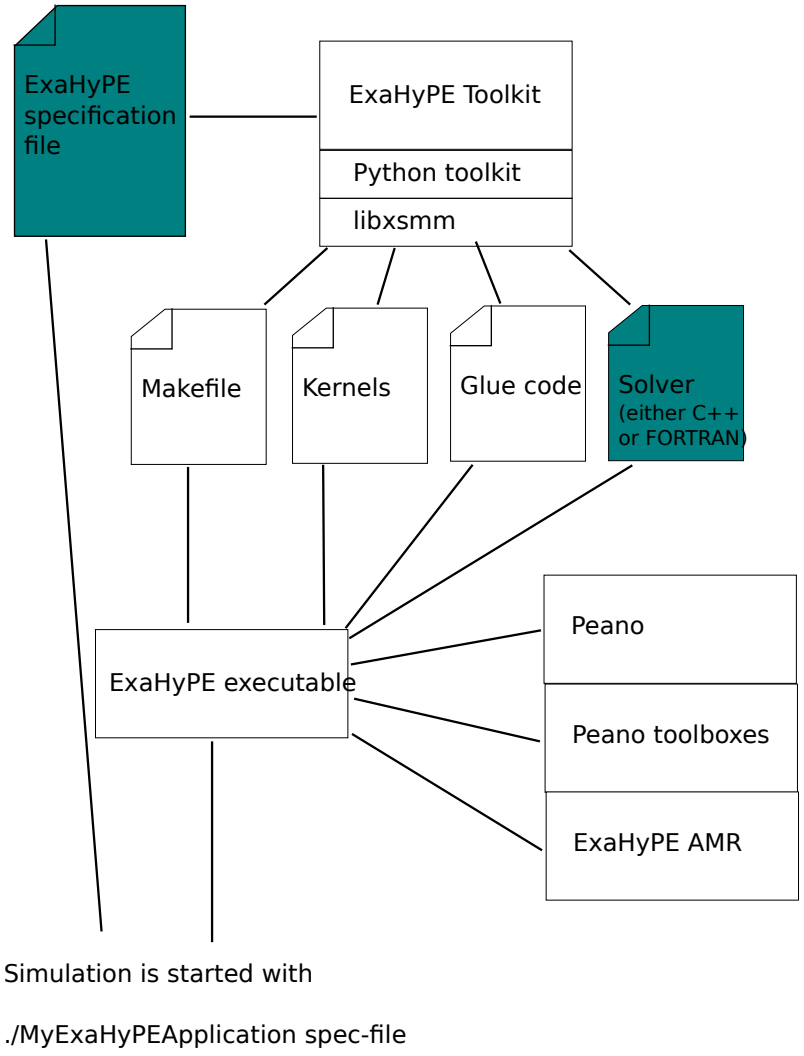


Figure 20.1: A typical ExaHyPE workflow.

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                   Select GNU compiler
  export COMPILER=Intel                 Select Intel compiler (default)

  export MODE=Debug                     Build debug version of code
 export MODE=Asserts             Build release version of code that is augmented with assertions
 export MODE=Profile             Build release version of code that produces profiling information
  export MODE=Release                   Build release version of code (default)

  export SHAREDMEM=TBB                 Use Intels Threading Building Blocks (TBB) for
                                        shared memory parallelisation
  export SHAREDMEM=OMP                  Use OpenMP for shared memory parallelisation
 export SHAREDMEM=None               Do not use shared memory (default if not indicated
```

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

export DISTRIBUTEDMEM=MPI               Use MPI
export DISTRIBUTEDMEM=None              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 managament

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 seperate directory trees.

In ExaHyPE, there is a lot of contributed code which results from endavours to master the ExaHyPE production workflow in a quick and cross plattform 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, compiliation, execution)
- Checking and looking up of build-specific environment variables
- Simulation managament
- 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.

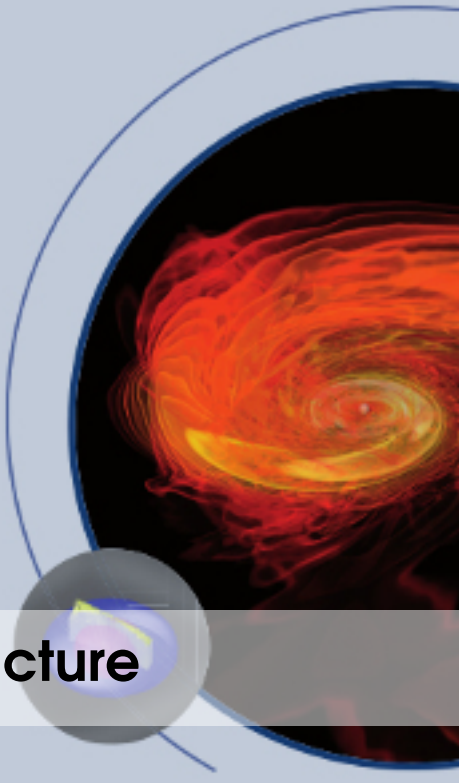
20.4 Simulation managament

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 task[1]. Thus there is an urgent need for tools managing the directory layout and queue managament. 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 managament 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 Python tookit, it is useful to understand specification files as common text files when it comes to batch processing. Typical attemps 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.

[1] as can be read at the website of `http://simfactory.org/`.



21. ExaHyPE grid and subcell structure

The computational grid in ExaHyPE is not uniform. This chapter is dedicated to explain the computational domain.

In ExaHyPE, the computational domain is covered by *cells* which are managed by the Peano software. These cells are always cubic and of different size, according to the AMR level where they live on.

The actual schemes in ExaHyPE work on an *subgrid* which is spawned inside each cell. The grid used is dependent on the scheme employed. Note that the computational grid does not have to match the grid which is used for data writeout (plotting). See section 17.1 for details about how data are written out.

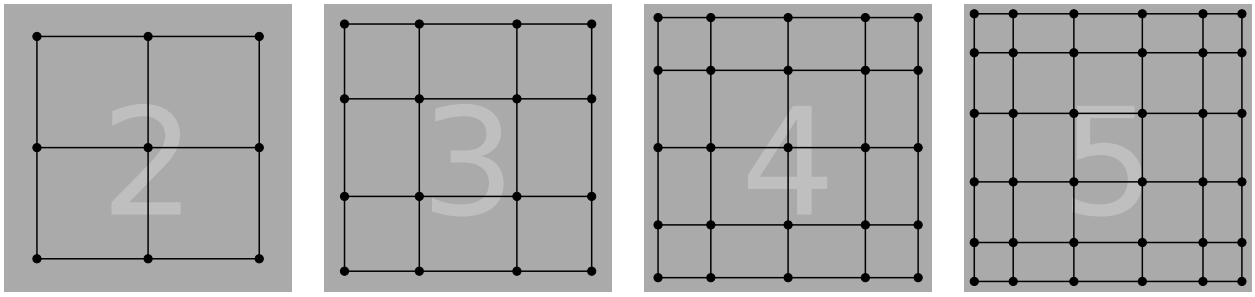
21.1 ADER-DG scheme

In ExaHyPE, the subgrid in the ADER-DG scheme are either the *Gauss-Legendre* or *Gauss-Lobotto* collocation points. Both basis yield a diagonal mass matrix.

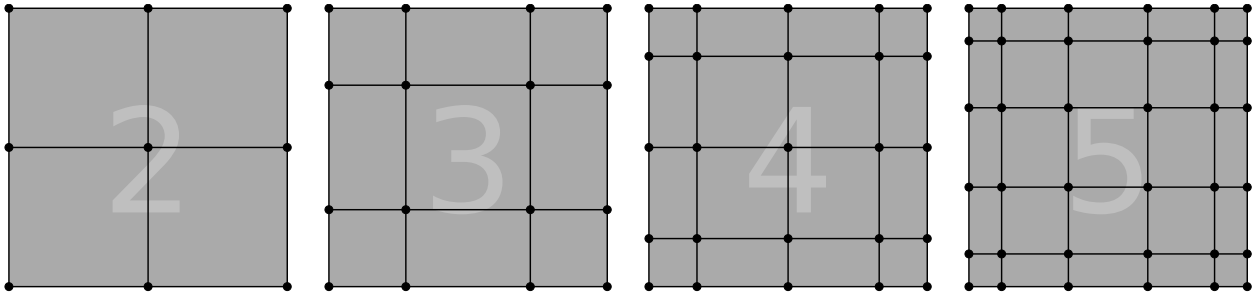
The Gauss-Legendre basis points are *non-uniformly* distributed and do *not* touch the cell boundary. In contrast, the Gauss-Lobotto basis points are *non-uniformly* distributed and *do* touch the cell boundary. Note that both use non-uniform distribution of collocation points but with different distributions. For the Gauss-Lobotto case, this touching the boundary means that that neighbouring cells may store (potentially different) field values at the same spatial positions. For a visualization of the basis node in 2D, see figure 21.1.

21.2 Finite Volume scheme

The Finite volume scheme in ExaHyPe always works on *uniform* grids of a specific size. This is sketched in figure 21.2. As introduced in section 5.7, in ExaHyPE, finite volume cells are always embedded as regular grids in bigger patches. Note that the figure does not take *ghost cells* into account. For instance, you could think of the $N = 4$ case in figure 21.2 as an $N = 2$ patch with two ghost layers, sufficient to be used for a second-order FV scheme.

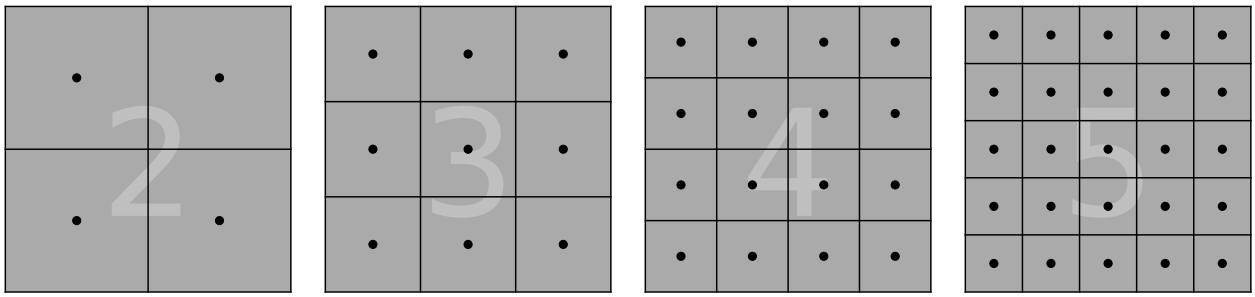


(a) Legendre, $p = 2$ (b) Legendre, $p = 3$ (c) Legendre, $p = 4$ (d) Legendre, $p = 5$



(e) Lobotto, $p = 2$ (f) Lobotto, $p = 3$ (g) Lobotto, $p = 4$ (h) Lobotto, $p = 5$

Figure 21.1: Subcell grid in the ADERDG scheme as implemented in ExaHyPE. The figures are for a 2D simulation, the extension to 3D is straightforward. Shown are the Gauss-Legendre basis nodes in comparison to the Gauss-Lobotto basis nodes for different polynomial orders p. The grey box indicates the cell.

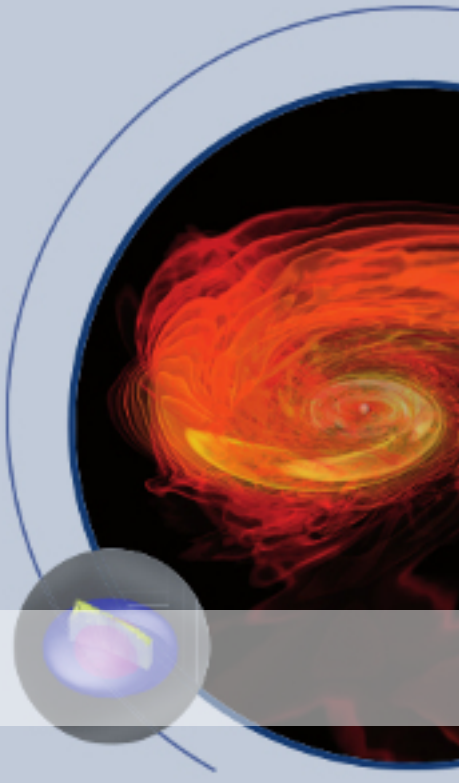


(a) Finite-Volume, $N = 2$ (b) Finite-Volume, $N = 3$ (c) Finite-Volume, $N = 4$ (d) Finite-Volume, $N = 5$

Figure 21.2: Subcell grid in the FV scheme as implemented in ExaHyPE. The figures are for a 2D simulation, the extension to 3D is straightforward. Shown are the uniform and equidistant cells embedded in a patch in ExaHyPE for different patch sizes N. For a direct comparison to figure 21.1, we choose similar numbers. Here, the cell-centers are marked with a dot. Read the main text for further comments.

21.3 Limiting ADERDG scheme

Figure 7.1 in chapter 7 explains how the FV and DG cells are mapped onto each other in 1D. This is not covered in this chapter.



22. Upscaling rationale

ExaHyPE's and Peano's domain decomposition is sometimes somehow non-orthodox. People tend to classify it as space-filling curve-based (SFC) parallelism. Yet, that falls short. In this chapter, we quickly sketch how it works. The sketches themselves are done in 1d to keep the explanations simple. All the real-world data presented stem from an Euler flow run as described before in this document. The output is analysed through Peano's Python script `domaindecomposition.py` throughout the first part of the section. The later text then recompiles ExaHyPE with `MODE=PeanoProfile` and uses the script `performanceanalysis.py` to squeeze out more data from the runs.

22.1 Greedy spacetree decomposition

We start off with thee greedy decomposition scheme, i.e. we use the flags

```
configure = {greedy-naive,FCFS,virtually-expand-domain}
```

Please note that we found the flag `virtually-expand-domain` be absolutely mandatory as it rescales the domain to MPI's needs. We therefore use it in all of our experiments. The flag is detailed in Section 14.5 and notably the Peano handbook.

ExaHyPE inherits Peano's tree decomposition concept, i.e. its MPI variant cuts the global spacetree into subtrees recursively as long as there are ranks available that can then take control of these cut-out subtrees. In this first version, the cutting-out is done greedy in a first come first served (FCFS) manner: whichever rank decomposes into more ranks further is successful with its further decomposition while other ranks that want to split their tree with someone else run into a lack of free ranks.
Such an approach can yield a situation as depicted by the sketch above: a white rank being responsible for the root of the spacetree forks into a blue rank. Though the white rank continues to traverse the tree, the blue rank is faster and is served quicker with the two other remaining ranks (red and green). We obtain a very ill-balanced outcome. Once we pass the output into

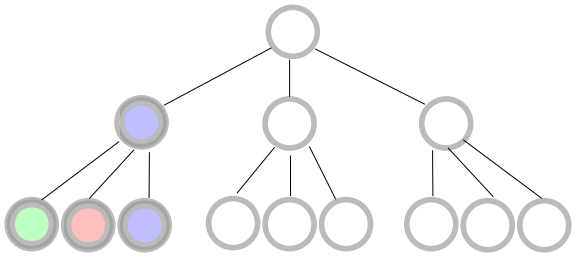
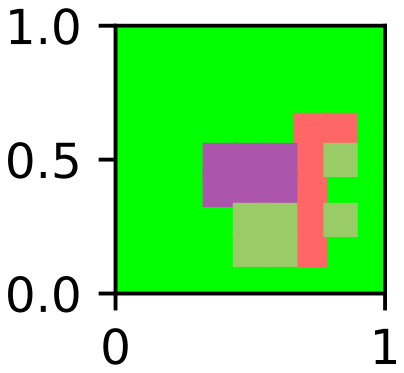
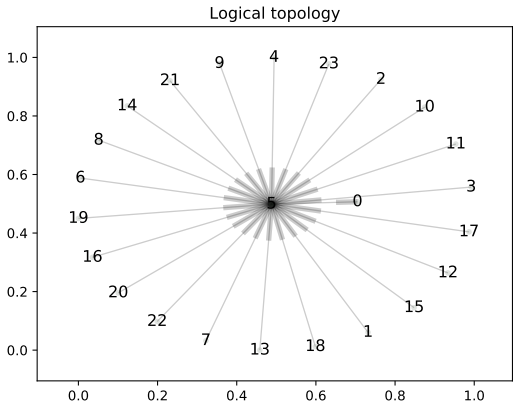


Figure 22.1: Spacetree encoding MPI ranks by color.

Peano's domain decomposition script, it clearly highlights the ill-balancing both through its logical topology tree (which is unreasonable deep) and the histogram of the work distribution.



(a) One splitting of a regular grid covering the unit square by a greedy tree decomposition with 24 ranks running on four nodes.



(b) The corresponding topology. Rank 1 is the most successful rank with booking additional workers. All other ranks find no more idle ranks left when they try to split their domain further.

Figure 22.2: Details on the pictured MPI balancing as shown in figure 22.1.

With this situation, we cannot scale up in terms of performance as we are radically ill-balanced—though adding more ranks might pseudo-randomly bring along a performance improvements—and notably soon run into memory footprint constraints.

22.2 A fairer load distribution

We change to the

```
configure = {hotspot,fair,virtually-expand-domain,ranks-per-node:6}
```

configuration next. It introduces two new/alternative concepts:

1. The tree is not greedily cut into pieces but always the rank with the largest load (the hotspot) is assigned additional ranks next. That is, the rank responsible for the largest tree cuts out subtrees from its tree. In the example below, where the x cell denotes an outer cell—cells outside the computational domain do not carry workload—the white rank (the global master) first splits up the workload between blue, red and green. Red and green thus become hotspots and are each given another rank as worker. We assume in

this sketch that we have six ranks in total. As one rank is reserved for load balancing and algorithm control (white), the load balancing stops now.

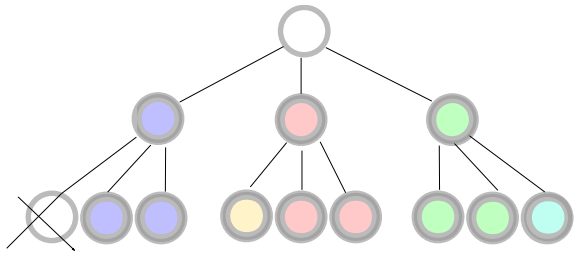


Figure 22.3: MPI Rationale: Fair

2. Fairness is built into the system by taking `ranks-per-node` into account. Per spacetree level, the load balancing tries to distribute the work among all compute nodes. Let two ranks (white/blue, red/yellow and green/light blue) be ran per compute node. As white is rank 0, the load balancing next employs ranks green and red from other nodes while distributing before it returns to blue.

The effect of the improved load balancing is a slightly better load balancing. It notably succeeds if the computational load can be evenly distributed among all compute nodes: If we run a totally regular grid (without the recommended virtual expansion) on $\geq 3^d + 1$ ranks, we should observe almost perfect scaling. The +1 results from the fact that ExaHyPE reserves one MPI rank for a global master always.

Yet, the scheme tends to yield a step pattern in the scalability curve: While it scales for $3^{d\ell} + 1$ ranks, adding a few more ranks does not improve the scaling further. It is only when we meet $3^{d(\ell+1)} + 1$ ranks that we have the next significant improvement in performance.

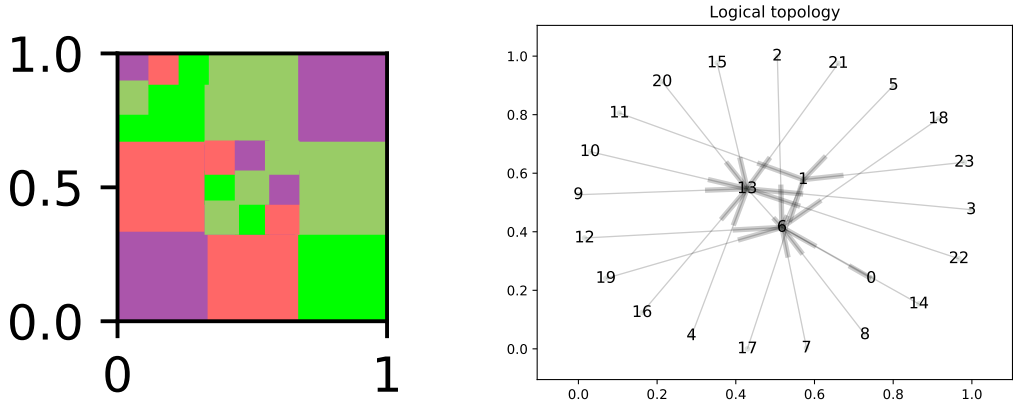
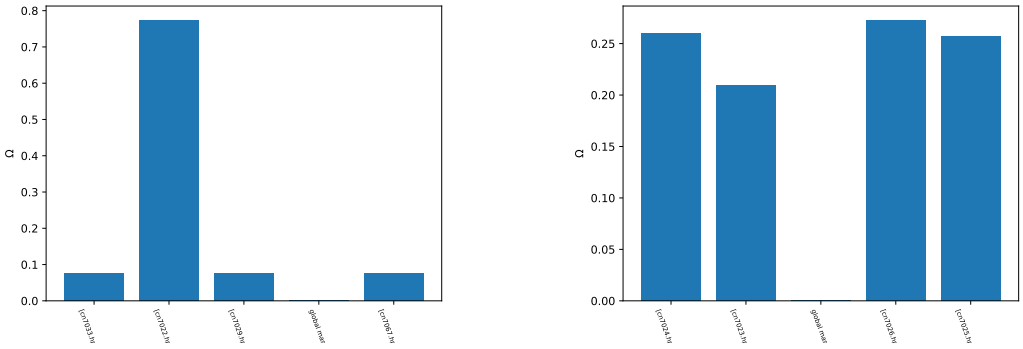


Figure 22.4: FCFS tree decompositions are replaced by a fair strategy. This yields better balanced trees (shallow compared to FCFS) and the subdomains obviously are better distributed.

22.3 Per-node workload distribution and diffusion along an SFC

For the subsequent idea, we weaken our notion of a fixed number of MPI ranks per node and instead assume that we have a number of MPI ranks per compute node that we want to use at least. These are called primary nodes. And then we have some more ranks as resiliency.



(a) Workload distribution among the four compute nodes for FCFS.

(b) A fair workload distribution.

Figure 22.5: Workload per Node: FCFS and FAIR

```
configure = {sfc-diffusion,hotspot,virtually-expand-domain,
  ranks-per-node:2,primary-ranks-per-node:1}
```

In the example above, such a setup means that we start our job with three MPI ranks per node, but we do accept/assume that usually only two of them are used. We stick to our hotspot cutting of the spacetree, but this new rank assignment strategy is an extension of the fair strategy. As long as some primary ranks remain available, it distributes those guys. Once the next hotspot decomposition strategy would use more than the primary ranks, the SFC diffusion allows one more load balancing step and then terminates the balancing. In this final wrap-up step, it tries to serve all those spacetree cut requests that can be fit into a SFC cut scheme, and it serves only those that improve the balancing per node slightly. It is one load diffusion step.

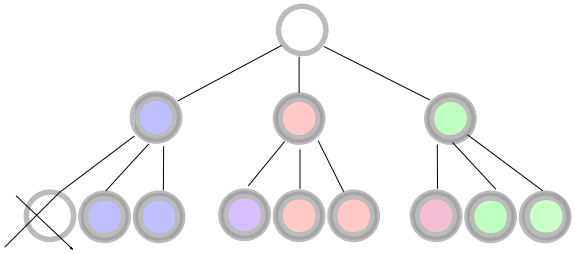


Figure 22.6: MPI rationale SFC

Let there be three compute nodes. For the example above our idea means: The global master splits up into blue, red, green. The partitioning is ill-balanced, yet there are no more primary nodes left. The load balancing thus starts to employ the remaining ranks. From all the ranks that might be able to decompose their partition further, it first allows green to deploy work to further nodes (not ranks). It books one additional rank from the node hosting the red primary rank. Next, the red rank is allowed to book one additional rank from the node that hosts the blueish ranks.

- With these concepts at hands, we have balanced the workload reasonably among the nodes and the subdomains deployed to one node are reasonably connected.

- We have not balanced the workload per rank.
- We have not balanced the ranks among the nodes.

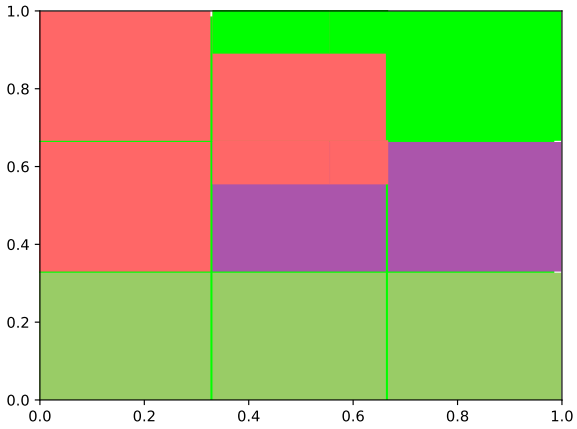


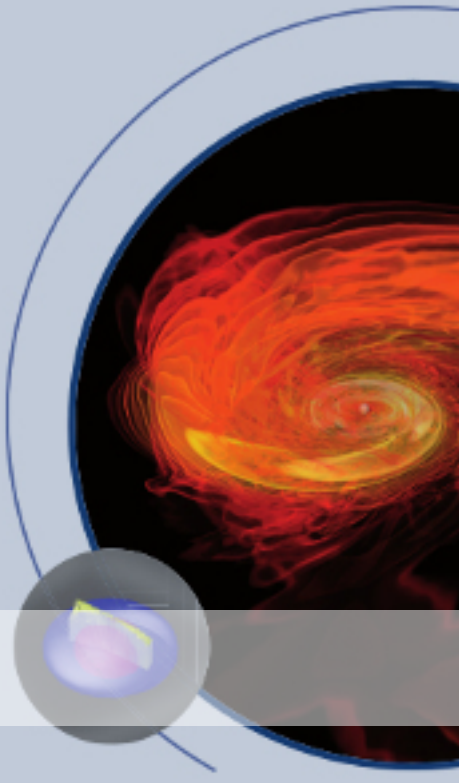
Figure 22.7: SFC-based decomposition. We observe a better connectivity of the subdomains.

The configuration so far thus lets us upscale in terms of memory. We do not run into out-of-memory problems that fast anymore. It does not yet allow us to upscale in terms of performance.

Some working setups:

- On Durham's 24-core Broadwells, we made good experience with six MPI ranks per node with four out of those acting as primary ranks.

VI Appendix



A. The ExaHyPE toolkit

The ExaHyPE toolkit is a small standalone Python program. The toolkit acts as sole interface for users, i.e. non-developers. We decided to realise it in Python 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 Python.
3. We prefer not to have several interfaces. With a Python front-end, we can directly generate the configuration rather than parsing configurations again.

Note that by default the toolkit will be called at runtime to validate the specification file. If you wish to avoid this behavior you can generate the glue code with the `-s` option (see below).

A.1 Dependencies

The toolkit itself requires Python 3.3 at least.
It also use some python libraries that can be either directly installed through pip or downloaded as a git submodule from ExaHyPE with the provided script

```
./Submodules/updateSubmodules.sh
```

A.2 Running the toolkit

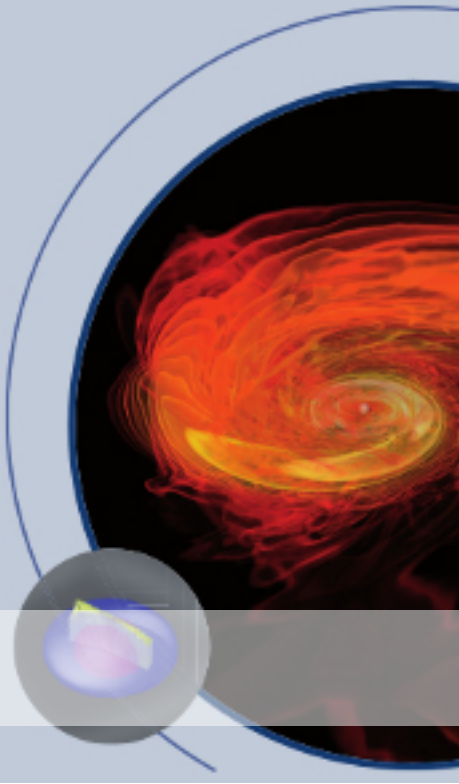
To run the toolkit, simply use the provided bash scrip

```
./Toolkit/toolkit.sh
```

The bash script is a wrapper over a Python call. It will automatically check the presence of all required dependencies.
The toolkit itself can run with multiple options. The main ones being:

- `-h`. Print the help and the list of available options.

- `-v`. Run in verbose mode, the toolkit is quiet by default.
- `-o`. Validate the given specification file. Do not generate any ExaHyPE glue code.
- `-j`. If the provided specification file isn't in JSON, the toolkit will generate a translated one. Can be combined with `-o` to only translate a specification file
- `-s`. The toolkit won't be called at runtime but the specification file used at runtime has to be in JSON. You can use `-jo` to (re)generate it



B. FAQ and further comments

This chapter collects a number of learned facts which have not found a better place in the document. The Frequently asked questions (FAQ) are divided into two sections: One for the developers who write code, one for the users who run code. The section on Bugs, limitations and compilation problems is another camouflaged FAQ section collecting known problems which are either the fault of ExaHyPE's authors or third party library authors.

B.1 Frequently asked developer questions

My code does not compile as _mm_... is not found. ExaHyPE uses aggressive memory prefetching to avoid latency penalties when it accesses the main memory through the caches. These techniques work on Intel systems as long as the system has the required libraries installed. If you run into problems, please add the parameter `-DnoTBBPrefetchesJobData` to your compile flags.

How do I link my own libraries against ExaHyPE? ExaHyPE uses the two environment variables `COMPILER_CFLAGS` and `COMPILER_LFLAGS` to build up all settings for the actual compile. If you add your own project settings to these variables (via `export` for bash, e.g.), then ExaHyPE will appends its own values to your settings.

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 9.3) 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 year header

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

initialise it and then use the `logInfo` macro:

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

[...]

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

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

  [...]
}
```

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

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 5). 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 an exact 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.

Can I create an executable with debug information that is not as slow as the code variant with assertions? Retranslate with `MODE=Profile`, e.g., or set the compiler manually with `EXAHYPE_CC=$CC" -g"`.

I haven't found the option to enable periodic boundary conditions. This is because we don't have nonlocal boundary conditions available in ExaHyPE. 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. If periodic boundary conditions are required, you have to dump (plot) the values yourself into a data structure and then impose these values manually through the adjust solution.

To be continued ...

B.2 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 however is always 3^i where i is the depth. While these statements hold for serial codes, parallel codes even might have other, prescribed, subdivision ratios. That is, ExaHyPE is tied to some resolutions and always chooses the coarsest resolution that still is finer or equal to a prescribed mesh size. Examples for possible mesh values for a serial code are given by table B.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 $\Delta x_{\max}$.

d	N	Δx	$\Delta x_{\max}$
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 B.1: Possible unigrid (regular grid) layouts with ExaHyPE. All quantities refer to the unit cube/square.

In practice, all your postprocessing has to be able to cope with the mesh size ExaHyPE gives you. In return, you can be sure that ExaHyPE always chooses mesh sizes that are at least as fine as required and that it chooses mesh sizes that are efficient w.r.t. runtime.
If the exact mesh size/character 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 `global-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 5) 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. As such, it is more of a developer question than a user question.

To be continued ...

B.3 Bugs and compilation problems

ExaHyPE yields invalid, oscillating results when I use MPI. Recompile with `MODE=asserts` and validate that the ping pong test passes.

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). If even a simple ping pong test fails—we've seen this only for Intel MPI so far—please translate with the four compiler flags:

```
-DnoParallelExchangePackedRecordsAtBoundary
-DnoParallelExchangePackedRecordsBetweenMasterAndWorker
-DnoParallelExchangePackedRecordsInHeaps
-DnoParallelExchangePackedRecordsThroughoutJoinsAndForks
```

Or turn packing off completely via:

```
-DnoPackedRecords
```

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 guidebook[1] 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.

I obtain a linker error undefined reference to symbol 'pthread_mutexattr_settype ...'. We have seen this error for some TBB installation on some platforms. Usually adding the pthreads to the linker command resolves the issue, i.e. add `PROJECT_LFLAGS+=-lpthread` to your settings.

When I try to compile with MPI, I get errors mentioning undefined `MPI_CXX` datatypes. In this case, ExaHyPE might still compile and work properly if you add

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

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

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 guidebook[2] 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 D.2. It is an IBM-based system where we could identify this issue and resolve it.

I try to compile with TBB but get a linker error alike "undefined reference to tbb::..." This error suggests that some environment variables are not set properly. See footnote on page 79 (Section 13).

Toolbox fail with messages alike 'Could not allocate metaspace: 1073741824 bytes'. See the remarks in Section **??** on the VM configuration.

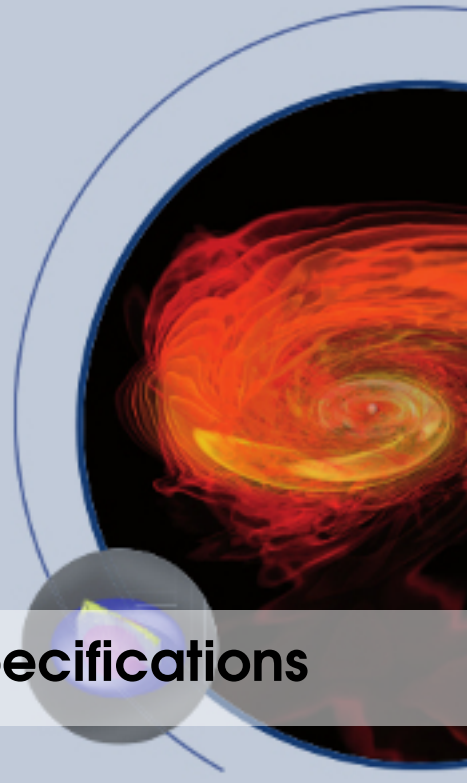
After a successful build, I obtain a parser error. The message should read similar to 'exahype::parser::Parser::getDomainSize() dimension: value notoken in specification file does not match -DDim2 switch in Makefile' even though exactly the same config file as the one passed to the toolkit is used. We have experienced this problem with GCC 4.8.5 20150623. If you update to 4.9, you should be fine[3].

To be continued ...

[1] `www.peano-framework.org`

[2] `www.peano-framework.org`

[3] Thanks are due to Ekaterina Tutlyaeva from the RSC Group for uncovering this issue.



C. 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. See figure C.1 for a quick overview of the available file formats.

C.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.

Overviews on

ExaHyPE volumetric file formats

Time comes on top
All file formats write d-dimensional data for each time (step) t, ie. you can think of this document as sketching files for a single time and writing repeated similar shaped data into one larger file or several files, one for each timestep, finally holding d+1 dimensional data.

All formats are lists of something
All formats presented on this page save lists of points or blocks. All formats represent the same data – with a difference in meta data and ordering. The „formats“ are in principle independent of the actual implementation details (ascii, binary, something like HDF5, …).

Key/Notation:

Table Shape (n,m,k,…) = n × m × k × ... matrix
d = Dimension of written file (or simulation). Typically: 2, 3
P = collocation points in each direction, ie. Typically: 3, 8, 10, 15
in ADERDG P = basisSize = order+1, in FV P=batchSize
N = Number of patches/cells in the simulation (w/o slicing) Typ: 1e4, 1e6
M = N * P^d = number of data points in the simulation Typ: 1e5, 1e8
W = Number of DOF written at each point. Typ: 5 (Hydro), 40 (Astro)

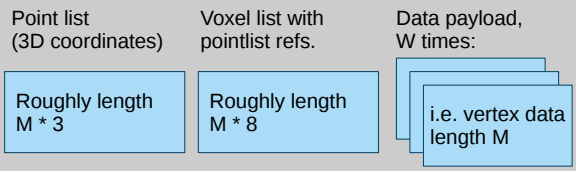
Block regular formats

These file formats exploit the regular grid structure in each block (patch or cell in ExaHyPE). Typically, we stick to Cartesian subgrids (Euler-Laguerre basis) but in theory one could store any subgrid coordinates. Due to saving most of the subcell grid overhead, typically saving 90% compared to the VTK file formats.

VTK file format

Fully implemented

Our VTK unstructured mesh defines the full geometry of all points. Thus we always describe voxels in 3D space. The only saving is indirection in the coordinates (point ↔ coordinate list). Here, I don‘t take the saving of a Cartesian projection into account, i.e. assuming plotting the Lagrange basis.



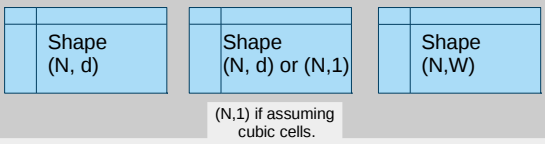
Advantages: Files really describe geometry in a widely-accepted standard and can be opened with any VTK compatible program.

Disadvantages: Files are extremely large, even in binary number format, because they explicitely store all M points grid position information („metadata“).

FLASH-like format (Roland Haas)

hypotethical

A *comparable* file format is written by FLASH code (http://flash.uchicago.edu). Is subject to be tested whether it is more storage-efficient.



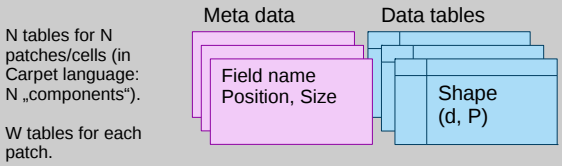
Desired advantages: Tables are substaintally larger than in the block-regular formats, we need N times less tables. Bigger continous tables are faster to read deal with. Can exploit HDF5 table compression.

Disadvantages: Has to use HDF5 extendible datasets or cache at memory (similar as assumably the peano VTK writers work). New file format, there exist no readers except we find the specs of the particular FLASH format and adopt the format similar to how it was done in CarpetHDF5.

Block-regular CarpetHDF5 Format

Fully implemented

CarpetHDF5 saves block-regular MPI patches („components“) in vertex-centered big regular grids.
Array of structures: Only *one* field (written unknown) per grid!



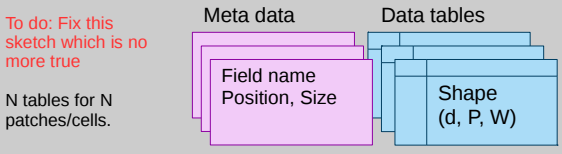
Advantages: File format is widely adopted (Visit, Amira, our group‘s tools).

Disadvantages: In ExaHyPE, file format is abused: Typically the number of number of components = number of MPI ranks is like 80-400 while in ExaHyPE the number of patches can be like 1e5. Serially reading programs will be unusable slow. We also still store way too much meta data HDF5 data tables are extremely small, thus we can neither make advantage of the HDF5 compression nor of quick readin for instance with Pythons PyHDF5. Files can be bigger than VTK files with same data.

Block-regular Peano Format

First Implementation availabe

Currently not really documented. Exploits several techniques such as indirection of coordinates which makes it extremely storage efficient. Also exploits ExaHyPEs Structure of Arrays approach:
All written quantities go in one dataset. Specification is quite implementation-agnostic, there are VTKish ascii/binary proposals as well as an ongoing HDF5 implementation (?).



Advantages: It‘s the natural file format for ExaHyPE (as CarpetHDF5 is for Carpet), representing the internal way how data in cells are stored (and thus probably suitable for checkpointing). Tables are substaintally larger than in the CarpetHDF5 format, we need W times less tables.

Disadvantages: New file format, there exist not yet readers (we could adopt the CarpetHDF5 writer if the HDF5 implementation was quite similiar).

Figure C.1: A sketch on ExaHyPE volumetric file formats, their advantages and disadvantages as well as their internal structure

- **Point time step data.** After the cell types, you find the time steps of the point data. Per vertex, we track at which time stamp its data is written if your plotter has to write data. Otherwise, the section is omitted. Per vertex, we write one scalar. For global time stepping, all values are equal. For adaptive time stepping, they might differ as the data gives the real time stamp of the data, not the simulation time when the data has been written.
- **Point data.** After the time stamp, you find per vertex one vector holding all the vertex data.
- **Cell time step data.** Analogous to vertices.
- **Cell data.** Analogous to vertices.

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

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

C.1.1 VTK cell data

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

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

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

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

C.1.2 VTK point data

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

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

Cell ID	**Cell Type**
0	Pixel
1	Pixel
2	Pixel
…	

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

C.2 HDF5 in ExaHyPE

C.2.1 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.

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"
```

Warn or crash if plotter fails

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 a 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"
```

C.3 Peano block regular fileformat

This file format is still subject to change, thus it is not documented in this current version of the guidebook. Details can however be found in the Peano guidebook from `www.peano-framework.org`. Besides further documentation efforts, we are currently developing a Paraview plugin for this particular file format.

C.4 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 $\mathrm{d}x$) possible. Each dataset holds a number of attributes which indicate the position $\vec{x}$ and point spacing $\mathrm{d}\vec{x}$ 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.

C.4.1 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.

C.4.2 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 Visit[1]. 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 Amira[2].

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.

scivis, scidata are two standalone Python libraries written by D. Radice to parse CarpetHDF5 files[4] and to generate pictures and movies in a batch step[5].

PostCactus is a standalone Python code written by W. Kastaun [6] 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. Guercilena [7] 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.

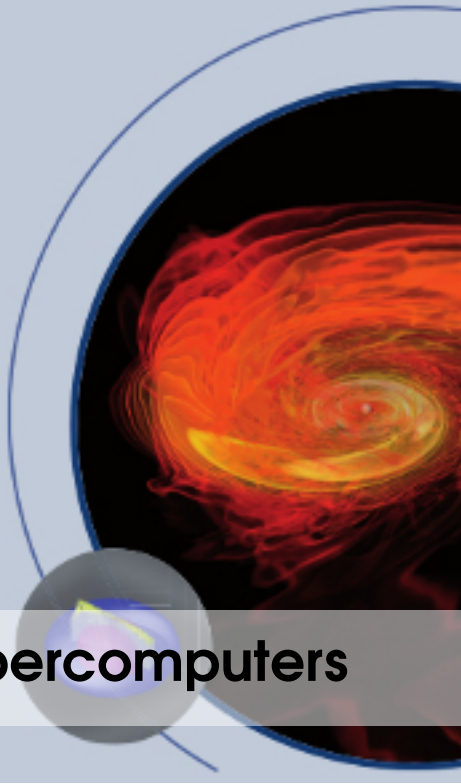
Inspect the Cactus page on Visualization for further references.

C.5 The FlashHDF5 output format in ExaHyPE

The FlashHDF5 output file format is inspired by the file format written out by the FLASH code[8]. There are a number of analysis tools for this file format, ie. QuickFLASH[9] 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.

[1] https://wci.llnl.gov/simulation/computer-codes/visit
[2] http://www.amira.com/
[3] https://bitbucket.org/dradice/pygraph
[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/



D. Running ExaHyPE on some supercomputers

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

D.1 Hamilton (Durham's local supercomputer)

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

```
module load intel/xe_2018.2
module load intelmpi/intel/2017.2
module load gcc/7.4.0
module load python/3.6.8 # for toolkit
```

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_2018.2
```

```
module load intelmpi/intel/2017.2
module load gcc/7.4.0

setenv I_MPI_FABRICS "tmi"


export I_MPI_FABRICS="tmi"

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.

Hamilton relies on Omnipath. Unfortunately, the default fabric configuration of Intel MPI seems not to work properly for ExaHyPE once the problem sizes become big. You have to tell MPI explicitly which driver/fabric to use. Otherwise, your code might deadlock. One version that seems to work is `dapl` chosen by

```
export I_MPI_FABRICS="dapl"
```

While `dapl` seems to be very robust, we found it slightly slower than `tmi` as used in the script above. Furthermore, it needs significantly more memory per MPI rank. Therefore, we typically use `tmi` which however has to be set explicitly via `export` on Hamilton.

One of the big selling points of Omnipath is that it is well-suited for small messages sizes. Compared to other (Infiniband-based) systems, it thus seems to be wise to reduce the package sizes in your ExaHyPE specification file. Notably, we often get improved performance once we start to decrease `buffer-size`.

D.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 switch intel/18.0
module switch tbb/2018
module switch gcc/5
```

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 switch intel/18.0
module switch tbb/2018
module switch gcc/5
```

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 case[1].

D.3 Archer's KNL partition (EPCC supercomputer)

Archer's default Java version does not meet ExaHyPE's requirements and the Java configuration does not provide the toolkit with enough heap memory (cf. Section **??**). Furthermore, we haven't used the Cray tools yet but stick to Intel and there have to load a well-suited GCC version manually:

```
module load java/jdk1.8.0_51
module swap PrgEnv-cray PrgEnv-intel
module load gcc
```

To accomodate the toolkit, we use the modified Java invocation:

```
java -XX:MaxHeapSize=512m -jar Toolkit/dist/ExaHyPE.jar
```

For shared memory support, we encountered three issues:

1. We should use EPCC's compiler macro `CC` instead of manually invocations of the compilers.
2. The module does not initialise the `TBB_SHLIB` variable that we use in our script. So we have to set it manually.
3. The default compiler behaviour links all libraries static into the executable. However, the TBB libs are not available in their static variant. To change this behaviour, we had to instruct the linker explicitly to link against shared memory library variants.

Overall, these three lines fix the behaviour:

```
export EXAHYPE_CC=CC
export TBB_SHLIB="-L/opt/intel/compilers_and_" \
  "libraries_2017.0.098/linux/tbb/lib/intel64/gcc4.7 -ltbb"
```

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

```
export CRAYPE_LINK_TYPE=dynamic
```

Similar to SuperMUC, we observe that a plain launch of executables through `aprun` *does not allow the codes to exploit shared memory parallelism.* We explicitly have to unlock the cores for the scripts in the run command through

```
aprun -n ... -d coresPerTask ... -cc depth
```

where `-cc` configures the pinning. According to the Archer documentation, this configuration still does not enable hyperthreading. If hyperthreading is required, we have to append `-j 4` to the invocation, too.

D.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 thatight `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.

D.5 CoolMUC 3

LRZ's KNL system CoolMUC 3 drives Omnipath as well. Therefore, ensure that you set the MPI fabric properly as soon as you use more than one node. Otherwise, ExaHyPE will deadlock:

```
export I_MPI_FABRICS="tmi"
```

D.6 Hazelhen (Cray)

Cray may configure the intel compiler to link in all libraries statically but TBB by default is not built statically so add the following to the `TBB_SHLIB`

```
-dynamic -ltbb
```

i.e. before the link command. The environment used is as follows

```
module switch PrgEnv-cray PrgEnv-intel

export COMPILER="Intel"
export DISTRIBUTEDMEM=MPI
export EXAHYPE_CC="CC"
export EXAHYPE_FC="ftn"
export MODE="Release"
export SHAREDMEM=TBB
export TBB_PATH=$INTEL_PATH/tbb
export TBB_INC="-I${TBB_PATH}/include"
export TBB_SHLIB="-L${TBB_PATH}/lib/intel64/gcc4.7␣-ltbb"
export CRAYPE_LINK_TYPE=dynamic
```

To run a simulation on Hazelhen, the following are required for the submit script, in addition to the normal variables

```
#PBS -m abe
#PBS -l nodes=5:ppn=24
#PBS -q test


module switch PrgEnv-cray PrgEnv-intel
module swap intel intel/18.0.1.163


export TBB_PATH=$INTEL_PATH/tbb
export TBB_INC="-I${TBB_PATH}/include"
export TBB_SHLIB="-L${TBB_PATH}/lib/intel64/gcc4.7␣-dynamic␣-ltbb"
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$TBB_PATH/lib/intel64/gcc4.7

export KMP_AFFINITY=disabled

aprun -cc none -n 30 -N 6 -d 4 ./ExaHyPE $SPECFILE
```

where n is the number of MPI ranks, N is the number of nodes, d is the number of threads.

D.7 Frankfurt machines

For the machines

- Generic Ubuntu laptop
- Iboga
- LOEWE
- FUCHS
- Hamilton
- SuperMUC

please see the configuration settings in the `ClusterConfigs` directory in the `Miscellaneous` in the main repository.

D.8 git on clusters

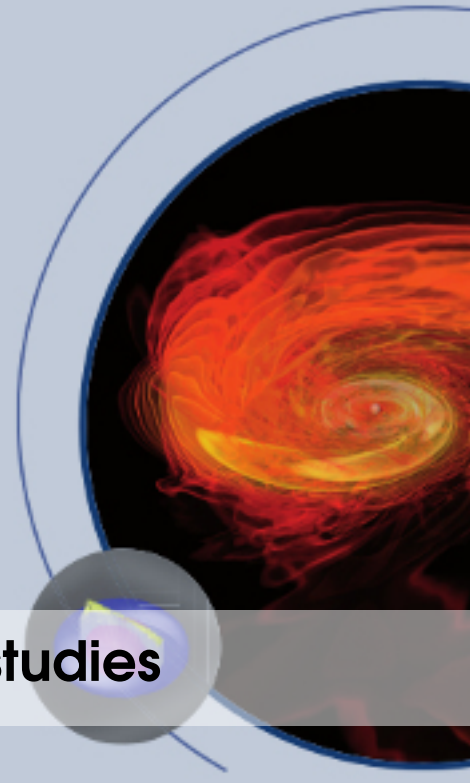
Some compute clusters, such as SuperMUC or Hazelhen, do not allow access to the outside world but instead require sshing in through known machines. To access git through these machines do the following. On your local machine add the following to the `.ssh/config` file

```
RemoteForward $PORT_NUM gitlab.lrz.de:22
```

where you specify `PORT_NUM` as a port number. Then on the compute machine, create a new ssh key and add it to your profile on Gitlab. Then add the following to the `.ssh/config` file

```
IdentitiesOnly yes

Host gitlab.lrz.de
  Hostname localhost
  Port $PORT_NUMBER
  User $USERNAME
  IdentityFile ~/.ssh/$PRIVATE_KEY
```

where `USERNAME` and `PRIVATE_KEY` are your username required to log into the gitlab and private key file respectively.



E. Performance and parameter studies

In this section, we collect some remarks on how to determine the performance of ExaHyPE applications.

E.1 Improving and measuring shared memory scalability

- For all speedup measurements, it is recommended to specify a fixed number of time steps (`time-steps=<int>`) instead of the simulation end time.
- If you turn predictor or update background jobs on, TBB may still use two cores during the time stepping and the mesh refinement iterations even if you specify that you only want to use a single TBB thread. You will then measure a runtime which is close to the one when specifying two TBB threads. Therefore, turn predictor and update background jobs off when measuring single-core runtimes.
- For single-node runs with uniform meshes, it is often more efficient to use parallel fors and no prediction or update background jobs. Spawning of background jobs incurs additional overhead. Turn them back on if you use AMR or MPI.
- When performing any scalability studies be aware that your application can only exploit multiple cores or nodes if your problem size is big enough. If the number of cells is small, you cannot expect that work is distributed efficiently and fairly among the used hardware processing units.
- For low orders or smaller PDEs, it often make sense to put ADERDG's space-time predictor and space-time volume flux arrays on the stack.

E.2 Performing performance and parameter studies with `sweep.py`

One of the major issues in High-Performance Computing has been identified as the difficulty to reproduce performance studies. ExaHyPE ships a tool which allows performing easily reproducible and portable performance studies. The tool named `sweep.py` can be found in folder `ExaHyPE-Engine/Benchmarks/python`.

`sweep.py` uses a single configuration file as input where you specify a parameter range, environment variables, and the hardware resources you want to use. You further specify which parameter variation requires building a new executable. Furthermore, the configuration file refers to a specification file template, and a job script template. The templates hold placeholders which have to match the parameter keys in the configuration file.

Building executables and submitting jobs

We start with building all required executables via:

```
<path_to_sweep>/sweep.py <path_to_configuration_file> build
```

During the build process, it is ensured that the parameters specified in the configuration map to a placeholder in the specification file template.

After building the required executables, the next step is to generate the job scripts. This is done via:

```
<path_to_sweep>/sweep.py <path_to_configuration_file> scripts
```

Again, consistency checks are performed.

Finally, the jobs are submitted via:

```
<path_to_sweep>/sweep.py <path_to_configuration_file> submit
```

Before submission, `sweep.py` checks if all executables and scripts exist and if all mandatory placeholders have been placed into the job script template. Furthermore, the tool checks if a log filter file named `exahype.log-filter` was placed into the project repository. Log filters are used to minimise the log output written by the application. Output can become a bottleneck in parallel applications. If any of the mentioned checks fails, the submission process is stopped and an error message is printed out.

Cancelling jobs and deleting files

Per batch of experiments, `sweep.py` memorises the jobs you have launched in a subfolder <myoutputfolder>/history. They can be all cancelled with:

```
<path_to_sweep>/sweep.py <path_to_configuration_file> cancel
```

If you have submitted multiple experiments in a row using the same output folder, you can find the configuration files in the <myoutputfolder>/history, too. They will have a hashed name though. This way you can also cancel the jobs submitted for previous experiments.

If you want to delete the whole output folder including executables, scripts, history and results, run:

```
<path_to_sweep>/sweep.py <path_to_configuration_file> cleanAll
```

You can delete individual subfolders via:

```
<path_to_sweep>/sweep.py <path_to_configuration_file> cleanBuild
<path_to_sweep>/sweep.py <path_to_configuration_file> cleanScripts
```

```
<path_to_sweep>/sweep.py <path_to_configuration_file> cleanResults
<path_to_sweep>/sweep.py <path_to_configuration_file> cleanHistory
```

Parsing runtimes

After completion of all jobs (or of some jobs if you want to have a peek), you can run

```
<path_to_sweep>/sweep.py <path_to_configuration_file> parseAdapters
```

which tells you which jobs have been processed successfully and which ones have not. However, the main purpose of this subprogram is to parse the measured CPU and real time spent in the different phases (*adapters*) of the algorithm from the output files written by your jobs. The parsed data is put into a CSV table which can be easily edited with any spreadsheet software.

Depending on the solver implementation (fused algorithmic phases vs. straightforward implementation using separate predictor, Riemann, and corrector loops), the algorithmic phases constituting a time step differ. `sweep.py` comes with another subprogram `parseTimeStepTimes` for post-processing the output of the `parseAdapters` subprogram. This subprogram is aware of the two different solver implementations when computing the CPU and real time spent in the time stepping adapters. Furthermore, if you have run the same experiment multiple times, it will compute the measured minimum, maximum, and mean times. It will also include the standard deviation of the measured times into the generated CSV table.

Parsing LIKWID metrics

It is possible to instruct `sweep.py` to use LIKWID's `perfctr` wrapper. In this case, you specify the LIKWID metric groups you want to measure in the `[general]` section of the configuration file, e.g.:

```
[general]
...
likwid = MEM_DP,L2,L3CACHE,L2CACHE
```

Note that these measurements will then be performed by the same job. You might need to increase the walltime in this case.

After the experiments have been run, parsing the metrics can be with just another parser subprogram:

```
<path_to_sweep>/sweep.py <path_to_configuration_file> parseMetrics
```

E.2.1 Example: ElasticWave3D - shared memory scalability

Below you will find files used for an comprehensive performance study examining the multi-core scalability of the linear ADER-DG application `ElasticWave3D`
(path: `ApplicationExamples/Linear/ElasticWave3D`). The example is used to detail the sections of the configuration file and to show examples of the job script template and the specification file template. The configuration file has the structure of an INI file and is processed by python3's built-in INI file parser.

The performance study varies the number of cores (and background job consumer tasks), the polynomial order, the ADER-DG implementation ("fused" vs. "nonfused"), if predictor background jobs are used, if update background jobs are used, if zero or two levels of AMR

should be used, and so on ... The parameter keys and their associated list of values can be specified as single parameter key with single parameter value:

```
<key0> = <value0>
```

or as a single key with multiple values

```
<key> = <value0>,<value1>
```

or as single key with multiple quoted values

```
<key> = "<value0>","<value1>"
```

or as a single key with a mix of quoted and unquoted values:

```
<key> = "<value0>",<value1>,"<value2>"
```

The parameter space is spanned as a Cartesian product of the single parameter key-value list combinations. Each parameter key is a dimension in the parameter space. Different values for a certain parameter key are separated by commas (','). You can further put parameter values in quotation marks ('"') if a parameter contains commas itself or in order to prevent trimming of leading or trailing whitespaces.

For each parameter combination, one experiment will be performed. Elements of the parameter space may be directly mapped to a job script or whole subspaces may be mapped to a job script: While the parameter combinations in `parameters` are directly mapped to a job script, the parameters in `parameters_grouped` are grouped onto the same job script.

It is further possible to prescribe and vary environment variables. For each different configuration of environment variables, a new executable is created. This prevents you from accidentally using a different compiler or build mode when rerunning an experiment. You specify environment variables exactly in the same way as parameters.

The workflow from which we extracted the files below assumes that the script is placed into the project folder. For convenience, we typically symlink `Benchmarks/python/sweep.py` into the project folder, too.

The configuration file used for performance study is given below. Comments start with an ; or # symbol.

```
[general]
exahype_root = <mypath>/ExaHyPE-Engine
project_name = Elastic
project_path = ApplicationExamples/Linear/ElasticWave3D/

spec_template = %(project_path)s/elastic.exahype2-template
job_template = %(project_path)s/hamilton7.job-template

output_path = %(project_path)s/tbb-scaling
make_threads = 24

run_command = ""
```

```
job_submission = sbatch --qos=no_core_limit
job_cancellation = scancel

compile_time_parameters = order,tempArrays,kernels

; you can also perform likwid measurements
; likwid = MEM_DP,L2,L3CACHE,L2CACHE

[jobs]
time = 12:00:00
mail = myemail@address.com

num_cpus = 24
; total ranks (!) x nodes x { cores : consumerTasks }
ranks_nodes_cores = 1 x 1 x {24:12,16:8,12:6,8:4,4:2,2:1,1:0}

; tag for the current run
run = 0
; any number is possible as tag; for each tag, a run is performed
; run = 1,2,3

[environment]
; for each environemnt variable combination, an executable is built
EXAHYPE_CC = icpc
COMPILER = Intel
MODE = RELEASE
DISTRIBUTEDMEM = NONE
SHAREDMEM = TBB
USE_IPO = on
PROJECT_CFLAGS = ""

[parameters]
; a job is created for each configuration
architecture = hsw
dimension = 3
; for each order an executable is built as order is listed in 'compile_time_parameters'
order = 3,4,5,6,7,8
timeSteps = 50

fused = false,true
predictorBackground = false,true
updateBackground = false,true

profilingTarget = whole_code

maximumMeshDepth = 0,1

[parameters_grouped]
; runs for these parameters are grouped onto the same job
; the number of generated jobs thus equals the number of
; combinations of parameters in section [parameters]
prolongationBackground = false
amrBackground = true
tempArrays = stack
minJobsPerRush = 2
```

```
maxJobsPerRush = 2147483647
kernels = optimised
timeStepping = globalfixed
batchFactor = 1.0
maximumMeshSize = 1.0
highPrioJobProcessing = one_at_a_time
lowPrioJobProcessing = run_if_no_high_priority_job_left
```

The matching specification file template is given below. It has to be in the novel ExaHyPE JSON specification file format. You can generate the new format with the `-j` switch from the original ExaHyPE specification file format.

```
{
  "project_name": "Elastic",
  "paths": {
    "exahype_path": "./ExaHyPE",
    "output_directory": "./ApplicationExamples/Linear/ElasticWave3D",
    "peano_kernel_path": "./Peano"
  },
  "architecture": "{{architecture}}",
  "computational_domain": {
    "dimension": {{dimension}},
    "offset": [
      -12.0,
      0.0,
      -12.0
    ],
    "time_steps": {{timeSteps}},
    "width": [
      24.0,
      24.0,
      24.0
    ]
  },
  "profiling": {
    "profiling_target": "{{profilingTarget}}"
  },
  "shared_memory": {
    "cores": {{coresPerRank}},
    "background_job_consumers": {{consumerTasks}},
    "high_priority_background_job_processing" : "{{highPrioJobProcessing}}",
    "low_priority_background_job_processing" : "{{lowPrioJobProcessing}}",
    "min_background_jobs_in_one_rush" : {{minJobsPerRush}},
    "max_background_jobs_in_one_rush" : {{maxJobsPerRush}},
    "properties_file": "sharedmemory.properties",
    "autotuning_strategy": "dummy",
    "thread_stack_size": 16777216
  },
  "distributed_memory": {
    "timeout": 60,
    "load_balancing_type": "static",
    "buffer_size": 64,
    "load_balancing_strategy": "hotspot",
    "node_pool_strategy": "fair",
```

```
    "ranks_per_node": {{ranksPerNode}}
  },
  "optimisation": {
    "fuse_algorithmic_steps": {{fused}},
    "fuse_algorithmic_steps_factor": 0.99,
    "spawn_predictor_as_background_thread": {{predictorBackground}},
    "spawn_update_as_background_thread": {{updateBackground}},
    "spawn_prolongation_as_background_thread": {{prolongationBackground}},
    "spawn_amr_background_threads": {{amrBackground}},
    "disable_vertex_exchange_in_time_steps": true,
    "time_step_batch_factor": {{batchFactor}},
    "disable_metadata_exchange_in_batched_time_steps": true,
    "double_compression": 0.0,
    "spawn_double_compression_as_background_thread": false
  },
  "solvers": [
    {
      "name": "MyElasticWaveSolver",
      "order": {{order}},
      "maximum_mesh_depth": {{maximumMeshDepth}},
      "maximum_mesh_size": {{maximumMeshSize}},
      "type": "ADER-DG",
      "time_stepping": "{{timeStepping}}",
      "aderdg_kernel": {
        "basis": "Legendre",
        "implementation": "{{kernels}}",
        "allocate_temporary_arrays" : "{{tempArrays}}",
        "adjust_solution": "patchwise",
        "language": "C",
        "nonlinear": false,
        "optimised_kernel_debugging": [],
        "optimised_terms": [],
        "space_time_predictor": {},
        "terms": [
          "flux",
          "ncp",
          "material_parameters",
          "point_sources"
        ]
      },
      "variables": [
        {
          "multiplicity": 3,
          "name": "v"
        },
        {
          "multiplicity": 6,
          "name": "sigma"
        }
      ],
      "material_parameters": [
        {
          "multiplicity": 1,
          "name": "rho"
        },
```

```
      {
        "multiplicity": 1,
        "name": "cp"
      },
      {
        "multiplicity": 1,
        "name": "cs"
      }
    ],
    "point_sources": 1,
    "parameters": {
      "amr_regularization": false
    },
    "plotters": []
  }
 ]
}
```

Finally, you need to specify a job script template in order to perform the performance study on your supercomputer of choice. We ran the study on Durham University's `hamilton7` machine which uses the SLURM scheduler. A template for this machine may look as follows:

```
#!/bin/bash
# Mandatory parameters are:
# time, ranks, nodes,
# job_name, output_file, error_file,
# body
#
# Optional parameters are:
# tasks, coresPerTask, mail

#SBATCH --job-name={{job_name}}
#SBATCH -o {{output_file}}
#SBATCH -e {{error_file}}
#SBATCH -t {{time}}
#SBATCH --exclusive
#SBATCH -p par7.q
#SBATCH --mem=MaxMemPerNode
#SBATCH --ntasks={{ranks}}
#SBATCH --nodes={{nodes}}
#SBATCH --cpus-per-task={{coresPerRank}}
#SBATCH --mail-user={{mail}}
#SBATCH --mail-type=END
module purge
module load slurm
module load intel/xe_2017.2
module load intelmpi/intel/2017.2
module load gcc
module load likwid

export TBB_SHLIB="-L/ddn/apps/Cluster-Apps/intel/xe_2017.2/tbb/lib/intel64/gcc4.7 -ltbb"

export I_MPI_FABRICS="tmi"
```

```
{{body}}
```

E.3 Overwriting default compiler flags

To overwrite default compiler flags such as `-O3`, `-xHost`, or `-xCore-AVX` which will be added by ExaHyPE's Makefile, create an environment variable `PROJECT_CFLAGS` in the `[environment]` section in your `ini` file and add all flags to the right hand side, e.g.:

```
PROJECT_CFLAGS="-g -no-vec -no-simd"
```

The flags specified here will be appended to the default compiler flags. Therefore, they overwrite the preceding compiler flags.

E.4 Supercomputer build environments and job script templates

You find build environments for a range of supercomputers in the `ExaHyPE-Engine/Benchmarks/environment` subfolder. You find job templates for a range of supercomputers in the `ExaHyPE-Engine/Benchmarks/job-templat` subfolder.

E.5 Further post-processing of CSV tables

You can find further useful tools in the `ExaHyPE-Engine/Benchmarks/python` for post-processing the CSV tables generated by `sweep.py`. It is often convenient to symlink them directly into your project folder.

The `tableslicer.py` tool provides the following options:

```
This is tableslicer.py: A small tool for extracting columns from a table which
is scanned according to a filter.

positional arguments:
  table The CSV table to work with.

optional arguments:
  -h, --help show this help message and exit
  --filter FILTER [FILTER ...]
                        Specify a list of key-value pairs. Example:
                        ./tableslicer.py --filter order=3 maximumMeshDepth=3
                        ...
  --cols COLS [COLS ...]
                        Specifiy the list columns you want to read from the
                        rows matching the filter. Example: ./tableslicer.py
                        ... --cols cores realtime_min
  --min [MIN] Specify the column you want to determine the minimum
                        value of. All (filtered) rows with that value will be
                        written out. If you do not specify anything, the last
                        column will be used. Example: ./tableslicer.py
                        --filter .. --cols order maximumMeshDepth ... --min
                        order
  --max [MAX] Specify the column you want to determine the maximum
                        value of. All (filtered) rows with that value will be
                        written out. If you do not specify anything, the last
```

```
                    column will be used. Example: ./tableslicer.py
                    --filter .. --cols order maximumMeshDepth ... --max
                    order
  --header Write a header to the output file.
  --no-header Write no header to the output file.
  --compress Remove columns where the same value is found in every
                    row.
  --no-compress Do not remove columns where the same value is found in
                    every row.
  -s SORT [SORT ...], --sort SORT [SORT ...]
                    Specify a list of sorting key columns. Order is
                    important. Example: ./tableslicer.py ... ... --cols
                    fused cores --sort cores fused
  --input-delim [INPUTDELIM]
                    Specify the delimiter used in the input table.
  --output-delim [OUTPUTDELIM]
                    Specify the delimiter for the output table.
  --output OUTPUT The output file.
```

There is further the `speedupcalculator.py` which can be used as follows:

```
This is speedupcalculator: A small tool for computing speedups for data stored
in a CSV table. The last column of a table row is assumed to be the data
column. The first row is assumed to store the reference value if no reference
value is explicitly specified.

positional arguments:
  table The CSV table to work on.

optional arguments:
  -h, --help show this help message and exit
  --header Write a header to the output file.
  --no-header Write no header to the output file.
  --keys Include the key columns to the output.
  --no-keys Remove the key columns from the output.
  --data Include the original data column into the output file.
  --no-data Remove the original data column from the output file.
  --input-delim [INPUTDELIM]
                    Specify the delimiter used in the input table.
  --output-delim [OUTPUTDELIM]
                    Specify the delimiter for the output table.
  --reference [REFERENCE]
                    Specify a reference value > 0.
  -o [OUTPUT], --output [OUTPUT]
                    Output file
```

Computing speedups if table contains different data sets

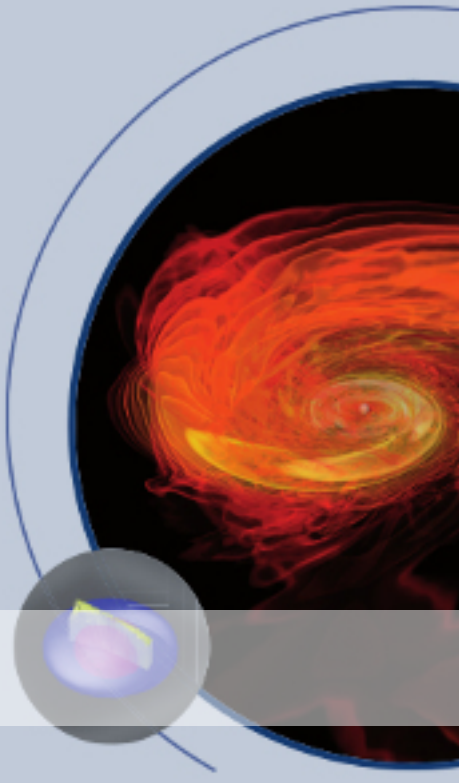
If you use `sweep.py`, you often pipe output from multiple different runs into the same results folder. Parsing these outputs will then create a big CSV table containing data sets for multiple orders, mesh sizes etc. In this case, you cannot use the speedup calculator tool directly as it will take the first column as reference value.

Here, a combination of the tableslicer and the speedupcalculator tool can do the trick. Assume your CSV table contains measurements for multiple polynomial orders and you want to compute the speedup for a specific one. In this case you can run, e.g.:

```
./tableslicer.py <mypath>/<myproject>-timestep-times.csv\
--cols order cores realtime_min --filter order=5 | ./speedupcalculator.py
```

Bash further allows writing for loops. Therefore, you can compute speedups for all considered orders (3,5,7 in our example) quite easily:

```
for o in 3 5 7; do ./tableslicer.py <mypath>/<myproject>-timestep-times.csv\
--cols order cores realtime_min --filter order=$o | ./speedupcalculator.py; done
```



F. Lists

F.1 List of authors

- Angelika Schwarz (TUM, Munich)
- Ben Hazelwood (Durham University)
- Dominic Etienne Charrier (Durham University)
- Fabian Guera (TUM, Munich)
- Jean-Matthieu Gallard (TUM, Munich)
- Leonhard Rannabauer (TUM, Munich)
- Luke Bovard (FIAS, Goethe University Frankfurt)
- Philipp Samfaß (TUM, Munich)
- Sven Köppel (FIAS, Goethe University Frankfurt))
- Tobias Weinzierl (Durham University)
- Vasco Varduhn (TUM, Munich)
- Anne Reinarz (TUM, Munich)

F.1.1 Detailed contributions

The editing on the guidebook is managed within the version control software `git`. That makes it in principle easy to track individual changes and to quantify the amount each author contributed. However, be *warned* that there is not a unique way of measuring authorship. We tested two methods which results are given in table F.1 and F.2, respectively. The most useful number is probably the total number of lines changed (`loc`). This does not tell anything about the originality, quality or information density of these edits. The number of commits (`coms`) is an estimate how frequently edits were made. The number of changed files (`fils`) is a first attemp to measure the number of added figures, especially in method II. Method I contains a neat overview about when respective authors made their first and last commit – note that several authors joined the project later while others left it already, this reflects also in these numbers.

Author	insertions	deletions	files	commits	lines changed	first	last
total	23962	7079	692	369	31041	2015-11-18	2018-05-09
Tobias Weinzierl	13286	4931	344	206	18217	2015-11-18	2018-04-24
Sven Köppel	5582	1080	203	62	6662	2016-03-04	2018-05-09
Dominic Charrier	4182	817	84	51	4999	2016-01-29	2018-04-25
Jean-Matthieu Gallard	400	133	19	18	533	2016-09-29	2018-05-07
Angelika Schwarz	274	53	22	16	327	2016-02-17	2016-07-12
Philipp Samfaß	95	2	2	2	97	2017-02-28	2017-03-07
Ben Hazelwood	66	1	1	1	67	2018-04-25	2018-04-25
ga96nuv	29	27	3	2	56	2016-08-02	2017-01-26
Vasco Varduhn	15	12	9	6	27	2016-03-07	2016-12-05
Luke Bovard	15	10	1	1	25	2018-02-20	2018-02-20
Fabian Guera	11	12	1	1	23	2016-06-19	2016-06-19
lrannabauer	4	0	1	1	4	2018-01-16	2018-01-16
Leonhard Rannabauer	2	1	1	1	3	2017-11-30	2017-11-30
Philipp	1	0	1	1	1	2017-03-08	2017-03-08

Table F.1: Authors contributions measured with method I.

```
Total commits: 387
Total files: 146
Total loc: 16882
+-----------------------+------+------+------+----------------+
| Author                |  loc | coms | fils |  distribution  |
+=======================+======+======+======+================+
| Tobias Weinzierl      | 8548 |  214 |   49 | 50.6/55.3/33.6 |
| Sven K?ppel           | 4492 |   67 |   54 | 26.6/17.3/37.0 |
| Dominic Charrier      | 3499 |   54 |   18 | 20.7/14.0/12.3 |
| Angelika Schwarz      |  195 |   17 |    7 |  1.2/ 4.4/ 4.8 |
| Ben Hazelwood         |   66 |    2 |    1 |  0.4/ 0.5/ 0.7 |
| Jean-Matthieu Gallard |   44 |   18 |    8 |  0.3/ 4.7/ 5.5 |
| Luke Bovard           |   15 |    1 |    1 |  0.1/ 0.3/ 0.7 |
| ga96nuv               |   11 |    2 |    2 |  0.1/ 0.5/ 1.4 |
| Vasco Varduhn         |    4 |    6 |    2 |  0.0/ 1.6/ 1.4 |
| lrannabauer           |    3 |    1 |    1 |  0.0/ 0.3/ 0.7 |
| Philipp Samfa?        |    3 |    2 |    2 |  0.0/ 0.5/ 1.4 |
| Leonhard Rannabauer   |    2 |    1 |    1 |  0.0/ 0.3/ 0.7 |
| Philipp               |    0 |    1 |    0 |  0.0/ 0.3/ 0.0 |
| Fabian Guera          |    0 |    1 |    0 |  0.0/ 0.3/ 0.0 |
+-----------------------+------+------+------+----------------+
```

Table F.2: Authors contributions measured with method II.

F.2 List of figures

F.3 List of tables

F.4 Open Todo Notes

Note F.1 This section is only included when the guidebook is not compiled in release mode.

F.5 References (Bibliography)

[1] Dumbser M, Guercilena F, Köppel S, Rezzolla L and Zanotti O 2017 *submitted to Physical Review D* (*Preprint* `http://www.arxiv.org/abs/1707.09910`)

[2] Dumbser, Zanotti, Loubere, Diot 2015. `https://arxiv.org/abs/1406.7416`

[3] Tecplot file format documentation. Look up.