ExaHyPE Guidebook

http://www.exahype.eu

May 23, 2016

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

Preamble

`ExaHyPE` is a collection of open source tools. It is mainly written in C++, Java and Matlab.

Dependencies and prerequisites

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

- `ExaHyPE` source code depends only on MPI and Intel's TBB or OpenMP if you want to run it with distributed or shared memory support. There are no further dependencies or libraries required. All examples from this guidebook run and have been tested with GCC 4.2 and Intel 12 or later. Earlier compiler versions might work.
- ExaHyPE's setup environment relies on Java, as parts of it come along as jar files.
- ExaHyPE's default build environment uses `make` and `tr`, i.e. only standard Unix commands.

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

Who should read this document

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

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

May 23, 2016
The `ExaHyPE` team

Contents

1. Download (and install)

ExaHyPE does not require any real installation. It is basically just a download. Please visit www.exahype.eu to get particular ExaHyPE source code variants. All remarks here refer to the latest ExaHyPE release.

Each release consists of three major source code packages:

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

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

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

A complete download/setup reads as follows:

```
> cd mywellsuiteddirectory
> wget http://www.exahype.eu/exahype.tar.gz
[...]
Saving to: exahype.tar.gz

exahype.tar.gz 100%[==========================>] 3.37M 929KB/s in 4.0s
[...]
> wget http://sourceforge.net/projects/peano/files/peano.tar.gz
[...]
Saving to: peano.tar.gz

peano.tar.gz 100%[==============================>] 3.37M 929KB/s in 4.0s
[...]
> wget http://www.exahype.eu/exahype.jar
[...]
Saving to: exahype.jar

exahype.jar 100%[==============================>] 3.37M 929KB/s in 4.0s
[...]
> tar -xzf exahype.tar.gz
> ls
Applications ExaHyPE exahype.jar exahype.tar.gz \peano\ peano.tar.gz
> tar -xzf peano.tar.gz -C \peano\
```

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

Dry run

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

```
> java -jar exahype.jar
```

This should give you the following welcome message:

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

 www.exahype.eu

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

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


ERROR: Please provide input file as argument
```

Using a newer/other Peano version (optional)

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

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

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

2. Setup a first experiment

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

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

A trivial project

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

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

  output-directory = ./Applications/trivialproject

  architecture = noarch

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

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

```
end exahype-project
```

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

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

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

```
make
```

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

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

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

A simple 2d Euler flow

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

For the actual computations, there are three types of kernels (PDE solver realisations) that can be used within `ExaHyPE`. You can either write the whole solver yourself, you can make `ExaHyPE` realise the solver while you specify the flux update operations, or you can use a pre-defined `ExaHyPE` kernel which gives you the best performance. Available options are described and detailed in Chapter **??**.

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

```
exahype-project MyEuler

  peano-path = ./Peano/peano
  tarch-path = ./Peano/tarch
  multiscalelinkedcell-path = ./Peano/multiscalelinkedcell
  sharedmemoryoracles-path = ./Peano/sharedmemoryoracles
  exahype-path = ./ExaHyPE
  output-directory = ./Applications/guidebook-chapter-2
```

```
  architecture = noarch

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


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

    plot vtk::ascii
      time = 0.0
      repeat = 0.05
      filename = ./solution
    end plot
  end solver

  shared-memory
    identifier = dummy
    cores = 2
    properties-file = sharedmemory.properties
  end shared-memory

  optimisation
    fuse-algorithmic-steps = on
  end optimisation

end exahype-project
```

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

```
#include "MySolver.h"

MyEuler::MySolver::MySolver( int kernelNumber):
  exahype::solvers::Solver("MySolver",exahype::solvers::Solver::ADER_DG,kernelNumber,5,3+1) {
}

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

// ...

void MyEuler::MySolver::adjustedSolutionValues(...) {
  if (tarch::la::equals(t, 0.0)) {
```

```
    const double GAMMA = 1.4;

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

bool Euler2d::MyEulerSolver::hasToAdjustSolution(...) {
  if (tarch::la::equals(t, 0.0)) {
    return true;
  }
  return false;
}
```

Actually, only proper initial conditions have to be provided. The remainder is pregenerated.

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

A remark on material parameters

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

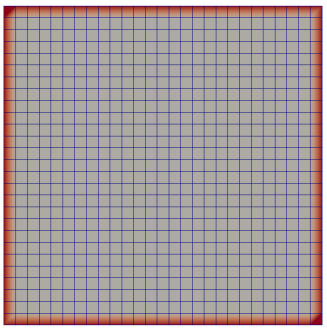
Plots

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

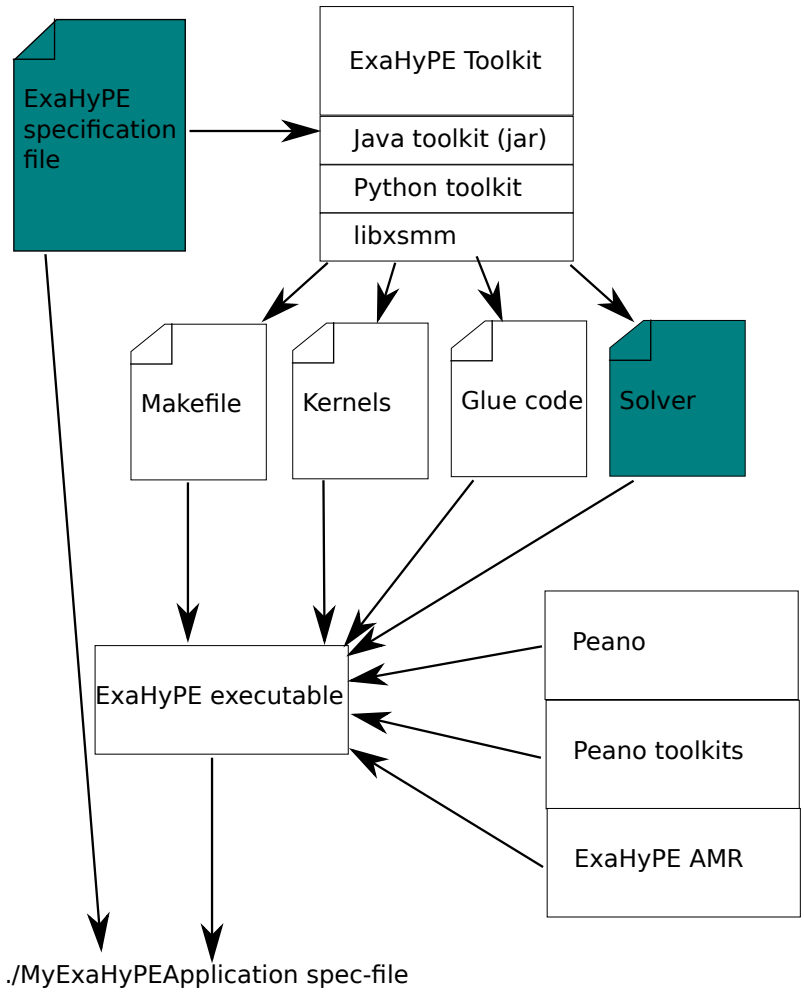
```
...
solver ADER-DG MySolver
  unknowns = 5
  parameters = 0
  order = 3
  kernel = kernel::euler2d

  plot vtk::binary
    time = 0.0
    repeat = 0.02
    filename = ./solution
  end plot
end solver
...
```

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

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



3. ExaHyPE architecture



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

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

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

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

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

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

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

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

4. ADER-DG solvers with user-defined kernels

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

For the Euler equations

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

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

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

which are discretised over Ω.

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

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

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

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

```
/**
 *Example code of Chapter 3 in ExaHyPE codebook
 */
exahype-project Euler2d
```

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

  output-directory = ./Applications/eulerflow2d

  architecture = noarch

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

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

    plot vtk::binary
      time = 0.0
      repeat = 0.05
      filename = ./solution
    end plot
  end solver
end exahype-project
```

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

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

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

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

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

4.1. Study the generated code

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

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

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

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

4.2. Setting up the experiment

To set up the experiment, we open the implementation file `Euler2d.cpp` of the class `Euler2d`. There are two routines (`getMinimumTreeDepth` and `adjustedSolutionValues` that allow us to setup the initial grid. We leave the default setting for the tree depth—we can use it later to set up adaptive initial grids as well—and focus on the initial values. The implementation of the initial values might look as follows[1]:

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

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

[1]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.

```
            ) *1.0e-3;
  }
}
```

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

- We only want to impose initial conditions so we check if the time `t` is zero with respect to some tolerance in the first line of the routine.

- While the unknown pointer `Q` itself does not give any hint about the actual size of the array it points at, we know by construction that we have five unknowns per grid point and thus choose the iteration space accordingly. Note that as usual for `C` type languages indexing of the array starts with zero.

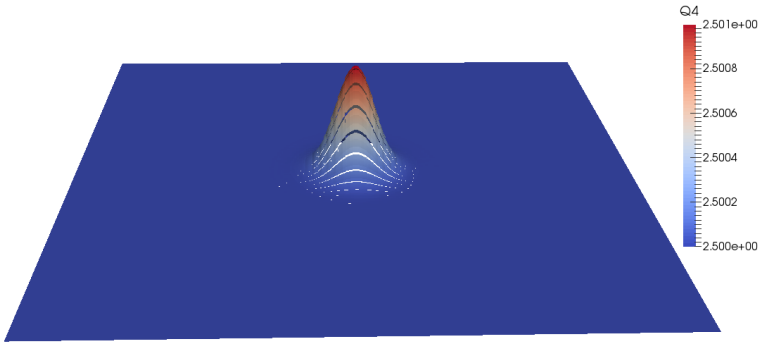
- The spatial position `x` is given as a pointer to a double array of size `dimension` which you have previously specified in the script `euler-2d.exahype`. Note that the indexing of the arrays starts here of course also with zero.

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

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

The routine's arguments `center` and `dx` refer to the center point of the cell and to its extent in each coordinate direction. The simulation time is here again denoted by `t`.

We conclude our experimental setup with a compile run and a first test run. This time we get a meaningful image and definitely not a program stop for an assertion, as we have set all initial values properly. However, nothing happens so far; which is not surprising given that we haven't specified any fluxes yet.



4.3. Realising the fluxes

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

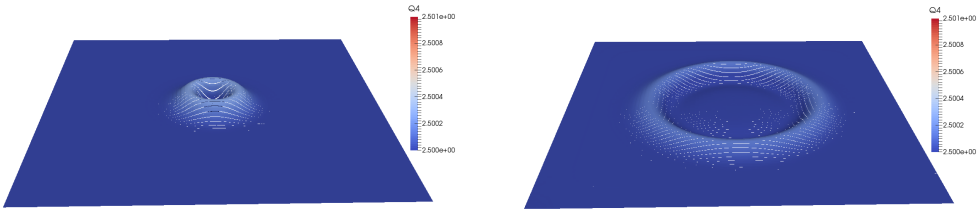
```
void Euler2d::MyEulerSolver::flux(
  const double* const Q,
  double* f,
  double* g
) {
  const double GAMMA = 1.4;

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

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

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

```
void Euler2d::MyEulerSolver::eigenvalues(
  const double* const Q,
  const int normalNonZeroIndex,
  double* lambda
) {
  const double GAMMA = 1.4;

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

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

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

  lambda[0] = u_n-c;

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

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

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

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



5. Shared memory parallelisation

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

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

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

Whenever you configure your project with shared memory support, the **default** build variant is a shared memory build (with TBB or OpenMP, respectively). However, you always are able to rebuild without shared memory support or a different shared memory model just by manually redefining environment variables and by rerunning the makefile. There is no need to rerun the toolkit again. Also not that all arguments within the `shared-memory` environment are read at startup time. If you change the core count, there's no need to recompile. The same statements holds for the other parameters.

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

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

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

Table 5.1.: Parameter choices for the multicore configuration. All values are read at program startup. There is no need to rerun the toolkit if you change any of them.

Parameter	**Options**	**Description**

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

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

Autotuning and tailoring

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

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

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

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

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

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

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

Hybrid parallelisation

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

This subsection yet has to be written.

6. Optimisation

6.1. High-level tuning

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

```
optimisation
  fuse-algorithmic-steps = off
end optimisation
```

at the end of your specification file.

Step fusion. Each time step of an ExaHyPE solver consists of four phases: local ADER-DG predictor, projection of the prediction onto the cell phases, solve of the Riemann problem, 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`.

6.2. Optimised Kernels

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

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

```
make generator
```

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

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

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

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

```
cat /proc/cpuinfo
```

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

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

7. Postprocessing, plotting & logs

7.1. Dumps and visualisation of the whole setup

The table below gives a brief overview over available plotter types. They are distinguished by their identifier in the specification file. While the number of plotters and their type have to be fixed at compile time—if you want to introduce a new plotter, you have to rerun the `ExaHyPE` toolkit and rebuild your code—all other parameters are read at runtime.

Identifier	Description
vtk-binary	Write a VTK binary file.
vtk-text	Write a VTK text file.

7.2. Logging

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

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

debug              tarch    -1                      black
debug              peano    -1                      black

info               tarch    -1                      black
info               peano    -1                      black

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

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

A. The ExaHyPE toolbox

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

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

A.1. Rebuild the toolbox

To build the toolbox, we rely on ant.

```
ant target
```

where `target` is from

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

B. Frequently asked developer questions & problems

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

- **To be continued ...**

C. Frequently asked user & problems

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

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

  Only global
  rently. You c
  configure yet.

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

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

  We have to m
  this rou
  speak in te
  of mesh si
  and we have
  make it dom
  dependent c
  we have AMR

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

- **To be continued ...**

D. ADER-DG solvers with user-defined kernels

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

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

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

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