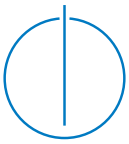
DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Simulation of the shallow water equations in the ExaHyPE-Engine with ADER-DG methods and dynamic adaptive mesh refinement

Stefan Haas



DEPARTMENT OF INFORMATICS

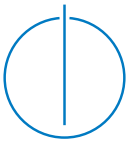
TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Simulation of the shallow water equations in the ExaHyPE-Engine with ADER-DG methods and dynamic adaptive mesh refinement

Simulation der Flachwassergleichungen in der ExaHyPE-Engine mit ADER-DG Methoden und dynamisch adaptiver Gitterverfeinerung

Author:	Stefan Haas
Supervisor:	Univ.-Prof. Dr. Michael Bader
Advisor:	M.Sc. Leonhard Rannabauer
Submission Date:	16.08.2018



I confirm that this bachelor's thesis in informatics is my own work and I have documented all sources and material used.

Munich, 16.08.2018 Stefan Haas

Abstract

ExaHyPE, an Exascale Hyperpolic PDE Engine, offers a game engine like solution to simulate hyperbolic partial differential equations, while tackling the challenges of future exascale supercomputers. In this thesis it is shown how to implement the shallow water equations in ExaHyPE as well as an additional Riemann solver to enable the wetting and drying of cells. It shows how much ExaHyPEs performance increases when using shared memory and distributed memory parallelisation and at which circumstances the best performance is achieved. The implementation of the solvers, especially the implementation of the additional Riemann solver, is then tested with two benchmarks. The results show that the implemented Riemann solver generates rather accurate results, but in the case of demanding problems it comes to its limits.

Contents

1 Introduction

Many physical processes can be described by partial differential equations (PDEs). They are used for simulations of, for instance, earthquakes and tsunamis. These simulations need a lot of computational resources, which is why they are run on large computer systems with a high number of cpus, i.e. supercomputers. The computing capability of these supercomputers have been increasing exponentially over the last years [13]. The fastest supercomputer as of the Top500 list of June 2018 achieved over 122 petaflops (10^{15} operations per second) as maximal LINPACK performance [15]. The next large step is a supercomputer, which can achieve one exaflop (10^{18} operations per second). This yields a lot of challenges for software to use this computing power efficiently. Some of these challenges are system power consumption, massive parallelism and less memory per core than in current systems [13]. ExaHyPE, an Exascale Hyperpolic PDE Engine, tackles these challenges by providing a game engine like solution to solve PDEs written in first order form. It uses explicit ADER-DG and finite volume schemes [14]. ADER-DG schemes make it possible to increase the accuracy of the simulation while staying within the same power budget [2]. To generate, refine and traverse grids, it uses the Peano space-filling curve, which has low memory footprint capabilities for the generated mesh. [16].

In this thesis I used the ExaHyPE engine and the shallow water equations (SWE), which are hyperbolic non-linear PDEs, to simulate waves. To increase the accuracy of the simulation I used the ADER-DG solver of ExaHyPE as well as the dynamic adaptive mesh refinement feature. Since the integrated solver of ExaHyPE does not support the wetting and drying of cells, I implemented an additional Riemann solver within a finite volume solver of ExaHyPE. ExaHyPE is thought to run on highly parallel systems, so I tested its performance improvements when using shared memory and distributed memory parallelisation.

The second chapter will briefly explain the SWE as well as some concepts of ExaHyPE. In the third chapter I will describe how I implemented the shallow water equations in ExaHyPE, especially how the Riemann solver for the wetting and drying of cells looks like, and how I initially tested the solvers. The performance tests are shown in

chapter 4, and chapter 5 is dedicated to benchmarks of the correctness of a limited solver, which uses the ADER-DG solver from ExaHyPE in assistance of a finite volume solver with the Riemann solver I implemented. The last chapter concludes this thesis and speaks about further possibilities what can be done with the resulting solver.

2 Theoretical Background

To simulate hyperbolic PDEs with initial conditions, like the SWE that are described in the next section, we need a numerical solver. ExaHyPE offers the spatial discretization and the numerical solvers in form of finite volume and ADER-DG solvers to do so. In this chapter I will describe the SWE and how ExaHyPE works.

2.1 Shallow Water Equations

As LeVeque [8] writes, the shallow water equations are non-linear PDEs, which he derives when considering a fluid in a channel of unit width and assuming that the vertical velocity is negligible and the horizontal velocity is roughly constant throughout any cross section of the channel. In order for these equations to be correct, the waves have to be small relative to the height of the water. The one-dimensional shallow water equations he derives are

$$\begin{bmatrix} h \\ hu \end{bmatrix}_t + \begin{bmatrix} hu \\ hu^2 + \frac{1}{2}gh^2 \end{bmatrix}_x = 0 \tag{2.1}$$

where h is the height of a unit of water and u the already mentioned horizontal velocity. The first term $[h, hu]_t^T$ describes the height and momentum of the unit at time t and the second term describes the flow in x-direction. The corresponding eigenvalues for the Jacobian-matrix of the flux are

$$\lambda_1 = u - \sqrt{gh},\ \lambda_2 = u + \sqrt{gh} \tag{2.2}$$

In two dimensions, like it is the case in ExaHyPE, LeVeque defines the SWE as follows

$$\begin{bmatrix} h \\ hu \\ hv \end{bmatrix}_t + \begin{bmatrix} hu \\ hu^2 + \frac{1}{2}gh^2 \\ huv \end{bmatrix}_x + \begin{bmatrix} hv \\ huv \\ hv^2 + \frac{1}{2}gh^2 \end{bmatrix}_y = 0 \tag{2.3}$$

where hu and hv are the momentum of the wave in the respective direction. And the corresponding eigenvalues are

$$\lambda_{x1} = u - \sqrt{gh},\ \lambda_{x2} = u,\ \lambda_{x3} = u + \sqrt{gh} \tag{2.4}$$

$$\lambda_{y1} = v - \sqrt{gh},\ \lambda_{y2} = v,\ \lambda_{y3} = v + \sqrt{gh} \tag{2.5}$$

Finally we have to add a non-conservative product to be able to use bathymetry as mentioned in [10] and get

$$\begin{bmatrix} h \\ hu \\ hv \end{bmatrix}_t + \begin{bmatrix} hu \\ hu^2 + \frac{1}{2}gh^2 \\ huv \end{bmatrix}_x + \begin{bmatrix} hv \\ huv \\ hv^2 + \frac{1}{2}gh^2 \end{bmatrix}_y + \begin{bmatrix} 0 \\ ghb_x \\ ghb_y \end{bmatrix} = 0 \tag{2.6}$$

where b_x and b_y are the derivations of b in the respective direction.

2.2 ExaHyPE

ExaHyPE offers an engine for solving hyperbolic PDEs, like our SWE, and plotting the solutions. To do so a configuration file has to be written and the toolkit of the engine will generate the necessary code and code framework, so that the PDEs can be easily implemented. ExaHyPE solves equations of the form of

$$\underbrace{P}_{\text{materialmatrix}} \frac{\partial}{\partial t} Q + \nabla \cdot \underbrace{F}_{\text{fluxes}}(Q) + \underbrace{\sum_{i=1}^{d} B_i(Q) \frac{\partial Q}{\partial x_i}}_{\text{ncp}} = \underbrace{S(Q)}_{\text{sources}} + \underbrace{\sum \delta}_{\text{pointsources}} \tag{2.7}$$

like it is mentioned in [14]. For our SWE we need $\frac{\partial}{\partial t} Q$, the fluxes and for the bathymetry we also need the non-conservative product (ncp). From formula 2.6 we get

$$\frac{\partial}{\partial t} Q = \begin{bmatrix} h \\ hu \\ hv \end{bmatrix}_t,\ F(Q) = \begin{bmatrix} hu \\ hu^2 + \frac{1}{2}gh^2 \\ huv \end{bmatrix}_x + \begin{bmatrix} hv \\ huv \\ hv^2 + \frac{1}{2}gh^2 \end{bmatrix}_y,\ B(Q) = \begin{bmatrix} 0 \\ ghb_x \\ ghb_y \end{bmatrix} \tag{2.8}$$

To calculate the solutions for such PDEs we need a numerical solver. ExaHyPE offers two different kinds of solvers, finite volume solvers and ADER-DG solvers. In chapter 3 I will discuss the solvers I used within ExaHyPE and one solver I extended by a special

Riemann solver to enable wetting and drying. It is also possible to use a limited solver in ExaHyPE, which uses an ADER-DG solver and for cells, that are marked as troubled, a finite volume solver. I used this feature to be able to combine the finite volume solver I implemented myself with the ADER-DG solver from ExaHyPE, so that the advantages of the ADER-DG solver can be used with wetting and drying.

From the SWE we can see that the state vector Q is $(h, hu, hv)^T$, but to be able to model sea floor topographies we need to consider bathymetry in our state vector. In ExaHyPE there are two possibilities to add the bathymetry, it can be added to either the variables or the parameters. For the implementation that basically makes no difference, except that ExaHyPE ensures that the parameters do not change. Since implementing it as parameter did not work at the beginning of this theses, I added it as variable. In further references Q is the vector $(h, hu, hv, b)^T$ and the bathymetry is also part of the PDE and considered as constant in time.

2.2.1 ADER-DG solver

ExaHyPE uses the method for the ADER-DG solver which is described in [4]. The numerical solution u_p of the vector of conserved quantities is represented by piecewise polynomials of degree N, which are spanned by orthogonal basis functions Φ_l, inside an element $T^{(m)}$ as a sum of degrees of freedom (dof) $\hat{u}_p^{(m)}$ and the space-only dependent basis functions of degree N as follows

$$u_p^{(m)}(\vec{\xi}, t^n) = \hat{u}_{pl}^{(m)}(t^n)\Phi_l(\vec{\xi}) \tag{2.9}$$

where $\vec{\xi} = (\xi, \eta, \zeta)$ are the spatial coordinates in a reference coordinate system. Then a $P_N P_M$ reconstruction operator is applied to generate piecewise polynomials of degree $M \geq N$, which are spanned by hierarchical orthogonal basis functions Ψ_l. After that they use a new local continuous space-time Galerkin method to evolve the polynomial data of degree M in time locally inside each element. To derive a fully discrete form of the $P_N P_M$ schemes they multiply the conservation law with a test function Φ_k and integrate over the space-time element $T^{(m)} \times [t^n; t^n + \Delta t]$

$$\int_{t^n}^{t^n+\Delta t}\int_{T^{(m)}} \Phi_k \frac{\partial}{\partial t} u_p dV dt + \int_{t^n}^{t^n+\Delta t}\int_{\partial T^{(m)}} \Phi_k \vec{F}_p \cdot \vec{n} dS dt - \int_{t^n}^{t^n+\Delta t}\int_{T^{(m)}} \frac{\partial \Phi_k}{\partial \vec{x}} \vec{F}_p dV dt = \int_{t^n}^{t^n+\Delta t}\int_{T^{(m)}} \Phi_k S_p dV dt \tag{2.10}$$

where they propose to use the solution of the local space-time Galerkin scheme with polynomial degree M to compute the fluxes and source terms. The last step is to update the degrees of freedom $\hat{u}_{lp}$ from time level n to time level $n + 1$.

2.2.2 Limiting

As I mentioned ExaHyPE offers a limited solver, where the accuracy of the ADER-DG solver is combined with the robustness of the finite volume solver for troubled cells. As described in [5] the limiter works by running the ADER-DG solver at time t^n for one time step to produce a so-called candidate solution at time t^{n+1}. The discrete representation of the solution within a general simplex element T_i, i.e. a cell, is denoted by $u_h(x, t^n)$ at the beginning of a time step and the candidate solution at t^{n+1} by $u_h^*(x, t^{n+1})$. Then troubled cells are detected a posteriori.

As mentioned in [14] ExaHyPE uses two criteria to detect troubled cells. The first one is the discrete maximum principle (DMP) which is defined in [5] by

$$\min_{y \in \mathcal{V}_i} \left(u_h \left(y, t^n \right) \right) - \delta \leq u_h^* \left(x, t^{n+1} \right) \leq \max_{y \in \mathcal{V}_i} \left(u_h \left(y, t^n \right) \right) + \delta \qquad \forall x \in T_i \tag{2.11}$$

The set $\mathcal{V}_i$ contains the current cell T_i and the cells that share at least a common node with T_i. Equation 2.11 shows that the candidate solution $u_h^* \left(x, t^{n+1} \right)$ must remain between the minimum and maximum values of the cell and its neighbours at the previous time step. δ is used as a parameter to relax the discrete maximum principle and is in [5] defined by

$$\delta = \max \left(\epsilon_0, \epsilon \left(\max_{y \in \mathcal{V}_i} \left(u_h \left(y, t^n \right) \right) - \min_{y \in \mathcal{V}_i} \left(u_h \left(y, t^n \right) \right) \right) \right) \tag{2.12}$$

where usually $\epsilon = 10^{-3}$ and $\epsilon_0 = 10^{-4}$. In ExaHyPE these two parameters can be set by the dmp-difference-scaling (ϵ) and the dmp-relaxation-parameter (ϵ_0) options.

right assignment

The second criteria is the Physical Admissibility Detection criteria (PAD), which checks if $u_h^* \left(x, t^{n+1} \right)$ verifies some physical admissibility constraints for the cell T_i. This constraint can be set within a limited solver by defining the, from ExaHyPE generated, method *isPhysicallyAdmissible* in the ADER-DG solver cpp-file.

If a cell is marked as troubled by one of these criteria the candidate solution of these cells is disregarded and recomputed with the finite volume solver. Therefore the DG polynomial u_h of order N of the cell is divided into $N_s = 2N + 1$ sub-cells, on which the finite volume solver is used on. The solution of the finite volume solver at time t^{n+1} is then gathered back into a valid cell-centered DG polynomial of degree N.

2.2.3 Adaptive Mesh Refinement

In most cases there are areas within the computational domain, that are more relevant for the correctness of the solution than other areas, e.g. in the scenario of section 5.1 the area where the run up takes place is the most important like I explained in 5.1.3. To reduce the computational effort and still be able to process such areas with a higher resolution, ExaHyPE offers adaptive mesh refinement (AMR). To use this feature the maximum-mesh-depth option in the configuration file has to be set greater than zero. ExaHyPE then creates a method, where a refinement criterion can be specified. This can be a static criterion to refine cells with special coordinates or a dynamic criterion to refine or coarse cells with special conditions, e.g. if they have a certain height or a certain gradient. In section 3.3.2 I used the water height to refine the mesh. Another possibility to use adaptive mesh refinement is to use a limited solver. There every cell which is marked as troubled will be refined to the finest mesh level.

Like it is described in [11] ExaHyPE uses a stack&stream approach to avoid random memory access. For the parallelisation of this approach it uses Peano Space-Filling Curves, which provides an algorithm for grid generation, refinement and traversal. [16] It's AMR functionality works on a cell level. A cell may be marked for refinement or coarsening, either through the refinement criterion or the limited solver. Then the grid is traversed and refined or coarsened with the corresponding interpolation operations.

3 Implementation

In this chapter I will talk about the implementation of the SWE in ExaHyPE as well as the implementation of a Riemann solver which was not part of the ExaHyPE engine. First I will describe my solution for the boundary conditions of the computational domain, then I will give a short overview of the different kind of solvers I used for the tests in chapter 4 and chapter 5. After that I will show the mathematical formulation of the wetting and drying solver I implemented and what initial conditions I used to test the solvers initially.

3.1 Boundary Conditions

To define the behaviour of the simulation at the boundary of the domain, we have to specify the boundary conditions of our solver. There are a few possibilities to choose from. For the simulations of this thesis wall boundaries are sufficient. This means that the cells at the boundary are set in a way, that they act like a wall.

To implement the wall boundary conditions for the finite volume solvers, we just need to copy the values of the state of the cells at the boundary within the domain to the cells outside of the domain and set the momentum of the cells outside in opposition to the cells inside respective to the direction. This means we simulate cells that have the same water height and move with the same speed against each other. In the following code excerpt the direction is given by the variable *normalNonZero*, which is 0 for the x-direction and 1 for the y-direction. This means, with *1 + normalNonZero*, we either set hu or hv.

```
std::copy_n(stateIn, NumberOfVariables, stateOut);
stateOut[1+normalNonZero] = -stateOut[1+normalNonZero];
```

For the ADER-DG solver this is not sufficient. There we also have to set the fluxes of the boundary cells. Therefore we use the flux function of our solver and set the flux of the outer cells to the flux of the respective direction.

```
double _F[2][NumberOfVariables]={0.0};
```

```
double* F[2] = {_F[0], _F[1]};
flux(stateOut,F);
std::copy_n(F[normalNonZero], NumberOfVariables, fluxOut);
```

3.2 Solvers

As mentioned in 2.2 ExaHyPE offers finite volume solvers as well as ADER-DG solvers. For the first verification a Godunov solver was used.

In the next step, I implemented an non-linear ADER-DG solver with Gauss-Legendre integration points. This worked exactly the same as with the Godunov solver, except that I had to change the boundary conditions like I mentioned in section 3.1. Performance tests for this solver are presented in chapter 4.

Since these solvers both do not support wetting and drying cells, I implemented another Riemann solver in the Godunov type finite volume solver, which I describe in the next section. ExaHyPE allows to implement custom Riemann solvers by overloading the default implementation. This solver was used for the tests in chapter 5. With every further reference of wetting and drying solver or finite volume solver this solver is meant.

The last solver I implemented was a limited solver, which used the non-linear ADER-DG solver with Gauss-Legendre integration points and the finite volume solver for wetting and drying cells. With this solver the accuracy of the solution can be improved while we still have wetting and drying.

3.3 Wetting and Drying

Since with equation 2.6 and the solvers of the engine no wetting and drying is possible I used the solver from [3] to implement a new Riemann solver. Here I will describe the Riemann solver as it is implemented with the interaction of two cells. For the calculation, the initial state of the two interacting cells is needed, which looks like this

explain why solver is not working

$$Q_L = \begin{pmatrix} h_L \\ (hu)_L \\ (hv)_L \\ b_L \end{pmatrix}, Q_R = \begin{pmatrix} h_R \\ (hu)_R \\ (hv)_R \\ b_R \end{pmatrix} \tag{3.1}$$

To distinguish between the two directions, we need the vector $\vec{n}_d$ where the d stands for the respective direction, i.e. x- or y-direction.

$$\vec{n}_d = \begin{pmatrix} 0 \\ \delta_{dx} \\ \delta_{dy} \\ 0 \end{pmatrix} \tag{3.2}$$

where d is either x or y and $\delta_{i,j}$ is the Kronecker delta.

$$\delta_{i,j} = \begin{cases} 0 & \text{if } i \neq j \\ 1 & \text{if } i = j \end{cases} \tag{3.3}$$

To get the wetting and drying to work we need $u_{L_{n_d}}$ and $u_{R_{n_d}}$, which are also dependent on the direction, but also on the selected value for ϵ.

$$u_{L_{n_d}} = \begin{pmatrix} 0 \\ (hu)_L \\ (hv)_L \\ 0 \end{pmatrix} \cdot \vec{n}_d^T \cdot h_L \cdot \sqrt{2} / \sqrt{h_L^4 + (max(h_L, \epsilon))^4} \tag{3.4}$$

$$u_{R_{n_d}} = \begin{pmatrix} 0 \\ (hu)_R \\ (hv)_R \\ 0 \end{pmatrix} \cdot \vec{n}_d^T \cdot h_R \cdot \sqrt{2} / \sqrt{h_R^4 + (max(h_R, \epsilon))^4} \tag{3.5}$$

The fluxes for the left and the right cell in dependency of the direction are displayed in 3.6.

$$F_{L_d} = \begin{pmatrix} h_L \\ h_L \cdot u_{L_{nx}} \\ h_L \cdot u_{L_{ny}} \\ 0 \end{pmatrix} \cdot \left(\begin{pmatrix} 0 \\ u_{L_{nx}} \\ u_{L_{ny}} \\ 0 \end{pmatrix} \cdot \vec{n}_d^T \right), \; F_{R_d} = \begin{pmatrix} h_R \\ h_R \cdot u_{R_{nx}} \\ h_R \cdot u_{R_{ny}} \\ 0 \end{pmatrix} \cdot \left(\begin{pmatrix} 0 \\ u_{R_{nx}} \\ u_{R_{ny}} \\ 0 \end{pmatrix} \cdot \vec{n}_d^T \right) \tag{3.6}$$

Then our lambdas are calculated the same as in section 2.1 with respect to the left or right cell.

$$\lambda_{1_{L_d}} = u_{n_{L_d}} + \sqrt{g \cdot h_L}, \; \lambda_{2_{L_d}} = u_{n_{L_d}} - \sqrt{g \cdot h_L}, \; \lambda_{3_{L_d}} = u_{n_{L_d}} \tag{3.7}$$

$$\lambda_{1_{R_d}} = u_{n_{R_d}} + \sqrt{g \cdot h_R},\ \lambda_{2_{R_d}} = u_{n_{R_d}} - \sqrt{g \cdot h_R},\ \lambda_{3_{R_d}} = u_{n_{R_d}} \tag{3.8}$$

Now we need the the maximum of all lambdas for the right and the left cell respective to the direction.

$$s_{max_d} = max(\lambda_{i_{j_d}}) : i \in [1,3] : j \in \{L,R\} \tag{3.9}$$

At last δ is calculated as follows

$$\delta = max(h_R + b_R - max(b_L, b_R), 0.0) - max(h_L + b_L - max(b_L, b_R), 0.0) \tag{3.10}$$

Now we can calculate the flows for both cells. We have the fluxes from equation 3.6, a further term for the fluxes and the non-conservative product.

$$f_{L_d} = 0.5 \cdot \left(\left(F_{L_d} + F_{R_d}\right) - s_{max_d} \cdot \begin{pmatrix} \delta \\ (hu)_R - (hu)_L \\ (hv)_R - (hv)_L \\ 0 \end{pmatrix} + \begin{pmatrix} 0 \\ g \cdot 0.5 \cdot (h_L + h_R) \cdot \delta \\ g \cdot 0.5 \cdot (h_L + h_R) \cdot \delta \\ 0 \end{pmatrix} \cdot \vec{n}_d \right) \tag{3.11}$$

$$f_{R_d} = 0.5 \cdot \left(\left(F_{L_d} + F_{R_d}\right) - s_{max_d} \cdot \begin{pmatrix} \delta \\ (hu)_R - (hu)_L \\ (hv)_R - (hv)_L \\ 0 \end{pmatrix} - \begin{pmatrix} 0 \\ g \cdot 0.5 \cdot (h_L + h_R) \cdot \delta \\ g \cdot 0.5 \cdot (h_L + h_R) \cdot \delta \\ 0 \end{pmatrix} \cdot \vec{n}_d \right) \tag{3.12}$$

Additionally I had to set u or v to zero, if the height of the cell was smaller than ϵ. I realised this be configuring the *adjustSolution* method in ExaHyPE as in equation 3.13. This method is run after every time step for every cell and is also used to set the initial conditions.

$$hu = \begin{cases} hu & \text{if } h \geq \epsilon \\ 0 & \text{if } h < \epsilon \end{cases} \qquad hv = \begin{cases} hv & \text{if } h \geq \epsilon \\ 0 & \text{if } h < \epsilon \end{cases} \tag{3.13}$$

3.3.1 Example

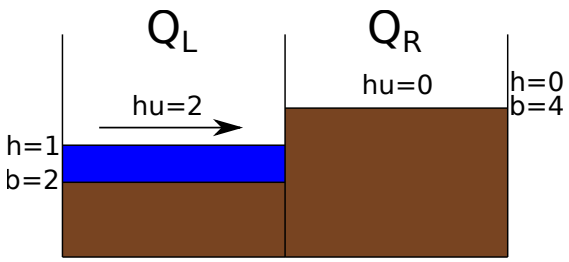


Figure 3.1: Representation of the initial state of the left and the right cell of the example for the Riemann solver

To demonstrate the result of the solver I used the scenario shown in Fig. 3.1. The initial states of the left and the right cell are

$$Q_L = \begin{pmatrix} 1 \\ 2 \\ 0 \\ 2 \end{pmatrix}, \qquad Q_R = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 4 \end{pmatrix} \tag{3.14}$$

Note that the selected ϵ does not influence this example as long as $\epsilon \leq 1$. In the scenario the water of the left cell moves against the higher bathymetry of the right cell and we would expect the water to get reflected from the right cell. In this case we calculate the movement in x-direction. For f_{L_x} as well as f_{R_x} we have to calculate F_{R_x} and F_{R_x}.

$$F_{L_x} = \begin{pmatrix} h_L \cdot u_{L_{n_x}} \\ h_L \cdot u_{L_{n_x}} \cdot u_{L_{n_x}} \\ h_L \cdot u_{L_{n_y}} \cdot u_{L_{n_x}} \\ 0 \end{pmatrix} = \begin{pmatrix} 2 \\ 4 \\ 0 \\ 0 \end{pmatrix}, F_{R_x} = \begin{pmatrix} h_R \cdot u_{R_{n_x}} \\ h_R \cdot u_{R_{n_x}} \cdot u_{R_{n_x}} \\ h_R \cdot u_{R_{n_y}} \cdot u_{R_{n_x}} \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} \tag{3.15}$$

with

$$u_{L_{n_x}} = 2, u_{L_{n_y}} = 0, u_{R_{n_x}} = 0, u_{R_{n_y}} = 0 \tag{3.16}$$

The eigenvalues are calculated with equation 3.7 and 3.8.

$$\lambda_{1_{L_x}} = 2 + \sqrt{9.81},\ \lambda_{2_{L_x}} = 2 - \sqrt{9.81},\ \lambda_{3_{L_x}} = 2 \tag{3.17}$$

$$\lambda_{1_{R_x}} = 0,\ \lambda_{2_{R_x}} = 0,\ \lambda_{3_{R_x}} = 0 \tag{3.18}$$

This results in

$$s_{max_x} = 2 + \sqrt{9.81} \tag{3.19}$$

It can be seen that $\delta = 0$. With this we can finally calculate the resulting flows for both cells.

$$f_{L_x} = 0.5 \cdot \left(\begin{pmatrix} 2 \\ 4 \\ 0 \\ 0 \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} - (2 + \sqrt{9.81}) \cdot \begin{pmatrix} 0 \\ -2 \\ 0 \\ 0 \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} \right) = \begin{pmatrix} 1 \\ 4 + \sqrt{9.81} \\ 0 \\ 0 \end{pmatrix} \tag{3.20}$$

Since the only difference between f_{L_x} and f_{R_x} is the addition or respectively the subtraction of the ncp, which is 0 in this case, f_{R_x} looks exactly like f_{L_x}.

To see how these flows influence the state of the cells the Godunov solver has to be considered. There the flows get multiplied by $\frac{timestep}{cellsize}$ and f_{L_x} gets subtracted from the left cell whereas f_{R_x} gets added to the right cells. So for the left cell we have a reflection, like we assumed, and the right cell gets wet. correct?

3.3.2 Initial Tests

To test the implementation of the solvers as well as AMR I used some basic initial conditions.

As basic test I used simple shock-shock and rare-rare problems. In these tests the computational domain has a width of 10 by 10 and I defined the initial state as follows

$$Q_S = \begin{pmatrix} 4 \\ hu(x,0)_S \\ 0 \\ 0 \end{pmatrix} \qquad Q_R = \begin{pmatrix} 4 \\ hu(x,0)_R \\ 0 \\ 0 \end{pmatrix} \tag{3.21}$$

where Q_S is the initial state of the shock-shock problem, Q_R is the initial state of the rare-rare problem and hu is dependent on the x-coordinate.

$$hu(x,0)_S = \begin{cases} 2 & \text{if } x < 5 \\ -2 & \text{if } x \geq 5 \end{cases} \qquad hu(x,0)_R = \begin{cases} -2 & \text{if } x < 5 \\ 2 & \text{if } x \geq 5 \end{cases} \tag{3.22}$$

As it can be seen in equation 3.21 we actually have a 1-dimensional problem. But since in ExaHyPE we work on a 2-dimensional domain, we have to set a width for the y-direction. Notice that the width in y-direction can be set to any value without changing the solution, since there is no water flow in y-direction and the cells with the same x-coordinate have the same state at each time step. The only difference is, that a smaller width of the y-direction would lead to smaller computational domain and therefore to less computational effort.

The next test uses a Gauss function as initial water height in a domain of 10 by 10, where the initial state Q is defined as follows

$$Q = \begin{pmatrix} e^{-(x-5)^2} + 1 \\ 0 \\ 0 \\ 0 \end{pmatrix} \tag{3.23}$$

Plus 1 is added to the Gauss function, since the ADER-DG solver does not support wetting and drying and the water height should not be zero at any point of the function. In this scenario the width of the y-dimension can be set to any value because of the same reasons as in the rare-rare and shock-shock tests. This initial condition was used for the performance tests in chapter 4.

I used this initial condition to test dynamic AMR. Therefore I used the ADER-DG solver and set the refinement criterion. As I mentioned in section 2.2.3 there are different ways to set the criterion, in my test I set it accordingly to the water height. Therefore I calculated the maximum height h_{max} of the dof in the cell and set the refinement criterion accordingly as follows

$$Refinement\ level = \begin{cases} CML + 2, & \text{if } 1.5 < h_{max} \\ CML + 1, & \text{if } 1.2 < h_{max} < 1.5 \\ CML, & \text{if } h_{max} < 1.2 \end{cases} \tag{3.24}$$

CML stands for coarsest mesh level and is the level which is normally used without AMR. In Fig. 3.2 the result of this test can be seen. It shows a part of the computational domain, where the domain is not fully shown in y direction, since as I mentioned the state of the cells does not change in y-direction. It shows the domain at $t = 0$ and with the refinement criterion as in equation 3.24. The water height is the highest in the middle of the domain and the water height gets smaller in both directions. The

different refinement levels can clearly be seen and are moving with the height of the water as it is simulated over time.

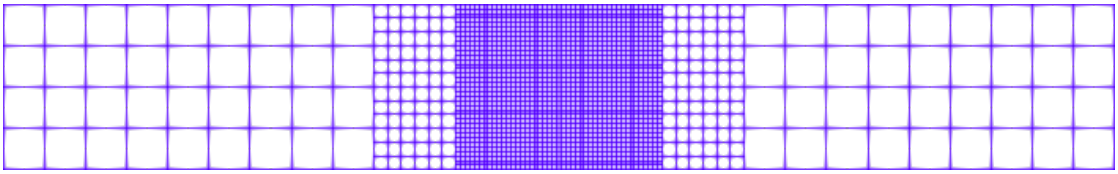


Figure 3.2: Part of the Gauss scenario at $t = 0$ with AMR and the refinement criterion as in equation 3.24. The curve is at $height = 2$ in the middle and gets smaller to the end of the domain in x-direction, therefore the refinement gets smaller accordingly.

4 Performance

Since the ExaHyPE engine is designed to run on future supercomputers it is necessary that its performance scales with the resources available. To use more than one core ExaHyPE offers to use shared memory parallelisation, distributed memory parallelisation or both together. As shared memory parallelisation options ExaHyPE supports TBB and OpenMP, from which I used just TBB for my performance tests, because the development of the TBB variant is typically one step ahead of the OpenMP support [14]. For the distributed memory parallelisation ExaHyPE uses MPI.

4.1 Test Setup

4.1.1 Hardware

Tests were run on the MAC-Cluster of the Leibniz Supercomputing Centre. This cluster is designated for research in development of parallel software [9]. To test the performance of my SWE Application I used the "snb" nodes of this cluster, which feature a dual socket with two Intel SandyBridge-EP Xeon E5-2670 and 128GB of RAM. Each of these Xeon E5-2670 have 8 cores with a base frequency of 2.60 GHz and a max turbo Frequency of 3.30 GHz and 16 Threads.

4.1.2 Configuration

Tests were executed with the ADER-DG solver I mentioned in section 3.2. As initial condition I used the Gauss-Scenario, as described in section 3.3.2, in both cases, i.e. with TBB and MPI. In the case of TBB I first set the maximum-mesh-size to 0.15, which leads to 6561 cells, and, to test a bigger resolution, for the second test to 0.05, which leads to 59049 cells. In the case of MPI I just ran tests with a maximum-mesh-size of 0.15, i.e. 6561 cells.

4.2 TBB

```
shared-memory
  identifier = dummy
  configure = {background-tasks:1}
  cores = 1
  properties-file = sharedmemory.properties
end shared-memory
```

The code example above shows the configuration I used for TBB performance tests. Identifier is set to dummy, which uses default values. The background-tasks option does not influence the result since in the optimization options spawn-predictor-as-background-thread is turned off. The cores option was always set according to the respective amount of cores for a test run.

The option cores defines the number of used TBB threads. So in the case of this cluster setting the option to 32 corresponds to 16 physically cores with 2 hyper-threads each.

As mentioned above, I used two different resolutions for the tests. Additionally I used different orders for the test with 6561 cells to increase the computational effort of the program. Since the runtime for using only one core would have exceeded the clusters time limit, I set the base of our measurements at 4 cores and increased the amount of cores used to 8, 16 and 32.

4.2.1 Speedup

To be able to calculate the speedup I used the formula [6]

$$S_p = \frac{T_1}{T_p} \tag{4.1}$$

where p is the amount of processors. S_p is the speedup. T_1 is the time the program needs to finish on one processor and T_p is the time a program needs to finish on p processors. In this case one processor corresponds to one core.

To get T_1 I assumed that the speedup for 4 cores was linear and calculated T_1 as follows: $T_1 = \frac{T_4}{4}$.

Like it is shown in Fig. 4.1, more cells lead to a higher speedup. The same is true for a higher order, where the speedup is increasing even more. The first doubling of the cores from 4 to 8 has a higher speedup than the doubling from 8 to 16 cores, e.g. the speedup with 6561 cells and an order of 7 has the factor 1,9 when doubling from 4 to 8

cores and the factor 1,8 when doubling from 8 to 16 cores. The doubling from 16 to 32 cores has almost no effect or even decreases the performance in the case of 59049 cells, which shows that hyper-threading has no influence.

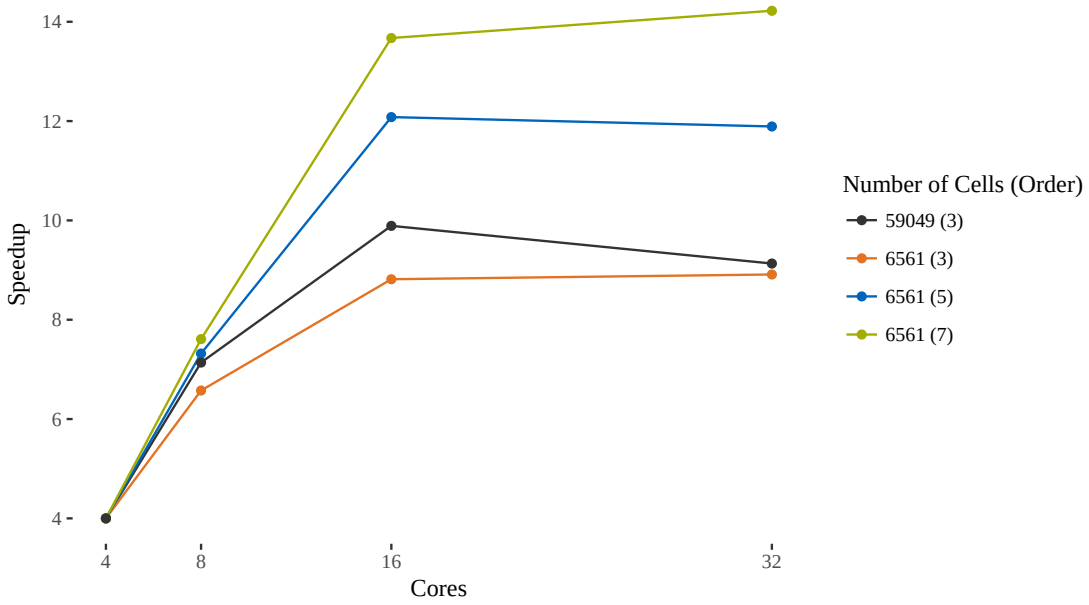


Figure 4.1: TBB shared memory speedup plot with the Gauss scenario from 3.3.2

4.2.2 Degrees of Freedom

The next aspect I considered was the dof per second. The degrees of freedom are calculated as follows

$$NumberOfCells \cdot (Order + 1)^2 \tag{4.2}$$

As it can be seen in Fig. 4.2 the most dof per second are achieved with 6561 cells and order 3. I assumed that the dof per second would increase as we increase the amount of cells or the order, since more degrees of freedom means more possible parallelisation, but as we increase the number of cells or the order, less dof per second are calculated.

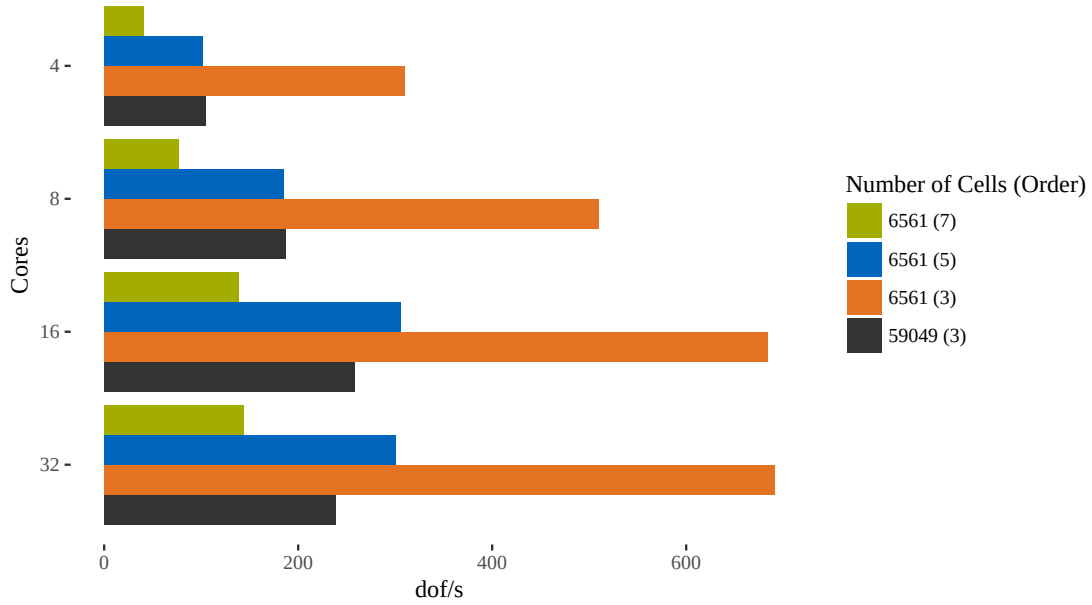


Figure 4.2: TBB shared memory dof plot with the Gauss scenario from 3.3.2

4.3 MPI

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

The code example above shows the configuration I used for the MPI performance tests. The identifier is set to static_load_balancing, since this is currently the only supported option. The option configure has the two arguments greedy-naive and FCFS, where greedy-naive guides the load balancing and is the recommended option according to [14] and FCFS controls the decision which ranks to use next. The buffer-size option specifies the amount of messages Peano shall internally bundle into one MPI message and the timeout parameter specifies the time after which a node triggers a time out if no message has arrived.

As I mentioned before, I tested the MPI performance with only 6561 cells. But as with TBB I used different orders to increase the computation effort.

To test the MPI performance I set the -n option of the mpirun command, which sets the number of processes, to 1, 2, 4, 8, 10, 16, 24, 32, 48, 64, and 80. Since one node has 16 physically cores, I ran the tests with the -n option set to 16 and less on one node. In the other cases I evenly distributed the number of processes on the minimum of nodes, so that on no node more than 16 processes were started and thus did not test any hyper-threading.

4.3.1 Speedup

For the speedup tests the same formula (4.1) was used as in the TBB tests.

The results of this tests can be seen in Fig. 4.3. The speedup from 1 to 4 processes is almost zero, while the speedup from 4 to 10 processes is more than 4. After increasing to 10 processes this pattern repeats. Until 64 processes we almost get no speedup and from 64 to 80 the speedup is 4 in the case of order=3 or in the case of order=7 it is even more than 14. This shows that the size of the problem influences the speedup performance of the program. The reason that the highest speedup increases are by 10 and 80 processes is the partitioning of the grid and that one rank is reserved for load balancing and administration [14]. The grid is three-partitioned in each dimension, which means in our case that the grid is firstly partitioned in 3 by 3 and if we use more processes every subgrid is again partitioned in 3 by 3, which means the whole grid is partitioned in 9 by 9. That means the optimal speedup should lie by 10 processes $((3 \cdot 3) + 1)$ and 82 processes $((9 \cdot 9) + 1)$, which the tests proof.

4.3.2 Degrees of Freedom

To calculate the number of dof the same formula (4.2) was used as in the TBB tests.

In Fig. 4.4 we get the same result as with TBB. The higher the order and consequently the dof, the less dof per second are calculated. But as the amount of processes increases the ratio between the dof per second of a higher order and the dof per second of a smaller order decreases, which means that with enough processes the dof per second of the different orders could be aligned.

If the TBB results from Fig. 4.2 and the MPI results from Fig. 4.4 are compared, then it can be seen that with a smaller amount of cores, i.e. with one node, TBB is more efficient than MPI.

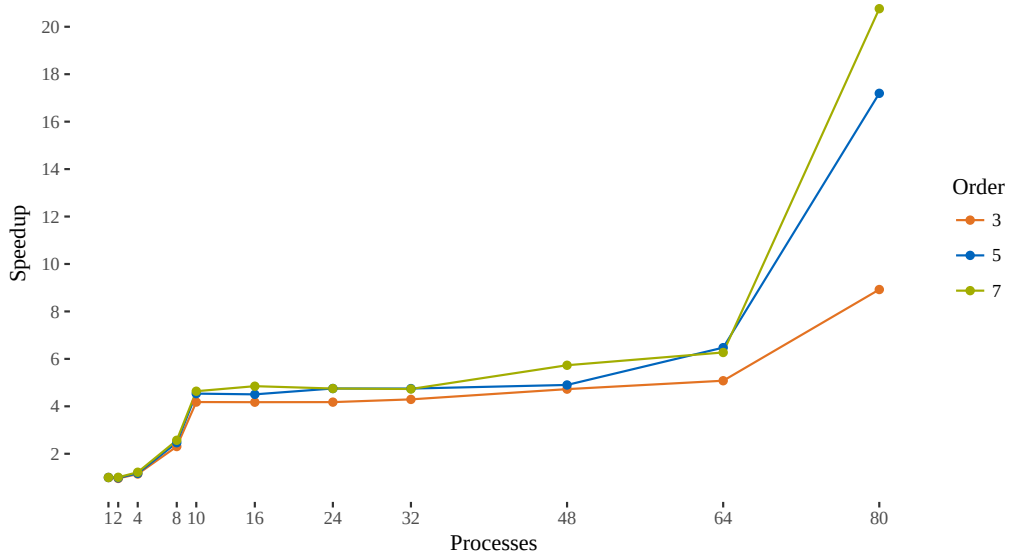


Figure 4.3: MPI distributed memory speedup plot with the Gauss scenario from 3.3.2

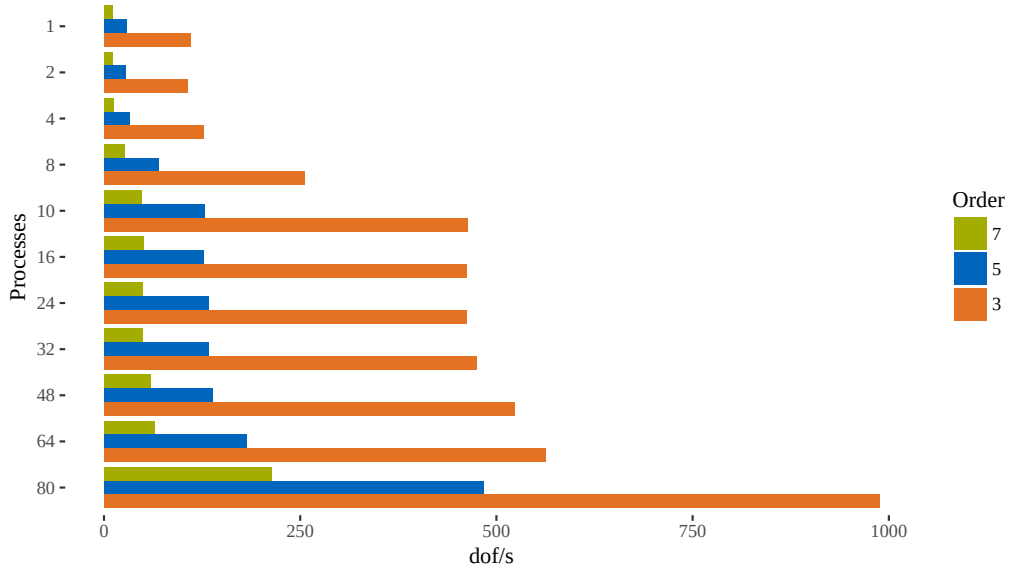


Figure 4.4: MPI distributed memory dof plot with the Gauss scenario from 3.3.2

5 Wetting and drying benchmarks

To test the physical correctness of the wetting and drying solver I mentioned in section 3.2. I ran two benchmarks. The first one tests the result of the wetting and drying solver with empirical data, gathered with a wave tank, and the second one tests it with a periodically moving water droplet. I used the finite volume solver for the initial tests and in the case of the first benchmark I used the limited solver to improve the results. In this chapter I will describe the structure of the computational domains for each benchmark, how I implemented and tested them and what the results were. For every tests g, i.e. the gravitation, was set to 9.81.

5.1 Solitary Wave on a Simple Beach

5.1.1 Benchmark Description

This benchmark uses the results of experiments, which where executed at the California Institute of Technology, and are described in [1]. In [7] González et al. used this benchmark to validate the software GeoClaw. The computational domain for the benchmark, which simulates the wave tank of the experiments, looks like in Fig. 5.1. The water has the overall height d and the initial wave is located at $x = x_0 + L$, which has the height H. The wave is moving towards the ramp, which displays a simple beach, where the wave is running up.

In the problem description the x-axis lies on the height d, therefore d actually stands for depth. Since in our solver the water cannot have a negative height, I moved the x-axis down to the bottom of the domain. The z-axis stays the same. The computational domain starts at $x = -10$ and ends at $x = 60$. At first I got these values for the size of the computational domain from [7]. While trying out different values for the tests of the benchmark, the initial wave started always within the domain and never got to the end of the domain while running up, so I kept them. Therefore in the settings of the configuration file the width for the x-axis is set to 70 and the offset to -10. Since we are working on a 2-dimensional computational domain, we have to set a width for the

y-axis as well. Because the wave only moves in x-direction we should set the width of the y-axis to a small value to decrease the overall computational domain. In my tests I set the width to 1 and the offset to 0.

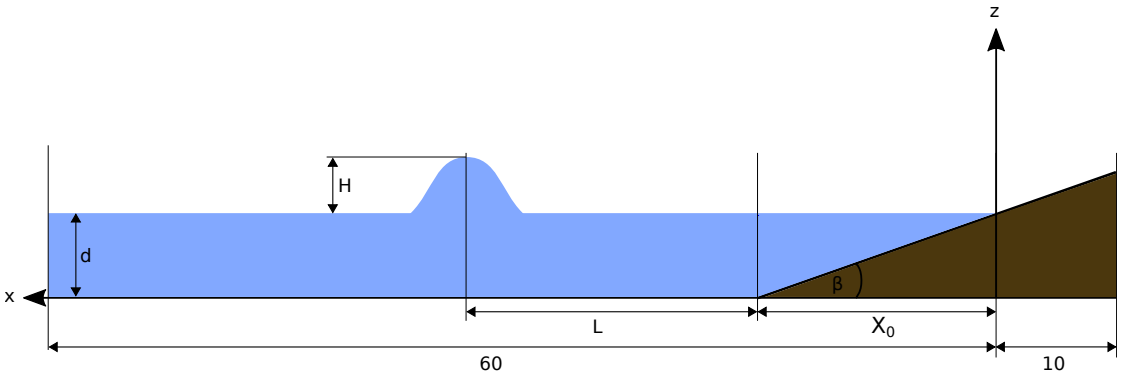


Figure 5.1: Sketch of computational domain

As in the problem description [1] is stated, the slope of the ramp is 1:19.85 which means we can calculate β as follows

$$cot\beta = 19.85 \iff \frac{1}{tan\beta} = 19.85 \iff \beta = atan\left(\frac{1}{19.85}\right) \tag{5.1}$$

γ is the same as in the problem description.

$$\gamma = \sqrt{\frac{3H}{4d}} \tag{5.2}$$

To be able to simulate the ramp, we have to calculate x_0. Since their is no standard cotanges function in C++ I used $\frac{cos(\beta)}{sin(\beta)}$.

$$x_0 = d \cdot cot(\beta) \iff x_0 = d \cdot \frac{cos(\beta)}{sin(\beta)} \tag{5.3}$$

L is calculated like in the problem description, except that I used $cosh^{-1}(a) = log\left(a + \sqrt{a^2 - 1}\right)$.

$$L = \frac{d \cdot log\left(\sqrt{20} + \sqrt{20 - 1}\right)}{\gamma} \tag{5.4}$$

For the formula of the initial condition $\eta(x,0)$ of the wave I used $sech(x) = \frac{1}{cosh(x)}$. Note that this would be the initial condition for our water height just if our x-axis would be at the same height as in the problem description. Since our x-Axis is at the bottom of the computational domain, we have to adjust the water height accordingly, which I show in equation 5.6 and 5.7.

$$\eta = H \cdot \left(\frac{1}{cosh\left(\frac{\gamma(x-(x_0+L))}{d}\right)} \right)^2 \tag{5.5}$$

With these equations we can set the initial condition within our solver. To do that we differentiate the three areas $x < 0$, $0 \leq x \leq x_0$ and $x_0 < x$. The height of the water is then calculated as follows

$$h(x,0) = \begin{cases} 0, & \text{if } x < 0 \\ x \cdot \frac{sin(\beta)}{cos(\beta)}, & \text{if } 0 \leq x \leq x_0 \\ \eta + d, & \text{if } x_0 < x \end{cases} \tag{5.6}$$

The bathymetry is set to imitate the ramp and has the value 0 at the rest of the domain.

$$b(x,0) = \begin{cases} -x \cdot \frac{sin(\beta)}{cos(\beta)} + d, & \text{if } x < 0 \\ d - h, & \text{if } 0 \leq x \leq x_0 \\ 0, & \text{if } x_0 < x \end{cases} \tag{5.7}$$

To set the momentum of the wave in x-direction, we calculate u with the velocity scale $U = \sqrt{gd}$ like in the problem description and multiply it with h.

$$hu(x,0) = -\eta \cdot \sqrt{\frac{g}{d}} \cdot h \tag{5.8}$$

At last I set the momentum of the wave in y-direction to 0, since the wave only moves in x-direction.

To get accurate results, the time, where we need to extract the current state of our simulation, needs to be calculated. Therefore we use the time scale $T = \sqrt{\frac{d}{g}}$ and can calculate the time t for our simulation by

$$t = t_b \cdot T \tag{5.9}$$

where t_b is the time of the lab data.

5.1.2 Finite Volume Solver Tests

At first I used the finite volume solver to test the wetting and drying solver. I set the maximum-mesh-size to 0.02, which leads to 616734 cells, and ϵ to 1e $-$ 7. To get numerical results that can be compared with the lab data, the only requirement is to set the ratio $\frac{H}{d}$ to the same value as in the respective lab test. Two different tests were made in the lab, one with $\frac{H}{d} = 0.0185$ and the second one with $\frac{H}{d} = 0.3$.

For the first test with $\frac{H}{d} = 0.0185$ I set d to 0.3, since the depth in the experiments was $\approx 30cm$, even though every other value would be suitable as well. The calculated times (t) for our simulation in respect to the time of the lab data (t_b) can be seen in table 5.1. Since the time steps of our simulation are dependent on the wave speed and are not continuous, I set the repeat value of the plotter in the configuration file to a very small value, so it would plot every time step of our simulation within the area of the calculated time. Then I used the time step which was the nearest at t. This time step can be seen in table 5.1 in the column t_s.

t_b	t	t_s
30	5.2462	5.24661
40	6.955	6.95501
50	8.7437	8.74371
60	10.4925	10.4925
70	12.2412	12.2414

Table 5.1: time values for $H/d = 0.0185$ with the FV solver

The results are shown in Fig. 5.2. Because the depth can be chosen freely, the lab data is given relative to d. So in Fig. 5.2 the graphs show the coordinates of the x-direction in relation to d on the x-axis and on the y-axis the wave height η in relation to d. As it can be seen our numeric solution is very close to the lab data points.

In the second test with $\frac{H}{d} = 0.3$ I oriented d again near to the actual depth of the experiments and set it to 0.15m. The time values can be seen in table 5.2 and the results in Fig. 5.3. For the times $t = 15$ and $t = 20$ the results do not match the lab data as well as the rest of our results do.

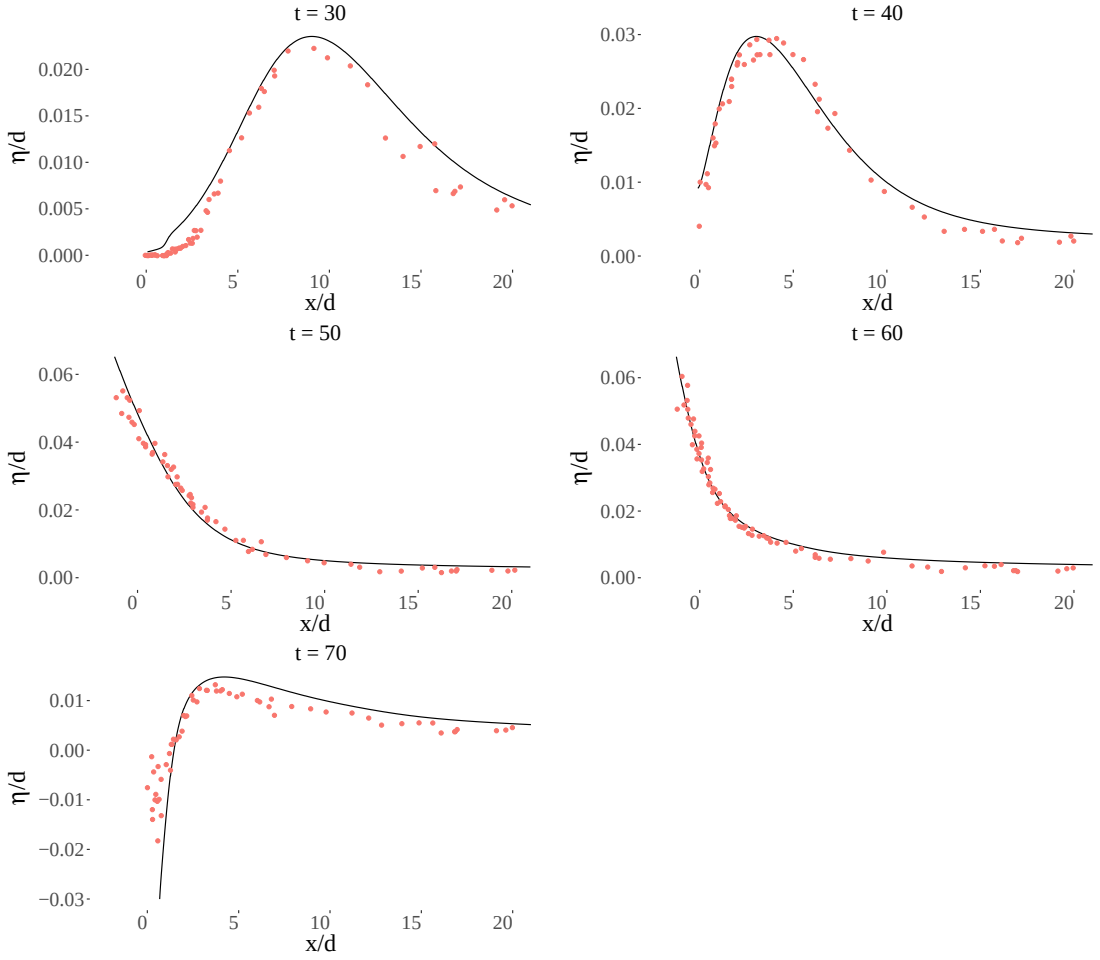


Figure 5.2: Comparison of numeric solution of the finite volume solver to lab data (red dots are the lab data) in the case of $\frac{H}{d} = 0.0185$

t_b	t	t_s
15	1.8548	1.85488
20	2.4731	2.47273
25	3.0914	3.09143
30	3.7096	3.70957

Table 5.2: time values for $H/d = 0.3$ with the FV solver

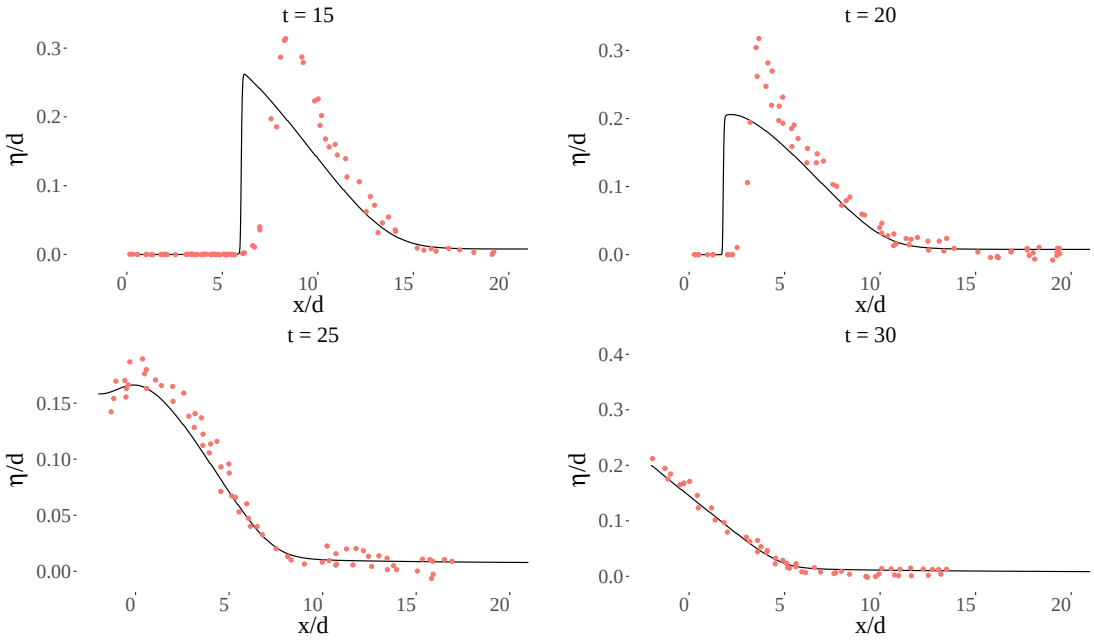


Figure 5.3: Comparison of numeric solution of the finite volume solver to lab data (red dots are the lab data) in the case of $\frac{H}{d} = 0.3$

5.1.3 Limited Solver Test with AMR

To improve the results I used the limited solver and ran the first test with $\frac{H}{d} = 0.0185$ again. To reduce the computational effort I also used AMR. Since in our test the run up is the point where it is most important to calculate correctly, there needs to be the refinement. In the limited solver every cell, which is troubled and therefore calculated

with the finite volume solver, has the maximum refinement level. The cells near the run up should be marked as troubled, since the water height is very small. To mark cells as troubled, I calculated the minimum and maximum height of the dof in the cells. With these two values I marked every cell as troubled which was smaller than $20 \cdot \epsilon$. With that criterion a big area of the ramp, where the water height is zero, because there is no water, is also marked as troubled and therefore also refined to the finest mesh level. This is a problem, because we need a lot of computational effort to calculate these areas, where nothing happens. For this reason I marked all cells with a minimum and maximum height of zero as not troubled, so that just the area around the run up was marked as troubled and therefore refined.

Since I set $\epsilon = 1\text{e} - 7$ for the test in Fig. 5.4, just a very small area around the run up is calculated with the finite volume solver and therefore refined. So we have to increase the refined area by setting a refinement criterion. For the test in Fig. 5.4 I set it as follows.

$$Refinement\ level = \begin{cases} CML + 3, & \text{if } h_{min} < 0.025 \\ CML + 2, & \text{if } 0.025 \leq h_{min} < 0.05 \\ CML + 1, & \text{if } 0.05 \leq h_{min} < 0.08 \\ CML, & \text{if } 0.08 \leq h_{min} \end{cases} \tag{5.10}$$

CML and h_{min} are described in section 3.3.2.

For the test I set the maximum-mesh-size to 0.5 and the maximum-mesh-depth to 3. This lead to 13388 cells, which is far less that the 616734 cells in our test without AMR, but the run up area has the same resolution. To get the closest time step of our simulation to the time of the lab data, I did the same as in the test with the finite volume solver.

The result for the run with $t = 30$ is displayed in Fig. 5.4. There the effect of AMR can be clearly seen as in right side of the graph the numerical solution has a stair-like structure and becomes smoother at the left side of the graph, where the water height gets smaller and the resolution is therefore higher. As the graph shows the numerical solution is close to the lab data points, although most of the domain has a lower resolution than before. The graph is just a small part of the whole domain, since it goes from $x = 0$ to approximately $x = 6$. This means we could increase the area where the refinement is at the maximum, which would mean a bigger part of the numerical solution of the graph would be smoother, and still have less overall cells than with the finite volume solver. Therefore we would need to adjust the refinement criterion in our

solver.

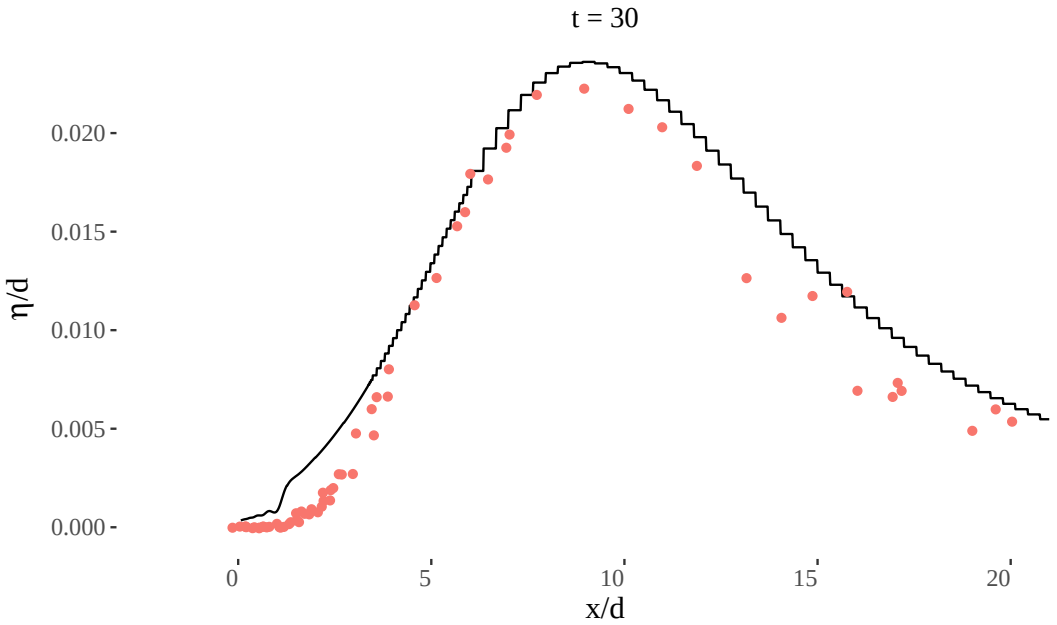


Figure 5.4: Comparison of numeric solution of the limited solver with AMR to lab data (red dots are the lab data) in the case of $\frac{H}{d} = 0.0185$

5.2 Oscillating Lake

5.2.1 Benchmark Description

In this benchmark a water droplet or lake is simulated within an elliptical paraboloid. The droplet is in its initial state at $t = 0$ central to the x-axis, but on right right side of the y-axis. In this initial state the droplet has a momentum parallel to the y-axis. The droplet will move periodically in the form of a circle through the paraboloid, so that after one rotation it will end in the same place as where it started. For our test I ran our simulation for approximately one rotation, since our time steps are not continuous and we have to get the closest time step to exactly one rotation, and evaluated that result with respect to the analytic solution. I tested this with different resolutions and values for ϵ.

For the implementation we use the formulas from [12]. With $\omega = \sqrt{0.2g}$ we get

$$h(x,y,t) = max\left(0, 0.05 \cdot \left(2 \cdot x \cdot cos\left(\omega t\right) + 2 \cdot y \cdot sin\left(\omega t\right)\right) + 0.075 - b\left(x,y\right)\right) \tag{5.11}$$

$$hu(x,y,t) = 0.5 \cdot \omega \cdot sin\left(\omega t\right) \cdot h \tag{5.12}$$

$$hv(x,y,t) = 0.5 \cdot \omega \cdot cos\left(\omega t\right) \cdot h \tag{5.13}$$

$$b(x,y) = 0.1 \cdot \left(x^2 + y^2\right) \tag{5.14}$$

To get the initial state we set $t = 0$.

5.2.2 Finite Volume Solver Test

To get the error of one cell ϵ_{cell} of our numeric solution I added one variable to the plotter and let the plotter write out the difference between the value of the current time step t and the analytic solution at exactly that time, like it is shown in equation 5.15, for every cell. Then I took the closest time step to the time of one rotation, which is at $\frac{2\pi}{\omega}$ as it can be seen in equations 5.11 - 5.14, and calculated the relative error of the sum of the errors of all cells and the number of cells which are plotted by ExaHyPE, like it is displayed in equation 5.16. X and Y are the sets of coordinates of all cells our plotter outputs.

$$\epsilon_{cell}(x,y,t) = |h_{numeric}(x,y,t) - h(x,y,t)| \tag{5.15}$$

$$RelativeError(t) = \left(\sum_{x \in X, y \in Y} \epsilon_{cell}(x,y,t)\right) / \#numberOfCells_{output} \tag{5.16}$$

With these formulas I calculated the relative error for three different resolutions, 729, 6561 and 59049 cells, and for each I set ϵ to $1e-1$, $1e-2$, $1e-3$. Note that the amount of cells are the number of cells ExaHyPE works with. Since we are working with $order = 3$, the number of outputted cells, i.e. the number we used in equation 5.16, is actually the number of cells multiplied by 9. Furthermore the RelativeError value for 729 cells with $\epsilon = 1e-3$ is missing, since the time step got very small and ExaHyPE

stops the simulation if the time step is too small. The same is true for smaller values than $1e-3$ for epsilon at all resolutions.

As it can be seen it Fig. 5.5 the relative error is the highest at $\epsilon = 1e-2$. We would expect the error to get smaller when ϵ gets smaller as well as when the resolution gets higher. This is partly true for $\epsilon = 1e-1$ and $\epsilon = 1e-3$, but $\epsilon = 1e-2$ has a significant larger RelativeError than both of them. In Fig. 5.6 the left upper picture shows the initial condition of the oscillating lake, which is also the correct solution at $t = \frac{2\pi}{\omega}$. In the right upper picture the result of the simulation with $\epsilon = 1e-1$ at $t = \frac{2\pi}{\omega}$ is shown and it can be seen that the result is rather accurate. The solution of the simulation with $\epsilon = 1e-3$ looks similar. At the bottom of Fig. 5.6 the left picture shows the result of the simulation with $\epsilon = 1e-2$ at $t = \frac{2\pi}{\omega}$. It can clearly be seen that the result with $\epsilon = 1e-2$ is worse than the result with $\epsilon = 1e-1$. The picture on the bottom right shows the solution for $\epsilon = 1e-4$ at $t = 0.7$. Some waves are building up at the right side of the lake, whose momentum are increasing, which leads to a smaller time step until the simulation is stopped shortly after.

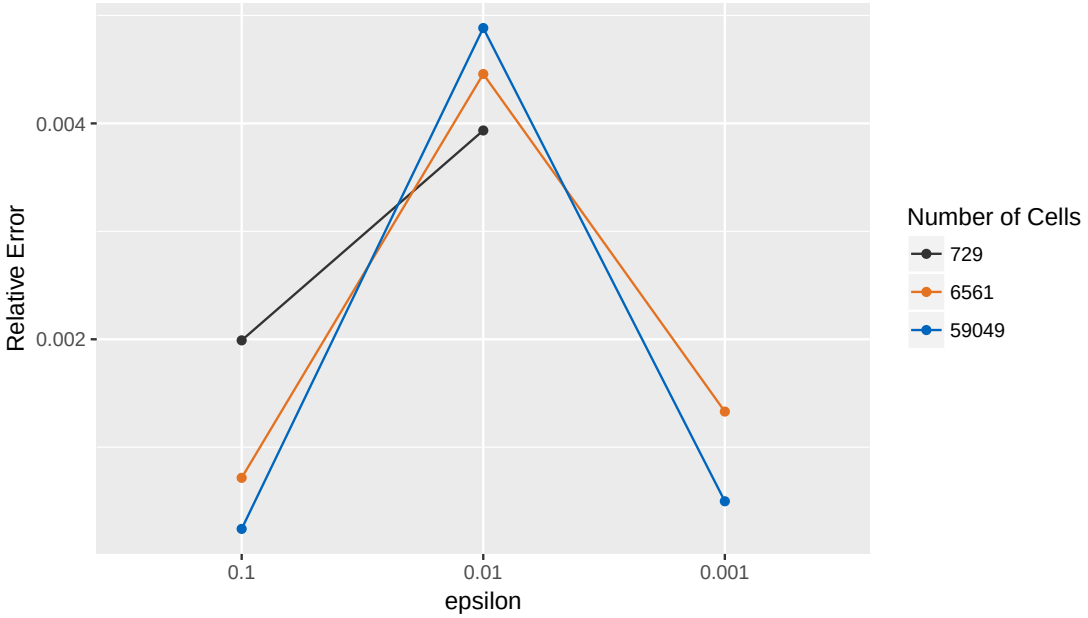


Figure 5.5: Comparison of relative errors in the oscillating lake benchmark with different resolutions and different ϵ

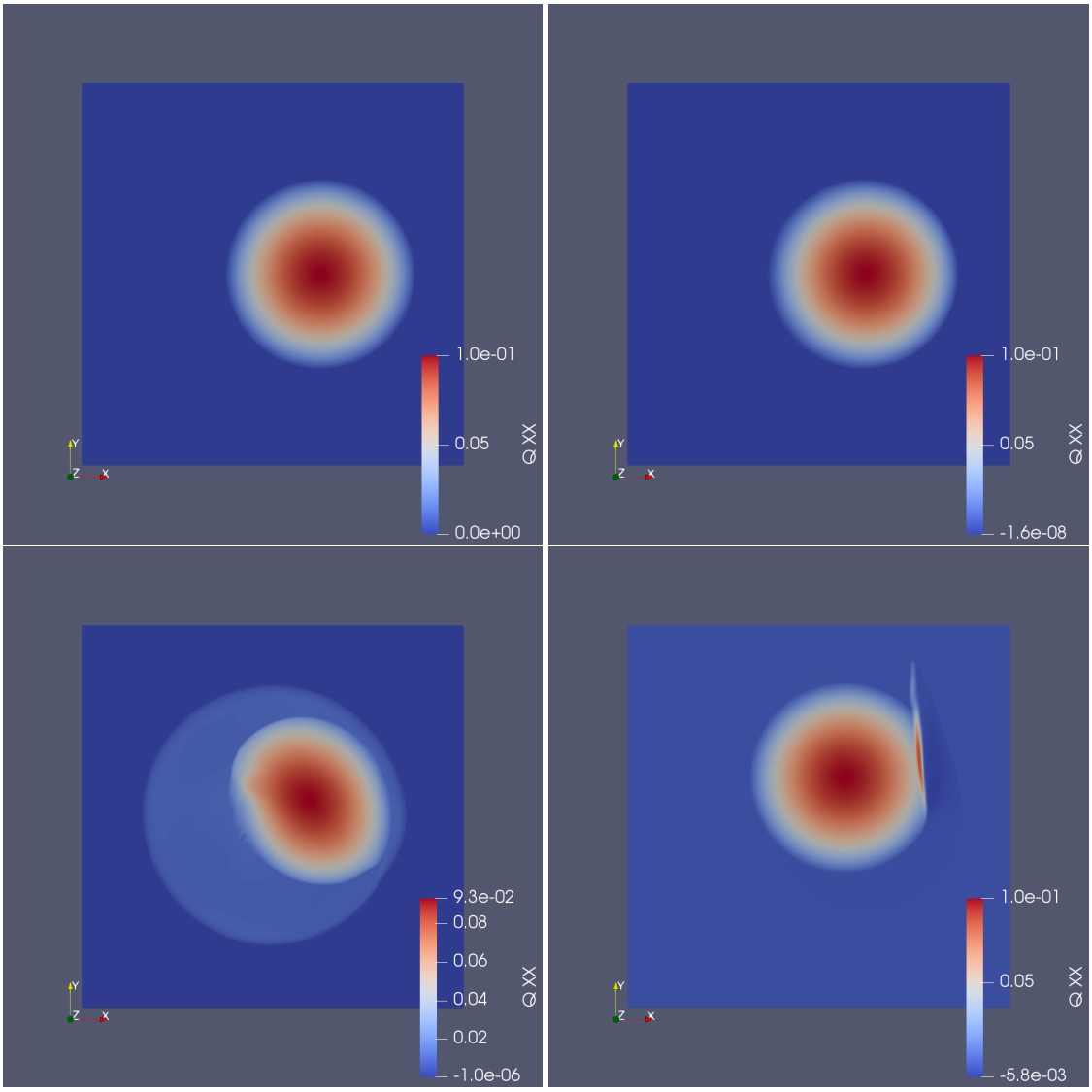


Figure 5.6: The two upper pictures together show what cells are in between 1e − 3 and 1e − 4. The picture on the bottom left shows the lake at $t = 2.7$ with 1e − 3 and the picture on the bottom right with 1e − 4.

6 Conclusion

In this thesis I described the implementation of the SWE in ExaHyPE and how I implemented a new Riemann solver to enable the wetting and drying of cells. I tested the performance of the solvers with shared memory and distributed memory parallelisation as well as the accuracy of a limited solver, which used the ADER-DG solver of ExaHyPE combined with a finite volume solver and the new Riemann solver for wetting and drying.

Chapter 3 described the mathematical structure of the new implement Riemann solver, how the boundary conditions were set for all tests and the initial tests for the different solvers I used. The performance tests in chapter 4 show that ExaHyPEs performance with shared memory parallelisation, i.e. TBB, scales, as expected, better when using a higher order or a higher resolution. When using distributed memory parallelisation, i.e. MPI, we need to use $9^x + 1$ processes to get the best results and here the same as for TBB was true when it comes to a higher order or a higher resolution. In both cases using a higher order led to less dof per second, which we did not expect. The "solitary wave on a simple beach" benchmark shows that our implemented Riemann solver produces rather accurate solutions and matches with the measured results of the laboratory experiment. We also showed that we can reduce the computational effort by using AMR without decreasing the quality of the result. The second benchmark showed that in a very demanding scenario our solver produces some unintentional waves which are reduced by increasing ϵ.

In further research the oscillating lake benchmark could be tested with the limited solver and AMR as well as the solvers could be tested with real tsunami events by using recorded data, for example of the tsunami events of Tohoku and Sumatra like it is done in [12].

List of Figures

List of Tables

Bibliography

[1] *Benchmark 4: Laboratory: Solitary Wave on a Simple Beach*. July 4, 2018. URL: https://github.com/rjleveque/nthmp-benchmark-problems/blob/master/BP04-JosephZ-Single_wave_on_simple_beach/Benchmark4_description.pdf.

[2] A. Breuer, A. Heinecke, L. Rannabauer, and M. Bader. "High-Order ADER-DG Minimizes Energy- and Time-to-Solution of SeisSol." In: *High Performance Computing*. Ed. by J. M. Kunkel and T. Ludwig. Cham: Springer International Publishing, 2015, pp. 340–357. ISBN: 978-3-319-20119-1.

[3] M. Dumbser and D. Balsara. "A new efficient formulation of the HLLEM Riemann solver for general conservative and non-conservative hyperbolic systems." In: 304 (Jan. 2016), pp. 275–319.

[4] M. Dumbser, D. S. Balsara, E. F. Toro, and C.-D. Munz. "A unified framework for the construction of one-step finite volume and discontinuous Galerkin schemes on unstructured meshes." In: *Journal of Computational Physics* 227.18 (2008), pp. 8209–8253. ISSN: 0021-9991. DOI: https://doi.org/10.1016/j.jcp.2008.05.025.

[5] M. Dumbser and R. Loubère. "A simple robust and accurate a posteriori sub-cell finite volume limiter for the discontinuous Galerkin method on unstructured meshes." In: *Journal of Computational Physics* 319 (2016), pp. 163–199. ISSN: 0021-9991. DOI: https://doi.org/10.1016/j.jcp.2016.05.002.

[6] D. L. Eager, J. Zahorjan, and E. D. Lazowska. "Speedup versus efficiency in parallel systems." In: *IEEE Transactions on Computers* 38.3 (1989), pp. 408–423.

[7] F. I. González, R. J. LeVeque, P. Chamberlain, B. Hirai, J. Varkovitzky, and D. L. George. "Validation of the geoclaw model." In: *NTHMP MMS Tsunami Inundation Model Validation Workshop. GeoClaw Tsunami Modeling Group*. 2011.

[8] R. J. LeVeque. *Finite volume methods for hyperbolic problems*. Vol. 31. Cambridge university press, 2002.

[9] *MAC Cluster*. June 20, 2018. URL: http://www.mac.tum.de/wiki/index.php/MAC_Cluster.

[10] O. Meister. "Sierpinski Curves for Parallel Adaptive Mesh Refinement in Finite Element and Finite Volume Methods." Dissertation. München: Technische Universität München, 2016.

[11] O. Meister, K. Rahnema, and M. Bader. "Parallel Memory-Efficient Adaptive Mesh Refinement on Structured Triangular Meshes with Billions of Grid Cells." In: *ACM Trans. Math. Softw.* 43.3 (Sept. 2016), 19:1–19:27. ISSN: 0098-3500. DOI: 10.1145/2947668.

[12] L. Rannabauer, M. Dumbser, and M. Bader. "ADER-DG with a-posteriori finite-volume limiting to simulate tsunamis in a parallel adaptive mesh refinement framework." In: *Computers & Fluids* (2018). ISSN: 0045-7930. DOI: https://doi.org/10.1016/j.compfluid.2018.01.031.

[13] D. A. Reed and J. Dongarra. "Exascale Computing and Big Data." In: *Commun. ACM* 58.7 (June 2015), pp. 56–68. ISSN: 0001-0782. DOI: 10.1145/2699414.

[14] A. Schwarz, D. E. Charrier, F. Guera, J.-M. Gallard, B. Hazelwood, P. Samfaß, S. Köppe, T. Weinzierl, and V. Varduhn. *ExaHyPE Guidebook*. Feb. 19, 2018.

[15] *TOP500 List - June 2018*. July 15, 2018. URL: https://www.top500.org/list/2018/06/.

[16] T. Weinzierl. "The Peano software - parallel, automaton-based, dynamically adaptive grid traversals." In: *CoRR* abs/1506.04496 (2015). arXiv: 1506.04496.

Todo list