Technische Universität München
Fakultät für Informatik
Lehrstuhl für Informatik mit Schwerpunkt Wissenschaftliches Rechnen

Sierpinski Curves for Parallel Adaptive Mesh Refinement in Finite Element and Finite Volume Methods

Oliver Meister

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Die Dissertation wurde am 24.08.2016 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 12.11.2016 angenommen.

Contents

Acknowledgements

First of all, I would like to cordially thank my supervisor, Prof. Dr. rer. nat. Michael Georg Bader, for providing scientific advise and ensuring continued funding of my work throughout these past years. Furthermore, my appreciation goes to my mentors Prof. Dr.-Ing. Rainer Helmig and PD Dr. rer. nat. Bernd Flemisch (both Universität Stuttgart) for their encouragement and enthusiasm in my work, motivating me to spend considerable effort in my research. I am also very grateful to my parents for their financial support during my studies and to Tina J. Schmidt for her valuable feedback during the final phase of this thesis. All other help, especially from the kind people who offered to read my first drafts, is much appreciated.

Abstract

Mesh-based methods for the numerical solution of partial differential equations must tackle the hardware trend towards many-core architectures with less memory per core and higher penalties from memory operations. Dynamically adaptive mesh refinement efficiently invests memory, however generalization to different discretizations and application to high-performance computing without sacrificing granularity is difficult. Implementations must provide a flexible interface and fast, parallel mesh refinement on fully adaptive grids. Currently existing approaches cover these requirements only partially.

This thesis investigates parallel, adaptive, structured triangular grids that are generated by newest vertex bisection corresponding to the Sierpinski space-filling curve. Grid traversals are implemented purely on stack- and stream-based data structures that grant inherent memory efficiency and, via the space-filling curve, scalable heuristics for parallelization. Based on this idea, the software package sam(oa)^2 was developed. It enables finite-element-type as well as finite-volume-type applications with matrix-free, element-oriented formulations to exploit adaptive meshes on clusters or supercomputers, while hiding the complexity of adaptivity and parallelization.

The performance of the code affects low-order discretizations in particular. Therefore, it was tested on a memory-bound two-phase porous media flow scenario with a semi-implicit mixed finite element and finite volume discretization, as well as a weakly compute-bound tsunami wave propagation scenario with explicit finite volume discretization. Good memory efficiency and excellent scalability of large problems on up to 8,000 cores are achieved. Furthermore, extending the porous media flow scenario towards a full reservoir model, the difficult SPE10 oil recovery benchmark was solved on 2.5D prism grids with horizontal mesh refinement. Hence, complex scenarios with billions of elements are executed efficiently on thousands of cores, using highly frequent, fully dynamically adaptive mesh refinement.

Zusammenfassung

Gitterbasierte Methoden für die numerische Lösung partieller Differentialgleichungen müssen die Tendenz zu Mehrkernarchitekturen mit weniger Speicher pro Rechenkern und höheren Einbußen bei Speicheroperationen bewältigen. Dynamisch adaptive Gitterverfeinerung setzt Speicher effizient ein, allerdings ist eine Verallgemeinerung für verschiedene Diskretisierungsverfahren und ein Einsatz im Hochleistungsrechnen schwierig, ohne dabei Granularität aufzugeben. Eine Implementierung muss eine flexible Schnittstelle und schnelle, parallele, adaptive Gitterverfeinerung bereitstellen. Zurzeit existierende Lösungen erfüllen diese Anforderungen nur teilweise.

Die Arbeit untersucht parallele, adaptive, strukturierte Dreiecksgitter, die passend zur raumfüllenden Sierpinskikurve über die *Newest Vertex Bisection*-Methode erstellt werden. Gittertraversierungen sind rein über Keller- und Strom-basierte Datenstrukturen realisiert, die inhärent speichereffizient sind und skalierbare Heuristiken für eine Parallelisierung gewähren. Basierend auf dieser Idee wurde das Softwarepaket sam(oa)^2 entwickelt. Es erlaubt Anwendungen, die Finite-Elemente und Finite-Volumen-Diskretisierungen mit elementweisen, matrixfreien Formulierungen verwenden, adaptive Gitter auf Rechnerverbünden oder Supercomputern auszunutzen, während der Anwendung die Komplexität von Adaptivität und Parallelisierung verborgen bleibt.

Diskretisierungen mit niedriger Ordnung sind besonders von der Geschwindigkeit der Software abhängig. Daher wurde sam(oa)^2 auf einem speicherbeschränkten Szenario für Mehrphasenströmungen in porösen Medien mit einer kombinierten Finite-Elemente- und Finite-Volumen-Diskretisierung, sowie einem schwach rechenbeschränkten Szenario für die Wellenausbreitung von Tsunamis mit einer expliziten Finite-Volumen-Diskretisierung getestet.

Eine gute Speichereffizienz und eine hervorragende Skalierbarkeit von großen Problemen auf bis zu 8.000 Kernen wurden erreicht. Darüber hinaus wurde das schwierige SPE10-Benchmark für Ölgewinnung auf 2.5D Prismengittern mit horizontaler adaptiver Gitterverfeinerung gelöst. Demnach kann sam(oa)^2 komplexe Szenarien, die Milliarden von Gitterzellen enthalten, auf Tausenden von Rechenkernen mit hochfrequenter, dynamisch voll-adaptiver Gitterverfeinerung simulieren.

Notation

The following typographic conventions are used for mathematical formulas in this thesis.

Variables

δ, a	Constant or scalar.
$M = \{a, b, c\}$	Set, an unordered collection.
$N = (a, b, c)$	Tuple, an ordered collection.
$\mathbf{a}^T = (a_1, a_2, a_3, \ldots)$	Row vector, typically in $\mathbb{R}^n$ for $n \in \mathbb{N}$.
$\mathbf{b} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \end{pmatrix}$	Column vector, typically in $\mathbb{R}^n$ for $n \in \mathbb{N}$.
$\mathbf{C} = \begin{pmatrix} c_{1,1} & c_{1,2} & \cdots \\ c_{2,1} & c_{2,2} & \\ \vdots & & \ddots \end{pmatrix}$	Matrix, typically in $\mathbb{R}^{m \times n}$ for $m, n \in \mathbb{N}$.

Sets

$\{a, \ldots, b\}$	Closed interval in $\mathbb{N}$ that includes both a and b for $a, b \in \mathbb{N}$.
$[a, b]$	Closed interval in $\mathbb{R}$ that includes both a and b for $a, b \in \mathbb{R}$.
$[a, b[$	Half-open interval in $\mathbb{R}$ that includes a and excludes b.
$]a, b]$	Half-open interval in $\mathbb{R}$ that excludes a and includes b.
$]a, b[$	Open interval in $\mathbb{R}$ that excludes both a and b.

Operators

$\mathbf{A}^T,\ \mathbf{a}^T$	Transpose of a matrix $\mathbf{A}$ or a vector $\mathbf{a}$.
$\mathbf{AB},\ \mathbf{Ab}$	Matrix product, where $(\mathbf{AB})_{i,j} = \sum_k A_{i,k} B_{k,j}$. Column vectors are treated as single-column matrices.
$\mathbf{a} \cdot \mathbf{b}$	Dot product or inner product $\mathbf{a}^T \mathbf{b}$ of two vectors.
$\mathbf{a}^2,$	Short form of $\mathbf{a}^T\ \mathbf{a}$.
$\lvert\mathbf{a}\rvert$	Euclidean norm $\sqrt{\mathbf{a}^T \mathbf{a}}$ of a vector or a scalar.
f'	Ordinary derivative of the function $f : \mathbb{R} \to \mathbb{R}$.
$\partial_x f,\ f_x$	Partial derivative of the function $f : \mathbb{R} \times \mathbb{R} \times \ldots \times \mathbb{R} \to \mathbb{R}$ with respect to the variable x.
f_{xy}	Short form of the second-order derivative $(f_x)_y$.
∇	Spatial derivative vector, defined as $\nabla = (\partial_x, \partial_y, \partial_z)^T$ in 3D.
∇f	Gradient of a scalar field f, in 3D equal to $(f_x, f_y, f_z)^T$.
$\operatorname{div}(\mathbf{v})$	Divergence $\nabla \cdot \mathbf{v} = u_x + v_y + w_z$ of a vector field $\mathbf{v} = (u, v, w)^T$ in 3D.
Δx	Difference of quantities. Another common meaning for this symbol is the Laplace operator, which is not used in this thesis.
$\int\limits_{\Omega} f\ d\Omega$	Volume integral of the function $f : \mathbb{R}^n \to \mathbb{R}$ over the set Ω.
$\int\limits_{\partial\Omega} \mathbf{f} \cdot\ d\mathbf{n}$	Surface integral of the function $f : \mathbb{R}^n \to \mathbb{R}$ over the boundary set $\partial\Omega$ with the normal $\mathbf{n}$.

PART I

INTRODUCTION

1 Efficiency, Flexibility, and Scalability

As of June 2016, the world's top ten supercomputers feature three heterogeneous systems with accelerator cards (Tianhe-2, Titan, Piz Daint), four of them use reduced instruction set computing architectures (TaihuLight, Sequoia, K Computer, Mira) and only three of them (Trinity, Hazel Hen, Shaheen II) could be considered classical homogeneous, general purpose systems, although each of them contains more than 150,000 compute cores [122]. While the bulk of the top 500 systems still falls into the latter category, there is a trend towards massive use of cheaper cores with less general and more specialized instructions, smaller clock rates, more arithmetics per data fetch, and less memory per core. Software in high performance computing must adapt to decreasing resources per core while utilizing the increasing degree of hardware parallelism [63, 96].

This thesis focuses on computational fluid dynamics in geosciences, where propagation of fluids or waves in domains with complex geometric features is investigated. These problems are modeled by partial differential equations that are discretized with finite element or finite volume methods. Time complexity of these systems increases in the best case linearly with the size, which may easily surpass a million unknowns for large scale problems. At this point the capabilities of a desktop machine are not sufficient anymore and clusters or supercomputers must be employed in order to return a solution in a feasible time.

1.1 Requirements

To tackle these challenges, three formal requirements for a software solution are extracted from the given problem setting:

First, resources must be managed efficiently. Memory may be invested only if it is necessary or beneficial to the solution. Fully adaptive mesh refinement should be used to create dynamically adaptive meshes that are capable of locally resolving features. For example, wave fronts in tsunami simulations are often of lower fractal dimension than the domain [93]. Inundation of coastal regions may reach only tens to hundreds of meters whereas the domain is typically thousands of kilometers wide. When memory is accessed, its access time should be minimized by applying cache-friendly algorithms. Computation time for compute-bound codes must be optimized by exploiting hardware capabilities and reducing program overhead. For parallel execution, this also implies minimizing synchronization time and limiting communication volume to achieve good scalability. Good heuristics for domain decomposition are required, as well as fast, effective load balancing techniques.

The second requirement is flexibility for a range of applications. A generic interface must be provided that supports finite element, finite volume and discontinuous Galerkin discretization. It should abstract from the underlying grid logic in order to enable the implementation of complex algorithms. Flexibility may be reduced in favor of efficiency to provide the best performance for a restricted set of problems.

The third requirement is scalability. Shared memory and distributed memory systems should be supported by the software in order to exploit the benefits of hierarchical parallelization. Large scenarios with billions of unknowns should scale well on clusters or supercomputers.

1.2 A Parallel Framework based on Space Filling Curve Traversal

In order to satisfy all three requirements in a single application, the framework sam(oa)^2 (Space-filling curves and Adaptive Meshes for Oceanic And Other Applications [103]) was developed based on a Fortran 95 prototype by Vigh [125].

Adaptive mesh refinement and resource-efficient grid management are provided by a traversal scheme based on the *Sierpinski space-filling curve*. This approach requires little structural overhead and manages data in a cache-oblivious access scheme. Finite volume and finite element methods are supported by a kernel-oriented software design, where scenarios are discretized in element-oriented formulations. This is an intentional restriction that guarantees applications the benefits from the efficient traversal scheme. Scalability on high-performance systems is given by a hybrid OpenMP+MPI parallelization, which is based on using space-filling curves for the solution of the 2D partitioning problem. Further techniques for management of communication data structures are also based on properties of the Sierpinski space-filling curve.

Combining all of the three requirements into a single software package is a complicated task due to the complex interactions between the solution components. Among the corollary prerequisites are interface support for adaptive mesh refinement and parallelization, load balancing techniques for dynamically adaptive grids and heterogeneous kernels, as well as scenario design for parallel execution on dynamically adaptive grids.

To define a start point for the thesis, Chapter 2 discusses previous work on space-filling curve traversals, including the methods and software developed so far. Afterwards, the scope of the thesis will be defined more clearly and a road map for the remaining chapters will be presented.

2 Space Filling Curves for Grid Traversals

Space-filling curves were first mentioned in 1890, when Peano realized that it is possible to find a continuous, surjective map from the unit interval $[0,1]$ in $\mathbb{R}$ to the square $[0,1]^2$ in $\mathbb{R}^2$ [90]. For the purpose of this thesis, we define a *curve* as the image of a continuous map $\gamma : [0,1] \to [0,1]^2$. A curve is called *space-filling* if the map γ is surjective. That is, for each point $(x,y)^T$ in the unit square there is an index $s \in [0,1]$ with $\gamma(s) = (x,y)^T$. If γ is surjective, but not necessarily continuous, the image of γ is called a *space-filling order*.

One year after Peano defined space-filling curves in general and the *Peano curve* in particular, Hilbert [55] defined a curve that was later referenced as the *Hilbert curve*, another example for a 2D space-filling order. Others soon followed over the years. The *Morton order* – sometimes called *Z-order* or *Z-curve* – is probably the most famous example of a space-filling order due to its simple bitwise mapping [83]. The declaration as a curve is not accurate however, as the corresponding map for the Morton order is not continuous.

2.1 A History of Space Filling Curve Traversals

In 2006, the framework Peano [89] introduced the concept of cache-oblivious grid iterators based on space-filling curve traversals [14, 126]. The spatial pattern of the Peano curve was used to generate block-structured adaptive meshes with a 3^d-subtree refinement scheme, where d denotes the dimension of the domain. An automation for adaptive grid traversals was suggested, where all numerical data is stored in vertices, which are purely accessed on stacks and streams in a recursive algorithm. Hence, there is no random or strided memory access and the method is cache-oblivious by design.

As meshes created by the Peano curve have an inherent 3^d block structure, they are non-conforming and require numerical methods that support the treatment of hanging nodes. A solution to this problem was known in 1991 already, when Mitchell proposed to apply *newest vertex bisection* to triangular meshes for adaptive mesh refinement [81]. In contrast to Peano grids, the meshes created by this method were always kept in a conforming state.

Bader et al. [13] suggested to combine space-filling curve traversals and newest vertex bisection into a recursive traversal algorithm for triangular grids based on the *Sierpinski curve* – see Figure 2.1a. Named after the Polish mathematician Wacław Sierpiński, the curve in its basic form is constructed by recursive newest vertex bisection of an isosceles right triangle. A detailed description of the curve is found in [131].

As a prototype, a geometric multigrid method was implemented to solve a Poisson equation. The price of conforming grids was the loss of higher-dimensional grids, as there is no straightforward transformation of the method to 3D space and beyond [8].

For adaptive grid traversals, a cache-oblivious scheme similar to Peano curve traversal was developed. As illustrated in Figure 2.1b, the refinement tree is a binary tree, that is encoded with a single bit per tree node, which requires little memory overhead in a serialized data structure. In Figure 2.2a, a conforming Sierpinski grid is traversed along the Sierpinski curve. The corresponding traversal automation

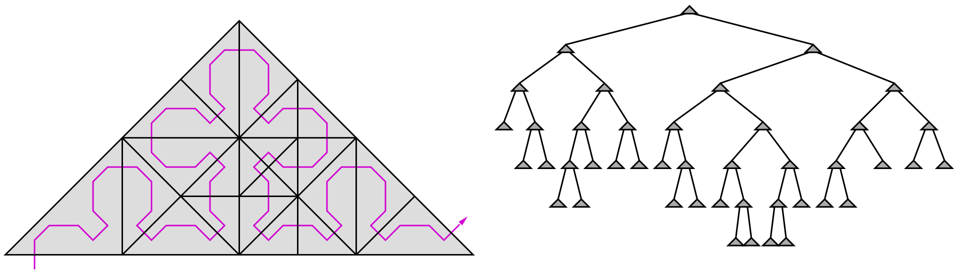

(a) Adaptive triangular mesh **(b)** Refinement tree

Figure 2.1: An adaptive mesh created by newest vertex bisection and the corresponding refinement tree. As each node of the binary tree must possess only the information whether it is a leaf or not, only a single bit per node is required for storage. Tree serialization is therefore possible with a very small memory footprint.

in Figure 2.2b purely uses stacks and streams for data storage. In contrast to Peano, cell and edge data are stored in addition to vertex data.

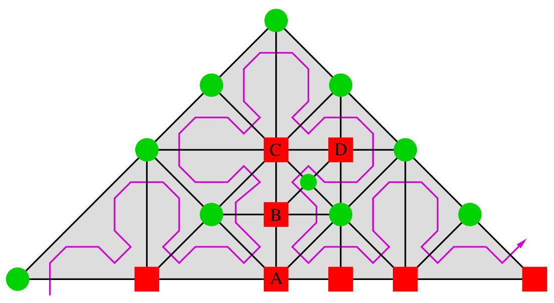
A second application for Sierpinski traversals was presented in [9] where a finite volume solver for the *Shallow Water Equations* was implemented in Fortran 95. Vigh extended the implementation by a fast iterative traversal scheme and parallelized the code for distributed memory architectures [12, 125]. On top of this idea, Schreiber [110] developed a parallel framework [115] for hyperbolic equations targeting many-core machines, which he applied to a first-order discretization of the Shallow Water Equations.

2.2 Scope of the Thesis

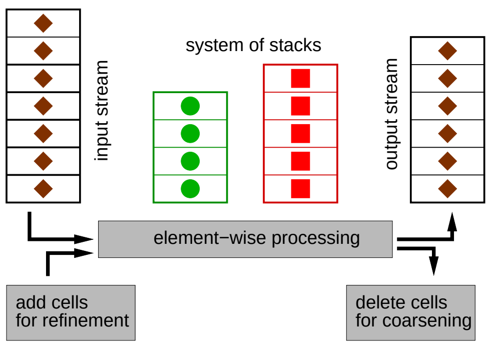
The starting point of the implementation for this thesis is the work by Vigh [125]. In order to provide support for larger scenarios, the parallelization was extended and improved to allow execution of the code on clusters and supercomputers. A parallelization toolkit for shared and distributed memory systems was added. It includes a hybrid OpenMP+MPI parallelization and offers several load balancing techniques for a range of test cases on high-performance systems. The corresponding additions are discussed in Chapter 3.

Additionally, the approach in [125] was generalized from a solver for the Shallow Water Equations to a framework for classes of Partial Differential Equations with a clear restriction to element-oriented methods. Direct access to neighbor elements violates the stack-&-stream paradigms and is prohibited. While this restriction excludes discretizations on large stencils, such as high order finite difference and finite volume methods, support for finite element and discontinuous galerkin methods is provided with the strict limitation of allowing interaction only between local data. The application interface of sam(oa)^2 is presented in Chapter 4.

To demonstrate the capability of solving complicated problems with the mentioned restrictions in place, two test cases will be discussed. The first test case in Chapter 5 is an oil recovery problem that is simulated by a two-phase porous media flow solver for pressure and saturation unknowns. A 2D implementation and a 2.5D extension for the purpose of solving a 3D benchmark problem will be presented. Following the theory, a numerical analysis in Chapter 6 investigates correctness and accuracy of the solution for a set of benchmark problems. Afterwards, the performance of the scenario is evaluated in Chapter 7, where metrics for sequential and parallel execution are defined and evaluated using reference data and scalability tests. The focus of the analysis lies on the efficiency of grid traversals



(a) Sierpinski mesh



(b) Stack-&-stream automaton

Figure 2.2: An adaptive mesh created via newest vertex bisection is traversed along the Sierpinski curve (violet line). Vertices are colored red and green if they are on the left and right of the curve, respectively. The red vertices A, B, C, and D are accessed in-order when the curve passes them for the first time from adjacent elements. When they are passed for the last time, they are accessed in reverse order. An automaton translates this access pattern to an algorithm that reads cell, edge, and vertex data from an input stream, stores them on separate red and green stacks when they are visited for the first time, and writes them to output streams when they are visited for the last time. In the next grid traversal, input and output streams are switched.

and the influence of communication on scalability.

The second test case is a tsunami wave propagation problem, described in Chapter 8. In this application, the 2D shallow water equations with bathymetry source terms are solved to simulate mass and momentum of oceanic tsunami waves. A short numerical analysis will follow in Chapter 9 in order to show correct solver behavior on some benchmark cases and on real tsunami data. The performance evaluation in Chapter 10 focuses on adaptive mesh refinement and load balancing, as the scenario is inherently imbalanced due to heterogeneous kernel execution times. The time steps are comparably cheap and affected much more by overhead from remeshing than the porous media flow test case.

Finally, the conclusion discusses how well the goals set by the thesis were achieved and gives an outlook on possible future improvements.

PART II

SOFTWARE CONCEPTS FOR 2D ADAPTIVE MESH REFINEMENT ON HIGH PERFORMANCE SYSTEMS

3 Parallelization on Sierpinski Sections

Parallelization of meshes always implies solving the particularly hard problems of multidimensional domain decomposition and load balancing. Finding a uniform partitioning for a mesh that minimizes communication corresponds to the *minimum k-section* problem in graph theory, where the set of vertices of a graph is divided into k subsets of equal size, while minimizing the edge cut. This problem is known to be NP-hard even for $k = 2$ [46].

METIS and ParMETIS [78] apply an edge matching on coarse graphs [62] to approximate the restricted k-way partitioning problem and support diffusive schemes for adaptive mesh refinement [107, 108]. Zoltan [133] uses hypergraph partitioning [22] and achieved similar results to ParMETIS in a comparative study in 2007 [31]. Both methods produce high quality partitions and are excellent for solving static load balancing problems but can ultimately become very expensive in a highly dynamic environment where frequent remeshing is necessary.

A less accurate, but simpler and faster approximation is partitioning based on quad-trees or octrees. In p4est [29, 59, 60], the domain is decomposed by cutting the unit interval into segments of equal size and transforming them into a 2D or 3D domain via Morton ordering. While the Morton order allows a very fast evaluation, it creates fragmented partitions, as the underlying map from 1D intervals to the domain is not continuous. Even though there are at most two such fragments per partition [27], increased communication may result compared to using a true space-filling curve.

An improvement is suggested in [66, 118], where octree partitioning is used for a Fast Multipole Method. Here, the strict condition of returning equally sized partitions is relaxed in favor of nicely shaped boundaries. After first distributing load via Morton ordering, a second algorithmic step maps partitions to full octants of a coarse octree in order to reduce the surface of each partition and the corresponding communication volume.

3.1 Sierpinski Curves for Parallelization

Due to its support for explicit finite volume methods, sam(oa)^2 targets frequently changing grids, where dynamically adaptive refinement and load balancing may be necessary in each time step. We therefore follow the quick, canonical approach of cutting the grid into parts of uniform load along the Sierpinski curve. In contrast to Morton ordering, the *Hölder-continuity* of the Sierpinski curve [134] ensures well-formed partitions, which are even edge-connected. Early experiments lead to good results [20, 135]. Coming from another direction, the *REFTREE* algorithm [82] searches for Hamilton paths in grids to define connected partitions and returns the Sierpinski curve for 2D grids based on newest vertex bisection.

Vigh [12, 125] parallelized a solver for the shallow water equations based on the Sierpinski traversal scheme explained in Chapter 2 by using Sierpinski cuts for partitioning. Here, sam(oa)^2 follows the approach by Vigh and applies a new load balancing algorithm, where load is modeled as an abstract cost function. Load is either defined as a weighted sum of the number of grid cells and grid vertices, or as an estimate for the true execution time based on high precision measurements. Coarse grid mapping of load is not supported by sam(oa)^2, but a related approach based on subtree partitioning was investigated in [110, 112].

For hybrid parallelization we allow assignment of multiple parts of the grid to a single process. We call these parts *sections* to indicate that they correspond to contiguous intervals of the Sierpinski curve. A section is an independent computation unit in the grid, that consists of the elements given by mapping a 1D interval to the Sierpinski curve, as well as the edges and vertices associated with the elements. Together, the union of all sections forms the grid.

3.2 Hybrid OpenMP+MPI Load Balancing Schemes

As sam(oa)^2 divides the grid into sections for parallelization, the load balancing mechanism must define number and size of sections and how to distribute them to cores, while ideally maintaining a uniform load. These two problems can be treated independently, where a node-local algorithm decides how many sections are created and how large they are, and a distributed algorithm assigns sections to cores as atomic units. Alternatively, a combined approach is possible where sections are split into sizes that guarantee a good load balance after distribution. Both approaches have advantages and will be discussed here.

3.2.1 Load Balancing of Atomic Sections

First, we consider the case of distributing atomic sections. The advantage of this approach is that it is particularly suitable for hybrid OpenMP+MPI load balancing. Figure 3.1 illustrates the process. In the first step, a computation phase is executed, where the cost of each section is determined and cells are marked for refinement and coarsening. The details of evaluating section costs are discussed in Section 3.2.3. Sections can now be distributed among MPI ranks before adaptive refinement by using knowledge of future refinements and coarsening to determine a load approximation. Afterwards, local repartitioning during grid refinement and coarsening restores sections of uniform costs that are easily processed in parallel by OpenMP threads.

The disadvantage of this approach is that load is not balanced optimally due to the strongly constrained subproblem of balancing tasks with varying costs to processes, while preserving their order. Section 3.3.1 discusses in detail how to solve this problem.

3.2.2 Load Balancing with Section Splitting

The alternative approach is to allow sections to be split for load balancing to ensure a uniform load distribution over all cores. Local repartitioning of sections is possible only during adaptive refinement and coarsening in sam(oa)^2. Therefore, section splitting requires load to be balanced after adaptive traversals as seen in Figure 3.2. Hence, sam(oa)^2 switches the order of adaptive refinement and load balancing for this approach. Hybrid OpenMP+MPI parallelization will be affected as the number of sections per core is not homogeneous. This does not affect the load model, but will increase the size of the boundary and therefore influence the real load of each section, i.e. execution and communication time. The benefit of splitting will be small for hybrid parallelization anyway, as a higher number of sections per process will be required per process for scheduling sections to OpenMP threads. A more fine-grained distributed balancing of atomic sections will automatically result that weakens the effect of splitting. Splitting is enabled in sam(oa)^2 at runtime with the command line argument

```
    samoa -lbsplit
```

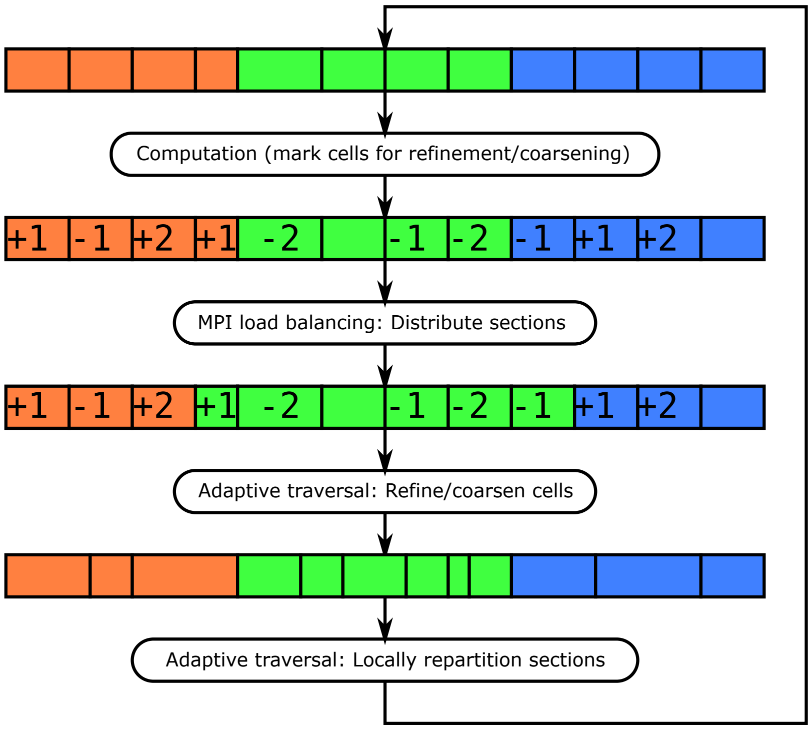

Computation (mark cells for refinement/coarsening)
+1 -1 +2 +1 -2 -1 -2 -1 +1 +2
MPI load balancing: Distribute sections
+1 -1 +2 +1 -2 -1 -2 -1 +1 +2
Adaptive traversal: Refine/coarsen cells
Adaptive traversal: Locally repartition sections

Figure 3.1: Hybrid OpenMP+MPI load balancing of atomic sections. Directly after each computation phase, load is balanced between MPI ranks. Afterwards, the mesh is adaptively refined and sections with homogeneous costs are created locally on each process. This method is particularly suitable for OpenMP parallelization. However, processes will usually be imbalanced after repartitioning due to the coarse-grained MPI load balancing.

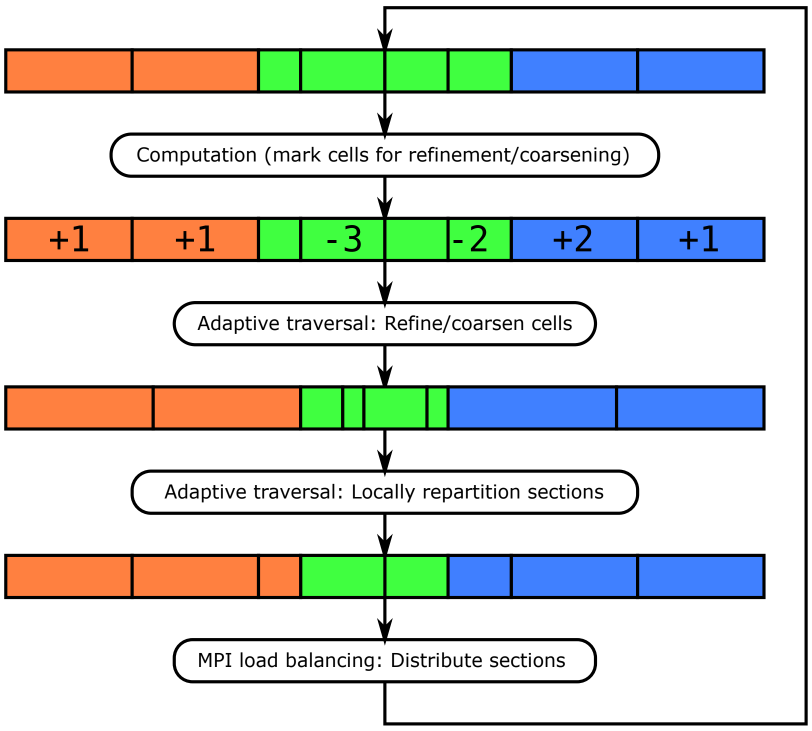

Computation (mark cells for refinement/coarsening)
+1
+1
-3
-2
+2
+1
Adaptive traversal: Refine/coarsen cells
Adaptive traversal: Locally repartition sections
MPI load balancing: Distribute sections

Figure 3.2: Hybrid OpenMP+MPI load balancing with section splitting. After each computation phase, an adaptive traversal that includes local repartitioning is executed to cut the grid into subsets of sections with equal costs that are distributed uniformly among cores. Next, sections are migrated to their intended cores during load balancing. While load is theoretically uniformly distributed over all cores, imbalances can still occur in practice, as sections of varying number and size are harder to balance in shared memory.

3.2.3 Evaluation of Section Costs

While the cost of a section is usually directly associated with cells or degrees of freedom in grid-based applications, some scenarios require a more general approach to define costs.

Simple kernels that perform mainly linear algebra operations generate load that is distributed uniformly on cells, edges, vertices or combinations thereof. For these cases, the cell-based or degree-of-freedom-based model mentioned above is perfectly suitable. However, complex, heterogeneous kernels with branches or while-style loops are not captured. Also, *dead cells*, i.e. cells without computation, are not modeled correctly.

To provide cost estimates for simple and more complex kernels, sam(oa)^2 defines an abstract cost value $c_i \geq 0$ for each section. Currently, two fundamentally different approaches are implemented, where the load of a section is modeled as the weighted sum of the number of cells and the number of vertices or measured by a high-precision timer that instruments all computation in a section and defines the cost c_i as the sum of the computation time.

Linear Cost Model

The linear model exploits that a 2D mesh is a planar, connected graph and therefore subject to Euler's formula, which implies that the number of edges e linearly depends on the number of cells f (minus the exterior face) and the number of vertices v. That is,

$$e + 1 = f + v. \tag{3.1}$$

Therefore, a linear cost model that includes e is as powerful as a linear cost model that neglects it. v could be replaced by the number of boundary edges e_b in the mesh by using

$$v = \frac{1}{2}(f + e_b) + 1. \tag{3.2}$$

However, with f and e_b, the number of degrees of freedom is still two. Hence sam(oa)^2 bases its linear cost model on cells and vertices instead, which usually correspond better to the degrees of freedom in a mesh than cells and boundary edges. The linear model is the default implementation in sam(oa)^2.

Time Based Cost Evaluation

In contrast, the time-based approach is implemented by adding high precision timer instructions before and after each traversal to measure the actual execution time of a grid traversal on each section. The resulting time difference Δt is used to estimate the cost per cell $\frac{\Delta t}{n}$, which is independent of adaptive refinement and coarsening. Time-based evaluation is enabled with the command line argument

```
    samoa -lbtime
```

Both implementations have advantages: the linear model returns consistent costs and is not affected by measuring errors, whereas the time-based approach is able to handle a much broader class of applications with a black-box view of the kernels. Applications for both approaches are presented in this thesis: The linear cost model will be mainly used for the porous media flow scenario in Chapter 5 and the time-based cost model is applied to the tsunami wave propagation scenario in Chapter 8.

3.2.4 Advanced Models for Cost Evaluation

Schaller [105] investigated a scenario on Cartesian grids, where a complex computation kernel performs heterogeneously depending on time and position in the domain. He compared linear and time-based cost evaluation with an advanced performance model of the kernel. He used a state-dependent approximation of kernel performance by classification of input parameters. However, due to its implied overhead, the method performed generally worse than the two standard methods and is not considered further.

3.2.5 Shared Memory Load Balancing

Shared memory load balancing is quite straightforward, as parallel processing units for threads are created easily by assignment of multiple sections to each process. For example, if a process has 4 physical cores, then the number of section per core is a multiple of 4 in sam(oa)^2 to allow for threaded traversal of sections. By design, these are atomic, independent units. Hence, it is simple to assign them statically to a set of concurrently running threads.

For load balancing on atomic sections in Section 3.2.1, load is not balanced uniformly and for load balancing with section splitting in Section 3.2.2, the number of sections per core is not uniform, which may also lead to load imbalance. Hence, a work-stealing mechanism will be useful to reduce intra-process imbalance. Therefore, sam(oa)^2 uses a manual, static assignment of sections to cores and adds optional OpenMP task stealing to mitigate load imbalance within a shared memory domain. Algorithm 3.1 shows how the NUMA (Non-Uniform Memory Architecture)-aware implementation of sam(oa)^2 creates a local task queue for each thread that may be accessed by other threads. With this design, each thread will process its own sections before trying to steal work from other threads, limiting migration of sections between NUMA domains.

Algorithm 3.1: A simplified shared memory version of parallel grid traversals in sam(oa)^2, using Fortran and OpenMP 3 syntax. Sections are manually assigned to cores and task stealing mechanisms are added on top for NUMA-aware parallelization.

```
subroutine traverse_grid(grid, enable_tasks)
  type(t_grid) :: grid
  logical :: enable_tasks
  integer :: i_thread, i_first_section, i_last_section

  i_thread = omp_get_thread_num()
  call grid%get_local_sections(i_thread, i_first_section, i_last_section)

  if (enable_tasks)
    do i = i_first_section, i_last_section
      !$omp task private(i) default(shared)
      traverse_section(grid%section(i))
      !$omp end task
    end do

    !$omp taskwait
  else
    do i = i_first_section, i_last_section
      traverse_section(grid%section(i))
    end do
  end if
end subroutine
```

3.2.6 Repartitioning in Shared Memory

As seen in Figure 3.1, sam(oa)^2 relies on a mechanism to locally repartition sections for distribution of load among threads and processes. To avoid an extra, inherently expensive grid traversal, repartitioning is combined with adaptive refinement into a single step that requires only one grid traversal.

Thread-parallel execution of this adaptive traversal is challenging, as multiple destination sections may require grid data from the same source sections. Threaded creation of destination sections consequently causes concurrent access to streams and stacks of the old grid. Concurrent access to stacks is easily avoided by employing distinct stacks for each thread. Concurrent access to streams may happen, but is restricted to concurrent read access and thus cannot cause race conditions. However, it may degrade performance, due to resource contention. If the number of sections per thread is set to at least two, the probability of concurrent stream access is reduced strongly and most read accesses will be exclusive again. In sam(oa)^2 the number of sections per core $\frac{n}{p}$ is set with the following command line argument:

```
samoa -sections <n/p>
```

The implementation of adaptive traversals will be discussed in detail in Chapter 4.5.

3.3 Distributed Load Balancing on Triangular Adaptive Meshes

Contrary to static grids, dynamic grids impose some restrictions on MPI load balancing, which are caused by adaptive coarsening. Any pair of neighboring cells that is feasible for coarsening, should be able to do so eventually. A policy or a control mechanism, i.e. defragmentation, must promote the case where the cells end up on the same core eventually.

In sam(oa)^2, hybrid load balancing additionally demands frequent merging of sections as displayed in Figure 3.1 and Figure 3.2. This is only possible for sections that are located on the same core and in consecutive Sierpinski order. Hence, in order to restrict data movement for coarsening and section merging, a policy must be employed to ensure that sections will always be located on the same core or on neighbor cores if they are in consecutive Sierpinski order. For an MPI load balancing algorithm, this policy acts as a constraint that must be satisfied at any time. The straightforward solution is to sort all the sections in Sierpinski order and to enumerate the cores by consecutive integers. Then sections are assigned to cores with a non-decreasing mapping, resulting in a 1D load balancing scheme with all sections sorted in Sierpinski order.

An additional complexity arises from the order of adaptive refinement and load balancing if sections are considered atomic. To guarantee a uniform number of sections per core for hybrid parallelization, load must be balanced before adaptive refinement. Hence, knowledge of future adaptive refinement and coarsening must be used to determine an accurate load estimate of a section. In this case, sam(oa)^2 uses the following extrapolation to update the cost c_i of a section

$$c_i \leftarrow c_i \frac{n_i^*}{n_i}, \tag{3.3}$$

where n_i denotes the current number of cells in section i and n_i^* is the expected number of cells after adaptive refinement and coarsening. If section splitting is active, load balancing will be called after adaptive refinement where n_i and n_i^* will be equal. Hence, sam(oa)^2 returns a consistent estimate at all times.

3.3.1 Chains on Chains Partitioning

First, the constrained load balancing problem is formally defined. Assume there are n tasks and p cores, which the tasks are supposed to be distributed to. Each task $i \in \{1, \ldots, n\}$ is characterized by its cost $c_i > 0$, which is measured in some unspecified cost unit. All cores are identical, so the same task has the same execution time on each core.

We define the prefix sum, also known as discrete integral,

$$C_i := \sum_{k=1}^{i} c_k. \tag{3.4}$$

The goal is to find a monotonically increasing map $s : \{0, \dots, p\} \to \{0, \dots, n\}$ that assigns each core $j \in \{1, \dots, p\}$ to an interval of tasks $\{s(j-1)+1, \dots, s(j)\}$, while minimizing the maximum *load per core*

$$t_s := \max_{j \in \{1,\dots,p\}} \sum_{i=s(j-1)+1}^{s(j)} c_i = \max_{j \in \{1,\dots,p\}} \left(C_{s(j)} - C_{s(j-1)} \right). \tag{3.5}$$

For convenience, let $s(0) := 0$ and $s(p) := n$. As implied by the definition of the chains-on-chains problem, neither the order of tasks, nor the order of cores can be changed.

This restriction of the NP-complete *multiprocessor scheduling* problem [45] is known as *chains-on-chains partitioning* and has been investigated in [91] for example, where several solution approaches are suggested. Multiprocessor scheduling is the problem of assigning n independent tasks with varying execution times $c_1, \dots, c_n$ to p cores in such a way that the maximum execution time over all cores is minimized.

Continuous Solution for Section Splitting

If we assume that tasks may be split to achieve a perfect load balance, meaning that s maps $[0, n]$ to $\{1, \dots, p\}$, the problem will be fairly easy to solve as we will achieve a perfect load balance then. To find the corresponding map $\hat{s}$, we determine a lower bound for the maximum load t_s. Fix an arbitrary distribution function s and denote by t_s the maximum load assigned to a core. Then we obtain for each core j

$$t_s \geq C_{s(j)} - C_{s(j-1)}. \tag{3.6}$$

Adding up (3.6) over all p cores yields

$$pt_s = \sum_{j=1}^{p} t_s \geq \sum_{j=1}^{p} \left(C_{s(j)} - C_{s(j-1)} \right) = C_{s(p)} - C_{s(0)} = C_n, \tag{3.7}$$

The value of t_s is therefore estimated by

$$t_s \geq \frac{C_n}{p} \quad \text{for any distribution function } s. \tag{3.8}$$

We now assume that there is a monotonically increasing map $\hat{s} : \{1, \dots, p\} \to \{1, \dots, n\}$ with

$$t_{\hat{s}} = \frac{C_n}{p}, \tag{3.9}$$

then (3.8) and (3.9) imply that $\hat{s}$ minimizes (3.5). In this case

$$t_{\hat{s}} p - C_n = \sum_{j \in \{1,\dots,p\}} \underbrace{\left(t_{\hat{s}} - C_{\hat{s}(j)} + C_{\hat{s}(j-1)} \right)}_{\geq 0} = 0, \tag{3.10}$$

and in (3.6) equality holds for each core j

$$\frac{C_n}{p} = t_{\hat{s}} = C_{\hat{s}(j)} - C_{\hat{s}(j-1)}.$$

To eliminate the double evaluations of $\hat{s}$, we construct the prefix sum from $k = 1$ to j of left- and right-hand side and obtain

$$\frac{j}{p} C_n = C_{\hat{s}(j)}. \tag{3.11}$$

Hence, for each $j \in \{1, \ldots, p\}$, $\hat{s}$ is given implicitly by

$$\hat{s}(j) = i \quad \text{if } \frac{C_i}{C_n} = \frac{j}{p}. \tag{3.12}$$

Now, $\hat{s}$ is well-defined only if the C_i are strictly increasing and $\hat{s}(j) \in \mathbb{N}$ for each core j. If the C_i are strictly increasing, $\hat{s}$ is also unique: Let s_1 and s_2 be any distributions that solve (3.12), then for each j

$$\frac{C_{s_1(j)}}{C_n} = \frac{C_{s_2(j)}}{C_n} = \frac{j}{p} \tag{3.13}$$

and therefore $s_1(j) = s_2(j)$.

If the C_i are reinterpreted as a continuous piece-wise linear function, then $\hat{s}$ always exists and $\hat{s}$ is a solution to splitting-based load balancing. In detail, if $\hat{s}(j) \notin \mathbb{N}$ for some core j, meaning $\hat{s}(j) = r + \alpha$, where $r \in \mathbb{N}$ and $\alpha \in [0, 1[$, then the fractional part α of $\hat{s}(j)$ decides where to split the cost of the corresponding task $r + 1$.

Midpoint Approximation for Atomic Sections

A perfect load balance on atomic sections requires a discrete solution for (3.12), which does not necessarily exist. However, the existence of the continuous solution $\hat{s}$ may be exploited to derive a discrete approximation based on rounding the function $\hat{f}$, defined by

$$\hat{f}(i) := \frac{C_i}{C_n} p. \tag{3.14}$$

Corresponding to the definition of $\hat{s}$, the function $\hat{f}$ returns the core j for the task i if task i is the last task of core j. For all other tasks assigned to core j, it returns a real number between $j - 1$ and j. To obtain an integer-valued map we round the result by

$$f_{\mathrm{ma}}(i) := \left\lfloor \frac{C_i - \frac{1}{2}c_i}{C_n} p \right\rfloor + 1 \qquad \text{for } i = 1, \ldots, n. \tag{3.15}$$

The idea is to assign tasks to cores so that the ratio $\frac{j}{p}$ in (3.12) is approximated as closely as possible. This is achieved by using the midpoint of the cost range for task i:

$$\frac{1}{2}(C_{i-1} + C_i) = C_i - \frac{1}{2}c_i \tag{3.16}$$

to decide the destination core for task i. An example is illustrated in Figure 3.3. The method was first suggested in [80]. We will refer to it henceforth as *midpoint approximation*. Algorithm 3.2 shows a part of the implementation in sam(oa)^2, which communicates on a distributed memory system via MPI.

Using this algorithm, each core eventually gains knowledge of the destination cores for each task it sends, and the source cores for each new task it receives. Load could be propagated either directly to its destination or iteratively, as originally suggested in [62, 109]. However, we found that in sam(oa)^2, an iterative strategy usually performs worse once a good distribution is determined [77], as the added communication volume outweighs the latency issues. Hence, load is propagated directly and the algorithm terminates.

Algorithm 3.2: Distributed load balancing by approximate chains-on-chains partitioning. This incomplete algorithm is executed on each core $j \in \{1, \ldots, p\}$ to find source and destination cores for task exchange. MPI is used to communicate between cores.

```
Input: $j \in \{1, \ldots, p\}$, $s_{j-1}, s_j$, $(c_i)_{i \in \{s_{j-1}+1, \ldots, s_j\}}$, $0 < \epsilon \ll 1$
Output: $k_{\text{start}}, k_{\text{end}}, l_{\text{start}}, l_{\text{end}}$

$l_j \leftarrow \sum_{i=s_{j-1}+1}^{s_j} c_i$ ;
$L_j \leftarrow$ MPI_Scan($l_j$) ;
$C_n \leftarrow$ MPI_AllReduce($l_j$) ;

for $i \in \{s_{j-1}+1, \ldots s_j\}$ do
    $C_i \leftarrow L_j - l_j + \sum_{k=s_{j-1}+1}^{i} c_k$ ;
    $f_i \leftarrow \left\lfloor \frac{C_i - \frac{1}{2}c_i}{C_n} p \right\rfloor + 1$ ;
end

MPI_IRecv(MPI_ANY_SOURCE , $l_{\text{start}}$, START_TAG) ;
MPI_IRecv(MPI_ANY_SOURCE , $l_{\text{end}}$, END_TAG) ;
for $k \in \left\{ \left\lceil \frac{L_j - l_j}{C_n} p \right\rceil + 1, \ldots, \left\lceil \frac{L_j}{C_n} p \right\rceil \right\}$ do
    MPI_Send($k, j$, START_TAG) ;
end
for $k \in \left\{ \left\lfloor \frac{L_j - l_j}{C_n} p \right\rfloor + 1, \ldots, \left\lfloor \frac{L_j}{C_n} p \right\rfloor \right\}$ do
    MPI_Send($k, j$, END_TAG) ;
end

$k_{\text{start}} \leftarrow \left\lfloor \frac{L_j - l_j}{C_n} p \right\rfloor + 1$ ;
$k_{\text{end}} \leftarrow \left\lceil \frac{L_j}{C_n} p \right\rceil$

// Missing steps:
// * Listen for tasks from source cores $l_{\text{start}}, \ldots, l_{\text{end}}$.
// * Send tasks to destination cores
    $\{f_i\}_{i \in \{s_{j-1}+1, \ldots, s_j\}} \subseteq \{k_{\text{start}}, \ldots, k_{\text{end}}\}$.
```

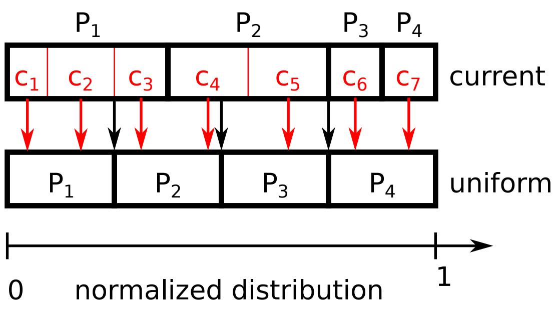


Figure 3.3: An illustration of distributed chains-on-chains partitioning by midpoint approximation on four cores. Tasks are assigned atomically to destination cores with the map f (red arrows). The destination core P_j is then informed of the range of tasks assigned to it by receiving notification messages (black arrows) from its first and last source core. Note that task 4 on core P_2 causes some imbalance as f returns only a partial fit to core P_2.

Optimal Solution Approaches

As mentioned before, chains-on-chains partitioning is a restriction of the NP-complete multiprocessor scheduling problem. It is therefore not immediately obvious whether the restricted problem remains NP-hard. The *EXACT-BISECT* algorithm by Pınar et al. [91] uses a binary search to minimize the imbalance and finds an optimal load distribution in time $\mathcal{O}\left(n + p\log(p)\log(n)\right)$, assuming that the cost functions are bounded by a constant. Unfortunately, the algorithm not only becomes more expensive with more tasks but is also inherently serial. Implementation requires the expensive MPI operations `MPI_Gather` for gathering load data in the root process and `MPI_Scatter` for scattering the new load distribution. Performance is therefore acceptable only up to a few hundred cores as the analysis in Secs. 7.3 and 10.1.2 shows. High-quality approximations for large systems exist [69], but no scalable algorithms are known to find an optimal solution. A variant of EXACT-BISECT is implemented in sam(oa)^2 and enabled with the command line argument:

```
    samoa -lbserial
```

3.3.2 Approximation Quality

The maximum load given by midpoint approximation is

$$t_{\mathrm{ma}}(p) := \max_{j\in\{1,\dots,p\}} C_{\hat{s}(j)} - C_{\hat{s}(j-1)}. \tag{3.17}$$

This value is bounded by the sum of the average load and the cost of the most expensive task

$$t_{\mathrm{ma}}(p) \le \frac{C_n}{p} + \max_{i\in\{1,\dots,n\}} c_i. \tag{3.18}$$

A proof for a similar conjecture is presented in [80]. A lower bound for the optimal maximum load $t_{\mathrm{opt}}(p)$ is given by

$$t_{\mathrm{opt}}(p) \ge \max\left(\frac{C_n}{p}, \max_{i\in\{1,\dots,n\}} c_i\right) \ge \frac{1}{2}\left(\frac{C_n}{p} + \max_{i\in\{1,\dots,n\}} (c_i)\right). \tag{3.19}$$

Therefore,

$$t_{\mathrm{ma}}(p) \le \frac{C_n}{p} + \max_{i\in\{1,\dots,n\}} (c_i) \le 2t_{\mathrm{opt}}(p). \tag{3.20}$$

Thus, midpoint approximation returns a maximum load that is at worst twice the optimal load. Naturally, lower and upper bounds must hold for an optimal chains-on-chains solver, too.

Further analysis is possible if the maximum cost of a task is assumed to be bounded by an application-dependent factor $\alpha \geq 1$ of the average cost of a task, i.e.

$$\max_{i \in \{1,\ldots,n\}} (c_i) \leq \alpha \frac{C_n}{n}. \tag{3.21}$$

Ignoring other sources of serialization, we estimate the parallel efficiency in a strong scaling environment by

$$\frac{t_{\mathrm{ma}}(1)}{p\, t_{\mathrm{ma}}(p)} \geq \frac{C_n}{p\left(\frac{C_n}{p} + \max\limits_{i \in \{1,\ldots,n\}} (c_i)\right)} \geq \frac{C_n}{p\left(\frac{C_n}{p} + \alpha \frac{C_n}{n}\right)} = \frac{1}{1 + \alpha \frac{p}{n}}. \tag{3.22}$$

Hence, given the ratio α, we obtain a lower bound for the theoretical parallel efficiency, which approaches 1 as $\frac{p}{n}$ goes to 0. In sam(oa)^2, the number of tasks per core $\frac{n}{p}$ is a command line argument:

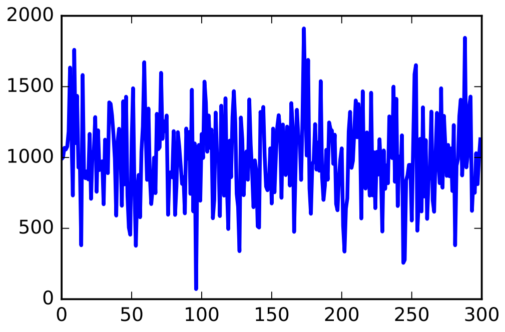
```
samoa -sections <n/p>
```

Consider a scenario with $\alpha \approx 1.4$. To achieve a lower bound of 85% for the parallel efficiency, a ratio of $\frac{n}{p} \geq 8$ is therefore required. In practice, increasing $\frac{n}{p}$ will cause overhead for section management and will reduce the parallel efficiency. There is always a sweet spot, which is usually between 4 and 16.

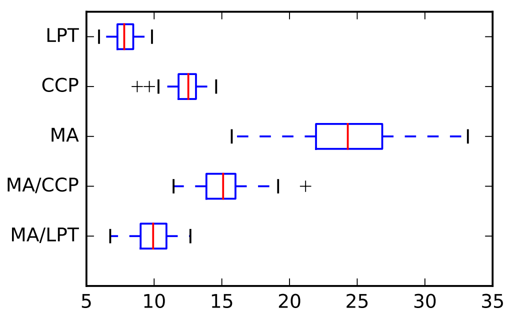
3.3.3 Chains on Chains Solvers in Comparison

While there is an optimal chains-on-chains solver [91], it is hard to judge how well it balances load in absolute numbers. To get an idea, we will take a look at load imbalances for some artificial cost functions and compare results of an optimal chains-on-chains solver, midpoint approximation, and *longest processing time*. Longest processing time is a $\frac{4}{3}$-approximation to the general multiprocessor scheduling problem [51], meaning that the maximum load returned by longest processing time is at most $\frac{1}{3}$ larger than in an optimal solution. In comparison, midpoint approximation is a 2-approximation. Longest processing time reorders tasks and is therefore not applicable to chains-on-chains partitioning, but it will provide a reference solution that shows the impact of the chains-on-chains restriction.

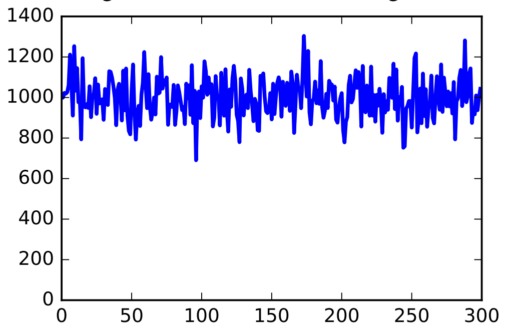
Figure 3.4 illustrates statistical results for artificial tasks with randomly generated cost functions. Midpoint approximation usually returns the worst approximation, especially for rough cost functions with high frequency components. The optimal chains-on-chains solver performs well and returns distributions that are comparable to the results for longest processing time when the variance of the cost function is low. If the variance is larger, the chains-on-chains solver performs worse, but will still return small imbalances most of the time. Hybrid approaches were also tested, where midpoint approximation is used as a fast estimator to distribute tasks onto groups of processes and a second, serial algorithm distributes tasks within groups which are executed concurrently. These variants would allow to apply longest processing time or an optimal chains-on-chains solver at the cost of some additional imbalance. However, as scalability limits group sizes to small numbers, hybrid methods naturally converge towards midpoint approximation on larger numbers of cores and the desired effect disappears. Hence, sam(oa)^2 provides an optimal chains-on-chains solver up to moderate numbers or cores, as well as midpoint approximation for extreme scalability. Hybrid methods are not provided, as we showed that they lose their advantage on high numbers of cores. A hierarchy of algorithms would be required to better cope with this issue.



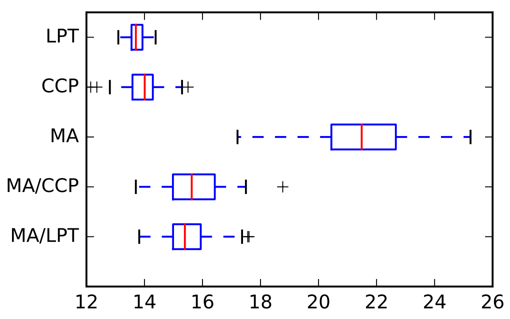
(a) Rough cost distribution with high variance



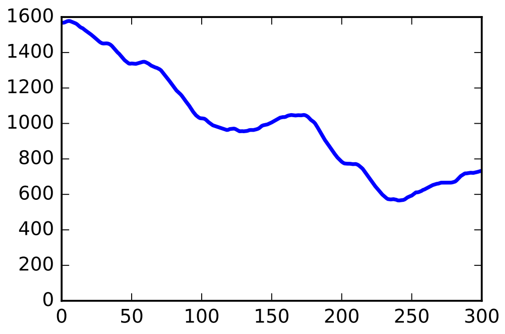
(b) Distribution of load imbalance[%] for (a)



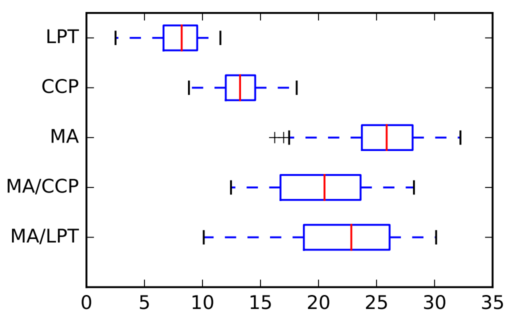
(c) Rough cost distribution with low variance



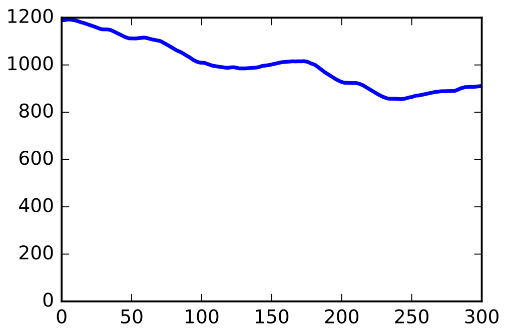
(d) Distribution of load imbalance[%] for (c)



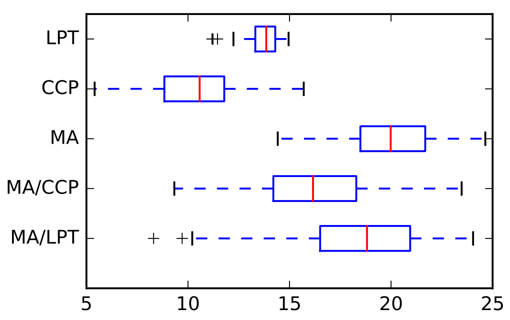
(e) Smooth cost distribution with high variance



(f) Distribution of load imbalance[%] for (e)



(g) Smooth cost distribution with low variance



(h) Distribution of load imbalance[%] for (g)

Figure 3.4: Randomly generated tasks and their theoretical load imbalance. Four different combinations of smoothness and variance parameters are chosen to generate 100 random cost functions for 300 tasks, which are distributed to 70 cores. The box-and-whisker diagrams show the predicted imbalance distributions for longest processing time (LPT), optimal chains-on-chains partitioning (CCP), midpoint approximation (MA), and hybrid methods that combine two algorithms hierarchically. Each median is represented by vertical red lines and boxes display 25% to 75% quartiles. High confidence intervals extend along dashed lines and outliers are plotted as crosses.

3.3.4 Chains on Chains Partitioning with Intermediate Synchronization Points

So far, we assumed that the cost of a task is an atomic value that returns uniform workload when it is distributed uniformly. However, this is a simplification, as a grid traversal contains multiple stages with possible synchronization points in-between, see Section 4.3 for more details. Additionally, sam(oa)^2 performs load balancing only when it is triggered by the application. Hence, multiple, possibly distinct traversals may be executed between two load balancing calls.

We model the plurality of traversal stages and traversals by splitting computation between two load balancing calls into m phases, where each task $i \in \{1, \dots, n\}$ is divided into m subtasks with phase-dependent cost $c_{k,i} > 0$ for $k \in \{1, \dots, m\}$. Each task, with all its subtasks, is still assigned to a single core and may not be reassigned between two load balancing calls. An optimal, monotonically increasing distribution function $s : \{0, \dots, p\} \to \{0, \dots, n\}$ therefore maps each core $j \in \{1, \dots, p\}$ to an interval of tasks $\{s(j-1)+1, \dots, s(j)\}$ and minimizes the accumulated maximum load per core now; that is

$$t_s := \sum_{k=1}^{m} \max_{j \in \{1,\dots,p\}} \left(\sum_{i=s(j-1)+1}^{s(j)} c_{k,i} \right). \tag{3.23}$$

Two issues emerge for this model. First, an evaluation of t_s requires reduction of the global maximum load per core over each phase. If there are many phases between two load balancing calls, this will quickly become expensive as global communication increases with each phase. Second, even an optimal solver for (3.23) will not always return a good load balance, as heavily heterogeneous phases are hard to balance. Figure 3.5b displays an example.

Instead, we consider a simplified multi-phase load model that minimizes the maximum accumulated load per core, which is defined as

$$\max_{j \in \{1,\dots,p\}} \sum_{k=1}^{m} \sum_{i=s(j-1)+1}^{s(j)} c_{k,i} = \max_{j \in \{1,\dots,p\}} \sum_{i=s(j-1)+1}^{s(j)} \sum_{k=1}^{m} c_{k,i}. \tag{3.24}$$

This problem reduces to the synchronization-free problem (3.5) again when setting

$$c_i := \sum_{k=1}^{m} c_{k,i}. \tag{3.25}$$

Next, the relation between the problems (3.23) and (3.24) is briefly discussed. For each phase $k \in \{1, \dots, m\}$ and each core $\tilde{j} \in \{1, \dots, p\}$

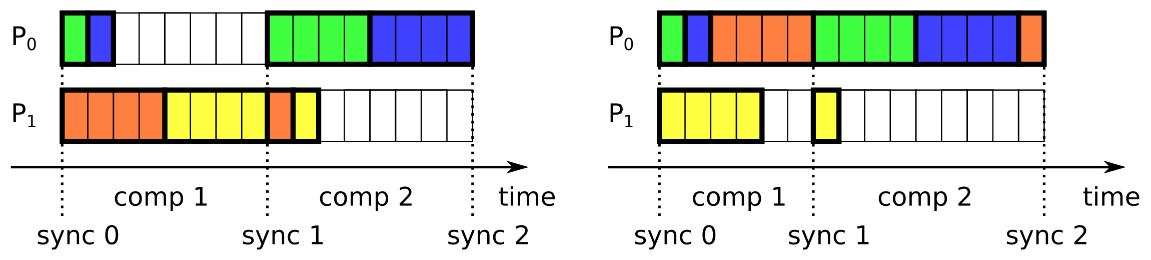
$$\sum_{i=s(\tilde{j}-1)+1}^{s(\tilde{j})} c_{k,i} \le \max_{j \in \{1,\dots,p\}} \sum_{i=s(j-1)+1}^{s(j)} c_{k,i}. \tag{3.26}$$

Summing up over the phases returns for each $\tilde{j} \in \{1, \dots, p\}$

$$\sum_{k=1}^{m} \sum_{i=s(\tilde{j}-1)+1}^{s(\tilde{j})} c_{k,i} \le \sum_{k=1}^{m} \max_{j \in \{1,\dots,p\}} \sum_{i=s(j-1)+1}^{s(j)} c_{k,i}, \tag{3.27}$$

and therefore

$$\max_{j \in \{1,\dots,p\}} \sum_{k=1}^{m} \sum_{i=s(j-1)+1}^{s(j)} c_{k,i} \le \sum_{k=1}^{m} \max_{j \in \{1,\dots,p\}} \sum_{i=s(j-1)+1}^{s(j)} c_{k,i}. \tag{3.28}$$



(a) Simplified load balancing by sam(oa)^2 **(b)** Optimal load balancing

Figure 3.5: Load distribution of four sections to two processes with an intermediate synchronization point. The solution of sam(oa)^2 is compared to an optimal solution. The execution time of a sections is represented by blocks of the same color, blank squares indicate waiting cycles at the corresponding synchronization points. There are two computation phases, each of which is distributed to two cores. **(a)** In sam(oa)^2, the maximum load over the sum of all phases is minimized and a distribution with 12 waiting cycles is returned. **(b)** An optimal distribution correctly minimizes the sum over the maximum load of each phase, where only 10 waiting cycles are needed. The computation takes 20 cycles total, thus even the optimal distribution performs badly.

Thus, any function s that returns a bad distribution in the synchronization-free system (3.24) will return a bad distribution in the synchronized system (3.23). This suggests that solving (3.24) is a suitable heuristic for (3.23). Thus, sam(oa)^2 implements this algorithm for the intermediate synchronization problem.

Minimizing the synchronization-free system (3.24) will return a uniform workload over the sum of all computation phases, which implies that only the *average load* of each core over all phases is balanced. On the one hand, the global communication issue is solved, as only one reduction is required per evaluation. On the other hand, load is balanced worse when approximating (3.23) with (3.24), as shown in (3.28). An example is illustrated in Figure 3.5, where the load distribution obtained by minimizing (3.24) is strongly imbalanced as it requires 12 waiting cycles, while the total computation accumulates to 20 cycles. As mentioned before, the optimal distribution according to (3.23) is not much better, as process P_1 idles for 10 cycles. Hence, even in the optimal case parallel strong scaling efficiency on two cores is limited to 67%. The example nicely illustrates that in general, there is no scalable solution to load balancing with intermediate synchronization.

This result implies that there are only two options: Circumvent the problem or weaken its conditions. Ideally, the problem is avoided completely by reducing and postponing synchronization points, e.g. with a pipelined data model. One example for such a pipelined model is the fused Conjugate Gradients solver that is discussed in Chapter 5.4.1. If intermediate synchronization cannot be omitted, intra-node work stealing mechanisms as described in Section 3.2.5 lower the imbalance. One could also introduce intermediate load balancing steps for time-based cost evaluation. However, they will improve the load balance only if load does not change too frequently, as load prediction always depends on previous measurements. propagated directly and the algorithm terminates.

3.4 Communication Structures for OpenMP and MPI

Communication in parallel frameworks is usually handled by exchanging a ghost boundary layer, which is a single extra layer of elements that surrounds a partition in the simplest case, usually called a *depth*-1 *ghost layer*. By exchanging ghost elements, operators that access adjacent elements retain a black-box view on the grid and assume the existence of neighbor data. The adaptive mesh refinement library p4est [28–30], which is used in the finite element library deal.ii [17] for example, employs a depth-1 ghost layer for communication on an octree-based grid. While the application of a kernel interface for such an approach is simple, the disadvantage of this approach is that the structural overhead for transfer of all edge and vertex data associated with an element may be large.

Thicker ghost layers are possible: firedrake [42] uses a depth-2 ghost layer in its PyOP2 backend [95], which contains two element layers instead of one. The benefit is the possibility of overlapping computation and communication; however, scalability may suffer for computationally cheap problems due to the large memory overhead.

In contrast, Peano [89] uses a depth-0 ghost layer to reduce the memory overhead, especially on higher dimensional grids [126]. In this variant, only the interface between elements is exchanged for communication. Peano additionally bases its communication purely on vertex migration, as exchange of any k-dimensional entities quickly becomes complex and expensive with growing k.

Due to the decision of prohibiting random access to vertex and edge data in sam(oa)^2, communication of element data is complicated on a ghost layer with depth ≥ 1. Thus, with memory overhead in mind, the canonical choice for sam(oa)^2 is a depth-0 ghost layer. A purely vertex-based communication as in Peano is not necessary though, as grids in sam(oa)^2 are always two-dimensional. Furthermore, edge data may be exchanged in addition to vertex data.

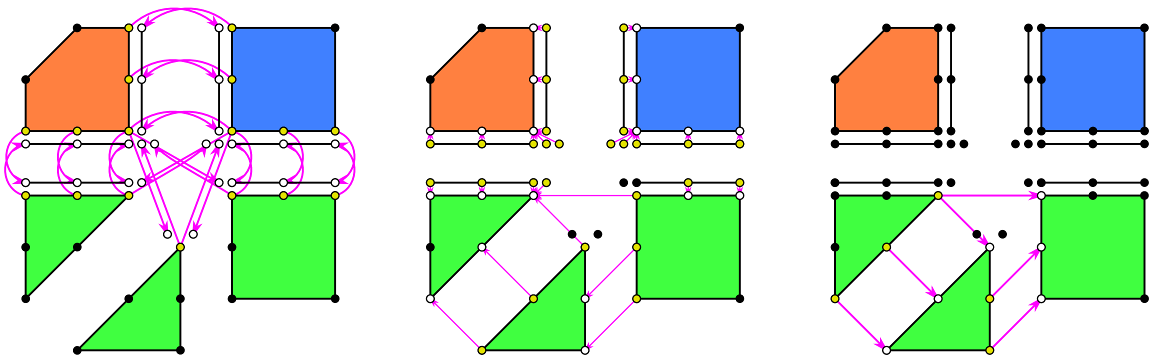
To be able to communicate boundary data, each boundary vertex/edge must know its respective communication sections in sam(oa)^2. Bounded by the vertex degree, up to 8 sections may share a single vertex, hence direct storage of communication information with the vertex/edge data is not feasible. For efficient access, communication information is compressed in a run-length encoded list, as suggested in [111]. First, we discuss how communication lists are used to exchange boundary data. Afterwards, the incremental generation of these lists is presented.

3.4.1 Mixed Peer to Peer and Master Slave Communication

Consider $k > 1$ sections that share one boundary vertex. Then, we assume that each copy owns partial data, which must be merged into a complete state by some accumulative operator. For example, this data could consist of partial residual updates, net updates from a flux solver, or boolean values that indicate whether a vertex is in a fluid obstacle. Once the partial data is fully accumulated, the complete data should be distributed to all copies of the vertex.

There are a couple of ways to exchange the data located on each copy of the vertex. In Figure 3.6a, a peer-to-peer communication pattern is used that issues k^2 messages and correspondingly requires k^2 storage slots for ingoing messages. The advantage is that communication is finished within a single iteration. Figure 3.6b and Figure 3.6c show a master-slave communication pattern where $2\,(k-1)$ messages are sent in total. Collecting, merging and broadcasting messages in a master section requires two communication iterations in total. Hence, the first option is better if communication is latency-bound and the second option is better if communication is volume-bound. Therefore, sam(oa)^2 applies the peer-to-peer method for MPI communication, which typically involves large latencies. The master-slave method is used for intra-process communication. OpenMP threads communicate implicitly via shared memory, which is more suitable for the master-slave method as thread synchronization has very little latency.

Together, a three-step communication scheme results. In the first step, the process boundary vertices/edges are sent as soon as a section is processed in order to overlap some communication with computation. We estimate the resulting execution time t_n for n sections by defining c_i and s_i as computation time and asynchronous communication time of a section $i \in \{1, \ldots, n\}$. Execution time is the elapsed time until all computation is done and all communication data has arrived at neighbor sections. The MPI communication latency is denoted by λ. With a proof by induction, one can show that in a



(a) MPI peer-to-peer exchange **(b)** Intra-process accumulation **(c)** Intra-process broadcast

Figure 3.6: Three-step communication scheme in sam(oa)^2. The first step is to exchange boundary vertices and edges in MPI between distinctly colored sections. In the second step, a designated owner vertex/edge collects updates from all its duplicates and the ghost vertices/edges. Finally, in the third step, updates are broadcast to the copies again. Yellow colored vertices are communication sources and white colored vertices are destinations.

peer-to-peer topology

$$t_n = \max_{k \in \{1,\ldots,n\}} \left(\sum_{i=1}^{k} c_i + \sum_{i=k}^{n} s_i \right) + \lambda \tag{3.29}$$

As the boundary data of the last section cannot be sent as long as its computation phase is not finished, latency cannot be hidden by the scheme. However, if $n > 1$ and $c_{i+1} > s_i$ for each $i < n$, most of the asynchronous communication will be hidden during computation. Then,

$$t_n = \left(\sum_{i=1}^{n} c_i \right) + s_n + \lambda, \quad \text{if } c_{i+1} > s_i \text{ for each } i \in \{1, \ldots, n-1\}. \tag{3.30}$$

Hence, there is some overlapping of communication and computation, but latency is still visible. The amount of visible, asynchronous communication s_n can be reduced by increasing the ratio of sections per core, which we already introduced as a command line parameter in sam(oa)^2.

After all sections are processed, the designated owner of each vertex merges the partial data from all process neighbors and local section neighbors. Finally, the data is distributed again to the local neighbor sections to establish a consistent data view again among sections.

The problem of finding an owner for each boundary entity is not trivial as each section should be able to decide independently if it is the owner of an entity in order to avoid unnecessary logic and communication. A simple, parallel scheme would decide ownership based on which participating section has the lowest index. On the one hand, the section with the lowest index would end up with the most work, as it would have to merge all its boundary data. On the other hand, the section with the highest index has nothing to do. To achieve a better balance, sam(oa)^2 assigns only the outer half of the entities shared by two sections to the section with the lower index, the inner half is assigned to the section with the higher index. Assume that a pair of sections shares four vertices, then the first and the last vertex are owned by the section with the lower index. The second and third vertex are owned by the section with the higher index. By assigning the first and the last vertex to the same section we avoid contradictions and ensure that all sections assign identical entities to the same section.

3.4.2 Incremental Generation of Sections and Communication Structures

Setup and initial distribution of the grid are handled in sam(oa)^2 by its adaptive mesh refinement and load balancing functionalities. The grid is incrementally refined and distributed, which eliminates the need for a heavy-weight partitioning scheme. Hence, the execution of a scenario is divided into two phases. An initialization phase sets up a grid with an initial condition for a system of partial differential equations. Afterwards, a time step phase implements the time integration scheme.

At the beginning of the initialization phase, the grid is a square that consists of two triangles in Sierpinski order. Step-by-step, the grid is successively refined and balanced, where more and more sections are generated and distributed to MPI ranks and OpenMP threads until a scenario-dependent local refinement criterion is met. If the requested degree of parallelism is not reached at this point, the problem is considered too small and it will run on a lower number of cores. Afterwards, the time integration phase follows, where the locally refined and partitioned grid is used for further computation.

This process is also suitable to build communication structures without the need of global communication. As stated before, run-length encoded lists are used for this purpose [111]. For the initial pair of triangles, a single list with a single entry is created that marks all boundary entities as part of the domain boundary. Consider a grid, for which valid communication lists exists. Then the only two components of sam(oa)^2, which can invalidate the lists, are MPI load balancing and remeshing. Hence, the lists must be updated only in these two components.

Update of Communication Structures during MPI Load Balancing

Only small changes are necessary to the communication structures during MPI load balancing. Each section uses the existing communication list to send its new location to all its local neighbor sections and its process neighbor sections, whether it is migrated or not. When the neighbor sections receive the information, both parties change their communication structures accordingly and the process is complete.

Update of Communication Structures during Remeshing

When sections are locally repartitioned during adaptive mesh refinement, they are rebuilt from scratch. Hence, the same action is necessary for the communication structures of each section. However, we exploit the knowledge that MPI communication does not change topologically during local repartioning. The algorithm additionally requires the existence of the distance of each boundary vertex to the root of the communication tree. To understand the origin of this number, Figure 3.7 illustrates how the Sierpinski curve splits the set of boundary vertices and edges into a red and a green tree. Each red/green boundary vertex stores its *vertex distance*. *Distance* in this context denotes the length of the shortest path to the red/green root vertex in the red/green communication tree. To allow integer computation, edge lengths are defined in the Manhattan metric. For simplicity, the crossed edges are considered part of the red communication tree in sam(oa)^2.

In addition to vertex distances, we define separately for the red and green tree the red/green *section distance* as the minimum over the distances of each red/green boundary vertex in the section. An algorithm that finds all communicating section pairs based on this information is described in Algorithm 3.3. It receives the list of section distances and the section distances of all process neighbors as input for a single color. For each section, a *search distance* is defined, which is $+\infty$ initially. Then, each section distance in the list is compared to the search distance in a radial search pattern. When a section with a distance less than or equal to the search distance is found, the matching section is added to the list of neighbor sections and the search distance is decreased to the distance of the matching section. Once the search distance is lower than the section distance, all neighbor sections have been found for the specific section and the algorithm proceeds with the next section.

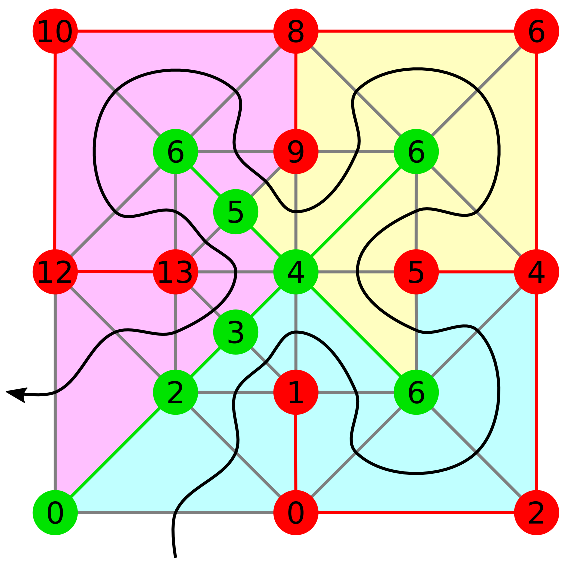


Figure 3.7: Setup of red and green communication trees. Each boundary vertex in the communication tree stores a *distance* value, which is defined as the Manhattan length of the path to the root node. For each section, the red/green section distance is the minimum of the red/green vertex distances. Hence, red/green section distances are $(0,0)$ for the teal section, $(4,4)$ for the yellow section, and $(8,0)$ for the green section. Gray edges are crossed by the Sierpinski curve, which is represented by the black line. Crossed edges can be treated independently, as they are always exchanged between consecutive sections in Sierpinski order.

The idea is that distances are equivalent to the stack index of a vertex in a sequential traversal. Hence, if two sections have a boundary vertex with a given stack index and no other section in between has a vertex with a lower stack index, then both vertices must be identical. Time complexity of the algorithm depends on the average number of neighbor processes x for each process. We estimate the number by modeling the communication topology as a planar, bounded-degree graph $G = (V, E)$, where the faces of the graph correspond to partitions in the grid. An edge corresponds to an interface between exactly two partitions, and a vertex corresponds to interfaces between three or more partitions. There is a communication interface for each pair of processes that shares an edge or a vertex j, hence there is 1 interface per edge and there are $\deg(j)$ *choose* 2 interfaces per vertex, where $\deg(j)$ denotes the degree of the vertex i. Therefore, the total number of communication interfaces i is

$$i \le e + \deg(G)^2 v = e + \deg(G)^2(e + 1 - f) \le (\tfrac{3}{2}f + \tfrac{3}{2}) + \deg(G)^2(\tfrac{1}{2}f + \tfrac{3}{2} + 1) \tag{3.31}$$

where $e = |E|$, $v = |V|$ and $\deg(V)$ is the maximum vertex degree in the graph. The average number of neighbor processes per process is therefore

$$x = \frac{i}{f} = \tfrac{3}{2} + \tfrac{1}{2}\deg(G)^2 + o(1). \tag{3.32}$$

Hence, if the algorithm is executed in parallel, the average time complexity is

$$\mathcal{O}\left(\frac{n}{p}\left(\frac{n}{p} + \frac{n}{p}x\right)\right) = O\left(\frac{n^2}{p^2}\right). \tag{3.33}$$

As $\frac{n}{p}$ is a constant, the algorithm runs in constant time on average. Note that the worst case complexity is linear in n, as a section can theoretically degenerate to a state where it shares interfaces with all other sections.

3.5 Conclusion: A Configurable Parallelization Toolkit

In this chapter, a configurable parallelization toolkit based on splitting a grid into Sierpinski sections was presented. Corresponding extensions to the kernel interface that allow flexibility in the design of parallel algorithms were added. A new load balancing approach was implemented, suitable for homogeneous and heterogeneous kernels while assuming a strict black-box view of computation kernels. Communication is optimized for hybrid parallelization, which reduces redundant computation within a process to the necessary minimum and allows some overlapping between computation and communication in distributed memory.

While many use cases are covered by the methods presented here, there are some limitations: Load balancing with intermediate synchronization points does not scale, which is problematic if a scenario contains multiple phases with heterogeneous load. A hierarchy of increasingly powerful and expensive load balancing algorithms that reflects the network topology could be built to mitigate the problem. Depending on the imbalance, load would be propagated either within processes, nodes, islands or full systems – as it is already done for shared and distributed memory. Still, the problem of accurately predicting load of heterogeneous phases would have to be solved. Currently this is not possible, as time-based cost evaluation can only extrapolate previous measurements.

Furthermore, sam(oa)^2 has little influence on the size of the partition boundaries, which may become arbitrarily large for any type of space-filling curve partitioning. There is no obvious solution, as even coarse grain partitioning cannot prevent the problem. In practice however, the worst case is highly unlikely as it is purely artificial.

Finally, the algorithms are mainly designed for homogeneous systems that consist of equivalent computation nodes. As heterogeneous systems – e.g. the SuperMIC system [119], a mixed Intel Ivy Bridge

Algorithm 3.3: Neighbor section search in sam(oa)^2, that compares vertex distances to find all adjacent sections. The algorithm is executed once for the red and once for the green communication tree. It receives the number of sections per process $\frac{n}{p}$, the lowest possible neighbor section index a, the highest possible neighbor section index b, and the neighbor section distances $\{d_a, \ldots, d_b\}$ as input. The output is a list of adjacent sections S_i for each local section $i = 1, \ldots \frac{n}{p}$.

```
Input: $\frac{n}{p}$, $a$, $b$, $\{d_a, \ldots, d_b\}$.
Output: $(S_i)_{i=1,\ldots,\frac{n}{p}}$.

// Define domain boundary distances
$d_a \leftarrow -\infty$ ;
$d_b \leftarrow -\infty$ ;

// Incrementally compare pairs of sections for distance overlaps
for $i \leftarrow 1$ to $\frac{n}{p}$ do
    $s \leftarrow +\infty$ ;
    for $j \leftarrow i - 1$ to $a$ step $-1$ do
        if $d_j \leq s$ then
            appendLeft $(S_i, j)$ ;
            $s \leftarrow d_j$ ;
            if $s < d_i$ then break ;
        end
    end

    $s \leftarrow +\infty$ ;
    for $j \leftarrow i + 1$ to $b$ do
        if $d_j \leq s$ then
            appendRight $(S_i, j)$ ;
            $s \leftarrow d_j$ ;
            if $s < d_i$ then break ;
        end
    end
end
```

and Intel Xeon Phi cluster – will become more and more prevalent in the future, high-performance software take process-dependent hardware capabilities into account. Hence, a short outlook with focus on load balancing of processes with heterogeneous execution rates is given in Appendix A. However, the theory is not considered central to the thesis, as scalability tests will be conducted purely on homogeneous high-performance clusters.

4 Generic Traversals for Adaptive Mesh Refinement

This chapter dicusses first, how to design an interface for finite element and finite volume methods that supports adaptive mesh refinement and parallelization. Second, an implementation of an interface based on Sierpinski curve traversals is presented.

For most interface components, well-established solutions already exist. Firedrake [42, 95] and Dolfin [71] are compilers for the domain-specific language UFL (Unified Form Language [3]) and part of the FENICs project [2,41,70]. Essentially, both projects are finite element frameworks for the solution of partial differential equations. Based on a problem definition in weak formulations, they allow translation of a high-level, mathematical description into a mesh discretization by automatic differentiation. In Dolfin, adaptive mesh refinement is supported by an automated generation of refinement indicators and interpolation/restriction methods [70]. However, the implementation is restricted to finite element methods.

The PDELab [18, 19, 40] extension of DUNE (Distributed Unified Numerics Environment [39]) chooses a residual-based formulation that separates grid iterations into three logical stages. A volume operator executes element-local computations, such as the evaluation of volume integrals. Afterwards, a skeleton operator iterates over the interfaces between each adjacent element pair. The operator has read and write access to both elements and is intended e.g. for flux computations in finite volume and discontinuous Galerkin methods. Third, a post-skeleton volume operator is called. Its functionality is identical to the volume operator. For example, it is used to apply element state updates after accumulation of interface fluxes. With support for a large collection of local function spaces, the interface offers great flexibility in design of finite element, finite volume and discontinuous Galerkin solvers. Adaptive mesh refinement is currently not provided.

The interface of Peano [89, 126] complies strongly with an event-based view on grid traversals, where iterators over vertices are realized by a *first-touch* and a *last-touch* policy in loops over elements. We define that a vertex is *touched* if it is adjacent to an element that is iterated in the loop. Hence, the *first touch* of a vertex is the first iteration, where a vertex is touched in the element loop, and analogously for *last touches*. Whenever a first touch or a last touch occurs, a corresponding first-touch operator or last-touch operator is invoked on the vertex. This design guarantees that operations on elements are always called after the first-touch operators and before the last-touch operators. Thus, multi-stage algorithms with data dependencies may be implemented in a single grid traversal.

Despite all its advantages, the UFL is not a suitable candidate for sam(oa)^2, as the main motivation for the interface should be flexibility. Support for adaptive mesh refinement is only provided for finite element methods in the language, which does not suffice for our purpose. However, the support of domain-specific languages is a possible future extension.

Due to their similar design principles, sam(oa)^2 adopts the finite element interface of Peano, supplying first-touch and last-touch operators for component-wise updates. Though, sam(oa)^2 defines them on all grid entities, which include cells, edges, and vertices. An *element operator* communicates between cell-local grid entities during the grid traversal.

To support finite volume solvers, sam(oa)^2 provides *skeleton operators* based on the design in DUNE

PDELab, but with different access rules. While the skeleton operator in PDELab allows direct manipulation of cell data, sam(oa)^2 permits read and write access only to the interface in the operator. This means that the input and output of the operator must be stored locally on each interface, which is necessary due to the constraints imposed by the stack-&-stream approach.

Any grid data in sam(oa)^2 may be accessed only via grid traversals, which are loops over all cells in Sierpinski order. Edge and vertex data is accessible inside of a traversal. Furthermore, adaptive mesh refinement is supported, but as topological changes to the grid cause instruction and memory overhead, sam(oa)^2 separates between implementations of *static grid traversals* and *adaptive grid traversals*. Static traversals may touch and change any data located on the grid, but do not issue topological changes during a traversal. Adaptive traversals explicitly offer an interface for refinement and coarsening of grid data. For example, linear solver iterations or file output are realized by static traversals; interpolation and restriction of unknowns by adaptive traversals. This classification into two traversal types allows sam(oa)^2 to reduce complexity and apply traversal-specific optimizations. This chapter focuses on the interface design of both traversal types and the underlying implementations.

4.1 Kernel Concept for Static Grid Traversals

In order to provide flexibility for applications, sam(oa)^2 offers finite element and finite volume interfaces for static grid traversals [76]. Applications are not restricted to a single interface type, both may be used during one program execution and even at the same time in a single grid traversal.

The design of the framework is event-based. Hence, the user implements kernels in callback functions that are invoked by the program core for static grid traversals. An overview of the available set of operators for static traversals is illustrated in Figure 4.1. Using a template implementation, realized by preprocessor macros, the grid layer implements a grid iterator based on Sierpinski traversals. For each *element* in the grid, the stack-&-stream automaton reads and writes cell, edge, and vertex data. In sam(oa)^2, an element is defined as the union of a cell, its three adjacent edges, and the corresponding vertices. As stack and stream sizes do not change in static traversals, there is no need to separate between input and output streams. Thus, sam(oa)^2 uses the same stream for data input and output, reducing memory requirements and improving the access speed due to improved caching behavior.

Applications may access any data in the element, but stencils that reach further than a single element are prohibited. The reason for this restriction is that these accesses would violates the rule of accessing data consecutively in the stack-&-stream system. Hence, the interface defined in the hook layer enforces these local data visibility rules. At the same time, the hook layer is designed to offer flexibility to applications that implement algorithms based on grid traversals in sam(oa)^2. Above the hook layer there is a kernel layer that implements numerical operators and other application-specific functionality, which is independent of the grid. Some examples for implementations of kernels in sam(oa)^2 will be presented in Chapter 5 and Chapter 8. Going into detail, the finite element interface is investigated first, and afterwards the finite volume interface is discussed.

4.1.1 Finite Element Interface

As mentioned earlier, sam(oa)^2 provides a finite element interface that is constructed similarly to the one in Peano [89]. Algorithm 4.1 shows the logical execution order of finite element operators in a single grid traversal. The first-touch and last-touch operators are called once for each entity in the grid in the beginning and the end of the traversal, respectively. The operators perform component-wise manipulations of vector data located on the entities, such as initialization of degrees of freedom in a *first DoF kernel* or application of an update in a *last DoF kernel*. In Peano, entities correspond to vertices, but sam(oa)^2 extends the definition to any cell, edge, or vertex in the grid to allow more flexibility in the

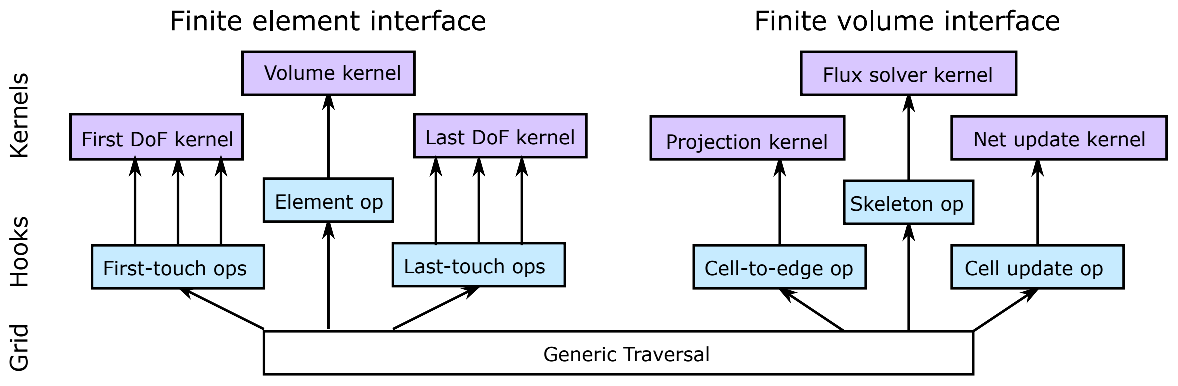


Figure 4.1: Overview of the kernel design for static grid traversals. In an event-based pattern, a finite element and finite volume interface provide sets of operators that are invoked from the grid layer. Application-dependent kernels are implemented on top of the operators.

design of algorithms. Between the first-touch and the last-touch operators an element operator is called that allows manipulation of element data. The operator is fairly powerful as it offers read and write access to the full element. It may be used to implement a matrix-vector product in a *volume kernel* based on element matrices for example. *Pre-traversal* and *post-traversal* operators are intended for scalar operators on global variables such as the computation of a global time step or an update step for a linear solver.

Algorithm 4.1: Finite element interface of sam(oa)^2. The first-touch and last-touch operators iterate over all entities (cells, edges, and vertices) in the grid, allowing execution of component-wise operations on vectors. An element operator has read and write access each cell and its adajcent edges and vertices. Pre- and post-traversal operators perform scalar updates on global variables.

```
Input: grid
Output: grid
call PreTraversalOp(grid);
foreach entity in grid do  call FirstTouchOp(entity) ;
foreach element in grid do  call ElementOp(element) ;
foreach entity in grid do  call LastTouchOp(entity) ;
call PostTraversalOp(grid);
```

4.1.2 Finite Volume Interface

The finite volume interface of sam(oa)^2 provides additional functionality to the finite element interface by offering methods specifically intended for cell-centered finite volume methods and discontinuous Galerkin schemes.

In the logical execution order of a traversal as described in Algorithm 4.2, a cell-to-edge operator is called first on each combination of adjacent cells and edges. Inside the operator, a cell representation of some kind is determined and stored on the edge. The skeleton operator then computes two cell updates from the representations and stores them in the edge. Afterwards, the cell-update operator applies the computed net updates to the cell. For classical finite volumes, the cell-to-edge operator would call a *projection kernel* that defines a cell representation simply as a copy of the cell state. The skeleton operator invokes a *flux solver kernel* is called which computes net updates from the cell states. Finally, the cell-update operator calls a *net update kernel* that accumulates the net updates in each cell to advance the cell state in time.

Algorithm 4.2: Finite volume interface of sam(oa)^2. The cell-to-edge operator is called first for each cell and each edge that are adjacent to each other. A cell representation is created and stored on the edge. The skeleton operator iterates over all edges and compute pairs of cell updates from the representations of both neighbor cells. Finally, these updates are applied to each cell in the cell-update operator.

```
Input: grid
Output: grid
foreach element in grid do
    foreach edge in element do
        call CellToEdgeOp(cell of element, edge);
    end
end
foreach edge in grid do  call SkeletonOp(edge) ;
foreach element in grid do
    call CellUpdateOp(cell of element, edges of element);
end
```

4.2 Kernel Concept for Adaptive Mesh Refinement

Adaptive mesh refinement is supported by sam(oa)^2 in a dedicated traversal type where the current grid is discarded and replaced by a new grid with a different topology. An overview of the corresponding interface is given in Figure 4.2. Requests for topology changes are issued in previous grid traversals by setting a flag in each cell, signaling that it should be refined, coarsened, or passed unaltered to the new grid.

When a new grid with the requested topology adjustments has been created, a traversal is started that processes both old and new grid synchronously. Algorithm 4.3 shows the execution order of operations for adaptive traversals. First-touch and last-touch operators are supported only on the destination grid, as the source grid is discarded after traversal. The refinement, coarsening, and transfer operators provide functionality for passing data from the source grid to the destination grid, while applying numerical operators. Each operator receives a single element from the source grid, a single element from the destination grid, and their correlation as input. The correlation is given by a refinement path, that is encoded in a tuple of integers, which indicate whether an element is the first or the second child of the previous element. Consider an element that is refined twice, where the destination element is the second child of the first child of the source element. Then the path argument is the tuple $(1, 2)$. With this method, an expensive change of basis is avoided and simpler schemes that exploit the geometrical structure of the grid may be used.

4.3 Extension of the Kernel Interface for Parallelization

Following the stack-and-stream approach, single sections are processed sequentially in Sierpinski order. Multiple sections execute *local operations* in parallel, which are defined as operations without overlap-

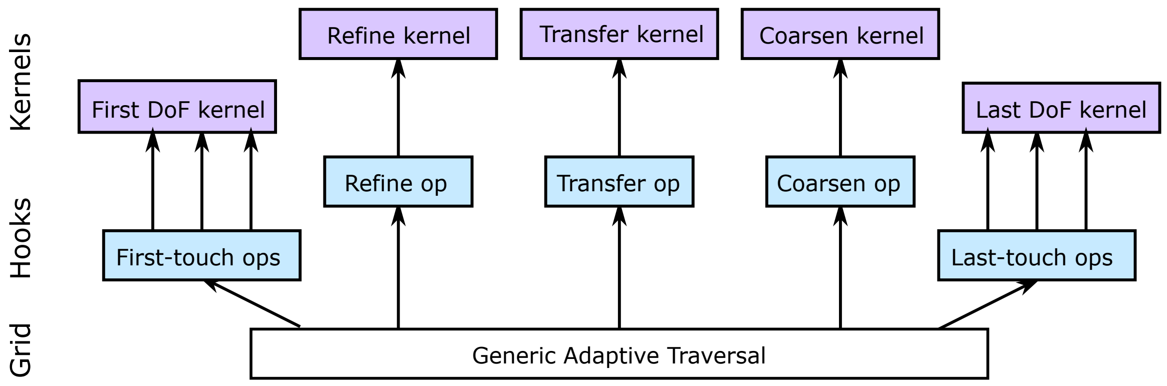


Figure 4.2: Overview of the kernel design for adaptive grid traversals. Similar to static traversals, sam(oa)^2 provides a set of operators for refinement, coarsening, and transfer, i.e. copying, of element data from an old grid to a new grid.

Algorithm 4.3: Adaptive traversal interface of sam(oa)^2. The first-touch and last-touch operators iterate over all entities in the grid, allowing execution of component-wise operations on vectors. An element operator has read and write access to the full element. Pre- and post-traversal operators perform scalar updates on global variables. The function evaluations isChildOf(a, b), isParentOf(a, b), and isEqual(a, b) are true if and only if element b is a child of element a, b is the parent of a, and b is equal to a, respectively.

```
Input: srcGrid
Output: destGrid
foreach entity in destGrid do call FirstTouchOp(entity) ;
foreach srcElement in srcGrid do
    foreach destElement in destGrid do
        if isChildOf(srcElement, destElement) then
            call RefineOp(srcElement, destElement);
        else if isParentOf(srcElement, destElement) then
            call CoarsenOp(srcElement, destElement);
        else if isEqual(srcElement, destElement) then
            call TransferOp(srcElement, destElement);
        end
    end
end
foreach entity in destGrid do call LastTouchOp(entity) ;
```

ping write access. For example, an operation that writes cell data by reading from adjacent operations is local, but an operation that writes from cell data to boundary vertex data is not local.

The finite volume interface as explained in Section 4.1.2 does not need adjustment, as the cell-to-edge, skeleton and cell-update operators follow the rules for local operations. In the finite element interface, the element operator is not local, as it allows write access to adjacent edge and vertex data. To handle this issue, sam(oa)^2 classifies this type of data access as one-sided write access to local copies of edges and vertices. Hence, element operators store incomplete information on boundary edges and vertices, which may be merged with missing data later to form a full update.

To perform this operation, we introduce an additional *merge operator* to the interface that receives any pair of boundary edges or vertices during a communication phase and merges their data. This addition specifically targets residual-based formulations that require accumulation of partial residuals on vertex or edge unknowns. The corresponding extensions to the kernel interface are displayed in Algorithm 4.4.

After merging, the last-touch operator is called on each vertex and each edge. Note that since this operator is executed after the communication phase during grid traversal, the last-touch operator must be called locally on each entity of the grid – especially boundary edges and vertices – to guarantee consistency of data across all sections. See Section 3.1 for the definition of sections.

Hence, in the logical execution order, the last-touch operator will be called multiple times on the same boundary entity. This causes problems if a global accumulation such as the computation of a vector norm is performed, which requires a unique invocation for each logical entity in the grid. For this purpose, an additional *reduce operator* is introduced into the interface. The operator provides only read access to each entity and is guaranteed to be called exactly once per logical entity. As the reduce operator is not allowed to modify data, it cannot lead to inconsistent copies of boundary entities across different sections.

Sections are processed in parallel, which means that access to global data must be restricted during the traversal. Additional operators are included to handle global data. The pre- and post-traversal operator of the grid allow sending global and process-local data such as time step sizes or residual norms back and forth between the grid and its sections. An additional pre- and post-traversal operator is executed concurrently on each section, where section-local variables may be initialized or processed.

Lazy Broadcasts in Shared Memory

Algorithm 4.4 reveals a problem in the communication pattern presented in Section 3.4.1. Communication of boundary data must be finished before the last-touch operator is invoked, as the last-touch operator expects fully accumulated updates in each boundary entity. Hence, first-touch and last-touch operators must be called symmetrically on section boundaries by each section that owns a copy of an entity. This is necessary to keep the grid in a consistent state among all sections. Executing the operators redundantly can cause scalability problems though if the operators happen to be very expensive. Hence, sam(oa)^2 postpones the broadcast phase of the communication in Figure 3.6c until the first-touch operators in the next traversal have been called. The first-touch and last-touch operators are no longer executed on each copy of a boundary entity, but only on sections that are local owners of the entity within a process. On process interfaces, we still have to execute the operators redundantly as it is not possible to postpone broadcasting to other processes without adding a second MPI communication phase. Hence, redundant calls to the operators are eliminated within processes, which is particularly beneficial for hybrid OpenMP+MPI parallelization.

Algorithm 4.4: Parallel extension of the finite element interface in Algorithm 4.1. The new operators are marked in red color. Pre- and post-traversal operators on the grid allow passing global and process-local data between grid and sections, pre- and post traversal on the section allows concurrent initialization and processing of section-local data. The merge operator is a communication operator for accumulating data on the interface between adjacent sections.

```
Input: grid
Output: grid
call PreTraversalOp(grid);
foreach section in grid do
    call PreTraversalOp(section);
    foreach entity in section do  call FirstTouchOp(entity) ;
    foreach element in section do  call ElementOp(element) ;
    // Exchange boundary data here
    foreach locally_owned_boundary_entity in section do
        call MergeOp(locally_owned_boundary_entity);
    end
    foreach entity in section do  call LastTouchOp(entity) ;
    call PostTraversalOp(section);
end
foreach globally_owned_entity in grid do
    call ReduceOp(globally_owned_entity);
end
call PostTraversalOp(grid);
```

4.4 Implementation of Static Grid Traversals

While the original implementation of grid traversals was based on recursive traversal of a refinement tree [9], it was soon discovered that significant speedup could be achieved by dropping the recursive traversal of the tree. Instead, a loop-based algorithm was applied [125] that treats the grid similar to triangle strips in computer graphics [87, 124] and iterates over a list of edge-connected elements. The function call overhead is reduced and compiler optimizations for loops are made possible, which otherwise would be prevented by the code structure. Furthermore *crossed edges*, i.e. edges that are crossed by the Sierpinski curve, are stored in a dedicated stream without requiring stack access to reduce the amount of memory operations. Eventually, the refinement tree is only required for adaptive refinement and coarsening. This optimization causes a significant performance improvement for memory-bound problems [10].

4.4.1 Stack and Stream Access Pattern

In sam(oa)^2, grid traversals are realized by the loop implementation presented in [125]. A disadvantage of this method is that all structural information must be stored explicitly in the cell data. For each element, the grid iterator must decide how much cell, edge, and vertex data is located on the streams and the stacks, which data is assigned to which entity, and how the entities are oriented and positioned in the grid. In contrast, a recursive traversal can pass this information through the refinement tree.

To solve the problems of data location and data assignment, sam(oa)^2 uses a *turtle grammar* for classification of elements as depicted in Figure 4.3 and described e.g. in [8]. Data on cells and the crossed edges p and n are always accessed from streams, independent of the triangle type. Hence, cells store structural information that contains the turtle grammar type of the cell. As described in [125], the color of the *color edge* c must be known to determine if c and its adjacent *color nodes* c_{in} and c_{out} are

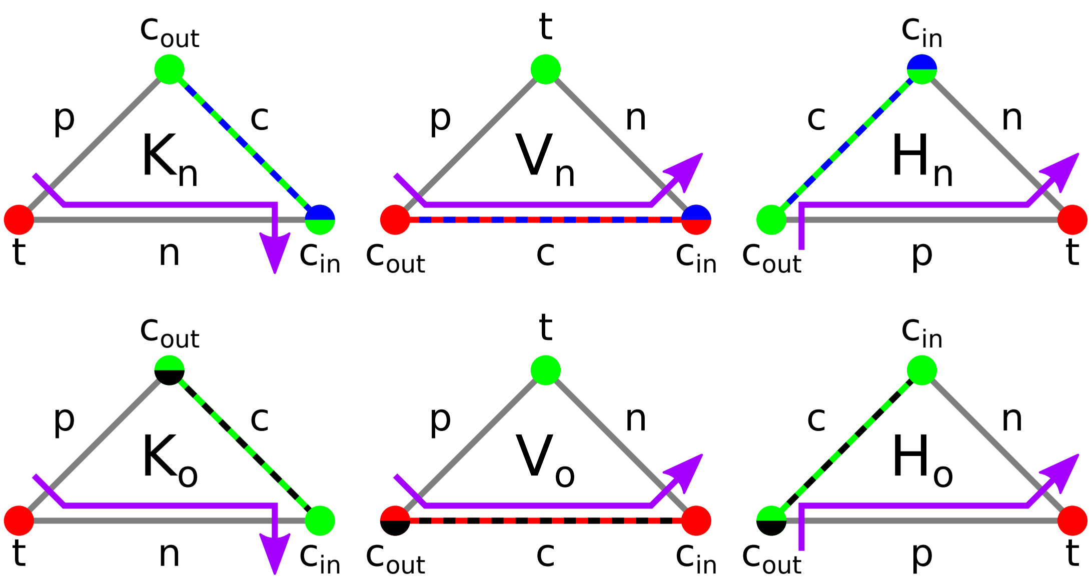


Figure 4.3: Turtle grammar for the classification of elements in the grid. The triangle type defines for each entity, whether it is accessed on a stream, on the red stack, or on the green stack. Both the previous and the next crossed edge p and n are always accessed from a dedicated crossed edge stream. Color edges and vertices are read from the input stream if they are marked in blue, and written to the output stream if they are marked in black. In all other cases, they are accessed on the stack. Note that the classification into the types K, V, and H is only relevant to the stack-&-stream system in order to determine the color of the color edge. The actual purpose is to define the orientation of each element relative to its predecessor in the Sierpinski order.

located on the red or green stack/stream. To decide whether a first touch or a last touch of an entity occurs entity, an additional classification into *old elements* and *new elements* is necessary. On the first touch, an entity is read from the stream and on the last touch it is written to the output stream. At any other time, it is accessed on the stack.

Hence, for each new element, c and the *input color node* c_{in} are touched for the first time and the *output color node* c_{out} is neither touched for the first nor last time. In old elements, c_{out} and c are touched for the last time, whereas c_{in} is touched neither for the first nor last time. The *transfer node* t is never touched for the first nor last time. To avoid branch evaluation during runtime for each case, sam(oa)^2 ensures that the most frequent triangle types are implemented in a template pattern. Hence, a single switch-statement per element is necessary to define the stream and stack access patterns.

The geometric correlation between c_{in}, c_{out}, t, the two leg vertices and the hypotenuse vertex is encoded in the turtle grammar in Figure 4.3. To determine the element orientation in space, an additional integer between 1 and 8 is stored to mark the 8 different orientations in 2D space where each orientation with consecutive numbers is obtained through a rotation by $45°$. We refer to this classification as *plotter grammar* [8]. Additionally, sam(oa)^2 distinguishes between forward and backward triangles, where the reverse orientation is indicated by a negative index between -8 and -1. Positions are tracked by incrementing the coordinates of the previous element in the traversal with the element orientation.

4.4.2 Additions for Temporary Data and Boundary Treatment

For the purpose of communication and treatment of boundary conditions, boundary data is stored in its own stream to provide a stride-free view that is used as a memory buffer for communication. Hence, the traversal automaton stores another flag in each cell to mark boundary elements. In boundary elements, the color edge and the two color vertices are classified as boundary entities that are read from or written to boundary streams.

In addition to persistent data, sam(oa)^2 also allows storage of light-weight temporary data for each vertex that exists only for the duration of a single traversal. This is realized for color edge and color vertices by simply adding the temporary data to the stack.

4.4.3 Data Access Optimizations

In our implementation described in [76], temporary cell, edge, and vertex data is stored in small ring buffers, as illustrated in Figure 4.4, where temporary data is cached until it is discarded. The problem of the ring buffer approach is the large amount of memory operations required to shift data back and forth. Vertex data in particular must be copied at least twice for each element traversal, namely into and out of the vertex buffer. To reduce the amount of memory traffic, sam(oa)^2 discarded vertex buffers and accesses vertex data directly on the stacks. Actual memory operations occur only when data is read from input streams or written to output streams, while all access on the stacks is performed in-place. Similarly, all color edge data is accessed directly on the stacks. Temporary cell and crossed edge data is still stored in ring buffers though.

To improve the cache efficiency, input and output streams share the same memory location in static grid traversals. This is a valid approach, as streams are implemented as arrays of fixed size in sam(oa)^2, due to their fast access compared to dynamic data structures. Any entity that is overwritten in the output stream must have already been read from the input stream and is stored either in the stack or in another location of the output stream. Hence, no true data dependencies occur.

4.4.4 Out of Order Execution of Interface Operators

During grid traversals, the actual order of operations in the finite element and finite volume interfaces intermingles calls without following the specified logical order. The reason for this behavior is that the exact location of an entity in memory is only determined during the traversal when cells are processed in Sierpinski order. Whether a grid entity is accessed for the first time, the last time, or in-between is defined by the turtle grammar. Then, the corresponding operators in the hook layer are invoked by the same logic. However, it is ensured that the logical order of operations is only relaxed where data dependencies are not violated by switching the order. Element operators are called as soon as all data of an element has been read and before it will be written back to the stack-&-stream system. Another reason for intermingling operator calls is to minimize the lifetime of temporary data during a traversal. With the existing scheme, a temporary data structure is initialized on the first touch and discarded on the last touch of each entity, requiring storage only on the stacks, which leave a smaller memory footprint than the streams. This memory optimization reduces the storage requirements for many algorithms that have to handle intermediate results. For example, a Jacobi solver for linear equation systems stores an accumulated residual in a temporary variable that is created at the beginning of an iteration and discarded at the end. This data is located purely on stacks during a grid traversal.

4.4.5 Pipelined Implementation of Finite Volume Operators

The reason for persistent storage of cell representations is that the finite volume algorithm as described in Algorithm 4.2 cannot be executed in a single traversal. The problem is that a cell-to-edge operator

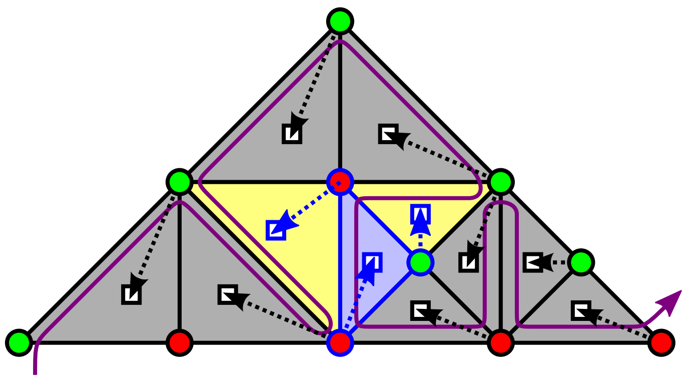


Figure 4.4: Ring buffer storage of vertex data in a Sierpinski grid: The implementation in [76] uses cell-associated vertex buffers, which are represented by white squares in each triangle center, to store temporary vertex data that are initialized on the first touch of a vertex and held in a ring buffer until it is discarded. The blue marked element along with its vertices and edges is currently processed in the traversal, the yellow elements hold vertex data required for the processed element and the gray elements are discarded. As a vertex is shared by up to eight elements, the ring buffer has to contain at least eight entries to avoid data collisions. Vertex buffers are replaced by direct stack access in sam(oa)^2.

can be called earliest when a cell is touched for the first time. Hence, whenever a skeleton operator is called on an edge, both cells adjacent to the edge must have called the cell-to-edge operator for the corresponding cell-edge pair. However, this implies that the iterator must have passed through one of the cells and is located already in the second cell. Therefore, the cell-update operator can be only applied to the second cell, as a return to the first cell is not possible in the same traversal. To solve this issue, either the cell-update operator must be postponed to the next traversal, or the cell-to-edge operator must be called in advance in the previous traversal.

The problem with late cell updates is that they violate data dependencies, as any access to cell data in the post-traversal operator will occur before the cell update and therefore, potentially include incomplete results. In contrast, early cell-to-edge calls only require an early creation of cell representations for the skeleton operator. Hence, if invocation of the skeleton operator is scheduled carefully, no data dependency problems occur.

For sam(oa)^2, we chose early cell-to-edge calls over late cell updates in order to uphold data dependencies. Applications are enforced to invoke the cell-to-edge operator in each traversal to ensure that the cell representations are always available. In a finite volume scheme, redundant calls to the operator are comparably cheap and therefore no major impact on the performance is expected. This implies that adaptive traversals call the cell-to-edge operator, too, to ensure that cell representations are available to the skeleton operator in the next traversal. In order to pass cell representations to the next traversal, a single cell representation for color edges is stored in persistent memory. All other representations and all updates are located in temporary storage. To sum up, by using a pipelining scheme for the cell-to-edge operator, a finite volume time step is implemented in a single grid traversal.

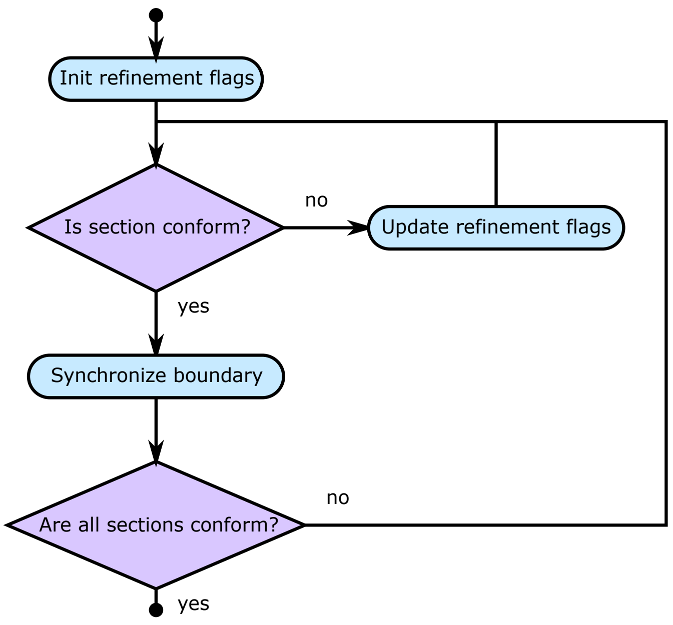


Figure 4.5: Flow chart for conformity traversals. First, each section, as defined in Section 3.1, initializes edge refinement and coarsening flags from the cell flags. Logical operators are iteratively applied in independent loops over each section until a consistent refinement state is achieved for all cells of a section. Edges that are shared between sections are synchronized when all sections are conform. If necessary, the process is repeated again.

4.5 Implementation of Adaptive Grid Traversals

Adaptive mesh refinement in sam(oa)^2 consists of three major components: First, conformity traversals adjust the refinement flags in each cell. Second, a new grid is allocated where stream and stack sizes are computed from their corresponding versions in the old grid and the refinement flags. Third, an adaptive grid traversal is invoked that transfers data between the current grid and the new grid. The three steps will be shortly explained in the following.

4.5.1 Conformity Traversals

In order to avoid the creation of *hanging nodes* during remeshing, the refinement flags in the cell must be adjusted for mesh conformity. Hanging nodes are vertices that split faces only on one side of an edge. Internally, sam(oa)^2 invokes its *conformity traversals* before each adaptive traversal to iterate over the refinement and coarsening flags until a consistent state is achieved. Refinement is performed if some cell requests it, coarsening is allowed if all involved cells permit it. Coarsening of two cells that do not belong to the same section is always forbidden. The scheme is implemented according to [111, 123] and is displayed in Figure 4.5.

4.5.2 Grid Allocation

Once conformity has been established, a new grid is allocated. Memory is reserved for all streams and stacks of each section, which will be filled with data during the adaptive traversal. As explained in Section 3.2.3, sam(oa)^2 balances sections based on a cost model. Hence, the number of elements in each

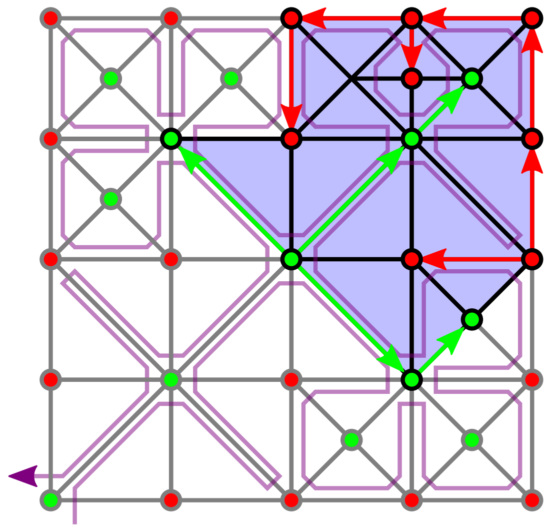


Figure 4.6: Estimating the number of boundary edges by the stack size. The blue elements mark a section, where the red and green edges are located on stacks. Arrows indicate in which direction the stacks are growing. Hence, the green stack contains at most 2 edges and the red stack contains at most 6 entries. Now, (4.1) implies that there are at most 18 boundary edges in the section, compared to 10 actual boundary edges.

section after adaptive refinement and local repartitioning should be determined from its intended cost. Algorithm 4.5 computes for each section the number of elements it receives after remeshing. If section splitting is enabled, the entire grid is cut into parts of uniform cost. Otherwise, if atomic sections are used, each partition is cut locally. To find the correct number of total elements in a section, the algorithm receives the post-refinement number of cells for each section as argument. This value is computed by updating the current number of cells with the refinement flags in each cell. The output of the algorithm is a queue that contains the number of cells in each destination section after the refinement and local repartitioning.

For a single destination section with f elements, estimating the size of the remaining data structure is fairly simple. In sam(oa)^2, the stack sizes s_{red} and s_{green} are tracked for both colors along with the conformity corrections. Using this information, the number of section boundary edges e_b is estimated by

$$e_b \leq 2\ (s_{\text{red}} + s_{\text{green}}) + 2. \tag{4.1}$$

This approximation is based on the idea that except for the two crossed edges on the boundary, all boundary edges correspond to color edges in the entire grid. An edge is a section boundary edge if and only if it is touched by exactly one element of the section, see Figure 4.6 for an illustration. Hence, if we traverse the full grid and look only at the stack-&-stream operations performed on the elements of the section, then all red section boundary edges must be either located already on the red stack before the traversal, or end up on the stack after traversal. Therefore, the size of the red boundary is at most twice the size of the red stack, as the stack is at most full before and after traversal and at least empty in-between. The same holds for the green boundary.

Every cell is adjacent to three edges and every edge is adjacent to two cells, except for the boundary edges, which are only adjacent to a single cell. Hence, the total number of cell-edge pairs must

Algorithm 4.5: Computing the number of cells that each section receives after local repartitioning. The input parameters are the number of sections n in the source grid, the total number of cores p, the number of sections per core α, the post-refinement element count of each source section f_i and the cost of each section c_i. The output is a queue $\hat{F}$ that contains the number of elements of each destination section.

```
Input: $n$, $p$, $\alpha$, $(c_i)_{i \in \{1,\dots,n\}}$, $(f_i)_{i \in \{1,\dots,n\}}$
Output: $\hat{F}$
$D \leftarrow$ queue();
for $i \in \{0, \alpha\}$ do
    $C_i \leftarrow \sum_{k=1}^{i} c_k$;
end
// Cut the total cost into partitions for destination sections.
if isSectionSplittingEnabled then
    // Section splitting:  cut global cost into uniform pieces.
    $C_{\text{process}} \leftarrow$ MPI_Scan($C_\alpha$);
    $C_{\text{total}} \leftarrow$ MPI_Allreduce($C_\alpha$);
    for $i = 1$ to $\alpha p$ do
        $d \leftarrow \frac{i}{\alpha p} C_{\text{total}} - C_{\text{process}} + C_\alpha$;
        if $d > 0$ and $d < C_\alpha$ then
            enqueue($D$, $d$);
        end
    end
    enqueue($D$, $C_\alpha$) ;
else
    // Atomic sections:  cut local cost into uniform pieces.
    for $i = 1$ to $\alpha$ do
        $d \leftarrow \frac{i}{\alpha} C_\alpha$;
        enqueue($D$, $d$);
    end
end

// Translate the cost partitions into numbers of elements.
$i \leftarrow 1$;
$\hat{F} \leftarrow$ queue();
$d_{\text{prev}} \leftarrow 0$;
while not isEmpty(D) do
    $d \leftarrow$ dequeue($D$);
    $\hat{f} \leftarrow 0$;
    // The number of elements in a destination
    // section is obtained by counting the
    // elements in the destination cost partition.
    while $d > C_{i-1}$ do
        $\hat{f} \leftarrow \hat{f} + |[d_{\text{prev}}, d] \cap [C_{i-1}, C_i]| \; \frac{f_i}{c_i}$;
        $i \leftarrow i + 1$
    end
    enqueue($\hat{F}$, $\hat{f}$);
    $d_{\text{prev}} \leftarrow d$;
end
```

be $3f = 2e - e_b$, where e denotes the total number of edges in the section. Therefore, we obtain

$$e = \tfrac{3}{2}f + \tfrac{1}{2}e_b, \tag{4.2}$$

As the number of crossed edges equals $f - 1$, the number of color edges e_c satisfies

$$e_c = \tfrac{1}{2}f - \tfrac{1}{2}e_b + 1 < \frac{1}{2}f. \tag{4.3}$$

Thus, a color edge stream of size $\frac{1}{2}f$ is sufficient to store all the color edges. A similar formula holds for the vertices.

4.5.3 Adaptive Grid Traversals

Most of the mechanisms for static grid traversals are also applicable to adaptive grid traversals. On the one hand, the stack-&-stream access pattern is the same and boundary data and temporary data are similarly supported during adaptive traversals. Input and output streams are not identical though, as the input streams are part of the old grid, while the output streams belong to the new grid and have different sizes. Hence, there is more memory traffic in adaptive traversals, as the data volume on the streams is doubled.

Apart from the addition of the kernel interface, two major implementation aspects were adjusted compared to [125]. First, the refinement tree is completely removed from sam(oa)^2. Second, a *local* m : n*-repartitioning* algorithm is provided, which transforms m input sections into n output sections in parallel for any numbers m and n.

The refinement tree was removed from adaptive mesh refinement in sam(oa)^2, as most of the information, which is necessary for refinement and coarsening is already contained in the turtle grammar type and the plotter grammar type of each cell. The corresponding grammar decides how turtle and plotter types of parent elements or child elements are defined [8]. The only missing information is whether an element of the source grid is the first or the second child during coarsening. However, this is easily tracked by a Boolean flag that is switched back and forth for each coarsened source element. As coarsening of elements across section boundaries is not allowed in sam(oa)^2, the pattern of coarsened elements is always alternating and starts with a first child as the first coarsened cell.

The actual grid traversal is realized by synchronized traversals of the source and the destination grid, where the destination sections are parallelized by OpenMP threads. The most challenging task is to find the correct start point in the source grid, i.e. the first source element that overlaps the destination element. In order to do so, the source section that contains the source element must be found first. Next, all streams in the source section must be incremented to the start location of the source element. Additionally, vertices and edges must be pushed to the stacks.

Here, a two-step scheme is used. First the destination cell index is searched within the source sections. Each source section already tracks its number of post-refinement cells as explained in Section 4.5.2. Hence, the prefix sum of the number of post-refinement cells is computed for all source and destination sections to determine start indexes of the source and destination sections. Then, a simple comparison-based search finds the desired section. The second step is to traverse the source section until the matching destination cell index is found. To do so, sam(oa)^2 executes empty traversals, which iterate through the stack-&-stream system until the desired cell index is reached. Afterwards, the data transfer starts and the operators defined in the adaptive traversal interface (see Section 4.2) are called on pairs of source and destination elements until the destination section is fully processed. Note that multiple threads may access the same source section concurrently. In order to avoid data races, access to the source grid follows a strict read-only policy. Each thread uses its own stack and stream data structures for traversal of the source grid. Once the destination section is finished, the next section is processed. See Figure 4.7 for an illustration.

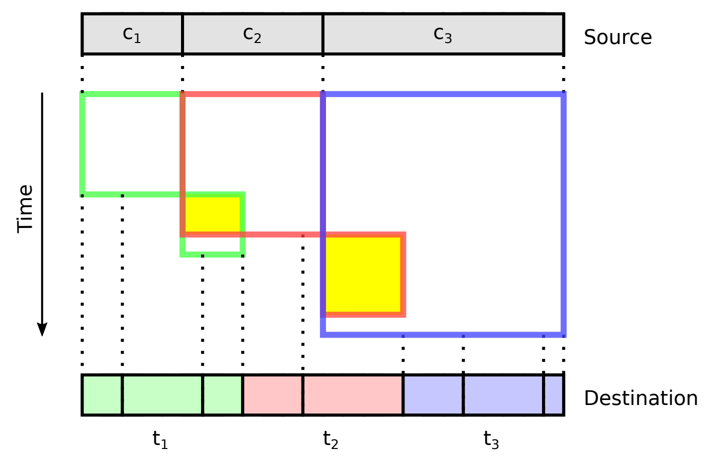


Figure 4.7: Concurrent access pattern for an adaptive traversal that transfers data from a source grid with 3 sections to a destination grid with 8 sections. 3 threads t_1, t_2 and t_3 work in parallel, where each is statically assigned to a set of sections in this example. The green, red, and blue frames mark the time spans where the corresponding thread accesses data from a source section. Empty traversals are executed in areas that do not contribute to the destination sections. Overlapping frames, marked by yellow areas, illustrate potential concurrent accesses to source sections.

Usually, the next section is the successor in traversal order. Hence sam(oa)^2 does not need to search for the second start point in the source grid, as the source grid iterator should already be set up correctly. If work stealing is enabled, it may happen though that a thread processes sections that are not consecutive in traversal order. In that case, the two-step search algorithm is applied once more to find the start point in the source grid.

4.6 Conclusion: A Fast and Flexible Framework

sam(oa)^2 provides a flexible, generic interface that supports adaptive mesh refinement and parallelization for finite element and finite volume methods. The finite element interface offers a set of operators for element-oriented formulations, which is an intentional restriction that will allow an application to benefit from the underlying efficient scheme.

The finite volume interface allows the implementation of explicit methods in a single grid traversal, promising a fast execution of time steps with little traversal overhead. The memory footprint is kept very low as most cell representations and all cell updates are stored in cheap temporary storage.

While the interface is powerful, it is kept at a low level and demands a big effort in order to implement scenarios. Interfaces with a high-level description, e.g. UFL [3], are in clear advantage here. A corresponding extension with a better usability and support for a more restricted class of problems would be beneficial.

The traversal back-end of sam(oa)^2 efficiently implements the stack-&-stream access system by reducing memory operations to the necessary minimum. It manages to completely hide the memory footprint of temporary data, allowing great flexibility for algorithm design.

One problem of the implementation is the redundant execution of the cell-to-edge operator for early evaluation. In finite volume methods, the cost is low and easily mitigated. However, for discontinuous

Galerkin methods, performance issues may arise, as the operator would have to execute numerical kernels instead of just moving data in memory. A possible solution for this problem is a lazy evaluation of traversals where traversal stages are connected on-the-fly. In this case, sam(oa)^2 has the freedom to decide when operators are executed and redundant calls are avoided. This idea could be developed further to an automatic resolution of data dependencies for a more high-level kernel interface.

Another issue of the static traversals is the amount of data operations and limitations due to management of temporary data. Even though memory operations are reduced by eliminating vertex buffers, temporary crossed edge and cell data must be managed in ring buffers, which cause instruction overhead and increased memory traffic. Without temporary data, first-touch and last-touch operators could be called in logical order instead of traversal order, which allows to discard the ring buffers in favor of vectorization and further optimizations.

Adaptive traversals are affected more by memory overhead than static grid traversals, as input and output streams cannot be merged to reduce memory traffic. Lists or vectors may tackle this problem, replacing the fixed-size arrays would be a trade-off though. Dynamic structures are slower than static data structures in general and will impair performance of static traversals. Concurrent read access and empty traversal overhead could be reduced by a faster way of finding the start element in a source section and setting the initial state of the traversal automaton. A unique assignment of source sections to threads during adaptive traversals would help here. Threads would only be allowed read access to their designated source and destination sections and no longer to other sections. Concurrent access is completely eliminated. Consequently, fragmentation and intra-process load imbalance will increase, but work stealing could mitigate the effect. The most radical optimization would be a clear separation of adaptive mesh refinement and repartitioning. This change would allow sam(oa)^2 to skip remeshing of regions, where no adaptive mesh refinement is necessary, and to perform repartitioning of local sections only where it is necessary.

PART III

TWO PHASE POROUS MEDIA FLOW

5 Two Phase Flow in Heterogeneous Porous Media

As our main test scenario, we implemented a porous media flow solver with a mixed, node-centered finite volume and finite element discretization. This example is a suitable test case for three goals:

- Demonstrating a use case of the finite element interface of sam(oa)^2 on a scenario that includes complicated algorithms such as linear solvers and a non-trivial interpolation method for mesh refinement and coarsening.
- Investigating the quality of adaptive mesh refinement and coarsening. The set of solvable Riemann problems spans a large test space that is suitable for a detailed error analysis.
- Examining the parallel performance of static traversals on a low-order memory-bound problem. This type of problem is computationally cheap and thus strongly affected by overhead from management of mesh structure and communication.

This problem is not suitable to examine performance of adaptive mesh refinement and load balancing as most of the execution time will be spent in linear solvers. A better test case for these components will be discussed in chapter 8.

5.1 Basic Concepts of Immiscible Porous Media Flow Physics

A good introduction to the topic of porous media flow simulations is given in [54], where the most important variables and their physical correlations are introduced. A short recapitulation based on the article will be given here.

We consider a domain filled with permeable material such as soil, sand or grain that permits flow of liquids or gases and assume that is contains two *fluid phases*, for example mixtures of water and oil in underground oil reservoirs or mixtures of water and gas in carbon reservoirs.

Phases, Saturation and Porosity

The fluid phases are distinguished into a wetting phase (in short: w) and a non-wetting phase (in short: n). Without going into detail, we will just assume that the wetting phase is water and the non-wetting phase is oil.

The domains that are considered are typically partial or whole reservoirs, whose lateral extent is in the order of kilometers, corresponding to the *meso scale* and the *field scale*. Pores are considered too small to be resolved and therefore fluid interfaces are not resolved either. Models that clearly separate both phases are not applicable. Instead, the averaged quantity $s_\alpha(x, y, z, t)$ is introduced that describes the saturation, which is the fraction per unit volume, of a phase $\alpha \in \{w, n\}$ in the position (x, y, z) at the time t. As saturation is a relative quantity,

$$s_w + s_n = 1. \tag{5.1}$$

Effective and Residual Saturation

In reality, the phases are never fully saturated nor fully desaturated; there is always some residual fluid present. These are modeled by residual saturations s_{wr} and s_{nr} that are assumed to have no effect on the flow.

Consequently, these residual terms have no impact on fluid flow and can be ignored in computations of mass transport. Only the *effective saturations* of wetting and non-wetting phase are considered instead, which are

$$\begin{aligned} s_{we} &:= \frac{s_w - s_{wr}}{1 - s_{wr} - s_{nr}}, \\ s_{ne} &:= \frac{s_n - s_{nr}}{1 - s_{wr} - s_{nr}}. \end{aligned} \tag{5.2}$$

Again,

$$s_{we} + s_{ne} = 1. \tag{5.3}$$

Darcy's Law

As an approximation, porous media flow is typically assumed to be laminary [54]. The only force considered here that acts on a phase, is caused by a pressure gradient. The resulting *phase velocity* can be derived from the Navier-Stokes equations and is described by

$$\mathbf{u}_\alpha = \lambda_\alpha(s_{\alpha e})\,\mathbf{K}(-\nabla p_\alpha). \tag{5.4}$$

This equation is called *Darcy's law*. The formula computes the volume transport (or phase velocity) $\mathbf{u}_\alpha$ of the phase α from a permeability tensor $\mathbf{K}$, the phase pressure gradient ∇p_α, and the phase mobility $\lambda_\alpha(s_{\alpha e})$, where the mobility is defined as

$$\lambda_\alpha(s_{\alpha e}) := \frac{\kappa_\alpha(s_{\alpha e})}{\mu_\alpha}, \tag{5.5}$$

which is a function of the relative permeability κ_α and the phase viscosity μ_α. The permeability tensor $\mathbf{K}(x, y, z)$ describes the amount and direction of a fluid passing through the medium when unit pressure gradients are applied to it. It is usually assumed to be diagonal.

If gravity is considered, the gravity vector $\mathbf{g}$ and the phase density ρ_α are added to the equation, resulting in the *extended Darcy's law*

$$\mathbf{u}_\alpha = \lambda_\alpha(s_{\alpha e})\,\mathbf{K}(-\nabla p_\alpha + \rho_\alpha \mathbf{g}). \tag{5.6}$$

Capillary Pressure

Surface tension on phase interfaces cause a force that acts on both phases and is modeled by a pressure discontinuity called *capillary pressure*. This effect is responsible for a multitude of physical phenomena in porous media. Capillary pressure is usually modeled as a function of the saturation. In our model, we consider only water and oil mixtures with a small capillary pressure, therefore we will neglect it here.

Relative Permeability

Many choices for the relative permeability term exist with increasing complexity and polynomial order. A linear model is defined by

$$\begin{aligned} \kappa_w(s_{we}) &= s_{we}, \\ \kappa_n(s_{ne}) &= s_{ne}. \end{aligned} \tag{5.7}$$

Unfortunately this model is not physical. In pressure-driven flow, both a shock wave and a rarefaction wave typically form at phase transitions. The linear model can only generate one of them at the same time however, as will be shown later in Section 6.1.1. To generate both wave types a non-convex flux function is necessary, which is given by a quadratic model

$$\begin{aligned}\kappa_w(s_{we}) &= s_{we}^2,\\ \kappa_n(s_{ne}) &= s_{ne}^2.\end{aligned}\tag{5.8}$$

While this term models the desired nonlinearity, it does not do so accurately. The Brooks-Corey model [25, 34]

$$\begin{aligned}\kappa_w(s_{we}) &= s_{we}^2 \cdot (s_{we})^{\frac{2}{\lambda}+1},\\ \kappa_n(s_{ne}) &= s_{ne}^2 \cdot (1-(1-s_{ne})^{\frac{2}{\lambda}+1}).\end{aligned}\tag{5.9}$$

extends the formula with the empirical pore-size distribution index $\lambda \in [0.2, 5]$, which is adapted to real world data and provides a much better approximation. As this model is quite complex, we will mostly concentrate on the quadratic model (5.8). It is sufficient to generate the desired wave types and easier to handle numerically due to its low polynomial order.

Transport equations

The mass of a phase per unit volume is given by $\rho_\alpha \Phi s_\alpha$, where $\Phi(x, y, z)$ is the *porosity*, i.e. the ratio of pore volume per unit volume in the medium. In general, porosity will be time-dependent if the material is compressible. For our scenarios the effect is negligible however. As the term $\rho_\alpha \Phi s_\alpha$ determines the phase mass, it is a conserved quantity in the system. This mass balance is expressed in the transport equation for each phase, using the extended Darcy's law (5.6)

$$(\rho_\alpha \Phi s_\alpha)_t + \operatorname{div}(\rho_\alpha \mathbf{u}_\alpha) = \rho_\alpha q_\alpha.\tag{5.10}$$

The equation states that a change of the phase mass is caused by transport of mass with the flux term $\rho_\alpha \mathbf{u}_\alpha$ and by a source term $\rho_\alpha q_\alpha$, where q_α is the incoming/outgoing phase volume over time at a source/sink. We further assume incompressible flow, hence the density ρ_α is constant and the transport equations simplify to the following expression

$$(\Phi s_\alpha)_t + \operatorname{div}(\mathbf{u}_\alpha) = q_\alpha.\tag{5.11}$$

Closed form for incompressible flow

For incompressible flow the equation system is closed at this point. With conservation of the total mass (5.1) and (5.3)

$$\begin{aligned}s_w &= 1 - s_n,\\ s := s_{we} &= 1 - s_{ne}.\end{aligned}$$

Capillary pressure is neglected, hence

$$p := p_w = p_n.$$

Plugging this into the extended Darcy's law (5.6) yields the phase velocities

$$\begin{aligned}\mathbf{u}_w &= \lambda_w(s)\,\mathbf{K}(-\nabla p + \rho_w \mathbf{g}),\\ \mathbf{u}_n &= \lambda_n(1-s)\,\mathbf{K}(-\nabla p + \rho_n \mathbf{g}).\end{aligned}\tag{5.12}$$

Addition of the phase transport equations (5.11) returns the total volume balance

$$\begin{aligned} (\Phi s_w)_t + \operatorname{div}(\mathbf{u}_w) &= q_w, \\ -(\Phi s_w)_t + \operatorname{div}(\mathbf{u}_n) &= q_n, \\ \Rightarrow \operatorname{div}(\mathbf{u}_w + \mathbf{u}_n) &= q_w + q_n. \end{aligned} \tag{5.13}$$

Effective and residual saturations are handled by substitution of the saturation in (5.13) with the effective saturation (5.2)

$$\begin{aligned} (\Phi(s_{we}(1 - s_{wr} - s_{nr}) + s_{wr}))_t + \operatorname{div}(\mathbf{u}_w) &= q_w, \\ -(\Phi(s_{we}(1 - s_{wr} - s_{nr}) + s_{wr}))_t + \operatorname{div}(\mathbf{u}_n) &= q_n. \end{aligned}$$

Simplification yields

$$\begin{aligned} (\Phi(1 - s_{wr} - s_{nr})s)_t + \operatorname{div}(\mathbf{u}_w) &= q_w, \\ -(\Phi(1 - s_{wr} - s_{nr})s)_t + \operatorname{div}(\mathbf{u}_n) &= q_n. \end{aligned}$$

Note that the term $(1 - s_{wr} - s_{nr})$ is a constant ratio of the saturation that can also be interpreted as a reduction of pore volume and applied to the porosity. Hence with a modified porosity term

$$\tilde{\Phi} := (1 - s_{wr} - s_{nr})\Phi,$$

the resulting model does not need explicit tracking of residual saturations any longer. For simplicity, we redefine the letter Φ to describe $\tilde{\Phi}$ henceforth and simplify the closed form to

$$\begin{aligned} (\Phi s)_t + \operatorname{div}(\mathbf{u}_w) &= q_w, \\ -(\Phi s)_t + \operatorname{div}(\mathbf{u}_n) &= q_n, \\ \operatorname{div}(\mathbf{u}_w + \mathbf{u}_n) &= q_w + q_n. \end{aligned} \tag{5.14}$$

Quasilinear form

For numerical analysis, it is useful to transform (5.14) into a *quasilinear form*

$$\alpha q_t + \mathbf{f}'(q) \cdot \nabla q = \Psi(q),$$

which resembles an advection equation with a flux derivative $\mathbf{f}'(q)$ and a source term $\Psi(q)$. As it is derived from the strong formulation of the partial differential equation system, this form is valid only where the solution q is continuous. The matrix $\mathbf{f}'(q)$ contains the signal speeds of the system [67], which can be used to derive a time step condition for discretizations. To obtain the quasilinear form, the pressure gradient in the extended Darcy's law (5.6) is replaced by the total velocity $\mathbf{u}_T := \mathbf{u}_w + \mathbf{u}_n$, and therefore

$$\begin{aligned} \mathbf{u}_T &= (\lambda_w(s) + \lambda_n(1-s))\,\mathbf{K}(-\nabla p) + (\lambda_w(s)\rho_w + \lambda_n(1-s)\rho_n)\,\mathbf{Kg}, \\ \mathbf{u}_w &= \frac{\lambda_w(s)}{\lambda_w(s) + \lambda_n(1-s)}\,(\lambda_w(s) + \lambda_n(1-s))\,\mathbf{K}(-\nabla p) + \lambda_w(s)\rho_w\mathbf{Kg}, \\ \Rightarrow \mathbf{u}_w &= \frac{\lambda_w(s)}{\lambda_w(s) + \lambda_n(1-s)}\,(\mathbf{u}_T - (\lambda_w(s)\rho_w + \lambda_n(1-s)\rho_n)\,\mathbf{Kg}) + \lambda_w(s)\rho_w\mathbf{Kg}. \end{aligned}$$

With some simplification one obtains the phase velocities

$$\begin{aligned} \mathbf{u}_w &= \frac{\lambda_w(s)}{\lambda_w(s) + \lambda_n(1-s)}\,(\mathbf{u}_T + \lambda_n(1-s)(\rho_w - \rho_n)\mathbf{Kg})\,, \\ \mathbf{u}_n &= \frac{\lambda_n(1-s)}{\lambda_w(s) + \lambda_n(1-s)}\,(\mathbf{u}_T + \lambda_w(s)(\rho_n - \rho_w)\mathbf{Kg})\,. \end{aligned} \tag{5.15}$$

from the total velocity $\mathbf{u}_T$. We insert (5.15) into the closed form (5.14) and receive the fractional flow formulation of the partial differential equation, which is

$$\begin{aligned}
&(\Phi s)_t + \operatorname{div}\left(\frac{\lambda_w(s)}{\lambda_w(s)+\lambda_n(1-s)}\left(\mathbf{u}_T + \lambda_n(1-s)(\rho_w-\rho_n)\mathbf{Kg}\right)\right) = q_w,\\
&-(\Phi s)_t + \operatorname{div}\left(\frac{\lambda_n(1-s)}{\lambda_w(s)+\lambda_n(1-s)}\left(\mathbf{u}_T + \lambda_w(s)(\rho_n-\rho_w)\mathbf{Kg}\right)\right) = q_n,\\
&\operatorname{div}(\mathbf{u}_T) = q_w + q_n.
\end{aligned} \tag{5.16}$$

Application of the chain rule to the first equation of (5.16) results in the system

$$\begin{aligned}
\Phi s_t &+ \frac{\lambda_w'(s)\lambda_n(1-s)-\lambda_w(s)\lambda_n'(s)}{(\lambda_w(s)+\lambda_n(1-s))^2}(\mathbf{u}_T\cdot\nabla s) + \frac{\lambda_w(s)}{\lambda_w(s)+\lambda_n(1-s)}(q_w+q_n),\\
&+\frac{\lambda_w'(s)\lambda_n^2(s)+\lambda_w^2(s)\lambda_n'(s)}{(\lambda_w(s)+\lambda_n(1-s))^2}(\rho_w-\rho_n)(\mathbf{Kg})\cdot\nabla s + \frac{\lambda_w(s)\lambda_n(1-s)}{\lambda_w(s)+\lambda_n(1-s)}(\rho_w-\rho_n)\operatorname{div}(\mathbf{Kg}) = q_w.
\end{aligned}$$

After restructuring, the quasilinear form is obtained, which is given by

$$\begin{aligned}
\Phi s_t &+ \left(\frac{\lambda_w'(s)\lambda_n(1-s)-\lambda_w(s)\lambda_n'(s)}{(\lambda_w(s)+\lambda_n(1-s))^2}\mathbf{u}_T + \frac{\lambda_w'(s)\lambda_n^2(s)+\lambda_w^2(s)\lambda_n'(s)}{(\lambda_w(s)+\lambda_n(1-s))^2}(\rho_w-\rho_n)\mathbf{Kg}\right)\cdot\nabla s,\\
&= \left(q_w - \frac{\lambda_w(s)}{\lambda_w(s)+\lambda_n(1-s)}(q_w+q_n) - \frac{\lambda_w(s)\lambda_n(1-s)}{\lambda_w(s)+\lambda_n(1-s)}(\rho_w-\rho_n)\operatorname{div}(\mathbf{Kg})\right).
\end{aligned} \tag{5.17}$$

From this formulation, the characteristic velocity ξ_w and the source term $\Psi(s)$ for the wetting phase can be extracted. Then,

$$\xi_w := \frac{\lambda_w'(s)\lambda_n(1-s)-\lambda_w(s)\lambda_n'(s)}{(\lambda_w(s)+\lambda_n(1-s))^2}\mathbf{u}_T + \frac{\lambda_w'(s)\lambda_n^2(s)+\lambda_w^2(s)\lambda_n'(s)}{(\lambda_w(s)+\lambda_n(1-s))^2}(\rho_w-\rho_n)\mathbf{Kg}, \tag{5.18}$$

$$\Psi(s) := q_w - \frac{\lambda_w(s)}{\lambda_w(s)+\lambda_n(1-s)}(q_w+q_n) - \frac{\lambda_w(s)\lambda_n(1-s)}{\lambda_w(s)+\lambda_n(1-s)}(\rho_w-\rho_n)\operatorname{div}(\mathbf{Kg}). \tag{5.19}$$

In short notation, we get

$$\Phi s_t + \xi_w\cdot\nabla s = \Psi(s).$$

The same equation is obtained for the non-wetting phase where the characteristic velocity is $\xi_n = \xi_w$. Hence, for every signal of the wetting phase there is a second signal of the non-wetting phase with the same velocity, which is a direct consequence of the total volume balance (5.13).

5.2 Three Models with Increasing Complexity

We will setup three models with increasing complexity based on the general problem formulation now, which will serve different purposes: A 1D model will be used for numerical analysis, a 2D channel flow model for performance analysis and the 3D model for comparison with benchmark data.

5.2.1 The Generalized Buckley Leverett Equations

First, we will consider a 1D scenario that offers many simplifications compared to the full model. For the relative permeability we allow any of the terms in (5.7), (5.8) and (5.9). Source terms are ignored, resulting in 1D conservative transport equations with the extended Darcy's law. That is,

$$\begin{aligned}
&(\Phi s_\alpha)_t + (u_\alpha)_x = 0,\\
&u_\alpha = \lambda_\alpha(s_\alpha)\,k(-p_x + \rho_w g).
\end{aligned}$$

In 1D all vectors $\mathbf{u}_w, \mathbf{u}_n, \mathbf{u}_T$ and the permeability tensor $\mathbf{K}$ become scalars u_w, u_n, u_T and k. Pressure gradient ∇p and flux divergence $\operatorname{div}(\mathbf{u}_w)$ are reduced to spatial derivatives p_x and $(u_w)_x$. The closed form of (5.14) therefore satisfies

$$\begin{aligned}(\Phi s)_t + (u_w)_x &= 0,\\ -(\Phi s)_t + (u_n)_x &= 0,\\ (u_T)_x &= 0.\end{aligned} \tag{5.20}$$

The actual *Buckley-Leverett Equations* are a special case of (5.20) that considers only the quadratic relative peremability model (5.8) and Darcy's law without gravity (5.4). They have been extensively researched in [26, 74, 130]. Instead, we will henceforth refer to (5.20) as Buckley-Leverett Equations, including gravity and general relative permeability models.

The 1D equations can be solved analytically and are therefore most suitable for error analysis. We will use them later to evaluate numerical solutions and to test the refinement and coarsening criteria.

5.2.2 2D Channel Flow with Heterogeneities

As a start point for a 2D scenario we choose the two-phase flow model in 2D, as described in [33]. It couples flow of two fluid phases, such as water and oil, in a heterogeneous porous medium, using the transport equations for incompressible flow (5.11) without source terms and Darcy's law (5.4) for the phase velocities. Thus,

$$\begin{aligned}(\Phi s_\alpha)_t + \operatorname{div}(\mathbf{u}_\alpha) &= 0,\\ \mathbf{u}_\alpha &= \lambda_\alpha(s_\alpha)\,\mathbf{K}(-\nabla p).\end{aligned}$$

Porosity is constant at

$$\Phi := 0.2.$$

The permeability tensor is assumed to be isotropic in this model and thus $\mathbf{K}$ is defined by

$$\mathbf{K} := k\mathbf{I},$$

where k is a a scalar field and $\mathbf{I}$ is the identity matrix. For the relative permeability the quadratic term in (5.8) is chosen:

$$\begin{aligned}\kappa_w(s) &:= s^2,\\ \kappa_n(1-s) &:= (1-s)^2.\end{aligned}$$

With insertion into the closed form of (5.14), the system

$$\begin{aligned}(\Phi s)_t + \operatorname{div}\left(\frac{s^2}{\mu_w}\,k(-\nabla p)\right) &= 0,\\ \operatorname{div}\left(\left(\frac{s^2}{\mu_w} + \frac{(1-s)^2}{\mu_n}\right) k(-\nabla p)\right) &= 0.\end{aligned} \tag{5.21}$$

is returned, where the non-wetting phase transport equation was omitted due to redundancy. Permeability is generated randomly from $\frac{1}{f^\alpha}$ noise with negative offsets at the top and bottom border. The smoothness parameter is chosen as $\alpha = 5$. Values are cut off at $0 \le k \le 10^{-8}$, thus the resulting field is a horizontal channel with smooth permeability transitions and large free flow areas. See Figure 5.1 for an example.

This scenario will be mainly used for performance analysis, as it is complicated enough to justify the usage of a pressure solver but has computationally cheap solver steps, suitable to measure traversal overhead.

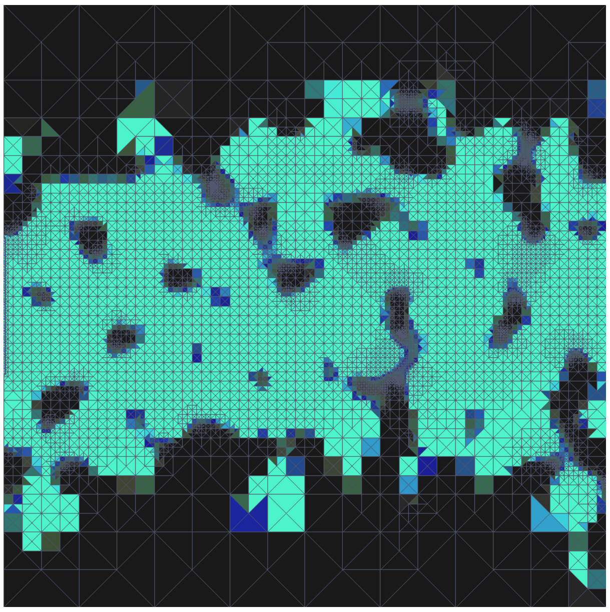


Figure 5.1: Randomly generated permeability for the 2D channel flow scenario. Black areas are impermeable, green areas have a permeability of 10^{-8}. In-between smooth transitions occur. Pressure-driven horizontal flow to the right causes displacement of a non-wetting phase by a wetting phase.

5.2.3 A 3D Model with Anisotropy and Gravity for the SPE10 Benchmark

First published in 2000, the second dataset of the Society of Petroleum Engineers' tenth benchmark problem (SPE10) [116] describes oil production by water injection in a cuboid domain. The 3D model for the SPE10 benchmark is based on the closed form for incompressible flow (5.14) using extended Darcy's law (5.6) with a diagonal, anisotropic permeability tensor and a variable porosity. The relative permeability is given by the quadratic term in (5.8). Here,

$$\begin{aligned}\kappa_w(s) &= s^2,\\ \kappa_n(1-s) &= (1-s)^2.\end{aligned}$$

The resulting closed form is given by

$$\begin{aligned}(\Phi s)_t + \operatorname{div}(\mathbf{u}_w) &= q_w,\\ -(\Phi s)_t + \operatorname{div}(\mathbf{u}_n) &= q_n,\\ \operatorname{div}(\mathbf{u}_w + \mathbf{u}_n) &= q_w + q_n.\end{aligned} \tag{5.22}$$

Permeability (Figure 5.2) is highly heterogeneous throughout the domain and spans eight orders of magnitude. The tensors have the form

$$\mathbf{K} = \begin{pmatrix} k_h & 0 & 0\\ 0 & k_h & 0\\ 0 & 0 & k_v \end{pmatrix}. \tag{5.23}$$

where k_h is the horizontal permeability and k_v is the vertical permeability. Hence, the permeability tensors are diagonal and horizontally isotropic. Porosity covers four orders of magnitude and reaches zero. The problem is ill-conditioned and actually harder to solve than scenarios obtained from realistic reservoir data. Choosing a homogeneous initial condition with an effective saturation of 0, the domain

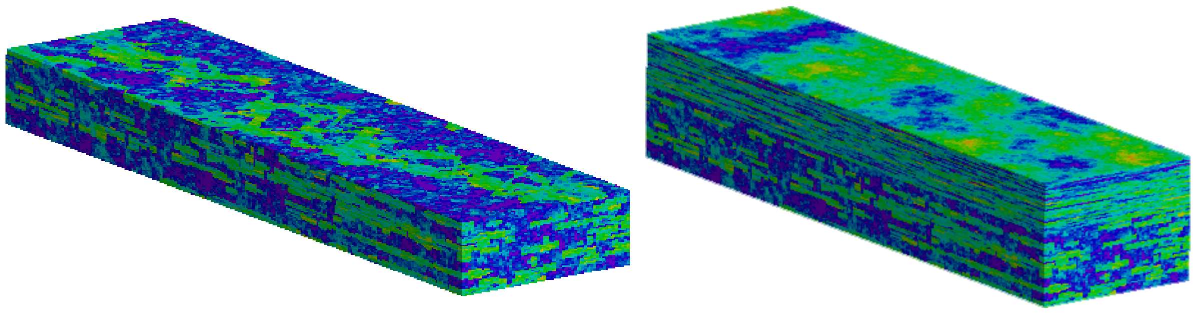


Figure 5.2: SPE 10 horizontal permeability field with a resolution of $60 \times 220 \times 85$ grid cells: The domain of the benchmark is divided into two horizontal layers, a channel-dominated *Tarbert* formation (left) and a fluvial *Upper Ness* formation (right, top 35 layers). Image source: [116].

is initially fully saturated with oil and residual water. Neumann boundary conditions with $\mathbf{u}_T = 0$ apply to all six walls. The source terms are wells, where a central, vertical well injects water at a constant rate and four vertical production wells P1, P2, P3 and P4 at the corners extract oil by pressure-induced flow. In time, the reservoir therefore fills with water that spreads out from the center and displaces the oil towards the production wells. We will take a closer look at the source terms in Section 5.5.2.

The goal of this scenario is to provide a test case with a hard-to solve problem in a production-ready environment that includes an advanced model and external data. We will evaluate the solution and analyze scalability to obtain an impression of performance under production conditions.

5.3 Mixed Discretization on Dual Grids

In this section, we will formulate a general solver for the coupled system described in (5.14) that can be applied to all three models defined in Section 5.2. The system can be solved semi-implicitly or fully implicitly with mixed discretization schemes [1, 32, 129]. We use an IMPES (IMplicit Pressure, Explicit Saturation) scheme, described in e.g. [7], that alternates between the implicit solution of a balance equation for the pressure p and the explicit solution of a transport equation for the saturation s. These equations correspond to the total volume balance and the phase transport equations in (5.14). The IMPES scheme is the canonical choice for the SPE10 benchmark [32] and easier to solve than an implicit formulation.

5.3.1 The Simulation Loop

We discretized the transport equation on the dual grid with node-centered finite volumes, obtaining the net update rule

$$s_j^{(t+\Delta t)} = s_j^{(t)} + \frac{\Delta t}{\Phi_j V_j} \left((Q_w)_j - \sum_{i \in \mathcal{N}(j)} A_{j,i} \, \mathcal{F}_w \left(s_j^{(t)}, s_i^{(t)} \right) \right) . \tag{5.24}$$

for each dual cell. The saturation update $s_j^{(t+\Delta t)} - s_j^{(t)}$ for each dual cell j is composed of the time step Δt, the effective cell volume $\Phi_j V_j$, the discrete source term $(Q_w)_j$ (nonzero only near wells) and net updates $\mathcal{F}(...)$ of each cell pair j, i, weighted by the surface area $A_{j,i}$. In sam(oa)^2 the dual grid is never assembled explicitly, instead each dual cell is composed of primary cell patches, causing little memory overhead. As numerical flux solvers $\mathcal{F}_w(s_l, s_r)$ and $\mathcal{F}_n(s_l, s_r)$, an upstream differencing scheme is used, described in detail in Section 5.3.2. The solution of the conservation equation in the closed form (5.14) is derived from the discrete transport equation (5.24) for both phases, hence

$$\sum_{i\in\mathcal{N}(j)} A_{j,i}\,\left(\mathcal{F}_w\left(s_j^{(t)}, s_i^{(t)}\right) + \mathcal{F}_n\left(s_j^{(t)}, s_i^{(t)}\right)\right) = (Q_w)_j + (Q_n)_j. \tag{5.25}$$

We use linear finite elements to discretize p and obtain a non-linear equation system for (5.25). This system must be solved in each IMPES time step, as its setup depends on the saturation s. An overview of the full simulation loop is given in Algorithm 5.1. Except for the initialization, all steps will be explained in detail in the following paragraphs:

- Section 5.3.2 explains the theory of upstream fluxes and the origin of the pressure system.
- In Section 5.3.3 setup of the non-linear system and its linearization are discussed.
- Section 5.3.2 implements the transport step.
- Section 5.3.5 covers the computation of the time step size Δt.
- In Section 5.3.6 an error indicator is defined for adaptive mesh refinement. Additionally, refinement and coarsening of pressure, saturation and further scenario data are explained.

5.3.2 Upstream Differencing for Two Phase Flow

Upstream differencing is an approximation of the Riemann solution on each interface of two cells j and i. We will apply the method only to Buckley-Leverett flow (5.20) here as this is sufficient to convey the general idea. Numerical fluxes $\mathcal{F}_w(s_l, s_r)$ and $\mathcal{F}_n(s_l, s_r)$ are computed on an interface between a left cell l and a right cell r by

$$\begin{aligned}
&\mathcal{F}_w(s_l, s_r) := \lambda_w^*(s_l, s_r)k(-p_x + \rho_w g),\\
&\mathcal{F}_n(s_l, s_r) := \lambda_n^*(s_l, s_r)k(-p_x + \rho_n g),\\
&\lambda_w^*(s_l, s_r) := \begin{cases} \lambda_w(s_l) & \text{if } \mathcal{F}_w(s_l, s_r) > 0 \\ \lambda_w(s_r) & \text{else} \end{cases} \quad \text{and} \quad \lambda_n^*(s_l, s_r) := \begin{cases} \lambda_n(1-s_l) & \text{if } \mathcal{F}_n(s_l, s_r) > 0 \\ \lambda_n(1-s_r) & \text{else} \end{cases},
\end{aligned} \tag{5.26}$$

where s_l and s_r are saturations of the left and right cell states at the interface, respectively. The system is implicit because the upstream mobilities λ_w^* and λ_n^* depend on the upstream directions, which are defined by the numerical fluxes. Additionally, the pressure derivative p_x is unknown, but can be solved for by expanding the total velocity

$$\begin{aligned}
u_T &= \mathcal{F}_w(s_l, s_r) + \mathcal{F}_n(s_l, s_r) = \lambda_w^*(s_l, s_r)k(-p_x + \rho_w g) + \lambda_n^*(s_l, s_r)k(-p_x + \rho_n g)\\
&= (\lambda_w^*(s_l, s_r) + \lambda_n^*(s_l, s_r))k(-p_x) + (\lambda_w^*(s_l, s_r)\rho_w + \lambda_n^*(s_l, s_r)\rho_n)kg,
\end{aligned} \tag{5.27}$$

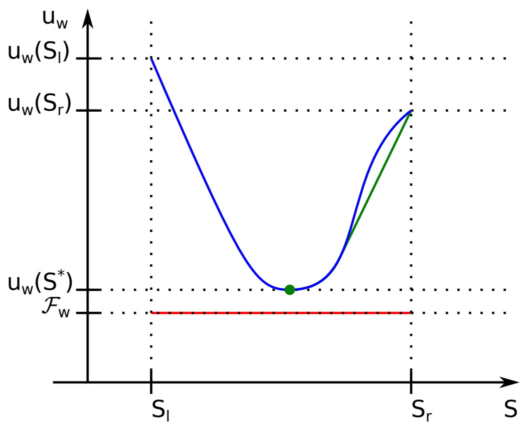
which is constant in 1D. Insertion into (5.26) returns

$$\begin{aligned}
&\mathcal{F}_w(s_l, s_r) = \frac{\lambda_w^*(s_l, s_r)}{\lambda_w^*(s_l, s_r) + \lambda_n^*(s_l, s_r)}(u_T + \lambda_n^*(s_l, s_r)(\rho_w - \rho_n)kg),\\
&\mathcal{F}_n(s_l, s_r) = \frac{\lambda_n^*(s_l, s_r)}{\lambda_w^*(s_l, s_r) + \lambda_n^*(s_l, s_r)}(u_T + \lambda_w^*(s_l, s_r)(\rho_n - \rho_w)kg),\\
&\lambda_w^*(s_l, s_r) := \begin{cases} \lambda_w(s_l) & \text{if } \mathcal{F}_w(s_l, s_r) > 0 \\ \lambda_w(s_r) & \text{else} \end{cases} \quad \text{and} \quad \lambda_n^*(s_l, s_r) := \begin{cases} \lambda_n(1-s_l) & \text{if } \mathcal{F}_n(s_l, s_r) > 0 \\ \lambda_n(1-s_r) & \text{else} \end{cases}.
\end{aligned} \tag{5.28}$$

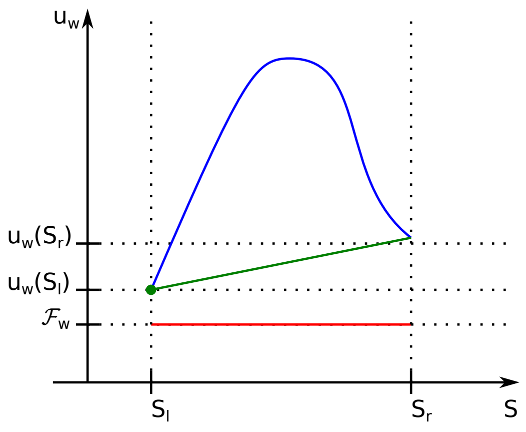
Algorithm 5.1: Pseudocode algorithm for the parallel simulation of two-phase porous media flow with adaptive refinement and coarsening. A single grid traversals is marked by a keyword. The linear solver possibly executes multiple traversals per call.

```
Input: $t_{max}$
Output: Saturation $s$, pressure $p$ at time $t_{max}$

traversal: Initialize $p$, $\mathbf{K}$ and $\Phi$;
traversal: Initialize $s$, set refinement flags, set up linear system matrix and right-hand side ;
traversals: Solve linear system for $p$;
// Grid setup with Poor Man's Multigrid
while refinement flags are set do
    traversal: Adapt grid (interpolate $p, s, \mathbf{K}, \Phi$) and balance load, ;
    while nonlinear pressure equation is not solved do
        traversal: Initialize $s$, set refinement flags, set up linear system matrix and right-hand
        side ;
        traversals: Solve linear system for $p$;
    end
end
// IMPES Time loop
$t \leftarrow 0$;
while $t < t_{max}$ do
    traversal: Set refinement and coarsening flags ;
    traversal: Adapt grid (interpolate/restrict $p, s, \mathbf{K}, \Phi$) and balance load ;
    while pressure equation is not solved do
        traversal: Set up linear system matrix and right-hand side ;
        traversals: Solve linear system for $p$;
    end
    traversal: Compute time step $\Delta t$;
    traversal: Update $s$ with an explicit transport step ;
    $t \leftarrow t + \Delta t$;
end
```



(a) Upstream flux for $(\rho_w - \rho_n)kg > 0$



(b) Upstream flux for $(\rho_w - \rho_n)kg < 0$

Figure 5.3: Upstream fluxes for density-driven flow: The images show solutions for $u_T = 0$, $s_l < s_r$ and the two cases $(\rho_w - \rho_n)kg > 0$ and $(\rho_w - \rho_n)kg < 0$. The phase velocities (5.15) define the algebraic flux $u_w(s)$ (blue line). allowing construction of a convex hull (green line). Then, the Godunov flux $u_w(s^*)$ (green dot) is the minimum of the convex hull in the domain $[s_l, s_r]$ [67]. The upstream flux (red line) is not greater than the Godunov flux if $s_l < s_r$ – a necessary condition for stability [24]. In **(b)**, s^* and s_l coincide and $u_w(s_l) = u_w(s^*)$.

Note that the method will return the phase velocities (5.15) if both fluxes have the same sign. Hence, in this case, the scheme will compute standard upwind fluxes for edge-centered velocities, as described in e.g. [67]. If the fluxes have different signs, then the result does not correspond to a physical flux.

Entropy requires that the upstream flux is bounded by the Godunov flux $u_w(s^*)$ [24, 102], therefore

$$0 \leq \frac{u_w(s^*) - \mathcal{F}_w(s_l, s_r)}{s_r - s_l} \leq \frac{u_w(s) - \mathcal{F}_w(s_l, s_r)}{s_r - s_l} \quad \text{for any } s \in [s_l, s_r] \tag{5.29}$$

Figure 5.3 illustrates a comparison of Godunov and upstream fluxes for the cases $u_T = 0$ and $(\rho_w - \rho_n)kg > 0$ (left image), $u_T = 0$ and $(\rho_w - \rho_n)kg < 0$ (right image). [102] additionally provides a stability condition

$$\Delta t \leq \frac{\Phi_j \Delta x_j}{\frac{\partial \mathcal{F}_w}{\partial s_l}(s_{j-1}, s_j) - \frac{\partial \mathcal{F}_w}{\partial s_r}(s_j, s_{j+1})}, \tag{5.30}$$

for upstream differencing using the partial derivatives $\frac{\partial \mathcal{F}_w}{\partial s_l}(s_l, s_r)$ and $\frac{\partial \mathcal{F}_w}{\partial s_r}(s_l, s_r)$. Here, Δt is the global time step size and Δx_j is the width of the cell j. We will discuss in Section 5.3.5 how the time step criterion is implemented in sam(oa)^2.

Simplification and Extension to Higher Dimensions

While the method in (5.28) is suitable for 1D Buckley-Leverett flow, it is not directly applicable to 2D and 3D models, as u_T is assumed to be a known constant in 1D. In higher dimensions however, $\mathbf{u}_T$ is not constant and must be determined locally. With

$$\mathbf{u}_T := \mathcal{F}_w(s_l, s_r) + \mathcal{F}_n(s_l, s_r), \tag{5.31}$$

we solve (5.25) for $\mathbf{u}_T$. In order to close the system, we substitute the upstream fluxes with the multidimensional upstream formula given by

$$\begin{aligned}
\mathcal{F}_w(s_l, s_r) &:= \lambda_w^* \mathbf{n}_{j,i}^T \mathbf{K}\left(-\nabla p + \rho_w \mathbf{g}\right) \\
\mathcal{F}_n(s_l, s_r) &:= \lambda_n^* \mathbf{n}_{j,i}^T \mathbf{K}\left(-\nabla p + \rho_n \mathbf{g}\right), \\
\lambda_w^* &:= \begin{cases} \lambda_w(s_l) & \text{if } \mathbf{n}_{j,i}^T \mathbf{K}(-\nabla p + \rho_w \mathbf{g}) > 0 \\ \lambda_w(s_r) & \text{else} \end{cases}, \\
\lambda_n^* &:= \begin{cases} \lambda_n(1 - s_l) & \text{if } \mathbf{n}_{j,i}^T \mathbf{K}(-\nabla p + \rho_n \mathbf{g}) > 0 \\ \lambda_n(1 - s_r) & \text{else} \end{cases},
\end{aligned} \tag{5.32}$$

Insertion into (5.25) returns the implicit system

$$\begin{aligned}
&\sum_{i \in \mathcal{N}(j)} A_{j,i} \left((\lambda_w^*)_{j,i} + (\lambda_n^*)_{j,i} \right) \mathbf{n}_{j,i}^T \mathbf{K}_{j,i} \left(-\nabla p_{j,i}\right) = \\
&(Q_w)_j + (Q_n)_j - \sum_{i \in \mathcal{N}(j)} A_{j,i} \left((\lambda_w^*)_{j,i} \rho_w + (\lambda_n^*)_{j,i} \rho_n \right) \mathbf{n}_{j,i}^T \mathbf{K}_{j,i} \mathbf{g},
\end{aligned} \tag{5.33}$$

that must be solved for the pressure p and the upstream mobilities λ_w^* and λ_n^*. Section 5.3.3 will explain in detail how to achieve this.

Upstream transport

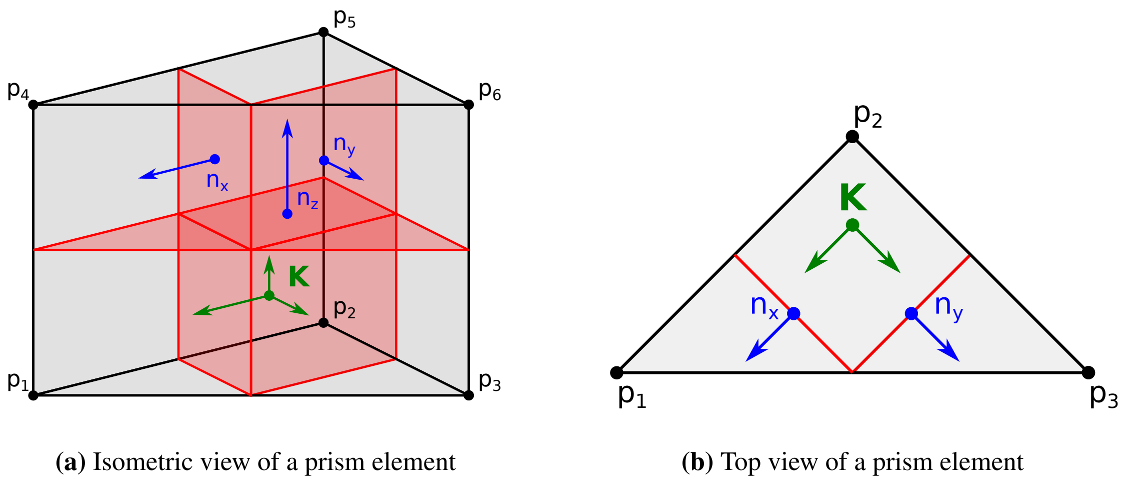
We assume now that the pressure system is solved and a time step is determined. To execute the upstream transport step we implement the net update rule (5.24) using the finite element interface. A straightforward implementation is shown in Algorithm 5.2. The volume kernel computes the fluxes on all dual cell interfaces and accumulates the results in the net updates $(F_w)_j$ for each dual cell. Once the net updates are computed, they are applied to the saturation in the last DoF kernel with an explicit Euler time step.

Algorithm 5.2: Upstream transport using the finite element interface of sam(oa)^2.

```
Input: $\mathbf{p}$, $\lambda_w^*$, $\mathbf{s}$, $\Delta t > 0$
Output: $\mathbf{s}$
traversal
    first DoF kernel: $(F_w)_j \leftarrow 0$;
    volume kernel: $(F_w)_j \leftarrow (F_w)_j + \sum_{i \in \mathcal{N}(j)} (\lambda_w^*)_{j,i} \mathbf{n}_{j,i}^T \mathbf{K}_{j,i}(-\nabla p_{j,i} + \rho_w \mathbf{g})$;
    last DoF kernel: $s_j \leftarrow s_j - \frac{\Delta t}{V_j}(F_w)_j$;
end
```

5.3.3 Solving the Nonlinear System for Pressure and Upstream Mobilities

To solve the system defined by (5.32) and (5.33), we employ a staggered scheme for coupled systems which alternates computing upstream mobilities and solving the pressure equation until convergence is reached. At first, (5.32) is solved for the upstream mobilities using the initial pressure guess p. Next, (5.33) is solved implicitly for the pressure. Then, the solution is tested by computing the upstream mobilities a second time to check if their solutions changed. Afterwards, (5.33) is solved once more and if the mobilities have not changed, the linear solver returns immediately and the solution is already found. If the mobilities have changed, the linear solver performs further iterations and the solution must be tested once again. Due to the piecewise constant nature of the upstream mobilities, convergence of the coupled scheme is usually reached after the first iteration already.



(a) Isometric view of a prism element **(b)** Top view of a prism element

Figure 5.4: Dual grid discretization for prism elements. The interfaces between the dual cells are represented by red areas. The normals are marked by blue arrows. Marked in green, the eigenvectors of the permeability tensor $\mathbf{K}$ are aligned with the normals and the grid is $\mathbf{K}$-orthogonal by design.

Setup of the Linear System

In order to solve the linear system, we claim that on all interfaces between dual cells j and i, the normal $\mathbf{n}_{j,i}$ is an eigenvector of the transposed permeability tensor $\mathbf{K}^T_{j,i}$ with the eigenvalue $k_{j,i}$. Hence,

$$\mathbf{K}^T_{j,i}\mathbf{n}_{j,i} = k_{j,i}\mathbf{n}_{j,i}. \tag{5.34}$$

The value of $k_{j,i}$ is

$$k_{j,i} = \mathbf{n}^T_{j,i}\mathbf{K}_{j,i}\mathbf{n}_{j,i}. \tag{5.35}$$

Grids with this property are called $\mathbf{K}$-*orthogonal*. For Buckley-Leverett flow and 2D channel flow described in Section 5.2.1 and Section 5.2.2, condition (5.34) is trivially fulfilled, as the permeability tensors are isotropic in both cases. For the SPE10 benchmark in Section 5.2.3, $\mathbf{K}$-orthogonality is not obvious, as the permeability tensor is only horizontally isotropic and element transformations on unstructured grids can return full local tensors that may also be asymmetric. However, as seen in Figure 5.4, the dual grid discretization is always $\mathbf{K}$-orthogonal if $\mathbf{K}$ is diagonal and horizontally isotropic. Inserting (5.34) into (5.33) and discretizing the pressure with linear finite elements, we obtain a *two-point flux approximation* now, as explained in [129] for example. The resulting equations are

$$\begin{aligned}&\sum_{i\in\mathcal{N}(j)} \frac{A_{j,i}}{\Delta x_{j,i}}\left((\lambda^*_w)_{j,i} + (\lambda^*_n)_{j,i}\right) k_{j,i}(-p_j + p_i)\\ =&(Q_w)_j + (Q_n)_j - \sum_{i\in\mathcal{N}(j)} A_{j,i}((\lambda^*_w)_{j,i}\rho_w + (\lambda^*_n)_{j,i}\rho_n)k_{j,i}\mathbf{n}^T_{j,i}\mathbf{g}.\end{aligned} \tag{5.36}$$

In matrix-vector notation this linear system is written as

$$\mathbf{A}\mathbf{p} = \mathbf{b}, \tag{5.37}$$

where the matrix coefficients are given by

$$\begin{aligned}
\mathbf{A}_{j,j} &= -\sum_{i\in\mathcal{N}(j)} \frac{A_{j,i}}{\Delta x_{j,i}}((\lambda_w^*)_{j,i} + \lambda_n^*)_{j,i})k_{j,i}, \\
\mathbf{A}_{j,i} &= \frac{A_{j,i}}{\Delta x_{j,i}}\left((\lambda_w^*)_{j,i} + (\lambda_n^*)_{j,i}\right) k_{j,i} \quad \text{for } j \neq i, \\
\mathbf{b}_j &= (Q_w)_j + (Q_n)_j - \sum_{i\in\mathcal{N}(j)} A_{j,i}\left((\lambda_w^*)_{j,i}\rho_w + (\lambda_n^*)_{j,i}\rho_n\right) k_{j,i}\mathbf{n}_{j,i}^T\mathbf{g}.
\end{aligned} \tag{5.38}$$

As the system matrix is symmetric and positive definite [129], a Jacobi-preconditioned Conjugate Gradients solver is applied for the solution.

5.3.4 A Diagonally Preconditioned Conjugate Gradients Solver for the Pressure System

Implementation of a Conjugate Gradients solver with diagonal preconditioning is fairly straightforward in sam(oa)^2. We follow the method described in [114] and implement all steps by respective kernels in Algorithm 5.3 using the finite element interface presented in Section 4.1.1. Data dependency enforces that an iteration is split into two traversals: As we need a full grid traversal to compute a dot product on a global vector, we can use the result only in the next grid traversal. The dot products $\mathbf{d}^T\mathbf{u}$ and $\mathbf{r}^T\mathbf{D}\mathbf{r}$ are needed to compute α and β respectively, resulting in the cyclic dependency

$$\beta \to \mathbf{d} \to \alpha \to \mathbf{r} \to \beta. \tag{5.39}$$

Thus it is necessary to split computation into two traversals and $2n + 1$ grid traversals are required for n solver iterations. The solver is enabled in sam(oa)^2 with the runtime flag

```
samoa -lsolver 1
```

As exit criteria, sam(oa)^2 uses the pressure difference of the domain

$$\mathbf{r}^T\mathbf{r} < \epsilon^2(\max(\mathbf{p}) - \min(\mathbf{p}))^2,$$

and the relative error norm

$$\mathbf{r}^T\mathbf{r} < \epsilon^2\mathbf{r}_0^T\mathbf{r}_0,$$

where $\mathbf{r}_0 = \mathbf{D}^{-1}(\mathbf{b} - \mathbf{A}\mathbf{x}_0)$ is the initial preconditioned residual vector. The value of ϵ is controlled with the argument

```
samoa -epsilon <value>
```

5.3.5 A Flux Based CFL Condition on Unstructured Grids

The goal of this section is to find a cheap stability condition for the time step size Δt. The CFL condition, named after F. Courant, K. O. Friedrich and H. Lewy [35,36], states that for each cell $j \in \{1, \ldots, n\}$

$$\left|\Delta t \sum_{i\in\mathcal{N}(j)} \frac{v_{j,i}}{\Delta x_{j,i}}\right| \leq 1. \tag{5.40}$$

where Δt is the global time step size, $v_{j,i}$ is the signal velocity at the interface between cells j and i and $\Delta x_{j,i}$ is the characteristic length of the interface j, i, defined by

$$\Delta x_{j,i} = \frac{\hat{V}_j}{A_{j,i}}, \tag{5.41}$$

Algorithm 5.3: Diagonal-preconditioned CG solver implemented in the finite element interface of sam(oa)^2.

```
Input: $A$ positive definite, symmetric, $\mathbf{D}$ diagonal, $\mathbf{x}, \mathbf{b}, \epsilon > 0$
Output: $\mathbf{x}$
$\alpha \leftarrow 0; \beta \leftarrow 0; \mathbf{d} \leftarrow 0;$
traversal
    volume kernel: $r \leftarrow \mathbf{D}^{-1}(\mathbf{b} - \mathbf{Ax});$
    last DoF kernel: $\omega \leftarrow \mathbf{r}^T\mathbf{r}; \gamma \leftarrow \mathbf{r}^T\mathbf{Dr};$
end
while $\omega > \epsilon$ do
    traversal
        first DoF kernel: $\mathbf{d} \leftarrow r + \beta\,\mathbf{d};$
        volume kernel: $\mathbf{u} \leftarrow \mathbf{Ad};$
        last DoF kernel: $\delta \leftarrow \mathbf{d}^T\mathbf{u};$
    end
    $\alpha \leftarrow \frac{\gamma}{\delta}; \gamma_{old} \leftarrow \gamma;$
    traversal
        first DoF kernel: $v \leftarrow \mathbf{D}^{-1}\mathbf{u}; \mathbf{x} \leftarrow \mathbf{x} + \alpha\,\mathbf{d}; \mathbf{r} \leftarrow \mathbf{r} - \alpha\,\mathbf{v};$
        last DoF kernel: $\omega \leftarrow \mathbf{r}^T\mathbf{r}; \gamma \leftarrow \mathbf{r}^T\mathbf{Dr};$
    end
    $\beta \leftarrow \frac{\gamma}{\gamma_{old}};$
end
```

which is the ratio of effective cell volume $\hat{V}_j$ to cell interface area $A_{j,i}$. Solving for Δt, we get

$$\Delta t \leq \frac{\hat{V}_j}{\left|\sum\limits_{i \in \mathcal{N}(j)} A_{j,i} v_{j,i}\right|}. \tag{5.42}$$

Applied to porous media flow, the effective volume translates to the pore volume $\hat{V}_j = \Phi_j V_j$ and the signal velocity $v_{j,i}$ to the maximum speed $\xi^-_{j,i}$ of all ingoing waves at the interface j, i, according to (5.18). Then,

$$\Delta t \leq \frac{\Phi_j V_j}{\sum\limits_{i \in \mathcal{N}(j)} A_{j,i} \xi^-_{j,i}}. \tag{5.43}$$

$\xi^-_{j,i}$ is typically hard to evaluate and must be approximated carefully. Usually, a simpler method is therefore chosen. In literature, a related formula is found that computes the time step size

$$\Delta t \leq \alpha \min\left(\frac{\Phi_j V_j}{\sum\limits_{i \in \mathcal{N}(j)} A_{j,i} \mathcal{F}^-(s_j, s_i)}, \frac{\Phi_j V_j}{\sum\limits_{i \in \mathcal{N}(j)} A_{j,i} \mathcal{F}^+(s_j, s_i)}\right), \tag{5.44}$$

from ingoing and outgoing net fluxes $\mathcal{F}^-_w(s_j, s_i)$ and $\mathcal{F}^+_w(s_j, s_i)$ [54] for some $\alpha < 1$. This method is quick and easy to evaluate and provides better results than the upstream condition (5.30) in our tests but requires empirical evaluation of the parameter α which is strongly problem-dependent. Another simple choice is to directly enforce that the saturation s_j in (5.24) remains in the physical range $[0, 1]$, hence

$$\begin{aligned} s_j + \frac{\Delta t}{\Phi_j V_j} \left((Q_w)_j - \sum_{i \in \mathcal{N}(j)} A_{j,i}\, \mathcal{F}_w\left(s_j, s_i\right) \right) &\geq 0, \\ s_j + \frac{\Delta t}{\Phi_j V_j} \left((Q_w)_j - \sum_{i \in \mathcal{N}(j)} A_{j,i}\, \mathcal{F}_w\left(s_j, s_i\right). \right) &\leq 1. \end{aligned} \tag{5.45}$$

resulting in the condition

$$\Delta t \leq \max\left(-\frac{\Phi_j V_j S_j}{(Q_w)_j - \sum\limits_{i \in \mathcal{N}(j)} A_{j,i} \mathcal{F}_w(s_j, s_i)}, \frac{\Phi_j V_j (1 - S_j)}{(Q_w)_j - \sum\limits_{i \in \mathcal{N}(j)} A_{j,i} \mathcal{F}_w(s_j, s_i)} \right). \tag{5.46}$$

sam(oa)^2 implements this stability condition using its finite element interface, see Section 4.1.1. Except for the global reduction of the time step size, the implementation evaluates and accumulates the wetting phase velocities similar to the transport step in Section 5.3.2. We will skip the details here.

5.3.6 Mass Conservative Refinement

The only missing step for the scenario is adaptive refinement and coarsening. This requires an error indicator that is able to decide for each element if it should be refined or not. Additionally, interpolation and restriction operators have to be defined for each variable of interest.

A Posteriori Error Indicators

In order to define an error indicator for the saturation, we have to consider that the analytic solution s_{exact} is discontinuous. Hence, a norm that mitigate peaks at discontinuities should be chosen for error analysis of a discretization s. A suitable candidate is the $\mathcal{L}_1$ norm, as it measures the integral over the absolute error, given by

$$\mathcal{L}_1(s) = \int_\Omega |s_{exact} - s| d\Omega.$$

In [65] an $\mathcal{L}_1$-error estimator was developed for upwind methods on finite volume discretizations. Applied to the full N-dimensional model (5.14) it is defined by

$$\begin{aligned} \int_{\Omega_j} |s^0_{exact} - s^0| d\Omega &< C_0 \mathrm{Tol}, \\ \max_{i \in \mathcal{N}(j)} \left((\Delta t + \Delta x_{ji}) \Delta x_{ji}^{N-1} |s_i^t - s_j^t| \right) &< C_x \mathrm{Tol}, \\ \Delta x_j^N |s_j^{t+\Delta t} - s_j^t| &< C_t \mathrm{Tol}. \end{aligned} \tag{5.47}$$

for each dual cell j, some constants C_0, C_x, C_t and a tolerance $\mathrm{Tol} > 0$. Δx_j is the cell diameter and Δx_{ji} are distances between cell centers. As the constants are problem-specific and hard to evaluate, we will not apply the estimator here, but use some of its ideas to define an error indicator.

We assume that s is an $\mathcal{L}_1$-optimal piecewise constant discretization of s_{exact}. The error in the $\mathcal{L}_1$ norm can be estimated by the sum over all primary cells j; that is,

$$\mathcal{L}_1(s) = \int_\Omega |s_{exact} - s| d\Omega \leq \frac{1}{2} \sum_{j=1}^{n} |\max_{x \in \Omega_j}(s_{exact}) - \min_{x \in \Omega_j}(s_{exact})| V(\Omega_j). \tag{5.48}$$

(5.48) suggests that additional degrees of freedom are invested well in cells Ω_j where

$$|\max_{x\in\Omega_j}(s_{exact}) - \min_{x\in\Omega_j}(s_{exact})|V(\Omega_j).$$

is large. The term is estimated in sam(oa)^2 with

$$\Delta s_j V(\Omega_j) := |\max_{x\in\Omega_j}(s) - \min_{x\in\Omega_j}(s)|V(\Omega_j).$$

This is a useful approximation only if s is not constant in Ω_j, hence the choice of primary cells as control volumes. Note that a similar criterion is obtained with the error estimator (5.47) if initial error and time discretization error are ignored. The full indicator is therefore given by

$$\Delta s_j V(\Omega_j) > \text{Tol}_s(s_{max} - s_{min})V(\Omega_{min}),$$

where the right-hand side is a product of the maximum saturation difference $s_{max} - s_{min} = 1$ and the minimum cell volume $V(\Omega_{min})$, causing a stricter criterion with increased maximum refinement depth. $\text{Tol}_s > 0$ is a dimensionless parameter that is chosen at runtime.

Pressure is approximated with piecewise linear functions, allowing strict analysis with the maximum norm. The indicator

$$\Delta p_j > \text{Tol}_p(p_{max} - p_{min})\frac{\Delta x_{min}}{x_{max} - x_{min}},$$

is used for pressure refinement in sam(oa)^2. The criterion is easy to evaluate and will refine regions with high pressure gradients in order to resolve sources and sinks, but it is not mathematically meaningful.

Interpolation and Restriction of Saturation

As we are aiming to simulate the SPE10 benchmark [116], we will require correct global quantities such as accumulated oil production for analysis. Hence, if mass is not conserved during adaptive refinement and coarsening, these quantities will be erroneous. Pointwise conservation of mass $m_\alpha = \rho_\alpha \Phi s_\alpha$ of a phase α cannot be achieved, because coarsening and refinement modifies the shape of dual cells and will inevitably change the saturation in some regions. Instead, element-wise conservation of mass can be achieved by using finite element theory. We define a set of variables that lives on the old grid: shape functions $\hat{\phi}$, porosity $\hat{\Phi}$, phase saturation $\hat{s}_\alpha$. Similarly we define a set of variables that lives on the new grid: shape function ϕ, porosity Φ, phase saturation s_α and a test basis ψ. Element-wise conservation of mass yields

$$\int_\Omega \rho_\alpha \Phi_j (\sum_i (s_\alpha)_i \phi_i)\psi_j d\Omega = \int_\Omega \rho_\alpha \hat{\Phi}_j (\sum_i (\hat{s}_\alpha)_i \hat{\phi}_i)\psi_j d\Omega.$$

In matrix-vector notation, two linear systems of equations

$$\begin{aligned}\rho_w \mathbf{M}_{\phi,\psi}\Phi_\phi \mathbf{s}_\phi &= \rho_w \mathbf{M}_{\hat{\phi},\psi}\hat{\Phi}_{\hat{\phi}}\hat{\mathbf{s}}_{\hat{\phi}},\\ \rho_n \mathbf{M}_{\phi,\psi}\Phi_\phi(\mathbf{1} - \mathbf{s}_\phi) &= \rho_n \mathbf{M}_{\hat{\phi},\psi}\hat{\Phi}_{\hat{\phi}}(\mathbf{1} - \hat{\mathbf{s}}_{\hat{\phi}}),\end{aligned}$$

for both phases are obtained, where $\mathbf{M}_{\phi,\psi}$ is the mass matrix of the new grid and $\mathbf{M}_{\hat{\phi},\psi}$ is a non-square mass matrix built from the shape functions of the old grid and the test functions of the new grid. The conditions simplify to

$$\begin{aligned}\mathbf{M}_{\phi,\psi}\Phi_\phi \mathbf{s}_\phi &= \mathbf{M}_{\hat{\phi},\psi}\hat{\Phi}_{\hat{\phi}}\hat{\mathbf{s}}_{\hat{\phi}},\\ \mathbf{M}_{\phi,\psi}\Phi_\phi &= \mathbf{M}_{\hat{\phi},\psi}\hat{\Phi}_{\hat{\phi}}.\end{aligned} \quad (5.49)$$

Hence, a well-defined solution exists only if the additional refinement condition for the porosity is fulfilled, too. As we choose piecewise constant bases for ϕ and ψ, the mass matrix $\mathbf{M}_{\phi,\psi}$ is diagonal. Effectively, we therefore need to compute the saturation of each dual cell on the new grid by averaging all saturations of dual cells on the old grid, weighted by the intersecting volumes of water in old and new cell. Algorithm 5.4 shows how this is implemented via the specific sam(oa)^2 kernels. All kernels perform essentially the same operation, except that source and destination element coincide in the transfer kernel. In refinement and coarsening kernels, intersection volumes must be determined.

Algorithm 5.4: Mass conservative saturation transfer from old grid (marked by cell volume $\hat{V}_j$, porosity $\hat{\Phi}_j$ and saturation $\hat{s}_j$) to new grid (V_i, Φ_i, s_i). Saturation s_j in cell j is the average over the saturations $\hat{s}_j$ of all old cells i which intersect with the new cell, weighted by the intersecting volumes of water.

```
Input: $\hat{s}, \hat{\Omega}, \Omega$
Output: $s$
traversal
    refine kernel/transfer kernel/coarsen kernel:
    $V_j^w \leftarrow \sum_i V(\Omega_j \cap \hat{\Omega}_i)\hat{\Phi}_i \hat{s}_i$; $V_j \leftarrow \sum_i V(\Omega_i \cap \hat{\Omega}_j)\hat{\Phi}_i$;
    last DoF kernel: $s_j \leftarrow \frac{V_j^w}{V_j}$;
end
```

Interpolation and Restriction of Pressure

For the pressure p, refinement and coarsening would optimally keep the linear system in a solved state, conserving the total velocity (5.25). Figure 5.5 shows that there is no local flux-conservative solution for subgrid refinement in general, as the resulting system would be over-determined. Hence, after adaptive refinement and coarsening, the pressure equation must be solved globally. Still, in order to find a good start value for the pressure solver, a heuristics for pressure refinement and coarsening must be defined. For simplicity, the arithmetic average

$$p_5 = \frac{p_1 + p_2 + p_3 + p_4}{4} \tag{5.50}$$

is chosen in the middle cell. An element-wise algorithm is obtained by weighting the old pressure $\hat{p}$ with intersection volumes of old and new cells $V(\hat{\Omega}_i \cap \Omega_j)$ to define the new pressure p, defined by

$$p_j := \frac{\sum_{i=1}^{\hat{n}} V(\hat{\Omega}_i \cap \Omega_j)\hat{p}_i}{V(\Omega_j)}. \tag{5.51}$$

The pressure is averaged over the new cell area, which is a generalization of (5.50) for refinement and coarsening. Better local methods are possible by including mobility and permeability into (5.51); however, sam(oa)^2 uses the simple method, as time stepping typically changes the linear system by a larger amount than adaptive mesh refinement.

The resulting algorithm is essentially the same as Algorithm 5.4 for the saturation refinement, except that the pressure is weighted by the total volume instead of the effective volume. We will therefore skip the implementation details.

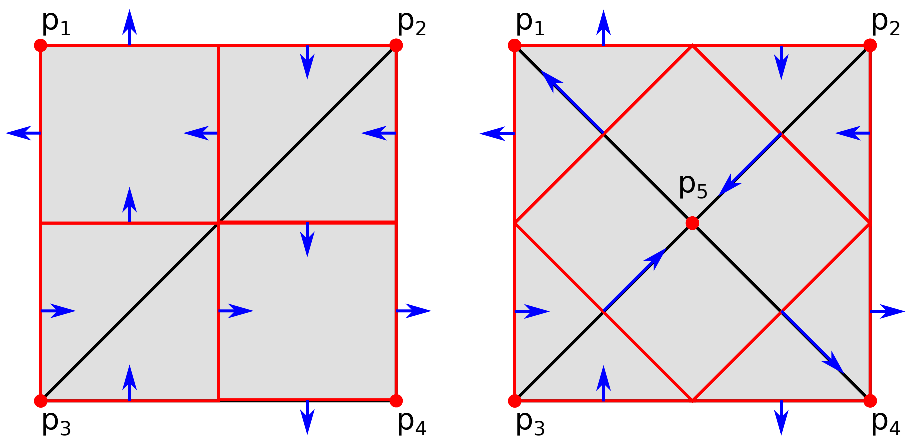


Figure 5.5: Example for refinement on a subgrid of a primary grid (black) and dual grid (red) discretization with constant saturation and permeability. The total velocities $\mathbf{u}_T$ are marked by blue arrows. After refinement, a new cell is generated with an additional pressure unknown p_5, which must be chosen to fulfill (5.36) in each of the 5 cells. To keep the method local, the boundary pressures p_1 to p_4 and the boundary fluxes would have to remain invariant, which is impossible as the resulting system would be over-determined. The solution of the SPE10 scenario is mostly affected by a Neumann condition, that determines inflow and outflow of the system. Hence, sam(oa)^2 attempts to preserve local fluxes instead of the local pressure. This requires a global pressure solution however.

Interpolation and Restriction of Porosity and Permeability

According to (5.49), strict mass conservation requires porosity refinement by volume-weighted averaging. Hence, after refinement of a primary cell into two cells and after coarsening of two primary cells into one cell, the porosity of the coarse cell should be equal to the average porosity of both fine cells.

For coarsening this is straightforward by simple averaging. For refinement, the condition can be fulfilled by on-the-fly integration of porosity data. sam(oa)^2 supports this approach, using an integration kernel shown in Algorithm 5.5. By using recursive bisection to integrate over the porosity in each element, the porosity integral is evaluated always on the same quadrature points, independent of adaptive refinement. This ensures that the integral is consistent and the porosity condition for mass conservation (5.49) is fulfilled. Furthermore, for numerical stability we want to avoid porosity evaluations on data discontinuities, e.g. on the domain boundary. Therefore, quadrature points are laid on the center of mass of each element. Internally, data management and retrieval is executed by the library ASAGI (A parallel server for adaptive geoinformation) [5], a student project developed by Sebastian Rettenberger.

Refinement of permeability is performed in the same manner as for porosity, using numerical integration of the permeability source. However, integration by arithmetic averaging of the input data usually does not return good results. Christie et al. [32] show that influence of the upscaling method for permeability is quite high in the SPE10 benchmark, see Fig. 5.6. sam(oa)^2 therefore supports different averaging methods for arithmetic and geometric averaging. The resulting permeability tensor always remains diagonal and horizontally isotropic, but is not fully isotropic in 3D due to the vertical component.

Algorithm 5.5: Pseudocode for 2D porosity and permeability adaption. Coarsening is done by averaging, transfer by copying and refinement by numerical integration over the destination element via recursive newest vertex bisection. $\hat{\mathbf{\Phi}}$ is the porosity vector of the old grid, $\mathbf{\Phi}$ is the porosity vector of the new grid, d_j is the refinement depth of cell j, d_{src} is the refinement depth of the source data and $\Phi(\mathbf{x})$ and $\mathbf{K}(\mathbf{x})$ are the porosity and permeability sources, defined in local coordinates for convenience. The function $\mathbf{t}$ parameterizes averaging by choosing appropriate component-wise transformations $\mathbf{t}_{ari} :=$ identity (arithmetic mean) and $\mathbf{t}_{geo} := \log$ (geometric mean).

```
Input: $\hat{\mathbf{\Phi}}$, $\hat{\mathbf{K}}$
Output: $\mathbf{\Phi}$, $\mathbf{K}$

Function integrate_internal($\mathbf{f}$, $\mathbf{x}_1$, $\mathbf{x}_2$, $\mathbf{x}_3$, $d$)
    if $d > 0$ then
        return $\frac{1}{2}$integrate_internal($\mathbf{x}_1$, $\frac{1}{2}(\mathbf{x}_1 + \mathbf{x}_3)$, $\mathbf{x}_2$, $d - 1$)
        $+\frac{1}{2}$integrate_internal($\mathbf{x}_2$, $\frac{1}{2}(\mathbf{x}_1 + \mathbf{x}_3)$, $\mathbf{x}_3$, $d - 1$) ;
    else
        return $\mathbf{f}(\frac{1}{3}(\mathbf{x}_1 + \mathbf{x}_2 + \mathbf{x}_3))$;
    end
end

Function integrate($\mathbf{f}$, $\mathbf{x}_1$, $\mathbf{x}_2$, $\mathbf{x}_3$, $d$, $\mathbf{t}$)
    return $\mathbf{t}^{-1}$(integrate_internal($\mathbf{t}(\mathbf{f})$, $\mathbf{x}_1$, $\mathbf{x}_2$, $\mathbf{x}_3$, $d$) );
end

traversal
    transfer kernel: $\Phi_j \leftarrow \hat{\Phi}_j; \mathbf{K}_j \leftarrow \hat{\mathbf{K}}_j$;
    coarsen kernel: $\Phi_j \leftarrow \mathbf{t}_{ari}^{-1}\left(\sum_i \frac{V(\Omega_j \cap \hat{\Omega}_i)}{V(\Omega_j)} \mathbf{t}_{ari}(\hat{\Phi}_i)\right)$ ; $\mathbf{K}_j \leftarrow \mathbf{t}_{geo}^{-1}\left(\sum_i \frac{V(\Omega_j \cap \hat{\Omega}_i)}{V(\Omega_j)} \mathbf{t}_{geo}(\hat{\mathbf{K}}_i)\right)$;
    refine kernel:
    $\Phi_j \leftarrow$integrate($\Phi$, $(1,0)^T, (0,0)^T, (0,1)^T, d_{src} - d_j$, $\mathbf{t}_{ari}$) ;
    $\mathbf{K}_j \leftarrow$integrate($\mathbf{K}$, $(1,0)^T, (0,0)^T, (0,1)^T, d_{src} - d_j$, $\mathbf{t}_{geo}$) ;
end
```

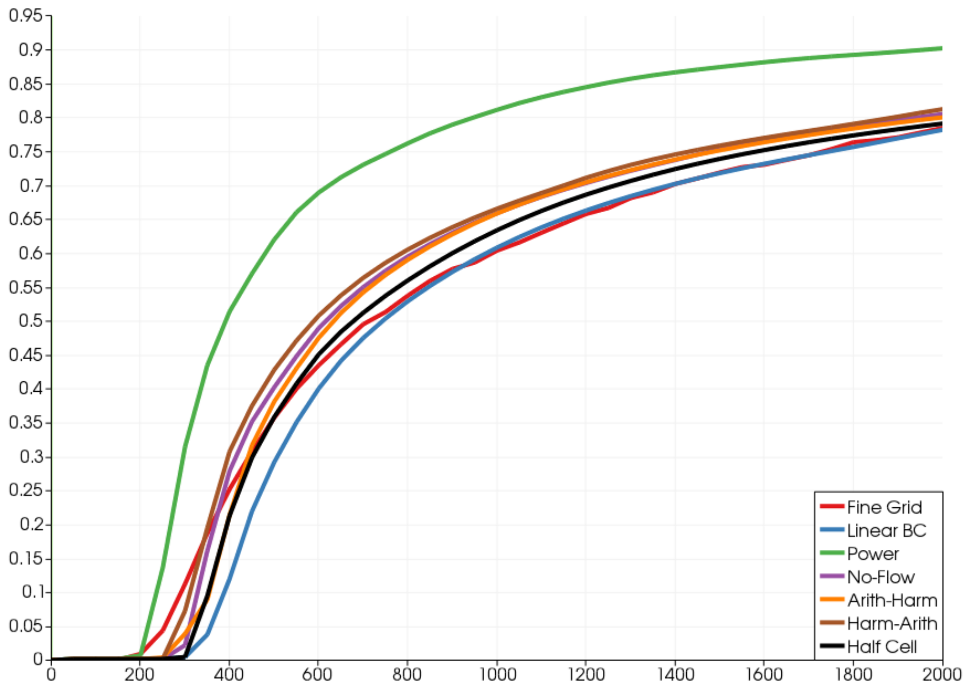


Figure 5.6: A comparison of different upscaling methods tested for the water cut over time of producer P1 in the SPE10 benchmark. Variance is high, which suggests that the choice of the method has a large impact on the outcome. Data source: [32]

5.4 Optimizing the Linear Solver

We will take a short detour in this section to investigate performance of the Conjugate Gradients solver as implemented in Section 5.3.4. An optimization for memory-bound performance will be presented here and different preconditioners will be examined for efficiency.

5.4.1 Increasing the Computational Intensity of the Linear Solver

Looking back at Algorithm 5.3 we see that two grid traversals are required for each iteration due to data dependency between the dot products and vector updates. For a memory-bound solver, this is unsatisfactory as in both traversals all data is touched, but only half of the unknowns are updated. Better performance could be achieved if all operations were performed in a single traversal. Indeed, analysis with the roofline model [128] in Section 7.2.1 will confirm that memory throughput is satisfying, however floating point performance appears to be far below the peak. The key issue here is that computational intensity of the Conjugate Gradients solver is extremely low, returning a performance limit that is far below the actual capabilities of the system.

Already in 1985 a solution was known to this problem, suggested first by Saad [100]. The idea is to precompute $\beta = \frac{\gamma}{\gamma_{old}}$ before evaluation of the new residual $\mathbf{r} - \alpha\mathbf{v}$ by replacing the computation of γ with

$$\gamma \leftarrow (\mathbf{r} - \alpha\,\mathbf{v})^T\mathbf{D}\,(\mathbf{r} - \alpha\,\mathbf{v}) = \mathbf{r}^T\mathbf{D}\,\mathbf{r} - 2\,\alpha\,\mathbf{v}^T\mathbf{D}\,\mathbf{r} + \alpha^2\,\mathbf{v}^T\mathbf{D}\,\mathbf{v}. \tag{5.52}$$

and evaluating β as before. A naive implementation of (5.52) would require two extra dot products $\mathbf{v}^T\mathbf{D}\,\mathbf{r}$ and $\mathbf{v}^T\mathbf{D}\,\mathbf{v}$ per solver iteration in addition to $\mathbf{r}^T\mathbf{D}\,\mathbf{r}$. However, we can exploit that the precon-

Algorithm 5.6: Fused Conjugate Gradients solver with diagonal preconditioning: Compared to classical CG (see Algorithm 5.3) this version executes only one grid traversal per iteration, but computes an extra dot product.

```
Input: $A$ positive definite, symmetric, $\mathbf{D}$ diagonal, $\mathbf{x}, \mathbf{b}, \epsilon > 0$
Output: $\mathbf{x}$
$\alpha \leftarrow 0$ ; $\beta \leftarrow 0$ ; $\mathbf{d} \leftarrow 0$ ; $\mathbf{v} \leftarrow 0$ ;
traversal
    volume kernel: $r \leftarrow \mathbf{D}^{-1}(\mathbf{b} - \mathbf{A}\,\mathbf{x})$ ;
    last DoF kernel: $\omega \leftarrow \mathbf{r}^T\mathbf{r}$ ; $\gamma \leftarrow \mathbf{r}^T\mathbf{D}\,\mathbf{r}$ ;
end
while $\omega > \epsilon$ do
    traversal
        first DoF kernel: $\mathbf{x} \leftarrow \mathbf{x} + \alpha\,\mathbf{d}$ ; $\mathbf{r} \leftarrow \mathbf{r} - \alpha\,\mathbf{v}$ ; $\mathbf{d} \leftarrow \mathbf{r} + \beta\,\mathbf{d}$ ;
        volume kernel: $\mathbf{u} \leftarrow \mathbf{A}\,\mathbf{d}$ ; $\mathbf{v} \leftarrow \mathbf{D}^{-1}\mathbf{u}$ ;
        last DoF kernel: $\delta \leftarrow \mathbf{d}^T\mathbf{u}$ ; $\nu \leftarrow \mathbf{v}^T\mathbf{u}$ ; $\omega \leftarrow \mathbf{r}^T\mathbf{r}$ ; $\gamma \leftarrow \mathbf{r}^T\mathbf{D}\,\mathbf{r}$ ;
    end
    $\alpha \leftarrow \frac{\gamma}{\delta}$ ; $\gamma_{old} \leftarrow \gamma$ ; $\gamma \leftarrow \alpha^2\,\nu - \gamma_{old}$ ; $\beta \leftarrow \frac{\gamma}{\gamma_{old}}$ ;
end
```

ditioned residuals are $\mathbf{D}$-orthogonal, hence

$$\begin{aligned}(\mathbf{r} - \alpha\,\mathbf{v})^T\mathbf{D}\,\mathbf{r} &= 0\\ \Rightarrow \alpha\,\mathbf{v}^T\mathbf{D}\,\mathbf{r} &= \mathbf{r}^T\mathbf{D}\,\mathbf{r},\end{aligned} \tag{5.53}$$

to eliminate the term $-2\,\alpha\,\mathbf{v}^T\mathbf{D}\,\mathbf{r}$ in (5.52). Saad originally proposed to remove a second dot product by determining γ incrementally instead of evaluating $\mathbf{r}^T\mathbf{D}\,\mathbf{r}$ in each iteration. Meurant [79] later claimed that this might lead to numerical instability and suggested to keep the dot product and use it as a correction term. Hence, (5.52) assumes the final form

$$\gamma \leftarrow \alpha^2\,\mathbf{v}^T\mathbf{D}\,\mathbf{v} - \mathbf{r}^T\mathbf{D}\,\mathbf{r}. \tag{5.54}$$

Applying (5.54), we switch the order of operations in Algorithm 5.3 and compute β along with α before the solution update. Most importantly, we may join search vector and solution update into a single traversal now, overlapping the matrix-vector product $\mathbf{u} \leftarrow \mathbf{A}\,\mathbf{d}$ and the application of the diagonal preconditioner $\mathbf{v} \leftarrow \mathbf{D}^{-1}\mathbf{u}$. No extra helper vectors are needed and therefore, the memory footprint is not increased by this fused version of the Conjugate Gradients solver. An implementation in sam(oa)^2 is given in Algorithm 5.6.

Note that there is still one extra dot product needed per iteration which affects scalability on large numbers of cores. However, the effect is not worse than in classical Conjugate Gradients if communication is latency-bound. Additionally, an extra iteration is necessary due to executing an empty solution update in the first iteration (as the new α is not known at this point yet) and successively postponing the solution update to the next iteration. Hence, the fused Conjugate Gradients solver executes $n + 2$ traversals for n solver iterations, a significant improvement over the $2\,n + 1$ traversals of the classical method in Algorithm 5.3 and an increase of the computational intensity and subsequent speedup of a factor 2.

The fused Conjugate Gradients solver is the default linear solver in sam(oa)^2 and is explicitly activated with the runtime option

```
samoa -lsolver 2
```

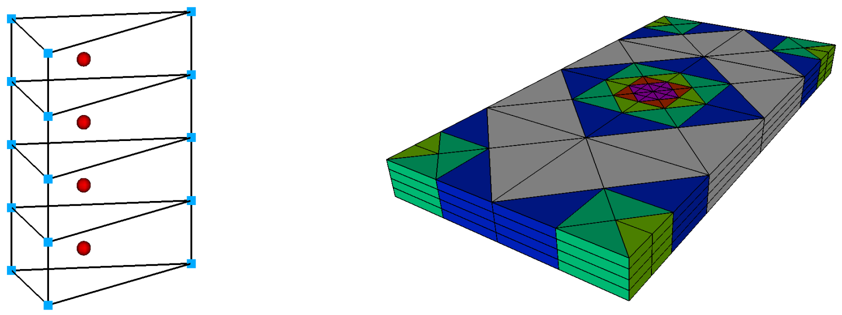


Figure 5.7: A prism grid with four layers in sam(oa)^2: The left image shows how in each 2D vertex, five pressure and saturation unknowns must be stored, corresponding to five 3D points (blue squares). In each triangle cell, four permeability, porosity and velocity values are stored, corresponding to the four prism cell centers (red spheres). Z-major order assures compact storage and unit-stride access for a column-wise processing. The full grid (right image) may be adaptively refined and coarsened in horizontal direction, but not in vertical direction.

5.4.2 Preconditioners for the Conjugate Gradients Solver

In his student project, Klimenko [64] investigated which preconditioners for the Conjugate Gradients solver return the best result for the linear systems defined in (5.38). He found that in general, a diagonal preconditioner provides good performance and improves the solution time drastically. An even better choice would be a Cholesky factorization, however Klimenko determined that the overhead required for this preconditioner would at least triple the amount of grid traversals while it would reduce the number of iterations only by half on average. Hence, the method would be too costly. Apart from Multigrid methods no local methods were found to improve the runtime. More details on the subject are available in Klimenko's thesis.

5.5 Towards Production Ready Code: The SPE10 Benchmark

We will now take a look at the requirements for simulation of the SPE10 scenario, which was mentioned in Section 5.2.3. At this point, most building blocks for parallel, adaptive simulation of a complete reservoir model are already present. In this chapter, we will discuss the last missing pieces: Integration of 2.5D elements into sam(oa)^2 and addition of source terms into the model.

5.5.1 From Triangular Grids to 2.5D Prism Grids

The extension to 2.5D grids has been explained in [75] already and will be shortly recapitulated here. sam(oa)^2 does not impose any restrictions on the amount of data stored in each entity (element, edge or vertex), a straightforward extension to 2.5D is therefore realized by storage of degree-of-freedom arrays of fixed size instead of single degree of freedom in each vertex. While each vertex has two-dimensional coordinates x and y already, the third dimension is added by associating each array entry in the vertex with a z-coordinate. For cell data, the same procedure is possible by transforming triangle cells to prism cells. The prism arrays must have one entry less than the vertex arrays, since cell data is associated with the cell centers, which lie between vertices on the vertical axis. Figure 5.7 shows a prism grid with four layers that stores five 3D points in each vertex and four prism cells in each triangle. The grid may be dynamically refined and coarsened in horizontal direction, but the number of vertical layers is constant.

As data is stored and processed in z-major order, a kernel implementation is easily vectorized over

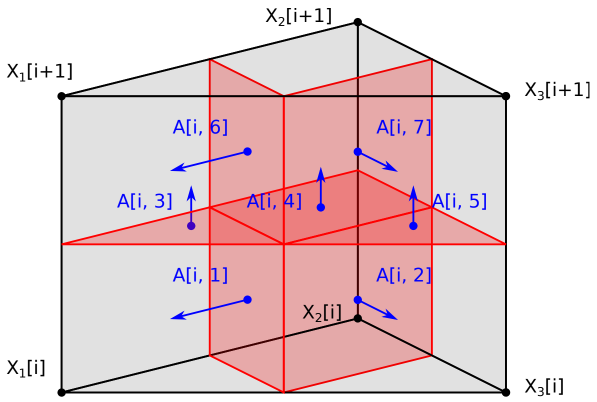


Figure 5.8: Storage scheme of the unknowns and the matrix for element-wise computation of the matrix vector product $\mathbf{Ax}$ in 2.5D porous media flow. Memory layout of the vector x is z-column-wise in the triangle vertices 1 to 3. Similarly, the matrix entries A (marked in blue) are stored z-column-wise in the triangle elements. A sparse pattern is used to store only one matrix entry for each of the 7 dual cell interfaces (red surfaces) in each prism element, exploiting sparsity and symmetry of the Two-Point Flux Approximation. Storage space for the element matrix is reduced from $6^2 = 36$ to 7 doubles per prism element.

the z-direction when the number of layers is large enough, due to independent access to unit-stride, fixed-size arrays. This has been exploited in the linear solver. For memory efficiency the element matrices are not stored as dense blocks, but in a sparse format that is generated during the setup phase for the linear system, exploiting symmetry and zero entries. Figure 5.8 shows the concept for a single prism element. Fluxes are defined on the dual cell interfaces (red) and depend only on two pressure unknowns due to the diagonal permeability tensor. Hence, one matrix entry (blue) per dual cell interface is sufficient. This reduces storage space for the element matrix from 36 to only 7 doubles per prism element. Note that in theory, no storage space would be required for the matrix if the fluxes were evaluated on-the-fly. However, due to the involved nonlinear operations, flux evaluations are expensive and slow down execution compared to loading the matrix in each linear solver iteration. Chapter 7 confirms this, when CPU times of all simulation components are compared on recent hardware.

The matrix-vector product is implemented in sam(oa)^2 by a volume kernel using Fortran array notation, shown in Algorithm 5.7. The advantage of the array notation in the kernel is auto-vectorization by the compiler. Some issues remain though, such as misaligned vector access. Further information on vectorization performance is found in the Master's thesis by Pachalieva [86].

5.5.2 Well Models for Reservoirs

Sources and sinks in reservoir simulation are modeled by wells that are typically cylindrical, vertical holes that either inject fluid into the ground (*injection wells*) to generate a pressure-induced flow or extract it (*production wells*) for production. The SPE10 scenario contains one injection well in the center and four production wells at the four corners, all are vertically complete through the domain.

Typically, well diameters are much smaller than the grid resolution and pressure gradients become large, hence the discretization error will be large in the region if left unattended. Assuming a horizontally isotropic permeability k_h as in (5.23), the Peaceman well model [88] uses an approach based on the assumption that near the well, flow is almost radial. The well is treated as an internal Neumann boundary

Algorithm 5.7: Matrix-vector product kernel in sam(oa)^2 for a full prism element, where `x1, x2, x3` are the unknown arrays located at the triangle vertices, `r1, r2, r3` are residuals and `A` is the matrix, stored in a sparse element matrix. The Fortran 2008 keyword **`contiguous`** is a hint to the compiler that the array entries are stored without stride, allowing for some auto-vectorization.

```
subroutine apply3D(x1, x2, x3, r1, r2, r3, A)
    real, contiguous, intent(in)       :: x1(:), x2(:), x3(:)
    real, contiguous, intent(inout)    :: r1(:), r2(:), r3(:)
    real, contiguous, intent(in)       :: A(:,:)

    !bottom horizontal contributions
    r1(1:_NZ) = r1(1:_NZ) + A(:,1) * (x1(1:_NZ) - x2(1: _NZ))
    r2(1:_NZ) = r2(1:_NZ) + A(:,1) * (x2(1:_NZ) - x1(1: _NZ))
    r3(1:_NZ) = r3(1:_NZ) + A(:,2) * (x3(1:_NZ) - x2(1: _NZ))
    r2(1:_NZ) = r2(1:_NZ) + A(:,2) * (x2(1:_NZ) - x3(1: _NZ))

    !vertical contributions
    r1(1:_NZ) = r1(1:_NZ) + A(:,3) * (x1(1:_NZ) - x1(2:_NZ+1))
    r1(2:_NZ+1) = r1(2:_NZ+1) + A(:,3) * (x1(2:_NZ+1) - x1(1:_NZ))

    r2(1:_NZ) = r2(1:_NZ) + A(:,4) * (x2(1:_NZ) - x2(2:_NZ+1))
    r2(2:_NZ+1) = r2(2:_NZ+1) + A(:,4) * (x2(2:_NZ+1) - x2(1:_NZ))

    r3(1:_NZ) = r3(1:_NZ) + A(:,5) * (x3(1:_NZ) - x3(2:_NZ+1))
    r3(2:_NZ+1) = r3(2:_NZ+1) + A(:,5) * (x3(2:_NZ+1) - x3(1:_NZ))

    !top horizontal contributions
    r1(2:_NZ+1) = r1(2:_NZ+1) + A(:,6) * (x1(2:_NZ+1) - x2(2:_NZ+1))
    r2(2:_NZ+1) = r2(2:_NZ+1) + A(:,6) * (x2(2:_NZ+1) - x1(2:_NZ+1))
    r3(2:_NZ+1) = r3(2:_NZ+1) + A(:,7) * (x3(2:_NZ+1) - x2(2:_NZ+1))
    r2(2:_NZ+1) = r2(2:_NZ+1) + A(:,7) * (x2(2:_NZ+1) - x3(2:_NZ+1))
end subroutine
```

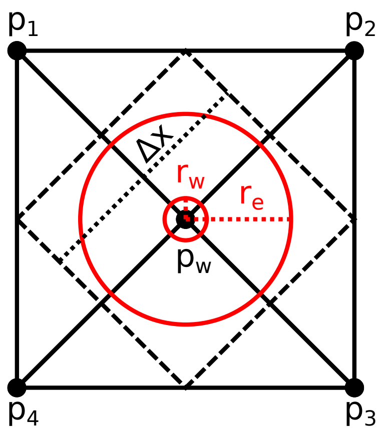


Figure 5.9: A vertical well in a 2D domain that is aligned with the center vertex of the grid. As the well radius r_w is usually small compared to the mesh width Δx, the steep pressure gradient near the well is not captured properly by naive discretization. Applying the well model (5.55), a logarithmic error term for the pressure function near the well is therefore introduced. In 2D the well element is modeled as a spherical cell (big red sphere) with an effective cell radius r_e that is chosen as a function of Δx to match the numerical solution on the discrete well element (marked by the dashed lines) with the analytic solution.

condition that adds an additional unknown and an additional equation to the system (5.25). The equation

$$Q = \int\limits_{\partial W} (\lambda_w + \lambda_n) k_h (p - p_w - \bar{\rho} g (z_w - z)) \left(\ln \left(\frac{r_e}{r_w} \right) + s \right) d\Omega, \tag{5.55}$$

correlates the total well flow rate Q and the well bottom hole pressure p_w, of which at least one value must be specified to close the system. Here,

∂W	is the well surface.
p	is the discretized cell pressure.
$\bar{\rho}$	is the mobility-weighted average density in the well (assuming saturation is constant in the well).
Q	is the total well flow rate.
p_w	is the bottom-hole well pressure, bounded by $(p_w)_{max} \geq p_w$ in injection wells.
z	is the position on the vertical axis.
z_w	is the bottom-hole z-coordinate.
r_w	is the well radius.
r_e	is the effective cell radius, a control parameter to match the analytic pressure solution.
s	is the skin factor that accounts for (e.g. drilling-induced) permeability changes near the well.

Figure 5.9 shows a vertical well in a 2D domain consisting of four primary elements. The well in the image aligns with a vertex, which is not necessarily the case though.

For injection wells, the bottom hole pressure p_w is usually assumed unknown and computed from the well flow rate Q. If p_w exceeds the maximum bottom hole pressure $(p_w)_{max}$, then the model switches to a constant well pressure $(p_w)_{max}$ and the well flow rate Q is determined from $(p_w)_{max}$ instead. For the production wells, the bottom hole pressure p_w is constant in our case.

Even though the well equation does not appear to fit the discretization of the grid, implementation of the model is possible in sam(oa)^2. The goal is to rewrite the well equation into the form of (5.38), so we are able to reuse the existing nonlinear solver with as little change as possible. Hence, we start by setting the discrete pressure degrees of freedom in the well (we call this subset W) to the well pressure

$$p_j := p_w + \bar{\rho} g (z_w - z_j) \qquad \text{for } j \in W. \tag{5.56}$$

For production wells, we mark the pressure values as Dirichlet conditions. For injection wells, an initial guess from the previous pressure solution or the maximum injection well pressure is chosen for p_w. Without further changes, the pressure solver would immediately violate (5.56), as it considers the pressure unknowns in the well column independent of each other and consequently applies independent updates. This undesired behavior can be avoided however, which will be explained in the next paragraph.

Well Pressure Equalization

The canonical way to prevent independent updates in the well degrees of freedom is to merge all dual cells in the injector into a single cell with a single unknown p_w and a single equation (5.55). This is not directly possible, since sam(oa)^2 assumes the number of unknowns per column is constant over all columns. However, as the Conjugate Gradients method (Algorithm 5.3) derives the update vector $\mathbf{d}$ only from the right-hand side $\mathbf{b}$ and matrix-vector products $\mathbf{Ax}$, we merely have to ensure that right-hand-side and matrix-vector product entries are equal for all degrees of freedom in the well, thus enforcing uniform changes to all entries. One way to achieve this is to average over all matrix entries and right-hand side entries that affect degrees of freedom in the cell. For any matrix vector product $\mathbf{r} = \mathbf{Ax}$ the result vector $\mathbf{r}$ is modified to return

$$\begin{aligned} r_j &\leftarrow \frac{1}{|W|} \sum_{k \in W} \sum_{i \in N(j)} A_{k,i} x_i = \sum_{i \in N(j)} \left(\frac{1}{|W|} \sum_{k \in W} A_{k,i} \right) x_i \quad \text{for } j \in W, \\ b_j &\leftarrow \frac{1}{|W|} \sum_{k \in W} b_k \quad \text{for } j \in W. \end{aligned} \tag{5.57}$$

This modified system is no longer symmetric in general but nevertheless, the Conjugate Gradients solver has no trouble with the term in our test cases. Note that the weight $\frac{1}{|W|}$ improves the condition of the linear system and does not affect the solution as it is applied both to the system matrix and the right-hand side.

Well Pressure Correction

As stated before, the analytic pressure solution near the well is a logarithmic term, which is known to be approximated with a harmonic function by first order finite elements discretization. Hence, a discretization error is introduced in the pressure solution that increases with the distance from the well, but is limited. Following Peaceman [88], the pressure error at infinite distance in a uniformly refined grid with mesh width Δx and homogeneous permeability k_h is approximately

$$\frac{Q}{2\pi k_h} \log \left(\frac{0.2 \Delta x}{r_w} \right). \tag{5.58}$$

In the well model (5.55), this corresponds to a skin of $s = 0$ and an effective cell radius of $r_e = 0.2\Delta x$. The error is not directly applied to the well pressure as a correction term in sam(oa)^2, because the permeability is in general heterogeneous near the well. Instead, the discrete derivative $\frac{p_i - p_j}{\Delta x_{j,i}}$ in (5.36) is modified to account for the pressure difference. We achieve this by replacing the local mesh width $\Delta x_{j,i}$ with a custom value $\Delta_w x_{j,i}$ for computation of the matrix entries $A_{j,i}$ in (5.38). Hence,

$$\Delta_w x_{j,i} := \Delta x_{j,i} \left(1 + \frac{2}{\pi} \log \left(\frac{0.2 \Delta x_{j,i}}{r_w} \right) \right) \qquad \text{if either } j \in W \text{or } i \in W. \tag{5.59}$$

We will skip the proof here and refer to numerical analysis in Chapter 6 instead, where simulation results are compared to the analytic solution for a simple heterogeneous problem.

Adding Boundary Conditions to the Transport Step

The final update is an addition of well boundary conditions to the transport step. By setting pressure Dirichlet conditions for production wells, we do not fulfill the mass balance in producer elements and therefore accumulate additional mass that must be removed from the cell. As the excess total mass for producers is given by the deviation of the total mass balance $(F_w)_j + (F_n)_j$, the wetting phase outflow is obtained by inserting the total mass deviation into the phase velocity (5.15). Projected to the well surface with the normal $\mathbf{n}$, it is

$$\mathbf{u}_w \cdot \mathbf{n} = \frac{\lambda_w(s)}{\lambda_w(s) + \lambda_n(1-s)} \left(\mathbf{u}_T \cdot \mathbf{n} + \lambda_n(1-s)(\rho_w - \rho_n)\mathbf{n}^T\mathbf{K}\mathbf{g}\right) \tag{5.60}$$

The well is vertical, which implies that the grid is $\mathbf{K}$-orthogonal in the well due to the horizontally isotropic permeability. Consequently, $\mathbf{n}^T\mathbf{K} = 0$ in each point of the well surface and the gravity term disappears from the equation. Integration of the well surface in the cell j therefore returns the wetting phase outflow

$$\frac{\lambda_w(s_j)}{\lambda_w(s_j) + \lambda_n(1-s_j)} \left((F_w)_j + (F_n)_j\right). \tag{5.61}$$

For injection wells, we assume a pure wetting phase injection that must be added to the element. Thus, the wetting phase inflow is obtained by

$$\frac{\lambda_w(1)}{\lambda_w(1) + \lambda_n(0)} \left((F_w)_j + (F_n)_j\right) = (F_w)_j + (F_n)_j. \tag{5.62}$$

The transport step is now modified by adding the inflow and outflow terms to the fluxes, as described in Algorithm 5.8.

Algorithm 5.8: Modified upstream transport with injector and producer cells.

```
Input: $\mathbf{p}$, $\lambda_w^*$, $\lambda_n^*$, $\mathbf{s}$, $\Delta t > 0$
Output: s
traversal
    first DoF kernel:
    $(F_w)_j \leftarrow 0; (F_n)_j \leftarrow 0;$
    volume kernel:
    $(F_w)_j \leftarrow (F_w)_j + \sum_{i \in \mathcal{N}(j)} (\lambda_w^*)_{j,i}\mathbf{n}_{j,i}^T\mathbf{K}_{j,i}(-\nabla p_{j,i} + \rho_w\mathbf{g});$
    $(F_n)_j \leftarrow (F_n)_j + \sum_{i \in \mathcal{N}(j)} (\lambda_n^*)_{j,i}\mathbf{n}_{j,i}^T\mathbf{K}_{j,i}(-\nabla p_{j,i} + \rho_n\mathbf{g});$
    last DoF kernel:
    if is_producer(j) then
        $s_j \leftarrow s_j - \frac{\Delta t}{V_j}\left((F_w)_j - \frac{\lambda_w(s_j)}{\lambda_w(s_j) + \lambda_n(1-s_j)}\left((F_w)_j + (F_n)_j\right)\right);$
    else if is_injector(j) then
        $s_j \leftarrow s_j - \frac{\Delta t}{V_j}\left((F_w)_j - ((F_w)_j + (F_n)_j)\right) = s_j + \frac{\Delta t}{V_j}(F_n)_j;$
    else
        $s_j \leftarrow s_j - \frac{\Delta t}{V_j}(F_w)_j;$
    end
end
```

5.6 Conclusion: Reservoir Simulation on Sierpinski Grids

We consider the presented scenario fairly advanced as it is able to handle many physical phenomena that occur in porous media flow. At this point, it is able to simulate a basic 3D oil reservoir with two distinct phases, strongly heterogeneous, (partially) anisotropic permeability and porosity fields, and wells. With the given implementation, we will be able to conduct some numerical and performance analysis on simple benchmarks and simulate the SPE10 scenario to compare results on a more complete and production-ready example.

The model is has a few restrictions: miscible multi-component flow for gas modeling, fully anisotropic permeability data and multi-phase flow with more than two phases are not supported. Additionally, numerical methods are limited. Proper permeability upscaling methods, for example [23], are needed to obtain good coarse scale results and the Conjugate Gradients solver is not an optimal choice for the SPE10 scenario. The condition number of the linear system deteriorates for larger resolutions, an effect which is further amplified by the strong heterogeneity of the permeability data. Geometric multigrid solvers are suitable candidates due to their element-wise hierarchical formulation, fitting the recursive nature of the space-filling curve. They have already been applied successfully to Sierpinski traversals [13], however they are hard to implement in the framework and require substantial theoretical work for good convergence speed, considering especially the strongly heterogeneous permeability data.

6 Numerical Analysis of Porous Media Flow

Integrating all the components of sam(oa)^2 with its complicated traversal algorithm and the rich functionality for adaptive mesh refinement and parallelization is a tedious, error-prone task with many potential points of failure. In this chapter we will

- Verify the applied methods on a set of benchmarks for the porous media flow scenario and the tsunami wave propagation scenario by simulation of a set of problems with known analytic solutions. These problems are chosen to span a large test space to provide evidence of general convergence.
- Evaluate the quality of the solution by investigating error convergence rates on adaptively refined grids. Results are compared to an optimal approximation for a given grid, returning an absolute measure for the solution quality.
- Run production scenarios to compare key values with reference simulations or real measurements.

We will start with an analysis of some benchmark problems and move on to production scenarios afterwards.

6.1 Riemann Problems in Porous Media Flow

As benchmark scenarios we use General Riemann problems for porous media flow. This is a special class of problems with simple initial conditions that can be solved analytically to allow comparison with simulation results. We will shortly define general Riemann Problems here and look at some of their solution properties.

6.1.1 Solution to General Riemann Problems

A *general Riemann problem* for the Buckley-Leverett Equations (5.20) is an initial value problem with a single discontinuity on an infinite 1D domain without sources or sinks. The initial condition is chosen as

$$\begin{aligned} s &:= \begin{cases} s_l & \text{if } x < 0, \\ s_r & \text{else} \end{cases}, \\ u_T &:= const, \\ g &:= const. \end{aligned} \tag{6.1}$$

where s_l and s_r are left and right initial saturations respectively, u_T is the total velocity and g is the gravity. Note that the initial condition for the total velocity is redundant as the closed form of the system (5.14) without sources or sinks already implies

$$(u_T)_x = 0 \Rightarrow u_T = const.$$

Material parameters are defined as

$$\begin{aligned} \Phi &:= 0.2, & k &:= 5 \cdot 10^{-12}, \\ \nu_w &:= 3 \cdot 10^{-4}, & \nu_n &:= 3 \cdot 10^{-3}, \\ \rho_w &:= 312, & \rho_n &:= 258.64. \end{aligned} \tag{6.2}$$

The parameter space is chosen by the three relative permeability models (5.7), (5.8), and (5.9) combined with four different initial states, resulting in twelve different problems in total. By choosing fluxes with increasing polynomial order in the relative permeability terms, complexity of the model is gradually increased. The four initial states cover the most important cases of the General Riemann Problem defined in (6.1).

Analytical solutions were derived from the quasilinear form (5.17) for all cases, but the calculations will not be shown explicitly here due to their length. The derivation of the pressure solution is simple in 1D however, as the pressure is derived directly from total mass balance in the Buckley-Leverett Equations (5.20). Thus,

$$\begin{aligned} u_T &= (\lambda_w(s) + \lambda_n(1-s))k(-p_x) + (\lambda_w(s)\rho_w + \lambda_n(1-s)\rho_n)kg, \\ p_x &= \frac{\lambda_w(s)\rho_w + \lambda_n(1-s)\rho_n}{\lambda_w(s) + \lambda_n(1-s)} g - \frac{1}{\lambda_w(s) + \lambda_n(1-s)} k^{-1} u_T, \\ p|_{x_2} - p|_{x_1} &= \left(\int_{x_1}^{x_2} \frac{\lambda_w(s)\rho_w + \lambda_n(1-s)\rho_n}{\lambda_w(s) + \lambda_n(1-s)}\, dx \right) g - \left(\int_{x_1}^{x_2} \frac{1}{\lambda_w(s) + \lambda_n(1-s)}\, dx \right) k^{-1} u_T, \end{aligned} \tag{6.3}$$

where x_1, x_2 is any pair of positions in the domain. For the initial condition $p(x, 0)$, the left state and right state saturations are constant, therefore

$$p(x,0) - p(0,0) = \begin{cases} \left(\frac{\lambda_w(s_l)\rho_w + \lambda_n(1-s_l)\rho_n}{\lambda_w(s_l) + \lambda_n(1-s_l)} g - \frac{1}{\lambda_w(s_l) + \lambda_n(1-s_l)} k^{-1} u_T \right) \cdot x & \text{in} \quad x < 0, \\ \left(\frac{\lambda_w(s_r)\rho_w + \lambda_n(1-s_r)\rho_n}{\lambda_w(s_r) + \lambda_n(1-s_r)} g - \frac{1}{\lambda_w(s_r) + \lambda_n(1-s_r)} k^{-1} u_T \right) \cdot x & \text{in} \quad x > 0. \end{cases} \tag{6.4}$$

Density Driven Flow: non-wetting phase displacement

Consider the Riemann problem

$$\begin{aligned} s &:= \begin{cases} 1 & \text{if } x < 0 \\ 0 & \text{else} \end{cases}, \\ u_T &:= 0, \\ g &:= -9.80665. \end{aligned} \tag{6.5}$$

This scenario consists of two layers of fluids, where the lighter fluid (oil) lies on top of the heavier fluid (water). Top and bottom are defined by the direction of the gravity here. No fluid exchange happens and the analytic solution is stationary. Ultimately, the goal of adaptive refinement is therefore only to resolve the discontinuity in $x = 0$ as well as possible in order to get a good approximation of the solution. Figure 6.1 (right column) shows that analytic solution and numerical results of a 1D simulation agree well for all three permeability models. Figure 6.2 (bright green line) shows the L1 error reduction of the scenario with increasing maximum refinement depth (left column) and increasing numbers of Degrees of Freedoms (right column). The plots are separated into linear (top image), quadratic (middle image) and Brooks-Corey (bottom image) relative permeability models. Table 6.1 (second row) and Table 6.2 (second row) show the corresponding convergence rates for increased maximum refinement depth and increased number of degrees of freedom. In this scenario the convergence rate is always 1.0, implying that the error decreases inversely proportional to the mesh width $\epsilon(\Delta x) = O(\Delta x^{-1})$ and the number of Degrees-of-Freedom $\epsilon(N) = O(N^{-1})$. In comparison, a grid that is uniformly refined in both dimensions could at best achieve a convergence rate of 0.5.

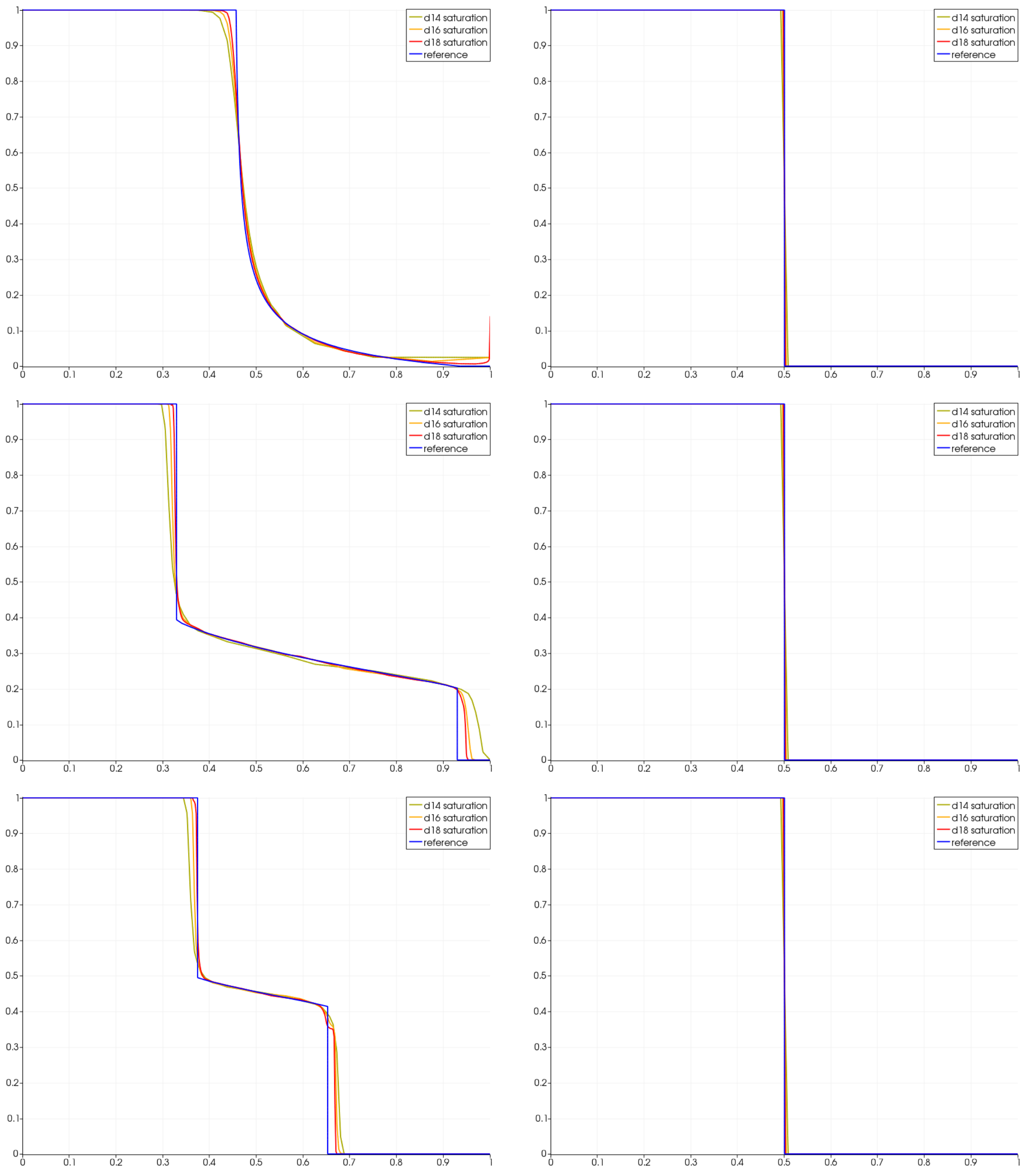


Figure 6.1: Riemann solutions to density-driven Buckley-Leverett flow: The plots show analytic solutions (in blue) and numerical solutions of the saturation for different maximum refinement depths (yellow: 14, orange: 16, red: 18), where points on the x-axis are 1D positions and the y-axis corresponds to wetting-phase saturation. Results for the three relative permeability models (5.7), (5.8)) and (5.9))) are shown from top to bottom. In the left column, gravity points to the right and a rarefaction wave at the interface results. The small peak at depth 16 in the top left plot is not an error, but caused by a wall condition on the right end of the domain. In the right column, gravity points to the left and the solution is stationary in all three cases.

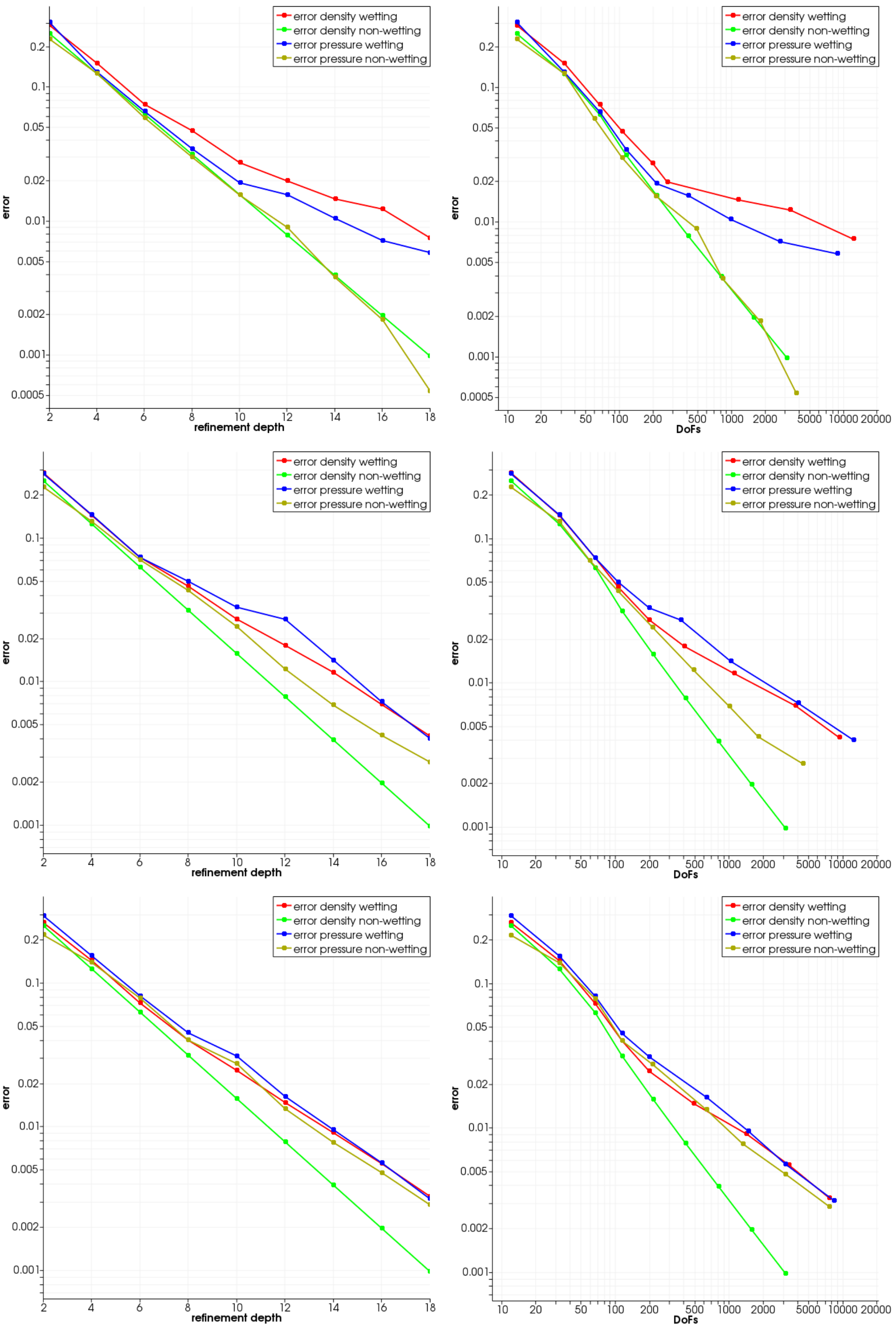


Figure 6.2: Error plots for all four considered Riemann problems. These double-logarithmic plots compare the change of the L1 error norm for increasing grid resolutions, denoted by the maximum refinement depth (left column) and the number of DoFs (right column) on the x-axis. Results are shown for linear (top row), quadratic (middle row) and the Brooks-Corey (bottom row) permeability model.

Riemann problem	linear model	quadratic model	Brooks-Corey model
density wetting	0.66	0.76	0.79
density non-wetting	1.00	1.00	1.00
pressure wetting	0.71	0.77	0.82
pressure non-wetting	1.09	0.80	0.78

Table 6.1: Convergence rate α of the L1 error norm for decreasing minimum mesh width Δx: $\epsilon(\Delta x) = O(\Delta x^{\alpha})$. The optimal value is 1. The minimum mesh width is computed from the maximum refinement depth d with the formula $\Delta x = 2^{-\frac{d}{2}}$. This table compares the solution asymptotically to the best possible approximation with the same minimum mesh width Δx, indicating the quality loss from adaptive coarsening. α was determined by a linear fit.

Riemann problem	linear model	quadratic model	Brooks-Corey model
density wetting	0.53	0.64	0.68
density non-wetting	1.00	1.00	1.00
pressure wetting	0.60	0.61	0.69
pressure non-wetting	1.05	0.75	0.67

Table 6.2: Convergence rate β of the L1 error norm for increasing numbers of DoFs: $\epsilon(N) = O(N^{-\beta})$. Grids that are uniformly refined in both dimensions achieve at best a value of 0.5 in 2D, the optimal value is 1. This table shows how well Degrees-of-Freedom contribute to the solution, indicating the overall quality of adaptive refinement and coarsening. β was determined by a linear fit.

Density Driven Flow: wetting phase displacement

We consider a similar Riemann problem now, but this time gravity has the opposite direction, i.e.,

$$\begin{aligned} s &:= \begin{cases} 1 & \text{if } x < 0 \\ 0 & \text{else} \end{cases}, \\ u_T &:= 0, \\ g &:= 9.80665. \end{aligned}$$

In contrast to the previous Riemann problem, the heavier fluid lies on top of the lighter fluid now. Inevitably, they will swap positions, so the saturation in $x < 0$ will decrease and the saturation in $x > 0$ will increase.

Figure 6.1 (left column) compares the analytic solution to the numerical results of a 2D simulation for variants of the relative permeability. The upwind solver is able to obtain good results for all problems, but some deviations from the numerical solution appear with higher resolutions. Still acceptable convergence is achieved with a convergence rate of at least 0.53 Table 6.1 (first row) and Table 6.2 (first row). The result improves with increasing nonlinearity of the relative permeability model, which is due to the formation of shock waves that are captured much better by adaptive refinement than rarefaction waves. For the linear model, a saturation peak at $x = 1$ is observed (Figure 6.1, top left image). The reason is not an error but the boundary that acts as a wall condition for the fluids. The main reason for the suboptimal convergence rate of all models is *viscous fingering*, e.g. [4]. To explain in detail, this effect is a physical instability at the fluid interface that occurs when the viscous phase is replaced by the non-viscous phase. In the first time step, the fluxes at the interface are identical. However, due to coarsening, the dual cells at the interface have different volumes and therefore the net updates produce different saturations (see Figure 6.3). Higher saturations locally increase the mobility of the non-viscous phase. The non-viscous phase is accelerated in non-viscous regions and the viscous phase is slowed down in viscous regions. The error is therefore amplified. Eventually the interface is visibly deformed, resulting in the viscous fingers. As the effects are caused purely by a discretization error, they are undesired but hard to avoid. Increased diffusion would solve this issue, for example by reducing the time step size or by application of diffusive flux solvers (Lax-Friedrichs, etc.), but it would also reduce the quality of the result.

In addition to the saturation we will also evaluate the pressure solution for this scenario as in this special case, the upwind solution is different from the analytic solution as explained in Section 5.3.2. According to (6.4), the initial pressure is

$$p(x, 0) = p(0, 0) + \begin{cases} \rho_w g x & \text{in} \quad x < 0 \\ \rho_n g x & \text{in} \quad x > 0 \end{cases}$$

We use the pressure difference

$$p(\tfrac{1}{2}, 0) - p(-\tfrac{1}{2}, 0) = \tfrac{1}{2}(\rho_w + \rho_n)g = 0.4058...$$

in pounds-per-square-inch (psi) as reference data and compare it to simulation results for different minimum mesh widths in the maximum norm. Table 6.3 shows the results for decreasing minimum mesh widths and increasing number of DoFs. The pressure error has a convergence of order 1.00 in the maximum norm in both cases. Therefore, considering that a linear approximation is used, an optimal convergence rate is achieved.

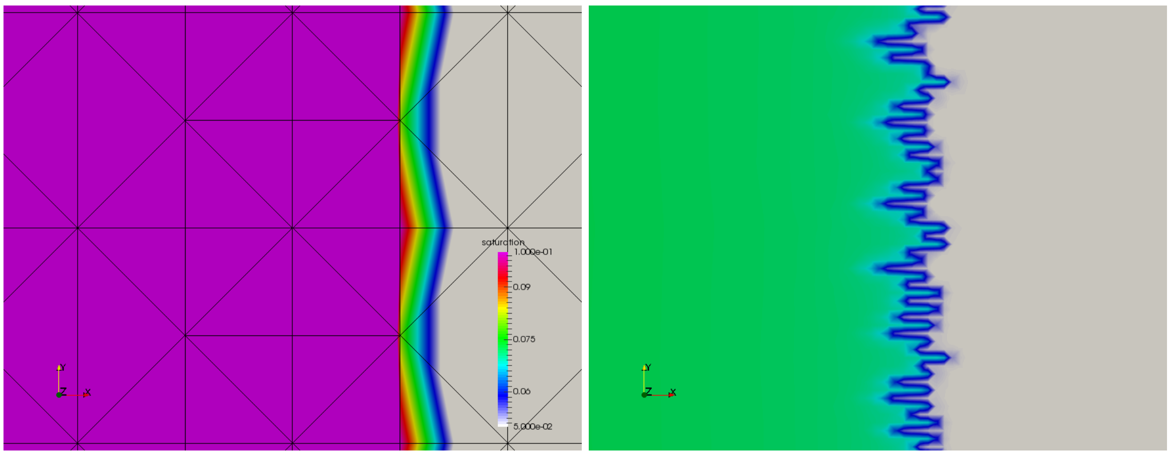


Figure 6.3: Density-driven flow in 2D: The left image shows the saturation (gray ≤ 0.05, pink ≥ 0.1) on the phase interface after the first time step. Due to coarsening at the interface, saturation is not homogeneous. This discretization error causes viscous fingering, an inhomogeneous saturation front (right image) in the solution (gray $= 0.0$, blue $= 0.2$, green $= 0.5$)

.

Minimum mesh width	2^{-1}	2^{-2}	...	2^{-7}	2^{-8}	2^{-9}
Degrees of Freedom	12	32	...	802	1572	3118
Pressure error (L_∞)	0.032	0.015	...	0.00047	0.00023	0.00012

Table 6.3: Maximum error norm of the initial pressure for density-driven flow with wetting phase displacement and a linear relative permeability model. The table shows results for decreasing minimum mesh widths $\Delta x = 2^{-\frac{d}{2}}$ and increasing Degrees of Freedom N, where d is the maximum refinement depth. In both error norms $\epsilon(\Delta x) = O(\Delta x^{\alpha})$ and $\epsilon(N) = O(N^{-\beta})$, the convergence rates α and β are 1.00.

Pressure Driven Flow: non-wetting phase displacement

The third case is a scenario without gravity but a non-zero total velocity $u_T < 0$. More precisely,

$$\begin{aligned} s &:= \begin{cases} 1 & \text{if } x < 0 \\ 0 & \text{else} \end{cases}, \\ u_T &:= -0.54.., \\ g &:= 0.0. \end{aligned}$$

The analytic solutions for this scenario are dominated by shocks as is shown in Figure 6.4 (right column). The linear relative permeability model (top right image) (5.7) produces a shock wave, the other models (middle right and bottom right image) produce an additional rarefaction wave for low saturations. The error measurements in Tables 6.1 and 6.2 suggest a (theoretically impossible) superlinear error convergence for the linear model. The reason for this number is that the convergence rate is only estimated by comparison of test results from coarse and fine resolutions. Here the coarse grid result is suboptimal and the difference of fine grid and coarse grid solution is insufficient to get a precise convergence rate. In Figure 6.2 (top left image, yellow line) this is visible as the error decreases at a constant rate until it suddenly increases at $d = 16$. For the other permeability models, the convergence rate is still good but slightly worse, which is again due to the addition of a rarefaction wave and the effect of viscous fingering.

Pressure Driven Flow: wetting phase displacement

The last case investigates the Riemann problem

$$\begin{aligned} s &:= \begin{cases} 1 & \text{if } x < 0 \\ 0 & \text{else} \end{cases}, \\ u_T &:= 0.54.., \\ g &:= 0.0. \end{aligned}$$

where the total velocity is positive and no gravity is present. The behavior of the analytic solution with increasing nonlinearity is exactly opposite to the third case as is shown in Figure 6.4 (left column). The linear relative permeability model (top right image) (5.7) produces a rarefaction wave, the other models (middle right and bottom right image) produce an increasingly strong additional shock wave. Convergence rates improve for increasing nonlinearity as seen in Tables 6.1 and 6.2. In Figure 6.2 the same behaviour is observed (all images, blue line) as the curves get steeper with increasing nonlinearity.

Conclusions from Benchmark Analysis

The observations made from solving the four types of Riemann problems are two-fold:

On the one hand, adaptive refinement is able to track shock waves optimally. This is shown in particular for pure-shock wave solutions, which all have a convergence rate of 1. The other problems are clearly solved better, the stronger the shock wave component of the solution is.

On the other hand, resolution of rarefaction waves is problematic. Viscous fingering causes a self-amplifying error at the wave front. Also, features are not one-dimensional as in the case of a shock wave and therefore it is harder for the refinement criterion to decide which cells should be refined and which ones coarsened.

Still, for all examples the achieved convergence rates were better than what a uniformly refined Cartesian grid could theoretically achieve. This is a strong indication that more complex examples will also benefit from adaptive refinement and production scenarios are worth investigating.

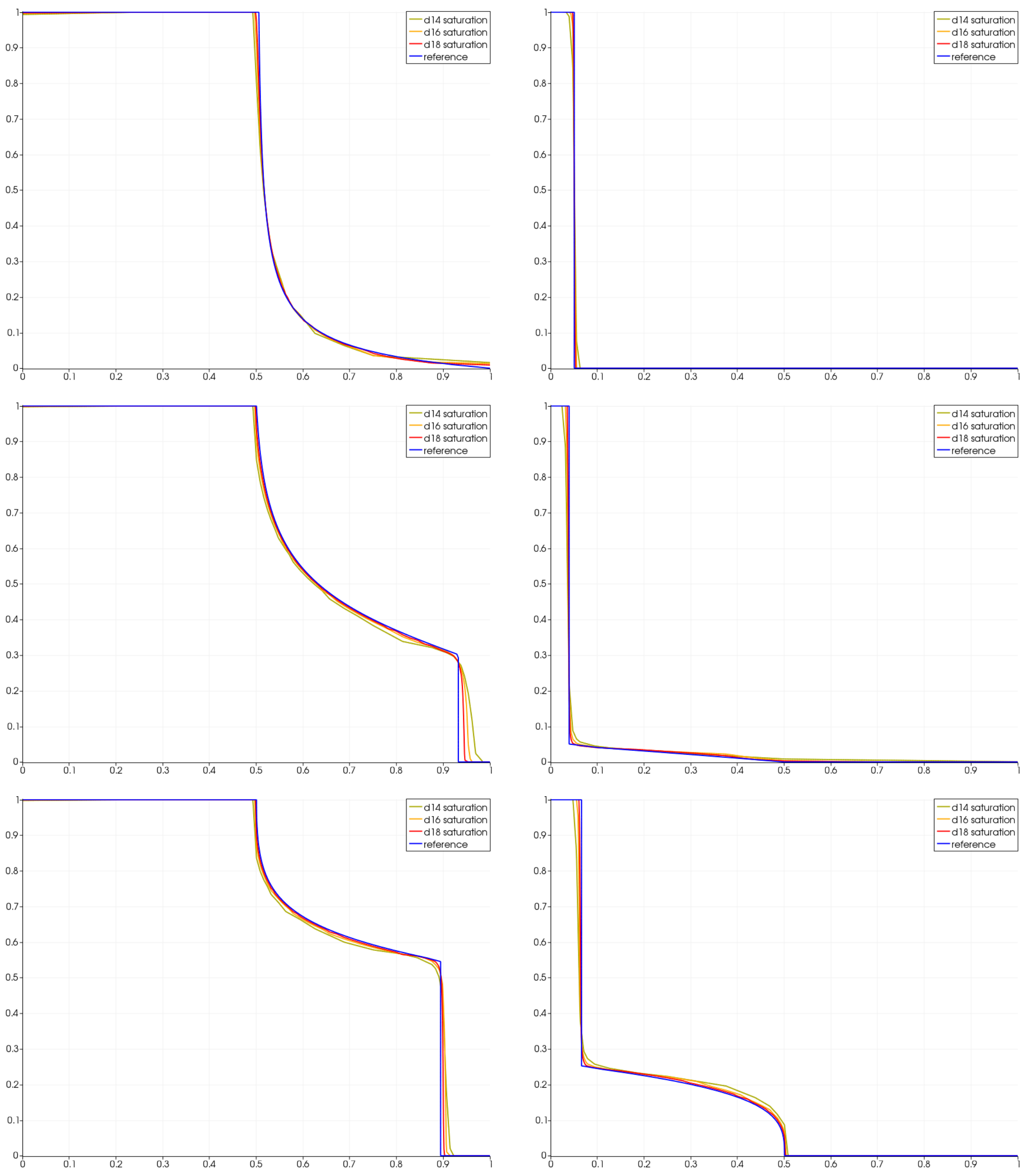


Figure 6.4: Riemann solutions to pressure-driven Buckley-Leverett flow: The plots show analytic solutions (in blue) and numerical solutions of the saturation in space for different maximum refinement depths (yellow: 14, orange: 16, red: 18). Results for the three relative permeability models (5.7), (5.8) and (5.9) are shown from top to bottom. In the left images, the force acts to the right and the wetting phase replaces the non-wetting phase. In the right images, the force acts to the left and the non-wetting phase replaces the wetting phase.

6.2 Well Model Accuracy

In order to verify the well model implementation introduced in Chapter 5.5.2 we set up a simple 2D test case on a square domain with heterogeneous permeability that consists of four producer wells with bottom hole pressure 0 in each corner and an injection well in the center of the domain that injects water at a constant injection rate. Four distinct permeability values are chosen and assigned to each quadrant of the domain, resulting in an axis-symmetric pressure solution with asymmetric fluxes. This test case will only consider the pressure correction term (5.59) and neglect the equalization (5.57) and transport step boundary conditions (5.61) and (5.62). This is a valid restriction, as an incorrect well pressure equalization will prevent a solution to the linear system (5.38) and incorrect boundary conditions will violate the mass balance and cause quick failure in the transport step (5.24). In contrast, an erroneous pressure correction term will not return an unphysical solution as it corresponds to a different pressure boundary condition.

Figure 6.5 (top) visualizes the permeability distribution in the domain and the pressure solution. Convergence of the injector pressure over increased refinement depth (bottom left image) is slow without the correction term and depends heavily on the well radius r_w. When the well is over-resolved, the target pressure is overshot (this would usually be handled however). With pressure correction (5.59), a good convergence speed is achieved and the corrected solution matches the analytic solution near the production well (bottom right image).

6.3 Comparison to SPE10 Reference Data

The SPE10 benchmark project provides lots of reference data that includes production rates, water cuts, average pressure and cumulative oil production for the four producers P1 to P4 over time.

In order to validate the implementation, the code was executed on fully refined grids with 85 layers and maximum depths 10, 12, 14 and 16, corresponding to 50k, 170k, 560k and 2.2M elements in the final grid. The wall-clock time of the full scenario was 1h 12min for 2.4M time steps on 4 Haswell nodes of the SuperMUC Phase II system [119]. Figure 6.7 shows a plot of oil production rates for maximum depth 10, 12, 14 and 16. On the bottom axis the simulation time is plotted, The solid lines are oil production rates in $\frac{BBL}{d}$ for the field and for producer P1. In the first 85 days, production rates are almost constant, then water starts to reach the producers, increasing the water cut and causing a rapid decline in the production rate. Comparison of the field oil rate to reference results in the top curves shows good agreement for all three simulations. Slight deviations are due to the decreased resolution. Deviations are more prominent for producer P1. This is caused by a discretization error in the total production rate of a producer. In contrast, this error does not occur in the total production rate of the field, as it is determined by the injection rate of 5,000 $\frac{BBL}{d}$ and thus affected only by numerical error. The water cut of producer P1 in Figure 6.8 (bottom image) shows that for simulation depths 10 and 12, water arrives too late compared to the reference simulations. As time progresses, the lines converge however.

Figure 6.6 shows the saturation profile in the domain over time, where water spreads from the injection well, causing a pressure-induced flow towards the production wells. It is clearly visible that propagation speed differs strongly in the different layers due to the strong heterogeneity in material permeability. On four Haswell nodes of the SuperMUC Phase II system [119], the scenario is solved roughly in an hour, executing 2.4 million explicit time steps.

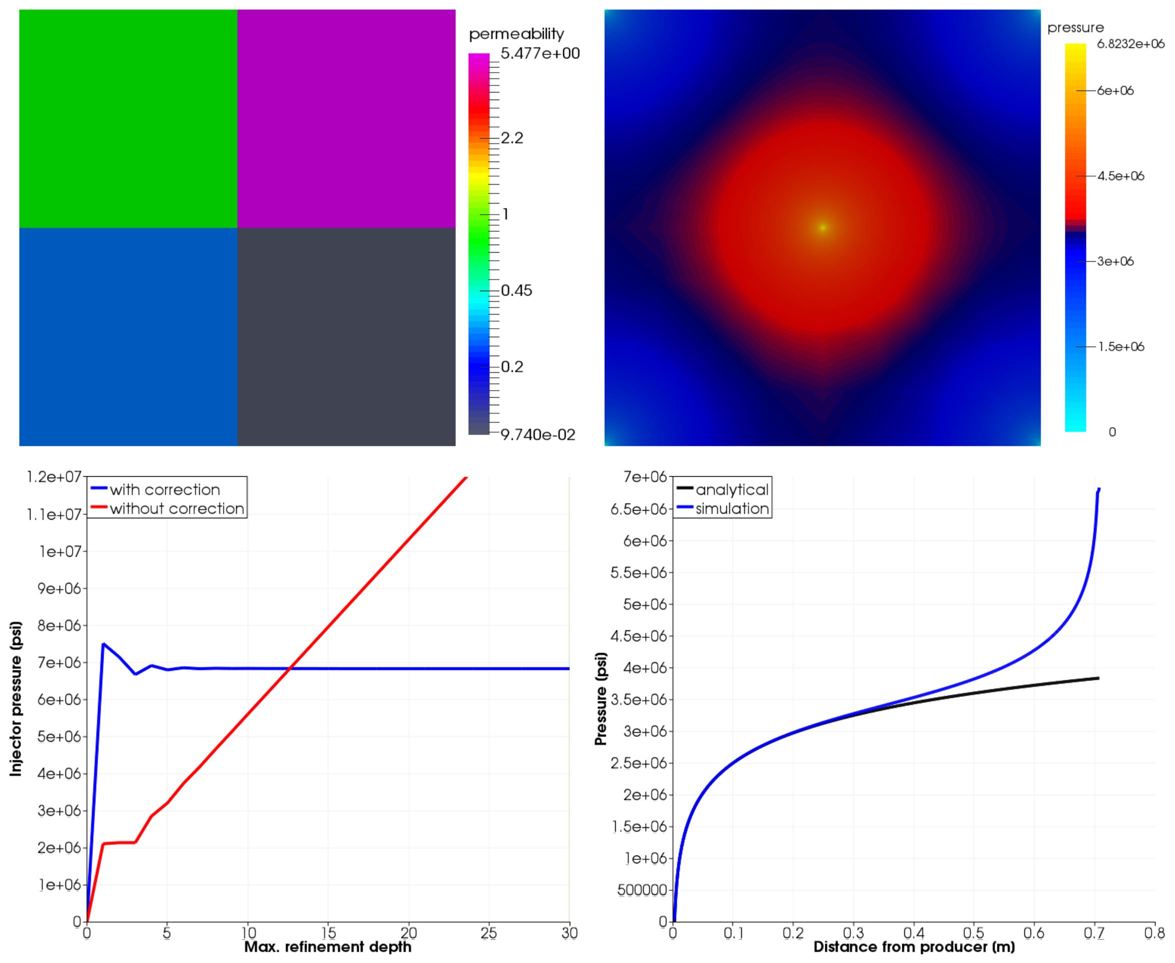


Figure 6.5: Validation of the well pressure correction term (5.59) for a simple heterogeneous 2D five-spot problem with uniform permeabilities in each quadrant of a $1m \times 1m$ square domain (top left image). Four producers are located at the corners of the domain and one injector is placed in the center. The top right image shows the symmetric pressure solution on a fine grid with maximum depth 30. In the bottom left image, the injection pressure is plotted at increasing refinement depths with (blue) and without (red) pressure correction term, where the corrected method is observed to converge quickly. A comparison of corrected and analytic single-well solution (bottom right image) over increasing distance to the producers shows good agreement near the well. At a distance of 0.4 m and beyond, the analytic solution does not hold any longer.

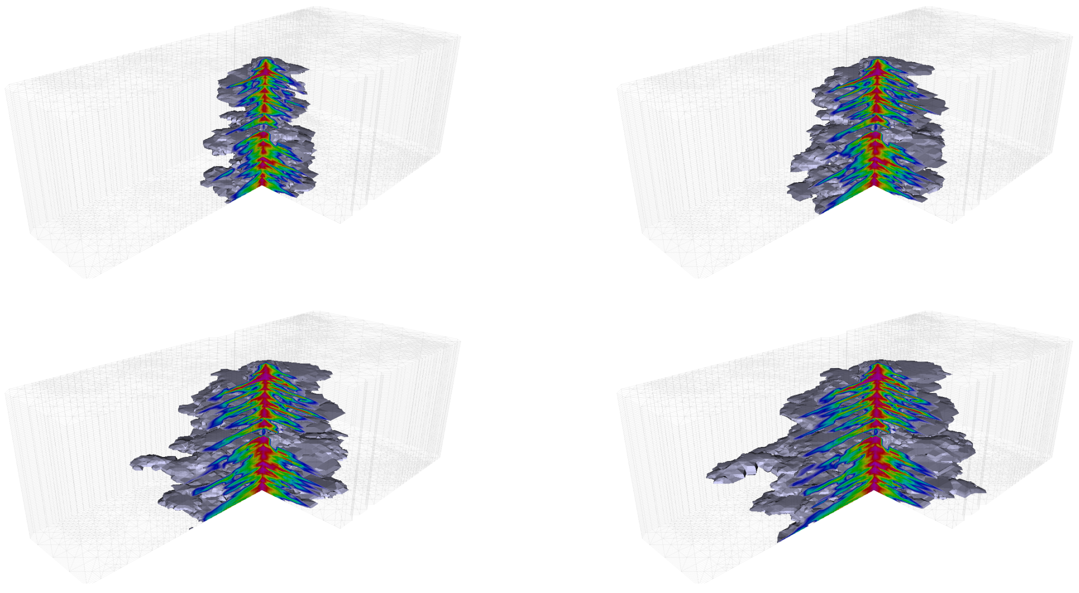


Figure 6.6: SPE10 3D saturation profile at 25, 50, 75 and 100 simulation days. The lower right corner has been clipped for better visibility.

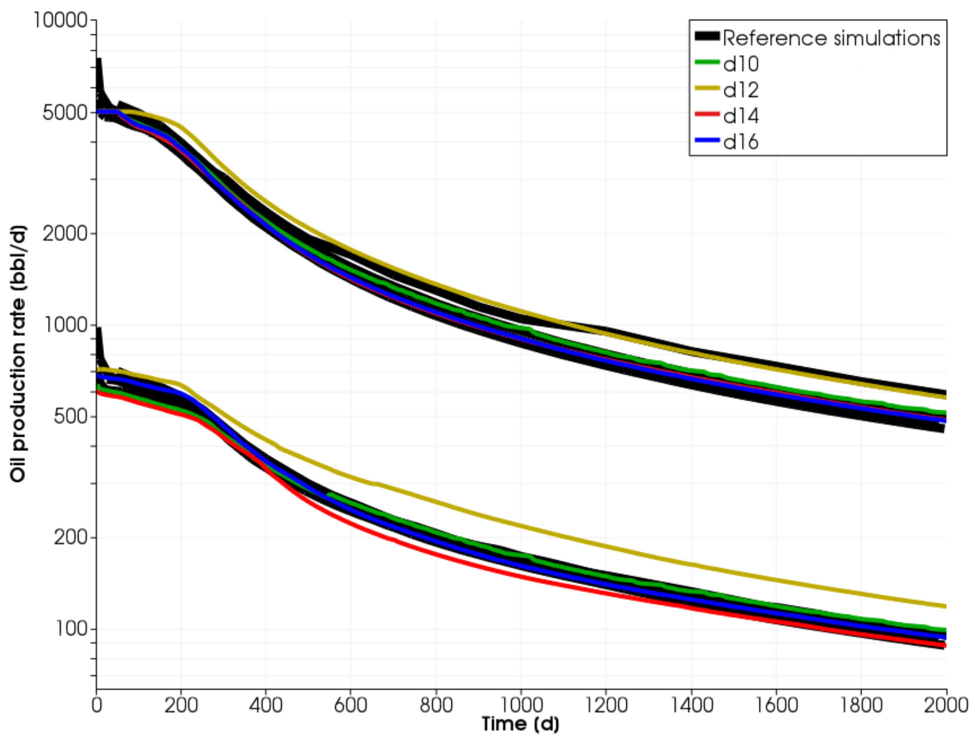


Figure 6.7: SPE10: Logarithmic plot of field (top lines) and producer P1 (bottom lines) oil production rates for simulations with maximum depths 10 to 16 over 2000 days of production. Fine scale reference data from the original SPE10 comparison project (black) is added for comparison.

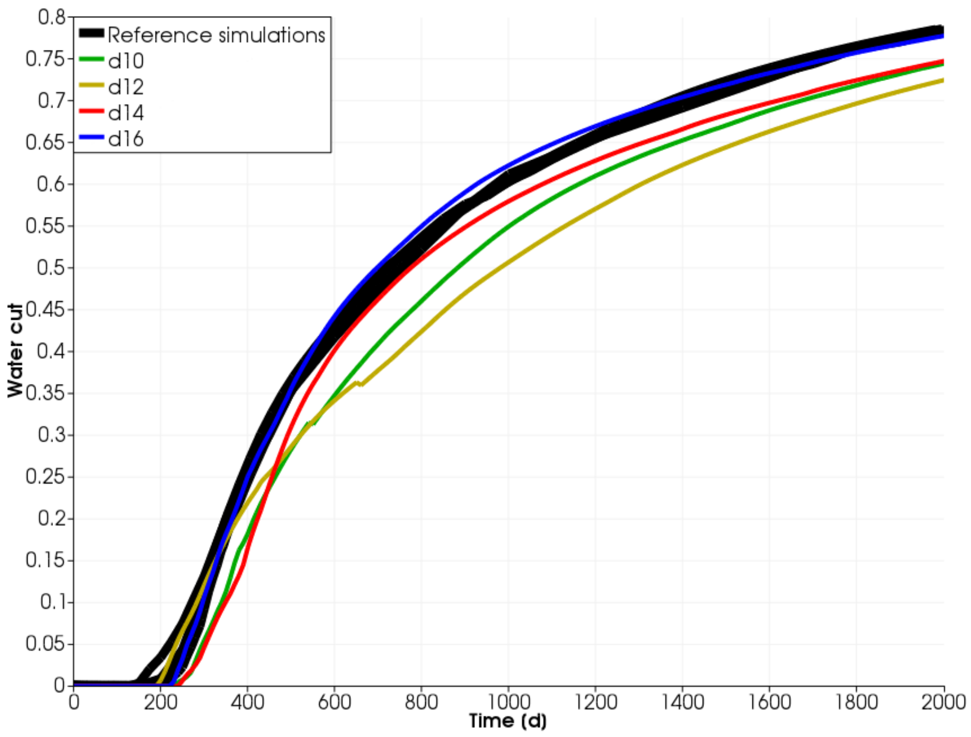


Figure 6.8: SPE10: Producer P1 water cut for simulations with maximum depth 10 to 16 over 2000 days of production. Fine scale reference data from the original SPE10 comparison project (black) is added for comparison.

6.4 Conclusion: Riemann Problems and the SPE10 Benchmark

In this chapter we verified that the numerical components of the porous media flow scenario work as expected. The nonlinear systems are set up and solved correctly as the Riemann solutions and well model comparisons showed. The SPE10 scenario was solved with satisfying accuracy, although some deviations still exist, especially on grids that do not fully resolve the problem. More work on permeability and porosity upscaling might be necessary to achieve better results.

7 Performance of Porous Media Flow

In this chapter, we will analyze performance of the porous media flow scenario with a parameter choice and an environment close to the conditions of a production run.

The target platform will be the SuperMUC supercomputer [119], whose subsystem specification is given in Table 7.1. The system is located at Leibniz Supercomputing Center in Munich, Germany and consists of two Petascale CPU clusters, a fat node cluster and a hybrid many-core cluster, of which the latter is neglected for this analysis.

Most of the performance tests were executed on the SuperMUC thin node cluster (phase 1), which features 9,216 light-weight nodes with eight cores on two sockets each, allowing concurrent execution on up to 147,456 cores. For single node memory bandwidth tests, we additionally used the SuperMUC fat node and Haswell nodes. Each fat node consists of four ten-core sockets and each Haswell node contains 28 cores on two sockets. A cache-coherent Non-Uniform Memory Access (ccNUMA) layout allows for shared memory parallelization in each node. Inter-node communication is realized by a peer-to-peer pruned-tree topology.

sam(oa)^2 was compiled with the Intel Fortran Compiler version 13.1 or higher. Two MPI libraries are available on SuperMUC: A customized MPI version by IBM and Intel MPI version 4.1 and above, both of which deliver comparable performance. For large scale tests we mainly used IBM MPI, for small scale tests Intel MPI. Arithmetic operations were computed in double precision.

As the porous media flow scenario is computationally cheap and rarely requires remeshing, the focus for performance analysis lies on determining the overhead of static grid traversals and communication. We will compute the traversal overhead by comparing memory throughput and floating point operations per second to the peak capabilities of a single node. Communication will affect the scenario mostly when it is executed on a large number of cores. Hence, we executed scalability tests to determine the impact of increased parallelism. Some of the performance tests presented here have already been published in [75, 77].

7.1 Efficiency Measures

For performance analysis, we are mostly interested in analyzing single node efficiency and parallel overhead of sam(oa)^2. Application-specific scalability indicators such as the number of linear solver iterations are important for production scenarios but will be neglected as they do not reflect parallel performance of the framework.

Since the resolution in dynamically adaptive meshes is not controlled by a static depth value but by dynamical, local criteria, it is hard to create a mesh with a fixed size that still reflects a production environment. Classical weak scaling is therefore difficult to realize, as the default efficiency measure $\frac{t(1)}{t(p)}$ compares execution times $t(p)$ for a test suite of scenarios with increasing mesh size and parallelism p, requiring constant numbers of elements per core.

Hence, instead of execution times, we use a different measure to evaluate performance based on the

Subsystem	Fat Nodes	Thin Nodes	Haswell Nodes
Processor	Intel Westmere Xeon E7-4870	Intel Sandy Bridge Xeon E5-2680 8C	Intel Haswell Xeon E5-2697 v3
Cores per node	40	16	28
Number of nodes	205	9216	3072
Memory bandwidth per node [GB/s]	136	102	137
Double precision ops per node [GFlop/s]	384	346	1165
Double precision ops total [TFlop/s]	79	3190	3580
Islands	1	16	6
Nodes Per Island	205	512	512
Interconnect	Infiniband QDR	Infiniband FDR10	Infiniband FDR14
Inter-Island Interconnect	none	4:1 Pruned Tree	4:1 Pruned Tree

Table 7.1: The SuperMUC CPU clusters and their specifications [119]

number of element touches per second per core:

$$\mu(p) := \frac{ET(p)}{p\,t(p)}. \tag{7.1}$$

$ET(p)$ is the total number of element touches, where each grid traversal counts a single touch per element. p is the degree of parallelism, i.e. the total number of concurrent threads over all MPI processes, and $t(p)$ is the measured wall clock time, depending on p.

Ideally, $\mu(p)$ is now constant for increased parallelism, but will include overhead from load balancing and communication. Hence, it is more suitable to measure weak scaling efficiency than $\frac{t(1)}{t(p)}$ as it allows variations in the grid size while still returning comparable numbers.

In a strong scaling setting, the number of grid traversals and element touches should be constant over all test runs as we ideally solve the same problem for each test. The only difference is the varying degree of parallelism p. Hence, $ET(p) = ET(1) = \mu(1)\,t(1)$ and (7.1) can be written as:

$$\mu(p) = \frac{t(1)}{p\,t(p)}\,\mu(1) \qquad \text{if } ET(1) = ET(p). \tag{7.2}$$

$\mu(p)$ therefore measures strong scaling efficiency $\frac{t(1)}{p\;t(p)}$ scaled by the constant $\mu(1)$.

In practice, there may be minor differences in the grid during strong scaling as coarsening is restricted to cells of the same section as mentioned in Chapter 4.5.1. With increased parallelism, the grid is split into more sections and a pair of cells is more likely to be separated by a section boundary, which prevents coarsening of the cells. Additionally, numerical error may cause differences in the last significant bits as the order of arithmetic operations is not necessarily the same for serial and parallel runs. This can lead to slightly varying numbers of linear solver iterations for the Conjugate Gradients solver, which is described in Section 5.3.4. These nondeterministic effects are small enough to be insignificant for scalability tests however.

We will also investigate performance in more detail by splitting up computation into its components. Here, a suitable measure is the *normalized component time per element per core*

$$\tau(c,p) = \frac{p\,t(c,p)\,s(c)}{ET(c,p)}, \tag{7.3}$$

where $t(c, p)$ is the total wall clock time spent in the component c with degree of parallelism p and $ET(c, p)$ is the total number of element touches in component c depending on p. Adding up the component times $t(c, p)$ and element touches $ET(c, p)$ returns $t(p)$ and $ET(p)$ respectively:

$$t(p) = \sum_c t(c, p) \quad \text{and} \quad ET(p) = \sum_c ET(c, p) \tag{7.4}$$

$s(c)$ is a typical number of grid traversals per component, empirically chosen and independent of the actual number. This normalization is necessary to provide comparable data for linear solver performance, whose iteration number varies with the problem size. Eventually, τ measures scalability of each component while considering how much time is usually spent in each component, but smoothing differences that occur in individual runs. With increased parallelism, we expect the values to grow. In a theoretical, perfectly scaling code, they would remain constant.

7.2 Traversal Overhead on a Single Node

The first goal of this analysis is to determine how close sam(oa)^2 approaches the theoretical performance of a single node in order to define a performance baseline for scalability. We will therefore execute a throughput performance evaluation in order to determine the ratio of actual performance to peak performance. Next, scalability on a single node is analyzed with different parallelization implementations.

7.2.1 Node Level Roofline Analysis

In order to get a rough idea of performance limits, we will first conduct a Roofline analysis [128] with the goal of investigating whether performance is bound by memory throughput, computation or other factors.

Figure 7.1 shows the results of an analysis for a 2D scenario and for 3D scenarios with increasing numbers of vertical layers, all of which correspond to the model in Chapter 5.2.3. Note that we already use the fused Conjugate Gradients solver from Chapter 5.4.1 to improve computational intensity. When the number of vertical layers is increased, computation, memory traffic and communication increase, however the number of stack-&-stream data fetches during traversal remains constant. Hence, floating point performance appears to be mainly limited by memory bandwidth, considering that the performance of sam(oa)^2 is fairly close to the output of the STREAM benchmark [73].

Table 7.2 displays the memory performance in detail for increasing numbers of layers. Memory throughput in sam(oa)^2 was estimated by summing up the amount of memory allocated for all cells and all vertices once per traversal. Hence, we assume all data is touched at least once per traversal to get a conservative estimate for the total amount of memory traffic. When this number is divided by the wall clock time for all traversals, an estimate for the memory throughput is returned that can be compared to the STREAM benchmark. We used a modified implementation of STREAM here that counts only a single access to an array, while executing an in-place copying operator. This pattern matches the stream access of sam(oa)^2 and returns a useful benchmark value. Overhead from cache misses and access to the stacks is not counted in sam(oa)^2, hence the ratio of the numbers returns a measure of memory efficiency.

Memory throughput starts at 36% of STREAM performance in 2D and converges to 68% with increased number of vertical layers, meaning sam(oa)^2 does not achieve the full bandwidth in 3D, but converges to a fairly high value. In the second row of Table 7.3 a comparison of memory throughput on all three SuperMUC clusters shows that sam(oa)^2 approaches the measured throughput by STREAM better on the other systems, reaching almost 82% on the Haswell nodes and 86% on the fat nodes. The reason for this discrepancy is that a thin node has the highest memory bandwidth per core with 6.4 GB/s, whereas the fat nodes have 3.4 GB/s per core, and the Haswell nodes have 4.9 GB/s per core. Hence, the

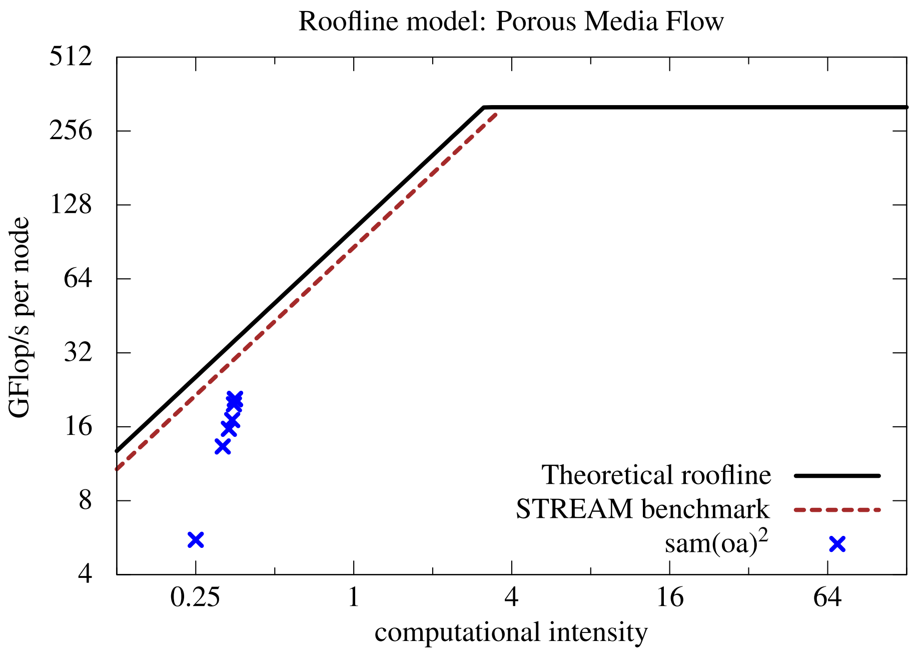


Figure 7.1: Roofline analysis of the porous media flow scenario on a single SuperMUC thin node. Floating point performance of sam(oa)^2 is compared to the theoretical peak performance for different numbers of vertical layers in the 3D model. Higher Flop/s correspond to higher numbers of layers. The output of the STREAM memory benchmark [73] is plotted as a ceiling in the graph.

Memory TP in GB/s (Ratio)	2D	1 layer	16 layers	64 layers	85 layers
STREAM	43 (100%)				
sam(oa)^2	15 (36%)	16 (38%)	25 (59%)	29 (68%)	29 (68%)

Table 7.2: Memory throughput of the porous media flow scenario in sam(oa)^2, measured for runs with 40M elements in 2D and 3D (1 to 85 layers) on a single SuperMUC thin node, compared to the STREAM benchmark.

thin nodes read and write data faster than the other nodes. Traversal logic and computation have a higher impact compared to the fat nodes and the Haswell nodes, which process memory slower while at the same time having more FLOP/s available in total per node. Hence, we can conclude that performance is limited mostly by memory throughput, however some influence of traversal logic and computation is visible, too.

Adding Lazy Broadcasts in Shared Memory

The memory throughput comparison is also suitable to investigate how lazy broadcasts in shared memory according to Section 4.3 affect performance. By removing redundant computation, we observe that the effective memory throughput increases on all three systems consistently by 15% to 17% for this scenario, reaching close to 100% on the fat nodes and the Haswell nodes.

Memory TP in GB/s (Ratio)		Fat Node	Thin Node	Haswell Node
STREAM		56 (100%)	43 (100%)	61 (100%)
sam(oa)^2	original	46 (82%)	29 (68%)	53 (86%)
	with lazy broadcasts	54 (97%)	34 (78%)	61 (100%)

Table 7.3: Memory throughput of the porous media flow scenario in sam(oa)^2 with 85 layers on a single fat node, thin node and Haswell node of the SuperMUC system. The scenario size was chosen sufficiently high to exceed the cache size on each system. Also compared is the addition of lazy broadcasts in shared memory, see Chapter 4.3.

7.2.2 Scalability on a Shared Memory Compute Node

Our next test is scalability on a single SuperMUC thin node. A SuperMUC thin node consists of two NUMA sockets with eight cores each, suitable for strong scaling tests with pure OpenMP, pure MPI, and hybrid OpenMP+MPI parallelization. We will test all three variants to try and identify potential scalability issues.

Figure 7.2 shows a component breakdown for the 2D channel flow scenario in Chapter 5.2.2 to evaluate the parallel efficiency from 1 to 16 cores.

The numerical components mentioned in the plot are described in Chapter 5.3.2, where

- “Permeability“ is the setup of the linear system which requires evaluation of permeabilities (Chapter 5.3.3).
- ”Pressure Solver” is the Conjugate Gradients solver (Chapter 5.3.4).
- “Gradient“ is the time step computation from evaluation of the pressure gradients (Chapter 5.3.5).
- ”Transport“ is the discrete transport step (Chapter 5.3.2).
- ”Error Estimate“ is the error indicator for adaptive refinement and coarsening (Chapter 5.3.6).

Additionally, structural components of sam(oa)^2 are integrated in the graph:

- ”Conformity” means the conformity correction traversals (Chapter 4.5.1).
- “Adaption“ is adaptive refinement and coarsening (Chapter 4.5).
- ”Neighbor Search“ is the neighbor data structure update. (Chapter 3.4.2).
- ”Load Balancing“ is the MPI load balancing component of sam(oa)^2 (Chapter 3.2).

Clearly, the pressure solver has the most impact on performance, as it dominates the other components. For the plot, we assumed 10 linear solver iterations per time step, applying the metric described in Section 7.1, to normalize the linear solver time for better comparability. Usually, the number is higher. All other components are counted only once per time step, which is accurate in most cases. With four sections per core, performance for pure OpenMP, pure MPI and hybrid execution is mostly similar. MPI performs a bit worse, which is more visible with eight sections per core. This is mostly due to overhead from process management and MPI calls. All components scale appear to scale well on a single node with only minor impediments.

A second component analysis in Figure 7.3 shows how ratios and total computation time per element change with increased number of layers for the 3D scenario. While the total time doubles from 2D to 3D

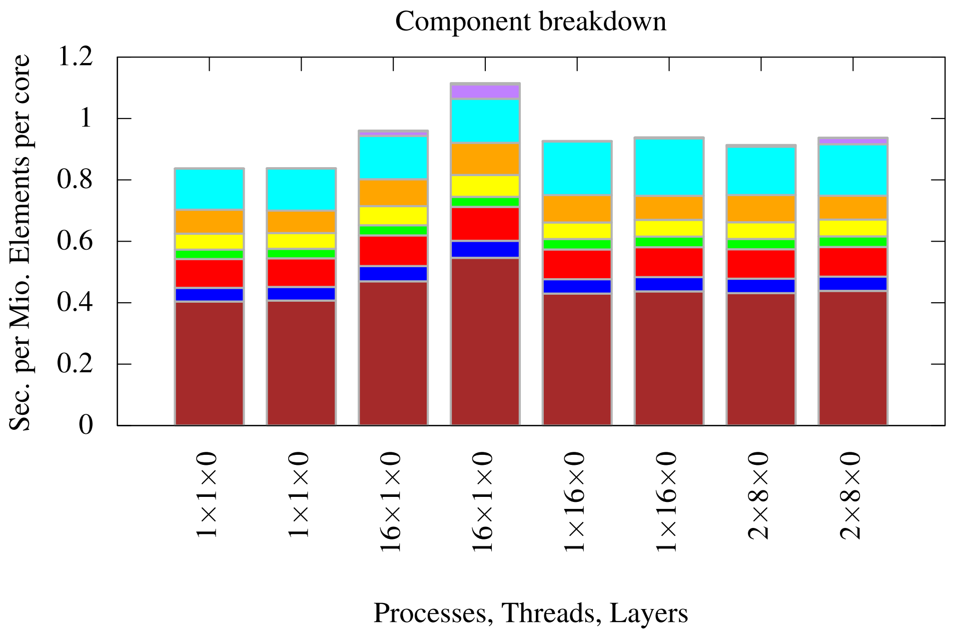


Figure 7.2: Strong scaling on a single thin node with MPI-only, OpenMP-only and hybrid parallelization. Two runs per configuration have been executed, one with four (left bar) and one with eight sections per core (right bar). OpenMP and hybrid parallelization perform best while MPI parallelization suffers from some overhead especially with higher numbers of sections.

(0 to 1 layer), it slowly decreases to a value similar to the 2D case. This behavior reflects the memory requirements per element well, which double from 2D to 3D. Since we need vertex columns of size $n+1$ for n layers of prism elements, twice as many vertices are needed for a single 3D layer than a 2D grid. However, as n approaches infinity, the ratio $\frac{n+1}{n}$ converges to 1 again like in the 2D case, explaining the decline in the plot with four layers and more.

7.3 Scalability on Multiple Distributed Nodes

In order to determine communication overhead we will use the single node performance analysis as baseline for absolute performance and determine parallel efficiency on large numbers of cores to find out how communication affects scalability.

For multi-node performance tests the SuperMUC thin nodes were used again. With 147,456 cores, they provide a suitable environment to test scalability in distributed memory. Our tests were conducted on up to 32,768 cores, which is a quarter of the full machine and the maximum possible without special permission for larger jobs.

For this study, we will investigate performance of both the 2D channel flow scenario and the 3D SPE10 scenario.

7.3.1 Scalability of the SPE10 Scenario

In this test, we will investigate scalability of the 3D scenario described in Chapter 5.2.3 and Chapter 5.5. The number of vertical layers was chosen to be 64. As local communication grows linearly with the number of layers but traversal overhead and global communication remain constant, scalability should

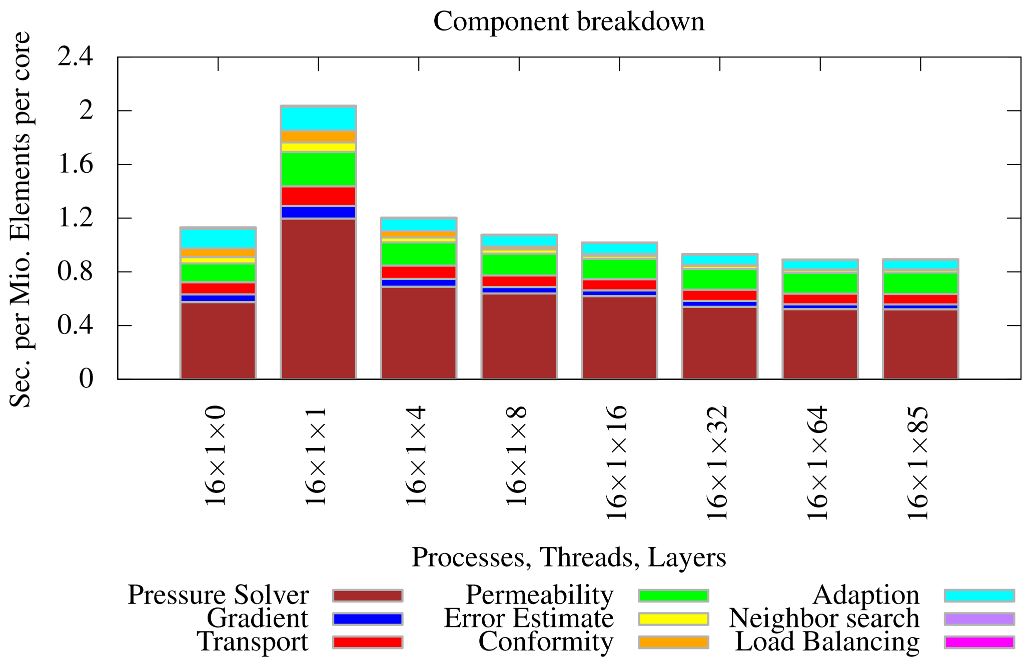


Figure 7.3: Component breakdown in 2D (0 layers) and 3D (1 to 85 layers): given is the time spent in a single element per time step, broken down by components. For this plot, the pressure solver was normalized to 10 iterations per time step.

be dominantly affected by local communication. We will show both a short weak and strong scaling study, as already presented in [75].

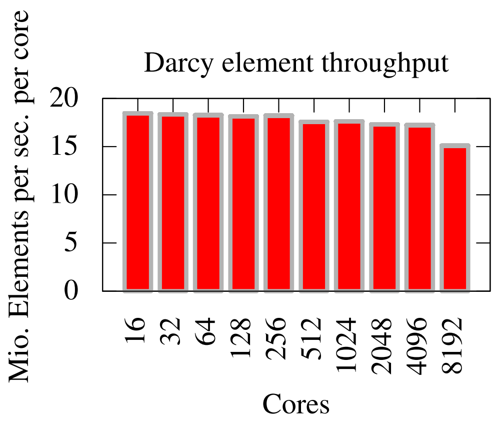
Weak Scaling

The purpose of a weak scaling test is to investigate how increase of problem size affects scalability, in order to predict performance of larger problems that may occur in future simulations.

For this study, the problem size was scaled from 500k elements on 1 core to 4G elements on 8192 cores. Figure 7.4 shows an excellent scalability with 86% from 1 to 8192 nodes and 91% from 16 to 8192 cores. Some performance loss occurs within a node due to the memory-bound nature of the problem. Once the memory throughput of the node is reached, it becomes the bottleneck of the simulation. On 8192 cores, the most prominent reason for performance loss is global communication. Compared are an optimal chains-on-chains solver and midpoint approximation for load balancing (see Chapter 3.2). As the optimal solver is inherently serial, the scenario does not benefit from it at large numbers of cores in this case.

Strong Scaling

As the SPE10 scenario consists only of a few million elements, scalability on a large number of cores is not achievable for a memory-bound solver. Hence, we over-resolve the scenario to a size of 80M elements and investigate strong scalability from 16 to 512 cores, eventually leaving each core with 150k elements on 512 cores. Figure 7.5 shows the results, where time-based cost evaluation is compared to the linear cost model. Due to the homogeneous kernels in the porous media flow scenario, the linear cost model returns a better performance as it is not prone to measurement errors. A parallel efficiency



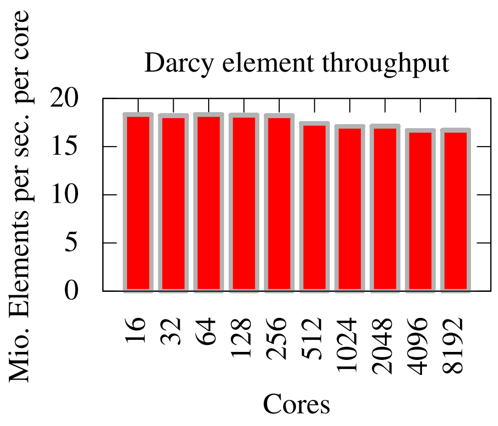


Figure 7.4: Weak scalability of an over-resolved SPE10 scenario executed on 1 to 8192 cores. On the left, an optimal chains-on-chains solver is used for load balancing and on the right midpoint approximation is applied (see Chapter 3.2). While the optimal solver returns a slightly better intermediate performance (96% at 512 cores for the optimal solver and 95% for midpoint approximation), its serial implementation kicks in at 8192 cores and returns a worse performance (82% vs. 91%).

of 87% is achieved on 512 cores.

7.3.2 Scalability Limits of 2D Channel Flow

While the model of the 3D scenario is more interesting, the 2D scenario is simpler with less arithmetics per kernel and therefore more sensitive to scalability issues caused by global communication and by traversal overhead. We will therefore perform a weak scaling study and a strong scaling study to examine how well the code behaves under increase of parallelism for problems with growing size and problems with fixed size, to find out at which point performance breaks down.

Weak scaling of 2D Channel Flow

sam(oa)^2 was executed with a problem size of 1 million elements per core and configured to sustain this number for all test runs. As adaptive refinement and coarsening are active, this is not guaranteed and some variation in the size is possible. The initialization phase of the grid, which is described in Section 3.4.2, and the first time steps are neglected for the scalability test as the grid is typically strongly imbalanced in the beginning and requires a few time steps to stabilize. This is a valid restriction as we try to emulate sustained performance in a production run, which is not affected by the initial phase.

The number of processes is increased from 16 up to 32,768 cores (four SuperMUC thin node islands), where eventually the grid contains around 40G elements. Results for the parallel efficiency are plotted in Figure 7.6. 88% weak scaling efficiency is achieved on up to 4,096 cores, decreasing to 70% on 32,768 cores. Some erratic behavior occurs which needs explanation though. Hybrid scaling is a bit worse in general, but comparable. Performance is slightly worse than for the 3D scenario due to the relatively higher global communication.

A breakup into components (Figure 7.7) illustrates that the erratic behavior at 512 and 2048 cores is caused purely by the linear solver. A closer investigation shows that in these two cases, the number of linear solver iterations was particularly low during the time of measurements, causing a worse memory throughput as the first Conjugate Gradients iteration is more expensive than the following ones.

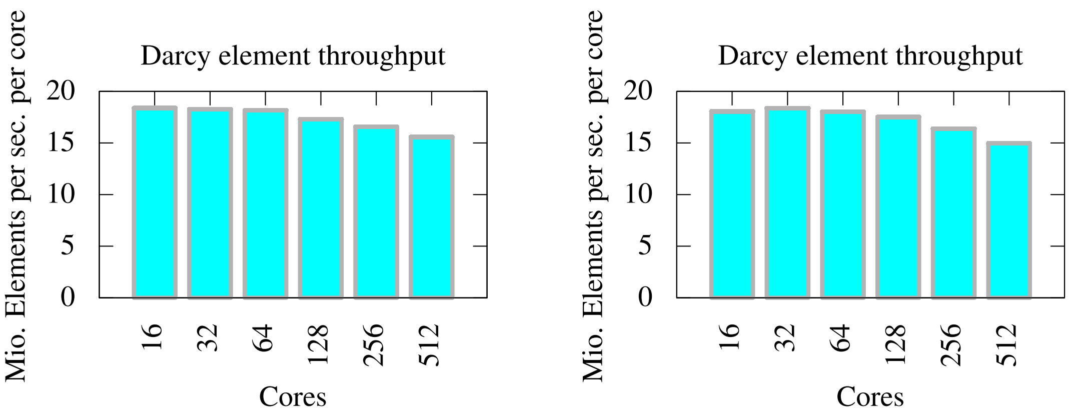


Figure 7.5: Strong scalability of an over-resolved SPE10 scenario with 80M elements. The left image shows the results from 16 to 512 cores for a linear cost model. In the right image, time-based cost evaluation is used. As the linear solver with its homogeneous kernels dominates performance, time-based cost model performs worse with 84% efficiency compared to 87% parallel efficiency.

The performance loss on higher numbers of processes is caused mostly by increased global communication, which is worse for pure MPI parallelization than for hybrid parallelization. Hybrid parallelization still returns a worse element throughput overall, however the reason is a different one. Figure 7.7 (top) shows that the neighbor search component appears to scale badly starting at $1,024$ cores. As the earliest point of communication in adaptive traversals is in the neighbor search, the actual cause is a postponed imbalance during adaptive traversal. Further analysis shows that thread-parallel access to the ASAGI [5] library is the root cause, which was not fully supported at the time this test was conducted. If adaptive traversals are neglected, hybrid parallelization performs much better: I.e. the memory throughput of the linear solver is 50% higher in hybrid parallelization compared to pure MPI parallelization and global communication time is reduced by over 70%.

Strong scaling of 2D Channel Flow

For a strong scaling setting we used a scenario with 77M elements, a minimum refinement depth of 26, and a maximum refinement depth of 40. Due to adaptive refinement, the number of element changes during the computation, continuously increasing the number of cells. We scaled the scenario from 16 to 8192 cores, a full island of the SuperMUC thin node cluster. The resulting performance is plotted in Figure 7.8. Hence, at 8,192 cores, each partition consists of 9,000 elements, which is a very tough test case for a memory-bound problem.

Parallel efficiency with Intel MPI ist at 88% on 512 cores and with IBM MPI at 83% on 512 cores. At 2,048 cores we still obtain 62% with Intel MPI and 58% with IBM MPI. Pure Intel MPI performance drops rapidly afterwards which is most likely due to switching from eager message sending to the rendezvous protocol for communication, which strongly increases latency. Hybrid parallelization is not affected yet as it works on a smaller number of processes and can still use the eager protocol. Eventually, efficiency is reduced to 36% on 8,192 cores. Note that at this point a single core holds only 10k cells, which is an extremely low number for a memory-bound simulation. According to [127], scalability cannot be upheld any longer under these conditions, hence the performance drop. Again, performance is worse than for the 3D channel flow scenario, as in the 2D case scalability is affected much more by global communication and traversal overhead.

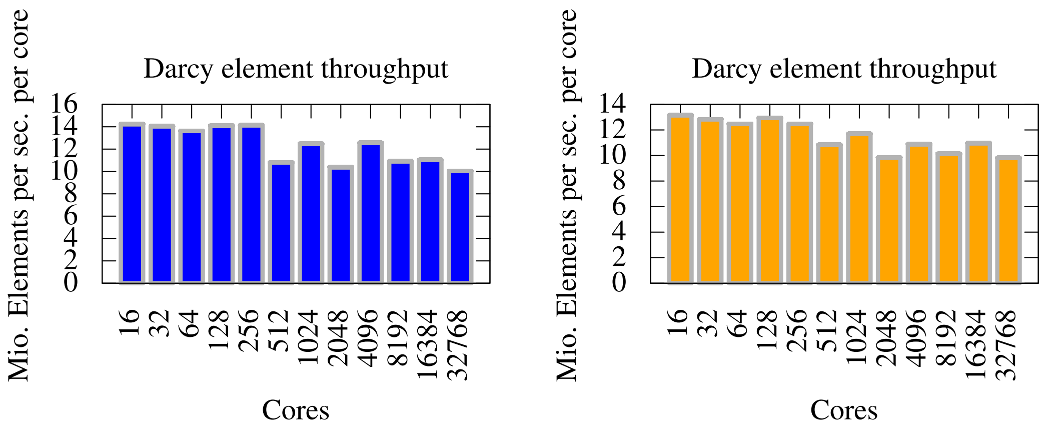


Figure 7.6: MPI Weak scaling of a dynamically adaptive simulation from 1 to 2,048 nodes and up to 40G elements on the SuperMUC thin nodes with pure MPI (left) and hybrid (right) parallelization. Performance oscillates due to varying numbers of CG iterations per time step in a weak scaling setting.

Sections per core	8	16	32
Memory TP in GB/s (Ratio)	334	340	326

Table 7.4: Finding the number of sections per core that maximizes memory throughput. An MPI-parallelized simulation on 512 thin node cores with 1M elements per core returns similar results for all three values, hence the choice has only a minor impact on performance. The optimal number of sections per core appears to be around 16.

Choosing an optimal number of sections per core

An influential factor on scalability is the choice of the parameter for the number of sections per core as explained in Chapter 3.2.2. If the value is too low, load balancing on atomic sections will return large imbalances and communication is not hidden well. If the value is too high, section overhead will start to affect performance. The optimal number is usually strongly dependent on the scenario and other parameters, hence we measure exemplary results for pure MPI parallelization. Table 7.4 shows that in this scenario the difference is rather small and an empirical value of 16 sections per core should be sufficient to achieve good results. Note that for hybrid parallelization, the number is typically smaller, as the distributed load balancing algorithm has more sections per process available for fine-grained balance.

7.4 Conclusion: A Scalable Memory Bound Problem

We showed in this chapter that the overall quality of parallel performance is satisfying for memory-bound problems. The baseline performance analysis showed that memory throughput of the porous media flow scenario is close to the on the test systems, and hence computationally very cheap. Hence, the scenario is also hard to scale as it is much more sensitive to communication overhead than a compute-bound problem. Nevertheless, weak scaling efficiency for the presented cases was shown to be around 90% on up to 8192 cores and strong scaling efficiency at 85% on 512 cores, which is an excellent result.

Most of the advanced load balancing techniques did not improve performance however, which is

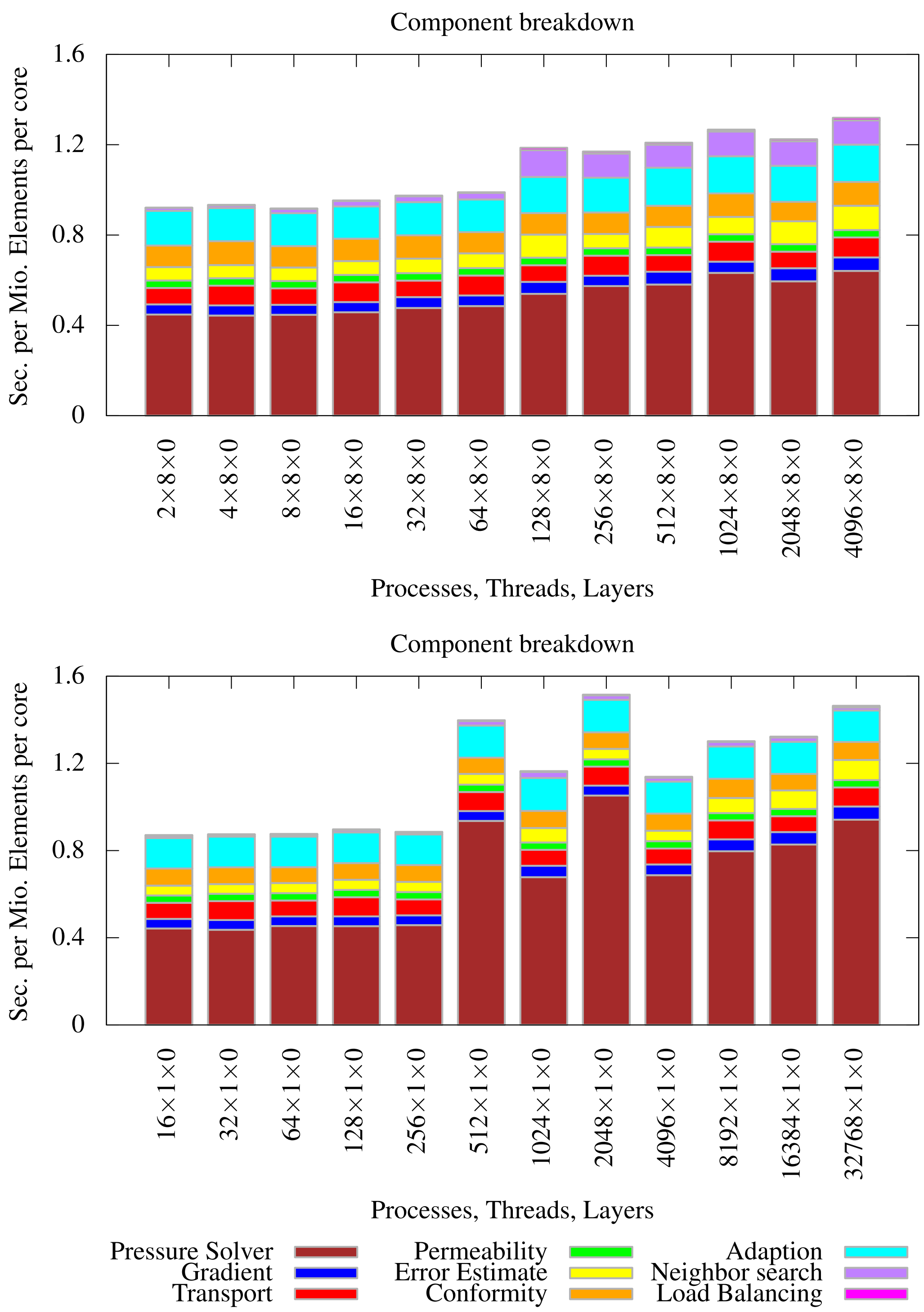
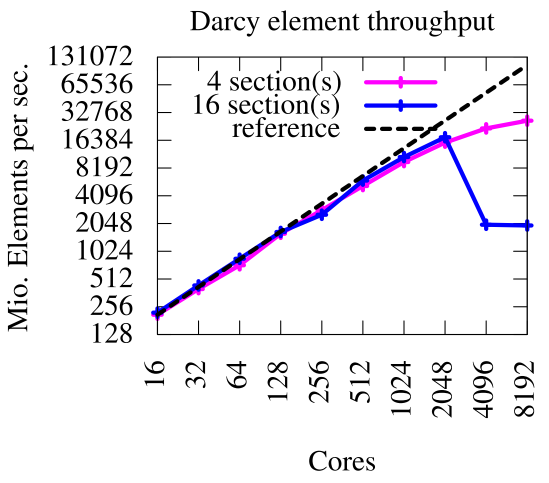


Figure 7.7: Component analysis of a weak scaling study for hybrid (top) and pure MPI (bottom) parallelization. Each bar shows the wall clock time per simulation time step per core for 1M elements, split by components. This is not the actual number of elements per core, which depends on adaptive refinement and coarsening, but a normalized value chosen for comparison.



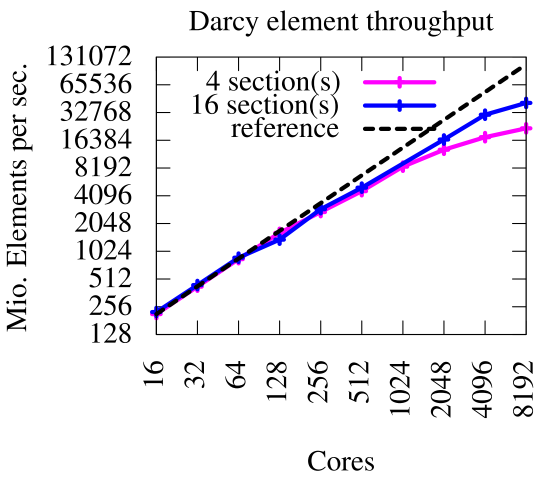


Figure 7.8: Pure MPI (blue) and Hybrid OpenMP+MPI (pink) strong scaling with 60M to 100M unknowns on 16 to 8192 cores of the SuperMUC thin nodes using Intel MPI (left image) and IBM MPI (right image). Partitions are split into 16 sections per core for the pure MPI run and 4 sections per core for the hybrid run. Intel MPI performance breaks down at 4096 cores while IBM MPI behaves more stable, but also starts degrading. Hybrid parallelization scales worse in general.

due to the homogeneous nature of the linear solver that dominates performance of the scenario. Another issue is scalability of the SPE10 scenario in its original size, which is too small to be parallelized on a large number of cores.

PART IV

TSUNAMI WAVE PROPAGATION

8 Modeling Tsunami Wave Propagation

This chapter discusses an implementation of a tsunami wave propagation scenario in the finite volume interface of sam(oa)^2. Performance is compute-bound, as we will discuss in the analysis in Section 10.2. Therefore, the scenario is not suitable for examination of memory performance. However, in contrast to the porous media flow scenario in Chapter 5, the overall computational effort per time step is low. Therefore, the scenario is a good test case for the following goals:

- Demonstrating a use case of the finite volume interface of sam(oa)^2 that includes communication over edges.
- Examining the performance of adaptive mesh refinement and load balancing. Due to the small computational effort per time step, the influence of these components is expected to be large in this scenario.

This test case will not be discussed as extensively as porous media flow, taking into account that many concepts are similar. Instead, the focus is mainly on new concepts, such as the finite volume interface.

8.1 The 2D Shallow Water Equations with Bathymetry Terms

For the simulation of shallow water waves, we consider a domain with three spatial dimensions that contains three phases: water, air, and solid. Water is assumed inviscid and incompressible. Typical wavelengths largely exceed the height of the water columns, as tsunami waves reach lengths of hundreds of kilometers, whereas the ocean depth is only a few kilometers deep. The ratio of wave length to water height defines the waves as *shallow*, allowing an approximation that integrates over the vertical axis and ignores vertical derivatives, as explained in e.g. [67]. Note that this assumption is not always true near coastal regions. The wave lengths may shrink to a few hundred meters and tsunamis can indeed become deep water waves when they hit the shore.

Due to the vertical integration, solid, water and air are each assumed to be contiguous. Breaking waves and complex geometric features such as caves or arcs cannot be represented. Instead, water is modeled to form a single vertical pillar in each 2D position $\mathbf{x} = (x, y)$ with space- and time-dependent height $h(\mathbf{x}, t)$ and particle velocity $\mathbf{u}(\mathbf{x}, t)$, see Fig. 8.1. The pillar starts at the sea floor, whose height is defined by the bathymetry data $b(\mathbf{x})$, and ends at the water surface elevation $\eta(\mathbf{x}, t) = b(\mathbf{x}) + h(\mathbf{x}, t)$.

The governing equations for the system are the *shallow water equations*. Here, we consider three different versions, starting with a simple 1D model, where the sea floor is assumed flat. Hence,

$$\begin{aligned} h_t + (hu)_x &= 0, \\ (hu)_t + \left(hu^2 + \tfrac{1}{2}gh^2\right)_x &= 0, \end{aligned} \tag{8.1}$$

where g is the gravitational acceleration. The hyperbolic system of partial differential equations (8.1) models the conservation of mass, represented by an advection term for the water height h, and the

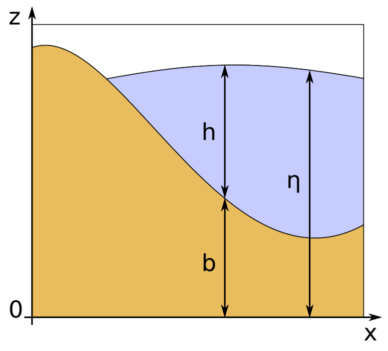


Figure 8.1: Modeling 2D ocean waves with the vertically integrated 1D shallow water equations and bathymetry source terms. Bathymetry data b and water height h add up to the water surface elevation η. The water height may become 0, which indicates a dry state.

conservation of momentum hu, where a similar advection term and the additional hydrostatic pressure term $\frac{1}{2}gh^2$ appear in the equation. If downgrade forces on the sea floor slopes are added, we obtain the 1D shallow water equations with bathymetry source terms:

$$\begin{aligned} h_t + (hu)_x &= 0, \\ (hu)_t + \left(hu^2 + \tfrac{1}{2}gh^2\right)_x &= -ghb_x, \end{aligned} \tag{8.2}$$

where momentum conservation is replaced with a *balance law*, i.e. a conservation equation with source terms. The additional term has a significant influence on the analytic solution as we will see in this section. Extension of (8.2) to 2D space adds derivatives in y-direction and an equation for the second momentum component; that is

$$\begin{aligned} h_t + (hu)_x + (hv)_y &= 0, \\ (hu)_t + \left(hu^2 + \tfrac{1}{2}gh^2\right)_x + (huv)_y &= -ghb_x, \\ (hv)_t + (huv)_x + \left(hv^2 + \tfrac{1}{2}gh^2\right)_y &= -ghb_y. \end{aligned} \tag{8.3}$$

Some complexity arises here due to the existence of a second flux function $\mathbf{g}$ and the change of scalar to vector functions. The system (8.3) can be expressed in the general form of a balance law, which in the 2D case is

$$\mathbf{q}_t + \mathbf{f}(\mathbf{q})_x + \mathbf{g}(\mathbf{q})_y = \Psi(\mathbf{x}), \tag{8.4}$$

where

$$\mathbf{q} := \begin{pmatrix} h \\ hu \\ hv \end{pmatrix}, \ \mathbf{f}(\mathbf{q}) := \begin{pmatrix} hu \\ hu^2 + \frac{1}{2}gh^2 \\ huv \end{pmatrix}, \ \mathbf{g}(\mathbf{q}) := \begin{pmatrix} hv \\ huv \\ hv^2 + \frac{1}{2}gh^2 \end{pmatrix}, \ \Psi(\mathbf{x}) := \begin{pmatrix} 0 \\ -ghb_x \\ -ghb_y. \end{pmatrix}$$

Systems of this form are well-known and hence, a couple of solutions and methods for general balance laws and the shallow water equations in particular are available, see for example [15,44,49,50].

Steady State Solutions

A particularly interesting subset of solutions for balance laws is the set of stationary solutions, which are by definition invariant in time; i.e. the solution $\bar{\mathbf{q}}$ satisfies

$$\bar{\mathbf{q}}_t = 0,$$

and hence,

$$\mathbf{f}(\bar{\mathbf{q}})_x + \mathbf{g}(\bar{\mathbf{q}})_y = \Psi(\mathbf{x}). \tag{8.5}$$

A solution $\bar{\mathbf{q}}$ with this property is called *steady state*. For the shallow water equations, a set of steady states is easily found by applying the chain rule to the terms $\left(\frac{1}{2}gh^2\right)_x$ and $\left(\frac{1}{2}gh^2\right)_y$ in (8.3). Then

$$\begin{aligned} h_t + (hu)_x + (hv)_y &= 0. \\ (hu)_t + \left(hu^2\right)_x + (huv)_y + gh\,(b+h)_x &= 0, \\ (hv)_t + (huv)_x + \left(hv^2\right)_y + gh\,(b+h)_y &= 0, \end{aligned} \tag{8.6}$$

suggesting that the equations will hold if $hu = 0$, $hv = 0$, and either $h = 0$ or $(b+h)_x = (b+h)_y = 0$ in each point. Hence, the *lake-at-rest* solution

$$\begin{aligned} h &= \eta_{\text{eq}} - b, \\ hu &= 0, \\ hv &= 0, \end{aligned} \tag{8.7}$$

is a steady state, where $\eta_{\text{eq}} > \max(b)$ is a constant surface elevation greater than the maximum bathymetry in the domain. Since hydrostatic pressure and downgrade forces are in an equilibrium, no change in the solution occurs. Allowing $\eta_{\text{eq}} \leq \max(b)$ requires a generalization to $h = \max(\eta_{\text{eq}} - b, 0)$ in (8.7), permitting dry areas. If inundation of coasts is neglected, this case can be treated as an internal wall boundary with complicated geometry though.

Steady states are important for numerical solvers, as they are non-trivial solutions that are hard to stabilize in a numerical scheme. A numerical method that preserves them is called *well-balanced*. See [15, 48, 67] for more information on this topic.

8.1.1 Quasilinear Form of the Shallow Water Equations

For the analysis we will also need the quasilinear form of the shallow water equations. The general layout for 1D balance laws is:

$$\mathbf{q}_t + \mathbf{f}'(\mathbf{q}) \cdot \mathbf{q}_x = \Psi(\mathbf{x}). \tag{8.8}$$

In the case of the 1D shallow water equations, the matrix $\mathbf{f}'(\mathbf{q})$ is easily derived from (8.2) by the application of the chain rule, which yields

$$\mathbf{f}'(\mathbf{q}) = \begin{pmatrix} 0 & 1 \\ -u^2 + gh & 2u \end{pmatrix}.$$

Diagonalization of $\mathbf{f}'(\mathbf{q})$ returns the matrix

$$\mathbf{\Lambda} = \begin{pmatrix} u - \sqrt{gh} & 0 \\ 0 & u + \sqrt{gh} \end{pmatrix}. \tag{8.9}$$

The eigenvalues in the diagonal define the signal speeds in the system and will be important for the numerical stability of the solution. Here they are used to determine a diffusion coefficient for a flux solver in Section 8.2.2 as well as a stability condition for explicit time stepping in Section 8.2.4.

Extension to 2D

In 2D, the layout of the quasilinear form of multivariate balance laws changes to

$$\mathbf{q}_t + \mathbf{f}'(\mathbf{q}) \cdot \mathbf{q}_x + \mathbf{g}'(\mathbf{q}) \cdot \mathbf{q}_y = \Psi(\mathbf{x}). \tag{8.10}$$

Hence, the derivatives of both flux functions $\mathbf{f}$ and $\mathbf{g}$ must be computed from (8.3), returning the matrices $\mathbf{f}'(\mathbf{q})$ and $\mathbf{g}'(\mathbf{q})$, which gives

$$\mathbf{f}'(\mathbf{q}) = \begin{pmatrix} 0 & 1 & 0 \\ -u^2 + gh & 2u & 0 \\ -uv & v & u \end{pmatrix}, \quad \mathbf{g}'(\mathbf{q}) = \begin{pmatrix} 0 & 0 & 1 \\ -uv & v & u \\ -v^2 + gh & 0 & 2v \end{pmatrix},$$

with the diagonal matrices

$$\mathbf{\Lambda_f} = \begin{pmatrix} u - \sqrt{gh} & 0 & 0 \\ 0 & u & 0 \\ 0 & 0 & u + \sqrt{gh} \end{pmatrix}, \quad \mathbf{\Lambda_g} = \begin{pmatrix} v - \sqrt{gh} & 0 & 0 \\ 0 & v & 0 \\ 0 & 0 & v + \sqrt{gh} \end{pmatrix}. \tag{8.11}$$

that contain the 2D eigenvalues.

8.2 Discretization on Cell Centered Finite Volumes

For the discretization, we adopt the finite volume approach by [68]. Unknowns are represented via cell-averaged values in each element. The corresponding element-local vector

$$\mathbf{q}_j^{(t)} = \begin{pmatrix} h_j \\ h_j u_j \\ h_j v_j \end{pmatrix}^{(t)} \tag{8.12}$$

contains the cell-averaged water height, momentum and bathymetry in element j at time t. Following a wave-propagation approach, so-called *net updates* $\mathcal{F}\left(\mathbf{q}_j^{(t)}, \mathbf{q}_i^{(t)}\right)$ are computed on the cell-incident faces. The net updates represent transport of mass and of momentum into and out of the elements through each face, which leads to the update scheme

$$\mathbf{q}_j^{(t+\Delta t)} = \mathbf{q}_j^{(t)} + \frac{\Delta t}{V_j} \sum_{i \in \mathcal{N}(j)} A_{j,i}\, \mathcal{F}\left(\mathbf{q}_j^{(t)}, \mathbf{q}_i^{(t)}\right), \tag{8.13}$$

which is an explicit Euler time step. The equation computes the temporal change of the quantity vector $\mathbf{q}_j^{(t)}$ in cell j with volume V_j from numerical fluxes that act on the interface between cells j and i with the intersection areas $A_{j,i}$. Furthermore, $\mathcal{N}(j)$ is the set of neighbor cells of j. Thus, a time step consists of evaluating fluxes on all element interfaces and accumulating net updates in each element. A flux solver computes the net updates $\mathcal{F}\left(\mathbf{q}_j^{(t)}, \mathbf{q}_i^{(t)}\right)$, where $\mathbf{q}_i^{(t)}$ and $\mathbf{q}_j^{(t)}$ are the quantity vectors of cells adjacent to the corresponding face. Different choices for the flux solver are discussed in Section 8.2.2.

8.2.1 The Simulation Loop

In sam(oa)^2, the resulting time stepping scheme is performed as described in Algorithm 8.1, where an initial state is set and the grid is incrementally refined and distributed. Next, a displacement is applied to the bathymetry data and the water height to create the initial tsunami wave. Finally, explicit time steps are executed in a loop and combined with adaptive mesh refinement in each iteration.

Except for initializing the bathymetry, water height and displacements, which are simple component-wise operations, the essential steps will be explained in detail in the following paragraphs.

Algorithm 8.1: Parallel simulation of tsunami wave propagation with static displacements and adaptive mesh refinement. In the first phase, the grid is set up and the displacements are applied. The second phase alternates explicit time steps and adaptive mesh refinement.

```
Input: $t_{eq}, t_{max}$
Output: Cell states $\mathbf{q}^{(t_{max})} = (h, hu, hv, b)$ at time $t_{max}$

traversal: Initialize $\mathbf{q}$, set refinement flags;
// Grid setup
while refinement flags are set do
    traversal: Adapt grid, interpolate $\mathbf{q}$, and balance load;
    traversal: Initialize $\mathbf{q}$, set refinement flags;
end
traversal: Displace bathymetry $b$ and water height $h$;
// Tsunami time steps
while $t < t_{max}$ do
    traversal: Adapt grid, interpolate/restrict $\mathbf{q}$, and balance load ;
    traversal: Compute time step $\Delta t$, update $\mathbf{q}$ with transport step, set refinement and coarsening
    flags;
    $t \leftarrow t + \Delta t$;
end
```

- Section 8.2.2 explains how fluxes are computed with a simple scheme and an external solver.
- Section 8.2.4 discusses the computation of the time step size Δt and update of unknowns with a transport step, using an optimized traversal scheme for cell-centered finite volumes.
- In Section 8.2.5, an error indicator is defined for adaptive mesh refinement. Additionally, interpolation and restriction of cell unknowns and bathymetry are analyzed.

8.2.2 From Lax Friedrichs Fluxes To Augmented Riemann Solvers

The most important component of the time stepping scheme in finite volume methods is the flux solver. Different implementations exist in sam(oa)^2, which we will slowly introduce by starting with the simplest case, a solver for the 1D conservation equations (8.1). Afterwards, we will proceed to more complicated flux solvers.

Solving the 1D Conservation Equations

As stated earlier, the 1D equations without bathymetry (8.1) are considered first, where state vectors have only two components

$$\mathbf{q}_j^{(t)} = \begin{pmatrix} h_j \\ h_j u_j \end{pmatrix}^{(t)}. \tag{8.14}$$

The *Lax-Friedrichs method*, described in [67] for example, computes updates for each cell j according to

$$\mathbf{q}_j^{(t+\Delta t)} = \frac{1}{2}\left(\mathbf{q}_{j+1}^{(t)} + \mathbf{q}_{j-1}^{(t)}\right) - \frac{\Delta t}{2\Delta x}\left(f\left(\mathbf{q}_{j+1}^{(t)}\right) - f\left(\mathbf{q}_{j-1}^{(t)}\right)\right).$$

In *flux formulation*, this corresponds to

$$\mathbf{q}_j^{(t+\Delta t)} := \mathbf{q}_j^{(t)} - \frac{\Delta t}{\Delta x}\left(\mathcal{F}_{\mathrm{lf}}\left(\mathbf{q}_j, \mathbf{q}_{j+1}\right) - \mathcal{F}_{\mathrm{lf}}\left(\mathbf{q}_{j-1}, \mathbf{q}_j\right)\right)$$
$$\mathcal{F}_{\mathrm{lf}}\left(\mathbf{q}_j, \mathbf{q}_{j+1}\right) := \frac{1}{2}\left(\mathbf{f}\left(\mathbf{q}_j\right) + \mathbf{f}\left(\mathbf{q}_{j+1}\right) + \frac{\Delta x}{\Delta t}\left(\mathbf{q}_j - \mathbf{q}_{j+1}\right)\right),$$

where an artificial diffusion term $\frac{\Delta x}{\Delta t}\left(\mathbf{q}_j - \mathbf{q}_{j+1}\right)$ is introduced. This term reduces accuracy but is necessary to maintain stability. Despite the diffusion term, the method converges to the true solution when $\Delta x \to 0$, since the term vanishes when the grid is refined infinitely [67]. The *local Lax-Friedrichs method*, also known as *Rusanov method* [99], reduces the amount of diffusion by replacing the global value $\frac{\Delta x}{\Delta t}$ with a local coefficient $\xi_{j,j+1}$, which is chosen as the absolute maximum of the signal speeds determined from the quasilinear form (8.8). In 1D the value is

$$\xi_{j,j+1} := \max\left(|u_j| + \sqrt{gh_j}, |u_{j+1}| + \sqrt{gh_{j+1}}\right), \tag{8.15}$$

with the flux function

$$\mathcal{F}_{\mathrm{llf}}\left(\mathbf{q}_j, \mathbf{q}_{j+1}\right) := \tfrac{1}{2}\left(\mathbf{f}\left(\mathbf{q}_j\right) + \mathbf{f}\left(\mathbf{q}_{j+1}\right) + \xi_{j,j+1}\ \left(\mathbf{q}_j - \mathbf{q}_{j+1}\right)\right).$$

Adding the Bathymetry Source Term

We shall modify the local Lax-Friedrichs method to solve (8.2) by adding a source term to the system and storing additional cell averaged bathymetry data b_j in each cell. The flux solver must be modified to handle the source terms and, in particular, should provide a well-balanced method. For the discrete system, this translates to balancing *quasi-steady state* solutions, which are a set of cell states $\bar{\mathbf{q}}_j$ that satisfy the condition

$$\mathbf{f}\left(\bar{\mathbf{q}}_{j+1}\right) - \mathbf{f}\left(\bar{\mathbf{q}}_j\right) = \Delta x\, \Psi_{j,j+1} \tag{8.16}$$

on all cell interfaces $j, j+1$. Quasi-steady states do not strictly solve (8.5). Instead, they approximate the steady state condition with a discrete source term. Using a finite difference approximation, we obtain

$$\Delta x\, \Psi_{j,j+1} = \begin{pmatrix} 0 \\ -\frac{1}{2}g\left(h_j + h_{j+1}\right)\left(b_{j+1} - b_j\right) \end{pmatrix}. \tag{8.17}$$

This particular choice ensures that the lake at rest as in (8.7) is a quasi-steady state. With an extension similar to [132], the Lax-Friedrichs method is modified to balance the lake at rest, which is

$$\mathbf{q}_j^{(t+\Delta t)} := \mathbf{q}_j^{(t)} - \frac{\Delta t}{\Delta x}\left(\mathcal{F}_{\mathrm{llfb}}\left(\mathbf{q}_j^{(t)}, \mathbf{q}_{j+1}^{(t)}\right) - \mathcal{F}_{\mathrm{llfb}}\left(\mathbf{q}_{j-1}^{(t)}, \mathbf{q}_j^{(t)}\right) + \Delta x\, \Psi_{j-1,j}^{(t)}\right),$$
$$\mathcal{F}_{\mathrm{llfb}}\left(\mathbf{q}_j, \mathbf{q}_{j+1}\right) :=$$
$$\frac{1}{2}\left(\mathbf{f}\left(\mathbf{q}_j\right) + \mathbf{f}\left(\mathbf{q}_{j+1}\right) - \Delta x\, \Psi_{j,j+1} + \xi_{j,j+1}\ \left(\begin{pmatrix} b_j + h_j \\ h_j u_j \end{pmatrix} - \begin{pmatrix} b_{j+1} + h_{j+1} \\ h_{j+1} u_{j+1} \end{pmatrix}\right)\right). \tag{8.18}$$

With a rearrangement, (8.18) is generalized to

$$\mathbf{q}_j^{(t+\Delta t)} := \mathbf{q}_j^{(t)} - \frac{\Delta t}{\Delta x}\left(\mathcal{F}_{\mathrm{llfb}}(\mathbf{q}_j^{(t)}, \mathbf{q}_{j+1}^{(t)}) - \mathcal{F}_{\mathrm{llfb}}(\mathbf{q}_j^{(t)}, \mathbf{q}_{j-1}^{(t)})\right),$$
$$\mathcal{F}_{\mathrm{llfb}}(\mathbf{q}_j, \mathbf{q}_i) := \frac{1}{2}\left(\mathbf{f}(\mathbf{q}_j) + \mathbf{f}(\mathbf{q}_i) - \Delta x\, \Psi_{j,i} + \xi_{j,i}\ \left(\begin{pmatrix} b_j + h_j \\ h_j u_j \end{pmatrix} - \begin{pmatrix} b_i + h_i \\ h_i u_i \end{pmatrix}\right)\right), \tag{8.19}$$

which holds for arbitrary pairs of neighbor cells (j, i). While the diffusion term next to the coefficient $\xi_{j,i}$ vanishes for the lake at rest, it will be nonzero in general. Other quasi-steady states are not preserved. Hence, the method is not well-balanced. On a side note, modifying the coefficient so that it handles all quasi-steady states returns Roe's method [98]. Because of its simple algorithm the extended local Lax-Friedrichs method is very useful as a quick reference solver; due to its aforementioned weaknesses, it should not be used for production runs however. The solver is enabled in sam(oa)^2 with the compilation flag

```
  scons flux_solver=llfbath
```

Advanced Riemann Solvers: Flux Differencing with Source Terms

The main problem of the Lax-Friedrichs solver is the diffusion term that degrades the quality of the solution. Advanced solvers reduce this effect by approximating the analytic solution on each interface between cells j and $j+1$. This is achieved by choosing a matrix $\mathbf{A}$ on each interface that approximates the derivative $\mathbf{f}'(\mathbf{q})$ in the quasilinear form (8.8) and solves the equation

$$\mathbf{f}(\mathbf{q}_j) - \mathbf{f}(\mathbf{q}_i) = \mathbf{A}\left(\mathbf{q}_j - \mathbf{q}_i\right). \tag{8.20}$$

Here we choose

$$\mathbf{A} := \begin{pmatrix} 0 & 1 \\ -\bar{u}^2 + g\bar{h} & 2\bar{u} \end{pmatrix}, \tag{8.21}$$

which is known as the Roe matrix in literature [98]. In (8.21), $(\bar{h}, \bar{u})^T$ denotes the *Roe average* of left and right cell states, more precisely

$$\bar{h} = \tfrac{1}{2}\left(h_j + h_{j+1}\right) \quad \text{and} \quad \bar{u} = \frac{\sqrt{h_j}\, u_j + \sqrt{h_{j+1}}\, u_{j+1}}{\sqrt{h_j} + \sqrt{h_{j+1}}}. \tag{8.22}$$

As $\mathbf{A}$ approximates the system matrix, waves and wave speeds of signals in the solution are also approximated by the eigenvectors and eigenvalues of $\mathbf{A}$, namely

$$\begin{aligned} \lambda_1 &:= \bar{u} - \sqrt{g\bar{h}} \quad \text{and} \quad \mathbf{v}_1 := \begin{pmatrix} 1 \\ \bar{u} - \sqrt{g\bar{h}} \end{pmatrix}, \text{ as well as} \\ \lambda_2 &:= \bar{u} + \sqrt{g\bar{h}} \quad \text{and} \quad \mathbf{v}_2 := \begin{pmatrix} 1 \\ \bar{u} + \sqrt{g\bar{h}} \end{pmatrix}. \end{aligned} \tag{8.23}$$

The main idea of the *f-wave* method [15] is to use the Roe matrix $\mathbf{A}$ to create an eigenvalue decomposition of the balance term at the interface into flux waves [67]. This yields

$$\mathbf{f}(\mathbf{q}_{j+1}) - \mathbf{f}(\mathbf{q}_j) - \Delta x\, \Psi_{j,j+1} = \sum_{i=1}^{2} \beta_i\, \mathbf{v}_i = \sum_{i=1}^{2} \mathcal{Z}_i.$$

where the coefficients β_i are determined by solving the corresponding linear system of equations. The flux solver decomposes the balance term into signals that carry deviations from the quasi-steady state. Net updates are computed by splitting the signals into left- and right-going contributions, resulting in

$$\begin{aligned} \mathbf{q}_j^{(t+\Delta t)} &:= \mathbf{q}_j^{(t)} - \frac{\Delta t}{\Delta x}\left(\mathcal{F}_{\text{fw}}^{-}(\mathbf{q}_j^{(t)}, \mathbf{q}_{j+1}^{(t)}) - \mathcal{F}_{\text{fw}}^{+}(\mathbf{q}_j^{(t)}, \mathbf{q}_{j-1}^{(t)})\right), \\ \mathcal{F}_{\text{fw}}^{-}(\mathbf{q}_j, \mathbf{q}_{j+1}) &:= \sum_{i=1}^{2} \beta_i^{-}\, \mathbf{v}_i = \sum_{i=1}^{2} \mathcal{Z}_i^{-}, \\ \mathcal{F}_{\text{fw}}^{+}(\mathbf{q}_j, \mathbf{q}_{j+1}) &:= \sum_{i=1}^{2} \beta_i^{+}\, \mathbf{v}_i = \sum_{i=1}^{2} \mathcal{Z}_i^{+}, \end{aligned} \tag{8.24}$$

where

$$\beta_i^- := \begin{cases} 0 & \text{if } \lambda_i > 0 \\ \frac{1}{2}\beta_i & \text{if } \lambda_i = 0 \\ \beta_i & \text{if } \lambda_i < 0 \end{cases} \quad \text{and} \quad \beta_i^+ := \begin{cases} \beta_i & \text{if } \lambda_i > 0 \\ \frac{1}{2}\beta_i & \text{if } \lambda_i = 0 \\ 0 & \text{if } \lambda_i < 0 \end{cases}.$$

Thus, the signal β_i is transported along the wave $\mathcal{Z}_i$ from cell j to cell $j+1$ if the wave speed λ_i is positive and in reverse direction if the wave speed is negative. If the wave speed is 0, the signal is split between both cells to ensure that $\beta_i^- + \beta_i^+ = \beta_i$ always holds and thus

$$\mathbf{f}(\mathbf{q}_{j+1}) - \mathbf{f}(\mathbf{q}_j) - \Delta x\, \Psi_{j,j+1} = \mathcal{F}_{\text{fw}}^-(\mathbf{q}_j, \mathbf{q}_{j+1}) + \mathcal{F}_{\text{fw}}^+(\mathbf{q}_j, \mathbf{q}_{j+1}). \tag{8.25}$$

The f-wave solver is not directly implemented in sam(oa)^2 but instead provided by the GeoClaw package [67]. It is enabled with the compilation flag

```
scons flux_solver=fwave
```

The package additionally contains an augmented Riemann solver that handles wetting and drying to capture the inundation of coasts. Readers are directed to [49] for details. The corresponding compilation flag is

```
scons flux_solver=aug_riemann
```

A verification of the flux solvers will be performed in Section 9.2 on a production scenario.

Extension to 2D

Finite volume discretization of a general 2D balance law (8.4) with an Euler time stepping scheme returns the update rule

$$\mathbf{q}_j^{(t+\Delta t)} = \mathbf{q}_j^{(t)} - \frac{\Delta t}{V_j} \sum_{i \in \mathcal{N}(j)} A_{j,i} \mathcal{F}(\mathbf{q}_j^{(t)}, \mathbf{q}_i^{(t)}), \tag{8.26}$$

which is in flux formulation for each cell j. The net updates are obtained by a flux solver $\mathcal{F}$ and the discrete state vector in 2D is

$$\mathbf{q}_j^{(t)} = \begin{pmatrix} h \\ hu \\ hv \end{pmatrix}_j^{(t)}. \tag{8.27}$$

Extension of flux solvers to 2D appears straightforward, but there is a problem: Assume we have a 1D flux solver $\hat{\mathcal{F}}$, such as the local Lax-Friedrichs flux (8.19), then the additional flux function $\mathbf{g}$ prevents us from directly reusing it on the 2D equations as the solver only knows the flux function $\mathbf{f}$. The solution is to transform all cell states, fluxes, and source terms from world space to normal space where the second flux function disappears. While this is a valid method, it neglects waves that pass through vertex-connected interfaces, and therefore involves a loss of accuracy [67].

Taking a closer look, the flux and source term for every state $\mathbf{q}$ on the interface between cells j and i with surface area $A_{j,i}$ and normal $\mathbf{n}_{j,i} := \left((n_x)_{j,i}, (n_y)_{j,i}\right)^T$ evaluate to:

$$\mathbf{f}(\mathbf{q})\ (n_x)_{j,i} + \mathbf{g}(\mathbf{q})\ (n_y)_{j,i} = \left(u\ (n_x)_{j,i} + v\ (n_y)_{j,i}\right) \begin{pmatrix} h \\ hu \\ hv \end{pmatrix} + \frac{1}{2} g h^2 \begin{pmatrix} 0 \\ (n_x)_{j,i} \\ (n_y)_{j,i} \end{pmatrix} \tag{8.28}$$

and

$$\Delta x\,\Psi_{j,i} = -\tfrac{1}{2}g\,(h_j+h_i)\,(b_i-b_j)\begin{pmatrix}0\\(n_x)_{j,i}\\(n_y)_{j,i}\end{pmatrix}. \tag{8.29}$$

Using the transformation

$$\mathbf{T}_{j,i} := \begin{pmatrix}1 & 0 & 0\\0 & (n_x)_{j,i} & (n_y)_{j,i}\\0 & -(n_y)_{j,i} & (n_x)_{j,i}\end{pmatrix}, \tag{8.30}$$

we define

$$\begin{pmatrix}h\\h\hat{u}\\h\hat{v}\end{pmatrix} := \mathbf{T}_{j,i}\begin{pmatrix}h\\hu\\hv\end{pmatrix} \quad\text{and}\quad \Delta x\,\hat{\Psi}_{j,i} := \mathbf{T}_{j,i}\,\Delta x\,\Psi_{j,i}. \tag{8.31}$$

With this, the flux and source term can be written as

$$\begin{aligned}\mathbf{f}(\mathbf{q})\,(n_x)_{j,i}+\mathbf{g}(\mathbf{q})\,(n_y)_{j,i} &= \mathbf{T}_{j,i}^{-1}\begin{pmatrix}h\hat{u}\\h\hat{u}^2+\frac{1}{2}\,g\,h^2\\h\hat{u}\hat{v}\end{pmatrix} = \mathbf{T}_{j,i}^{-1}\,\mathbf{f}(\mathbf{T}_{j,i}\,\mathbf{q}),\\ \Delta x\,\Psi_{j,i} &= \mathbf{T}_{j,i}^{-1}\begin{pmatrix}0\\-\frac{1}{2}\,g\,(h_j+h_i)\,(b_i-b_j)\\0\end{pmatrix} = \mathbf{T}_{j,i}^{-1}\,\Delta x\,\hat{\Psi}_{j,i}.\end{aligned} \tag{8.32}$$

Intuitively, the linearity of $\mathbf{f}$ and $\Delta x\,\Psi_{j,i}$ in $\mathbf{T}_{j,i}$ should also apply to the flux solver $\mathcal{F}$, as the returned net update is typically either a physical flux or a balance term. Hence, we substitute the 2D flux solver $\mathcal{F}$ in (8.26) with a 1D flux solver $\hat{\mathcal{F}}$ and the corresponding forward and backward transformations to obtain

$$\mathbf{q}_j^{(t+\Delta t)} = \mathbf{q}_j^{(t)} - \frac{\Delta t}{V_j}\sum_{i\in\mathcal{N}(j)} A_{j,i}\,\mathbf{T}_{j,i}^{-1}\,\hat{\mathcal{F}}\left(\mathbf{T}_j\,\mathbf{q}_j^{(t)},\mathbf{T}_i\,\mathbf{q}_i^{(t)}\right). \tag{8.33}$$

Mapping cell states and net updates between world space and normal space, we are now able to apply the 1D flux solvers defined in Section 8.2.2 to compute 1D net updates, using only the flux function $\mathbf{f}$ and the transformed source term $\Delta x\,\hat{\Psi}_{j,i}^{(t)}$. As an example, in 2D the extended local Lax-Friedrichs method has the form

$$\begin{aligned}\mathbf{q}_j^{(t+\Delta t)} &= \mathbf{q}_j^{(t)} - \frac{\Delta t}{V_j}\sum_{i\in\mathcal{N}(j)} A_{j,i}\,\mathcal{F}_{\text{llfb}}(\mathbf{q}_j,\mathbf{q}_i),\\ \mathcal{F}_{\text{llfb}}(\mathbf{q}_j,\mathbf{q}_i) &:= \mathbf{T}_{j,i}^{-1}\,\hat{\mathcal{F}}_{\text{llfb}}(\mathbf{T}_{j,i}\,\mathbf{q}_j^{(t)},\mathbf{T}_{j,i}\,\mathbf{q}_i^{(t)}),\\ \hat{\mathcal{F}}_{\text{llfb}}(\hat{\mathbf{q}}_j,\hat{\mathbf{q}}_i) &:= \frac{1}{2}\left(\mathbf{f}(\hat{\mathbf{q}}_j)+\mathbf{f}(\hat{\mathbf{q}}_i)-\Delta x\,\hat{\Psi}_{j,i}+\xi_{j,i}\left(\begin{pmatrix}b_j+h_j\\h_j\hat{u}_j\\h_j\hat{v}_j\end{pmatrix}-\begin{pmatrix}b_i+h_i\\h_i\hat{u}_i\\h_i\hat{v}_i\end{pmatrix}\right)\right).\end{aligned} \tag{8.34}$$

Note that the lake at rest is a quasi-steady state in the transformed space and hence, the method is still well-balanced in 2D.

8.2.3 A Stability Condition for Shallow Water Solvers

Computing the CFL condition [35, 36] for the time step size is straightforward in this scenario. We simply apply the formulas for porous media flow from Section 5.3.5 and obtain the condition

$$\Delta t \leq \frac{V_j}{\sum\limits_{i \in \mathcal{N}(j)} A_{j,i}\, \xi^-_{j,i}}. \tag{8.35}$$

The inbound wave propagation speed $\xi^-_{j,i}$ at the interface (j, i) depends on the choice of the flux solver. For an f-wave solver the smallest eigenvalue of the Roe matrix (8.11) is the maximum ingoing wave speed:

$$\xi^-_{j,i} = \left| \bar{u} - \sqrt{g\, \bar{h}} \right|. \tag{8.36}$$

Hence, to compute the maximum allowed time step size Δt, condition (8.35) must be evaluated on each cell and minimized over all cells.

8.2.4 Pipelined Euler Time Stepping on Finite Volumes in a Single Traversal

Now we are able to implement a time stepping scheme in sam(oa)^2 using the finite volume interface presented in Section 4.1.2. Algorithm 8.2 shows a straightforward implementation that is based on a projection kernel to apply the local cell-to-edge transformations and that stores a representation of the cell data. For each cell-edge pair, the matrix $\mathbf{T}_{j,i}$ projects the quantity vector $\mathbf{q}^{(t)}_j$ to the normal space of each edge, then the geometry-independent net updates are computed with a 1D flux solver $\hat{\mathcal{F}}$, and finally the quantity vector is updated with the accumulated, back-transformed net updates.

While this is a valid approach, it is not necessarily efficient. As stated in Section 4.4.5, the projection kernel is pipelined and will be executed in the previous time stepping traversal for color edges. However, if adaptive mesh refinement is performed between traversals, the projection kernel must also be called also in adaptive traversals on each color edge, causing redundant work. Hence, to improve the performance, as little work as necessary should be performed in the projection kernels.

This gives rise to a second approach based on the 2D flux solver kernel described in Algorithm 8.3. For each cell-edge pair, the quantity vector is copied to the edge, then the net updates are computed on each edge using the 2D flux solver $\mathcal{F}$ and finally the quantity vector is updated with the accumulated net updates. This implementation is more generic than Algorithm 8.2 as any 2D flux solver may now be applied for $\mathcal{F}$. Additionally, the projection kernel no longer executes obsolete Floating point operations when evaluated redundantly.

8.2.5 Well Balanced Refinement

Due to the hyperbolic nature of the shallow water equations, interpolation and restriction of unknowns on dynamically adaptive grids is very similar to the porous media flow scenario in Section 5.3.6 and many of the concepts are applied similarly.

A Posteriori Error Indicator

Starting with the refinement indicator, a relative criterion based on upwind differences of the water height is chosen:

$$\left| \frac{h^{(t+\Delta t)}_j - h^{(t)}_j}{\Delta t} \right| V(\Omega_j) > \mathrm{Tol}_{\dot{h}}\, V(\Omega_{\min}),$$

Algorithm 8.2: Time stepping scheme for the tsunami wave propagation scenario using an approach based on a 1D flux function $\hat{\mathcal{F}}$.

```
Input: $\mathbf{q}^{(t)}$, $\Delta t > 0$
Output: $\mathbf{q}^{(t+\Delta t)}$
traversal
    projection kernel: $\hat{\mathbf{q}}^{(t)}_{j,i} \leftarrow \mathbf{T}_{j,i}\mathbf{q}^{(t)}_j \quad \forall i \in \mathcal{N}(j)$;
    flux solver kernel: $\hat{\mathbf{r}}^{(t)}_{i,j} \leftarrow \hat{\mathcal{F}}\left(\hat{\mathbf{q}}^{(t)}_{i,j}, \hat{\mathbf{q}}^{(t)}_{j,i}\right); \hat{\mathbf{r}}^{(t)}_{j,i} \leftarrow \hat{\mathcal{F}}\left(\hat{\mathbf{q}}^{(t)}_{j,i}, \hat{\mathbf{q}}^{(t)}_{i,j}\right)$;
    net update kernel: $\mathbf{q}^{(t+\Delta t)}_j \leftarrow \mathbf{q}^{(t)}_j + \frac{\Delta t}{V_j} \sum_{i \in \mathcal{N}(j)} A_{j,i}\, \mathbf{T}^{-1}_{j,i}\, \hat{\mathbf{r}}^{(t)}_{j,i}$;
end
```

Algorithm 8.3: 2D flux-based time stepping scheme for the tsunami wave propagation scenario. This implementation is functionally identical to Algorithm 8.2, but more efficient as redundant calls to the projection kernel are cheaper. Internally, the 2D flux function $\mathcal{F}$ may call a 1D flux function $\hat{\mathcal{F}}$ again and apply the corresponding transformations.

```
Input: $\mathbf{q}^{(t)}$, $\Delta t > 0$
Output: $\mathbf{q}^{(t+\Delta t)}$
traversal
    projection kernel: $\mathbf{q}^{(t)}_{j,i} \leftarrow \mathbf{q}^{(t)}_j \; \forall i \in \mathcal{N}(j)$;
    flux solver kernel: $\mathbf{r}^{(t)}_{i,j} \leftarrow \mathcal{F}\left(\mathbf{q}^{(t)}_{i,j}, \mathbf{q}^{(t)}_{j,i}\right)$; $\mathbf{r}^{(t)}_{j,i} \leftarrow \mathcal{F}\left(\mathbf{q}^{(t)}_{j,i}, \mathbf{q}^{(t)}_{i,j}\right)$;
    net update kernel: $\mathbf{q}^{(t+\Delta t)}_j \leftarrow \mathbf{q}^{(t)}_j + \frac{\Delta t}{V_j} \sum_{i \in \mathcal{N}(j)} A_{j,i}\, \mathbf{r}^{(t)}_{j,i}$;
end
```

for an arbitrary, but fixed $\mathrm{Tol}_{\hat{h}} > 0$. Considering the discrete update rule in (8.26), the left-hand-side effectively translates to the flux divergence. This criterion is an empirical choice with the goal of refining cells with a high inflow or outflow, indicating areas of strong change, and coarsening cells with a static water height. With this choice, wave fronts will be refined and coarsening will mainly occur for lakes at rest.

Well Balanced Interpolation and Restriction

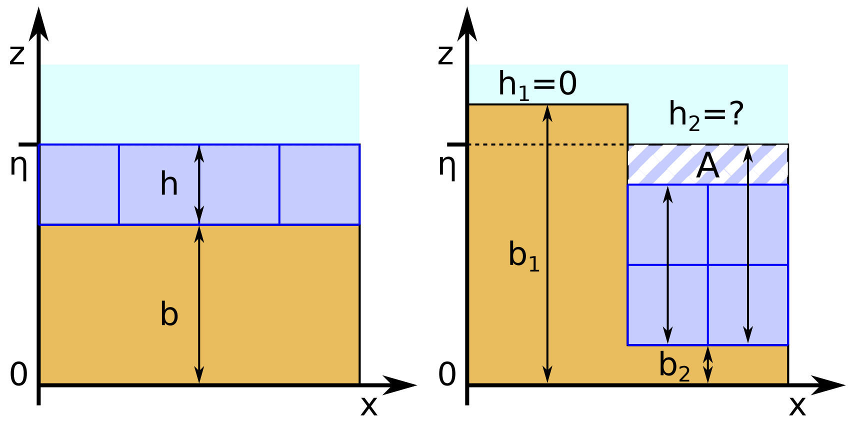
In order to refine and coarsen cell data, we will use a method based on mass and momentum conservation as in [48]. Most importantly, some quasi-steady states are preserved during refinement and coarsening in order to keep errors local. If, for example, we refined a lake-at-rest solution and did not preserve the quasi-steady state in (8.16), we would generate waves that propagate from the erroneous state and introduce a global error in the solution. Clearly, this must be avoided.

Consider a 1D scenario, where we refine a single 1D cell with state $(\hat{h}, \hat{h}\hat{u}, \hat{b})^T$ and volume $\hat{V}$. When splitting the cell into two cells with the states (h_1, h_1u_1, b_1) and (h_2, h_2u_2, b_2) and volumes $V_1 = \frac{1}{2}\hat{V}$ and $V_2 = \frac{1}{2}\hat{V}$, respectively, discrete mass conservation requires

$$\hat{h}\hat{V} = h_1V_1 + h_2V_2 = \tfrac{1}{2}(h_1 + h_2)\hat{V}. \tag{8.37}$$

Similarly, discrete momentum conservation implies

$$\left(\hat{h}\hat{u}\right)\hat{V} = \tfrac{1}{2}(h_1u_1 + h_2u_2)\hat{V}. \tag{8.38}$$



(a) Before refinement **(b)** After refinement

Figure 8.2: An example for adaptive refinement of cell states, where mass conservation and quasi-steady-state preservation are mutually exclusive, even though (8.40) is satisfied. A coarse 1D cell in **(a)** with bathymetry b and water height h, is split into two cells after refinement **(b)**, one of which is dry. In this case, the lake at rest is preserved if and only if area A in **(b)** is filled with water, i.e. $h_2 = \eta - b_2$. Mass conservation is fulfilled if and only if the striped area is left empty, i.e. $h_2 = 2h < \eta - b_2$.

According to (8.7) a necessary condition to preserve quasi-steady states is that the lake at rest has a constant surface elevation η_{eq}. Hence,

$$\eta_{\text{eq}} = \hat{b} + \hat{h} = b_1 + h_1 = b_2 + h_2, \tag{8.39}$$

which is a refinement condition for the water heights h_1 and h_2. At this point, we observe a problem, as satisfying conditions (8.37) and (8.39) simultaneously implies

$$\hat{b} = \tfrac{1}{2}(b_1 + b_2). \tag{8.40}$$

This condition is not easily satisfied, because bathymetry data is external and only defined on the fine scale. Hence, when bathymetry data is set in a coarse cell, the value must be chosen in such a way that condition (8.40) is satisfied if the cell is further refined. The problem is very similar to porosity refinement in Section 5.3.6, and indeed the solution is the same. We apply volume-weighted averaging to the bathymetry data. Algorithm 8.4 shows a 2D implementation of the cell state adaption scheme that uses an integration kernel for bathymetry refinement.

In wet cells, the full scheme conserves mass and momentum as expected. At coasts however, mass conservation or the quasi-steady state can be violated when a cell is dried or flooded during adaptive refinement, as shown in Figure 8.2. As violation of mass conservation triggers only a local error, we accept it as the less destructive solution. In tsunami simulation, most of the domain is de facto in a lake at rest, thus preserving the state is necessary for stability of simulations on dynamically adaptive grids. However, other quasi-steady states are not preserved during adaptive refinement. This problem was investigated e.g. in [38] with focus on dry zones and will not be considered further.

8.2.6 Extension to a Shallow Water Model with Hydrodynamic Pressure

One of the basic assumptions of the shallow water model is that anywhere in the domain, the wave lengths are much bigger than the water height. This condition allows to neglect hydrodynamic pressure effects without significant changes in the solution. Near coastal regions however, the assumption does

Algorithm 8.4: 2D cell state interpolation and restriction. Bathymetry coarsening is realized by averaging, transfer by copying and refinement by numerical integration over the destination element via recursive newest vertex bisection. Here, $\hat{\mathbf{q}}$ is the state vector of the old grid, $\mathbf{q}$ is the state vector of the new grid, d_j is the refinement depth of cell j, d_{src} is the refinement depth of the source data and $b(\mathbf{x})$ is the bathymetry source, defined in local coordinates. Water height and momentum use simpler rules based on the corresponding conservation laws.

```
Input: $\hat{\mathbf{q}}_j = \left(\hat{h}_j, \hat{h}_j\hat{u}_j, \hat{h}_j\hat{v}_j, \hat{b}_j\right)^T$ for $j = 1, 2, \ldots, \hat{n}$
Output: $\mathbf{q}_j = \left(h_j, h_j u_j, h_j v_j, b_j\right)^T$ for $j = 1, 2, \ldots, n$

Function integrate ($f$, $\mathbf{x}_1$, $\mathbf{x}_2$, $\mathbf{x}_3$, $d$)
    if $d > 0$ then
        return $\frac{1}{2}$integrate ($f$, $\mathbf{x}_1$, $\frac{1}{2}(\mathbf{x}_1 + \mathbf{x}_3)$, $\mathbf{x}_2$, $d - 1$)
        $+\frac{1}{2}$integrate ($f$, $\mathbf{x}_2$, $\frac{1}{2}(\mathbf{x}_1 + \mathbf{x}_3)$, $\mathbf{x}_3$, $d - 1$) ;
    else
        return $f(\frac{1}{3}(\mathbf{x}_1 + \mathbf{x}_2 + \mathbf{x}_3))$;
    end
end

traversal
    transfer kernel: $(h_j, h_j u_j, h_j v_j, b_j)^T \leftarrow \left(\hat{h}_j, \hat{h}_j\hat{u}_j, \hat{h}_j\hat{v}_j, \hat{b}_j\right)^T$;
    coarsen kernel: $(h_j, h_j u_j, h_j v_j, b_j)^T \leftarrow \sum_i \frac{V(\Omega_j \cap \hat{\Omega}_i)}{V(\Omega_j)} \left(\hat{h}_i, \hat{h}_i\hat{u}_i, \hat{h}_i\hat{v}_i, \hat{b}_i\right)^T$;
    refine kernel:
    $(h_j, h_j u_j, h_j v_j)^T \leftarrow \sum_i \frac{V(\Omega_j \cap \hat{\Omega}_i)}{V(\Omega_j)} \left(\hat{h}_i, \hat{h}_i\hat{u}_i, \hat{h}_i\hat{v}_i\right)^T$;
    $b_j \leftarrow$integrate ($b$, $(1,0)^T, (0,0)^T, (0,1)^T, d_{\text{src}} - d_j$) ;
end
```

not hold any longer, as tsunami waves typically grow, while their wavelength decreases when they approach the shore.

The addition of a hydrodynamic pressure term becomes necessary to mitigate the model error [57]. Hence, Samfass [101] extended the scenario in sam(oa)^2 with a non-hydrostatic pressure model that adds the solution of a linear system of equations for a pressure term to each time step. Schaller [106] extended the previous work by parallelizing the implementation to allow execution of larger scenarios on supercomputers. While the addition of non-hydrostatic pressure significantly increases the time-to-solution, it was shown that for certain benchmark scenarios, the simulation returned far more accurate results.

8.3 Conclusion: Tsunami Simulation on Sierpinski Grids

The implementation described in this chapter is able to model the core principles which are necessary for tsunami simulation, i.e. the conservation of mass, momentum balance, influence of gravity, and source terms for initial displacements as well as bathymetry data. As such, the scenario is sufficient for tsunami simulation but has limitations, because non-hydrostatic pressure, bottom friction, and Coriolis forces are not considered. The first issue was addressed in a student thesis, the other two were neglected in this scenario due to their expected minor impact on performance as additional right-hand-side terms. Moreover, sam(oa)^2 also imposes restrictions on the extensibility of the scenario. The element-wise formulation forbids flux functions with stencils that span more than direct neighbor elements. Consequently, higher order finite volume schemes or complex limiters may require extensions to the interface. Discontinuous Galerkin approaches are supported though and currently implemented in the ASCETE project [6].

9 Numerical Analysis of Tsunami Wave Propagation

In this chapter, a brief numerical analysis of the tsunami wave propagation scenario in Chapter 8 is performed. Many of the applied methods are similar to porous media flow. Hence, this chapter contains only a short recapitulation of a numerical study of Riemann problems for the shallow water equations, where results of sam(oa)^2 and simulations on Cartesian grids are compared. Afterwards, a more complex production scenario is investigated, where the simulation output is verified on buoy measurements from a real tsunami.

9.1 Riemann Solutions on Cartesian Grids and Sierpinski Grids

Similarly to the analysis of the porous media flow scenario, the tsunami wave propagation scenario can be numerically evaluated on analytic solutions to Riemann problems. As part of her Bachelor's thesis, Rodrigues Monteiro [97] evaluated numerical results of a Riemann problem for the 1D shallow water model without bathymetry described in (8.1). Comparison of a 2D simulation in sam(oa)^2 to the analytic solution and to equivalent simulations on Cartesian meshes in the teaching code SWE [120] returned good agreement. Furthermore, convergence to the analytic solution with increasing refinement depths of adaptive meshes in sam(oa)^2 was demonstrated. The convergence rate was shown to be better than on a Cartesian mesh.

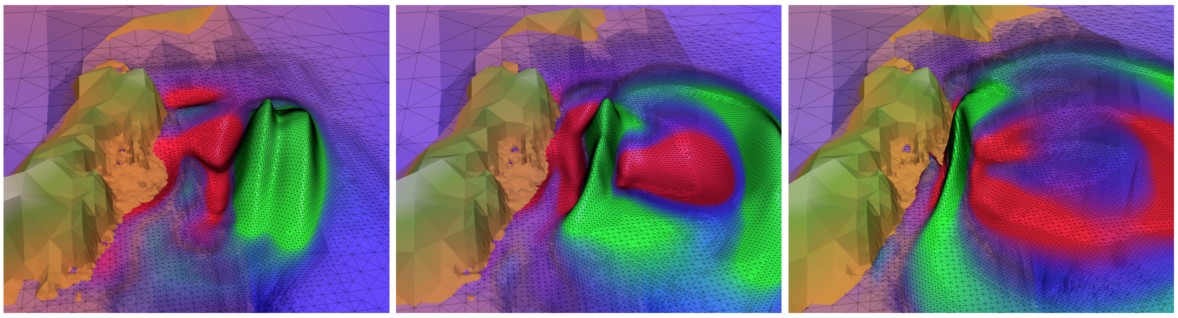
9.2 The Tohoku Tsunami in 2011

The second test case for the tsunami wave propagation scenario is the Tohoku tsunami from 2011 that was caused by tectonic shift and was preceded by an earthquake of magnitude 9 near the coast of Japan. We used the GEBCO_2014 Grid, version 20150318 [47], as bathymetry data of the northern Pacific ocean and the Sea of Japan in order to obtain a height map of the sea floor. The data is transformed from spherical to Cartesian coordinates by an azimuthal equidistant projection. As origin, we chose the earthquake hypocenter and thus, the propagation speed of the tsunami is fairly accurate.

The initial condition for the simulation is given by a lake at rest with height 0. Following Okada's model [85], a static displacement of the bathymetry and the water surface, adjusted for tidal waves, is applied to simulate vertical plate movements [56]. As shown in Figure 9.1, sam(oa)^2 simulates the subsequent tsunami propagation, as well as the inundation of the Japanese coast.

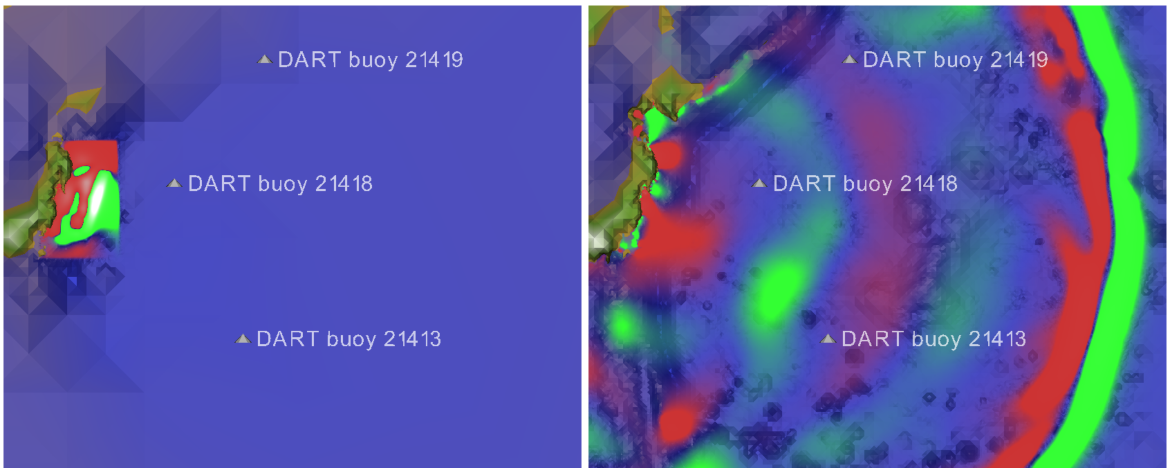
We compared our results with the corresponding water displacements measured by DART (Deep-ocean Assessment and Reporting of Tsunamis) buoys [84]. Figure 9.2 shows the placement of the buoys in the domain and the propagation of the tsunami waves. The grid size varies between 22k and 140k elements during the simulation, as the wavefront is progressively refined while it propagates through the domain.

Tests with varying refinement depths in Figure 9.3 show that with increased resolution, the wave features become more prominent, resulting in higher amplitudes, while the frequencies of the waves



(a) Time: 2 minutes a.e. **(b)** Time: 10 minutes a.e. **(c)** Time: 20 minutes a.e.

Figure 9.1: Wave propagation of the Tohoku tsunami near the coast of Japan 2011. An adaptively refined mesh of the Japanese sea and the coast of Japan is used to simulate 3 hours of wave propagation after the initial earthquake. The water elevation is exaggerated by a factor of 100 for better visibility. Green and red colored areas mark positive elevations above 1m and negative elevations below 1m, respectively. Initially, an earthquake causes a vertical fault displacement that lifts the body of water above the Pacific plate. The subsequent tsunami wave propagates radially from the fault and is simulated in sam(oa)^2. Snapshots are denoted by the elapsed time since the beginning of the earthquake.



(a) Time: 20 seconds a.e. **(b)** Time: 3 hours a.e.

Figure 9.2: Initial condition and final state in a simulation of the Tohoku tsunami from 2011. An adaptively refined mesh of the Japanese sea and the coast of Japan is used to simulate 3 hours of wave propagation after the initial earthquake. Green and red areas have a positive elevation above 0.1m and a negative elevation below 0.1m, respectively. The three DART near the coast of Japan provide measurement data [84] that was used for verification of simulated results. Snapshots are denoted by the elapsed time since the beginning of the earthquake.

remain similar. At depth 26, the bathymetry data is fully resolved and increased resolutions do not affect the solution anymore. The agreement with the measured data from DART buoys is good up to 2.5 hours, i.e. time step 70.34, of simulation time after the earthquake. Then, the DART buoy 21418 reports later arrival of a coastal reflection wave than the simulation predicts.

Figure 9.4 shows that the choice of the flux solver among those described in Section 8.2.2 has a comparably small influence on the simulation outcome. While the extended local Lax-Friedrichs solver introduces a small artificial diffusion, the f-Wave solver and the augmented Riemann solver do not show any significant differences. This behavior can be explained by measuring data far away from the shore, as the inundation model provided by the augmented Riemann solver will mostly affect the coastal flow.

Sources for errors are two-fold. On the one hand, simulation data is erroneous due to various model and input data restrictions. For instance, the model neglects bottom friction, as described in Section 8.3. Moreover, displacement data originates from an empirical study and is not generated from actual earthquake measurements [56]. Thus an uncertain, incomplete initial condition is provided for the time stepping scheme. On the other hand, buoy data is affected by noise such as surface waves, which cause high frequency oscillations in the measurements.

9.3 Conclusion: Riemann Problems and the Tohoku Tsunami

This chapter shows that the scenario implementation solves Riemann problems accurately for the shallow water equations and that plausible solutions are returned for a real tsunami wave propagation scenario. Reflected waves are not correctly captured which is attributed to restrictions in the model and the input data. As the discretization in use is fairly simple, it is not further verified and we continue with performance analysis in Chapter 10.

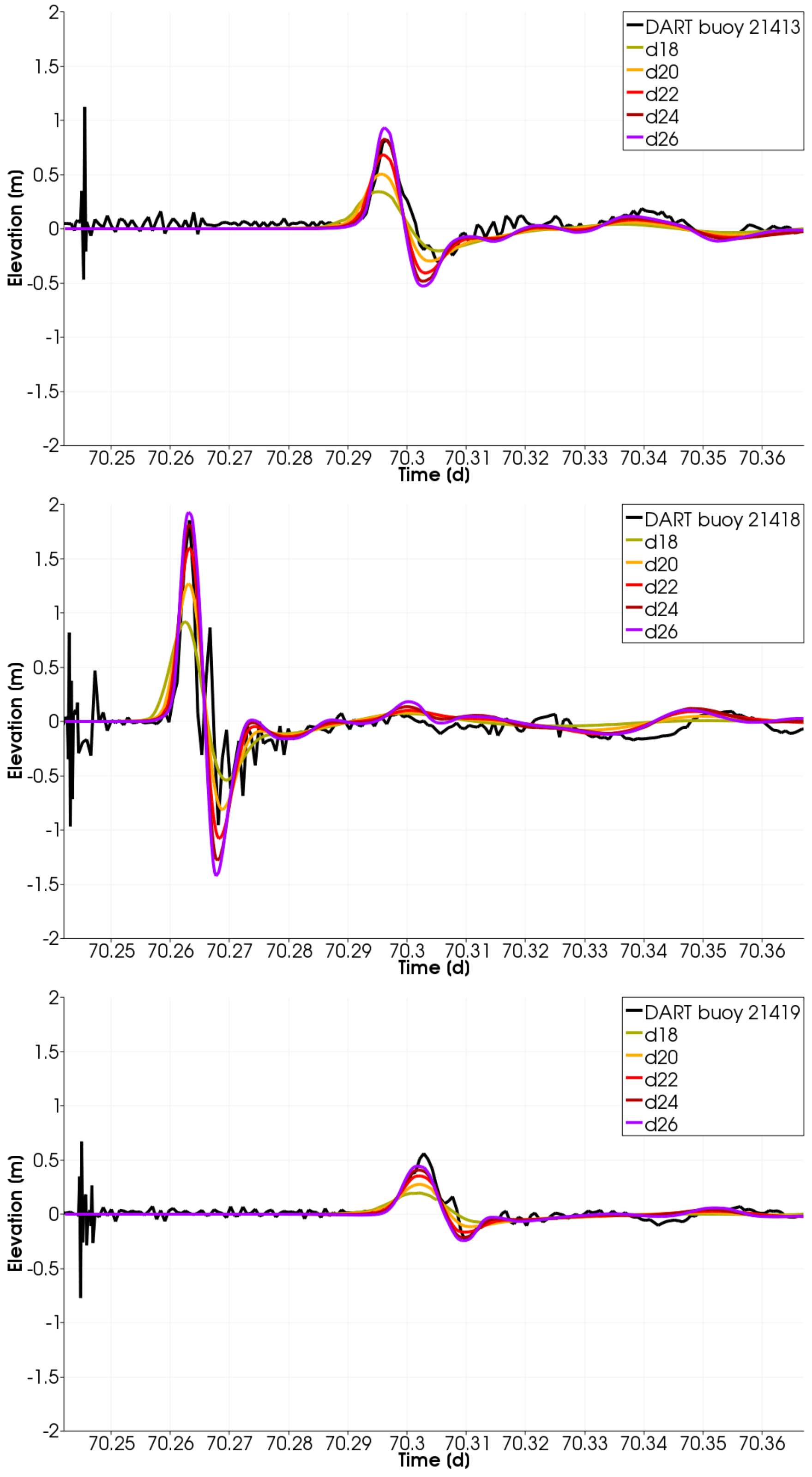


Figure 9.3: A comparison of measured and simulated water elevation, defined as the difference to standard sea level. The horizontal axis denotes time in days since New Year's Day. From top to bottom, results with the augmented Riemann solver and varying maximum refinement depths are compared at the location of DART buoys 21413, 21418, and 21419 [84].

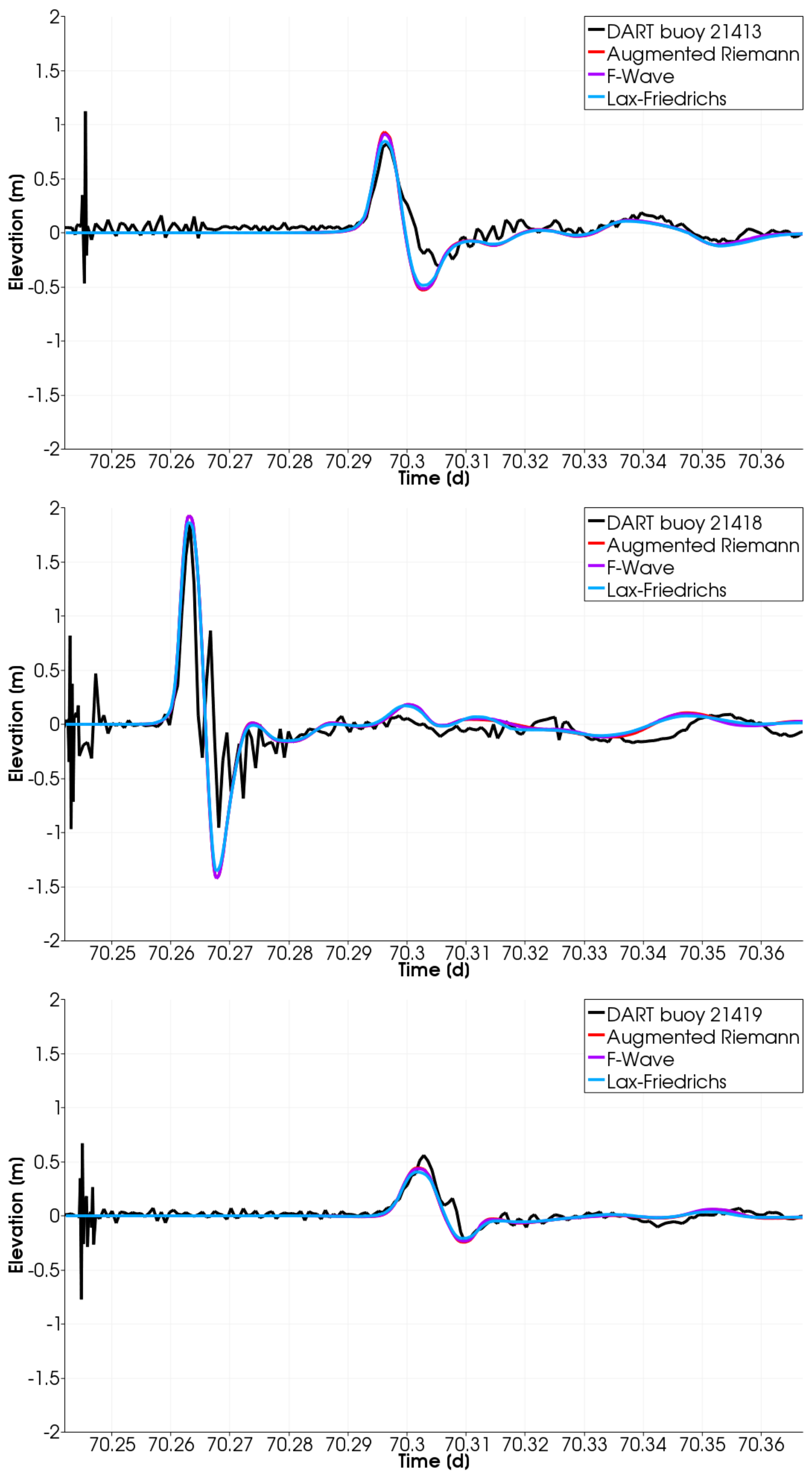


Figure 9.4: A comparison of measured and simulated water elevation, defined as the difference to standard sea level. The horizontal axis denotes time in days since New Year's Day. From top to bottom, results with a fixed maximum refinement depth 26 and three different flux solvers are compared at the location of DART buoys 21413, 21418, and 21419 [84].

10 Performance of Tsunami Wave Propagation

The tsunami wave propagation scenario offers different challenges in performance optimization than the porous media flow scenario due to its heterogeneous flux solver kernels and a much higher remeshing frequency. The computation in each time step is not memory-bound in general and therefore, static traversal overhead is not as high. Patch-based approaches [28, 127] show good performance results for this type of problem; however, they sacrifice fine-granular refinement and coarsening, which we strive to support.

The augmented Riemann solver as described in Chapter 8.2.2 is heavily branched and produces different workload depending on space and time. The load balancing algorithms of sam(oa)^2 in Chapter 3.2 are designed to handle this behavior and are examined here. Time steps are explicit and their size matches the propagation time of the fastest wave in a cell. Hence, remeshing is required in each time step in order to track the fastest signals. Therefore, we investigate the overhead caused by conformity, adaptive grid traversals, and load balancing to determine the impact of adaptive mesh refinement on the solver.

Simulation parameters are chosen to match an environment similar to the conditions of a production run. Hence, in all test cases simulations on real tsunami data with adaptive mesh refinement and load balancing are executed. The target system is the SuperMUC thin node cluster [119]. Some of the performance tests presented here have already been published in [77].

10.1 Load Balancing Techniques

While the porous media flow scenario produces mostly uniform load due to its homogeneous kernels, the tsunami wave propagation scenario is expected to behave differently. Some of the Riemann solvers described in Chapter 8.2.2 use early exit conditions for dry cells, implying that kernel execution times will vary strongly depending on the location in the domain. Hence, this scenario is expected to benefit from the advanced load balancing techniques in Chapter 3.2, which are designed for management of heterogeneous load. All experiments were conducted on the SuperMUC [119] thin node cluster to obtain data from actual production-ready simulation runs and to investigate how performance is affected by heterogeneity. The only flux solver used in these tests is the augmented Riemann solver, which is described in Chapter 8.2.2. Its advanced model render it the best choice for production runs. At the same time, it is heavily branched and has the least predictable and most heterogeneous performance of the flux solvers available in sam(oa)^2.

10.1.1 Work Stealing in Shared Memory

In shared memory, sam(oa)^2 supports both work stealing and work sharing as load balancing algorithms, which are realized by OpenMP task constructs around static load assignment, as presented in Chapter 3.2.5. In general, work stealing returns a better performance than work sharing, as shown by Table 10.1. The time-to-solution improves by 10% when work stealing is turned on. When the number

Sections per core	Work sharing	Work stealing
16	78.19 s	71.37 s
32	82.28 s	74.24 s

Table 10.1: Exemplary time-to-solution for a tsunami wave propagation scenario with 1M elements executed on a single node of the SuperMUC thin node cluster using OpenMP parallelization. Work sharing and work stealing are both applied. The number of sections per core is increased from 16 to 32 to test how a more fine-grained partitioning affects work stealing.

of sections per core is increased, the additional traversal and scheduling overhead prevents further gain of the execution time. Hence, for scenarios with small section sizes and uniform load, work sharing will be the better choice, as it has much less algorithmic overhead. Larger scenarios will benefit more from work stealing.

10.1.2 Load Balancing Strategies for Hybrid Parallelization

To understand how each combination of load balancing techniques affects performance, we will analyze a test case thoroughly. A tsunami scenario with approximately 1 billion elements is executed on 512 cores of the SuperMUC thin node cluster, where the initial grid refinement phase and the first time steps are neglected due to their heavy imbalances.

In general, there is no best setup that is suitable for all problems as the performance strongly depends on the simulation properties, but the advantages and disadvantages of the available techniques are discussed to determine where they are beneficial. Some combinations that do not make any sense, e.g. section splitting with optimal chains-on-chains partitioning, are neglected.

Figure 10.1 and Figure 10.2 show component analyses for hybrid and pure MPI parallelization with different choices for the cost model, the load distribution algorithm, and with or without section splitting. The ratio of time stepping to the remaining components differs in the two plots, because remeshing and load balancing were scheduled in every time step for the first plot, and only in every tenth time step for the second plot.

Surprisingly, node-level hybrid parallelization performs worse than socket-level and pure MPI parallelization in both plots. Due to reduced MPI communication, less computational effort, and enabling OpenMP work stealing, it should be the other way around. Figure 10.1 shows that this is mainly caused by an imbalance of adaptive traversals in shared memory. Synchronization after adaptive traversals is scheduled only during the neighbor search and hence the apparent execution time increases when hybrid parallelization is enabled. This is also the reason, why in the first test case in Figure 10.1, pure MPI performs best. In Figure 10.2, remeshing is reduced to every tenth time step and socket-level hybrid parallelization is fastest.

Cost models

Time-based cost evaluation does not appear to improve the performance in some cases of Figure 10.1, which is caused by running into the problem of chains-on-chains partitioning with intermediate synchronization, which is explained in Chapter 3.3.4. As time steps and remeshing are two comparably expensive phases with different load distributions, it is hard to balance them properly and sam(oa)^2 sometimes fails to do so. This issue is much less problematic when remeshing is carried out less often, as seen in Figure 10.2. Here, time-based cost evaluation almost always performs better than the linear cost model. More precisely, time-based cost evaluation usually returns a better performance if there is a dominant heterogeneity in the process load. Otherwise, the linear cost model is the better choice.

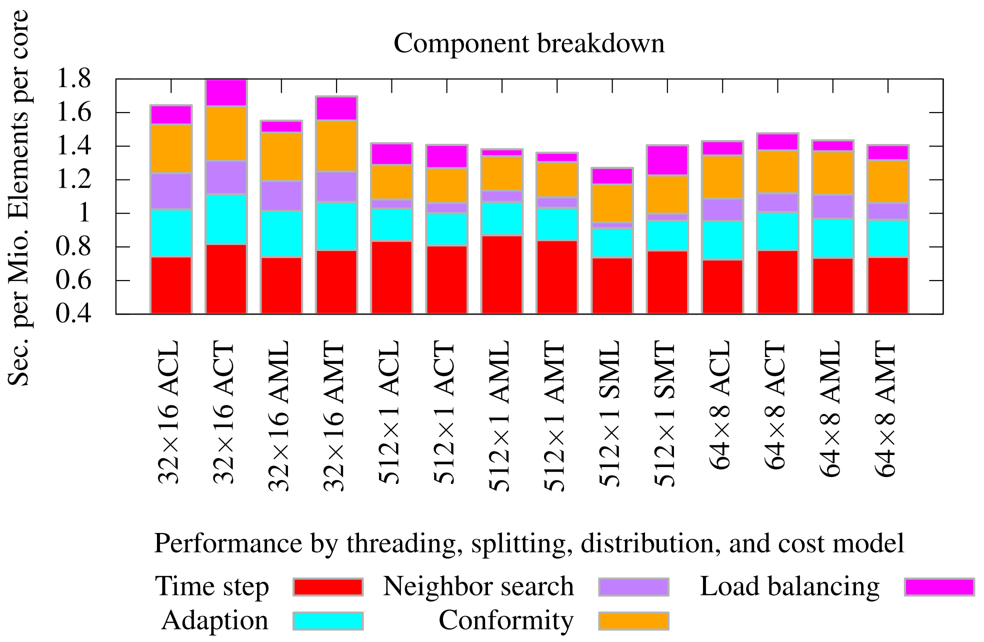


Figure 10.1: Comparison of execution times for different load balancing strategies on 512 cores. Remeshing and load balancing are performed in every time step. In each label, $p \times t$ is the number of MPI processes p and OpenMP threads per process t. Section splitting is indicated by S, atomic sections by A, optimal chains-on-chains partitioning by C, midpoint approximation by M, time-based cost evaluation by T, and the linear cost model by L. The simulation is divided into the advective transport step (Time step), conformity correction (Conformity), remeshing (Adaption), regeneration of communication structures (Neighbor search) and load balancing (Load balancing).

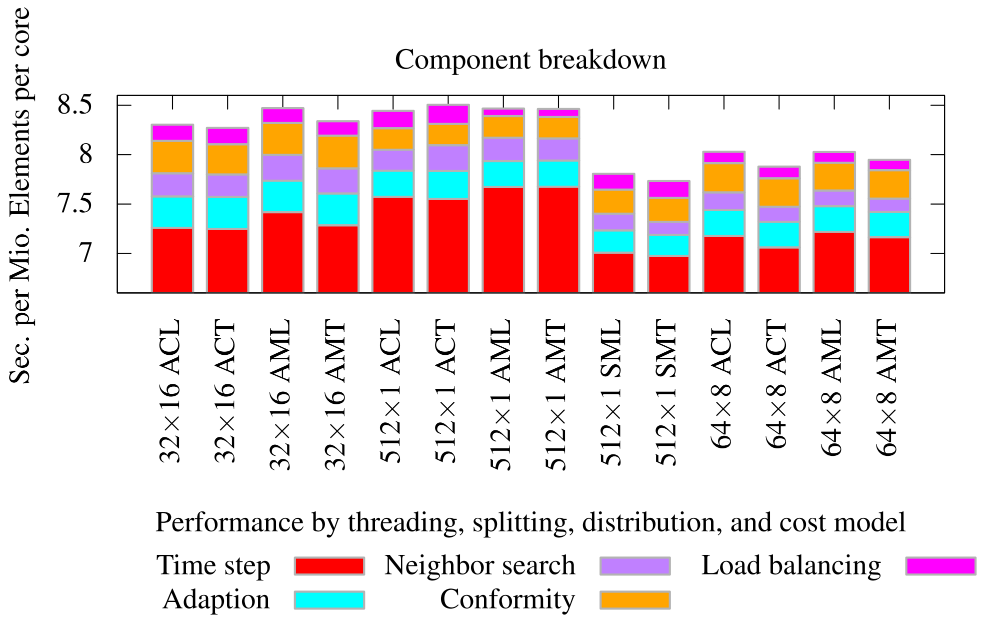


Figure 10.2: Comparison of load balancing strategies similar to Figure 10.1. Remeshing and load balancing are performed in every tenth time step. In each label, $p \times t$ is the number of MPI processes p and OpenMP threads per process t. Section splitting is indicated by S, atomic sections by A, optimal chains-on-chains partitioning by C, midpoint approximation by M, time-based cost evaluation by T, and the linear cost model by L.

Distribution Algorithms

As discussed in Section 3.3.1, an optimal chains-on-chains solver is inherently serial and does not scale on large systems. In Figure 10.1 the benefit of better distributions is mostly outweighed by the additional load balancing time for pure MPI parallelization. With hybrid parallelization enabled, load is not always balanced better when the optimal solver is used. This is caused by side effects of the load balancing algorithm, which becomes expensive enough to cause an imbalance itself, which only affects a single thread per process. Figure 10.2 shows that with less frequent remeshing, the optimal solver indeed always returns the best performance. Hence, on small clusters or problems without frequent remeshing, optimal chains-on-chains partitioning is the best distribution algorithm. Otherwise, midpoint approximation is recommended.

Section Splitting and Atomic Sections

In both plots, section splitting works very well for this scenario, which is mostly due to the small problem size. When the number of cores exceeds 512, each core only holds around 100k elements and benefits strongly from decreased traversal overhead. With enabled section splitting, the number of sections per core is decreased to 4, as load balancing will return a perfect balance independent of the parameter. As mentioned in Section 3.2.2, only MPI parallelization can benefit from this, as during hybrid parallelization a higher number of sections is assigned per process. Hence, according to Section 3.3.2 a good load balance in distributed memory is guaranteed. Splitting will lose its effect on larger numbers of cores, as a smaller number of sections per core will cause a decreased overlapping of synchronization and communication. Increasing the number of sections per core fixes this issue, but also mitigates the advantage of splitting. Hence, on large clusters, it makes more sense to switch to atomic sections.

10.1.3 Load Distribution Algorithms on Cartesian Grids

In [105] Schaller compared the quality of chains-on-chains partitioning algorithms to the longest processing time algorithm. Tests were implemented in SWE [120], a parallel finite volume solver for the shallow water equations that parallelizes Cartesian meshes via blocking. Schaller used the augmented Riemann solver described in Section 8.2.2 to generate heterogeneous load and showed that the longest processing time algorithm returned the best balance in general. It is worth mentioning that the chains-on-chains solvers were not far behind and proved to be competitive.

10.2 Impact of Adaptive Mesh Refinement

In the tsunami wave propagation scenario, performance of time stepping is compute-bound, as the augmented Riemann solver used for flux computations has a high computational intensity, which, in contrast to the porous media flow scenario, is hardly affected by memory throughput. This is confirmed in [11], where a detailed performance analysis of the flux solver was conducted. For their analysis, Bader et al. used SWE [120], where a single precision, slightly simpler variant of the GeoClaw solver than the version by sam(oa)^2 is provided, i.e. the solver `AugRieFun`. The flux solvers in sam(oa)^2 and SWE are both not vectorized and should return similar execution times for the same setup. Hence, they are suitable for a performance comparison. Apart from SWE, we also compared performance to GeoClaw, which was analyzed by Malcher [72] on a single node of the Stampede supercomputer [117]. Stampede nodes have the same hardware configuration as SuperMUC thin nodes, hence, performance results are directly comparable.

SWE solves the tsunami scenario on a static Cartesian mesh, thus we use it to estimate the overhead of dynamically adaptive execution of the tsunami scenario in sam(oa)^2. To measure performance, we count Riemann solutions per second, which are the total number of calls to the flux solver over the wall-clock time of the simulation. Hence, in all three codes, all components that affect the time-to-solution are considered.

In single precision and serial execution, sam(oa)^2 achieves 3.2M Riemann solutions per second. On the same system and for a similar problem, SWE is able to execute 6.8M Riemann solutions per second. GeoClaw achieves 6.9M Riemann solutions per second, which is roughly the same number. Hence, SWE and GeoClaw are twice as fast as sam(oa)^2. In order to understand this ratio we conducted a component analysis that breaks down the total execution time of sam(oa)^2 into parts of the framework. Figure 10.3 shows the results of a simulation on a single core and a single thin node of the SuperMUC system with pure MPI, pure OpenMP, and socket-level hybrid parallelization.

Excluding communication, sam(oa)^2 spends 60% of its execution time on pure computation in the time step on a single core and 52% on a thin node consisting of 16 cores. This ratio decreases further when parallelism is increased. On 8,192 cores, i.e. a full SuperMUC thin node island, only 37% of the total execution time is used for computation. Hence, a significant fraction of time is required for dynamical adaptivity, communication and load balancing, causing most of the performance loss. The influence of dynamical adaptivity on the performance is presented in a weak scaling study in Figure 10.4. There, pure MPI and hybrid parallelization both have an almost linear speedup on a static grid in the range of 16 to 8,192 cores, whereas the performance on a dynamic grid starts dropping at 1,024 cores, which is mostly caused by adaptive mesh refinement.

To understand whether the shown results are competitive, we compare normalized execution times of remeshing to some other known adaptive mesh refinement packages in the field. Note that these results are only qualitative, as the performance is strongly affected by the configuration of the target machine, structure and shape of grid elements, as well as grid dynamics. Still, Table 10.2 shows that sam(oa)^2 performs well compared to SAMRAI and p4est.

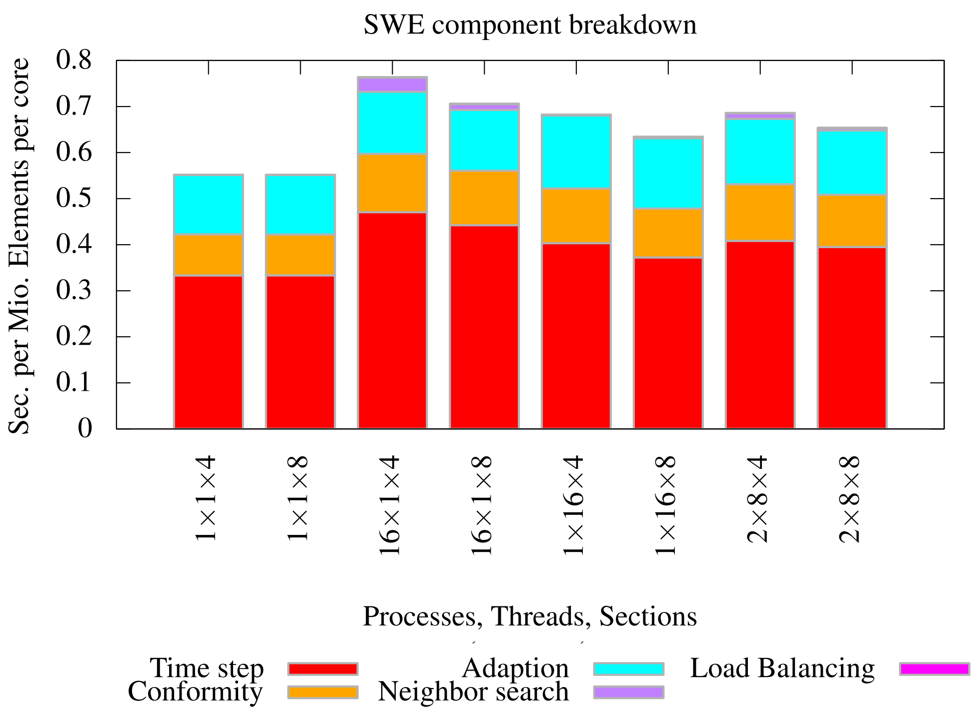
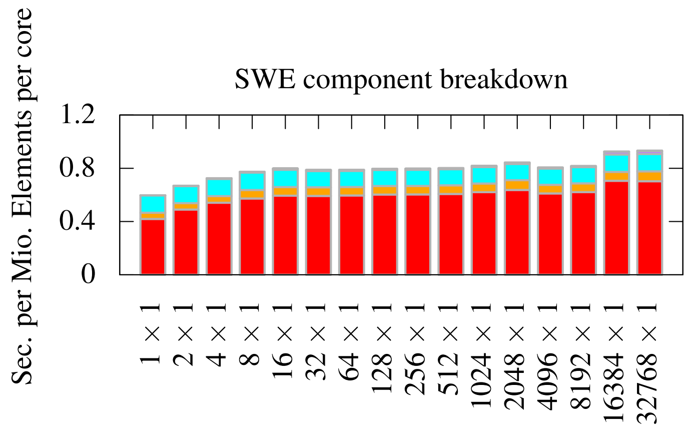


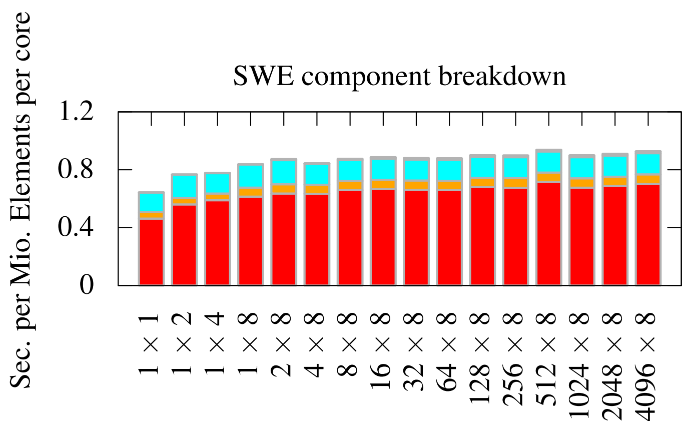
Figure 10.3: Component analysis for the execution of the tsunami wave propagation scenario on a single core and a single node with pure MPI, pure OpenMP, and socket-level hybrid parallelization. The label $p \times t \times s$ is an abbreviation for p processes, t threads per process, and s sections per thread.

Code		Machine	Conformity	Refinement	Load Balancing	Total
SAMRAI		Sequoia				0.60s
deal.II (p4est)		Ranger		2s	1s	3s
sam(oa)^2	static	SuperMUC	0.22s	0.13s	0.01s	0.39s
	dynamic	SuperMUC	0.18s	0.25s	0.05s	0.48s

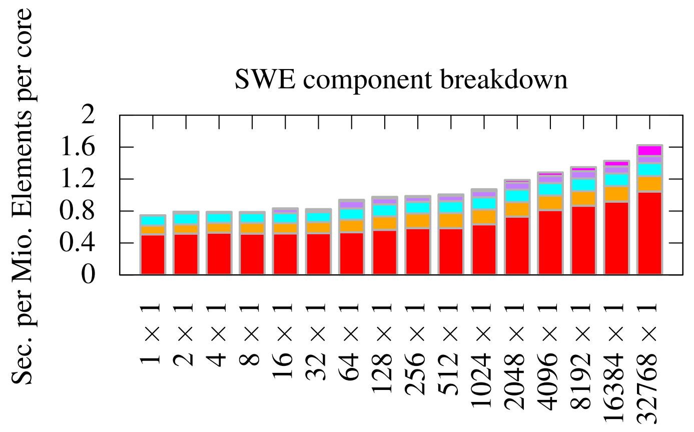
Table 10.2: Remeshing and load balancing execution times for sam(oa)^2 and other adaptive mesh refinement codes. Execution times are normalized to 1M degrees of freedom per core. Based on p4est, a mantle convection problem was solved in deal.II [16, 17] on the Ranger supercomputer [94]. SAMRAI [52,53] is a software package for structured adaptive mesh refinement and was executed on Sequoia [113], a BlueGene/Q supercomputer which is ranked fourth in the Top500 list of June 2016 [122].



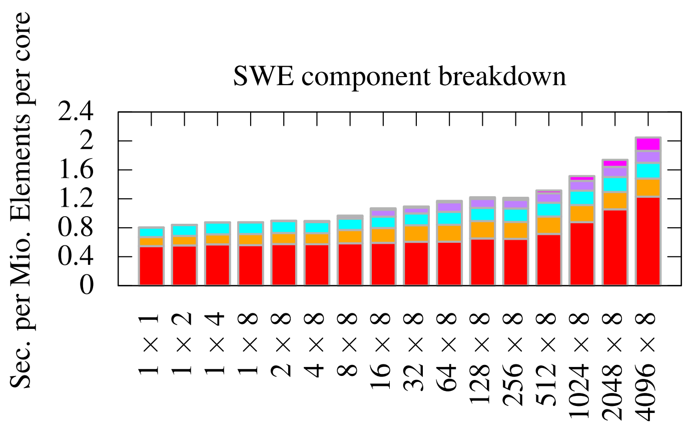
(a) Processes × threads for pure MPI, static grid



(b) Processes × threads for OpenMP+MPI, static grid



(c) Processes × threads for pure MPI, dynamic grid



(d) Processes × threads for OpenMP+MPI, dynamic grid

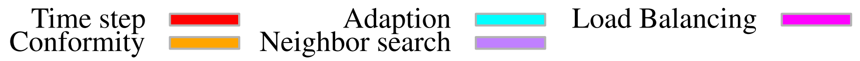


Figure 10.4: Impact of dynamically adaptive mesh refinement on performance of a weak scaling study on the SuperMUC thin node cluster. Results for pure MPI and socket-level hybrid parallelization are compared for a single time step on a static grid and 25 time steps on a dynamically adaptive grid that grows by 10% during the measurement.

Code	Interface ratio	Edge ratio
p4est (3D)	1.90	4.80
sam(oa)^2	1.64	1.07

Table 10.3: Partition quality for sam(oa)^2 and a 3D execution of p4est. Given are the ratios of partition interfaces to an ideal, theoretical block-structured partitioning. The optimal value is 1 in all cases.

As partitions are generated on-the-fly with fast heuristics, they tend to be worse than those of high-quality algorithms. Hence, we investigate the number of process boundary interfaces and process boundary edges per partition size compared to a block-structured partitioning in Table 10.3. The ideal number of boundary interfaces i_{id} is defined as the total number of vertex-connected neighbors for p square-shaped partitions, that is

$$i_{\text{id}} := 8p.$$

In this estimate, we assume that p is large enough to be able to ignore the influence of the domain boundary. The ideal number of boundary edges e_{id} is defined as the total number of edges for p square-shaped partitions that consist of $\frac{f}{p}$ triangles each. More precisely,

$$e_{\text{id}} := 4p\sqrt{\frac{f}{2p}}.$$

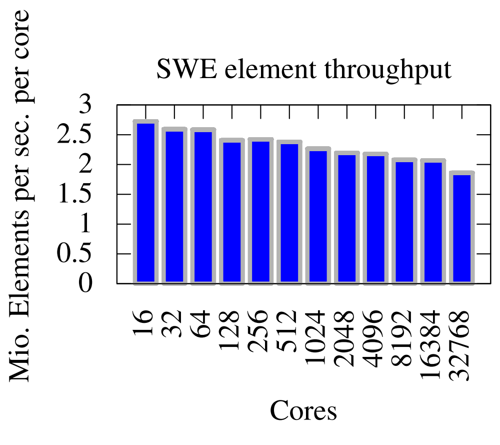
Here, consistent values for grids with a sufficient size and different degrees of parallelism are returned by sam(oa)^2. For reference, the values of a similar study on p4est [30] are included in Table 10.3. Again, the results are not directly comparable, as p4est was executed in 3D and is based on non-conforming grids, whereas sam(oa)^2 was executed in 2D on conforming grids. However, the data may be used as upper bounds for sam(oa)^2, which handles the easier 2D case. As the interface and the edge ratio both are comparably low, we deduce that the continuity of the underlying space-filling curve strongly promotes the formation of well-formed partitions in sam(oa)^2.

10.3 Scalability on Multiple Distributed Nodes

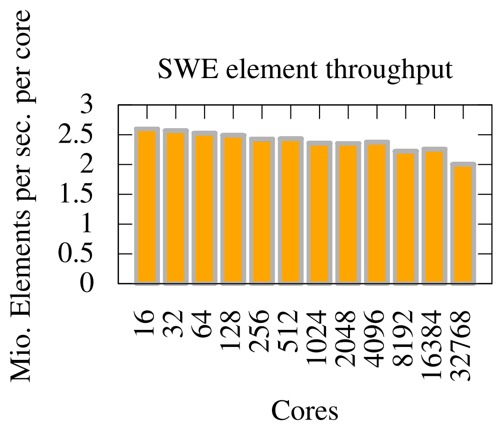
Our final study is a scalability analysis on multiple nodes of the SuperMUC thin node cluster. As for the porous media flow scenario, a weak scaling test is performed first to observe how load balancing and adaptive mesh refinement combined with global communication are affected by increased parallelism. Next, a strong scaling study shows how the additional increase of local communication will affect performance.

Weak scaling

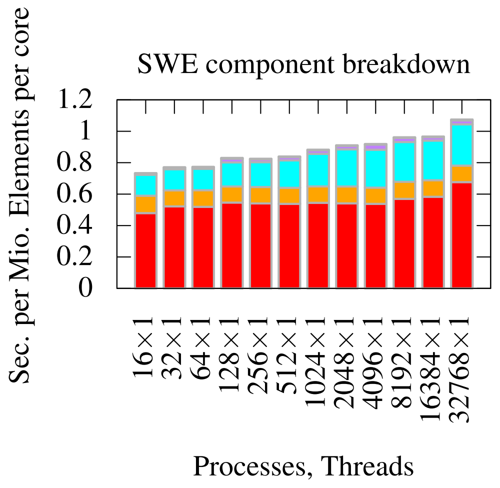
In the weak scaling test, a scenario with 1M elements per core was chosen and scaled up to 30G elements from 16 to 32,768 cores with pure MPI and socket-level hybrid parallelization. Figure 10.5 shows that time steps and conformity scale well until 8,192 cores, at which point global communication starts to affect performance. Adaptive traversals scale worse, which is clearly caused by global communication, as hybrid parallelization performs better in this case. With 8 times as many MPI ranks, the pure MPI version performs significantly more global communication. A parallel efficiency of 75% is achieved from 16 to 32,768 cores, which is a good result, considering computation is only moderately expensive and not able to hide the costs of communication, remeshing, and load balancing.



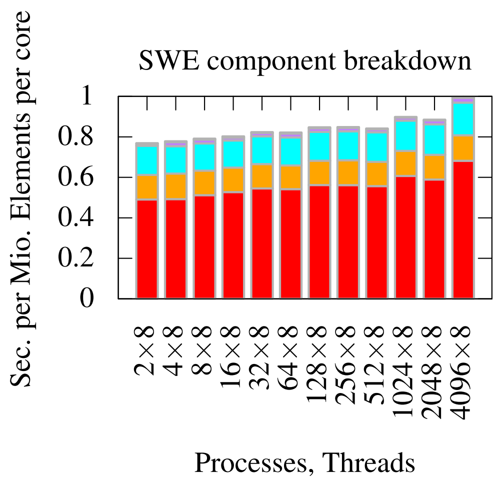
(**a**) Parallel efficiency, pure MPI



(**b**) Parallel efficiency, OpenMP+MPI



(**c**) Component analysis, pure MPI



(**d**) Component analysis, OpenMP+MPI

Figure 10.5: Weak scalability for pure MPI and hybrid socket-level parallelization. Parallel efficiency is shown for both cases. A component analysis breaks down the wall-clock time into time steps, conformity traversals, remeshing with neighbor search, and load balancing.

Strong scaling

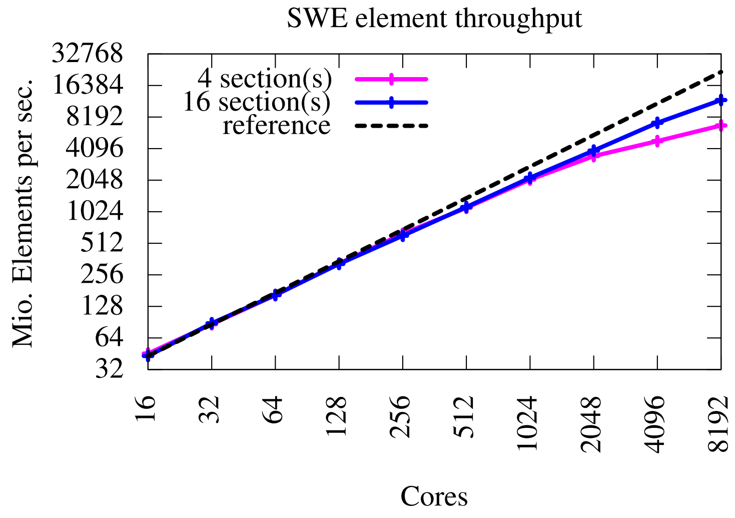
For strong scaling, we picked a scenario with 60M elements, a minimum refinement depth of 8 and a maximum refinement depth of 32. Scalability was tested from 1 node to 512 nodes of the SuperMUC thin node system. Therefore, each partition eventually contains 7,000 elements on 8,192 cores. In Figure 10.6 we see that strong scalability of pure MPI and hybrid parallelization both is fine up to 512 cores and then starts dropping. When 512 cores are reached, we obtain an efficiency of 82%. The early performance loss is mostly caused by a load imbalance due to the heterogeneous flux solver. On 2048 cores, an efficiency of 71% is obtained, and on 8,192 cores efficiency drops to 54%. Compared to the 2D porous media flow scenario, which achieves 36% efficiency on 8,192 cores as seen in Section 7.3.2, tsunami wave propagation scales worse in the beginning, but performs better at large numbers of cores. The tsunami scenario is affected more by adaptive mesh refinement and load balancing, however the increase of communication appears to have a stronger impact on performance. As Figure 10.6 shows, performance of the tsunami scenario starts to deteriorate on 2,048 cores, where load balancing and adaptive mesh refinement become increasingly expensive, especially for hybrid parallelization. On 8,192 cores each core only holds 7,000 elements, which is a very low number, and strong scalability is only 47%. Hence, the problem size per core is clearly too small at this point.

10.4 Conclusion: Scalable Remeshing and Load Balancing

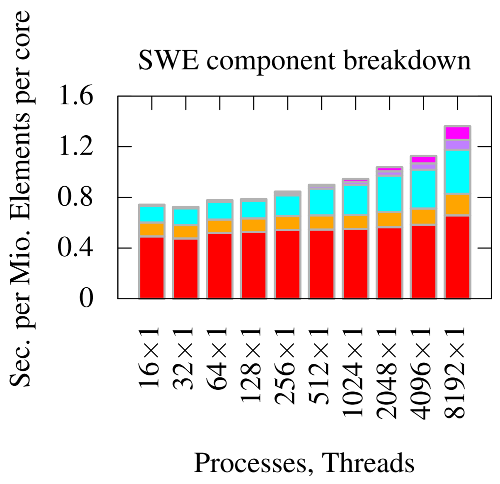
The single node analysis showed that the tsunami wave propagation scenario offers a good baseline performance when compared to a Cartesian mesh solver, but overhead from adaptive mesh refinement is significant. The flux solver kernels are compute-bound, but clearly not expensive enough to dominate performance.

Adaptive mesh refinement was observed to scale well in a weak scaling test up to 2048 cores and in a strong scaling test up to around 512 cores. The tsunami wave propagation exhibits good weak scalability with 75% efficiency from 16 to 32,768 cores. In particular, this test shows that load balancing scales well to the extreme scale if 600k elements or more are assigned to each core. We also tested advanced load balancing techniques, namely time-based cost evaluation, section splitting and optimal chains-on-chains partitioning, of which the latter two are most suitable for execution on up to a few hundred cores. Furthermore, time-based cost evaluation proved to be beneficial for this scenario in many cases.

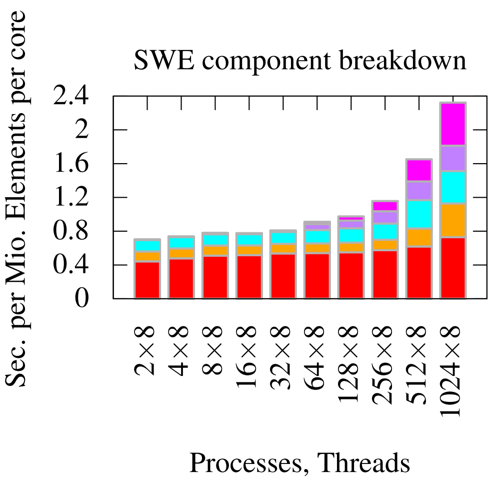
In a strong scaling setting, a parallel efficiency of 82% from 16 to 512 cores was obtained. At some point, adaptive mesh refinement and load balancing start to impair performance seriously; however, the overhead from global communication is still larger, as we saw in the comparison to the porous media flow scenario. Another issue is that hybrid parallelization did not improve strong scalability, but instead, load balancing and adaptive mesh refinement perform worse in this case. Some of our tests indicate that the reason for this behavior is a serialization of MPI messages when OpenMP threads attempt concurrent MPI communication. In a peer-to-peer-topology, this method is never faster than the fully concurrent communication of a pure MPI parallelization. Nevertheless, with its design for large scale systems, sam(oa)^2 shows very good scalability both in a weak scaling and a strong scaling setting.



(a) Strong scalability, pure MPI and OpenMP+MPI



(b) Component analysis, pure MPI



(c) Component analysis, OpenMP+MPI

Time step 🟥 Adaption 🟦 Load Balancing 🟪
Conformity 🟧 Neighbor search 🟪

Figure 10.6: Strong scaling of tsunami wave propagation on up to 512 nodes of the SuperMUC thin node system. Scalability is shown for pure MPI (blue) and socket-level hybrid parallelization (pink) using IBM MPI. A component analysis is shown for pure MPI and socket-level hybrid parallelization.

Summary

Looking back at the motivation of this thesis, we formulated three major requirements for high-performance software to cope with the current transition of hardware towards massive parallelism with increasingly restricted resources per core. These requirements are efficiency, flexibility, and scalability. We will shortly summarize the outcome of each of them and consider potential improvements afterwards.

The kernel interface of sam(oa)^2 demonstrates flexibility, as it supports finite element and finite volume discretizations, as well as adaptive mesh refinement and parallelization. Based on this interface, two test cases were implemented. First, a porous media flow scenario was realized that simulates oil recovery from underground reservoirs and second, an oceanic wave propagation scenario for tsunami simulation was realized.

The efficiency requirement is satisfied by sam(oa)^2 trough a traversal scheme that is based on the Sierpinski space-filling curve and that supports fully adaptive mesh refinement as well as cache-oblivious data access. By exploiting local data dependencies, a single grid traversal suffices for implementation of explicit time stepping schemes in the kernel interface. Temporary data is realized without a persistent memory footprint.

Scalability was shown for both test cases. For the porous media flow scenario, up to 90% parallel efficiency on 8,192 cores in a weak scaling test and 85% effiency on 512 cores in a strong scaling test were achieved. The tsunami wave propagation scenario scaled up to 32,768 cores with 75% efficiency and strong scalability of 71% on 2,048 cores was obtained. Furthermore, both scenarios demonstrate a good baseline efficiency. The porous media flow scenario is memory-bound and achieves 80% of the STREAM benchmark on a single SuperMUC thin node. Moreover, the tsunami wave propagation scenario performs well compared to a Cartesian mesh code and other adaptive mesh refinement codes.

When sam(oa)^2 is compared to other software in the field, its performance is a clear selling point. However, to become truly competitive there are still some issues left that need to be resolved. First, the implementation of scenarios in the interface is too complicated. A kernel layer that couples sam(oa)^2 with an abstract description or to a library with a fully opaque interface would be beneficial. Second, an improved support for hybrid parallelization and many-core architectures will become more important as heterogeneous hardware becomes more prevalent in high-performance computing. Finally, some of the functionality for adaptive mesh refinement should be optimized to deal better with local changes in the grid in order to decrease the overhead from grid management. However, a big step towards a generic framework for parallel, adaptive mesh refinement is taken. In its current state, sam(oa)^2 provides fast, fully adaptive mesh refinement and dynamic load balancing for the efficient execution of heterogeneous applications with billions of grid cells on thousands of cores.

A Outlook: Heterogeneous Chains on Chains Solvers

In this chapter, we present a short outlook on the extended chains-on-chains partitioning problem for heterogeneous hardware, which processes identical tasks on different cores with different execution times. Note that sam(oa)^2 does not yet support any of the extensions described here.

The problem of load balancing on heterogeneous hardware has been discussed for some time. An efficient optimal solver and various approximations for heterogeneous chains-on-chains partitioning [92] have been developed. Further work on the topic indicates that the best load balance is achieved if the system execution time model is determined at runtime from profiling data [21, 37, 43, 58, 61, 104]. While some of these approaches develop very detailed performance models, they are not scalable, assume knowledge of task costs, require training phases during the execution, or simply converge slowly. Clearly, sam(oa)^2 needs a scalable solution with fast a convergence rate, as grids are processed in parallel and refined frequently. Training phases are problematic, too, as they have to reflect the performance of the real computation to predict execution times accurately. The knowledge of tasks and heterogeneities is assumed in the beginning of this chapter, but later on, this restriction is dropped for a more flexible model. We will restrict ourselves to the discussion of the heterogeneous chains-on-chains problem though, which is the model that corresponds to the load balancing policy of sam(oa)^2.

A.1 Modeling Heterogeneous Hardware

The chains-on-chains model (3.5) as described in Chapter 3.3.1 is extended to the following setting. Let $c_i > 0$ be the cost of each task i for $i = 1, \ldots, n$. We assume that the cost is independent of the target core and is measured in some arbitrary cost unit. For $j = 1, \ldots, p$ we denote by $v_j \geq 0$ the *throughput* of core j, which we measure in cost units per second. The throughput v_j may be a static value that reflects the processor clock rate or the memory bandwidth, but it could also be evaluated and updated at runtime, allowing some degree of fault-tolerance by adjusting to turbo modes, overheating, or resource contention. In the context of invasive computing – see [121] for an overview of the topic – it may even incorporate a logical flag that indicates whether a core is enabled or disabled to handle invasion and retreat mechanisms. To model this, the value v_j needs to be allowed to be zero. However, for the sake of simplicity, we assume that $v_j > 0$ at any time and relax the condition later. Assuming that c_i and v_j are known, the time to execute task i on core j is

$$t_{i,j} = \frac{c_i}{v_j}. \tag{A.1}$$

Similarly to Section 3.3.1, the prefix sums

$$C_i := \sum_{k=1}^{i} c_k \quad \text{for } i = 0, \ldots, n \quad \text{and} \quad V_j := \sum_{k=1}^{j} v_k \quad \text{for } j = 0, \ldots, p. \tag{A.2}$$

are defined. The problem is to find a map from cores to tasks, which assigns each core j to a set of consecutive tasks $\{s(j-1)+1, \ldots, s(j)\}$ while minimizing the maximum execution time over all cores. Hence, we have to find a monotonically increasing function $s : \{0, \dots, p\} \to \{0, \dots, n\}$ with $s(0) := 0$ and $s(p) := n$ that minimizes the maximum load per core

$$t_s := \max_{j \in \{1,\dots,p\}} \sum_{i=s(j-1)+1}^{s(j)} t_{i,j} \;=\; \max_{j \in \{1,\dots,p\}} \sum_{i=s(j-1)+1}^{s(j)} \frac{c_i}{v_j} \;=\; \max_{j \in \{1,\dots,p\}} \frac{C_{s(j)} - C_{s(j-1)}}{v_j}. \tag{A.3}$$

A.1.1 Continuous Solution

As before, we will first derive a lower bound for t_s that depends on the load of each core j, more precisely

$$t_s \geq \frac{C_{s(j)} - C_{s(j-1)}}{v_j}, \quad \text{for all } j \in \{0 \dots, p\},$$

or equivalently,

$$t_s v_j \geq C_{s(j)} - C_{s(j-1)}. \tag{A.4}$$

Summing (A.4) over all cores $j = 1, \dots, p$ yields

$$t_s \sum_{j \in \{1,\dots,p\}} v_j = t_s V_p \geq \sum_{j \in \{1,\dots,p\}} C_{s(j)} - C_{s(j-1)} = C_{s(p)} - C_{s(0)} = C_n,$$

or in short

$$t_s \geq \frac{C_n}{V_p}. \tag{A.5}$$

Hence, a lower bound for the maximum load is the cost of all tasks over the sum of all throughputs. To find a continuous solution, we assume that there is a monotonically increasing map $\hat{s}$ from $\{1, \dots, p\}$ to $\{1, \dots, n\}$ with

$$t_{\hat{s}} = \frac{C_n}{V_p}. \tag{A.6}$$

Then, (A.5) and (A.6) imply that $\hat{s}$ minimizes the maximum load as defined in (A.3). Now,

$$t_{\hat{s}} V_p - C_n = \sum_{j \in \{1,\dots,p\}} \underbrace{t_{\hat{s}} v_j - \big(C_{\hat{s}(j)} - C_{\hat{s}(j-1)}\big)}_{\geq 0} = 0. \tag{A.7}$$

In (A.4) equality holds, implying

$$t_{\hat{s}} v_j = C_{\hat{s}(j)} - C_{\hat{s}(j-1)},$$

for all $j \in \{1, \dots, p\}$. Hence,

$$\frac{C_n}{V_p} v_j = C_{\hat{s}(j)} - C_{\hat{s}(j-1)}$$

for all $j \in \{1, \dots, p\}$ due to (A.6). Computing the prefix sum on left- and right-hand side returns

$$\frac{C_n}{V_p} V_j \;=\; \frac{C_n}{V_p} \sum_{k=1}^{j} v_k \;=\; \sum_{k=1}^{j} \big(C_{\hat{s}(k)} - C_{\hat{s}(k-1)}\big) \;=\; C_{\hat{s}(j)}, \tag{A.8}$$

Thus for each core $j \in \{1, \dots, p\}$, the value $\hat{s}(j)$ is defined implicitly by

$$\hat{s}(j) = i \quad \text{if } \frac{C_i}{C_n} = \frac{V_j}{V_p}. \tag{A.9}$$

Note that $\hat{s}$ is well-defined and unique if the C_i and the V_j are strictly increasing. Again, reinterpretation of the C_i as a piece-wise linear function guarantees existence of a continuous solution, which can be used for section splitting, see Section 3.2.2. In the next section, we exploit the existence of $\hat{s}$ to find a discrete approximation to the continuous version of the problem.

A.2 Explicit Treatment of Heterogeneities

While (A.9) is not as easy to invert as its simpler variant for homogeneous systems from (3.12), a continuous solution still exists and may be exploited once again to derive a discrete approximation based on rounding to integer values, similar to (3.15), which was derived for the homogeneous case. In addition to the approximation, we will also discuss an optimal solution to the problem in this section.

A.2.1 Midpoint Approximation

Figure A.1 illustrates an extension of the midpoint approximation in Chapter 3.3.1 to the case of assigning tasks to heterogeneous cores. One disadvantage of the approach is that for strongly heterogeneous systems, remote communication of notification messages will occur even if no changes to the load distribution are necessary. The method returns a map s that satisfies

$$t_s \le \frac{C_n}{V_p} + \frac{\max\limits_{i \in \{1,\dots,n\}} c_i}{\min\limits_{j \in \{1,\dots,p\}} v_j}, \tag{A.10}$$

which says that the imbalance is bounded by the cost of the most expensive task over the throughput of the slowest core. This ratio can be rather high and will cause problems on strongly heterogeneous hardware.

A.2.2 Optimal Solution

Computing an optimal solution to heterogeneous chains-on-chains partitioning is surprisingly easy, as the EXACT-BISECT algorithm in [91] requires only minor adjustments to handle heterogeneous throughput data, as shown in [92]. An extension of the optimal solver in sam(oa)^2 is straightforward.

A.3 Implicit Treatment of Heterogeneities

Instead of assuming explicit knowledge of throughputs, there is also the possibility to assume they are unknown. First, we attempt to use time-based cost evaluation as described in Section 3.2.3 to solve the problem, which will fail. Second we successfully model the heterogeneities as unknowns that are fitted by profiling the application.

A.3.1 Time Based Load Balancing

As mentioned before, hardware heterogeneities are not modeled explicitly in sam(oa)^2. However, a possibility to treat them implicitly is provided by time-based cost evaluation from Section 3.2.3. Denote by c_i the cost of section i and $v_i^{(k)}$ the throughput of the core assigned to section i in the k-th load

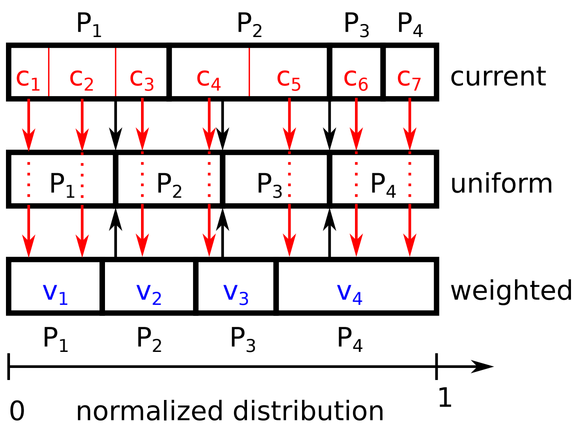


Figure A.1: Sketch of distributed load balancing on heterogeneous cores by approximate chains-on-chains partitioning. First, tasks are distributed uniformly to intermediate cores purely based on the cost distribution $(c_i)_{i=1,\dots,n}$ marked by the red arrows in the top. Next, each intermediate core passes its tasks further along to their destination cores according to the throughput distribution $(v_j)_{j=1,\dots,p}$, which is indicated by the red arrows in the bottom. Assignment of tasks to intermediate cores, as indicated by the top black arrows, is identical to the homogeneous case, where notification messages to communicate task ranges are sent. On intermediate cores, assignment functions from tasks to destination cores are constructed from notification messages sent by the destination cores. The process is represented by the black arrows in the bottom.

balancing iteration. Since the cost of a section is estimated by its actual execution time, the chains-on-chains solver will try to find a map that minimizes

$$t_s^{(k)} := \max_{j\in\{1,\dots,p\}} \sum_{i=s(j-1)+1}^{s(j)} t_i^{(k)} = \max_{j\in\{1,\dots,p\}} \sum_{i=s(j-1)+1}^{s(j)} \frac{c_i}{v_i^{(k)}}, \tag{A.11}$$

where

$$t_i^{(k)} := \frac{c_i}{v_i^{(k)}} \quad \text{for all } i \in \{1,\dots n\}.$$

Tasks that are executed on a slow core are assumed to be more expensive and distributed to other cores. This behavior suggests that the chains-on-chains solver will recognize imbalances and reduce the load of slow cores.

Pachalieva [86] investigated performance of time-based cost evaluation in sam(oa)^2 on the heterogeneous SuperMIC cluster of the SuperMUC system [119]. The cluster consists of 32 Intel Ivy Bridge nodes with two Intel Xeon Phi cards each. Compared to an Ivy Bridge node, the Xeon Phi card runs at a lower clock rate and with less memory bandwidth per core, but has superior computing capabilities, as it supports 120 shared memory threads per card and supplies vector registers that fit 8 double precision numbers. Pachalieva found that sam(oa)^2 adapts well to the heterogeneity of the system on up to a few hundred MPI ranks, which are distributed to a single Ivy Bridge node and two Xeon Phi cards in symmetric mode. A memory throughput comparison of pure Ivy Bridge, pure Xeon Phi and symmetric mode executions implies good load balancing. However, for larger numbers of nodes, suboptimal distributions and load oscillations occurred, which were not further investigated in the thesis [86].

Convergence with Atomic Sections

With some analysis, it is possible to show that the issues on multiple nodes are due to misinterpretation of hardware heterogeneity as a change of costs. Consider the following example with two identical tasks on two strongly heterogeneous cores:

$$\hat{c}_1 := 1, \hat{c}_2 := 1, \hat{v}_1 := 4, \hat{v}_2 := 1. \tag{A.12}$$

We assume that an optimal chains-on-chains solver, such as EXACT-BISECT, is applied to compute distributions. Distributing the tasks to both cores returns the maximum load

$$\max\left(\frac{\hat{c}_1}{\hat{v}_1}, \frac{\hat{c}_2}{\hat{v}_2}\right) = \max\left(\tfrac{1}{4}, 1\right) = 1.$$

Executing both tasks on Core 1 instead returns

$$\max\left(\frac{\hat{c}_1}{\hat{v}_1} + \frac{\hat{c}_2}{\hat{v}_1}, 0\right) = \max\left(\tfrac{1}{4} + \tfrac{1}{4}, 0\right) = \tfrac{1}{2} < 1.$$

As Core 1 is much faster than Core 2, the best solution is to keep both tasks on the first core. We assume this state is the initial state and define

$$t_1^{(0)} := \frac{\hat{c}_1 + \hat{c}_2}{\hat{v}_1} = \tfrac{1}{2} \quad \text{and} \quad t_2^{(0)} := 0 \tag{A.13}$$

As the chains-on-chains solver is not aware of the heterogeneity, it assumes that both cores have the same speed. Therefore, the second task is moved to the second core and the optimal distribution will not be preserved, because

$$\max\left(\frac{\hat{c}_1}{\hat{v}_1}, \frac{\hat{c}_2}{\hat{v}_1}\right) = \tfrac{1}{4} < \max\left(t_1^{(0)}, t_2^{(0)}\right) = \tfrac{1}{2}. \tag{A.14}$$

However, due to the low throughput of the second core, execution time will be longer instead of shorter. We have

$$t_1^{(1)} = \frac{\hat{c}_1}{\hat{v}_1} = \tfrac{1}{4} \quad \text{and} \quad t_2^{(1)} = \frac{\hat{c}_2}{\hat{v}_2} = 1, \tag{A.15}$$

which satisfies

$$\max(t_1^{(1)}, t_2^{(1)}) = 1 > \max\left(t_1^{(0)}, t_2^{(0)}\right) = \tfrac{1}{2}. \tag{A.16}$$

The solver incorrectly assumes that the cost of the second task has changed and executing both tasks on the first core would add up their load; that is

$$\max(\frac{\hat{c}_1}{\hat{v}_1} + \frac{\hat{c}_2}{\hat{v}_2}, 0) = \tfrac{5}{4} > \max(t_1^{(1)}, t_2^{(1)}) = 1. \tag{A.17}$$

At this point, the algorithm terminates. While the solver recognizes an imbalance in the final state, it is satisfied under the wrong impression that the imbalance cannot be reduced further. So the tasks are kept on different cores and a suboptimal solution is returned.

Convergence with Section Splitting on Two Cores

Consider the problem from the previous paragraph once more. The chains-on-chains solver terminated when it was not able to assign the load in a more fine-grained way to the two tasks, even though it recognized the imbalance. Hence, the next canonical step is to use section splitting, as described in Section 3.2.2, which allows subdivision of tasks into arbitrary size. On two cores a simple update rule for the load of both cores is obtained, namely

$$\begin{aligned} t_1^{(k+1)} &= t_1^{(k)} + \min\left(\frac{t_2^{(k)} - t_1^{(k)}}{2}, 0\right) + \max\left(\frac{t_2^{(k)} - t_1^{(k)}}{2}, 0\right)\frac{v_2}{v_1}, \\ t_2^{(k+1)} &= t_2^{(k)} + \min\left(\frac{t_1^{(k)} - t_2^{(k)}}{2}, 0\right) + \max\left(\frac{t_1^{(k)} - t_2^{(k)}}{2}, 0\right)\frac{v_1}{v_2}. \end{aligned} \tag{A.18}$$

The idea is that one of both cores gives away half of the load by which it exceeds the load of the other core. The other core gains the migrated amount of load, but executes it at its own speed, therefore the execution time is adjusted by the factor $\frac{v_i}{v_j}$. At iteration $k+1$ the load difference satisfies

$$t_2^{(k+1)} - t_1^{(k+1)} = \begin{cases} \frac{t_2^{(k)} - t_1^{(k)}}{2}\left(1 - \frac{v_1}{v_2}\right) & \text{if } t_2^{(k)} < t_1^{(k)} \\ \frac{t_2^{(k)} - t_1^{(k)}}{2}\left(1 - \frac{v_2}{v_1}\right) & \text{if } t_2^{(k)} \geq t_1^{(k)} \end{cases}. \tag{A.19}$$

If $v_1 \geq v_2 > 0$, then

$$1 > 1 - \frac{v_2}{v_1} \geq 0 \quad \text{and} \quad 1 - \frac{v_1}{v_2} \leq 0. \tag{A.20}$$

If $v_2 \geq v_1 > 0$, then

$$1 > 1 - \frac{v_1}{v_2} \geq 0 \quad \text{and} \quad 1 - \frac{v_2}{v_1} \leq 0. \tag{A.21}$$

Without loss of generality we assume that $v_1 \geq v_2$, as otherwise the indexes can be switched. If $t_2^{(0)} < t_1^{(0)}$, then

$$t_2^{(1)} - t_1^{(1)} = \frac{t_2^{(0)} - t_1^{(0)}}{2}\left(1 - \frac{v_1}{v_2}\right) \geq 0. \tag{A.22}$$

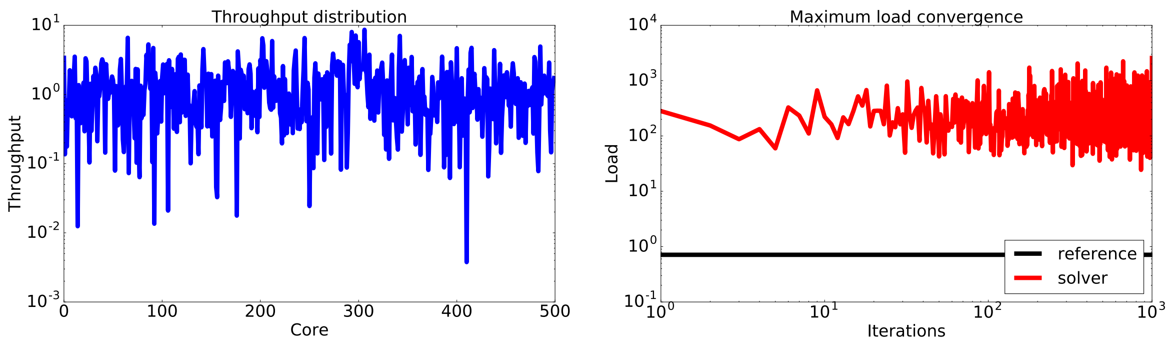
Furthermore, if there is a $k \in \mathbb{N} \cup \{0\}$ with $t_2^{(k)} \geq t_1^{(k)}$, then

$$t_2^{(k+1)} - t_1^{(k+1)} = \frac{t_2^{(k)} - t_1^{(k)}}{2}\left(1 - \frac{v_2}{v_1}\right) \geq 0. \tag{A.23}$$

Thus $t_2^{(k)} \geq t_1^{(k)}$ for each $k \in \mathbb{N}$ and

$$t_2^{(k+1)} - t_1^{(k+1)} = \left(t_2^{(k)} - t_1^{(k)}\right)\frac{1 - \frac{v_2}{v_1}}{2} = \left(t_2^{(1)} - t_1^{(1)}\right)\left(\frac{1}{2} - \frac{v_2}{2v_1}\right)^k. \tag{A.24}$$

As $0 \leq \frac{1}{2} - \frac{v_2}{2v_1} < \frac{1}{2}$, this term converges to 0 for $k \to \infty$. Therefore, the optimal load balance is found at some point.



(a) Random throughout distribution **(b)** Load imbalance over time

Figure A.2: Load balancing with time-based cost evaluation. Tasks with random costs are distributed to 512 cores with heterogeneous throughputs. The algorithm does not appear to converge.

Convergence with Section Splitting on Multiple Cores

Here, we will try to extend the result to multiple cores now where the update rule is more complicated. Fix some $p > 2$. Then the update rule is for each $i \in \{1, \ldots, p\}$:

$$\begin{aligned} t_i^{(k+1)} &= \sum_{j=1}^{p} l\left(\left[T_{j-1}^{(k)}, T_j^{(k)}\right] \cap \left[\frac{i-1}{p}T_p^{(k)}, \frac{i}{p}T_p^{(k)}\right]\right)\frac{v_j}{v_i} \\ &= \sum_{j=1}^{p} \max\left(0, \min\left(t_j^{(k)}, T_j^{(k)} - \frac{i-1}{p}T_p^{(k)}, \frac{i}{p}T_p^{(k)} - T_{j-1}^{(k)}, \frac{1}{p}T_p^{(k)}\right)\right)\frac{v_j}{v_i}, \end{aligned} \tag{A.25}$$

where $l(I)$ is defined as the length of the 1D interval I and

$$T_i^{(k)} := \sum_{j=1}^{i} t_j^{(k)} \tag{A.26}$$

is the prefix sum over the current load of each core i. In this case, it is uncertain if the algorithm converges. Indeed, the sandbox tests in Figure A.2 indicate that the algorithm will not converge if the throughput of the cores varies strongly.

In fact, whether or not the algorithm converges is not the main issue as the algorithm has a fundamental problem, which is already visible in the case $p = 2$. Due to the cost multiplier $\frac{v_j}{v_i}$ of inbound tasks, the algorithm always overloads slow cores. This behavior leads to strongly imbalanced intermediate states and divergence or slow convergence in general. In simulations on dynamic grids, the imbalanced state is never resolved.

Some improvements are possible. First, cores can be sorted by their throughput to align the direction of task migration. However, if there are large jumps in the sorted throughputs, oscillations may still occur. Second, reducing the number of exchanged subtasks by an under-relaxation mitigates oscillations, but it slows down convergence further, resulting in even longer periods where the grid is in an imbalanced state. Eventually, forcing convergence might be possible at the price of slow convergence speed with 10^2 to 10^4 iterations. This is far from practical however, as a method that needs more than 10^2 iterations for convergence is not suitable to resolve dynamic imbalances.

A.3.2 A Black Box Regression Model for Cost and Throughput Estimates

The previous approach showed that ignoring the heterogeneity of the system can lead to bad performance and slow convergence speed of the load balancing algorithm. Hence, it is better to preserve the heterogeneity data instead of discarding it. The idea is to model the hardware heterogeneities as in (A.1), but assume both the cost vector

$$\mathbf{c} := (c_i)_{i=1,\dots,n} \tag{A.27}$$

and the throughput vector

$$\mathbf{v} := (v_j)_{j=1,\dots,p} \tag{A.28}$$

are degrees of freedom this time, subject to $c_i > 0$ and $v_j > 0$, which is similar to [43]. However, in contrast to Galindo et al. we still restrict ourselves to the chains-on-chains problem.

Similar to time-based cost evaluation, the execution time of each task is measured at runtime. We assume that there are m independent measurements, where the k^{th} measurement t_k is executed for a task $i(k) \in \{1,\dots,n\}$ on a core $j(k) \in \{1,\dots,p\}$ and the resulting time is stored in a vector $\mathbf{t} := (t_k)_{k=1,\dots,m}$. To determine the vectors $\mathbf{c}$ and $\mathbf{v}$, we set up a system of equations now, namely

$$\frac{c_{i(k)}}{v_{j(k)}} = t_k \quad \text{for every } k \in \{1,\dots,m\}. \tag{A.29}$$

Note that the solution of (A.29) cannot be unique, as multiplying cost and throughput vectors with any non-zero constant will not change the left-hand side. To linearize the system, we transform it into logarithmic space and obtain

$$\underbrace{\log(c_{i(k)})}_{\tilde{c}_{i(k)}} - \underbrace{\log(v_{j(k)})}_{\tilde{v}_{j(k)}} = \underbrace{\log(t_k)}_{\tilde{t}_k} \quad \text{for every } k \in \{1,\dots,m\}. \tag{A.30}$$

Now, the problem reduces to solving a linear system of equations, which can be written as

$$\mathbf{M}_c\tilde{\mathbf{c}} - \mathbf{M}_v\tilde{\mathbf{v}} = \tilde{\mathbf{t}}, \tag{A.31}$$

where $\mathbf{M}_c \in \{0,1\}^{m\times n}$ is 1 in each entry $(k, i(k))$ and 0 everywhere else. Similarly, $\mathbf{M}_v \in \{0,1\}^{m\times p}$ is 1 in each entry $(k, j(k))$ and 0 everywhere else. As $k > n + p$ in most cases, the system is usually overdetermined. Thus, a least squares approach is applied, where a quadratic function $f : \mathbb{R}^{n+p} \to \mathbb{R}$ with

$$f(\mathbf{c},\mathbf{v}) := \left(\mathbf{M}_c\tilde{\mathbf{c}} - \mathbf{M}_v\tilde{\mathbf{v}} - \tilde{\mathbf{t}}\right)^2. \tag{A.32}$$

is minimized with respect to $\mathbf{c}$ and $\mathbf{v}$ to approximate the solution to (A.31). The minimum of f is found by solving the least squares system

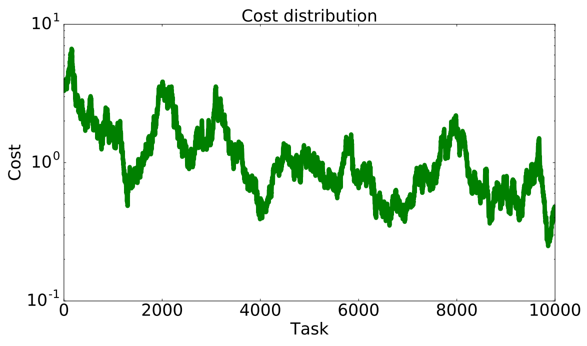
$$\begin{aligned} -\mathbf{M}_v^T\left(\mathbf{M}_c\tilde{\mathbf{c}} - \mathbf{M}_v\tilde{\mathbf{v}} - \tilde{\mathbf{t}}\right) &= \mathbf{0}, \\ \mathbf{M}_c^T\left(\mathbf{M}_c\tilde{\mathbf{c}} - \mathbf{M}_v\tilde{\mathbf{v}} - \tilde{\mathbf{t}}\right) &= \mathbf{0}. \end{aligned} \tag{A.33}$$

To simplify the description of the system, we define $\mathbf{D}_v := \mathbf{M}_v^T\mathbf{M}_v$, $\mathbf{D}_c := \mathbf{M}_c^T\mathbf{M}_c$, and $\mathbf{S} := \mathbf{M}_c^T\mathbf{M}_v$. The matrices $\mathbf{D}_c$ and $\mathbf{D}_v$ are diagonal, and thus easy to invert.

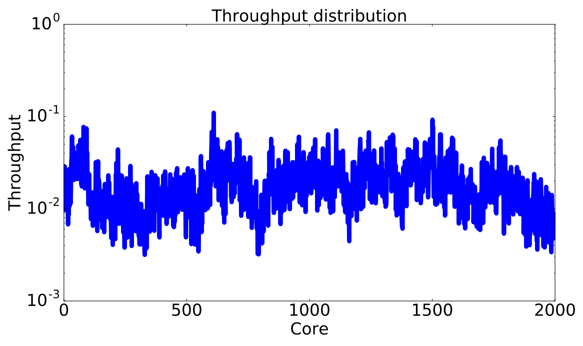
$$\begin{aligned} \mathbf{D}_v\tilde{\mathbf{v}} - \mathbf{S}^T\tilde{\mathbf{c}} + \mathbf{M}_v^T\tilde{\mathbf{t}} &= \mathbf{0}, \\ \mathbf{D}_c\tilde{\mathbf{c}} - \mathbf{S}\tilde{\mathbf{v}} - \mathbf{M}_c^T\tilde{\mathbf{t}} &= \mathbf{0}. \end{aligned} \tag{A.34}$$

Either $\tilde{\mathbf{v}}$ or $\tilde{\mathbf{c}}$ can now be eliminated from the system of equations. Assuming $n \geq p$, we eliminate $\tilde{\mathbf{c}}$ and obtain

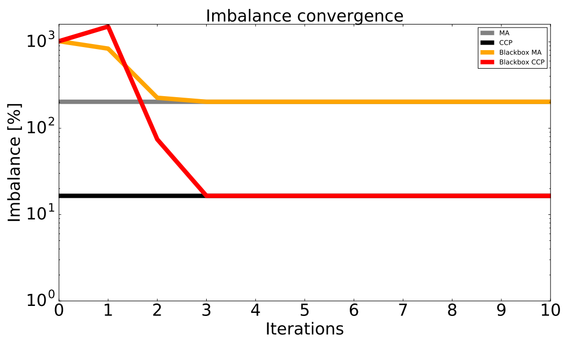
$$\begin{aligned} \underbrace{\left(\mathbf{D}_v - \mathbf{S}^T\mathbf{D}_c^{-1}\mathbf{S}\right)}_{\mathbf{A}}\tilde{\mathbf{v}} &= \underbrace{-\left(\mathbf{M}_v^T - \mathbf{S}^T\mathbf{D}_c^{-1}\mathbf{M}_c^T\right)\tilde{\mathbf{t}}}_{\mathbf{b}}, \\ \mathbf{D}_c\tilde{\mathbf{c}} &= \mathbf{S}\tilde{\mathbf{v}} + \mathbf{M}_c^T\tilde{\mathbf{t}}, \end{aligned} \tag{A.35}$$



(a) Random cost distribution



(b) Random task distribution



(c) Convergence plot

Figure A.3: Imbalance over the number of iterations for black-box modeling and explicit modeling of n = 10,000 heterogeneous tasks on p = 2,000 heterogeneous cores. Cost and throughput distributions are randomly generated, ranging over at least two orders of magnitude. The result are from explicit solvers and regression solvers using heterogeneous variants of optimal chains-on-chains partitioning and midpoint approximation to balance load estimates.

which is solved successively. One can show that the matrix $\mathbf{A}$ of the upper system is symmetric and positive semi-definite. It must be singular, as we already observed that (A.31) does not have a unique solution. Indeed, the all-ones vector $(1, 1, \ldots, 1)^T$ is always an eigenvector of $\mathbf{A}$ with the eigenvalue 0. This issue is fixed by adding the equation $\tilde{v}_1 = 0$ to the system. Only the top left entry in the matrix $\mathbf{D}_v$ must be modified and incremented by 1. The resulting matrix is symmetric and positive definite if enough measurements are available. The main advantage of this formulation is that it boils down to the solution of a linear system, for which scalable methods exist. For instance, a parallel Conjugate Gradients method solves the system quickly.

This method is slightly similar to the ideas in [104]. However, in our case, a least squares solver is used to fit the task and core heterogeneities, in contrast to determining an execution time model from known task heterogeneities for each core independently. In our case, finding a good load distribution simply means solving the explicit heterogeneous chains-on-chains problem. Additionally, due to the simple model, no training phase is required, as the unknowns are fitted on-the-fly using time-based cost evaluation.

Results on artificial cost and throughput distributions are illustrated in Figure A.3. The regression solver converges quickly, as only three time steps are required to find accurate distributions and to obtain an imbalance which is as good as for explicit methods. However, the algorithm is not cheap and does not scale per se, as it solves a linear system of equations in each iteration that grows with the number of cores p. To simplify the solution of the system and to keep memory requirements low, measurements can be stored in a distributed cache with a replacement strategy based on hash values generated from the task and core indexes i and j. Additionally, there is no need to set up $\mathbf{M}_c$, $\mathbf{M}_v$, and $\tilde{\mathbf{t}}$ explicitly. Instead, the matrices $\mathbf{D}_c$, $\mathbf{D}_v$, $\mathbf{S}$, and the right-hand sides are constructed incrementally. Moreover, $\mathbf{S}$ remains sparse as the number of non-zero entries in $\mathbf{S}$ is bounded by the cache size.

A.4 Conclusion: Explicit and Implicit Treatment of Heterogeneous Hardware

To conclude, implicit treatment of heterogeneities by time-based cost evaluation clearly fails, as it does not appear to converge in acceptable time. Modeling the heterogeneities is the method of choice. On the one hand, additional knowledge by means of a throughput model for each core may be applied. On the other hand, throughput can be considered to be unknown, in which case a linear system of equations must be solved in each time step to evaluate the heterogeneities correctly.

We do not consider this problem further, as it implies knowledge of heterogeneities in the hardware that we do not assume to have. The test systems we employed are homogeneous, where any variations in hardware performance are volatile and unpredictable.

I love deadlines. I love the whooshing noise they make as they go by.
Douglas Adams (1952 – 2001)

Bibliography

[1] J. E. Aarnes, V. Kippe, and K.-A. Lie. Mixed multiscale finite elements and streamline methods for reservoir simulation of large geomodels. *Advances in Water Resources*, 28(3):257–271, 2005.

[2] M. Alnæs, J. Blechta, J. Hake, A. Johansson, B. Kehlet, A. Logg, C. Richardson, J. Ring, M. Rognes, and G. Wells. The fenics project version 1.5. *Archive of Numerical Software*, 3(100), 2015.

[3] M. S. Alnæs, A. Logg, K. B. Ølgaard, M. E. Rognes, and G. N. Wells. Unified form language: A domain-specific language for weak formulations of partial differential equations. *ACM Trans. Math. Softw.*, 40(2):9:1–9:37, March 2014.

[4] U. G. Araktingi and F. M. Orr Jr. Viscous fingering in heterogeneous porous media. *SPE Adv. Tech.*, 1(1):71–80, April 1993.

[5] ASAGI – a parallel server for adaptive geoinformation. `http://www.github.com/tum-i5/asagi`. Retrieved in July, 2016.

[6] ASCETE – advanced simulation of coupled earthquake and tsunami events. `www.ascete.de/`. Retrieved in July, 2016.

[7] K. Aziz. *Petroleum Reservoir Simulation*. Applied Science Publishers, 1979.

[8] M. Bader. *Space-Filling Curves – An Introduction with Applications in Scientific Computing*, volume 9 of *Texts in Computational Science and Engineering*. Springer-Verlag, 2013.

[9] M. Bader, C. Böck, J. Schwaiger, and C. Vigh. Dynamically adaptive simulations with minimal memory requirement—solving the shallow water equations using Sierpinski curves. *SIAM J. Scientific Computing*, 32(1):212–228, 2010.

[10] M. Bader and A. Breuer. Teaching parallel programming models on a shallow-water code. In *ISPDC 2012 - 11th International Symposium on Parallel and Distributed Computing*, pages 301–308. IEEE Computer Society, November 2012.

[11] M. Bader, A. Breuer, W. Hölzl, and S. Rettenberger. Vectorization of an augmented riemann solver for the shallow water equations. In Waleed W. Smari and Vesna Zeljkovic, editors, *Proceedings of the 2014 International Conference on High Performance Computing and Simulation (HPCS 2014)*, pages 193–201. IEEE, August 2014.

[12] M. Bader, K. Rahnema, and C. Vigh. Memory-efficient Sierpinski-order traversals on dynamically adaptive, recursively structured triangular grids. In K. Jónasson, editor, *Para 2010: State of the Art in Scientific and Parallel Computing*, volume 7134 of *Lecture Notes in Computer Science*, pages 302–312. Springer, 2011.

[13] M. Bader, S. Schraufstetter, C. A. Vigh, and J. Behrens. Memory efficient adaptive mesh generation and implementation of multigrid algorithms using Sierpinski curves. *International Journal of Computational Science and Engineering*, 4:12–21, 2008.

[14] M. Bader and C. Zenger. A cache oblivious algorithm for matrix multiplication based on peano's space filling curve. In R. Wyrzykowski, J. Dongarra, N. Meyer, and J. Waśniewski, editors, *Parallel Processing and Applied Mathematics, 6th International Conference, PPAM 2005*, volume 3911 of *Lecture Notes in Computer Science*, pages 1042–1049, March 2006.

[15] D. Bale, R. J. LeVeque, S. Mitran, and J. A. Rossmanith. A wave-propagation method for conservation laws and balance laws with spatially varying flux functions. *SIAM J. Sci. Comput.*, 24:955–978, 2002.

[16] W. Bangerth, C. Burstedde, and M. Heister, T.and Kronbichler. Algorithms and data structures for massively parallel generic adaptive finite element codes. *ACM Trans. Math. Softw.*, 38(2):14:1–14:28, January 2012.

[17] W. Bangerth, T. Heister, L. Heltai, G. Kanschat, M. Kronbichler, M. Maier, B. Turcksin, and T. D. Young. The `deal.ii` library, version 8.1. *arXiv*, 2013.

[18] P. Bastian, M. Blatt, A. Dedner, C. Engwer, R. Klöfkorn, M. Ohlberger, and O. Sander. A generic grid interface for parallel and adaptive scientific computing. Part I: Abstract framework. *Computing*, 82(2–3):103–119, 2008.

[19] P. Bastian, F. Heimann, and S. Marnach. Generic implementation of finite element methods in the distributed and unified numerics environment (dune). *Kybernetika*, 46(2):294–315, 2010.

[20] J. Behrens and J. Zimmermann. Parallelizing an unstructured grid generator with a space-filling curve approach. In A. Bode, T. Ludwig, W. Karl, and R. Wismüller, editors, *Euro-Par 2000 Parallel Processing*, volume 1900 of *Lecture Notes in Computer Science*, pages 815–823. Springer Berlin Heidelberg, 2000.

[21] M. E. Belviranli, L. N. Bhuyan, and R. Gupta. A dynamic self-scheduling scheme for heterogeneous multiprocessor architectures. *ACM Trans. Archit. Code Optim.*, 9(4):57:1–57:20, January 2013.

[22] E. G. Boman, U. V. Catalyurek, C. Chevalier, and K. D. Devine. The Zoltan and Isorropia parallel toolkits for combinatorial scientific computing: Partitioning, ordering, and coloring. *Scientific Programming*, 20(2):129–150, 2012.

[23] C. Braun. *Ein Upscaling-Verfahren fr Mehrphasenströmungen in Porösen Medien*. Dissertation, Universität Stuttgart, Stuttgart, 2000.

[24] Y. Brenier and J. Jaffré. Upstream differencing for multiphase flow in reservoir simulation. *SIAM J. Numer. Anal.*, 28(3):685–696, May 1991.

[25] R. H. Brooks and A. T. Corey. *Hydraulic Properties of Porous Media*. Colorado State University Hydrology Papers. Colorado State University, 1964.

[26] S. E. Buckley and M. C. Leverett. Mechanisms of fluid displacement in sands. *Transactions of the American Institute of Mining and Metallurgical Engineers*, 146:107–116, 1942.

[27] C. Burstedde. Morton curve segments produce no more than two distinct face-connected subdomains. *CoRR*, 2015.

[28] C. Burstedde, D. Calhoun, K. T. Mandli, and A. R. Terrel. ForestClaw: hybrid forest-of-octrees AMR for hyperbolic conservation laws. In *Parallel Computing – Accelerating Computational Science and Engineering (CSE)*, volume 25 of *Advances in Parallel Computing*, pages 253–262. IOS Press, 2013.

[29] C. Burstedde, O. Ghattas, M. Gurnis, T. Isaac, G. Stadler, T. Warburton, and L. Wilcox. Extreme-scale amr. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–12, Washington, DC, USA, 2010. IEEE Computer Society.

[30] C. Burstedde, L. C. Wilcox, and O. Ghattas. `p4est`: Scalable algorithms for parallel adaptive mesh refinement on forests of octrees. *SIAM Journal on Scientific Computing*, 33(3):1103–1133, 2011.

[31] U.V. Catalyurek, E.G. Boman, K.D. Devine, D. Bozdag, R.T. Heaphy, and L.A. Riesen. Hypergraph-based dynamic load balancing for adaptive scientific computations. In *Proc. of 21st International Parallel and Distributed Processing Symposium (IPDPS'07)*. IEEE, 2007. Best Algorithms Paper Award.

[32] M. A. Christie and M. J. Blunt. Tenth spe comparative solution project: a comparison of upscaling techniques. *SPE Reservoir Evaluation & Engineering*, 4(04):308–317, 2001.

[33] C. C. Chueh, M. Secanell, W. Bangerth, and N. Djilali. Multi-level adaptive simulation of transient two-phase flow in heterogeneous porous media. *Computers & Fluids*, 39(9):1585 – 1596, 2010.

[34] A. T. Corey. The interrelation between gas and oil relative permeabilities. *Producers Monthly*, 19:38–41, November 1954.

[35] R. Courant, K. Friedrichs, and H. Lewy. über die partiellen differenzengleichungen der mathematischen physik. *Mathematische Annalen*, 100:32–74, 1928.

[36] R. Courant, K. Friedrichs, and H. Lewy. On the partial difference equations of mathematical physics. *IBM J. Res. Dev.*, 11(2):215–234, March 1967.

[37] R. Y. d. Camargo. A load distribution algorithm based on profiling for heterogeneous gpu clusters. In *Applications for Multi-Core Architectures (WAMCA), 2012 Third Workshop on*, pages 1–6, Oct 2012.

[38] R. Donat, M. C. Mart, A. Martnez-Gavara, and P. Mulet. Well-balanced adaptive mesh refinement for shallow water flows. *Journal of Computational Physics*, 257, Part A:937 – 953, 2014.

[39] DUNE – distributed and unified numerics environment. `https://www.dune-project.org`. Retrieved in July, 2016.

[40] DUNE PDELab. `https://www.dune-project.org/pdelab`. Retrieved in July, 2016.

[41] FEniCS project. `https://fenicsproject.org`. Retrieved in July, 2016.

[42] Firedrake – an automated system for the portable solution of partial differential equations using the finite element method (fem). `http://www.firedrakeproject.org/`. Retrieved in July, 2016.

[43] I. Galindo, F. Almeida, and J. M. Badía-Contelles. Dynamic load balancing on dedicated heterogeneous systems. In *Proceedings of the 15th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 64–74, Berlin, Heidelberg, 2008. Springer-Verlag.

[44] T. Gallouet, J.-M. Herard, and N. Seguin. Some approximate godunov schemes to compute shallow-water equatons with topography. *Comput. Flui*, 32:479–513, 2003.

[45] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1979.

[46] M. R. Garey, D. S. Johnson, and L. Stockmeyer. Some simplified np-complete graph problems. *Theoretical Computer Science*, 1(3):237 – 267, 1976.

[47] GEBCO_2014 Grid. `http://www.gebco.net/data_and_products/gridded_bathymetry_data/gebco_30_second_grid`. Retrieved in July, 2016.

[48] D. L. George. *Finite Volume Methods and Adaptative Refinement for Tsunami Propagation and Inundation*. PhD thesis, PhD Thesis, University of Washington, USA, 2006.

[49] D. L. George. Augmented Riemann solvers for the shallow water equations over variable topography with steady states and inundation. *Journal of Computational Physics*, 227(6):3089–3113, 2008.

[50] E. Godlewski and P. A. Raviart. *Numerical Approximation of Hyperbolic Systems of Conservation Laws*. Number Nr. 118 in Applied Mathematical Sciences. Springer, 1996.

[51] R. L. Graham. Bounds on multiprocessing timing anomalies. *SIAM Journal on Applied Mathematics*, 17(2):416–429, 1969.

[52] B. N. Gunney. Scalable mesh management for patch-based amr. Technical report, UNT Digital Library, Livermore, California., 2012. `http://digital.library.unt.edu/ark:/67531/metadc845234/m1/1/`.

[53] B. N. Gunney. Dynamic adaptively refined mesh simulations on 1m+ cores. `http://sc15.supercomputing.org/sites/all/themes/SC15images/tech_poster/tech_poster_pages/post115.html`, 2015.

[54] R. Helmig, J. Niessner, B. Flemisch, M. Wolff, and J. Fritz. Efficient modeling of flow and transport in porous media using multiphysics and multiscale approaches. In W. Freeden, M. Z. Nashed, and T. Sonar, editors, *Handbook of Geomathematics*, pages 417–457. Springer Berlin Heidelberg, 2010.

[55] D. Hilbert. *Über die stetige Abbildung einer Linie auf ein Flächenstück*, pages 1–2. Springer Berlin Heidelberg, Berlin, Heidelberg, 1891.

[56] USGS model for mar 11, 2011 earthquake, honshu , japan. `http://earthquake.usgs.gov/earthquakes/eventpage/usp000hvnu`. Retrieved in July, 2016.

[57] J. Horrillo, Z. Kowalik, and Y. Shigihara. Wave dispersion study in the indian ocean-tsunami of december 26, 2004. *Marine Geodesy*, 29(3):149–166, 2006.

[58] Susan Flynn Hummel, Jeanette Schmidt, R. N. Uma, and Joel Wein. Load-sharing in heterogeneous systems via weighted factoring. In *Proceedings of the Eighth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '96, pages 318–328, New York, NY, USA, 1996. ACM.

[59] T. Isaac, C. Burstedde, and O. Ghattas. Low-cost parallel algorithms for 2:1 octree balance. In *Parallel Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, pages 426–437, May 2012.

[60] T. Isaac, C. Burstedde, L. C. Wilcox, and O. Ghattas. Recursive algorithms for distributed forests of octrees. *SIAM Journal on Scientific Computing*, 37(5):C497–C531, 2015.

[61] R. Kaleem, R. Barik, T. Shpeisman, B. T. Lewis, C. Hu, and K. Pingali. Adaptive heterogeneous scheduling for integrated gpus. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, PACT '14, pages 151–162, New York, NY, USA, 2014. ACM.

[62] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing*, 20(1):359–392, 1998.

[63] D. E. Keyes. Exaflop/s: The why and the how. *Comptes Rendus Mecanique*, 339:70–77, February 2011.

[64] D. Klimenko. Evaluation of preconditioners for element-oriented conjugate gradients solvers. Studienarbeit/sep/idp, Institut für Informatik, Technische Universität München, April 2015.

[65] D. Kröner and M. Ohlberger. A posteriori error estimates for upwind finite volume schemes for nonlinear conservation laws in multi dimensions. *Mathematics of Computation*, 69:25 – 39, 2000.

[66] I. Lashuk, A. Chandramowlishwaran, H. Langston, T.-A. Nguyen, R. Sampath, A. Shringarpure, R. Vuduc, L. Ying, D. Zorin, and G. Biros. A massively parallel adaptive fast multipole method on heterogeneous architectures. *Commun. ACM*, 55(5):101–109, May 2012.

[67] R. J. LeVeque. *Finite Volume Methods for Hyperbolic Problems*. Cambridge Texts in Applied Mathematics. Cambridge University Press, 2002.

[68] R. J. LeVeque, D. L. George, and M. J. Berger. Tsunami modelling with adaptively refined finite volume methods. *Acta Numerica*, 20:211–289, 5 2011.

[69] M. Lieber and W. E. Nagel. Scalable high-quality 1d partitioning. In *High Performance Computing Simulation (HPCS), 2014 International Conference on*, pages 112–119, July 2014.

[70] A. Logg, K.-A. Mardal, and G. N. Wells. *Automated Solution of Differential Equations by the Finite Element Method*, volume 84 of *Lecture Notes in Computational Science and Engineering*. Springer, 2012.

[71] A. Logg and G. N. Wells. Dolfin: Automated finite element computing. *ACM Trans. Math. Softw.*, 37(2):20:1–20:28, April 2010.

[72] Andre Malcher. Performance optimization of riemann solvers for the shallowwater equations. Master's thesis, Department of Informatics, Technical University of Munich, May 2016.

[73] J. D. McCalpin. Memory bandwidth and machine balance in current high performance computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pages 19–25, December 1995.

[74] D. B. McWhorter and D. K. Sunada. Exact integral solutions for two-phase flow. *Water Resources Research*, 26(3):399–413, 1990.

[75] O. Meister and M. Bader. 2d adaptivity for 3d problems: Parallel spe10 reservoir simulation on dynamically adaptive prism grids. *J. Comput. Sci.*, 9(0):101–106, May 2015.

[76] O. Meister, K. Rahnema, and M. Bader. A software concept for cache-efficient simulation on dynamically adaptive structured triangular grids. In *Applications, Tools and Techniques on the Road to Exascale Computing*, volume 22 of *Advances in Parallel Computing*, pages 251–260, Gent, 2012. ParCo 2012, IOS Press.

[77] O. Meister, K. Rahnema, and M. Bader. Parallel, memory efficient adaptive mesh refinement on structured triangular meshes with billions of grid cells. *Transactions on Mathematical Software*, 43(3):19:1–19:27, September 2016.

[78] MeTis – unstructured graph partitioning and sparse matrix ordering system, version 4.0. `http://www.cs.umn.edu/~metis`. Retrieved in July, 2016.

[79] G. Meurant. Multitasking the conjugate gradient method on the cray x-mp/48. *Parallel Computing*, 5(3):267–280, 1987.

[80] S. Miguet and J.-M. Pierson. *Heuristics for 1D rectilinear partitioning as a low cost and high quality answer to dynamic load balancing*, pages 550–564. Springer Berlin Heidelberg, Berlin, Heidelberg, 1997.

[81] W. F. Mitchell. Adaptive refinement for arbitrary finite-element spaces with hierarchical bases. *Journal of Computational and Applied Mathematics*, 36:65–78, 1991.

[82] W. F. Mitchell. A refinement-tree based partitioning method for dynamic load balancing with adaptively refined grids. *Journal of Parallel and Distributed Computing*, 67(4):417–429, 2007.

[83] G. Morton. A computer oriented geodetic data base and a new technique in file sequencing. Technical report, International Business Machines Co. Ltd., Ottawa, Ontario, Canada, 1966.

[84] National Centers for Environment Information – deep-ocean assessment and reporting of tsunami buoys, march 11, 2011. `http://www.ngdc.noaa.gov/hazard/dart/2011honshu_dart.html`. Retrieved in July, 2016.

[85] Y. Okada. Surface deformation due to shear and tensile faults in a half-space. *BSSA*, 75(4):1135–1154, 1985.

[86] A. Pachalieva. Dynamically adaptive 2.5d porous media flow simulation on xeon phi architectures. Master's thesis, Institut für Informatik, Technische Universität München, March 2016.

[87] R. Pajarola. Large scale terrain visualization using the restricted quadtree triangulation. In *Proceedings of the Conference on Visualization '98*, VIS '98, pages 19–26, Los Alamitos, CA, USA, 1998. IEEE Computer Society Press.

[88] D. W. Peaceman. Interpretation of well-block pressures in numerical reservoir simulation with nonsquare grid blocks and anisotropic permeability. *SPE J.*, 23(3):531–543, June 1983.

[89] Peano – a framework for pde solvers on spacetree grids. `http://www5.in.tum.de/peano`. Retrieved in July, 2016.

[90] G. Peano. Sur une courbe, qui remplit toute une aire plane. *Mathematische Annalen*, 36(1):157–160, 1890.

[91] A. Pınar and C. Aykanat. Fast optimal load balancing algorithms for 1d partitioning. *J. Parallel Distrib. Comput.*, 64(8):974–996, August 2004.

[92] A. Pınar, E. K. Tabak, and C. Aykanat. One-dimensional partitioning for heterogeneous systems: Theory and practice. volume 68 of *11*, pages 1473 – 1486, 2008.

[93] S. Popinet. Quadtree-adaptive tsunami modelling. *Ocean Dynamics*, 61(9):1261–1285, 2011.

[94] Ranger – system specifications. `https://www.top500.org/site/1739`. Retrieved in July, 2016.

[95] F. Rathgeber, D. A. Ham, L. Mitchell, M. Lange, F. Luporini, A. T. T. McRae, G.-T. Bercea, G. R. Markall, and P. H. J. Kelly. Firedrake: Automating the finite element method by composing abstractions. *CoRR*, abs/1501.01809, 2015.

[96] D. A. Reed and J. Dongarra. Exascale computing and big data. *Commun. ACM*, 58(7):56–68, June 2015.

[97] J. Rodrigues Monteiro. Comparison of cartesian and dynamically adaptive grids for the parallel simulation of shallow water waves. Bachelor's thesis, Institut für Informatik, Technische Universität München, February 2015.

[98] P. L. Roe. Approximate riemann solvers, parameter vectors, and difference schemes. *Journal of Computational Physics*, 43(2):357 – 372, 1981.

[99] V. V. Rusanov. Calculation of interaction of non-steady shock waves with obstacles. 1961.

[100] Y. Saad. Practical use of polynomial preconditionings for the conjugate gradient method. *SIAM Journal on Scientific and Statistical Computing*, 6(4):865–881, 1985.

[101] P. Samfaß. A non-hydrostatic shallow water model on triangular meshes in sam(oa)^2. Studienarbeit/sep/idp, Institut für Informatik, Technische Universität München, April 2015.

[102] P. H. Sammon. An analysis of upstream differencing. *SPE (Society of Petroleum Engineers) Reserv. Eng.; (United States)*, 3:3, Aug 1988.

[103] sam(oa)^2 – space filling curves and adaptive meshes for oceanic and other applications. `http://www.github.com/meistero/samoa`. Retrieved in July, 2016.

[104] L. Sant'Ana, D. Cordeiro, and R. Camargo. Plb-hec: A profile-based load-balancing algorithm for heterogeneous cpu-gpu clusters. In *2015 IEEE International Conference on Cluster Computing*, pages 96–105, Sept 2015.

[105] R. Schaller. Dynamic load balancing for riemann solvers with unsteady runtimes. Bachelor's thesis, Institut für Informatik, Technische Universität München, September 2014.

[106] R. Schaller. Parallelization of a non-hydrostatic shallow water model in sam(oa)^2. Studienarbeit/sep/idp, Institut für Informatik, Technische Universität München, October 2015.

[107] K. Schloegel, G. Karypis, and V. Kumar. A unified algorithm for load-balancing adaptive scientific simulations. In *Proceedings of the 2000 ACM/IEEE Conference on Supercomputing*, SC '00, Washington, DC, USA, 2000. IEEE Computer Society.

[108] K. Schloegel, G. Karypis, and V. Kumar. Parallel static and dynamic multi-constraint graph partitioning. *Concurrency and Computation: Practice and Experience*, 14(3):219–240, 2002.

[109] K. Schloegel, G. Karypis, V. Kumar, R. Biswas, and L. Oliker. A performance study of diffusive vs. remapped load-balancing schemes. In *ISCA 11th Intl. Conf. on Parallel and Distributed Computing Systems*, pages 59–66. International Society for Computers And Their Applications, 1998.

[110] M. Schreiber. *Cluster-Based Parallelization of Simulations on Dynamically Adaptive Grids and Dynamic Resource Management*. Dissertation, Institut für Informatik, Technische Universität München, 2014.

[111] M. Schreiber and H.-J. Bungartz. Cluster-based communication and load balancing for simulations on dynamically adaptive grids. International Conference on Computational Science (ICCS), Elsevier, June 2014. short paper.

[112] M. Schreiber, H.-J. Bungartz, and M. Bader. Shared memory parallelization of fully-adaptive simulations using a dynamic tree-split and -join approach. In *HiPC'12*, pages 1–10, Puna, India, December 2012. IEEE International Conference on High Performance Computing (HiPC), IEEE Xplore.

[113] Sequoia – system specifications. `https://www.top500.org/system/177556`. Retrieved in July, 2016.

[114] J. R. Shewchuk. An introduction to the conjugate gradient method without the agonizing pain. Technical report, Carnegie Mellon University, Pittsburgh, PA, USA, 1994.

[115] Sierpinski framework. `http://www5.in.tum.de/sierpinski/`. Retrieved in July, 2016.

[116] SPE10 – Society of Petroleum Engineers comparative solution project 10 model 2. `http://www.spe.org/web/csp/datasets/set02.htm`. Retrieved in July, 2016.

[117] Stampede – system specifications. `https://www.tacc.utexas.edu/stampede`. Retrieved in July, 2016.

[118] H. Sundar, R. S. Sampath, and G. Biros. Bottom-up construction and 2:1 balance refinement of linear octrees in parallel. *SIAM Journal on Scientific Computing*, 30(5):2675–2708, 2008.

[119] SuperMUC – system specifications. `http://www.lrz.de/services/compute/supermuc/systemdescription`. Retrieved in July, 2016.

[120] SWE – an education-oriented code for parallel tsunami simulation. `http://www5.in.tum.de/wiki/index.php/SWE`. Retrieved in July, 2016.

[121] J. Teich, J. Henkel, A. Herkersdorf, D. Schmitt-Landsiedel, W. Schröder-Preikschat, and G. Snelting. *Invasive Computing: An Overview*, pages 241–268. Springer New York, New York, NY, 2011.

[122] Top500 list of supercomputers in june 2016. `https://www.top500.org/lists/2016/06/`. Retrieved in July, 2016.

[123] K. Unterweger. *High-Performance Coupling of Dynamically Adaptive Grids and Hyperbolic Equation Systems*. Dissertation, Technische Universitt München, München, 2016.

[124] L. Velho, L. . de Figueiredo, and J. Gomes. Hierarchical generalized triangle strips. *The Visual Computer*, 15(1):21–35, 1999.

[125] C. A. Vigh. *Parallel Simulation of the Shallow Water Equations on Structured Dynamically Adaptive Triangular Grids*. Dissertation, Technische Universitt München, München, 2012.

[126] T. Weinzierl and M. Mehl. Peano – a traversal and storage scheme for octree-like adaptive cartesian multiscale grids. *SIAM Journal on Scientific Computing*, 33(5):2732–2760, 2011.

[127] T. Weinzierl, R. Wittmann, K. Unterweger, M. Bader, A. Breuer, and S. Rettenberger. Hardware-aware block size tailoring on adaptive spacetree grids for shallow water waves. In *HiStencils 2014 – 1st International Workshop on High-Performance Stencil Computations*, pages 57–64. HiStencils, 2013.

[128] S. Williams, A. Waterman, and D. Patterson. Roofline: An insightful visual performance model for multicore architectures. *Commun. ACM*, 52(4):65–76, April 2009.

[129] M. Wolff. *Multi-Scale Modeling of Two-Phase Flow in Porous Media Including Capillary Pressure Effects.* Dissertation, Universität Stuttgart, Stuttgart, 2013.

[130] Y.-S. Wu, K. Pruess, and Z. X. Chen. Buckley-leverett flow in composite porous media. *SPE Advanced Technology Series*, 1(02):107–116, 1993.

[131] W. Wunderlich. über peano-kurven. *Elemente der Mathematik*, 28:1–10, 1973.

[132] Y. Xing and C.-W. Shu. A survey of high order schemes for the shallow water equations. *J. Math. Study*, 47:221–249, 2014.

[133] Zoltan – parallel partitioning, load balancing and data-management services. `http://www.cs.sandia.gov/Zoltan`. Retrieved in July, 2016.

[134] G. Zumbusch. On the quality of space-filling curve induced partitions. *Zeitschrift für Angewandte Mathematik und Mechanik*, 81, Suppl. 1:25–28, 2001.

[135] G. Zumbusch. Load balancing for adaptively refined grids. *Proceedings in Applied Mathematics and Mechanics*, 1:534–537, 2002.