TECHNISCHE UNIVERSITÄT MÜNCHEN

Department of Informatics
Scientific Computing

Parallel Simulation of the Shallow Water Equations on Structured Dynamically Adaptive Triangular Grids.

Csaba A. Vigh

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.

Vorsitzende:	Univ.-Prof. Gudrun J. Klinker
Prüfer der Dissertation:	1. Univ.-Prof. Dr. Hans-Joachim Bungartz
	2. Univ.-Prof. Dr. Arndt Bode
	3. Assistant Prof. Richard W. Vuduc, Georgia Institute of Technology / USA (nur schriftliche Beurteilung)

Die Dissertation wurde am 09.07.2012 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 11.12.2012 angenommen.

Abstract

One of the most important computational challenges in the context of the numerical treatment of Partial Differential Equations is the generation, management, and dynamic adaptivity of grids. Dynamic adaptivity is extremely important in applications that require frequent changes of the grid pattern during a simulation run. One such application example is Tsunami simulation, where waves must be tracked with highly resolved local grids. Arbitrary unstructured grids that can handle dynamic adaptivity have a considerable memory and computing time overhead. Therefore, the focus of this work is on the Sierpinski space-filling curve-based, recursively structured and dynamically adaptive triangular grid management system.

Space trees recursively split the geometrical domain into smaller sub-domains according to certain predefined sub-division rules. In this thesis we concentrate on the recursive splitting of triangles. The depth-first traversal of the binary refinement tree inherently orders the leaf triangles according to the Sierpinski space-filling curve. We address the challenges of traversal and management of the dynamically adaptive triangular grid, in serial and parallel computing environment. The target application is the parallel simulation of a simplified version of the shallow water equations.

While unstructured grids can require more than 1000 bytes per triangle cell for grid maintenance purposes, our approach uses less than 50 bytes. Due to linearization of the refinement tree, and to the sophisticated data storage and access scheme, the grid traversals exhibit excellent cache hit-rate, and the numerical computation achieves very good MFLOP/sec rates. The re-meshing is about 3.5 times more expensive than one Euler time step in terms of execution time. This ratio will decrease significantly when higher-order discretization schemes will be applied. The full SWE simulation with dynamic adaptivity in each time step reaches up to 90% strong speed-up efficiency. Fast re-meshing and sustainable parallel scaling capabilities will make it possible to run sub-realtime Tsunami simulations with increased number of unknowns, resulting in much higher accuracy than possible before.

Acknowledgments

This work was accomplished throughout five years at the Chair of Scientific Computing in Computer Science of the Technische Universität München. Funding was provided in form of scholarship by the International Graduate School of Science and Engineering (IGSSE) and it is gratefully acknowledged.

I thank my supervisor Prof. Dr. Hans-Joachim Bungartz for his advice and mentoring throughout my work. He convinced me to start this work and created a comfortable working atmosphere that made everything possible. Very many thanks to Prof. Michael Bader, the leader of our research group, for all the great ideas and solutions throughout the last five years. I acknowledge the help and feed-back of Dr. Miram Mehl, the head of the research group focusing on the Peano space-filling curve. Very special thanks to my friend and colleague Kaveh Rahnema for his engagement in debugging and performance measurements towards the end of my thesis work. Without his help it would have taken more time to finish.

Very, very special thanks to Prof. George Biros and Prof. Richard Vuduc for their involvement and countless brainstorming sessions that jump-started the hunt for performance, while, at the same time, I was humping around in crutches and a broken leg, in the hallways of the Klaus building at GeorgiaTech. Those are unforgettable times, I very kindly appreciate their support.

Warm thanks to the friends correcting language-related mistakes, translating from English to German, or debugging weird problems related to overheating of the laptop, when necessary, in the end phase of this Dissertation: Angela Edlinger, Francesco Della Coletta, Diana Babiac, Stefanie Schraufstetter, Vlad Simu, and Andrei Radulescu. I say a very kind “thank you” to all my friends who had to put up with me when things were not that good; they took me to “codename McDonalds”, organized private street festivals, or simply just cooked dinner, when I needed it most. I acknowledge the help of Marouene Mansour and Vlad Dittrich when I broke my leg, the help of Özgür Ertaç for finding a job. Towards the end of my work I needed help in accommodation and I am grateful for the help from François Beaugrand, Kai Redeker, Tobias Jöst, Simina Roşa, Rareş Şufană, Cristian Rădulescu, Carmina Şufană, Cristi Neagu, Kevin Burger, Nuhro Makko, Atyab Imtaar, Georg Lorenz and Ahmet Yilmaz. Thanks to János Benk and Daniel Butnaru for their part of support. I further thank my friends for their support from the famous Felsennelkenanger dormitory, a list that is too long to complete here.

Árpád Kovács and Prof. Béla Bognár are the key people who jump-started my international career, and I thank them a lot for their encouragement. Prof. Bognár – who manages a Hungarian Scholarship Fund in the US – contributed to my studies, when I was in acute need of financing, which I gratefully acknowledge.

My uncle and physics mentor and teacher Jenő “Csocsó” Tellmann was the first person in my life, who got me in love with the realities. Ilona Libál, Katalin Kürthy and Sándor Párhonyi were my math and physics teachers from high-school and their personality and pedagogic skills reinforced that love.

Finally I thank my family, without whom I could not have made it this far. I thank my mother Lidia-Izabella Vigh; my uncles Jenő “Csocsó” Tellmann and Tibor Takácsik; my aunts Éva Tellmann, Antonia Varga and Judit Takácsik; and my cousins Mária Varga, Tibor Takácsik Jr. and Juhász Bernadett.

Table of Contents

1. Motivation for Fast Parallel Simulation of the Shallow Water Equations

On December 26, 2004, 00:58 UTC, a magnitude 9.1 megathrust earthquake occurred approximately 250 km off the west coast of northern Sumatra, Indonesia. It was the third largest earthquake in the world since the year 1900. (Source: USGS – "Largest Earthquakes in the World Since 1900"). It generated the *2004 Indian Ocean Tsunami* that was one of the deadliest disasters in modern history.

The *2004 Indian Ocean tsunami* devastated the shores of Indonesia, Sri Lanka, India, Thailand, and other countries with waves up to 30 meters high and run-up distances as far as 4 km inland. Because of the distances involved on the Indian Ocean, the arrival times range anywhere from 15 minutes to 7 hours to reach several coastlines. It caused serious damage and casualties as far as the east coast of Africa. The death toll from the earthquake, tsunami, and flooding totals over 200,000 casualties, and millions of people lost their homes. The factors partly responsible for the great loss of life mentioned in Kawata et al. (2005) are: many people lived on low-lying coastal areas without refuge buildings, they lacked tsunami knowledge, and there was no early tsunami warning system.

Since 2004, Tsunami forecasting has considerably improved. For example, an observational network of DART buoys (Deep-ocean Assessment and Reporting of Tsunamis) are deployed at strategic locations in the ocean. DART buoy systems use a *Bottom Pressure Recorder* on the ocean floor to send information to the surface buoy that in turn transmits data to ground systems via Iridium satellites. (For details on locations and data of DART systems see NOAA website). According to NOAA "*Tsunami Forecasting*" as well as Titov (2011), seismic measurements and DART data are used for inversion in order to find the tsunami source. This, in turn, is used to sort through a precomputed generation/propagation forecast database to select an appropriate *linear* combination of scenarios that most closely match the observations. These estimates of tsunami characteristics in deep-ocean are then used as initial conditions for non-linear inundation algorithms of specific coastal regions.

Analyzing the effects of the March 11, 2011 Japan Tsunami – which caused a nuclear catastrophe, an estimated 20,000 casualties including some in the US, and flooding damage even in Chile 22 hours after the earthquake – Titov (2011) points out that "*accurate tide predictions are necessary for accurate inundation predictions"*. This is where my thesis work is laying down its contribution.

The expressed goal of this work is to show the feasibility of our grid management methods for a future sub-realtime simulation of oceanic wave propagation. Our grid management implementation, based on the Sierpinski space-filling curve, is simple enough to achieve high serial floating-point performance, is flexible enough to enable full dynamic adaptivity, and is suitable for higher-order spatial and temporal discretization schemes. Parallelization is based on the Sierpinski curve, too, and both the numerical computation and the dynamic adaptivity work in parallel with excellent strong speed-up efficiency well above 80% (chapter 5).

We use a simplified version of the shallow water equations with a finite volume type spatial discretization method and an explicit Euler time-stepping on the two-dimensional, recursively structured, and dynamically adaptive triangular grid. Dynamic adaptivity is essential in order to capture moving features interacting on a variety of scales of different magnitudes, such as the wavefront. The 2D Sierpinski space-filling curve and its special properties are used for grid traversal and adaptive refinement. Access to numerical- or grid-management information is based on stacks and streams in order to achieve excellent cache-efficiency. Communication between parallel partitions is performed via message passing. A special challenge was to preserve/predict grid cell neighbor relationships during adaptive mesh refinement across partition boundaries. Partition-local algorithms had to be developed for all mesh modification tasks, with minimal global communication, in order to facilitate parallel speed-up results similar to those achieved by the numerical computation.

While our discretization method is of low order, the information access pattern on our grid is the same as it would be for a higher-order discontinuous Galerkin spatial discretization method combined with a high order Runge-Kutta time-stepping scheme. A low order simulation with a relatively low amount of floating-point operations per grid cell is the appropriate way to demonstrate the efficiency of our grid management system both in serial- and in parallel computing environment. Adding more FLOPs would only increase both the floating-point performance of single CPU cores as well as decrease the communication-to-computation ratio which, in turn, would yield even better and more sustainable parallel speed-up results.

In our simulation we did not include ocean floor topology, but instead used a simple “swimming pool” setting in which a column of water is collapsing and causes a wave that propagates outwards. Figure 1.1 illustrates eight snapshots of such a simulation on 9 processors (represented by different colors) with dynamic adaptivity and load-balancing. Currently the simulation does not run sub-realtime; mainly because of the low-order discretization schemes, and in part because of the relatively low amount of parallel processing units used. It is assumed that a higher-order discretization scheme would increase the simulated time-step faster than it would increase the real run-time to compute it, making the goal of a sub-realtime simulation on an appropriate number of parallel processors look ever more realistic.

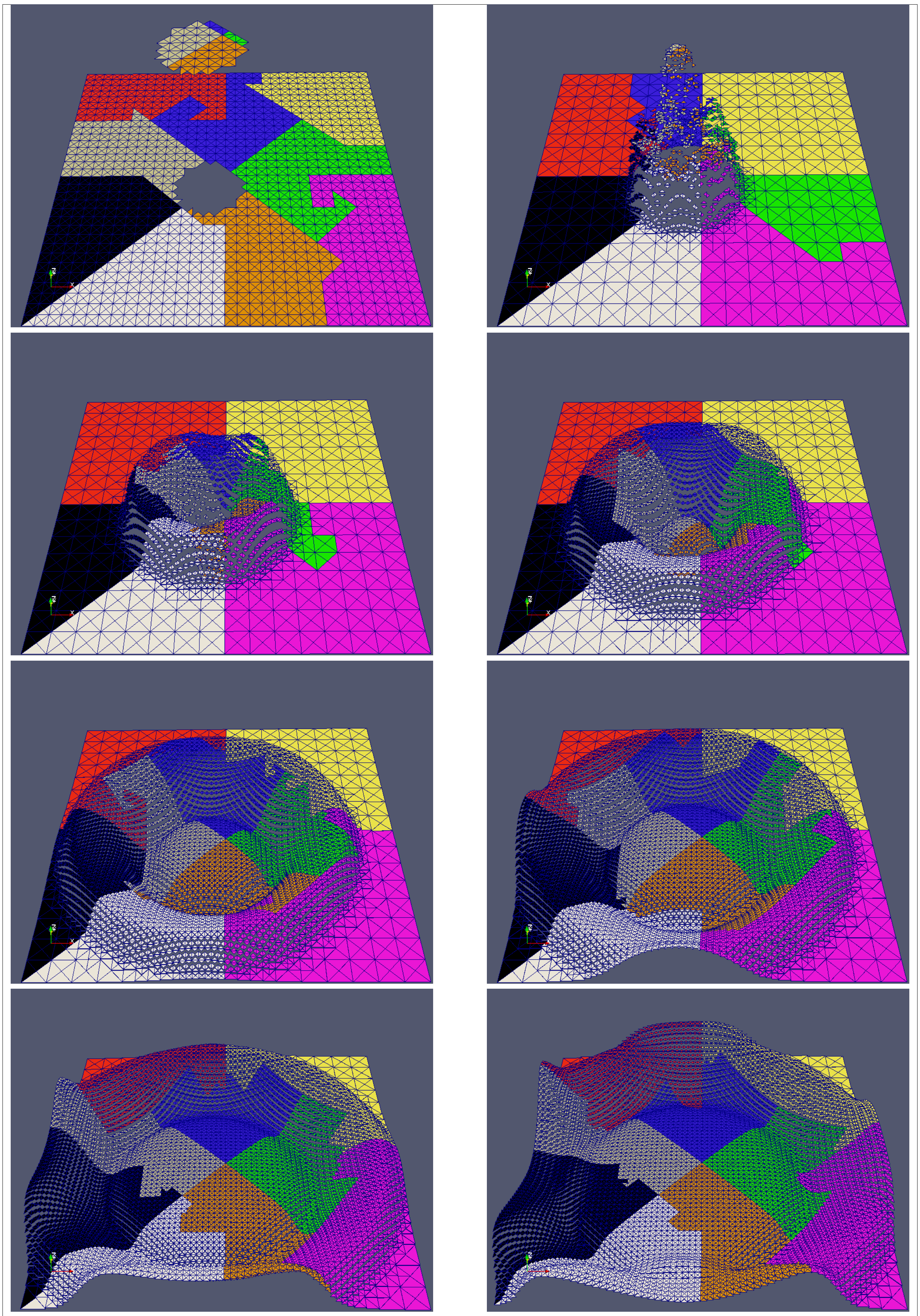


Figure 1.1: Parallel SWE simulation with adaptive mesh refinement on 9 CPUs.

1.1. Shallow water equations for tsunami modeling

The shallow water equations (SWE) are a simplified version of the classical fluid dynamics equations that can be derived by averaging over the depth dimension. They are used in scenarios where a homogeneous solution in the vertical direction may be assumed, or where the characteristic wave length of the horizontal wave propagation is large in comparison to fluid depth.

The average depth of the Indian- or the Pacific ocean is 4 kilometers, and the width in latitudinal and longitudinal direction has a range of a couple of thousand kilometers (LeVeque, 2007). A Tsunami wave propagation has a wave length as long as 200 kilometers (LeVeque, 2007). As a consequence, the 2D SWE are successfully used to simulate oceanic wave propagation in tsunamis generated by earthquakes or asteroid impacts (Behrens 1998; George and LeVeque 2006; Glimsdal et al. 2004; Harig et al. 2008; Imamura et al. 2006; LeVeque 2007; LeVeque et al. 2011).

In this work we use a simplified version of the SWE in which the unknowns are the water height ξ and the velocities in X- and Y direction $\mathrm{v} = (u, v)$. Formula 1.1 uses the formulation from Aizinger and Dawson (2002) and from Remacle et al. (2006), but neglects viscosity, friction, and Coriolis forces.

$$\frac{\partial \xi}{\partial t} + \nabla \cdot (\mathbf{v}\xi) = 0,$$

$$\frac{\partial \mathbf{v}}{\partial t} + \mathbf{v} \cdot \nabla \mathbf{v} + g \nabla \xi = 0$$

Formula 1.1: Simplified SWE neglecting viscosity, friction and Coriolis forces.

However, both the simplified version as well as the full SWE can be rewritten in a vector form as in Formula 1.2.

$$\frac{\partial \mathbf{u}}{\partial t} + \mathbf{div}\,\mathbf{F}(\mathbf{u}) = \mathbf{r}$$

Formula 1.2: Generalized vector form of the shallow water equations.

The unknown $\mathrm{u} = (\xi,\ \xi u,\ \xi v)$ is slightly adjusted to hold the discharge instead of the velocities, and vectors $\mathrm{F(u)}$ and r are suitably chosen. This way our computational and numerical approach to solving the system in Formula 1.1 is also easily transferable to the full SWE. For more details see chapter 2.3.

1.2. Discontinuous Galerkin and finite volume discretization for solving the SWE

The discontinuous Galerkin (DG) method has recently become a popular approach to solving the SWE (Behrens, 1998; Dawson et al. 2006; Eskilsson and Sherwin 2005; Giraldo, 2006). It is strongly element oriented, and data is exchanged between grid cells only via the so-called flux terms. A global system of equations needs not be assembled, instead, computation is local on the grid cells. In the 2D case it means that the unknowns are cell-based and neighboring grid cells exchange data – the flux term – through the common edge. There is absolutely no need for node-based unknowns, and, therefore, I eliminated the topological information regarding the nodes from our Sierpinski-based grid management. This simplification contributed a lot in achieving good floating-point performance by reducing grid management overhead. The parallel data access pattern during numerical computation is also simple; data is only exchanged through edges that lie on the process boundary.

Further advantages of the discontinuous Galerkin method are, for example, the stable description of the convective transport and the fitness for adaptive solution techniques (Cockburn et al. 2000). The numerical computation in any grid cell depends solely on the internal unknowns and the (information-) flow across the surrounding edges. The numerical operators are independent of the size or shape of the neighboring grid cells, and they operate the same way whether one uses a uniform grid or a conforming adaptive grid with no hanging nodes. Our Sierpinski-based grid is dynamically adaptive and conforming, and adaptation is triggered by a cell-local criterion following the dynamical change of the solution.

Another advantage of the DG method is its support of high-order accuracy by simply increasing the order of the polynomial basis functions (Cockburn et al. 2000). Our core numerical routines, however, only use constant (0^{th} order polynomial) basis functions in each grid cell. This piece-wise constant approximation is similar to a finite volume type method (Aizinger and Dawson, 2002; Remacle et al. 2006). For time discretization we use a simple explicit Euler time-stepping scheme that is computed in a single grid traversal. Since the goal of this work is to provide a memory-, MFLOPS-, and parallel efficient dynamically adaptive grid generator for future tsunami simulations, the numerical accuracy was not a main priority.

A real-life tsunami simulation would require the use of the full SWE equations, higher-order accuracy DG discretization, a higher-order Runge-Kutta type time integration and geometry information (ocean floor depth, islands and coastline). The full SWE equations in their vector form are similar to our simplified version, and extending the numerical kernel should be fairly simple. A higher order DG method needs additional memory for the unknowns as it stores higher order polynomials in each grid cell, and the flux terms are also somewhat different. It requires more floating point computations per grid cell, but the data access pattern remains the same as in our simplified case (details in chapter 2.3). Instead of one grid traversal per Euler time step, a Runge-Kutta time integration needs one traversal per slope evaluation. It is believed that such a simulation with higher-order overall accuracy would increase the floating point performance as well as the parallel speed-up. This is also suggested by some of the performance experiments in chapter 5.

1.3. Parallel adaptive mesh refinement and coarsening

The piece-wise constant discontinuous Galerkin – or finite volume-type – discretization of the simplified shallow water equations runs on our Sierpinski space-filling curve-based, recursively structured, triangular, and dynamically adaptive grid.

Adaptivity is essential when observing/simulating phenomena with interaction on a variety of scales. It can help to resolve local small scales that interact with global scales in a consistent way (Behrens, 2005a). The coastline may need a fine grid resolution of perhaps a few meters, whereas the global dynamics that drives the oceanic wave propagation is perhaps four or five orders of magnitude higher. The prescribed and fixed refinement regions – islands and coastline – are determined by the a-priori knowledge of experts. In contrast, dynamic adaptivity can capture moving features like propagating wave fronts. Dynamic adaptivity – especially in the tsunami wave front tracking – requires frequent re-meshing according to certain dynamic refinement criterion that is based on the evolution of the solution.

There are several reasons why adaptive grids are used instead of uniform ones. Assuming that a vast area like the Indian Ocean is covered with a uniform grid of mesh size that is small enough to capture small scale features of a few meters, a lot of computing would be performed in huge regions where the solution is uniform. Furthermore, there is a limit of available memory in any computer, in which all unknowns would have to be stored. A fixed but adaptive grid will save both memory and computing time. On the other hand, a dynamically adaptive grid will inevitably add some re-meshing overhead. Even if the saved computing time is mostly spent on the grid modification, because of the limited amount of available memory, a dynamically adaptive grid may still be the only option to capture the moving features of a tsunami wave. In our Sierpinski-based grid management a re-meshing step costs about two to three Euler time-steps in terms of computing time (see chapter 5). This ratio would shrink even further in the case of a higher-order DG implementation with Runge-Kutta time-stepping.

Adaptive grids can be structured or unstructured. While structured grids have some sort of rule to access their elements, usually an iteration over an index vector, unstructured grids require large amounts of additional memory to explicitly store their topological structure. Refinement and coarsening of an unstructured grid offers unlimited flexibility, but it is computationally quite expensive and usually can not be afforded in each time-step. Solvers implemented on data structures used in unstructured grids tend to have inefficient access patterns to memory which leads to poor cache hit rate, poor vectorization properties, and poor overall computational performance. Structured grids are more advantageous in terms of cache hit rate, vectorization and computational performance; but often lack the ability for adaptive refinement or coarsening.

Our Sierpinski grid management system offers an excellent trade-off between overall computational performance and flexibility for dynamically adaptive refinement and coarsening. Initial grid generation is based on recursive subdivision of triangles, which was introduced as the *newest vertex bisection* method by Mitchell (1991) and Bänsch (1991). Grid traversal is on the predefined order induced by the Sierpinski curve, and numerical computation is done locally in every grid cell without the need to assemble and solve a global system of equations. Traversal is implemented with linearized binary trees which permit loop-based traversals over arrays instead of recursive function calls common for traversing a full binary tree. Another advantage is the easiness of starting and stopping a traversal on a parallel partition without having to consider the binary refinement tree. Grid topology, or neighbor information, is implicitly represented by a system of stacks and streams, instead of explicitly storing it. Stack- and stream based implementation enables the compilers to produce well-optimized code, and hardware prefetching is also more successful in fetching the right data from the memory to the CPU. As a consequence, the grid traversals exhibit excellent cache hit-rates, as we published in Bader et al. (2012).

Our adaptive mesh refinement produces conforming grids in the same manner as **amatos** from Behrens (2004), which is implemented for oceanic applications, and it is a reference for our approach. Refinement cascades may be triggered by a single refinement of a triangle cell in order to eliminate hanging nodes. Parallelization of grid traversals is done using the Sierpinski space-filling curve. Refinement cascades need to propagate across parallel partition boundaries. In order to rebuild the stack-and-stream based system that represents the implicit topological information, neighbor index prediction is performed for the new grid. All grid manipulation tasks are traversal-based and partition-local. The amount of global communication, other than load-balancing, is of the order of the number of parallel partitions, rather than overall grid size. Load-balancing is done in an MPI_All-to-All fashion in which the triangle cells get redistributed equally among the parallel processors.

The efficiency aspects of our Sierpinski-based grid management were partially demonstrated in Bader and Zenger (2006), Bader et al. (2008 and 2012). A more recent performance analysis is given in chapter 5.

1.4. Related work

There are several recursively structured grid generators in use, and some of them use similar ideas and solutions to the arising problems as we do. The following grid generators were sometimes used for checking correctness of our algorithms, and were sometimes the inspirational source for innovation.

The triangular grid generator **amatos** from Behrens (2004) is primarily intended for atmospheric and ocean modeling – tsunami simulation included. It uses a set of coarse initial triangles that cover the area of interest, and these triangles are recursively bisected at a *marked edge* until the required mesh size is reached. This grid generation process, as mentioned before, is also referred to as *newest vertex bisection*. Similar to unstructured grids, amatos stores the grid topology explicitly. Numerical computation is done with a so called *gather-scatter approach* in which a global system of equations is built and solved. Dynamic adaptivity produces *conforming grids* with no hanging nodes allowed. Additional refinements or cascades of refinements are triggered to eliminate the hanging nodes. amatos also uses the Sierpinski space-filling curve for parallelization and partitioning.

Some initial Sierpinski-based algorithms were running on the amatos grid that was also used to check their correctness. In contrast to amatos, our approach stores almost no explicit topology information, but replaces it with a system of stacks and streams that use a minimal amount of memory. We also avoid building a global system of equations, and the computation is performed locally on the grid cells.

Peano is a grid generator based on recursive trisection of squares, cubes, or any higher-dimensional hypercubes recently presented by our research group. It is a family of methods that uses the Peano space-filling curve for grid traversal and processing with a low memory footprint, and implements parallel iterative multigrid solvers with full adaptivity and load-balancing – see Günther et al. (2006), Mehl et al. (2006), Weinzierl (2009) and Neckel (2009). The topology is not stored explicitly; a system of stacks is used instead, just like in the Sierpinski case. Numerical computations are performed locally on the grid elements without building any global system of equations. Adaptive refinement also has restrictions that may trigger refinement cascades: the refinement level of any two geometrically neighboring grid cells may not differ by more than 1 in the refinement tree. Parallelization is done with the help of the Peano space-filling curve.

Peano could be considered the 'closest relative' of our Sierpinski-based grid management because many of the solutions are similar in nature and differ only in details of implementation: initial grid generation based on recursive subdivision; grid traversal based on space-filling curves; topology information replaced by a system of stacks; numerical computation local on the grid without building global system of equations; adaptivity restriction to maintain grid conformity; parallelization with space-filling curves.

Dendro is a rectangular grid generator package based on recursive bisection of squares or cubes using quadtrees or octrees. It is open-source software from the group of Prof. G. Biros available at "www.cc.gatech.edu/csela/dendro". The octree is linearized in the Morton order in which the leaf nodes are stored in an array together with their locational codes. Element connectivity is stored in a compressed format with look-up tables that help evaluate a partial differential operator in only one loop-based traversal (Sundar et al. 2007a and 2007b). A global system of equations is assembled and solved. Adaptive refinement restriction is the same as in the Peano case, and it is called *2:1 balancing;* the refinement cascade is called *ripple effect*. All operations are also efficiently parallelized.

The idea from Dendro that most inspired me was the linearized octree. In our simulation we only perform operations on the leaves of the refinement tree, and, by storing them in a linear array, we can perform loop-based grid traversals instead of recursive ones. Advantages of loop-based traversals are clearly shown in chapter 5. Load-balancing is also less complicated when the data that has to be exchanged between processors is in the form of simple arrays, instead of complex data structures such as binary trees.

GeoClaw is a software package used for example by LeVeque (2007) and LeVeque et al. (2011) for Tsunami simulations with finite volume methods on rectangular grids. It is used for depth-averaged flows – like the two-dimensional SWE – with adaptive mesh refinement. Adaptive mesh refinement is used on specified regions, and dynamic adaptivity automatically follows the wave. With a moving wet/dry interface of the grid cells, inundation is simulated as well.

p4est is a software library with scalable algorithms for parallel adaptive mesh refinement on forests of octrees in both 2D and 3D (Burstedde et al. 2011). Similar to Dendro, the octants are stored in linear arrays following the Morton order. Adaptive refinement and coarsening can be performed both in non-recursive and in recursive ways, maintaining the 2:1 balance constraint. The non-recursive version replaces octants with its children or their common parent, and the recursive version can radically change the grid in one call. All routines are shown to scale well on over 200,000 CPU cores with good speed-up efficiencies.

Our Sierpinski grid management system incorporates a variety of features and methods that may be found in some of the mentioned grid generators, and combines them in a way that yields both very good serial floating-point performance and impressive parallel speed-up for the solution of the SWE. amatos generates the same grid as the Sierpinski system, but it carries huge overhead by storing topology information explicitly. Peano would generate a similar dynamically adaptive rectangular grid, but it does not linearize the refinement tree, which in our case leads to excellent serial floating-point performance. In the publications about Dendro, dynamic adaptivity in-between computation traversals is not mentioned, and GeoClaw is not yet mentioned to work in parallel. We believe that the current combination of functionality incorporated in our Sierpinski grid management system, together with future enhancements of the mathematical model and discretization schemes, will enable the sub-realtime simulation of oceanic wave propagation. This, in turn, could be used in future tsunami early warning systems, and would lead to faster and more precise hazard estimation.

In the next chapters I will describe in detail how the Sierpinski grid management works. Chapter 2 describes the initial grid generation process, the space-filling curve driven traversals, and the numerical computations. Dynamic adaptivity is the subject of chapter 3. How all the above works in parallel is shown in chapter 4, and chapter 5 will present the performance analysis of our code. Conclusions and future work will be discussed in chapter 6.

2. Sierpinski-based Grid Management

This chapter describes the initial grid generation process: the classification of the triangle cells according to properties induced by the Sierpinski space-filling curve, the stack-and-stream based edge system that transports information between neighboring triangles during a grid traversal, and how discontinuous Galerkin computations are performed. These algorithms we partially published in Bader and Zenger (2006), Bader et al. (2008 and 2010). Space-filling curves in general are well described in Sagan (1994).

2.1. Triangle system and the Sierpinski curve

The triangular grid is recursively constructed with the method known as *newest vertex bisection* which was introduced by Mitchell (1991) and Bänsch (1991). A depth-first traversal of the accompanying binary refinement tree orders the triangles according to the Sierpinski curve. Basic properties of single triangles, based on their relationship to the Sierpinski curve, are introduced. These properties are used to define the edge system and to synchronize to it during grid traversals.

2.1.1. Recursive construction and traversal

The grid generation starts by covering the area of interest with an initial set of coarse triangles, like in Figure 2.1. In this work we only use the unit square arrangement.

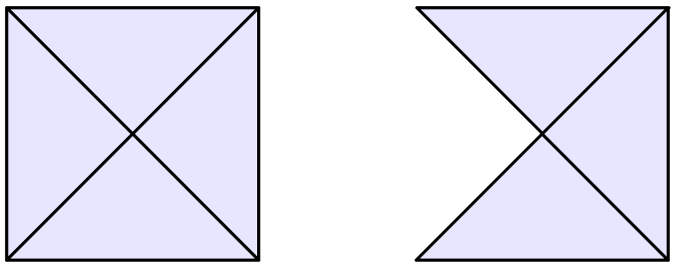


Figure 2.1: Initial triangulation examples. Unit Square (right) and Re-entrant Corner (left). Throughout this work the Unit Square is used.

These initial parent triangles are recursively split at so-called *marked edges* until the desired mesh size is reached (see Figure 2.2). Our implementation works with right isosceles triangles only, and the hypotenuse takes the role of the marked edge. This method is used by amatos, too, and can be extended to arbitrarily shaped triangles as long as the topological structure of the recursive bisection is preserved, as described in Behrens et al. (2005c).

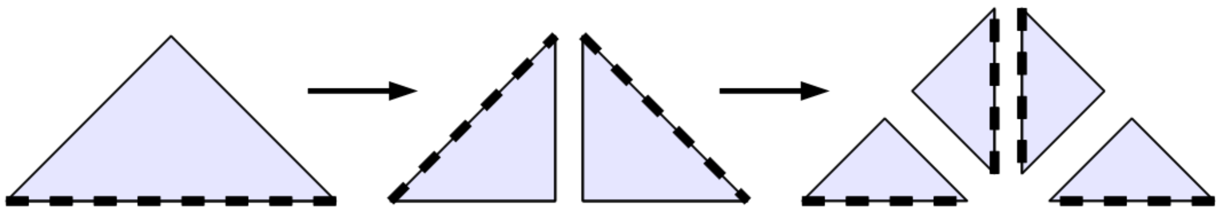


Figure 2.2: Newest vertex bisection: recursive splitting along the marked edge.

The recursive bisection can be uniform on the entire grid, or it can be adaptive too, as the example in Figure 2.3 illustrates.

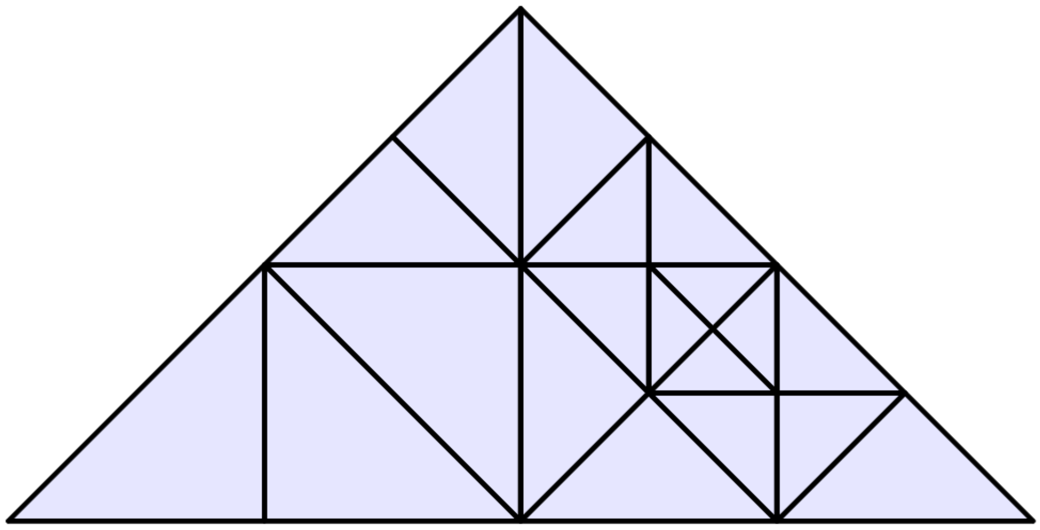


Figure 2.3: Recursive bisection of the initial triangle, with adaptivity.

Whether uniform or adaptive, the refinement process can be represented by a corresponding binary tree which we call *refinement tree*. Figure 2.4 below illustrates the refinement tree that is used to construct the adaptive grid in Figure 2.3. The storage costs of a binary tree are 1 bit per tree node, which is the equivalent of 2 bits per leaf triangle because the number of leaves is half of all the nodes.

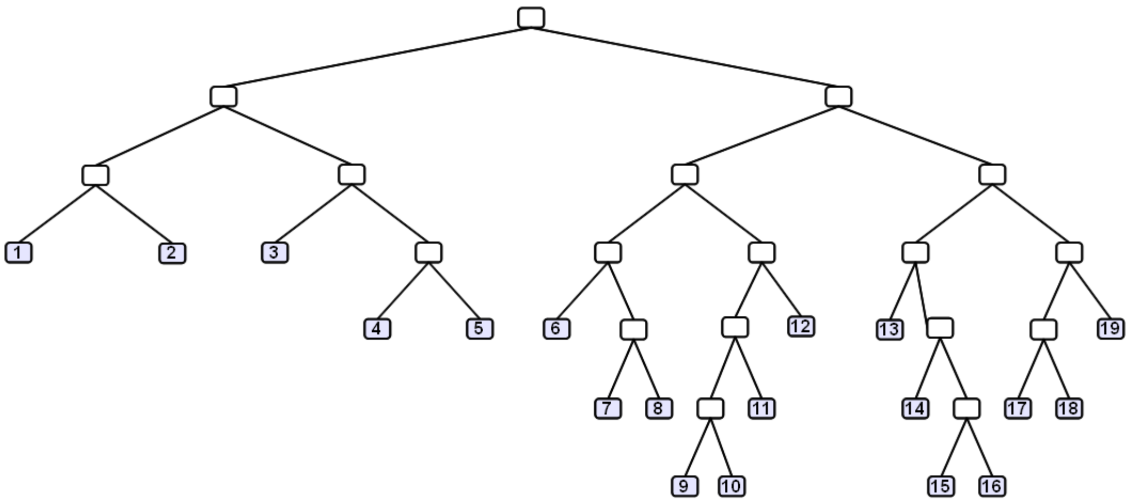


Figure 2.4: Refinement tree used to refine one of the the initial triangles.

A depth-first traversal of the refinement tree orders the leaves, and, consequently, the grid cells, according to the Sierpinski space-filling curve (see Mitchell, 2007 and Sagan, 1994), as shown in Figure 2.5.

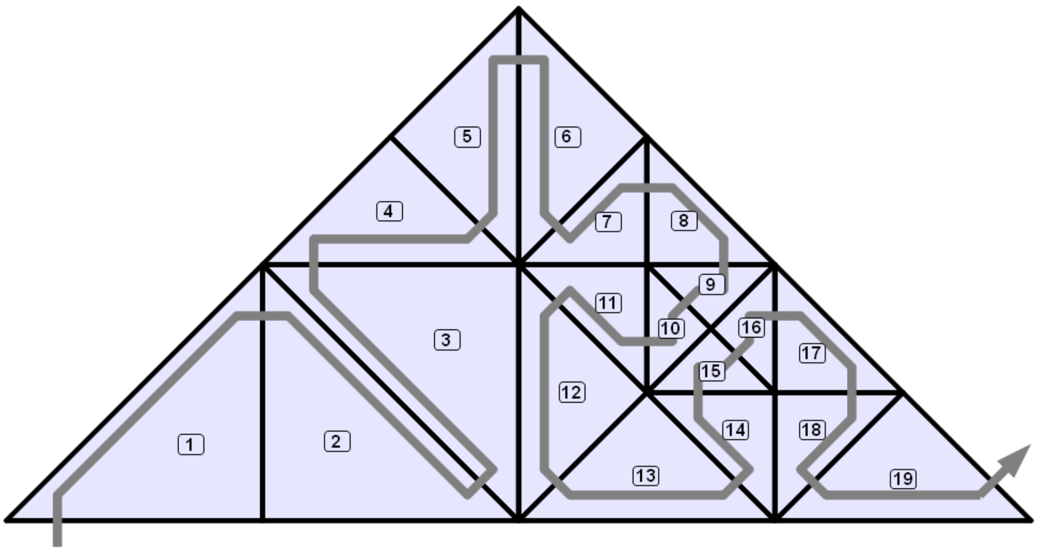


Figure 2.5: Depth-first traversal of the refinement tree gives the Sierpinski order of the triangles. Here the Sierpinski curve runs along the hypotenuse.

The first implementation of this storage scheme was in Vigh (2007), and it substituted the triangle system supplied until then by amatos for the work of Schraufstetter (2006) in which several Sierpinski-based numerical- and topological methods from Bader and Zenger (2006) were implemented and tested.

The implementation of numerical algorithms requires knowledge of the neighbor relationship between cells, faces, edges, and nodes in order to evaluate local discretization stencils. From the refinement tree alone, these relationships are not easily available. While a full grid traversal is an *O(N)* algorithm – whether fully recursive or optimized loop-based – instead of *O(1),* the random access to a specific element is of order *O(log N)*, which is the depth of the binary tree. For this reason, most grid generators that work with adaptive grids invest considerable amounts of memory to store the neighbor relations explicitly. Our aim was to replace most of the topological information with an efficient stack-system that uses a minimal amount of memory but still facilitates numerical computations. The stack-system is presented in section 2.2, and it is based on properties and classifications of the grid elements that are based on the Sierpinski curve. Sections 2.1.2 to 2.1.4 define these properties and their recursive nature as seen in Bader and Zenger (2006).

2.1.2. Sierpinski color classification

The approximation of the Sierpinski space-filling curve does not fill the space, but rather, splits it into two sides. Looking at the picture in Figure 2.6 the nodes on the left-hand side of the curve in the traversal direction are colored in green, and the others are in red. This is the *RED/GREEN* color classification of the nodes.

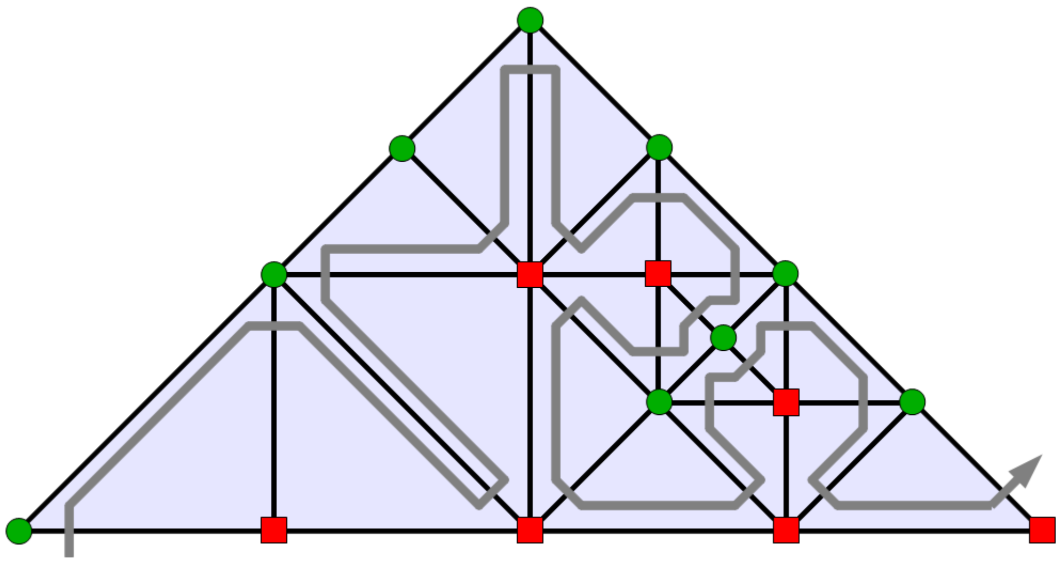


Figure 2.6: Red and green nodes (boxes and circles). The Sierpinski curve separates the nodes on the left-hand-side from those on the right-hand-side.

The *RED/GREEN* classification can be extended to those edges that connect nodes of the same color, as seen in Figure 2.7. These red or green edges are collectively called *color-edges*. The other edges that connect two nodes of different colors are crossed by the Sierpinski curve and are called *crossed-edges*.

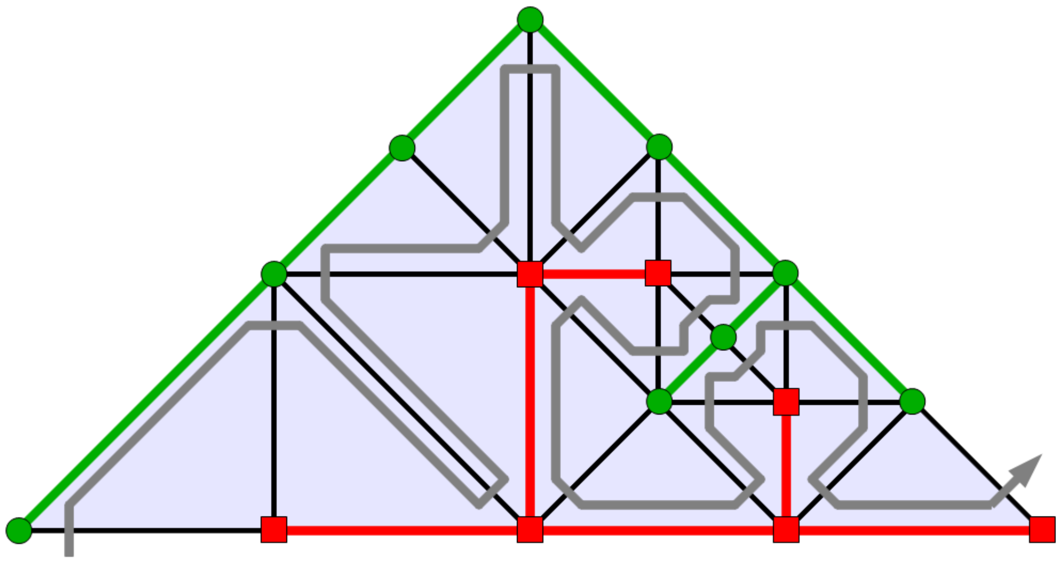


Figure 2.7: Red-, green- and crossed edges (black). Edges that are entirely on one side of the Sierpinski curve have the color of the nodes they connect.

The color of a whole triangle will be by definition the color of its node in the right-angle corner. An equivalent but more precise definition is the color of the node opposite to the *marked edge*, which in our current implementation is the node opposite to the hypotenuse – at the right angle. This choice of definition is made in order to clearly and uniquely define a triangle's color; a property that can easily be computed for the child triangles during recursive bisection.

Figure 2.8 shows the colors of the triangles as well. The color of a triangle does not necessarily correspond to the color of its color-edge. If the color-edge is a leg, then it has the same color as the triangle. On the contrary, if the hypotenuse is the color-edge, then it has the opposite color than the triangle. This ambiguity is solved by the entry/exit classification in the next section.

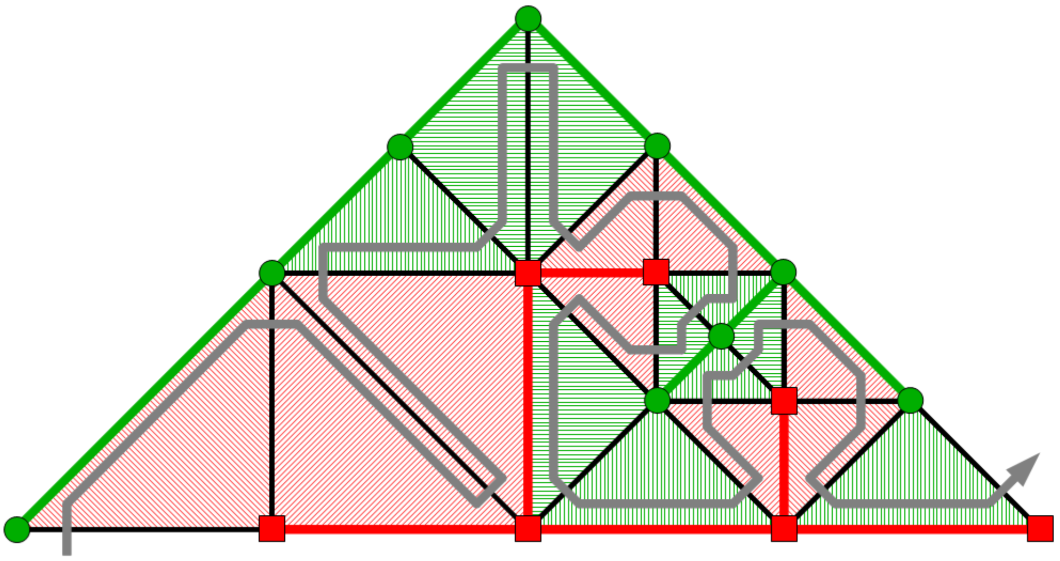


Figure 2.8: Red- and green triangles. The reference color of a triangle is always the color of the node on the right angle. Note: the color of a triangle does not necessarily match the color of its color-edge.

2.1.3. Sierpinski entry/exit and old/new classification

Regarding individual triangles, there are three different ways the Sierpinski curve can enter and exit them, as displayed in Figure 2.9. Each triangle has one color-edge and two crossed-edges. Based on the entry edge, classification is as follows:

- curve enters through the hypotenuse and exits through a leg is called type *H*
- curve enters through a leg, has two possibilities to exit:
  - exit through the other leg is called type *V* (because of the up-side-down shape of the entry-exit legs)
  - exit through the hypotenuse, and is called type *K* (from the German word “Kathete”, which means the entry leg).

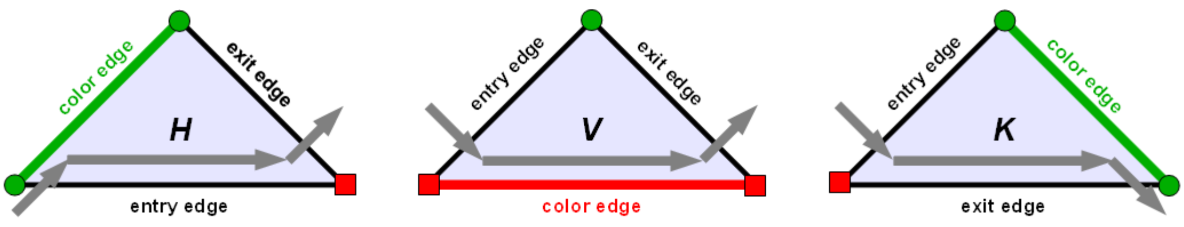


Figure 2.9: Sierpinski entry/exit or H-V-K classification.

If the *RED/GREEN* color and the *H-V-K* type of a triangle is known, then the color of the color-edge can be determined as follows:

- if type is *H* or *K*, the color-edge is a leg and has the same color;
- if type is *V*, the color-edge is the hypotenuse and has the opposite color.

The color of any node on a color-edge takes the same edge color by definition, and the node opposite to the color-edge takes the other color.

During a grid traversal along the Sierpinski curve, the triangle cells are accessed in a prescribed order exactly once. The edges are accessed exactly twice, unless they are on the domain boundary, in which case they are accessed only once per traversal. The nodes are accessed several times ranging between 1 and 8. When implementing a numerical algorithm it might be useful to know whether we access a certain edge or node for the first time, for the last time, or sometime in between.

Figure 2.10 shows that the entry edge is always *old*, and the exit edge is always *new*. The color-edge is the only one that is undefined. Therefore, this *old/new* property of the color-edge in a certain triangle is introduced as an attribute of the triangle itself, and will be tracked. The triangle names *H*, *V* and *K* will get an “*o*” or an “*n*” index for *old* or *new*.

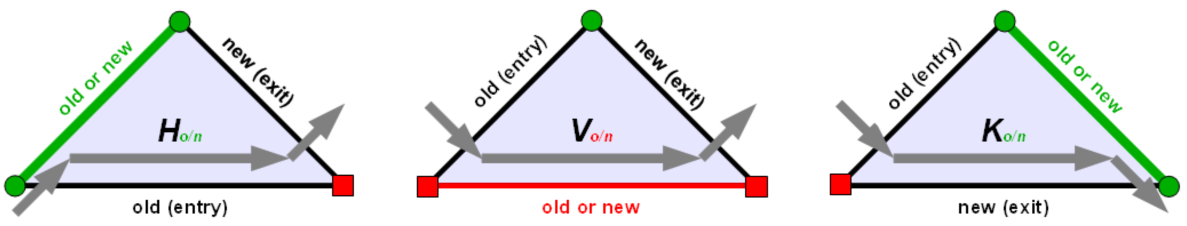


Figure 2.10: Sierpinski old/new classification.

The *old/new* property of the edges clarifies the first or last access to them. If an edge is *new* in a certain triangle cell, then it is encountered for the first time. If it is *old*, then it is encountered for the last time in the current traversal.

The availability of the *old/new* attributes for all edges in a triangle yields the following access types to the nodes:

- node is encountered for the first time if both attached edges are *new*;
- node is touched between first- and last time (neither first, nor last) if one attached edge is *new* and the other is *old*;
- node is encountered for the last time if both attached edges are *old*.

Note that in case of a reverse traversal the types *H* and *K* switch roles, while type *V* stays the same. If a certain triangle was *old* in one traversal direction, then it will become *new* in the reverse direction. The color property of any grid entity is traversal-invariant.

2.1.4. Bisection and color-, entry/exit-, old/new properties

The availability of only the color-, entry/exit-, and old/new type of a triangle makes it possible to derive the colors and access types (first, last, intermediate) to all of its edges and nodes. The bisection rules illustrated in Figure 2.11 show the downward propagation of these attributes during a bisection. Knowing these properties for all the initial coarse triangles, and applying these rules recursively, makes it possible to compute the respective properties of the leaf triangles.

The color of the child triangles is always the opposite of the parent triangle because the newly created node and the right angle of the children will lie on the hypotenuse of the parent. The hypotenuse of the parent splits and at least one of the child edges becomes a color-edge. The bisecting edge is always a crossed-edge.

The *H-V-K* classification and the *old/new* properties change according to the illustration in Figure 2.11, and are formalized in Formula 2.1 – the bisection rules. The numbering of the edges and nodes during a bisection are shown in Figure 2.12. These rules make it possible to compute all attributes and all coordinates in the grid starting from the initial coarse triangles during a recursive initialization traversal.

$$H_o \rightarrow (V_o, K_o), \quad H_n \rightarrow (V_n, K_o),$$

$$V_o \rightarrow (H_o, K_o), \quad V_n \rightarrow (H_n, K_n),$$

$$K_o \rightarrow (H_n, V_o), \quad K_n \rightarrow (H_n, V_n).$$

Formula 2.1: Bisection rules.

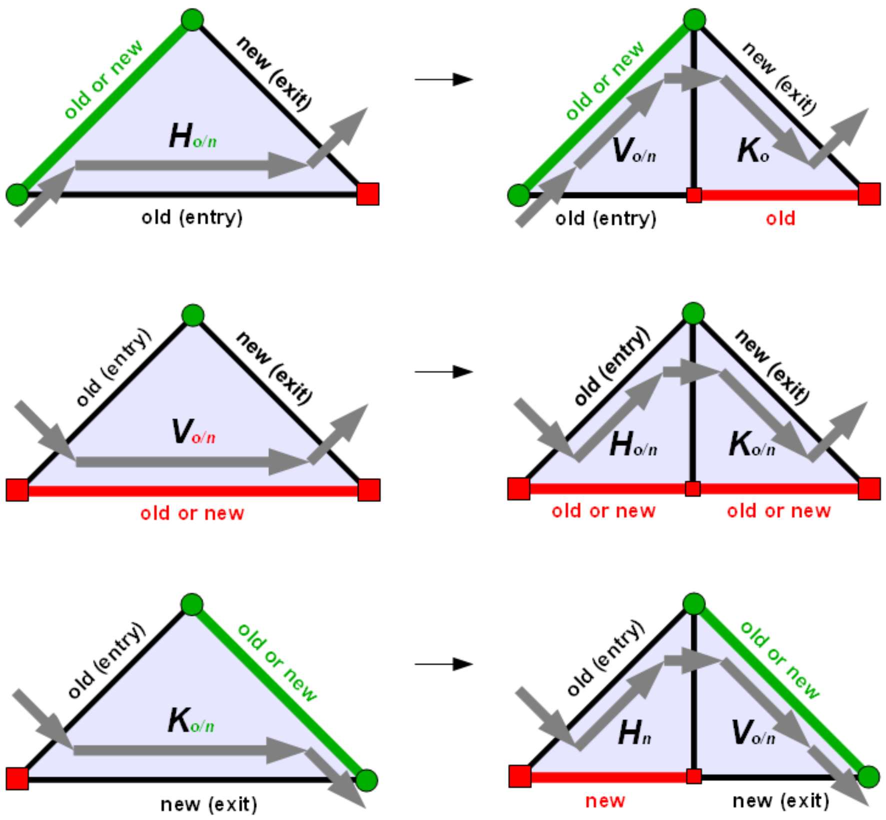


Figure 2.11: Bisection rules for color-, entry/exit-, and old/new attributes.

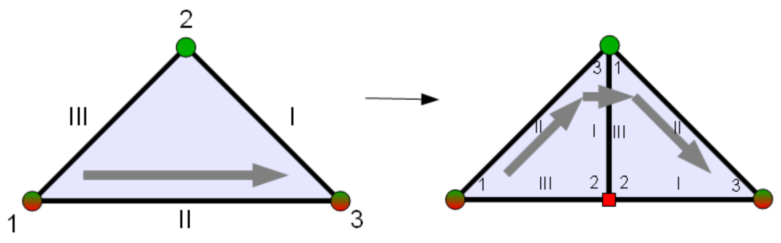


Figure 2.12: Numbering of nodes and edges before and after bisection.

2.2. Edge-stack system – an alternative to explicit neighbor indexing

During traversal and element-wise processing, in each triangle numerical methods need access to the neighboring triangles. If the triangles are ordered according to the Sierpinski curve, two out of three neighbors are in consecutive memory space: the previous and the next triangle. The third neighbor is located somewhere else in memory, and is called the *far neighbor*, as illustrated in Figure 2.13. The information flow to and from the previous- and next triangle can be dealt with on the fly, but between the current triangle and its far neighbor a more sophisticated solution is needed.

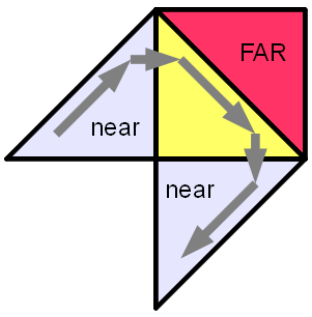


Figure 2.13: Near and far neighbors in memory.

For the discontinuous Galerkin method it would suffice to have the following in each triangle: index of the far neighbor, or a pointer to it, as well as additional storage for an intermediate flux value, that is stored for when the traversal reaches the far neighbor. I call this the *triangle-only solution*. This solution, however, needs twice the amount of memory for the information flow through the color-edges – as storage is allocated in both triangles and only used in one of them – than an *edge system* that holds the exact amount of color-edges with the flux values for transfer.

The introduction of any *edge system* requires some connectedness to the triangle cells. A simple solution is for each triangle to have a pointer or an index to its corresponding color edge. A more sophisticated solution is to have an edge system without explicit storage of the connections between triangles and edges, and also to have some operations with which the two systems can be synchronized to each other. Such a system exists and is called *edge-stack system*. Its operation is based on the *color-*, *entry/exit-,* and *old/new* properties of the grid entities and, hence, it is strongly connected to the Sierpinski space-filling curve. A performance comparison between *triangle-only* and *edge-stack system* solutions will be given in chapter 5 section 5.1.3.

Other methods also need access to unknowns placed on nodes. A *node-stack system* exists as well, and its rules of operation are very similar to the edge-stack system because the access rules to the nodes are based on the edges, as seen in section 2.1.3. The next section describes the edge-stack system on a macro scale as it synchronizes to the triangle traversal. A description of the detailed operations upon entering and exiting a triangle or crossing between two triangles is presented in section 2.2.2.

2.2.1. Edge-stack system during traversal

Figure 2.14 illustrates a triangular grid recursively refined up to depth 6 with the Sierpinski curve crossing it. The best spot to observe the stack-like access pattern is to look at the inner red edges in the middle; then follow the curve ascending on their left-hand side and descending on the right. In between the first- and the last access to these red edges they may be pushed on top of a temporary stack, and then popped from it. This is also true for the green inner edges.

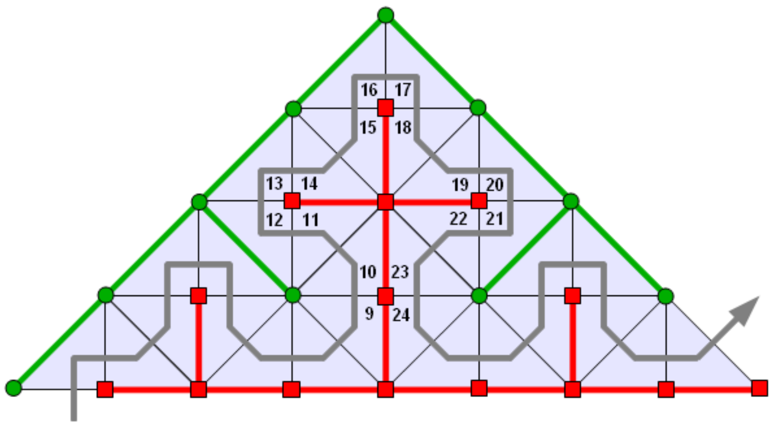


Figure 2.14: Stack-like access to the inner color-edges along the Sierpinski curve.

Assuming the red edges are sorted on an input stream according to their first encounter, they can be taken from such a stream before first use. After the last (second) usage these edges can be written to an output stream. If the first- and last access coincides in the case of a domain boundary, the edge will be taken from the input stream and written directly to the output stream. On the output stream these edges are then sorted according to their last encounter. This way, when the traversal changes direction, the old output stream will be processed backwards as the new input stream. Figure 2.15 sketches the concept of the stack system with separated red and green input- and output streams and temporary stacks. Figures 2.16 and 2.17 animate a portion of the traversal.

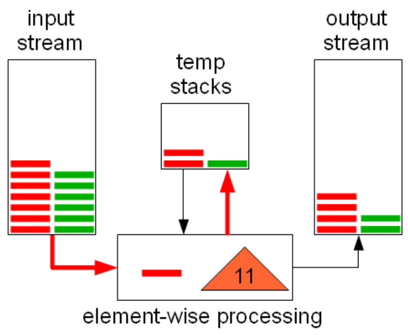
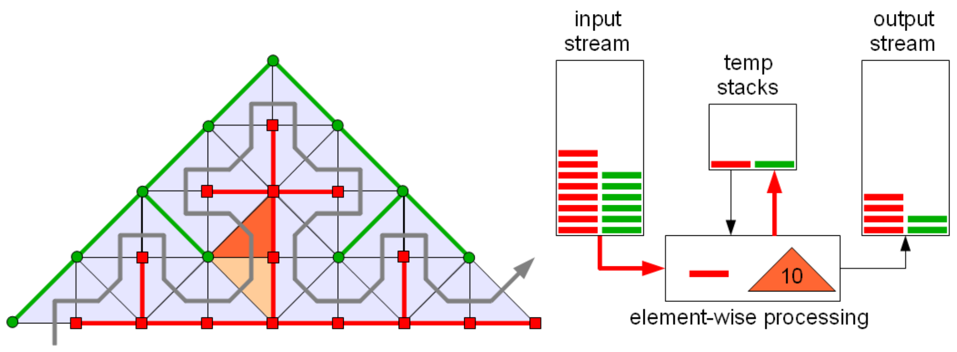
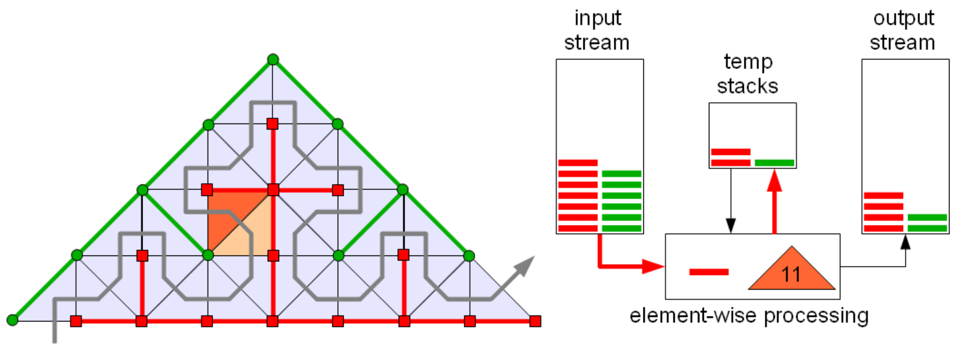


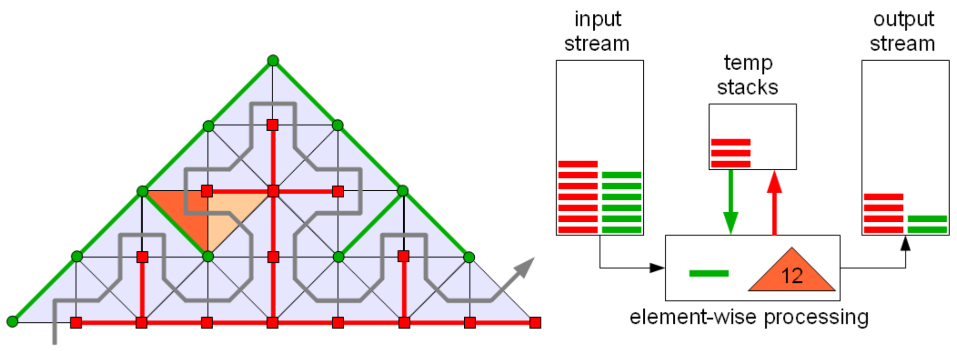
Figure 2.15: Edge-stack system: RED and GREEN streams and stacks.



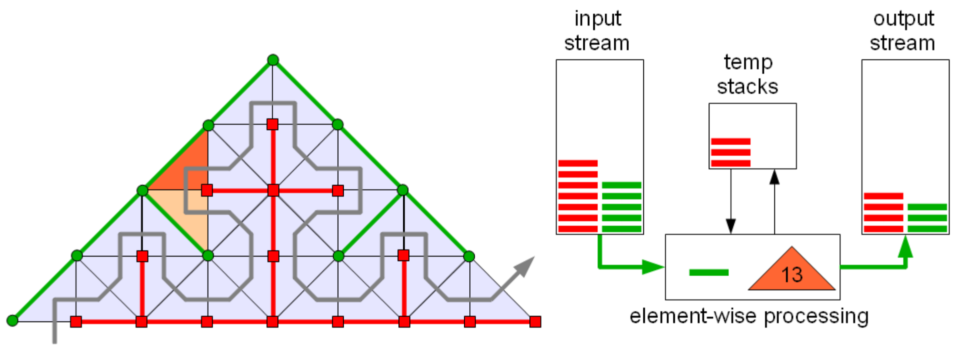
Crossing from 9 to 10. RED edge from 9 to *temp*. RED edge for 10 from *input*.



RED edge from 10 to *temp*. RED edge for 11 from *input*.

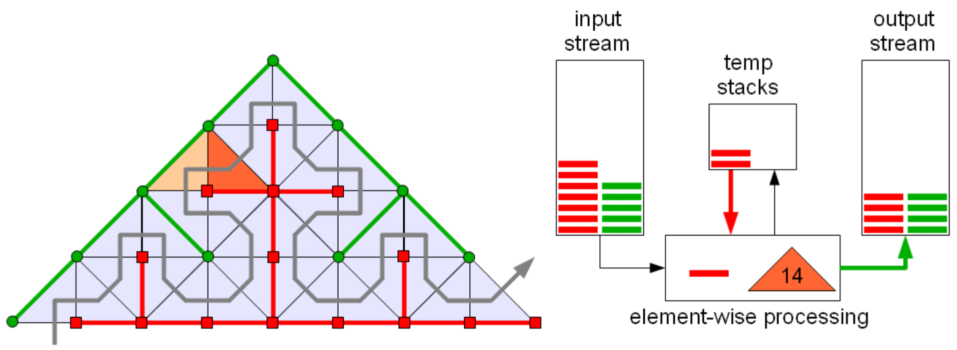


RED from 11 to *temp*. GREEN for 12 from *temp*, placed there earlier from 5.

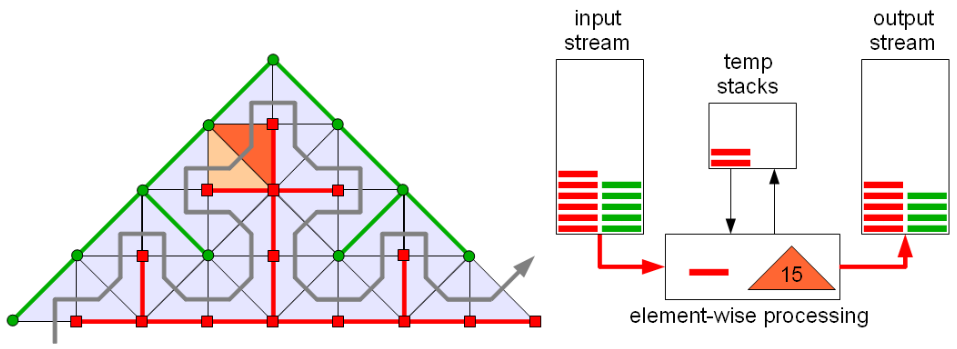


GREEN from 12 to *output*. GREEN boundary for 13 from *input*.

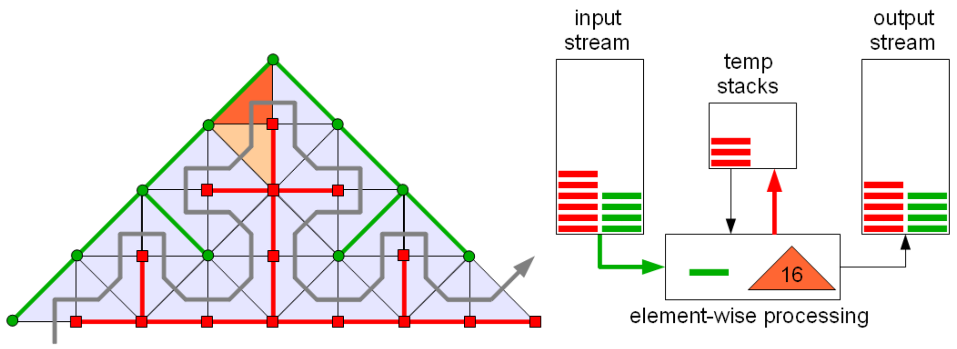
Figure 2.16: Crossing through triangles 10 – 13.



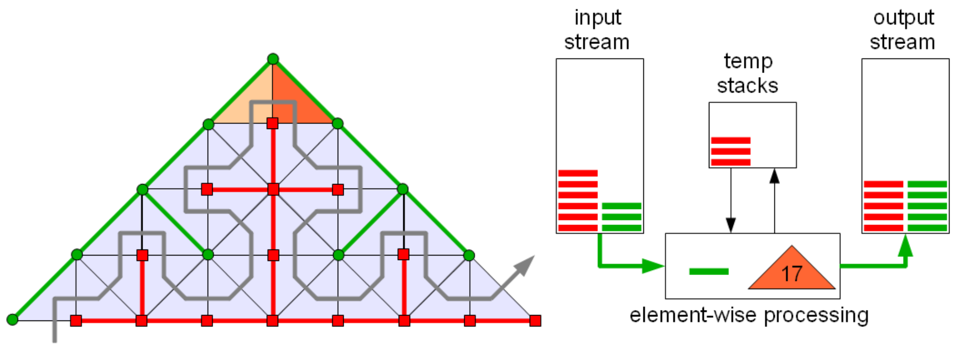
GREEN boundary from 13 directly to *output*. RED for 14 from *temp*.



RED from 14 to *output*. RED for 15 from *input*.



RED from 15 to *temp*. GREEN boundary for 16 from *input*.



GREEN boundary from 16 directly to *output*. GREEN boundary for 17 from *input*.

Figure 2.17: Crossing through triangles 14 – 17.

The empirical upper-bound for the size of the temporary stacks needed during a traversal of *N* triangles is of order *O(log(N))*. An analytical proof could be done as part of future work.

Access pattern to the crossed edges is different and far simpler. They are accessed sequentially one after the other, always two of them at once in each triangle. They are stored in a simple array, and storage is only allocated for adaptive refinement and coarsening information (see chapter 3) that has to persist between grid management traversals. Numerical information exchange between subsequent triangles is achieved with a few temporary variables, and the crossed edges are not used during numerical traversals.

Access pattern to the nodes is stack-like as well, like in the case of color edges, because every node is either red or green. The stack operations on individual triangles in the next section will show some differences between treating the nodes and the color edges. Mainly because a node may need to be accessed anywhere between 1 and 8 times, while inner edges are accessed twice and boundary edges just once.

The separation of red from green entities in the temporary stacks is necessary, otherwise, they would mix up. However, the separation in the input- and output streams is not necessary. In fact, the node-stack system implemented first by Schraufstetter (2006) does not separate the input and output streams into colors. That stack system handled every type of unknown placed logically onto any type of grid element. It had a predefined order of access to unknowns within one triangle cell on its corner nodes, its edges, and the cell interior. All unknowns were placed in the 'first encounter' access order on the input stream, and after traversal they were placed in the 'last encounter' order on the output stream. The only separation occurred on the temporary stacks according to the colors assigned to the grid entities to which a particular unknown was assigned. For details on that system the main reference is Schraufstetter (2006) and partially Bader et al. (2008 and 2010).

I chose to break up that mix of unknowns into three different categories based on what type of grid entity they lay on. *Cell-interior unknowns* are stored in an array, and access to them during a grid traversal is stream-like – in the same Sierpinski order that defines the access to the triangle cells. *Edge-based unknowns* are managed by the edge-stack system which is synchronized to the grid traversal in such a way that each triangle has access to its corresponding edges. *Corner node unknowns* were in yet another node-stack system that works similarly to the edge-stack system. However, the node-stack system is not used in our SWE simulation because the discontinuous Galerkin discretization uses only cell- and edge-based unknowns. Although I describe the detailed synchronization operations for both the edge-stack system and the node-stack system in the next section, only the former is in use and successfully implemented for parallel execution (section 4.3).

This new categorization of unknowns makes possible to treat them separately, and, in our case, to eliminate the corner nodes. This leads to a simpler and more efficient code by eliminating many branch instructions, thus eliminating grid management overhead.

Related research

The collaborating research group that maintains the Peano grid generator studied similar stack-and-stream based systems for adaptive space-filling curve-based rectangular grids. They use *d*-dimensional Peano space-filling curve to build adaptive Cartesian grids; where, in each refinement step, a hypercube is divided in 3^d congruent sub-cubes (see Günther et al. 2006, and Mehl et al. 2006). Illustration of the 2D case with the typical stack-like access is shown in Figure 2.18. Together with this group we found that, for a stack-and-stream based system to work on an adaptive grid, the following requirements are necessary:

- recursive construction of the grid has to match the construction of the corresponding space-filling curve;
- the space-filling curve needs to be edge-connected in 2D or face-connected in 3D to allow information exchange with stacks;
- the "stack-property" needs to hold:
  - in 2D the unknowns placed on any coarse-level edge have to be traversed in opposite direction by the curve on opposite sides;
  - in 3D the unknowns on any coarse-level face have to be traversed by a 2D space-filling curve of the same type and in opposite direction.

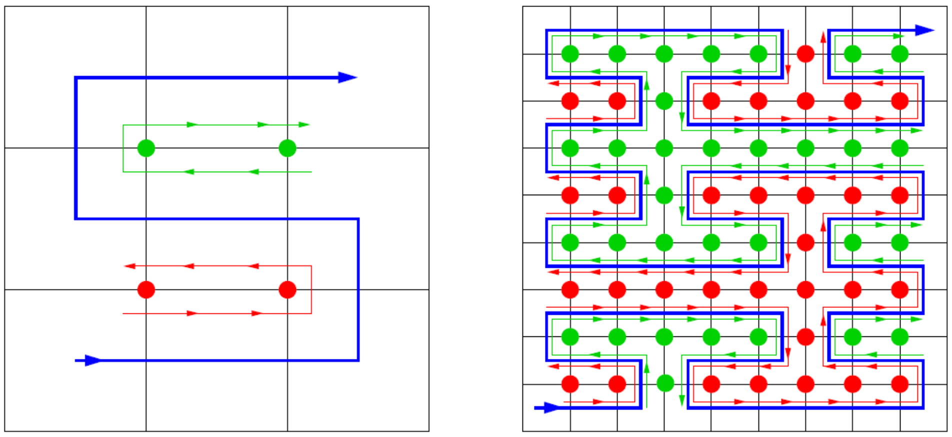


Figure 2.18: Data access suitable for stacks and streams with the Peano curve. Access to the nodes left and right of the curve is stack-like. Access to rectangular cells in the Peano order is stream-like. (Neckel, 2009).

Stack-and-stream based traversals were derived for quadtree-refined grids with 2D Hilbert curve. However, the Lebesgue curve that uses the Morton ordering is not edge-connected, and the stack-property is not satisfied either. In 3D no Hilbert curve was found to satisfy the stack-property. The Peano curve, on the other hand, was demonstrated to work with a stack-and-stream approach for any *d*-dimensional case with only *2·d* temporary stacks (Weinzierl, 2009). Face-connected Sierpinski curves can not be generated in general (see Mitchell, 2005). For a specific 3D tetrahedral bisection scheme stack-and-stream based approach was demonstrated to work in Haug (2006), but that version of the Sierpinski curve has only weak locality properties. For 2D triangular grids the curve is edge-connected, and the performance evaluations in chapter 5 show the memory- and floating-point efficiency; also in comparison to the triangle-only solution with no edge-stack system.

2.2.2. Stack system operations on individual triangles

Cell-based unknowns are stored in an array, and are accessed sequentially as the traversal advances through the triangles. Crossed-edges are stored in another array, and the access to them is intertwined to the triangles. They are only used in grid management traversals during adaptive refinement because the numerical computation manages the information flow through them with a couple of temporary variables. The access to color-edges and nodes is described from two perspectives: cell-based entry and exit (Figure 2.19), and crossing from one cell to the next (Figure 2.20). As stated earlier, nodes are not used in the current work, but the access rules are given for completeness.

In Figure 2.19 a ***green*** triangle of **type *H*** is encountered by the Sierpinski curve. The color-edge is a leg and is *green*, connecting nodes 1 and 2.

Upon entry:

- the **color-edge** is read
    - from the *green* temporary stack, if the triangle is *old, or*
    - from the *green* input stream, if the triangle is *new*.
- **nodes 1 and 3** on the entry-edge are reused from the previous triangle.
- **node 2** is read
    - from the *green* temporary node-stack, if the triangle is *old* (color-edge is *old*, exit-edge is *new*), or

  - from the *green* node-input stream, if the triangle is *new* (color-edge is *new*, exit-edge is *new*).

Upon exit:

- the **color-edge** is written
  - to the *green* output stream, if the triangle is *old*, or
  - to the *green* temporary stack, if the triangle is *new*.
- **nodes 2 and 3** on the exit edge are reused by the next triangle.
- **node 1** is written
  - to the *green* node-output stream, if the triangle is *old* (color-edge is *old*, entry-edge is *old*), or
  - to the *green* temporary node-stack, if the triangle is *new* (color-edge is *new*, entry-edge is *old*).

With regard to the colors of the edges and nodes, and the old/new classification of the triangles and its color edge, similar rules apply for triangles of type *V* and *K*. All these properties are well defined, as discussed earlier in section 2.1.4.

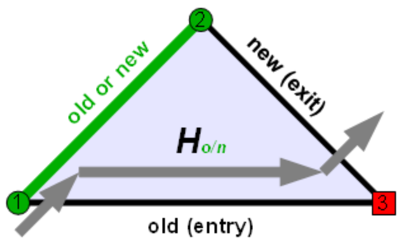


Figure 2.19: Entry and exit of type H.

In the example in Figure 2.20 the curve crosses from a ***red*** cell of **type *V*** to another ***red*** cell of **type *K old***. In the first triangle the color-edge is *green* (hypotenuse), while in the second is *red* (leg).

In triangle *V*:

- the **color-edge** is written
  - to the *green* output stream, if *V* is *old*, or

    - to the *green* temporary stack, if *V* is *new*.
- **node 1** is written
    - to the *green* node-output stream, if *V* is *old* (color-edge is *old*, entry-edge is *old*), or
    - to the green temporary node-stack, if V is new (color-edge is new, entry-edge is *old*).
- **nodes 2 and 3** are transferred across to triangle *K* as nodes 2 and 1, respectively.

In triangle K:

- the **color-edge** is read from the *red* temporary stack (*K* is *old*).
- **nodes 1 and 2** are overtaken from triangle *V*, the former nodes 3 and 2, respectively.
- **node 3** is read from the temporary node-stack (color-edge is *old*, exit-edge is *new*).

Similar rules apply when crossing between different triangle type. These rules are applied after taking into consideration the color-, entry/exit-, and old/new attributes of the individual grid entities.

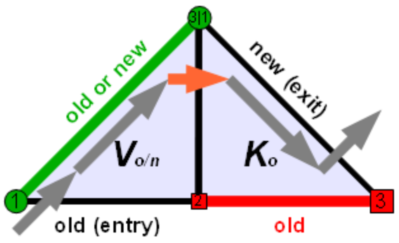


*Figure 2.20: Crossing from triangle type **V** to type **K**.*

There is an exception to the above access rules for the color-edges that lie on a domain boundary or an inter-process boundary in the parallel case. Such a color-edge is always read from the input stream and written directly to the output stream because it will never be encountered again during the current traversal. If the boundary condition was not considered, such an edge would remain on a temporary stack at the end of the traversal. In fact, in the unified stack-system from Schraufstetter (2006) did not consider this boundary exception, and all unknowns on a domain boundary were left on temporary stacks. This did not influence the backward traversal because the unknowns on temp-stacks were on correct reverse order, and they often had to be post-processed after a traversal. At first I changed it because I wanted a traversal-invariant placement of the edges as such that all edges are moved from the input stream to the output stream by a full traversal. However, in the parallel case (see section 4.3), the color-edges on inter-process boundaries have to be processed at once. After a traversal it would be more difficult to determine the correct neighbor partition for each such loose edge. This way, after a traversal, all inter-process boundary edges are in place to be exchanged between neighboring processes, and all other edges were moved to the output stream.

2.2.3. Initial triangulations and the stack-property

The initial triangulation we use for our SWE simulation is shown in Figure 2.21. The color-, entry/exit-, and old/new attributes are chosen such, that the so-called stack-property remains valid on the common edges between the initial triangles as well. This way the edge-stack system will also stay synchronized when switching between initial refined triangle systems throughout the whole traversal.

If the stack-property can not be maintained for a specific initial triangulation, then separate edge-stack systems, which synchronize to their own partitions, could be used. The separate partitions would then communicate through a special edge boundary system similar to the one used in the parallel version. In the parallel version the “inter-process boundaries” are the means of communication between partitions on MPI processes, and they will be described in chapter 4. This special boundary system could easily be altered to solve such “no stack boundaries”, but in this work we only use the initial triangulation of the unit square presented in Figure 2.21.

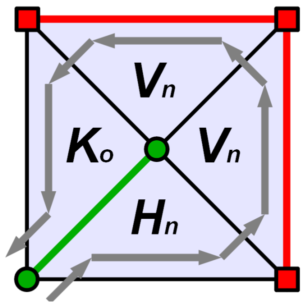


Figure 2.21: Initial triangulation with valid stack property.

2.3. Discontinuous Galerkin method on a Sierpinski grid

With the edge-stack system synchronized to the triangle traversal, it is already possible to describe our piece-wise constant (0th order polynomial) discontinuous Galerkin (DG) method for simulating the shallow water equations. This lowest order DG method is similar to a finite volume type method (Aizinger and Dawson, 2002; Remacle et al. 2006). Unknowns are cell-based, and flux terms are exchanged through edges via the edge-stack system. Dynamic adaptivity is the subject of chapter 3, but because it produces conforming grids without hanging nodes, it fits exactly to the data access pattern of the DG method. Hence, the computations in the numerical traversal are exactly the same for uniform- and for dynamically adaptive conforming grids.

Before the discontinuous Galerkin discretization of the shallow water equations, several other numerical examples were computed on the Sierpinski-based, dynamically adaptive, triangular grid. Using nodal, edge or cell-based unknowns, these were used to demonstrate the possibilities and advantages, or pinpoint the disadvantages of this traversal-based grid processing. Two examples to mention: multigrid preconditioned conjugate gradient for the Poisson equation by Schraufstetter (2006), which we published in Bader et al. (2008); time-dependent heat equation by Radzieowski (2007), in which explicit Euler and Runge-Kutta time integration was implemented. The early version of the shallow water equations with discontinuous Galerkin spatial- and explicit Euler time discretization was implemented in cooperation with Böck (2008), Schwaiger (2008), Demirel (2009), and Obeidat (2009), and we published it in Bader et al. (2010). Parallel version was implemented in cooperation with K. Rahnema. Some of the performance aspects of the Sierpinski traversals, including our version of the discontinuous Galerkin kernel we published in Bader et al. (2012).

2.3.1. Triangle orientation and predefined normal vectors

Most numerical computations need the geometrical information of the triangles such as the normal vectors to the edges, leg-, and hypotenuse size or triangle area. The latter three are computed during recursive bisection, starting from the size of the initial triangle, and are stored for all triangle sizes (all refinement depths).

The normal vectors pointing outwards of each triangle can be categorized as well. Figures 2.22 and 2.23 illustrate the eight occurring triangle orientations and gives the recursive inheritance rules for each of them. Each orientation type has three normal vectors that are precomputed. In addition to the color-, entry/exit-, and old/new properties, the geometric orientation type and the size of the triangles is tracked as well, and is available during grid traversals.

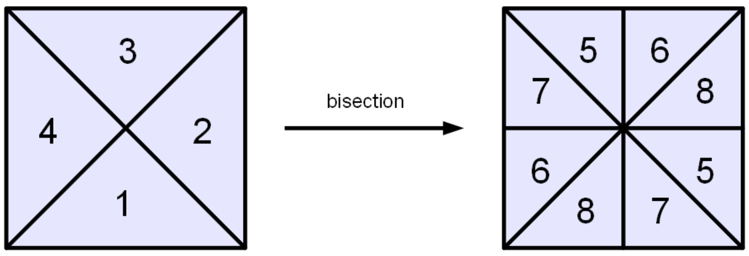


Figure 2.22: Triangle orientations in the Unit Square before and after bisection. Orientation types 1 – 4 will have children of types 5 – 8.

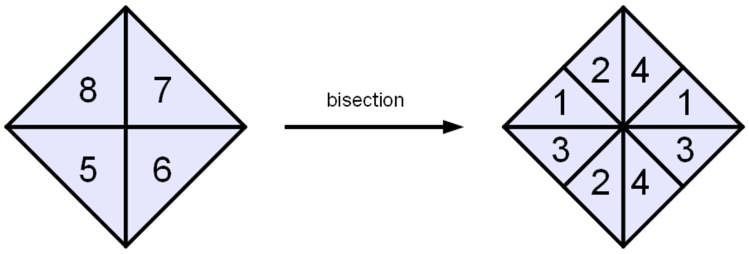


Figure 2.23: Triangle orientation types 5 – 8 have children of orientation type 1 – 4 after bisection.

If we used arbitrary shaped triangles, not just right-angle isosceles ones, then the tracking of the geometry information would need a different solution. A solution might be to store such information in the array together with the cell-based numerical unknowns, but this discussion is out of the scope of this work.

2.3.2. Discontinuous Galerkin discretization of the shallow water equations

This section describes the discretization method we already published in Bader et al. (2010), and serves the purpose of building a computational data access pattern during a grid traversal, which is presented in the next section. For an in-depth reading about the shallow water equations and discontinuous Galerkin discretizations see, for example, Aizinger and Dawson (2002), Behrens (1998), Cockburn et al. (2000), Crouzeix and Raviart (1973), Dawson et al. (2006), Eskilsson and Sherwin (2005), Giraldo (2006), Imamura et al. (2006), and Remacle et al. (2007).

As stated earlier, the version of the shallow water equations that we use neglects viscosity, friction, and Coriolis forces, as shown in Formula 2.2. It is the formulation from Aizinger and Dawson (2002) and from Remacle et al. (2006), where the unknown $\mathrm{v} = (u,\ v)$ is the velocity and ξ represents the sea surface height.

$$\frac{\partial \xi}{\partial t} + \nabla \cdot (\mathbf{v}\xi) = 0,$$
$$\frac{\partial \mathbf{v}}{\partial t} + \mathbf{v} \cdot \nabla \mathbf{v} + g\nabla \xi = 0$$

Formula 2.2: Simplified shallow water equations. Neglected viscosity, friction and Coriolis forces.

Both the full shallow water equations and this simplified version can be rewritten in a vector form presented in Formula 2.3, which makes our approach transferable to the full shallow water equations. Here $\mathrm{u} = (\xi,\ \xi u,\ \xi v)$, $\mathrm{div} = (\nabla\cdot,\ \nabla\cdot,\ \nabla\cdot)$, while vectors $\mathrm{F}(\mathrm{u})$ and r are suitably chosen for the simplified- or the full version of the equations.

$$\frac{\partial \mathbf{u}}{\partial t} + \mathbf{div}\,\mathbf{F}(\mathbf{u}) = \mathbf{r}$$

Formula 2.3: General vector form of the shallow water equations.

For numerical solution we assume that the computational domain is divided by a set of triangular elements that result from our grid generation process described earlier. The discontinuous Galerkin method uses a weak form of the partial differential equations – similar to finite element methods. After applying the divergence theorem to the vector form of the equations, the weak formulation is obtained in Formula 2.4 for each triangular element T_k of the computational grid. Here w denotes the test functions and n is the normal vector to each of the cell boundaries in T_k.

$$\int_{T_k} \frac{\partial}{\partial t}\mathbf{u}\mathbf{w}\,\mathrm{d}\omega - \int_{T_k} \mathbf{F}(\mathbf{u})\nabla\mathbf{w}\,\mathrm{d}\omega + \int_{\partial T_k} \mathbf{F}(\mathbf{u}) \cdot \mathbf{n}\,\mathbf{w}\,\mathrm{d}s = 0$$

Formula 2.4: Weak form of the shallow water equations.

The discontinuous Galerkin method approximates the unknown u locally in each element T_k with a set of polynomial basis functions for each of its components. Continuity of the solution is not enforced on the element boundaries, and the boundary integral can not be computed directly. Instead, it is approximated by a *numerical flux term* similar to those in finite element methods. We obtain Formula 2.5 where $(\mathbf{F}\cdot\mathbf{n})_e^*$ is the numerical flux on an edge of a triangle cell.

$$\int_{T_k} \frac{\partial}{\partial t}\mathbf{u}\mathbf{w}\,\mathrm{d}\omega - \int_{T_k} \mathbf{F}(\mathbf{u})\nabla\mathbf{w}\,\mathrm{d}\omega + \sum_{e\in\partial T_k}\int_e (\mathbf{F}\cdot\mathbf{n})_e^*\,\mathbf{w}\,\mathrm{d}s = 0$$

Formula 2.5: Weak form of the shallow water equations with numerical flux approximation.

In our current implementation we use constant 0th order polynomial functions in each cell, and this leads to a method similar to a finite volume type method (Aizinger and Dawson, 2002; Remacle et al. 2006). In each triangle we use orthogonal polynomials (even if we used linear- or higher order) like in the *Crouzeix-Raviart* element (Crouzeix and Raviart, 1973) in which the unknowns lie on the three edge midpoints. For the ξ component of the vector $\mathbf{u}$ the element-local discontinuous Galerkin discretization of Formula 2.5 can be written in the form of Formula 2.6, where the mass matrix M is diagonal. The matrices C_x and C_y represent the discrete convective transport terms from Formula 2.2. Similar formulations can be obtained for the velocity components of Formula 2.2.

$$\mathcal{M}\frac{\partial}{\partial t}\begin{pmatrix}\xi_0\\ \xi_1\\ \xi_2\end{pmatrix} - \mathcal{C}_x\begin{pmatrix}\xi_0\\ \xi_1\\ \xi_2\end{pmatrix} - \mathcal{C}_y\begin{pmatrix}\xi_0\\ \xi_1\\ \xi_2\end{pmatrix} + \begin{pmatrix}|e_0|(\mathbf{F}\cdot\mathbf{n})_{e_0}^*\\ |e_1|(\mathbf{F}\cdot\mathbf{n})_{e_1}^*\\ |e_2|(\mathbf{F}\cdot\mathbf{n})_{e_2}^*\end{pmatrix} = 0$$

Formula 2.6: Element-local discretization for the ξ component. Formulation is similar for the velocity components.

The flux terms need to be computed for all three edges. Since the solution is discontinuous, the flux computation averages the contribution on both sides of any edge. We use the *Lax-Friedrichs* flux as given in Formula 2.7, where contributions result from both interior- and exterior values ξ^{i-} and ξ^{i+}, and α is suitably chosen.

$$(\mathbf{F}\cdot\mathbf{n})^*_{e_i}(\xi^{i+},\xi^{i-}) = \frac{1}{2}\left(\mathbf{F}(\xi^{i+})\cdot\mathbf{n}_{\mathrm{e_i}} + \mathbf{F}(\xi^{i-})\cdot\mathbf{n}_{\mathrm{e_i}}\right) - \alpha\left(\xi^{i+} - \xi^{i-}\right)$$

Formula 2.7: Lax-Friedrichs flux averages the interior (ξ^{i-}) and exterior (ξ^{i+}) contributions.

All terms involving system matrices M, C_x, and C_y from the discrete formulation in Formula 2.6 are element-local and can be computed from the unknowns and geometry information in the local triangle cell alone. Only the flux computation needs exterior flux contribution from all neighboring triangle elements (Formula 2.8).

$$(\mathbf{F}\cdot\mathbf{n})^*_{e_i}(\xi^{i+},\xi^{i-}) = \underbrace{\frac{1}{2}\mathbf{F}(\xi^{i-})\cdot\mathbf{n}_{\mathrm{e_i}} + \alpha\xi^{i-}}_{=:\,(\mathbf{F}\cdot\mathbf{n})^{*-}_{e_i}} + \underbrace{\frac{1}{2}\mathbf{F}(\xi^{i+})\cdot\mathbf{n}_{\mathrm{e_i}} - \alpha\xi^{i+}}_{=:\,(\mathbf{F}\cdot\mathbf{n})^{*+}_{e_i}}$$

Formula 2.8: Interior- and exterior flux contributions on an edge.

2.3.3. Data access pattern in the discontinuous Galerkin traversal

The discontinuous Galerkin discretization leads to a system of ordinary differential equations in Formula 2.9, with $\mathrm{u_h}$ as the unknown vector containing all $(\xi, \xi u, \xi v)$ components in all triangles T_k. Matrices M, C_x, and C_y are the global, block-structured counterparts of the same element matrices from Formula 2.6, and $\mathcal{F}^{*-}$ and $\mathcal{F}^{*+}$ are the interior- and exterior flux terms in global matrix notation.

$$\mathcal{M}\frac{\partial}{\partial t}\mathbf{u}_h = \mathcal{C}_x\mathbf{u}_h + \mathcal{C}_y\mathbf{u}_h - \mathcal{F}^{*-}\mathbf{u}_h - \mathcal{F}^{*+}\mathbf{u}_h$$

Formula 2.9: Discrete system of ordinary differential equations.

An explicit Euler scheme leads to the computation in Formula 2.10 in each time step.

$$\mathbf{u}_h^{(n+1)} = \mathbf{u}_h^{(n)} + \tau \mathcal{M}^{-1} \left(\mathcal{C}_x \mathbf{u}_h^{(n)} + \mathcal{C}_y \mathbf{u}_h^{(n)} - \mathcal{F}^{*-} \mathbf{u}_h^{(n)} - \mathcal{F}^{*+} \mathbf{u}_h^{(n)} \right)$$

Formula 2.10: Explicit Euler time-stepping.

The goal is to update all unknowns in all triangles once in each grid traversal. That way one Euler time-step is equivalent to one traversal. In each triangle the flux values are needed from the previous-, next-, and far neighbor before the cell-based unknowns are updated to represent the next time-step. For the update '*n+1*', all the flow values use data in each triangle from traversal '*n*'. In the example in Figure 2.24 the yellow triangle obtains the flow value from the entry-edge directly because that value was already used in the previous triangle. The flux through the exit-edge can be computed by simply accessing the local unknowns in the yellow- and in the next triangle because these are stored in consecutive locations in the numerical array. This is possible because both the current- and the next triangle still holds values computed in traversal '*n*'.

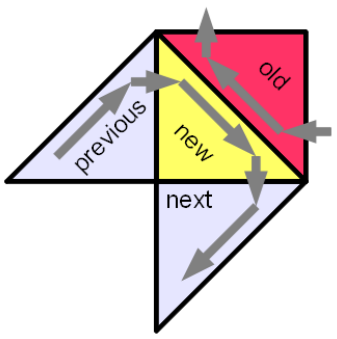


Figure 2.24: Information exchange during DG traversal. The yellow triangle is encountered first in the traversal, the red triangle is encountered later.

The flux computation through the color-edge poses the only difficulty. There are two possible solutions: the *triangle-only* traversal, in which there is direct access to the color-neighbor via a pointer or an index; and the *edge-stack system* traversal, in which we pass information through the color-edges. Both solutions are implemented, but the *triangle-only* is less efficient, though easy to implement, and serves only for benchmark purposes.

In the *triangle-only* solution, when the traversal arrives to the yellow triangle (of type *new*), it accesses both the yellow- and the red triangle and computes the flux through the color-edge. This flux is then written to temporary storage in the red triangle's numerical data because, by the time the traversal reaches it, the yellow triangle's unknowns are already updated to represent the state after time-step '*n+1*'. Therefore, the flux computation on the color-edge is done in every triangle of type *new*, and the stored value is reused in triangles of type *old*.

In the *edge-stack system* solution, when the traversal reaches the yellow triangle of type *new*, there is no way of directly accessing anything from the red triangle. Information can only be propagated forward in the traversal direction via the *edge-stack system*. The partial flow from the yellow triangle could reach the red triangle later in the traversal, but the partial flow from the red triangle can definitely not reach the yellow triangle. The only way to process the yellow triangle correctly is if the corresponding flux value from the red triangle is already available on the color-edge.

Placing the correct flux value onto the color-edge is only possible in the previous traversal after the cell-based unknowns were updated from time-step '*n-1*' to time-step '*n*'. This way, in the current update '*n+1*' we take the correct flux value from the color-edge, and, together with the fluxes from the entry- and exit-edge, we update the cell-based unknowns to represent time-step '*n+1*'. After this update we need to provide the interior flux contribution on the color-edge for use in the next time-step. In a *new* triangle this partial flux is computed and stored on the color-edge separately from the flux that is still in use by the current traversal and placed on a temporary color-stack. In an *old* triangle the partial flux is computed and summed up with the part from the other side residing on the temporary color-stack. The total flux value is then stored in the color-edge, replacing the flux for the current update, and is placed on the output stack ready for use in the next traversal '*n+2'.*

With this computation pattern we compute one Euler step with a single Sierpinski traversal, and information flow is achieved with the edge-stack system. The flux exchange via edges is done similarly when the grid is adaptive and conforming. Conforming grids do not have so-called hanging nodes, and an edge always connects at most two triangles, not more (see chapter 3). Furthermore, no global discretization matrix is assembled; all computations are strictly element-local. The applications using the *Peano* grid management system work in a similar fashion, while, for example, with *amatos* or *dendro* a global system of equations has to be constructed and solved.

If the algorithm were extended to perform Runge-Kutta time integration, each slope would be computed analogously to the Euler time-step in one grid traversal. Additional memory would need to be allocated in the triangle cells for each slope and each unknown, to store intermediate slopes, while the memory used for flux terms would be allocated once and reused for all slopes.

3. Sierpinski-based Adaptive Mesh Refinement and Coarsening

For tracking the wave fronts accurately in the oceanic wave propagation, it is desirable to have smaller triangle cells in surroundings where the differences in water height are considerably large and to have bigger triangles in regions with small- or no difference. Since the wave propagation is time-dependent, there is no way to predict its course in advance, and the adaptive mesh refinement and coarsening has to happen dynamically. Refinement-, coarsening-, or no change requests are posted for each triangle during a discontinuous Galerkin traversal, based on certain criteria like the local change of the unknowns. Adaptation follows in several traversals, which will be described in the course of this chapter.

Our adaptive mesh refinement will always produce so-called *conforming triangular grids* in which no hanging nodes are allowed. In such a grid any edge lies exactly in between two adjacent triangles, but not more. Thus, the data access pattern during numerical computation described in the previous chapter – data exchange between neighbor triangles exclusively through edges – will be preserved. This consistency check requires additional traversals after the initial refinement requests.

During the actual dissection and joining of the triangles the numerical data of each triangle cell persists, and interpolation/restriction is required for refinement and coarsening operations. If it was used, the node-stack system would be persistent too; nodes with interpolated values would be added or removed from the respective stacks and streams (for details on this see Vigh (2007)). The edge-stack system, on the other hand, is rebuilt from scratch during a *re-initialization traversal* on the new grid. A persistent edge-system is not required for our discontinuous Galerkin traversals, and the rebuilding traversal is also used to search for inter-process boundaries in the parallel implementation (see chapter 4). The initial flow values that are needed on the color-edges before the discontinuous Galerkin traversal may start again, and are calculated in this re-initialization traversal too.

Adaptive mesh refinement and coarsening must be performed after each time-step for fully adaptive grids. However, as will be presented in the performance measurements of chapter 5, the cost of an adaptive step in terms of computing time is about 3 or 4 Euler time-steps. We believe that in the future, when using higher order spatial discretization with perhaps fourth- or higher order Runge-Kutta time-stepping, this ratio may drop significantly, perhaps to 1 or even below. This assumption is based on the fact that one computation traversal needs to be executed for each Runge-Kutta slope. This effectively increases computing time for one time-step, while the adaptation workload increases only slightly due to an increase in numerical data volume.

3.1. Refinement scenarios of a single triangle cell

Various refinement scenarios of a single triangle cell are illustrated in Figures 3.1 through 3.4. The corresponding partial binary refinement trees are depicted on the right-hand-side of each picture. In Figure 3.1 the original triangle is bisected through the hypotenuse with the blue segment. In Figures 3.2 and 3.3 in addition the left- and the right child, respectively, is (recursively) bisected through its own hypotenuse. In Figure 3.4 both children are bisected, delivering a fully refined sub-grid and sub-tree of depth two.

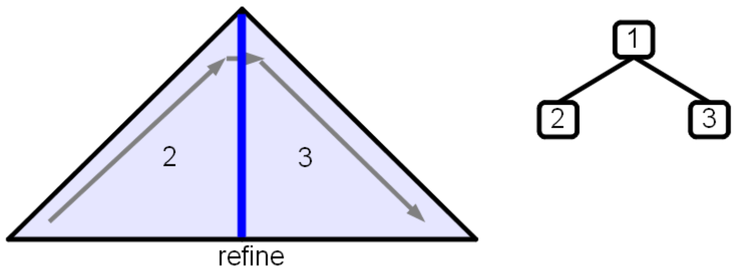


Figure 3.1: Single refinement – triangle is bisected through the hypotenuse.

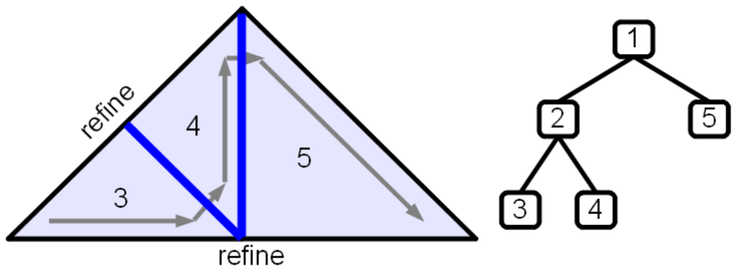


Figure 3.2: Multiple refinement – left child is bisected as well.

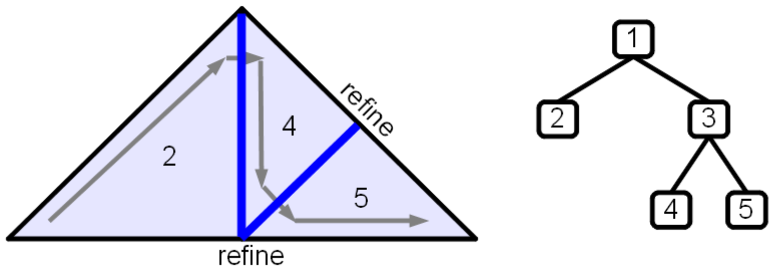


Figure 3.3: Multiple refinement – right child is refined as well.

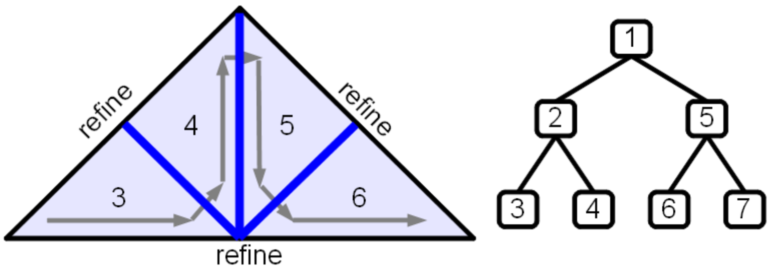


Figure 3.4: Maximum refinement – both children are refined, delivering four grandchildren.

While in theory it is possible to introduce further refinement on the grandchild triangles, we limit the refinement in one adaptive step to 2 depth levels per triangle cell. These are required to solve the hanging node problem described next.

3.2. Hanging node problem and refinement cascade

A hanging node occurs when a triangle on one side of an edge is refined while the triangle on the other side is left unchanged. Figure 3.5 illustrates such a scenario, in which the common edge is the hypotenuse of both neighbor triangles. If such hanging nodes were allowed, the original hypotenuse would connect three adjacent triangles. In addition, special numerical treatment would be necessary in the discontinuous Galerkin traversal or, in fact, in any kind of numerical traversal. The edge-stack system, as described in the previous chapter, would need to be modified to work correctly and would become much more complicated.

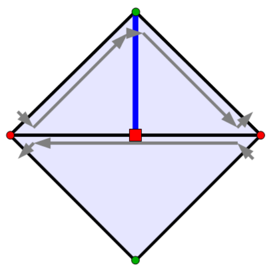


Figure 3.5: Hanging node problem – triangle on top was refined while the bottom triangle remained unchanged.

If the original refinement request from the triangle on the top is transmitted through the connecting edge to the lower triangle, then a forced bisection eliminates the hanging node, as depicted in Figure 3.6. Therefore, the refinement requests are stored in the edges in form of a *please_refine* flag.

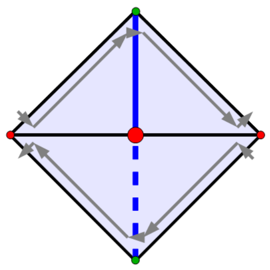


Figure 3.6: Edge-based refinement propagation. Hypotenuse-hypotenuse neighbor relationship.

In Figure 3.7, the smaller triangle on the top-left is bisected by the blue line. The common edge connecting the original triangles is a hypotenuse in one, and a leg in the other triangle. The refinement, when propagated through to the big triangle, has to trigger two bisections along the blue dashed lines. Therefore, the hypotenuse of the big triangle has to be marked for refinement too.

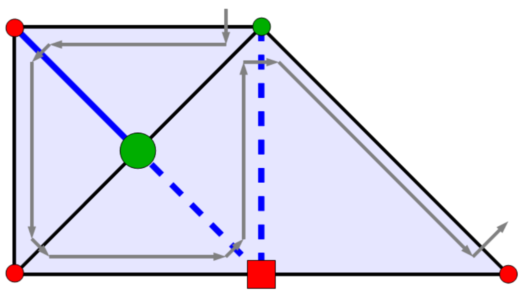


Figure 3.7: Edge-based refinement propagation. Hypotenuse-leg neighbor relationship. Refinement is enforced on the hypotenuse of the big triangle before the hanging node is eliminated on the common edge.

It may happen that a refinement through a hypotenuse-leg neighbor relationship triggers a cascade of refinements, like in the example in Figure 3.8. In this example, the cascade involves triangles in both forward- and backward direction of the Sierpinski curve. Because of the bidirectional nature of such a cascade, following it directly would be computationally too expensive. Instead, to propagate the cascade in both directions, so-called *consistency traversals* will be executed at least twice after the initial refinement requests were posted. However, forced refinements can, in turn, trigger even more refinements. The consistency traversals need to be executed until there are no more additional forced refinements at all. The operation in each triangle is to mark the hypotenuse for refinement (set the *please_refine* flag) if one of the legs is already marked. Stevenson (2008) showed that all additional bisections to retain conformity inflate the final number of triangles by at most a constant factor. This means that the *consistency traversals* will eventually terminate, and in my experience they do generally stop after 2 to 4 traversals. After adaptation, the new grid corresponds to the grid that would be generated by **amatos**, with the same initial grid and same refinement requests.

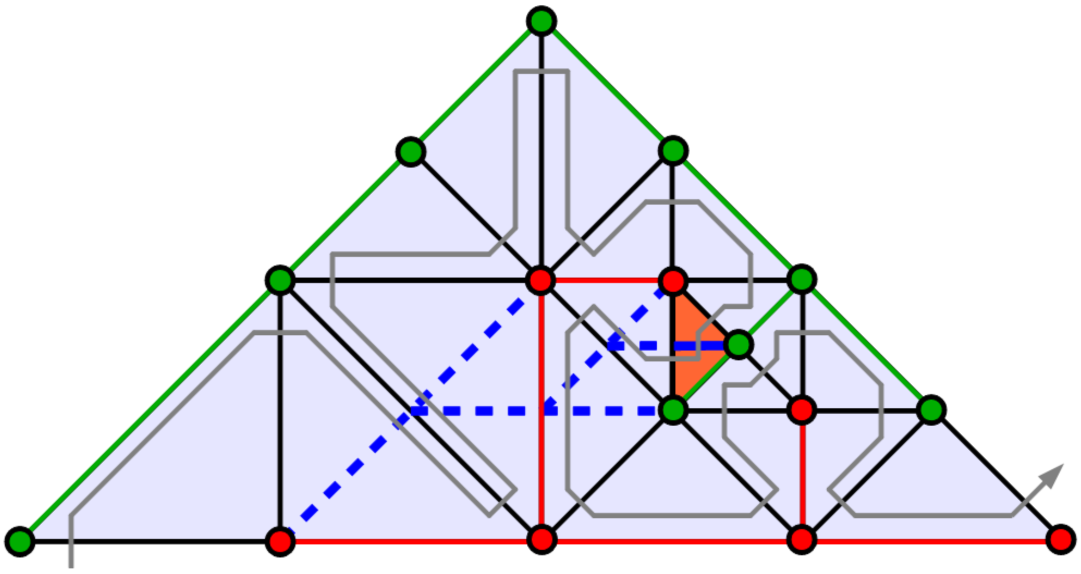


Figure 3.8: Refinement cascade – refinement requested in the orange triangle triggers a cascade of refinements in other triangles (blue dashed lines).

For adaptive Cartesian grids constructed with quadtrees or octrees (e.g. **Dendro** by Sundar et al, (2007a and 2007b) or **p4est** by Burstedde et al, (2011)) or with space-trees based on recursive trisection (e.g. **Peano** by Weinzierl (2009) and Neckel (2009)), there are always some form of hanging nodes. The conformity condition holds if the difference in depth level in the refinement tree of any two geometrically neighboring grid cells is not more than 1. That way, all hanging nodes are so-called *level-1 hanging nodes*.

Figure 3.9 shows an adaptive grid refined with the Peano concept, in which the *level-1* hanging nodes are marked with circles. The *level-2* hanging nodes marked with dark squares trigger additional refinements. The refinement information is propagated with the help of a node-stack system (see Weinzierl, 2009).

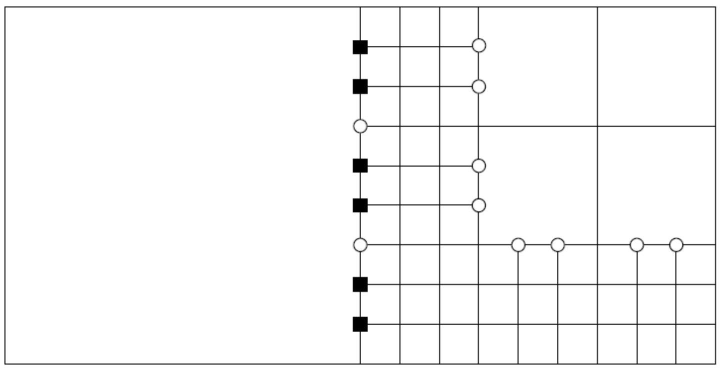


Figure 3.9: Peano level-1 (circles) and level-2 (dark squares) hanging nodes in 2D. (Neckel, 2009).

In Figure 3.10 the ripple-effect is shown in the adaptive grid refined with quad-trees, as a result of the so-called 2:1 balancing when using Dendro. In Dendro locational codes of the octants are used, the ripple-effect is tracked and conformity is enforced within a single grid traversal.

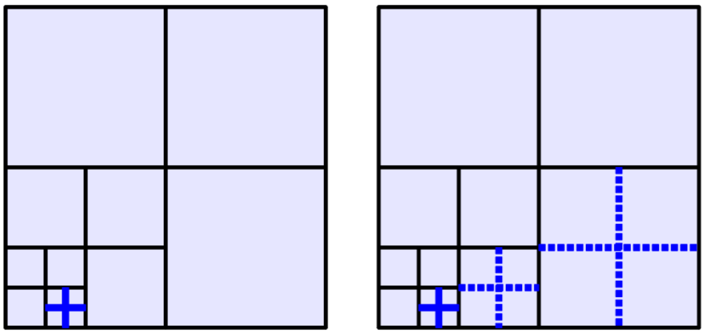


Figure 3.10: 2:1 balancing for a quadtree grid in 2D. Ripple effect: refinement on the left (blue lines) triggers additional refinements (blue dotted lines).

3.3. Coarsening

Compared to refinements or no-change requests, coarsening, or merging two sibling triangle cells has low priority. Coarsening does not cause any cascades, however, care must be taken to avoid hanging nodes. In Figure 3.11, when the upper two triangles (1 and 2) are merged, it is necessary to merge the lower two as well (3 and 4). All four triangles have to agree on the coarsening action, in which case in the new grid the inner edges II and IV disappear and edges I and III unite to form the hypotenuse of the merged triangles.

Coarsening has to be aborted if any of the outer edges V to VIII are marked for refinement. Considering a case when, for example, edge VII is marked for refinement, in order to cancel the coarsening operation and keep the grid conforming, it has to be known in both the upper- and in the lower triangle couple. During the adaptation traversal in which the actual coarsening happens (along with all the refinements as well), in order to be merged triangles 1 and 2 will be accessed simultaneously. The restriction from edge VII in triangle 3 has to arrive to at least one of them, and this can happen through edge III. For this purpose, during the *consistency traversals*, leg edges are marked as *no_coarsening* in triangles in which the hypotenuse is marked for refinement or in triangles in which *no-change* request was posted. This way, if a restriction is in effect, that information is communicated to at least one edge that would form the new hypotenuse (edge I or III), and the coarsening operation will not be performed.

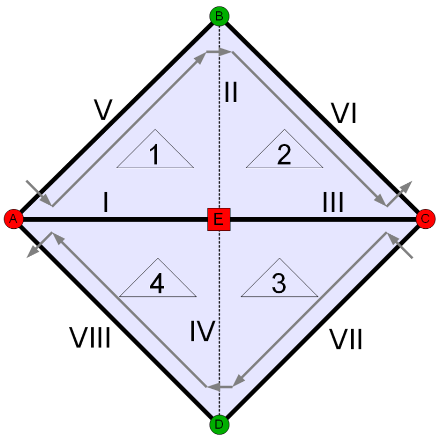


Figure 3.11: Coarsening agreement – all internal edges (I to IV) must agree on coarsening. Any restriction needs to be propagated to at least one of the edges that would form the new hypotenuse (I or III).

Another correctness check during coarsening is that the two subsequent triangle cells to be merged have to be siblings in their refinement tree. Figure 3.12 illustrates a correct coarsening scenario, in which the sibling triangles are correctly merged into their respective parents. The grid is correctly represented by its adjusted refinement tree and the linear ordering of the leaf triangles is still the one imposed by the Sierpinski curve.

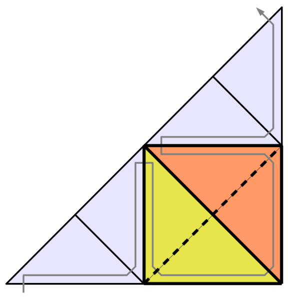


Figure 3.12: Correct coarsening – merging siblings is the correct inverse operation of a bisection. The merged triangles keep the linear Sierpinski order.

Example of what can happen, if non-siblings are merged, is in Figure 3.13. In the left picture, the merged triangles do not align into the Sierpinski order. On the right picture, the merged triangles form a square.

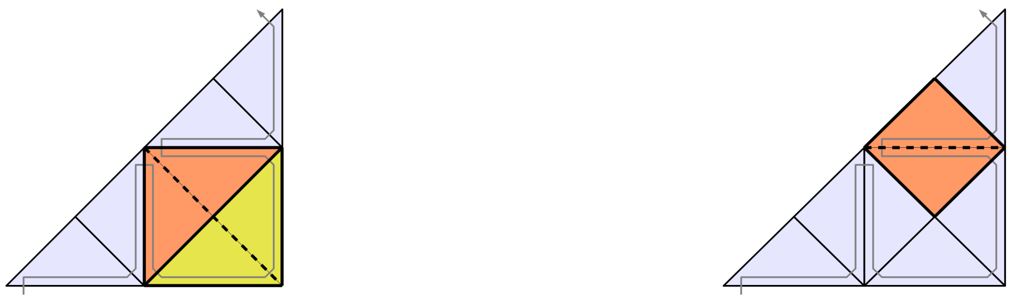


Figure 3.13: Incorrect coarsening – merging non-siblings leads to erroneous triangle position (left) or to a square (right).

In a recursive traversal it is trivial to check the sibling relationship of two subsequent leaf triangle cells. The implementation of every type of traversal was recursive at first, and the refinement tree(s) had to be updated as well, in addition to the leaf triangle arrays containing numerical unknowns. The first adaptation traversal was implemented in Vigh (2007), and could perform only refinements. Coarsening was added by me later, with the implicit sibling-checking offered by the nature of the recursive traversals. Recently I switched completely to loop-based traversals for obvious performance considerations discussed in chapters 4 and 5. In the loop implementation the refinement tree is not used anymore, and I use binary tree locational codes for the leaf triangles to identify siblings.

3.4. Full step of adaptive refinement and coarsening

The full adaptive step, after which the discontinuous Galerkin traversals can be resumed on the new grid, consists of the several management traversals illustrated in Figure 3.14. It begins with one that transfers the triangle-based refinement-, coarsening- and no-change requests into the edge-stack system. This step will in the future be merged into a discontinuous Galerkin traversal, which would reduce the total execution time of a full adaptive step by about 5%.

The consistency traversals are executed multiple times, and spread the refinement and coarsening information around the grid. The cycle is stopped when the last consistency traversal did not change anything anymore. At this point all refinement cascades and coarsening scenarios were resolved, and actual adaptation can start without additional consistency checks (except sibling checking for coarsening).

The recursive adaptation traversal bisects or merges the triangles by updating the refinement tree and builds the new leaf triangle array with the numerical unknowns. Interpolation and restriction is performed where necessary. At the end of the traversal the edge-stack system becomes obsolete and is discarded.

The re-initialization traversal builds a new edge-stack system on the new and conforming grid, and simultaneously computes and stores initial flow values on the color-edges. These flow values on color-edges are needed by the first discontinuous Galerkin traversal, as already shown in the previous chapter, section 2.3.3, “Data access pattern in the discontinuous Galerkin traversal”.

The computation time of a recursive full adaptive step is equivalent of about five or six discontinuous Galerkin traversals performing the Euler time-stepping. Switching to loop-based traversals reduces this ratio to three or four. The parallel full adaptive step with exclusively loop-based traversals looks slightly different, and will be presented in the next chapter, “Parallelization and Performance Optimization”. Details of the performance gain induced by loop-based traversals will be shown in chapter 5, “Performance Analysis”.

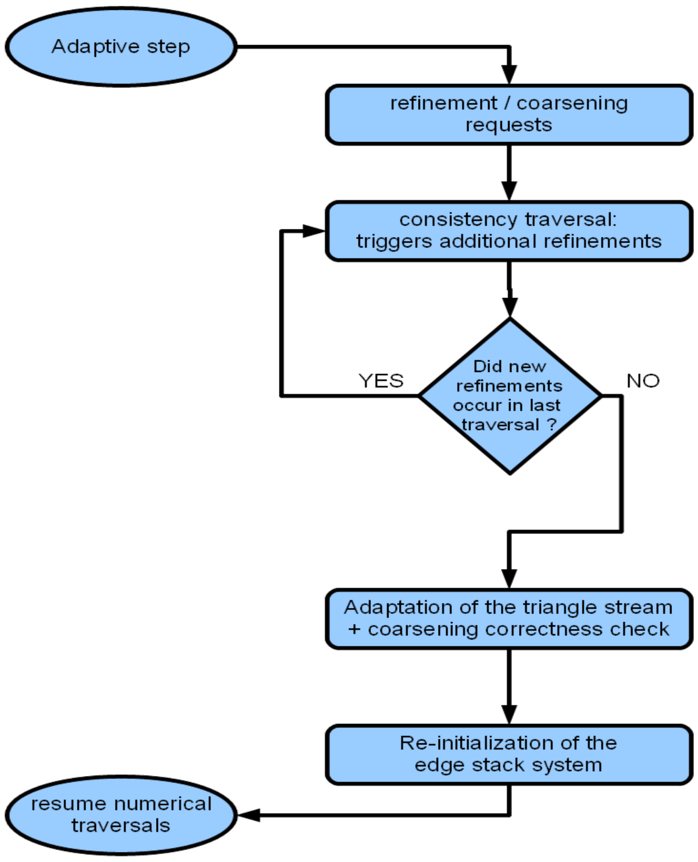

Adaptive step
refinement / coarsening requests
consistency traversal: triggers additional refinements
Did new refinements occur in last traversal ?
YES
NO
Adaptation of the triangle stream + coarsening correctness check
Re-initialization of the edge stack system
resume numerical traversals

Figure 3.14: Full adaptive step consists of several management traversals. Consistency traversals ensure that the new triangle system will be conforming, with no hanging nodes. Numerical unknowns will be adjusted to the new grid by interpolation and/or restriction. Edge-stack system is recreated to match the new grid. Initial flow values are computed and stored in the color-edges, ready for use in the next discontinuous Galerkin traversal.

4. Parallelization and Performance Optimization

Parallelization of the traversals is done using Message Passing Interface (MPI), which is optimized for distributed memory systems. Partitioning is based on space-filling curve methods, given the built-in Sierpinski curve. Loop-based traversals are introduced not only to optimize execution times for single traversals, but also to facilitate easy traversal-cutting – starting and stopping a traversal at any point in the grid – when processing single parallel partitions. The explicit binary tree is not updated, and locational codes of the triangle cells are used instead, when needed, during coarsening consistency checks. The information exchange between neighboring partitions is exclusively edge-based, as it is required by our discontinuous Galerkin traversal, and is embedded into the existing edge-system. Global reduction operations are performed in between consistency traversals during adaptive mesh refinement, to check the global stopping condition. Load-balancing redistributes the new grid equally among the processors in an "*MPI_AllToAllV*" fashion, and the parallel edge-system is reinitialized.

4.1. Domain decomposition with the Sierpinski curve

The Sierpinski space-filling curve offers a straightforward approach to partitioning. Triangles are sequentially ordered by the curve and, by cutting the curve into equal parts, each partition gets the same amount of triangles. This method is applied both for the initial grid creation and for load-balancing after the adaptive mesh refinement. For example, in Figure 4.1 a uniform grid of 32 triangles is divided into three equal parts along the Sierpinski curve, marked with different colors. Furthermore, Figure 4.2 shows two snapshots of an expanding semi-circle, along which the grid was refined. Repartitioning with load-balancing shows that the partitions can radically change their geometric positions. This domain decomposition method is also used by the parallel version of *amatos* – called *pamatos* (Behrens and Zimmermann, 2000).

Quality of the partitions is relevant in two aspects: *equal amount of computational load*, and *low surface-to-volume ratio*. Partitions obtained by space-filling curve approaches are by construction optimal in the first aspect. Due to the good locality properties of the Sierpinski curve, our partitions are also well-behaved regarding the second aspect. Griebel and Zumbusch (1998) showed that partitions created with space-filling curves in general have low surface-to-volume ratio, which is desirable for reducing the amount of inter-process communication.

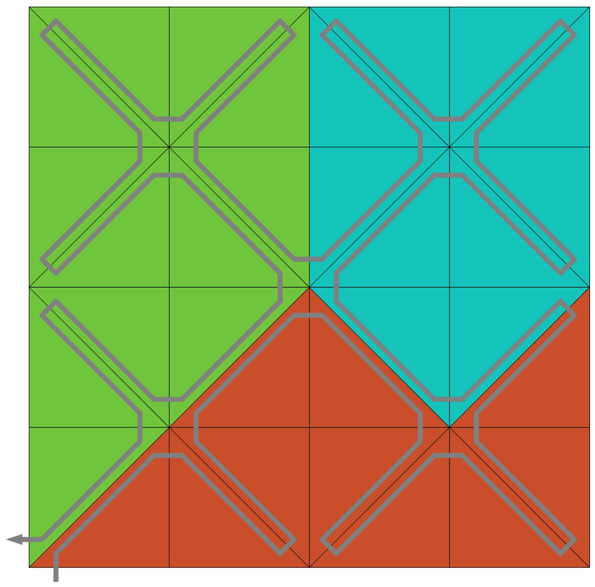


Figure 4.1: Uniform domain of 32 triangles decomposed into 3 sub-domains along the Sierpinski curve. Red and blue domains have 10 triangles each, and the green domain gets the remaining 12.

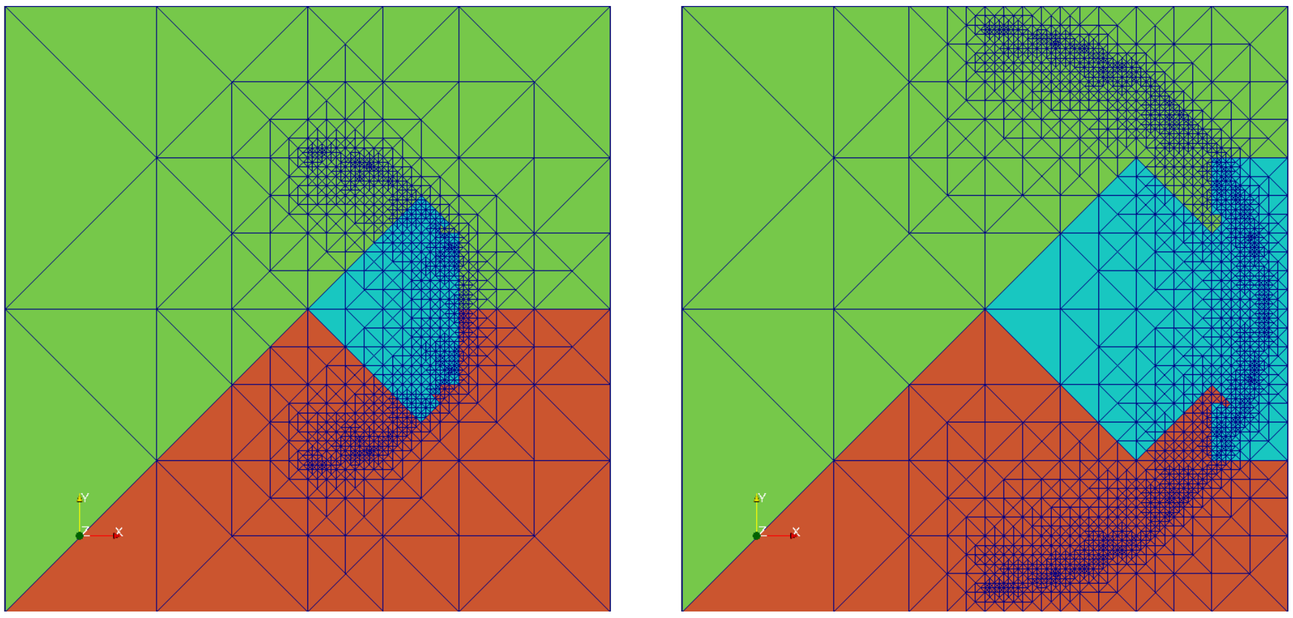


Figure 4.2: Expanding semi-circle. Grid adaptively refined along the semi-circle, divided into 3 sub-domains. Sierpinski curve is omitted for clarity. Load-balancing causes the partitions to radically change their geometric position.

For a parallel discontinuous Galerkin traversal we need to traverse single partitions efficiently (in section 4.2), and need to extend the edge-stack system to deal with inter-process boundaries (in section 4.3). Adaptive mesh refinement in parallel will be discussed subsequently in section 4.4.

4.2. Loop-based vs. recursive traversals

As we have seen so far, the Sierpinski-based grid management algorithms are inherently recursive. The Sierpinski-specific triangle attributes from chapter 2 are recursively computed according to the given rules and formulas. However, the effective work performed on the numerical unknowns is done only on the leaves of the supporting binary tree. In fact, the numerical unknowns of the leaf triangles are stored in a simple linear array. The idea is to have the triangle attributes of the leaves saved in an array too, of the same length as the numerical array, and then a simultaneous loop over these two arrays replaces the recursive traversal routine.

The inspiration of loop-based traversals came from *Dendro*, when I was on a research stay at Georgia Tech. *Dendro* uses tree structures only to generate the leaf cells in the desired order and then it discards the tree structure completely (Sundar et al. 2007a). Another grid generator called *p4est* from Burstedde et al. (2011), which generates rectangular grids using forests of octrees and the space-filling Morton curve (like *Dendro*), also adopted the storage of leaf octants in an array. This linearization of the leaf elements is – according to my knowledge – not present in *amatos* or *Peano*.

The first attempts to linearize the leaf triangles rendered the loop-based Sierpinski traversal three times faster than its recursive counterpart. This impressive improvement faded somewhat in the course of the code evolution, but it is still present with 25% to 50% computational speed gain, depending on the traversal type. This gain comes with a cost of around 16 bytes per triangle, in which the triangle attributes are stored. However, this is only 30% of the total memory usage per leaf triangle, and is a fixed cost. Should the numerical scheme change, only the amount of memory allocated for numerical unknowns will have to be adjusted. The first performance measurements comparing the loop-based vs. the recursive traversal we published in Bader et al. (2012), and will be presented in chapter 5, “Performance Analysis”.

Dropping the binary tree and using loop-based traversals has advantages in the parallel implementation, too. Traversal cutting is simplified, the linear arrays contain only the partition-local triangle information. In a recursive traversal a whole binary tree would be traversed, including parts that contain foreign elements. In contrast to our linearized solution, *Peano* adjusts the space-trees in each parallel partition in such way, that parts which do not contain local elements are coarsened to the maximum extent possible.

Also load-balancing becomes simpler with the loop-based traversals, since it involves relocation of array structures only, in an *MPI_All-to-All* fashion (details in section 4.4).

4.3. Parallel edge system and parallel traversals

Figure 4.3 shows an example of inter-process boundary edges (thick lines) between two partitions. By construction of the partitions the crossed-edges are always inner edges, except for the first- and the last crossed-edge. The first- and last crossed-edge can be either inter-process boundary or geometric domain boundary. All the other inter-process boundaries are color-edges.

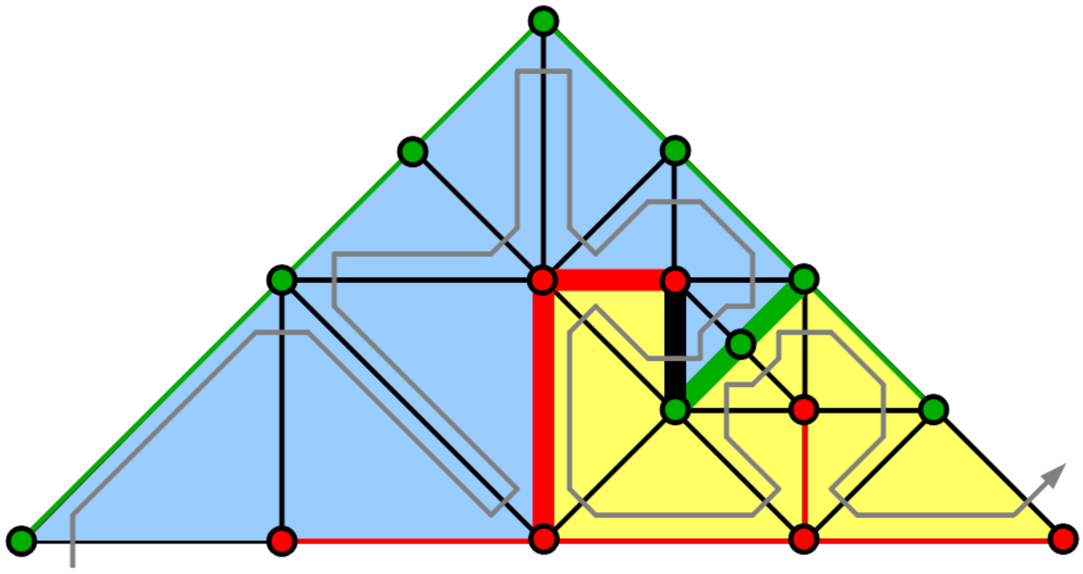


Figure 4.3: Inter-process boundaries (thick lines): red-, green- and crossed-edge boundaries between blue and yellow partitions.

Before starting a discontinuous Galerkin traversal, the inter-process boundary edges have to contain the flux values in advance, similarly to the regular color-edges. Computing the correct flux value happens in two stages. First, during a traversal the process-interior contribution is calculated and stored. Secondly, at the end of the traversal, the stored values are exchanged between partitions and summed up to form the full flux value.

During the discontinuous Galerkin traversal, the inter-process boundary edges are accessed once. Their stored flux is used to update the cell-based unknowns, and the new partial flux value is computed and written to them. At the end of the traversal, the inter-process edges are exchanged with their respective partition neighbors and flux values are summed up in preparation for the next traversal.

The access pattern to the inter-process boundaries of the same color during a traversal is stream-like. Two separate arrays will be allocated for red- and green colors for each neighbor of the partition in a process, as shown in Figure 4.4. Two arrays per neighbor are necessary because the access orders in the two communicating neighbors differ. The size of these arrays is of the order of the *temp-stacks* for the standard color-edges. The first and last crossed-edge is treated at the beginning of the traversal, and stored on one of the color process boundary arrays.

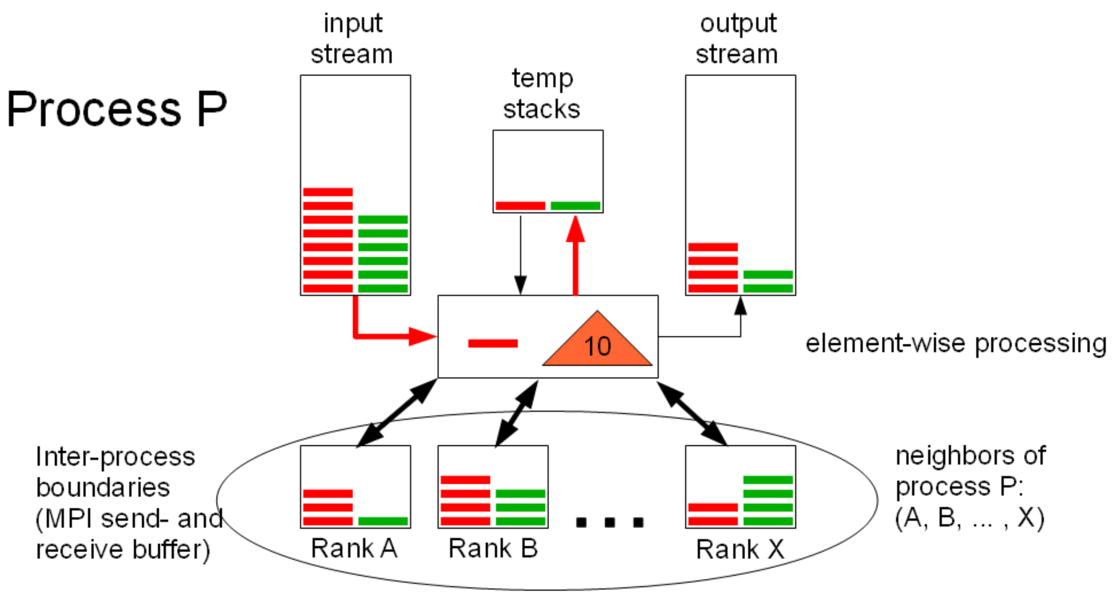

Process P
input stream
temp stacks
output stream
10
element-wise processing
Inter-process boundaries (MPI send- and receive buffer)
Rank A
Rank B
. . .
Rank X
neighbors of process P: (A, B, ... , X)

Figure 4.4: Parallel edge-stack system. Inter-process boundary arrays used with simple blocking MPI communication.

During a traversal, the stack-system operations – described in chapter 2, section 2.2.2 – are slightly modified. If the color-edge of a triangle cell is inter-process boundary, then it is read from its dedicated array, and, after usage, it is written back to it. The fact that whether the color-edge of a triangle is partition-local or a process boundary, is stored in the triangle array introduced for loop-based traversals, together with the rest of the Sierpinski attributes. It simply contains the rank of the neighbor on the other side of the color-edge.

In the initial construction of the parallel edge-stack system, in addition to constructing the standard color-edges, a search for the inter-process boundaries is performed. Since there is absolutely no topological information available at first, a global traversal is executed on the whole domain. Because of the global nature, this algorithm does not scale in parallel, and will not be reused at the end of an adaptive step or after load-balancing.

During this initial construction traversal, when a color-edge is created in a triangle of type *new*, the global index of the triangle is written to it and pushed onto the temporary color-stack. When the color-edge is encountered for the second time in the other triangle of type *old*, the two triangle indexes are compared. If both triangles belong to the partition owned by the process, the edge is an internal color-edge and is stored accordingly. If both triangles belong to foreign partitions, the edge is discarded. If one of the triangles belongs to the own process while the other is foreign, an inter-process boundary edge was found. The edge is written to the respective process boundary array and in the own triangle – for which also the Sierpinski attributes are saved – the rank of the foreign neighbor is stored. At the end of the global search, the parallel edge-system is ready for use in parallel loop-based traversals on its own partition.

The current implementation of the parallel edge-system supports non-blocking MPI communication as illustrated in Figure 4.5. It uses a receive- and a send buffer in addition to a working copy of the inter-process boundary arrays.

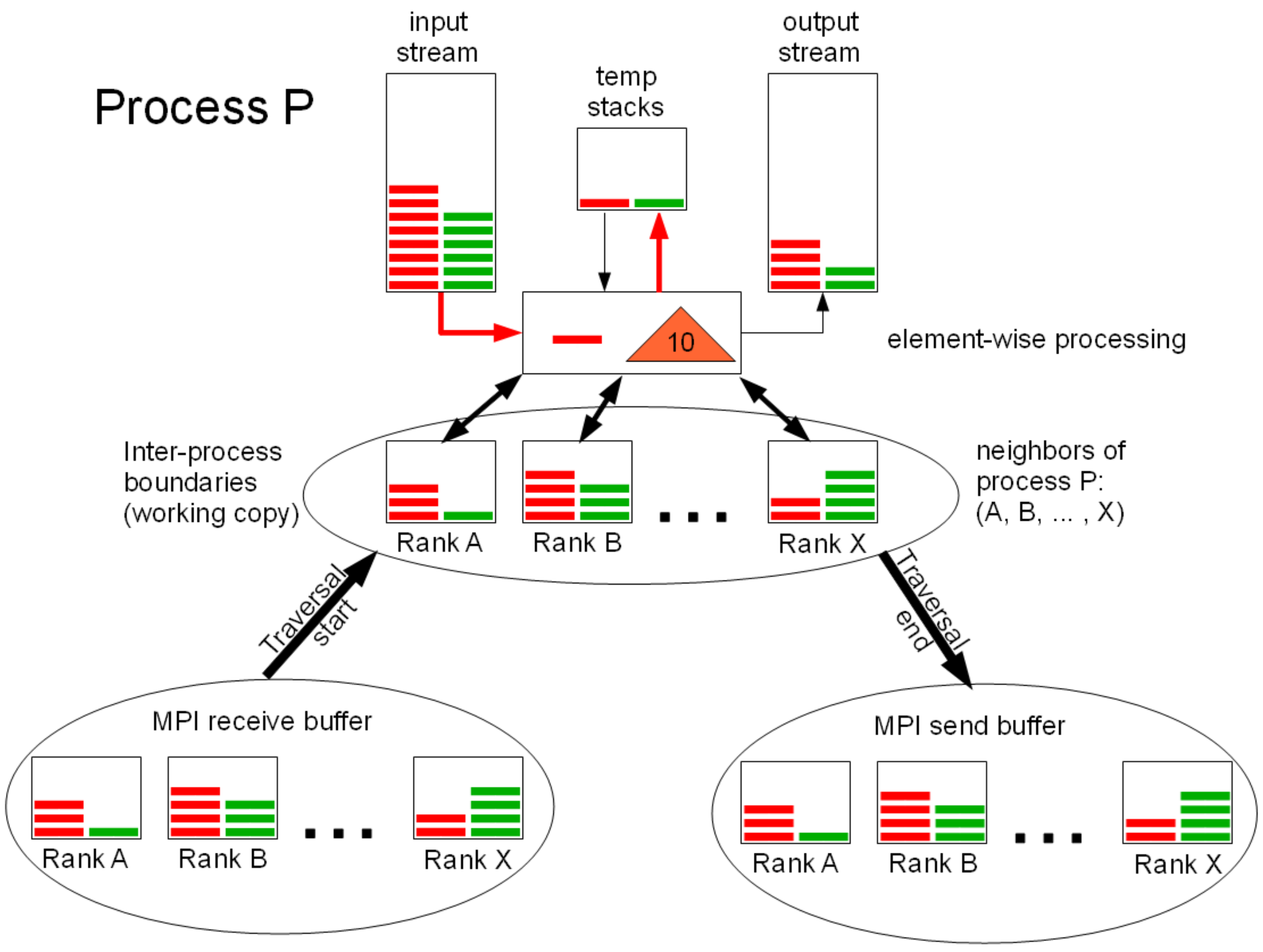


Figure 4.5: Parallel edge-stack system. Inter-process boundaries with non-blocking MPI communication. Additional send- and receive buffers needed in addition to the working copy.

4.4. Adaptive refinement and coarsening in parallel

Parallelization of the traversals in the adaptive mesh refinement step presented in chapter 3 includes a few challenges. The flow-chart in Figure 4.6 reiterates the sequential adaptive step and highlights the problems for the parallel case.

The first issue is the stopping condition in the consistency check loop, which has to be global. Otherwise it may happen that one partition stops the loop early, while another performs some additional traversals. Traversal directions could become opposite in different partitions, which would cause a mix-up in the inter-process boundary orders. Furthermore, a refinement request might be propagated onto an inter-process boundary that would not be picked up by the neighbor partition which has stopped the loop early. This would lead to hanging nodes and an inconsistent grid. Therefore, the solution is to perform a global "*MPI_AllReduce*" operation on the stopping condition at the end of the loop, to avoid all the inconsistencies mentioned above.

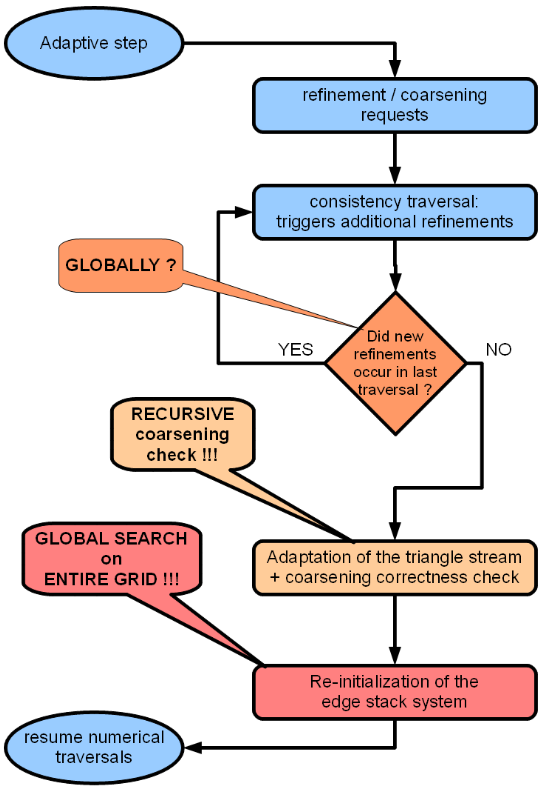

Adaptive step
refinement / coarsening requests
consistency traversal: triggers additional refinements
GLOBALLY ?
YES
Did new refinements occur in last traversal ?
NO
RECURSIVE coarsening check !!!
GLOBAL SEARCH on ENTIRE GRID !!!
Adaptation of the triangle stream + coarsening correctness check
Re-initialization of the edge stack system
resume numerical traversals

Figure 4.6: Parallelization issues during an adaptive step. Stopping condition for consistency loop is local and needs to become global. Coarsening correctness check (only siblings allowed to merge) is inherently recursive and based on binary tree traversal, which should not be used. Re-initialization of the inter-process boundaries is a global search on the entire grid, which hinders parallel speed-up.

The other two issues are more complex, and will therefore be discussed more in depth in the following subsections. Recursive coarsening check for siblings has to be replaced by a loop-based algorithm, presented in the next section. Re-initialization of the parallel edge-stack system, as presented before, is a global search on the entire grid. Not only that the entire grid is not available in every processor, but also parallel speed-up is out of the question. A partition-local solution will be presented in section 4.4.2 involving neighbor index prediction before the actual refinement and coarsening happens. Finally the full adaptive step in parallel with load-balancing is the subject of section 4.4.3.

4.4.1. Loop-based coarsening check

Merging two triangles is only allowed if they are siblings in the binary tree. Merging incorrect triangles leads to an inconsistent grid, as shown earlier in chapter 3, Figure 3.13. In order to check the relationship between two subsequent triangles, we need their binary tree location codes. Storage requirement is one additional integer per triangle cell. In Figure 4.7 an example of a binary tree is given with location codes and the sibling detection rules.

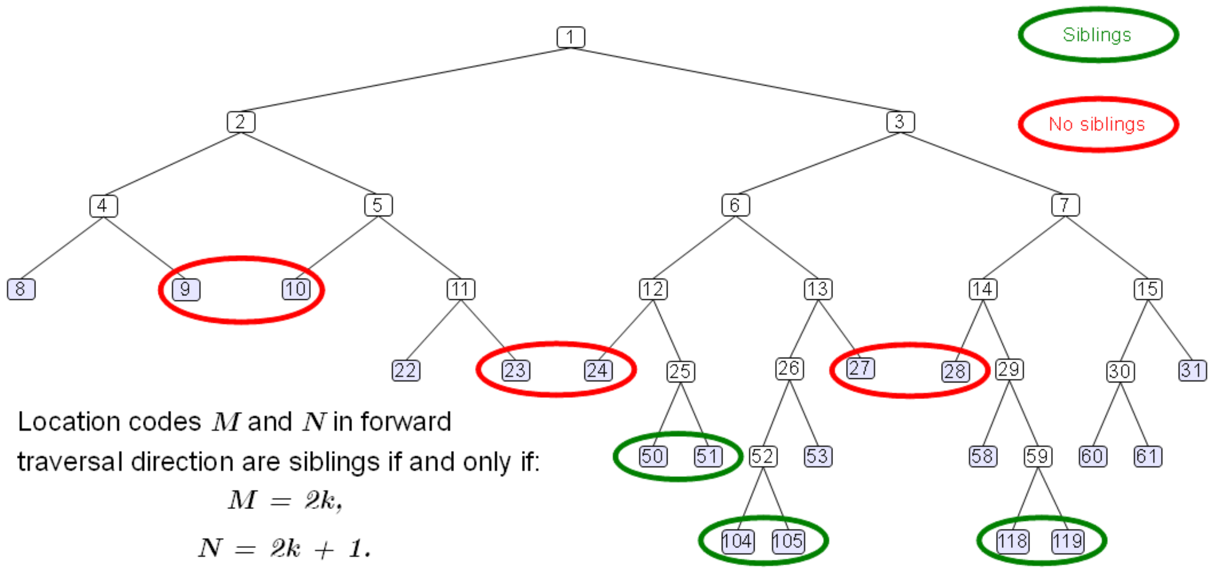


Figure 4.7: Binary tree location codes. Sibling detection based on location codes.

As a result, an additional traversal is introduced to delete those coarsening requests that would cause inconsistencies. In addition, this *coarsening_check* traversal computes the new size of the local partition in the new grid, by summing up the amount of triangles based on all refinement- and coarsening requests. This new size will be valid after the mesh refinement and before load-balancing. At the end of the traversal all processes communicate the new size to all other processes and each process can adjust its new start- and end indexes to the global values. These values will be used later during the actual refinement and coarsening traversal.

Another issue is the coarsening of two siblings that lie on two different partitions. This inter-process coarsening would cause additional communication between subsequent processes during adaptive refinement and I chose to avoid it. The initial partitioning of the grid must ensure that every pair of siblings belongs to the same process. Siblings produced by refinement are on the same process before any load-balancing. Siblings may become separated only as a result of coarsening. Therefore, after the adaptive mesh refinement and the optional load-balancing, all separated siblings will be moved to only one of the two processes that hold them. This sibling exchange happens just before the re-initialization of the parallel edge-stack system.

4.4.2. Neighbor index prediction

Re-initialization of the parallel edge-system without any topological information is basically a global search for color-edges that are inter-process boundaries. As described in section 4.3, those color-edges are searched for, which connect triangles of two different partitions. Furthermore, only those edges are relevant which connect the partition-local triangles to foreign partitions.

The idea is to generate the global index of the new color-edge neighbor for each triangle in the new grid. If every triangle knows the index of its color-neighbor, then it also knows whether that neighbor lies in the same partition or in a foreign one. This way the re-initialization traversal can be performed on the local partition, generating the local color-edges and also all the inter-process boundary edges. As a consequence, a global search is avoided and parallel speed-up can be achieved.

Generating the color-edge neighbor index for all the new triangles requires a simulation of the adaptive mesh refinement. During this simulation traversal a local counter is incremented and stored in the edges of the existing edge-system. When an edge is encountered for the first time, it will store the current counter value for transfer to its other side. If the edge is refined, it will transfer two index values. When the edge is encountered the second time, it contains the index value(s) of the triangle(s) from its other side, and can match them to the current value of the counter. Thus, the future color-neighbor relationship is determined and adjusted to the global indexes obtained from the previous *coarsening_check* traversal, and saved into a dedicated array.

At the end of the simulation traversal, the existing inter-process boundary edges are exchanged and the color-neighbor relationships are resolved for those triangles that are/will be on process boundaries too.

After this neighbor index prediction the actual adaptive mesh refinement may start. The precomputed neighbor indexes are written to the new triangles, in the location where, during regular traversals, the color-edge type or the inter-process neighbor rank is stored. This way, with or without load-balancing, the parallel edge-system can be re-initialized with a simple partition-local traversal.

4.4.3. Full adaptive step with load-balancing

Figure 4.8 shows the updated flow-chart of the parallel adaptive step. In the *consistency loop* an *MPI_AllReduce* operation is inserted for the global stopping condition. The *coarsening check* is loop-based, uses binary tree location codes and computes new partition lengths. *New neighbor prediction* simulates the adaptive mesh refinement to compute color-edge neighbors in the new grid. The actual *adaptive mesh refinement* creates the new triangles. The numerical unknowns are interpolated. The Sierpinski attributes are computed according to the rules presented in chapter 2, and the color neighbor indexes are placed where the type of the color-edge (internal or process boundary rank) will be stored later. The binary tree location codes are also recomputed accordingly.

At the end of the *adaptive mesh refinement* traversal the old edge-system is discarded, and optionally load-balancing may be executed. Load-balancing has to balance three arrays in calls to *MPI_AllToAllV*, in which a process specifies the indexes and the amount of elements to send and to receive to- and from all other processes. It may happen that a process exchanges elements not only with the previous- or the next partition in the Sierpinski curve direction, but with partitions that are further away. This is the case when a massive amount of refinements happen in one partition, but few or none in the next, and the geometric position of the partitions changes extremely. An example of extreme changes in partitions is illustrated in Figure 4.9, in which snapshots of an oceanic wave propagation on 7 partitions are shown.

The three arrays that have to be exchanged are the numerical unknowns, the Sierpinski attributes and the binary tree location codes. Since these arrays are all linked to the leaf triangle cells, some or all of them could be merged together, and there would be less *MPI_AllToAllV* calls. These performance aspects were not checked, since the *parallel adaptive mesh refinement* is development of last minute.

After the optional load-balancing each process checks its first- and last triangle's binary tree location code, and exchanges it with the previous- and next partition. If siblings are separated by a crossed-edge process boundary, then one of those process pairs gets both triangles, in order to avoid inter-process coarsening in the next adaptive step.

Finally the parallel edge-system is rebuilt in a single partition-local traversal. Inter-process boundaries are found based on the neighbor indexes stored in the adaptive refinement traversal. Initial flux values are computed and stored on color-edges and on inter-process boundaries and numerical traversals may be resumed on the new grid.

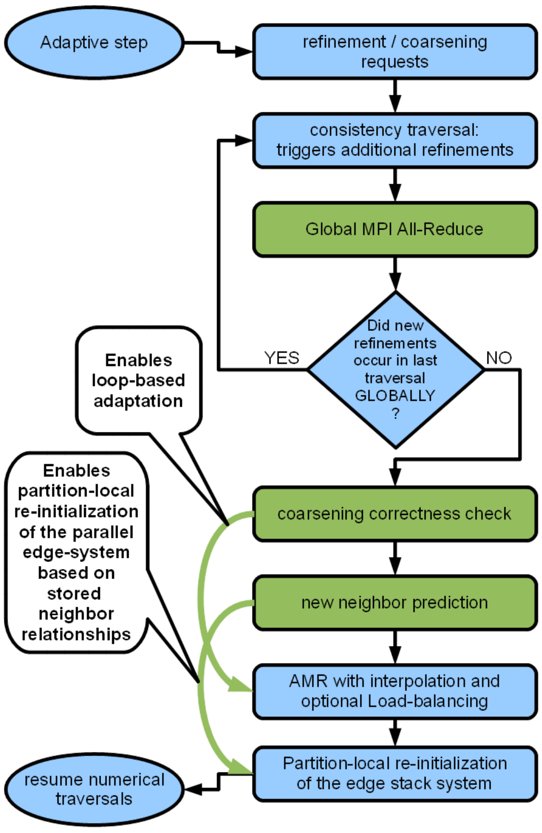

Adaptive step
refinement / coarsening requests
consistency traversal: triggers additional refinements
Global MPI All-Reduce
Did new refinements occur in last traversal GLOBALLY ?
YES
NO
Enables loop-based adaptation
Enables partition-local re-initialization of the parallel edge-system based on stored neighbor relationships
coarsening correctness check
new neighbor prediction
AMR with interpolation and optional Load-balancing
Partition-local re-initialization of the edge stack system
resume numerical traversals

Figure 4.8: Parallel loop-based full adaptive step. Arrays assigned to leaf triangles are persistent. Edge-system with inter-process boundaries is rebuilt after adaptive mesh refinement and load-balancing. No binary tree used or updated. No node information is considered.

Our adaptive mesh refinement resembles those of Behrens and Zimmermann (2000) and Mitchell (2007), but in our case the only persistent data is assigned to leaf triangles. We do not update, merge or exchange explicit binary tree data and no nodal data. The edge-based data is not persistent either, but is recomputed. Performance aspects of our parallel adaptive step will be given in the next chapter, “Performance Analysis”.

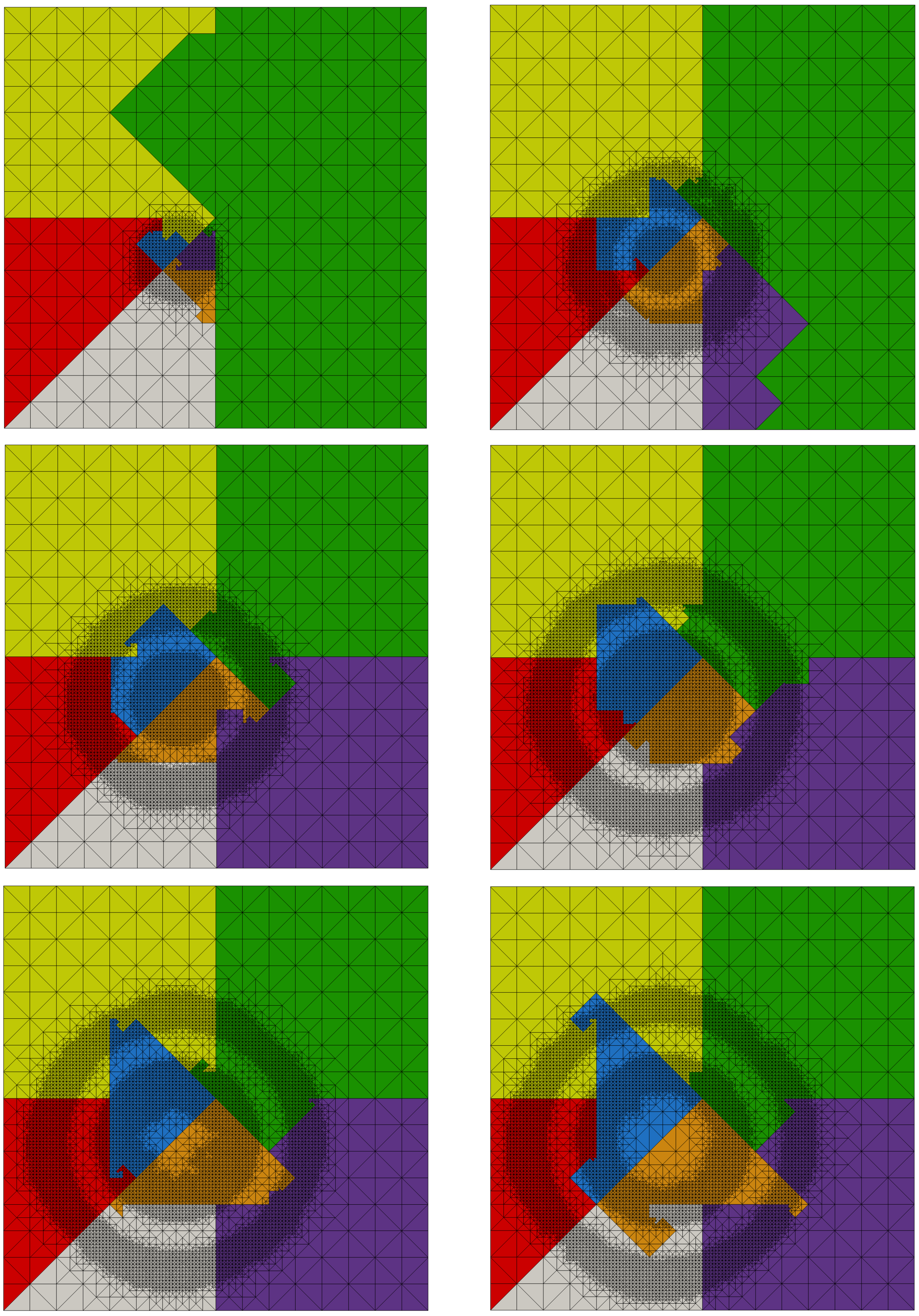


Figure 4.9: Wave-propagation with load-balancing on 7 partitions, top view. Partitions change geometric positions. (Joint work with K. Rahnema, 2011).

5. Performance Analysis

The first part of this chapter presents serial performance aspects of the Sierpinski-based traversals. In anticipation of higher-order discontinuous Galerkin discretization and of Runge-Kutta type time integration, traversals with varying amount of artificial floating-point operations and with varying amount of numerical unknowns per triangle cell were executed. The performance is compared to the current discontinuous Galerkin traversal that solves the shallow water equations. Furthermore, the performance difference between loop-based and recursive traversals is also given. A comparison between edge-stack system- and triangle-only implementation of the numerical traversals shows why the former is preferable to the latter. Last but not least, the execution time of the full adaptive step – with the distinct traversal types it contains – is compared to the benchmark discontinuous Galerkin traversal.

The second part analyses the parallel performance in terms of parallel speed-up. The discontinuous Galerkin traversal results are presented at first without any adaptive mesh refinement. Consistency traversals are tested as well, because these contain no floating-point operations. The full adaptive mesh refinement step – which performs grid manipulations but no FLOPs – is measured separately. The complete solution of the shallow water equations with adaptive mesh refinement after every time step is presented at the end (joint work with K. Rahnema).

5.1. Serial performance analysis

When executing traversals using a single processor, we are interested in *floating-point performance*, *memory throughput speed* and *memory usage per triangle*. Depending on the traversal type and the amount of memory needed per triangle, a traversal may tend to be computation-bound or memory-bound. *Triangle processing speed* is a more common measure and is sometimes used to compare different traversal types. For *memory usage per triangle*, only the bytes actually touched (read or written) by a traversal are considered, but the full amount of memory needed per triangle for the program to run may differ significantly. This is the case when, for example, adaptive mesh refinement needs to duplicate certain arrays, but a standard traversal uses only a single instance of them.

The serial performance measurements were performed on two platforms, with the characteristics summarized in Table 5.1. The information is an excerpt from the official Intel product description website. The platform called *E7400* is a desktop system, and while *T7700* is a laptop system.

Platform name	**E7400**	**T7700**
Full product name of CPU	Intel Core 2 Duo Processor E7400 (3M Cache, 2.80 GHz, 1066 MHz FSB)	Intel Core 2 Duo Processor T7700 (4M Cache, 2.40 GHz, 800 MHz FSB)
Code name	Wolfdale	Merom
Launch date	Q1'08	Q2'07
# of Cores	2	2
Clock speed	2.8 GHz	2.4 GHz
Cache	3 MB L2 cache	4 MB L2 cache
System bus	1066 MHz	800 MHz
Lithography	45 nm	65 nm
Main memory (RAM)	~3GB	~4GB
Operating system	Linux	Windows XP
Compiler	Intel Fortran Compiler 11.0	Intel Fortran Compiler 10.0.025

Table 5.1: Platforms used for performance testing. E7400 is a desktop system and T7700 is a laptop.

In order to get a general overview about the capabilities of the hardware, we introduce a reference benchmark that is relevant for the class of problem we are solving. A simple LINPACK benchmark for measuring the MFLOPS rate in our case makes little sense, since it computes a dense LU decomposition, which is a different class of problem. Our discontinuous Galerkin traversal resembles a matrix-vector multiplication with a sparse band matrix. For this purpose we implemented a simple block-diagonal matrix-vector multiplication with a very large matrix, and the performance of it we consider as the best that can be expected from this class of problem.

Measuring the memory bandwidth of a platform can be done by looping over huge vectors and performing simple operations on their elements. For example, a DAXPY operation updates each element of one vector using two floating point operations as follows: `A(i) = A(i) + q*B(i)`. In Bader et al. (2012) we showed with a cache analysis that such a vector update has the same order of compulsory cache misses as our traversals. McCalpin (1995 and 1991-2007) measures memory bandwidth in a similar fashion, but uses more types of operations during vector updates. However, since the block-diagonal matrix-vector multiplication is also a long vector update, with 7 floating-point operations per element, I chose to use it as reference also for memory throughput speed.

Table 5.2 lists the reference performance of the block-diagonal matrix-vector product and also for DAXPY vector update. LINPACK values were measured with “Intel Math Kernel Library 10.3.2.005” and the theoretical peak value is computed by multiplying the clock frequency with the number of pipelines. One can observe that the highlighted block-diagonal matrix-vector product achieves roughly a quarter MFLOPS of the LINPACK, and about 65-70% MB/sec of the DAXPY vector update.

	E7400		**T7700**	
	MFLOPS	**MB/sec**	**MFLOPS**	**MB/sec**
Block-diagonal matrix-vector product	2,476	2,361	1,970	1,878
DAXPY	447	3,413	373	2,847
LINPACK	9,840		7,880	
Theoretical peak	2.8 GHz x 4 = 11,200 MFLOPS		2.4 GHz x 4 = 9,600 MFLOPS	

Table 5.2: Reference performance of the two platforms. Block-diagonal matrix-vector product (gray) will be of relevance in our further discussion.

5.1.1. Artificially varying FLOP performance

Before implementing the discontinuous Galerkin traversal for solving the shallow water equations (*DG-SWE*), we wanted to see what the Sierpinski traversal with the edge-stack system can achieve in terms of MFLOPS and MB/sec. Eventually, our DG-SWE traversal uses three double-precision unknowns per triangle cell, and the flux terms transported on color-edges also use three doubles. It is obvious, that a higher-order discontinuous Galerkin discretization combined with a higher-order time integration scheme will use more than three double-precision variables per triangle. Therefore, I prepared a version of the traversal with 3 doubles per triangle, and one with 9. It is possible that 9 doubles are not even sufficient for a higher-order discretization scheme, but it shows the behavioral trend of the traversals.

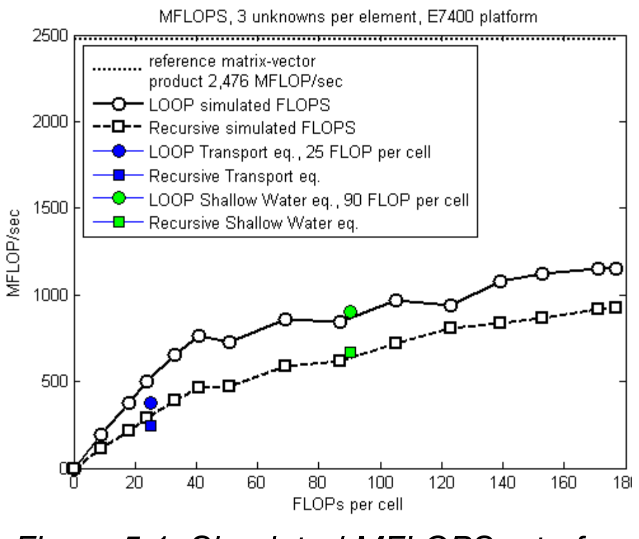
The current DG-SWE traversal performs on average 90 floating-point operations per triangle, while, for example, the consistency traversal during adaptive mesh refinement performs none. Since the higher-order DG-SWE traversal is still work in progress at the time of this writing, it is not clear how many floating-point operations it will use exactly. Hence, I introduced 177 artificial floating-point operations at first, and then I prepared further traversals by reducing this amount gradually to 0.

Increasing the amount of FLOPs per cell should get us closer to the situation where the performance is computation-bound. Increasing the amount of memory and lowering the FLOPs per cell should lead to a situation in which the performance is more memory throughput bound.

Execution time was measured for 100 traversals, for both loop-based and recursive versions, on a static but a-priori adapted non-uniform grid of 2 million triangles. Similar experiments we already published in Bader et al. (2012) with similar outcome. The current results are slightly better, since the edge-stack system was optimized by storing flux values on the color-edges only, while in Bader et al. (2012) the crossed-edges stored flux terms as well.

Loop-based and recursive traversals using 3 (left) and 9 (right) doubles per triangle are shown in Figure 5.1 for the E7400 desktop system. The MFLOP/sec rate is plotted against increasing amount of floating-point operations per triangle. One can observe, that the MFLOP/sec rate increases automatically for increasing computational load per triangle. The loop-based traversal always performs better than the recursive version. The limit performance of the loop-based traversal is slightly above 1 GFLOP/sec for both “thin” and “fat” traversal variants, which amounts to 40-50% of the reference performance achieved by the matrix-vector product. This limit performance seems to be independent of the amount of memory used per triangle.

The current DG-SWE traversal uses 3 doubles per triangle with 90 FLOPs, and its performance is directly comparable to the artificial FLOP traversals. The green markings in the left side of Figure 5.1 align well with the artificial FLOP traversals. The blue markings indicate the performance of a transport equation traversal with 25 FLOPs per triangle.



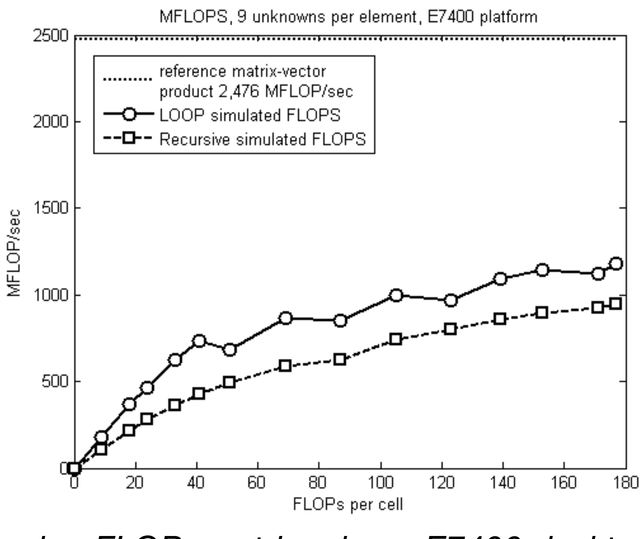


Figure 5.1: Simulated MFLOPS rate for varying FLOP per triangle on E7400 desktop system. 3 doubles per cell (left) and 9 doubles per cell (right).

The memory throughput rate of the same traversals is plotted in Figure 5.2. For low amount of FLOPs per triangle the traversals behave as if they were memory-bound, and up to 30-40 FLOPs per triangle are executed "for free", in the same amount of execution time (here equivalent to the inverse MB/sec rate). With no floating-point operations, the loop-based traversal with 9 unknowns gets very close to the maximum throughput achieved by the matrix-vector product. The rates for the 3 unknowns are lower than the ones with 9, and the Zero-FLOP loop-based traversal achieves only half of the matrix-vector product. Zero-FLOP traversals are used intensively during adaptive mesh refinement, and the 50% lower execution time of the loop-based over the recursive traversals is of utmost importance.

The blue markings representing the performance of the DG transport equation traversal on the left side of Figure 5.2 are not aligned with the artificial FLOP traversals, and the MB/sec rates are even lower. This drop in performance demonstrates that it is harder to achieve a good performance with both low memory footprint and low computational load per triangle cell. However, we also see that improved results can be expected for higher-order discretization with large amount of data volume and considerably higher computational load per triangle element. The expected 1 GFLOP/sec would be an exceptional performance for a matrix-free PDE solver on a fully adaptive grid.

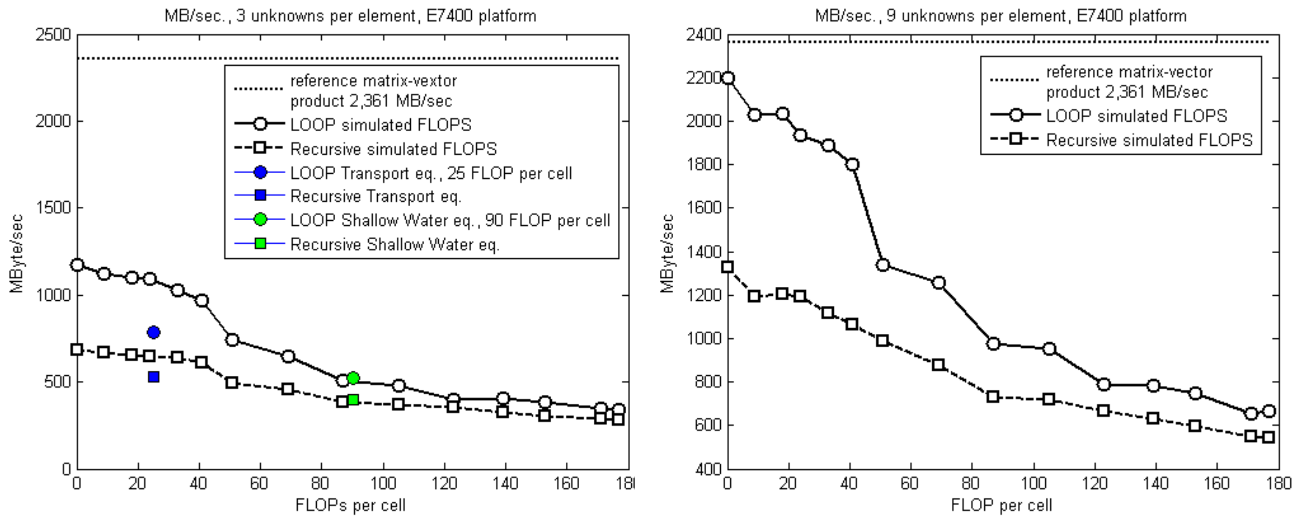


Figure 5.2: Simulated memory throughput rate for varying FLOP per triangle on E7400 desktop system. 3 doubles per cell (left) and 9 doubles per cell (right).

The same experiments were carried out on the T7700 desktop system, and the results confirm the conclusions drawn above. The MFLOP/sec rates are in Figure 5.3 and the MB/sec rates in Figure 5.4.

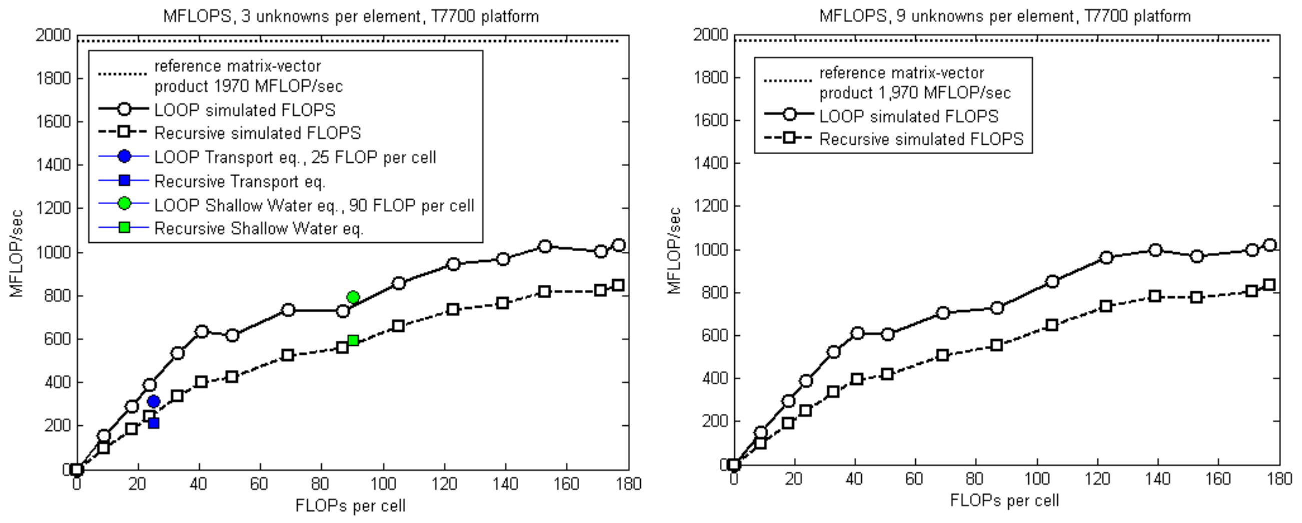


Figure 5.3: Simulated MFLOPS rate for varying FLOP per triangle on T7700 laptop system. 3 doubles per cell (left) and 9 doubles per cell (right).

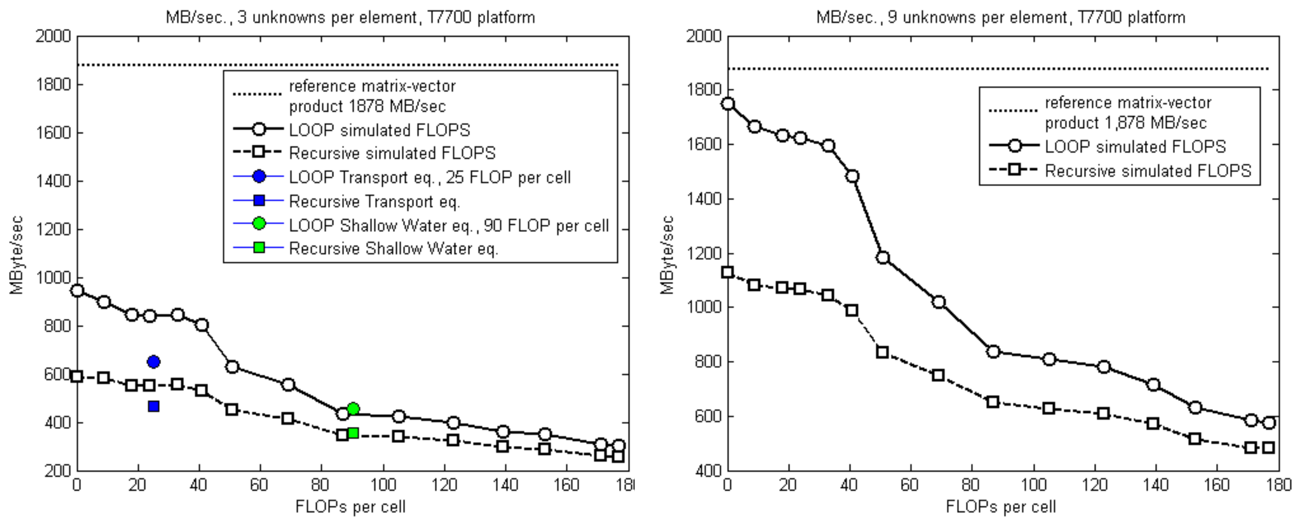


Figure 5.4: Simulated memory throughput rate for varying FLOP per triangle on T7700 laptop system. 3 doubles per cell (left) and 9 doubles per cell (right).

5.1.2. Discontinuous Galerkin loop vs. recursive traversal

The loop-based traversals always perform better than their recursive counterparts. The performance difference is higher for Zero-FLOP traversals, while the high-FLOP traversals tend to be computation-bound, where the difference is lower, but still significant. As presented in the previous chapter, the loop-based traversals are also beneficial to the parallel implementation, removing the complexity associated with the (unnecessary) traversal and/or adaptive refinement of a binary tree structure. All these advantages have a cost in terms of additional memory per triangle cell. Since the Sierpinski-based color-, entry/exit- and old/new properties can not be computed on the fly, in contrast to a recursive traversal, they have to be stored for the leaf triangles.

Table 5.3 lists the memory usage of the grid components that are used during a discontinuous Galerkin traversal. The binary tree uses 1 byte per node and it contains twice as many nodes as leaf triangles. A bit-wise implementation would use only 2 bits per triangle. A leaf triangle array will contain the numerical unknowns, which – in the current implementation – are 3 double-precision numbers, and, for the loop-based traversal, the Sierpinski attributes in 12 bytes. A color-edge contains 3 doubles for flux transport and 3 bytes for management purposes. The optimizing compiler translates the raw 27 bytes into chunks of 32. Since most of the time a color-edge is “shared” between two neighboring triangles, its memory usage is also “shared” for simplicity, and counted as 16 bytes per triangle.

Summing up the components used by the recursive- and loop-based traversal gives per triangle 42 bytes for the former and 52 bytes for the latter. The additional 10 bytes per triangle is only 25% of the otherwise 42 bytes, and is a constant factor. If we consider a future higher-order discontinuous Galerkin discretization, then the amount of memory for the numerical unknowns has to be increased, but not the 12 bytes for the Sierpinski attributes.

Grid entity	Entity components	bytes	per cell usage multiplier	bytes per cell
binary tree	binary tree nodes	1	2x	2
Leaf triangle cell (array)	leaf triangle Sierpinski attributes	12	1x	12
	leaf triangle numeric data (water height)	3x double = 24		24
Color edge (stack system)	edge management data	3	~ 0.5x - shared by two triangle cells - border edges are not shared	0.5x 32 = 16 (treated as a single entity, the total of 27 bytes are rounded up to 32 by the optimizing compiler)
	edge numeric data (flux)	3x double = 24		
Total used by LOOP DG-SWE traversal (no binary tree)				**52**
Total used by RECURSIVE DG-SWE traversal (no SFC data)				**42**

Table 5.3: Memory usage in DG-SWE traversal – bytes per triangle.

Tables 5.4 and 5.5 list the results of experiments on the E7400 and T7700 platforms, respectively. Loop-based and recursive traversals were performed on different grids (uniform and a-priori adaptive) and execution time of 100 traversals was measured. The amount of triangles processed per second is given in the first of the last three columns as a universal comparison between any kind of traversals. The other measures are the memory throughput rate in MB/sec and the floating-point performance in MFLOP/sec. Loop-based results are highlighted in gray color. In the last three rows the average performance of the loop-based- and the recursive traversals is given, followed by the benchmark block-diagonal matrix-vector product.

On both platforms the loop-based DG-SWE traversal exceeds the recursive with about 35% advantage. By processing 10 million triangles per second on E7400, it achieves 25% of the memory throughput speed of the block-diagonal matrix-vector product and roughly 40% of its floating-point performance.

DG-SWE on E7400	ref. depth	# triangles	total memory used (MB)	exec. time of 100 traversals (sec)	Million triangles per second	memory throughput (MB/sec)	Floating point perf. (MFlops)
loop traversal	20	2,097,152	109.12	21.27	9.86	513.02	887.37
recursive traversal	20	2,097,152	113.31	28.74	7.3	394.26	656.73
loop traversal	21	4,194,304	218.17	40.72	10.3	535.78	927.03
recursive traversal	21	4,194,304	226.56	55.56	7.55	407.78	679.42
loop traversal	22	8,388,608	436.34	81.77	10.26	533.62	923.29
loop traversal	23	16,777,216	872.55	162.99	10.29	535.34	926.41
loop, a-priori adaptive	20-24	2,121,520	110.38	22.52	9.42	490.14	847.85
recursive, a-priori adaptive	20-24	2,121,520	114.63	29.01	7.31	395.14	658.18
loop, a-priori adaptive	21-25	4,228,784	219.96	41.73	10.13	527.1	912.03
recursive, a-priori adaptive	21-25	4,228,784	228.42	55.84	7.57	409.06	681.57
Average LOOP DG-SWE trav. perf. (E7400)					**10.04**	**522.5**	**904**
Average RECURSIVE DG-SWE trav. perf. (E7400)					**7.43**	**401.56**	**668.98**
Block-diagonal matrix-vector product (E7400)						**2,361 MB/sec**	**2,476 MFLOPS**

Table 5.4: DG-SWE performance on E7400 platform. Loop version is 35% faster than the recursive one, achieves 22% of the memory throughput speed and 36% of the reference MFLOPS rate.

DG-SWE on T7700	ref. depth	# triangles	total memory used (MB)	exec. time of 100 traversals (sec)	Million triangles per second	memory throughput (MB/sec)	Floating point perf. (MFlops)
loop traversal	20	2,097,152	109.12	23.59	8.89	462.57	800.1
recursive traversal	20	2,097,152	113.312	31.72	6.61	357.23	595.03
loop traversal	21	4,194,304	218.17	48.09	8.72	453.63	784.96
loop traversal	22	8,388,608	436.34	95.69	8.77	455.99	788.98
loop traversal	23	16,777,216	872.55	191.08	8.78	456.64	790.22
loop, a-priori adaptive	20-24	2,121,520	110.38	24.25	8.75	455.18	787.37
recursive, a-priori adaptive	20-24	2,121,520	114.63	32.42	6.54	353.58	588.95
Average LOOP DG-SWE trav. perf. (T7700)					**8.78**	**456.8**	**790.33**
Average RECURSIVE DG-SWE trav. perf. (T7700)					**6.58**	**355.4**	**591.99**
Block-diagonal matrix-vector product (T7700)						**1,878 MB/sec**	**1,970 MFLOPS**

Table 5.5: DG-SWE performance on T7700 platform. Loop version is 33% faster than the recursive one, achieves 24% of the memory throughput speed and 40% of the reference MFLOPS rate.

The same experiments were performed also with the transport equation kernel, which – as stated before – has 25 FLOPs per triangle. The exact measurement data is omitted for brevity, instead a summary of the results is given in Table 5.6. The loop traversal with the transport equation kernel beats the recursive version by 53% and 45% on E7400 and T7700 platforms, respectively. By processing 15 million triangles per second on E7400, it achieves 33% of the reference memory throughput speed of the block-diagonal matrix-vector product, while MFLOP/sec is at 15%. Similar percentage rates were achieved on T7700.

Loop-based traversals vs. reference performance	FLOP per triangle	Beats recursive version by (%)	Million triangles per second	memory throughput (% of reference)	Floating point perf. (% of reference)
E7400					
DG-SWE	90	35%	10.04	22%	36%
TRANSPORT	25	53%	15.08	33%	15%
E7400 reference block-diagonal matrix-vector product				100% 2,361 MB/sec	100% 2,476 MFLOPS
T7700					
DG-SWE	90	33%	8.78	24%	40%
TRANSPORT	25	45%	12.46	34%	16%
T7700 reference block-diagonal matrix-vector product				100% 1,878 MB/sec	100% 1,970 MFLOPS

Table 5.6: Discontinuous Galerkin traversals with SWE and TRANSPORT kernels vs. reference performance of the block-diagonal matrix-vector product in percentage points.

5.1.3. Edge-stack system vs. explicit neighbor indexing

Another set of experiments is meant to show the advantages of using an edge-stack system for (flux-) information transport between *far neighbor* triangles, instead of a simple explicit neighbor indexing solution. The theoretical description of both information transport solutions and their differences were presented in chapter 2, section 2.2. The explicit neighbor indexing solution of the shallow-water equations was also referred to as *triangle-only* solution, because it involves data structures associated to the triangle cells alone. In this section I refer to it as *Neighbor index traversal*, since it highlights the implementation details better.

Only the loop-based version of the *Neighbor index traversal* was implemented and Table 5.7 lists its memory footprint. The actual index of the far neighbor across the missing color-edge was included in the 12 bytes of Sierpinski attributes, so for that purpose no additional memory was used. However, additional memory is allocated in each triangle cell for the flux term that needs to be exchanged with the far neighbor. This amounts to 3 doubles per triangle cell. This storage scheme supplementing the color-edges uses roughly twice the amount of memory for flux transfer during a traversal, than the color-edge system. The partial flux is computed and stored in the first triangle encountered during a traversal, and later the far neighbor reaches back to it for usage. The storage allocated in the second triangle is not used in the current traversal, but only in the next one. Hence the

23% more memory needed by the *Neighbor index* traversal relative to the *edge-stack system solution* of the DG-SWE.

Grid entity	Entity components	bytes	per cell usage multiplier	bytes per cell
Leaf triangle cell (array)	leaf triangle Sierpinski attributes	12 (incl. index pointing to far neighbor)	1x	64 12 + 24 + 24 = 60, but treated as a single entity, the total of 60 bytes are rounded up to 64 by the optimizing compiler
	leaf triangle numeric data (water height)	3x double = 24		
	temporary flux storage for reuse in the far neighbor (replaces the edge-stack system)	3x double = 24		
Total used by "Neighbor index" traversal				**64**
Total used by "DG-SWE" traversal				**52**

Table 5.7: Memory usage in "Neighbor index" traversal – bytes per triangle. With 64 bytes per triangle it uses 23% more memory than the reference "DG-SWE".

Tables 5.8 and 5.9 list the performance of the *Neighbor Index traversal* on the E7400 and T7700 platforms, respectively. The DG-SWE traversal is slightly faster, but not by much. Similar results were obtained with the transport equation kernel, where the edge-stack system solution was 10% faster than the neighbor index traversal. The fact, that the triangles are ordered according to the Sierpinski space-filling curve, already carries the huge advantage of excellent cache behavior. Level-1 cache hit-rates are above 99% and are attributed primarily to the locality properties of the Sierpinski curve. Cache-efficiency is addressed in more detail in Bader et al. (2012). If the *neighbor index* solution would traverse the triangles in the Cartesian order, then the performance would probably decrease due to the increased amount of cache-misses. Unfortunately, such a traversal was not implemented due to time constraints. However, the advantage of using an edge-stack system is not only the lower – perhaps optimal – memory footprint, but it also serves as basis for inter-process communication in the parallel case, as described in chapter 4.

Neighbor index on E7400	ref. depth	# triangles	total memory used (MB)	exec. time of 100 traversals (sec)	Million triangles per second	memory throughput (MB/sec)	Floating point perf. (MFlops)
loop traversal	20	2,097,152	134.35	21.71	9.66	618.84	869.39
loop traversal	21	4,194,304	268.57	41.39	10.13	648.88	912.03
loop traversal	22	8,388,608	537.13	84.42	9.94	636.26	894.31
loop traversal	23	16,777,216	1074.00	164.61	10.19	652.45	917.29
Average “Neighbor index” traversal perf. (E7400)					**9.98**	**639.11**	**898.25**
Average “DG-SWE” traversal perf. (E7400)					**10.04**	**522.5**	**904**

Table 5.8: “Neighbor index” implementation performance on E7400 platform. Edge-stack system solution is only slightly faster.

Neighbor index on T7700	ref. depth	# triangles	total memory used (MB)	exec. time of 100 traversals (sec)	Million triangles per second	memory throughput (MB/sec)	Floating point perf. (MFlops)
loop traversal	20	2,097,152	134.35	25.34	8.28	530.19	744.84
loop traversal	21	4,194,304	268.57	48.03	8.73	559.17	785.94
loop traversal	22	8,388,608	537.13	101.53	8.26	529.04	743.6
Average “Neighbor index” traversal perf. (T7700)					**8.42**	**539.47**	**758.13**
Average “DG-SWE” traversal perf. (T7700)					**8.78**	**456.8**	**790.33**

Table 5.9: “Neighbor index” implementation performance on T7700 platform. Edge-stack system solution is only slightly faster.

5.1.4. Adaptive mesh refinement step vs. DG-SWE

Dynamically adaptive refinement and coarsening of the grid is the result of several management traversals, as described in chapter 3, and the parallel version in chapter 4. We want to know how much computing time it costs relative to a numerical DG-SWE traversal. In the experiments presented in this sub-section we measured the parallel implementation with loop-based traversals running on a single processor.

The execution time of a full adaptive step depends not only on the size of the initial grid, but also on that of the final grid. Traversals exclusively on the initial grid are "*marking initial refinement and coarsening requests*", "*consistency*" and "*coarsening correctness check*". "*New neighbor prediction*" is also running on the initial grid, but managerial work depends on the size of the final grid. "*Adaptation*" traversal with interpolation takes the initial grid as input and constructs the triangle system of the final grid. The "*re-initialization*" traversal runs on the final triangle system and rebuilds the edge-stack system, ending the full adaptive step.

For performance measurements, I prepared five scenarios of adaptive mesh refinement, whose results are listed in Table 5.10 and continued in Table 5.11. All scenarios start on an initial grid of 2 million triangles, and they differ in the size of the final grid:

- "*full double refinement*" refines every triangle twice, the final grid has 8 million cells; this is the maximum refinement rate (4x size of initial grid);
- "*full single refinement*" bisects every triangle, the final grid has 4 million cells;
- "*partial refinement*" refines about 25% of the initial grid (triangles of type *k1n*), and subsequent consistency traversals refine an additional 25%, yielding a final grid of 3 million cells;
- "*no refinement*" leaves the grid as it is;
- "*full coarsening*" merges every pair of siblings, final grid has 1 million cells.

This way one can clearly identify traversals dependent on the initial grid from those that depend on the final- or on both grids.

The first- and last row of each scenario lists the two reference execution times of the DG-SWE traversal on the initial- and on the final grid, respectively. The rows in between list the execution times of the other traversals and of the total adaptive step (as a sum of all the traversals). The execution time relative to the DG-SWE traversals is given in the last two columns of the table.

Performance of specific traversals in detail:

- "*Marking initial refinement and coarsening requests*" is a Zero-FLOP traversal on the initial grid. Execution time is constant throughout all five scenarios with about 35% of the "*DG-SWE on initial grid*".

- "*Consistency*" is also a Zero-FLOP traversal on the initial grid and is usually performed multiple times in an adaptive step. In these scenarios it is performed exactly twice and its execution time of 35% of "*DG-SWE on initial grid*" is counted twice in the sum of the "*Total adaptive step*".

- "*Coarsening correctness check*" is a very fast traversal on the initial triangle array alone, checking for siblings in coarsening requests. It is not using the edge-stack system and, therefore, its execution time is 10 times faster than a standard Zero-FLOP traversal. Hence the execution time was added to the time of the "*Consistency*" traversals in the tables below.

- "*New neighbor prediction*" is traversing the initial grid, and it computes the neighbor indexes of the future grid that does not exist yet. Because the computational work is dependent on the final grid, one can observe that its execution time is influenced by it. In the "*Full double refinement*" scenario it amounts to 45% of "*DG-SWE on initial grid*" and it gradually decreases to 28% in the scenario with the smallest final grid.

- "*Adaptation with interpolation*" is the process of taking the initial triangles with the initial numerical unknowns and produce the new triangles together with the new unknowns. This traversal is dependent on both the initial- and the final grid. A pattern can be observed in scenarios with growing final grid, where the execution time is around 30% of the "*DG-SWE on final grid*". For scenarios where the final grid is smaller or equal to the initial grid, the execution time tends to be 25-28% of the "*DG-SWE on initial grid*".

- "*Re-initialization*" is a traversal on the final triangle array, and the new edge-system is rebuilt. Its execution time is 70% of the "*DG-SWE on final grid*".

- "*Total adaptive step*" is the sum of the execution time of these traversals, with the "*Consistency*" counted twice. Execution time relative to the "*DG-SWE on initial grid*" is highest when the final grid is 4 times as big as the initial one, in which case it is 5.48 times more expensive. This ratio sinks gradually in the other scenarios with decreasing size of the final grid, with the extreme of 1.93 for the *full coarsening* scenario in which the initial grid shrinks to half its size. The trend is in reversed when compared to execution time of the "*DG-SWE on final grid*", with 1.33 times more expensive on the *full double refinement* scenario and gradually increasing up to 3.5 on the *full coarsening*.

Dynamic AMR vs. DG-SWE E7400	initial grid # triangles	final grid # triangles	traversal exec. time	% of DG-SWE initial grid	% of DG-SWE final grid
Full DOUBLE REFINEMENT of the initial grid, each triangle bisected twice:					
DG-SWE – initial grid	2,097,152	8,388,608	0.29		24.17
Marking			0.10	34.48	8.33
Consistency avg. (2x) + coarsening check			0.10	34.48	8.33
New neighbors			0.13	44.83	10.83
Adapt (+ interpolate)			0.38	131.03	31.67
Re-initialization			0.78	268.97	65
Total adaptive step			1.59	548.28	132.5
DG-SWE – final grid			1.20	413.79	
Full SINGLE REFINEMENT of the initial grid, each triangle bisected once:					
DG-SWE – initial grid	2,097,152	4,194,304	0.29		48.33
Marking			0.10	34.48	16.67
Consistency avg. (2x) + coarsening check			0.10	34.48	16.67
New neighbors			0.11	37.93	18.33
Adapt (+ interpolate)			0.18	62.07	30
Re-initialization			0.40	137.93	66.67
Total adaptive step			0.99	341.38	165
DG-SWE – final grid			0.60	206.9	
Partial refinement of the initial grid, about 25% of the triangles originally bisected. Consistency refines more as needed.					
DG-SWE – initial grid	2,097,152	3,144,704	0.29		64.44
Marking			0.10	34.48	22.22
Consistency avg. (2x) + coarsening check			0.10	34.48	22.22
New neighbors			0.09	31.03	20
Adapt (+ interpolate)			0.14	48.28	31.11
Re-initialization			0.30	103.45	66.67
Total adaptive step			0.83	286.21	184.44
DG-SWE – final grid			0.45	155.17	

Table 5.10 Adaptive mesh refinement traversals vs. DG-SWE times.

Dynamic AMR vs. DG-SWE E7400	**initial grid # triangles**	**final grid # triangles**	**traversal exec. time**	**% of DG-SWE initial grid**	**% of DG-SWE final grid**
NO REFINEMENT of the initial grid:					
DG-SWE – initial grid	2,097,152	2,097,152	0.29		
Marking			0.10	34.48	
Consistency avg. (2x) + coarsening check			0.10	34.48	
New neighbors			0.09	31.03	
Adapt (+ interpolate)			0.08	27.59	
Re-initialization			0.20	68.97	
Total adaptive step			0.67	231.03	
DG-SWE – final grid			0.27		
Full COARSENING of the initial grid:					
DG-SWE – initial grid	2,097,152	1,048,576	0.29		181.25
Marking			0.10	34.48	62.5
Consistency avg. (2x) + coarsening check			0.10	34.48	62.5
New neighbors			0.08	27.59	50
Adapt (+ interpolate)			0.07	24.14	43.75
Re-initialization			0.11	37.93	68.75
Total adaptive step			0.56	193.1	350
DG-SWE – final grid			0.16	55.17	

Table 5.11: Adaptive mesh refinement traversals vs. DG-SWE times (continued).

A graphical representation of the values in Tables 5.10 and 5.11 is in Figure 5.5, in which the execution time is plotted against the amount of triangles added by the adaptive step to the initial grid of 2 million cells.

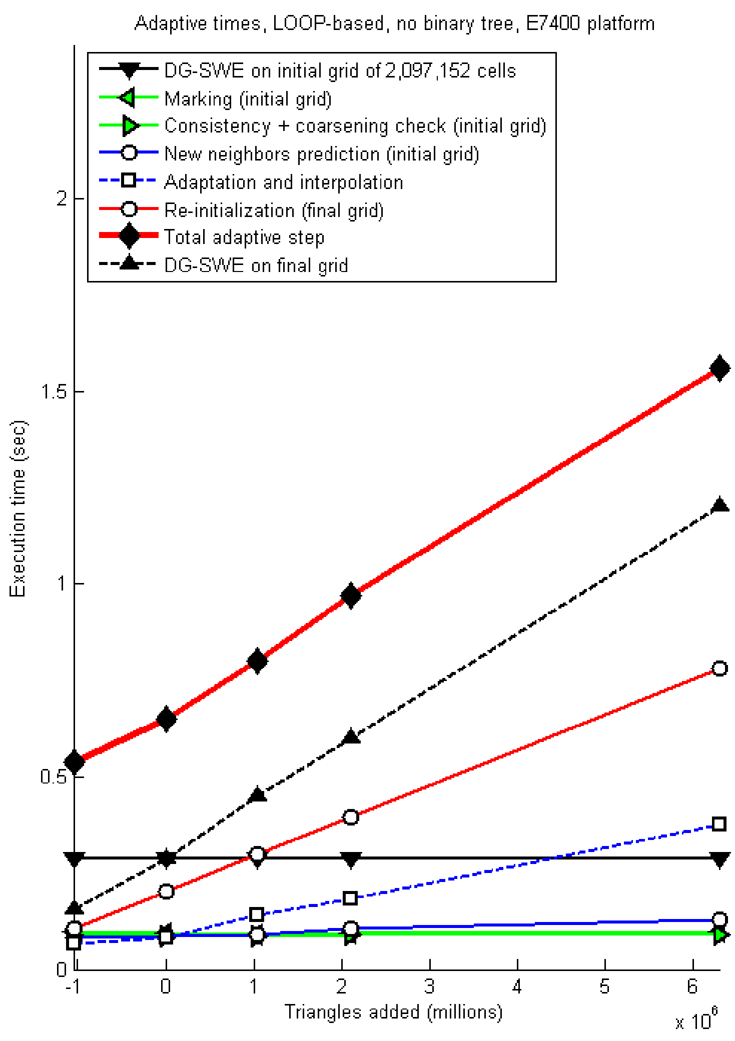


Figure 5.5: Adaptive mesh refinement traversal execution times and DG-SWE on initial- and final grid.

Although it is not impossible, it is unlikely for the initial grid to grow with maximum rate, or to shrink to half its size at once in our SWE simulation. Therefore, when trying to answer the question “*How many DG-SWE traversals could be performed in the time used for a full adaptive step ?*”, we can exclude the extreme scenarios of “*full double refinement*” and “*full coarsening*”. It is also probably safe to say that usually the grid will not double its size, but even if it does, the time taken for that particular adaptive mesh refinement is equivalent to 3.41 DG-SWE traversals on the initial grid. The full adaptive step in the “*partial refinement*” scenario, where the grid grows by 50%, is equivalent to 2.86 DG-SWE traversals on the initial grid, and the cost tends to decrease for less changes in the grid.

When a future high-order implementation will be available, with 4 or more discontinuous Galerkin type traversals per time step, and with more floating-point operations per triangle, the cost of adaptive mesh refinement will become more and more affordable. The execution time of the adaptive step would increase with a considerably slower pace than that of a high-order Runge-Kutta time step, since only the time to interpolate and to re-initialize the color-edge flux values would be an increasing factor. Furthermore, the Zero-FLOP traversals that run exclusively on the initial grid could be merged with the numerical computation, reducing some or all of the cost currently incurred by "*marking*" and "*consistency*".

Finally, in the serial performance experiments, the loop-based adaptive step is faster than the older recursive implementation with at least 30%, as illustrated in Figure 5.6. Similar results on the T7700 platform also confirm the statements presented in this section.

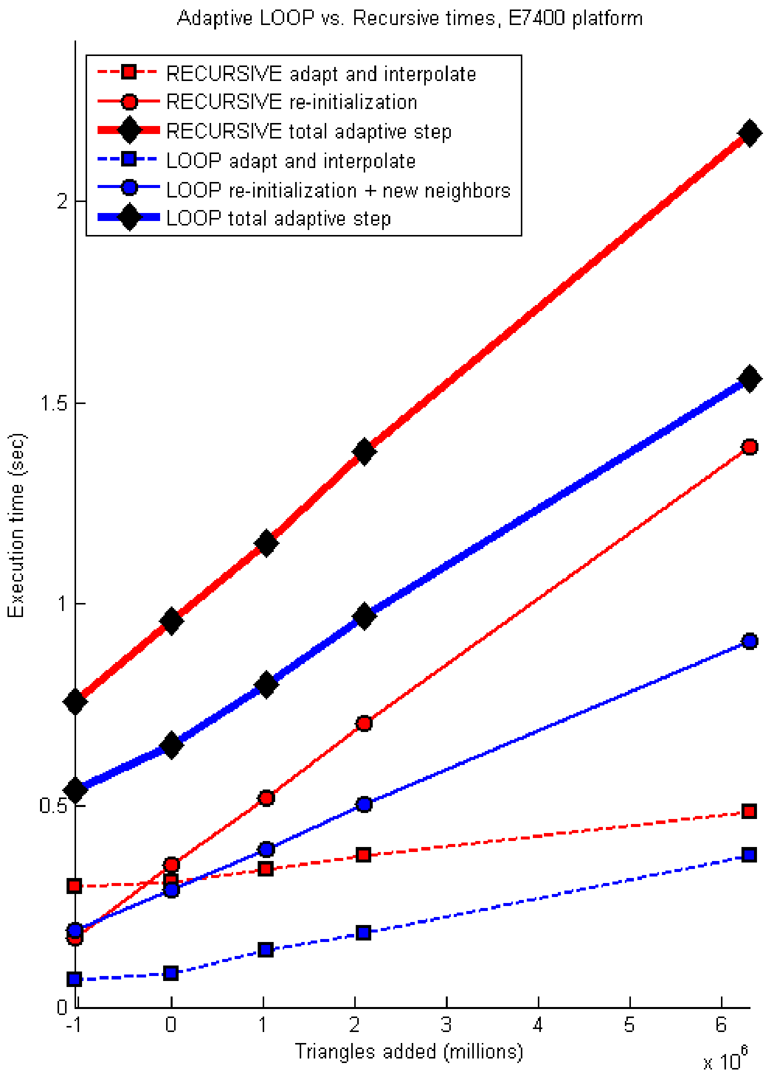


Figure 5.6: Serial loop-based (blue) vs. recursive (red) adaptation performance. Loop-based traversals pay off in adaptive mesh refinement too.

5.1.5. Evaluation of the serial performance results

In the series of experiments presented above, I tried to show certain qualitative and quantitative aspects of the Sierpinski traversals for numerical computation and for grid management purposes. Different traversal types use different amounts of floating-point operations, and, therefore, I created traversals with artificial FLOPs, varying their amounts from 0 to 177 per triangle. The traversals were executed with a “thin” and a “fat” memory footprint of 3 and 9 doubles per triangle, respectively. As a reference performance of the hardware I used a block-diagonal matrix-vector multiplication with 7 floating-point operations per element. The MFLOP/sec rate achieved by the Sierpinski traversals increased automatically with the increasing computational load per triangle. The highest MFLOP/sec rate of about 1 GFLOP/sec for both thin and fat variants, which amounts to 40-50% of the reference performance of the matrix-vector product. The memory throughput rate of the traversals decreases with increasing computational load per triangle. The highest rate is achieved by the fat Zero-FLOP traversal using 9 doubles per triangle, which is 90% of the reference throughput of the matrix-vector product.

The superior performance of the loop-based vs. the recursive traversal was also demonstrated for both artificial FLOP and for the DG-SWE traversals. The loop-based Zero-FLOP traversals are almost twice as fast as the recursive implementation, which are used in consistency checks during adaptive mesh refinement. The difference in execution time diminishes with increasing computational load. The DG-SWE traversal is about 35% faster in the loop than in the recursive version.

The performance of the edge-stack system in the DG-SWE traversal was compared to an explicit neighbor indexing implementation. In this implementation, the flux information transport across far neighbor triangles was done by directly accessing the corresponding triangle element in the linear array. The edge-stack system implementation is better in terms of memory usage, since the index implementation uses 23% more memory per triangle. In terms of computing time, the edge-stack system version of the DG-SWE traversal is slightly faster, about 2-5%. The excellent cache efficiency of the Sierpinski traversal is already given by the space-filling curve ordering of the triangles.

Last but not least, the performance of the adaptive mesh refinement and coarsening with several artificial scenarios was compared to DG-SWE traversals on the input- and on the final grid. We concluded that the average cost of dynamic adaptivity is that of about 2.8 – 3.4 DG-SWE traversals. The loop implementation of all the traversals required to perform the adaptive step is at least 30% faster than the recursive implementation. Furthermore, the cost of dynamic adaptivity relative to the cost of computing a time-step in a higher spatial- and temporal discretization scheme will sink considerably. The complexity of the numerical computation will increase due to the amount of floating-point operations per triangle and more traversals per time-step, while the adaptive mesh refinement and coarsening algorithm remains basically the same, except for interpolation and restriction. Therefore, the cost of dynamic adaptivity will become even more attractive, affordable and acceptable than it is in the current implementation.

5.2. Parallel performance analysis

When executing programs on multiple processors, we are mainly interested in how well they scale in the parallel environment. The long-term goal of this work is to achieve sub-real-time simulation of the oceanic wave propagation, and, thus, the main interest is the strong speed-up efficiency. In the following sub-sections I am presenting strong speed-up results for numerical- and consistency traversals on a static grid, for the full adaptive step scenario called "*partial refinement*" in the previous section, and, finally, the Tsunami simulation with adaptive mesh refinement after each time step. The Tsunami simulation scenario is joint work with K. Rahnema.

5.2.1. Parallel traversals on static grids

Better parallel speed-up is expected for the discontinuous Galerkin traversals than for a Zero-FLOP traversal used in adaptive mesh refinement. In the numerical traversal the amount of computational work is considerably higher per triangle than in a *Consistency* traversal, while the MPI communication overhead is the same. However, good parallel speed-up can be achieved as long as the communication-to-computation ratio is low enough, which is the case for large grids.

In figures 5.7 and 5.8 the grid contained a total of 67 million triangle cells equally distributed among the processors. Non-blocking MPI communication was used for inter-process communication through the process-boundary edges on the Infiniband Cluster of the Technische Universität München. This system has 32 "AMD Opteron 850" nodes with 4 cores each running on 2.4 GHz clock frequency. Each node has 8 GB main memory and are interconnected with an InfiniBand Switch (MTEK43132) from Mellanox Technologies.

Strong speed-up is shown for up to 128 cores for the DG-SWE traversal in Figure 5.7. The speed-up efficiency is worst at about 80% with 96 processors, but it gets well above 90% on 112 and 128 processors. The sudden performance drop on 96 processors might be caused by either a bug in the implementation or by a disadvantageous arrangement of the parallel domains for the MPI communication. Due to time constraints, this matter has not been investigated yet. However, in my opinion, the efficiencies obtained are a very good first result.

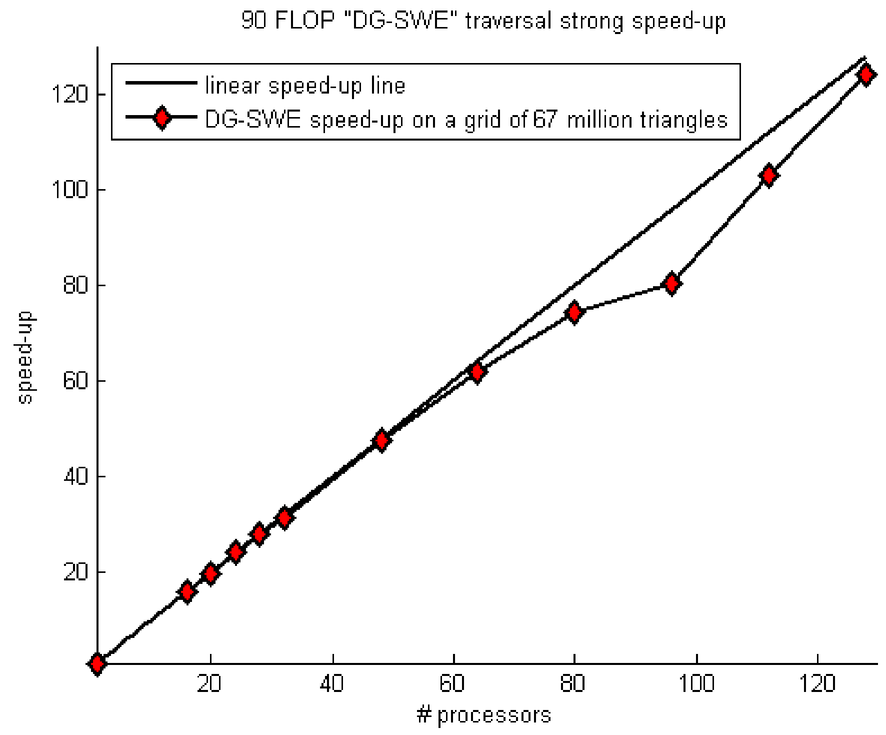


Figure 5.7: DG-SWE traversal on a grid of 67 million cells. Strong speed-up of up to 128 processors on the Infinicluster.

For the Consistency traversal the speed-up for up to 112 processors is shown in Figure 5.8, with overall efficiencies between 85% and 90%.

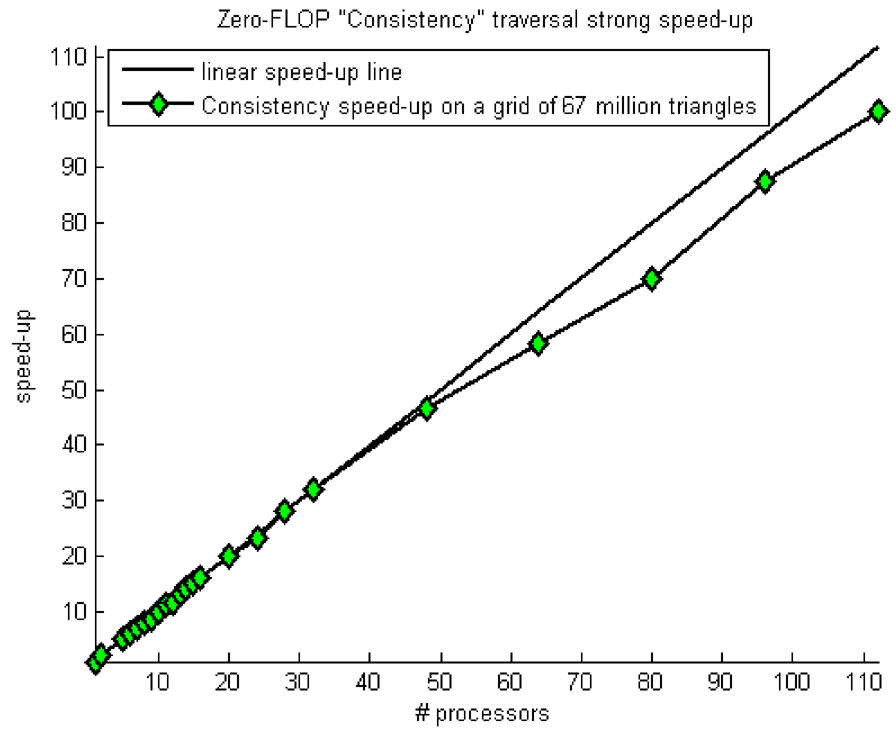


Figure 5.8: Zero-FLOP consistency traversal on a grid of 67 million cells. Strong speed-up of up to 112 processors on the Infinicluster.

5.2.2. Parallel adaptive mesh refinement

The "*partial refinement*" scenario introduced previously in section 5.1.4 was tested on the Infiniband Cluster using up to 128 processors. The full adaptive step in this experiment starts on an initial grid of 33 million triangles distributed equally among the processors, and creates a final grid of 50 million cells, with load-balancing switched off.

Strong speed-up efficiencies for the full adaptive step, involving several management traversals, are illustrated in Figure 5.9. The catastrophic drop in efficiency for 64 and 80 processors is either a bug or a disadvantageous arrangement of the parallel domains regarding MPI communication. With about 65% for 64 processes and 30% for 80 it is far worse than for the numerical traversal on the static grid from the previous section. However, the performance recovers for 96, 112 and 128 processors with excellent speed-up efficiency of around 85%, which, in my opinion, is a very impressive first result.

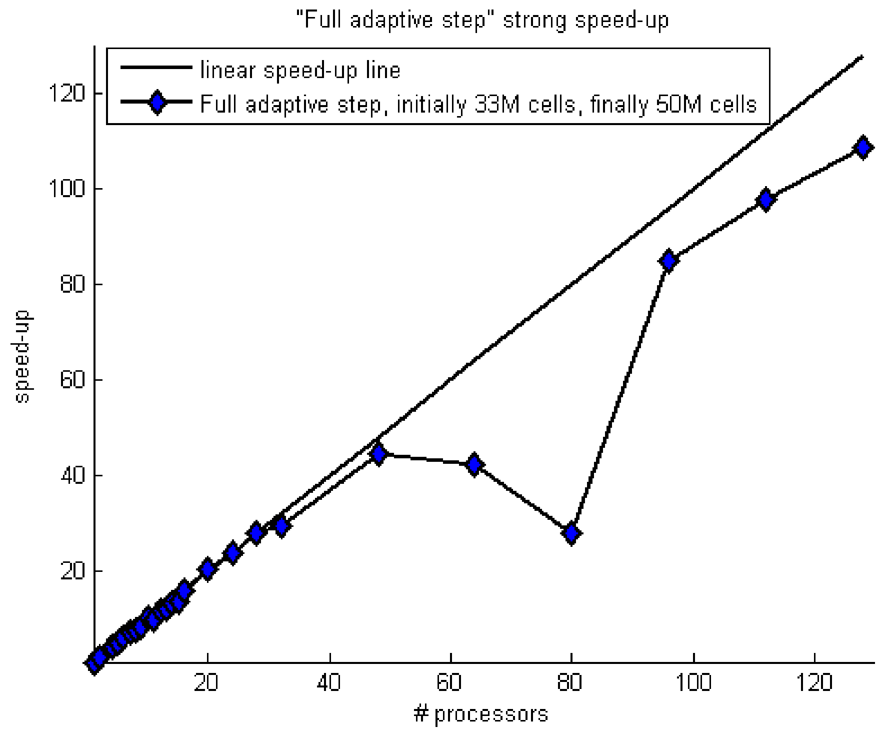


Figure 5.9: Full adaptive step without load-balancing, starting on an initial grid of 33 million cells and producing a grid of 50 million cells. Strong speed-up of up to 128 processors on the Infinicluster.

If load-balancing is switched on, then the full adaptive step contains an additional "*MPI_All-To-All-V*" type of extra communication. This involves redistribution of the three arrays that represent the numerical unknowns, the Sierpinski attributes and the binary tree location codes. The performance penalty incurred by this redistribution is highly dependent on the type and nature of the load imbalance, on the communication hardware and on the MPI implementation as well. Because of the rich variety of the possible types of load imbalance that can occur, and the shortage of time, no extensive studies were performed by myself in this regard. All I can say is that in the very few test scenarios that I tested on the Infiniband Cluster, the load-balancing incurred a performance penalty of roughly 17%. However, this estimate is neither a lower-, nor an upper limit, and additional performance studies are necessary for a better estimation.

5.2.3. Parallel SWE simulation with full re-meshing

Finally, the solution of the shallow water equations with full re-meshing after each time step is shown in Figure 5.10, which is joint work with K. Rahnema. This test was executed on the SGI-ICE cluster at the Leibnitz Supercomputing Center in Garching, Germany. It has 32 dual-socket quad-core Nehalem nodes with 24 GB main memory in each node.

After each DG-SWE traversal a full adaptive step is executed. Refinements in each triangle are based on the change of water level in comparison to the previous time step. Here we see again a performance drop for the ill-fated case of 80 processors. This observation on two different machines suggests that the problem is not caused by the hardware, but – as stated earlier – is most likely a bug that appears in this special case. However, bugs get fixed over time, and the good news is that the strong speed-up efficiency recovers for 96, 112 and 128 processors. For the latter case the efficiency of the strong speed-up compared to the performance on 8 cores is slightly above 90%.

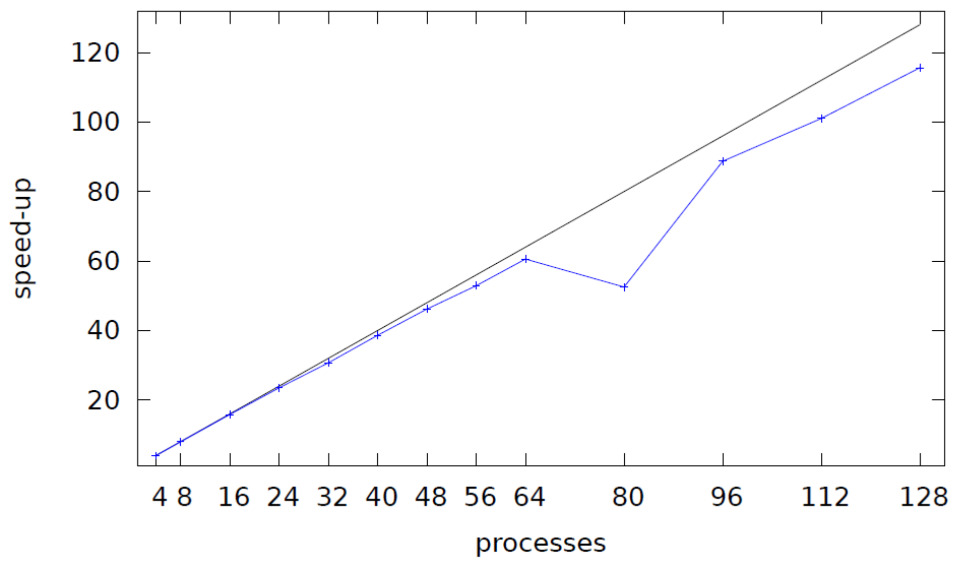


Figure 5.10: Solution of the shallow water equations with full re-meshing at every time step. The grid contains on average 33 million triangle cells. Strong speed-up of up to 128 processors on the SGI-ICE cluster. Joint work with Kaveh Rahnema.

5.2.4. Evaluation of the parallel performance results

In the parallel performance experiments I tested the strong scaling of the different Sierpinski traversals. The first set of experiments measured the strong speed-up of the DG-SWE and the zero-FLOP consistency traversal on a static grid for up to 128 cores. The strong speed-up efficiency of well above 80% for both type of traversals is a very good first result, considering that no special optimization effort was spent on the parallel messaging overhead. The worst speed-up of the DG-SWE traversal on 96 processors is at 80% efficiency, while for 112 and 128 processors is well above 90%. The ill-behaved case on 96 processors is either a bug in the implementation, or is a disadvantageous arrangement of the parallel domains for the MPI communication. Since the zero-FLOP consistency traversal does not exhibit similar drop in efficiency, it is fairly possible that there are some “features” in the code that need to be addressed.

The speed-up efficiency of the adaptive mesh refinement and coarsening was measured for the *partial refinement* scenario without load-balancing, where the initial grid of 33 million elements grows to 50 million. Here, too, we see a large performance drop when using 64 or 80 processors. However, the strong speed-up performance recovers when using 96, 112 or 128 processors and achieves 85% efficiency, which is an excellent result. Load-balancing, if switched on, incurs a performance penalty of roughly 17% on the Infiniband Cluster for this scenario. Load-balancing depends not only on the specific scenario and the amount of load imbalance, but also depends strongly on the hardware, since it has to exchange data on the network that connects the processors. Unfortunately extensive load-balancing studies were not performed due to lack of time.

The solution of the shallow water equations with full dynamic adaptivity after each time-step was tested on the SGI_ICE cluster (joint work with K. Rahnema). It performs a DG-SWE traversal and adaptive mesh refinement and coarsening in each time-step, without load-balancing. Performance drop was again observable on 80 processors, which, too, suggests a bug. However, strong speed-up efficiency is above 90% on 96, 112 and 128 processors relative to the performance on 8 cores.

Currently 90 floating-point operations are performed per triangle per time step. The adaptive mesh refinement dominates the execution time, since an adaptive step costs approximately 3.5 numerical traversals (or Euler time steps in this case). This ratio will change in favor of the computation, when a higher order discontinuous Galerkin spatial discretization with Runge-Kutta type time stepping will be implemented. But, the fact, that this excellent speed-up efficiency was reached with the current setting, demonstrates the feasibility of our Sierpinski-based solution method. With more floating point operations per triangle, and more numerical traversals in-between adaptive mesh refinement steps, the strong speed-up efficiency will most likely be sustainable for even more processors. With sustainable strong speed-up efficiency on a – perhaps one- or two orders of magnitude – higher amount of processors, the ultimate goal of a sub-realtime Tsunami simulation starts to look more and more realistic.

6. Conclusion

This work presented our approach for the fast simulation of the oceanic wave propagation, based on algorithms using the Sierpinski space-filling curve. In the event of a tsunami, a sub-realtime simulation of the wave propagation on the open ocean would enable much more precise inundation predictions on coastal regions. In the current tsunami forecasting systems the wave propagation is not simulated for the real scenario, but, rather, a linear combination of precomputed scenarios is selected as initial condition for the inundation algorithms. An accurate and fast simulation of the wave propagation, and, thus, of the whole tsunami event itself, before it unfolds, would lead to far better hazard estimation for the authorities responsible for tsunami warnings.

We used a simplified version of the two-dimensional shallow water equations as the mathematical model for the oceanic wave propagation, where viscosity, friction and Coriolis forces were neglected. The geometry was a simple swimming-pool setting, with an elevated column of water in the middle as initial condition, which generates outward propagating waves. We used simplified mathematical model and geometry because our focus was on the quality of our algorithms. It is more difficult to achieve good performance with low amount of computation per grid cell, than with a higher amount. We chose a discontinuous Galerkin spatial discretization scheme with constant (0^{th} order polynomial) basis functions, which is similar to a finite volume type method. For the time discretization we used an explicit Euler scheme. Our Sierpinski-based grid system provided a conforming triangular grid with dynamically adaptive mesh refinement and coarsening in each time step. Numerical unknowns are strictly cell-based, and information exchange between neighboring cells is strictly edge-based. Nodes are completely eliminated from the grid, effectively simplifying not only the computation traversal, but also the management of the dynamically adaptive grid. All grid traversals – whether for computation or for grid management – work in parallel computing environment using MPI, with optional load-balancing after dynamic adaptivity.

Besides the actual simulation of the shallow water equations, several benchmark tests were performed in order to test the capabilities of the Sierpinski grid management system both in serial and in parallel. The purpose of the tests was to show the feasibility of the grid system for future use, when an improved mathematical model and a higher-order discretization scheme will automatically lead to an even better computational performance, and, perhaps, to sub-realtime simulation.

In the serial tests, for example, it was shown that best floating-point performance can be achieved with high amount of numerical operations per triangle, rather than low amount or no operations. The highest rate of 1 GFLOP/sec achieved by the traversal with 177 artificial FLOPs per triangle is a very good result, while the current DG-SWE traversal with 90 FLOPs per triangle runs with 900 MFLOP/sec. This amounts to 40% floating-point performance of the block-diagonal matrix-vector product. The linearization of the refinement tree is a great performance boost, since the loop-based traversal performs 35% faster than a full recursive tree traversal in our discontinuous Galerkin implementation. The additional memory per triangle needed by the linearization is a fix cost of 16 bytes, which will not change for future implementations with higher-order discretization schemes. The edge-stack system uses optimal amount of memory to facilitate the information flow between neighboring triangle cells. The low amount of memory used for grid management purposes made it possible to perform fully adaptive simulations with as much as 10 million triangles on a laptop computer. The time to perform the adaptive mesh refinement and coarsening – consisting of several traversals – is equivalent of roughly 3.5 DG-SWE traversals in the current implementation.

The parallel performance experiments were meant to show the strong speed-up efficiency of computational- and management traversals, of the full adaptive mesh refinement and coarsening step, and that of the complete simulation of the wave propagation with adaptation in each time step. Except for some unfortunate cases with 64 and 80 processors, where a bug is suspected, the strong speed-up results are quite promising, given that no special effort was spent on parallel performance optimization. The DG-SWE traversal gets an impressive 90% speed-up efficiency on 128 processors, and the consistency traversal is at 85%. In the full adaptive step, where the grid grows by 50% in size, the speed-up efficiency is at 85% on 128 processors. The full simulation with dynamic adaptivity in each time step achieves also an impressive 90% efficiency. Load-balancing incurs a performance drop of about 17% on the Infinicluster, but it is highly dependent on the communication hardware. The effect of load-balancing in special scenarios and on several hardware configurations was not examined thoroughly. The effects of using dynamically changing number of worker threads or GPU threads to be used when the grid grows as a result of adaptive refinement is out of scope of this work.

Additional work needs to be completed in the future in order to achieve sub-realtime simulation performance. First, the mathematical model should be extended to the full shallow water equations, or at least to a more complex version that is adequate for tsunami simulations. Since the equations can be written in vector form, like in Formula 1.2, the solution process would not change significantly. Second, a higher order spatial and temporal discretization is needed, in order to increase the accuracy of the simulation. In a high-order discontinuous Galerkin method the degree of the polynomial basis functions in each triangle cell would need to be increased, but the data access pattern would remain as it is during a computation traversal. A high-order Runge-Kutta method would need one traversal per slope computation, and additional memory per triangle to store the result of the intermediate slopes, in order to complete a full time step. Third, the ocean floor topology with the islands and the coastline geometry needs to replace the simplistic unit-square setting.

Some advantages would be “for free” when the above tasks will be implemented in our Sierpinski grid management system. A more precise mathematical model together with a higher-order spatial- and temporal discretization will need considerably more than 90 floating-point operations per triangle cell during a traversal. By using better algorithms for the simulation, one tries to increase the length of the simulated time step relative to the real computation time needed to calculate it. As we have seen in the artificial FLOP experiments, more FLOP per triangle leads automatically to better MFLOP/sec rates, and, consequently, to a more efficient use of the hardware. Since the high-order Runge-Kutta method needs one traversal per slope, some or all of the consistency traversals of the subsequent adaptive mesh refinement step could be merged into the slope computation traversals. That way, about 30% of the full adaptive step would be intertwined with computation. This means, that the real time for adaptive mesh refinement would decrease by this amount. Currently, an adaptive step costs ~3.5 DG-SWE traversals – or time steps – in execution time. This ratio of dynamic adaptivity relative to a time step will sink considerably, not just because of traversal merging, but also because more execution time will be spent on a time step when computing the intermediate Runge-Kutta slopes. Last, but not least, with more floating-point operations per triangle cell, and per traversal, the already impressive 80-90% parallel speed-up efficiency achieved currently on 128 processors, will likely be sustainable on a growing number of computing units. A sustainable parallel speed-up on one- or two orders of magnitude higher amount of processors is the key to a future sub-realtime tsunami simulation, which, with the current achievements and the aforementioned implicit advantages, looks ever more realistic.

Bibliography

Aizinger, V. and Dawson, C. (2002) "*A discontinuous Galerkin method for two-dimensional flow and transport in shallow water*", Adv. Water Res., 25, pp. 67-84.

Bader, M., Böck, C., Schwaiger, J. and Vigh, C. (2010) "*Dynamically adaptive simulations with minimal memory requirement – solving the shallow water equations using Sierpinski curves*", SIAM J. Sci. Comput., 32(1):212-228.

Bader, M., Rahnema, K. and Vigh, C. (2012) "*Memory-Efficient Sierpinski-order Traversals on Dynamically Adaptive, Recursively Structured Triangular Grids*", Springer Berlin, Lecture Notes in Computer Science, 2012, vol. 7134, p. 302-312 .

Bader, M., Schraufstetter, S., Vigh, C. and Behrens, J. (2008) "*Memory efficient adaptive mesh generation and implementation of multigrid algorithms using Sierpinski curves*", Int. J. of Comput. Sci. and Engineering, 4, pp. 12-21.

Bader, M. and Zenger, C. (2006) "*Efficient storage and processing of adaptive triangular grids using Sierpinski curves*", Computational Science – ICCS 2006, Lecture Notes in Computer Science 3991, pp. 673-680.

Bänsch, E. (1991) "*Local mesh refinement in 2 and 3 dimensions*", Impact Comput. Sci. Eng., 3, pp. 181-191.

Behrens, J. (1998) "*Atmospheric and ocean modeling with adaptive finite element solver for the shallow-water equations*", Appl. Numer. Math., 26, pp. 217-226.

Behrens, J. (2004) "*Adaptive mesh generator for atmospheric and oceanic simulations – amatos*", Technical Report TUM-M0409, Technische Universität München.

Behrens, J. (2005a) "*Adaptive Atmospheric Modeling – Key techniques in grid generation, data structures, and numerical operations with applications*", Habilitationsschrift, Technische Universität München.

Behrens, J. (2005b) "*Adaptive atmospheric modeling – scientific computing at its best*", Computing in Science and Engineering, Vol. 7, No. 4, pp. 76-83.

Behrens, J., Rakowsky, N., Hiller, W., Handorf, D., Läuter, M., Päpke, J. and Dethloff, K. (2005c) "*Parallel adaptive mesh generator for atmospheric and oceanic simulation*", Ocean Modelling, Vol. 10, pp 171-183.

Behrens, J. and Zimmermann, J. (2000) "*Parallelizing an Unstructured Grid*

Generator with a Space-Filling Curve Approach", Euro-Par 2000 Parallel Processing – 6th International Euro-Par Conference Munich, Germany, Lecture Notes in Computer Science, vol. 1900, Springer Verlag, 2000, pp. 815-823.

Böck, C. (2008) "*Discontinuous-Galerkin-Verfahren zum Lösen der Flachwassergleichungen auf adaptiven Dreiecksgittern*", Diplomarbeit, Technische Universität München.

Bungartz, H., Mehl, M., Neckel, T. and Weinzierl, T. (2010) "*The PDE framework Peano applied to fluid dynamics*", Comput. Mech., 46(1).

Burstedde, C., Wilcox, L. C., Ghattas, O. (2011) "*p4est: Scalable Algorithms for Parallel Adaptive Mesh Refinement on Forests of Octrees*", SIAM Journal of Scientific Computing 33 no. 3, pp. 1103-1133.

Cockburn, B., Karniadakis, G. and Shu, C.-W. (2000) "*The development of discontinuous Galerkin methods*", Discontinuous Galerkin Methods: Theory, Computation and Applications, Lecture Notes in Comput. Sci. Eng. 11, Springer, Berlin, New York, 2000, pp. 3–50.

Crouzeix, M. and Raviart, P. (1973) "*Conforming and nonconforming finite element methods for solving the stationary Stokes equation*", RAIRO Anal. Numer., 7, pp 33-76.

Dawson, C., Westernik, J., Feyen, J. and Pothina, D. (2006) "*Continuous, discontinuous and coupled discontinuous-continuous Galerkin finite element methods for the shallow water equations*", Int. J. Numer. Methods Fluids, 52, pp. 63-88.

Demirel, Ö. (2009) "*Parallelization of a Discontinuous Galerkin Solver for the Shallow Water Equation*", Master Thesis, Technische Universität München.

Eskilsson, C. and Sherwin S. (2005) "*Discontinuous Galerkin spectral/hp element modeling of dispersive shallow water systems*", J. Sci. Comp., 22/23, pp. 269-288.

George, D. and LeVeque, R. (2006) "*Finite volume methods and adaptive refinement for global tsunami propagation and local inundation*", Science of Tsunami Hazards, 24, pp. 319-328.

Giraldo, F. (2006) "*High-order triangle-based discontinuous Galerkin methods for hyperbolic equations on a rotating sphere*", J. Comput. Phys., 214, pp. 447-465.

Glimsdal, S., Pedersen, G., Atakan, K., Harbitz, C., Langtangen, H. and Løvholt, F. (2006) "*Propagation of the Dec.* 26, 2004*, Indian Ocean Tsunami: Effects of dispersion and source characteristics*", Int. J. Fluid Mech. Res., 33, pp. 15–43.

Griebel, M. and Zumbusch, G. (1998) “*Hash-storage techniques for adaptive multilevel solvers and their domain decomposition parallelization*”, Contemporary Mathematics, 218:279-286.

Günther, F., Mehl, M., Pögl, M. and Zenger, C. (2006) “*A cache-aware algorithm for PDEs on hierarchical data structures based on space-filling curves*”, SIAM Journal of Scientific Computing, vol. 28, no. 5, pp. 1634-1650.

Harig, S., Chaeroni, Pranowo, W. S. and Behrens, J. (2008) “*Tsunami simulations on several scales: Comparison of approaches with unstructured meshes and nested grids*”, Ocean Dynamics, 58, pp. 429-440.

Haug, A. (2006): “*Sierpinski-Kurven zur speichereffizienten numerischen Simulation auf adaptiven Tetraedergittern*”, Diplomarbeit. Technische Universität München.

Imamura, F., Yalciner, A. and Ozyurt, G. (2006) “*Tsunami Modeling Manual*”, Tsunami Engineering Laboratory, Tohoku, Japan. Website: http://www.tsunami.civil.tohoku.ac.jp/hokusai3/E/projects/manual-ver-3.1.pdf

ITIC – International Tsunami Information Center, “*11 March 2011, MW 9.0, Near the East Coast of Honshu Japan Tsunami*”, an UNESCO/IOC – NOAA partnership. Website: http://itic.ioc-unesco.org/

Kawata, Y. et al., (2005) “*Comprehensive analysis of the damage and its impact on coastal zones by the 2004 Indian Ocean tsunami disaster*”, from TOHOKU University - research was financially supported by Grant-in-Aid for Special Purposes Research Report, Ministry of Education, Culture, Sports, Science and Technology (Grant Number 16800055, Leader Yoshiaki Kawata, Kyoto University). Website: http://www.tsunami.civil.tohoku.ac.jp/sumatra2004/report.html

LeVeque, R. J. (2007) “*Adaptive Mesh Refinement in CLAWPACK and GeoClaw*”, CIG AMR Tutorial. Website: http://www.clawpack.org

LeVeque, R. J., George, D. L., Berger, M. J. (2011) “*Tsunami modeling with adaptively refined finite volume methods*”, Cambridge University Press, Acta Numerica (2011), pp. 211-289.

McCalpin, J.D. (1995) “*Memory Bandwidth and Machine Balance in Current High Performance Computers*”, IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter, December 1995.

McCalpin, J.D. (1991-2007) “*STREAM: Sustainable Memory Bandwidth in High Performance Computers*”, a continually updated technical report (1991-2007), available at: "http://www.cs.virginia.edu/stream/".

Mehl, M., Weinzierl, T. and Zenger, C. (2006) "*A cache-oblivious self-adaptive full multigrid method*", Numer. Lin. Alg. Appl., 13(2-3):275-291

Mitchell, W.F. (1991) "*Adaptive refinement for arbitrary finite-element spaces with hierarchical bases*", J. Comp. Appl. Math. 36, pp. 65-78.

Mitchell, W.F. (1998) "*The Refinement-Tree Partition for Parallel Solution of Partial Differential Equations*", NIST Journal of Research 103, pp. 405-414.

Mitchell, W.F. (2005) "*Hamiltonian Paths Through Two- and Three-Dimensional Grids*", NIST Journal of Research 110, pp. 127-136.

Mitchell, W.F. (2007) "*A Refinement-tree Based Partitioning Method for Dynamic Load Balancing with Adaptively Refined Grids*", Journal of Parallel and Distributed Computing, Vol. 67, No. 4, pp. 417-429.

Neckel, T. (2009) "*The PDE Framework Peano: An Environment for Efficient Flow Simulations*", Dissertation, Technische Universität München.

NOAA – National Oceanic and Atmospheric Administration, "*Japan (East Coast of Honshu) Tsunami, March 11, 2011* ", NOAA/ PMEL / Center for Tsunami Research. Website: http://nctr.pmel.noaa.gov/honshu20110311/

NOAA – National Oceanic and Atmospheric Administration, "*Tsunami Event - December 26, 2004 Indonesia (Sumatra)* ", NOAA/ PMEL / Center for Tsunami Research. Website: http://nctr.pmel.noaa.gov/indo_1204.html

NOAA – National Oceanic and Atmospheric Administration, "*Tsunami Forecasting*", NOAA/ PMEL / Center for Tsunami Research. Website: http://nctr.pmel.noaa.gov/tsunami-forecast.html

Obeidat, A. (2009) "*Parallel Refinement and Coarsening of recursively structured adaptive triangular grids*", Master Thesis, Technische Universität München.

Radzieowski, C. (2007) "*Numerische Simulation zeitabhängiger Probleme auf dynamisch-adaptiven Dreiecksgittern*", Diplomarbeit, Technische Universität München.

Remacle, J.-F., Soares Frazão, S., Li, X. and Shepard, M. (2006) "*An adaptive discretization of shallow-water equations based on discontinuous Galerkin methods*", Int. J. Numer. Methods Fluids, 52, pp. 903-923.

Sagan, H. (1994) "*Space-Filling Curves*", Springer-Verlag, Berlin, New York.

Sampath, R. S., Adavani, S. S., Sundar, H., Lashuk, I. V. and Biros, G. (2008)

"*Dendro: Parallel Algorithms for Multigrid and AMR Methods on 2:1 Balanced Octrees*", Talk at Supercomputing 2008, Austin, Texas.

Schraufstetter, S. (2006) "*Speichereffiziente Algorithmen zum Lösen partieller Differentialgleichungen auf adaptiven Dreiecksgittern*", Diplomarbeit, Technische Universität München.

Schwaiger, J. (2008) "*Adaptive Discontinuous-Galerkin-Verfahren zum Lösen der Flachwassergleichungen mit verschiedenen Randbedingungen*", Diplomarbeit, Technische Universität München.

Stevenson, R. (2008) "*The completion of locally refined simplical partitions created by bisection*", J. Math. Comp. vol.77, (2008), pp. 227-241.

Sundar, H., Sampath, R. S. and Biros, G. (2007a) "*Bottom-up construction and 2:1 balance refinement of linear octrees in parallel*", University of Pennsylvania Technical Report, MS-CIS-07-05.

Sundar, H., Sampath, R. S., Adavani, S. S., Davatzikos, C. and Biros, G (2007b) "*Low-constant Parallel Algorithms for Finite Element Simulations using Linear Octrees*", SC07 November 10-16, 2007, Reno, Nevada, USA.

Titov, V., (2011) "*NOAA Tsunami Forecast Development*", NOAA Center for Tsunami Research, Pacific Marine Environmental Laboratory.

USGS – U.S. Geological Survey, "*Largest Earthquakes in the World Since 1900*". Website: http://earthquake.usgs.gov/earthquakes/world/10_largest_world.php

Vigh, C. (2007) "*Memory-efficient Adaptive Grid Generation using Sierpinski Curves*", Master Thesis, Technische Universität München.

Weinzierl, T. (2009) "*A Framework for Parallel PDE Solvers on Multiscale Adaptive Cartesian Grids*", Dissertation, Technische Universität München.

Zumbusch, G. (2001) "*Adaptive parallel multilevel methods for partial differential equations*", Habilitationsschrift, Universität Bonn.