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

Vectorization and Patch-Based Adaptive Mesh Refinement for Finite Volume Solvers

Cháulio de Resende Ferreira

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(r):	Prof. Gudrun J. Klinker, Ph.D.
Prüfer der Dissertation:	1. Prof. Dr. Michael G. Bader
	2. Prof. Dr. Olaf Schenk

Die Dissertation wurde am 16.07.2019 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 25.08.2019 angenommen.

Contents

Acknowledgements

Special thanks are due to some people without whom this work would not have been possible. To my supervisor Prof. Dr. Michael Bader, for his valuable guidance throughout these four years. To my parents Maria and Mateus, for always supporting me in every possible way. To my wife Thamyrys, for always being by my side in the good and bad moments. And to Prof. Dr. Kyle Mandli, for hosting me at the Columbia University in the City of New York during three months. I am also very grateful for the financial support provided by the Brazilian Council of Technological and Scientific Development (CNPq, grant no. 234439/2014-9) and by the TUM Graduate school, as well as for the resources provided by the Technical University of Munich and by the Leibniz Supercomputing Center.

Abstract

In recent years, the width of the SIMD instructions provided by high performance architectures has increased remarkably. While a decade ago high-end processors would typically provide 128-bit vector instructions, nowadays the latest generations of Intel Xeon and Intel Xeon Phi processors provide the AVX-512 instruction set, which operates simultaneously on 8 double-precision (or 16 single-precision) values. Thus, in order to benefit from the full potential of those architectures, scientific applications need more than ever to consider ways to achieve efficient vectorization, which encourages the implementation of loop-oriented algorithms operating on regular data structures.

On the other hand, large-scale applications often require adaptive mesh refinement (AMR) techniques to efficiently compute accurate solutions, as they allow to selectively apply higher resolutions in regions of interest, or where higher accuracy is desired. Since managing adaptive meshes requires complex algorithms and dynamic data structures, it is especially challenging to efficiently combine adaptivity and vectorization. A compromise between the conflicting desires for regular data structures and for dynamic adaptivity can often be found by using patch-based adaptive meshes, which allow to substantially reduce the computational costs both in terms of time and space while also supporting vectorization. However, achieving efficient vectorization and minimizing the simulation's time-to-solution on patch-based adaptive meshes are not trivial tasks, especially for domain-specific developers who are often not HPC experts and may put less effort in performance engineering.

This thesis investigates vectorization of finite volume solvers in the context of two different PDE frameworks that implement AMR, using two variations of the shallow water equations as example applications. For GeoClaw, a simple rearrangement of the arrays used to store the simulation data was enough to achieve successful vectorization. On the other hand, for the sam(oa)^2 framework it was necessary to replace its inherent cell-wise AMR strategy with a patch-based approach in order to have more regular data structures that support vectorization. In addition, we developed a generalization of the high performance numerical scheme that was implemented for sam(oa)^2, which allows it to be applied to various hyperbolic PDEs with relative ease. That is accomplished via a highly customizable programming interface that transparently manages vectorization, AMR and other HPC features, allowing application developers to focus solely on application-specific decisions and numerical algorithms.

1 Introduction

Most modern processors currently used for large-scale scientific computing applications rely on multiple levels of parallelism to deliver high performance. In addition to supporting shared-memory parallelism over tens to hundreds of processor cores, high-end architectures such as the latest generations of Intel Xeon and Intel Xeon Phi products also provide sets of SIMD instructions that can operate simultaneously on 256-bit or 512-bit words (4–8 double-precision or 8–16 single-precision values). Moreover, such processors are often combined in supercomputing clusters with hundreds or thousands of distributed-memory nodes. Therefore, in order to benefit from the full potential of those high-performance platforms, scientific applications need to be designed taking all of those different levels of parallelism into account. In particular, considering the recent increases in the typical SIMD widths of current processors, such applications should more than ever look for opportunities to apply vectorization (i.e., SIMD parallelism).

In this thesis we investigate strategies to achieve efficient vectorization of finite volume solvers for systems of hyperbolic partial differential equations (PDEs) on discretizations with adaptive mesh refinement. More specifically, we present our work on developing vectorized versions of the finite volume schemes implemented in two different PDE frameworks: GeoClaw [9,28] and sam(oa)^2 [49,63]. Each of these frameworks implements a different strategy for providing adaptive mesh refinement, which is a key feature for large-scale applications that need to efficiently compute accurate solutions, as it allows high resolutions to be selectively applied in specific regions of interest or where the numerical scheme requires higher accuracy. Since adaptive mesh refinement is generally implemented with dynamic data structures and, on the other hand, vectorization requires regular data structures, coupling both features in an optimal way is not a trivial task and requires careful choice of algorithms and data structures.

For each framework, achieving successful vectorization of the numerical routines required different modifications. As GeoClaw uses regular grid patches as building blocks for its adaptive mesh refinement implementation, its mesh structure was already appropriate for SIMD parallelism. Nevertheless, achieving efficient vectorization still required a rearrangement of the arrays used in the finite volume scheme, as the original layout was not suitable for vectorization. On the other hand, sam(oa)^2 required much more implementation effort, because it was originally designed for fine-grained cell-wise adaptivity, which was an obstacle for straightforward vectorization. This was addressed by replacing the inherent cell-wise adaptivity in sam(oa)^2 with a patch-based discretization that represents a fair balance between the conflicting needs for regular data structures (to support vectorization) and for dynamic data structures (to allow adaptivity). After those modifications in both frameworks, vectorization of the main solver loop was achieved by reorganizing the input data of the so-called Riemann problems into temporary arrays with vectorization-friendly layouts and by employing directives to assist on auto-vectorization by the compiler.

In addition to our main goal of improving the performance of the finite volume solvers in GeoClaw and sam(oa)^2, we investigate, as a secondary goal, ways to simplify the process of implementing new simulation scenarios in sam(oa)^2 – the fact that developing a new scenario requires too complicated implementations was one of the main shortcomings of the framework, as stated in the concluding remarks of the previous work that served as a starting point for this thesis [49]. With that goal in mind, after implementing efficient patch-based adaptive mesh refinement and vectorization approaches in sam(oa)^2 we also developed a generalization of our high-performance finite volume schemes that allows easy customization for various other systems of hyperbolic PDEs. That is accomplished via a programming interface that transparently manages the adaptive meshes and other HPC features in sam(oa)^2, requiring application developers to deal solely with application-specific decisions and numerical algorithms.

We evaluate the performance of all our implementations on two modern Intel processors: Intel Xeon "Haswell" and Intel Xeon Phi "Knights Landing" (KNL). While the former provides a set of 256-bit SIMD instructions, the latter supports 512-bit SIMD instructions, which allows us to assess the speedups achieved by vectorization on two different SIMD widths. Additionally, although we focus on vectorization performance, all our experiments also consider shared-memory parallelism, such that we strive to extract the maximum performance from each of those many-core architectures.

In our implementations and experiments we consider finite volume solvers for oceanic simulations based on two variations of the shallow water equations, which we use as example applications both for GeoClaw and for sam(oa)^2: the single-layer and the two-layer shallow water equations. In addition to being used for evaluating the performance of our implementations, these two systems of hyperbolic PDEs are also used in many implementation examples throughout this thesis and serve as example applications to demonstrate the flexibility and usability of the programming interface that was developed for sam(oa)^2.

1.1 Content Overview

In the following four chapters, we discuss various aspects relevant to the work described in this thesis. In Chapter 2 we explain basic concepts related to vectorization and discuss what is usually necessary to achieve efficient compiler auto-vectorization. Chapter 3 concerns adaptive mesh refinement, with a focus on the two different strategies for implementing adaptive meshes that are used by GeoClaw and sam(oa)^2 and the challenges involved in combining adaptive mesh refinement with vectorization. In Chapter 4 we give an overview of the relevant numerical background, including basic theory of hyperbolic PDEs, finite volume methods, Riemann solvers and the two variations of the shallow water equations that are used as example applications in this work. Then, in Chapter 5 we describe the high-performance platforms that we use for all performance experiments presented in this thesis.

The main contributions achieved by this work are then presented in the subsequent chapters. In Chapter 6 we describe our work on adding vectorization to the numerical routines of the GeoClaw package, with special focus on the Riemann solvers. Chapter 7 presents on our work on sam(oa)^2, in which we first needed to implement a new patch-based discretization in order to apply the same vectorization approach that was used for GeoClaw. Then, in Chapter 8 we describe the generic programming interface that was created for sam(oa)^2, which allows easy customization of the patch-based implementation that was developed in the previous chapter. In each of those three chapters we include a section with an experimental performance analysis of the developed implementations. Finally, in Chapter 9 we give a summary of our main contributions and list suggestions for future improvements.

Notes On Source Code Examples

In this thesis, we often illustrate our implementations with source code examples. Since both frameworks considered in our work have been implemented in Fortran, all source code examples are given in that programming language. However, we note that the code snippets shown here are only used for illustration purposes and are not meant to be exact representations of the actual code used in the frameworks and in our implementations. Therefore, in many cases the code contained in those examples has been modified with the purposes of facilitating their understanding and of highlighting the implementation details that are most relevant to the respective discussion. For instance, we often omit pieces of the code that are not important for the respective example and that could potentially cause confusion to the reader. When that happens, it is usually noted by "`[...]`" passages in the code snippets. In addition, we note that variables declared as `real` in those examples always refer to double-precision floating-point variables, even if that is not explicitly stated in the codes.

2 SIMD Parallelism

SIMD stands for *single instruction multiple data*, a paradigm for data-level parallelism that consists in applying a same operation simultaneously to multiple data elements. The process of implementing an algorithm such that it can effectively use SIMD instructions in its execution is often referred to as *vectorization*.

Modern high-performance architectures provide sets of SIMD instructions that can operate on relatively wide vector registers. For example, one of the experimental platforms we use in this work (which we describe more detailedly in Chapter 5) provides 512-bit SIMD instructions, which can operate on up to eight double-precision or sixteen single-precision floating-point variables within one processor cycle.

There are multiple options for implementing vectorization on architectures with SIMD instructions. In this work we focus on *compiler auto-vectorization*, which is the most portable and in many cases the simplest way to achieve vectorization. Other options include for example assembly-level implementations or intrinsic functions. However, these are not addressed here, as we found that compiler auto-vectorization was sufficient for our needs.

2.1 Compiler Auto-Vectorization

The simplest option for achieving vectorization is to use the compiler to convert high-level source code into assembly code that uses SIMD instructions – this process is often referred to as *auto-vectorization*. Assuming that the source code has been implemented in a way that is suitable for vectorization, developers often only need to enable auto-vectorization in the compilation settings in order to use this approach[1]. In some cases, however, further work is necessary to assist the compiler in the vectorization process, as we discuss further later.

When auto-vectorization is enabled, the compiler inspects all loops in the code, checking whether they can be safely and efficiently executed with SIMD instructions and, if that is the case, converts them into vectorized assembly code. A very simple example of *vectorizable* code consists of a loop performing a single arithmetic operation over a set of arrays, as we show in Code 2.1. In this example we show a loop that computes `c = a + b` for all 10 positions of three arrays `a`, `b` and `c` and can be easily auto-vectorized by the compiler. Assuming double-precision arithmetic and a processor that performs 256-bit SIMD instructions, each SIMD instruction is able to operate simultaneously on up to four positions of each array. Thus, while a *serial* implementation requires 10 arithmetic instructions to execute this loop, a *vectorized* loop is able to do the same job using only three SIMD instructions – this is illustrated in Fig. 2.1.

In addition to being able to vectorize simple loops like the one shown in the example above, modern compilers can often vectorize considerably more complex loops, like the ones we will address in this thesis. However, in order to be vectorizable, a loop needs to follow a set of

[1]For the Intel Fortran Compiler used in this work, auto-vectorization is enabled by default for optimization levels `-O2` or higher.

Code 2.1: Simple loop that can be easily auto-vectorized by the compiler.

```fortran
real :: a(10), b(10), c(10)

! [...] (Initialize arrays 'a' and 'b')

! The compiler can easily auto-vectorize this loop:
do i=1,10
    c(i) = a(i) + b(i)
end do
```

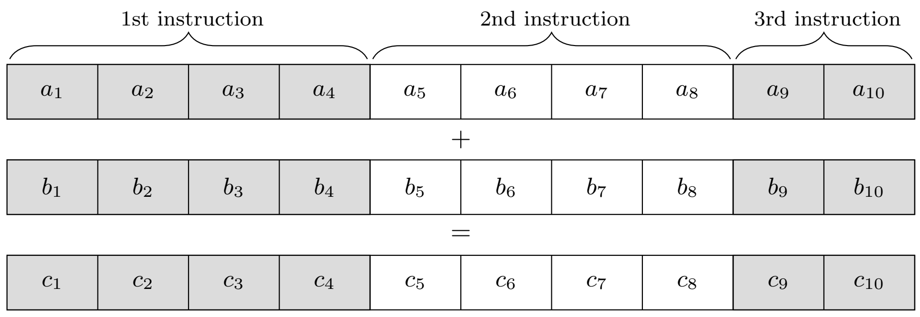

1st instruction
2nd instruction
3rd instruction
a_1 a_2 a_3 a_4 a_5 a_6 a_7 a_8 a_9 a_{10}
+
b_1 b_2 b_3 b_4 b_5 b_6 b_7 b_8 b_9 b_{10}
=
c_1 c_2 c_3 c_4 c_5 c_6 c_7 c_8 c_9 c_{10}

Figure 2.1: Vectorization of a simple loop that computes `c = a + b` on three arrays with 10 values each (Code 2.1). Here we assume that the processor's SIMD registers can fit up to four of such values, so that the vectorized loop only needs three SIMD instructions to process those arrays. In the figure, we highlight the operands of each SIMD instruction with white/gray background colors.

specific guidelines [33]. In the following, we give a few examples of obstacles that prevent compiler auto-vectorization:

- early exits from the loop;
- data dependencies across iterations;
- calls to external libraries;
- non-vectorizable instructions (e.g., I/O instructions).

Additionally, there are other factors that may become obstacles for straightforward vectorization, but sometimes can still be successfully vectorized by modern compilers. These include:

- if-then-else branches;
- non-contiguous memory accesses;
- unaligned arrays;
- loops with non-unit strides.

Modern compilers can sometimes perform additional code transformations in order to successfully vectorize such codes. E.g., compilers often vectorize loops containing if-then-else branches by applying *masked instructions*, and may restructure the data in the vector registers to allow vectorization of operations on non-contiguous and/or unaligned data. However, that comes at the expense of performance, since those extra operations can introduce considerable overhead to the vectorized loop and should therefore be avoided.

Code 2.2: Example of *structure of arrays* (SoA) layout. Since consecutive iterations of this loop access contiguous memory positions, this loop can be efficiently vectorized.

```fortran
type t_rectangles
    real :: widths(N)
    real :: heights(N)
    real :: areas(N)
end type

type(t_rectangles) :: rectangles

! [...] (Initialize arrays ‘width’ and ‘height’ in ‘rectangles’)

! Loop with unit stride accesses to memory
do i=1,N
    rectangles%areas(i) = rectangles%widths(i) * rectangles%heights(i)
end do
```

Code 2.3: Example of *array of structures* (AoS) layout. Here consecutive loop iterations do not access contiguous memory positions, so this loop is not suitable for vectorization.

```fortran
type t_rectangle
    real :: width
    real :: height
    real :: area
end type

type(t_rectangle) :: rectangles(N)

! [...] (Initialize fields ‘width’ and ‘height’ of all elements in ’rectangles’)

! Loop with non-unit stride accesses to memory
! (e.g., rectangles(i)%area is not contiguous to rectangles(i+1)%area)
do i=1,N
    rectangles(i)%area = rectangles(i)%width * rectangles(i)%height
end do
```

2.2 Data Layouts Suitable for Vectorization

When dealing with data structures more complex than simple one-dimensional arrays, developers must carefully choose their data layout in order to support efficient vectorization. It is especially important to organize the data such that accesses to the arrays have unit stride across consecutive loop iterations. This can usually be achieved by using the so-called *structure of arrays* (SoA) layout, in contrast to the *array of structures* (AoS) layout that is also used in many applications. In Code 2.2, we show a vectorizable loop that operates on data organized with an SoA layout, while in Code 2.3 we show an analogous implementation using an AoS layout. While the former code operates on data stored contiguously, the latter performs non-unit stride accesses to the memory, and can therefore not be efficiently vectorized.

Also when using arrays with multiple dimensions it is necessary to carefully choose their layout with respect to the ordering of the array's indices. As before, the goal is to support vectorization by performing only unit stride accesses across consecutive iterations. That can be achieved by choosing the data layout such that the main (innermost) loop iterates over the index for which consecutive positions are stored contiguously in memory. For programming languages that store multi-dimensional arrays using column-major order like Fortran, that means that vectorized loops should iterate over the leftmost array index.

In Code 2.4 we show a vectorizable loop operating on a two-dimensional array – note that

Code 2.4: Vectorizable loop that operates on a two-dimensional array. Vectorization is only possible because the data layout defined by the order of the array indices resembles the SoA layout, i.e., the loop iterates over the leftmost index.

```fortran
! Here we use positions (:,1) for widths, (:,2) for heights and (:,3) for areas
real :: rectangles(N,3)

! [...] (Initialize positions rectangles(:,1) and rectangles(:,2) )

! Loop with unit stride accesses to memory
do i=1,N
    ! areas          = widths          * heights
    rectangles(i,3) = rectangles(i,1) * rectangles(i,2)
end do
```

this code is very similar to the two examples discussed above. This code can only be efficiently vectorized because the loop iterates over the leftmost index of the array declared as `rectangles(N,3)`, such that consecutive iterations access positions that are stored contiguously in memory. Conversely, if this array had been declared as `rectangles(3,N)`, two consecutive loop iterations would access array positions `(1,i)` and `(1,i+1)` that are not stored contiguously, preventing efficient vectorization.

2.2.1 Data Alignment

In addition to requiring an appropriate layout for the arrays used in vectorized loops, efficient vectorization also depends on proper *alignment* of those arrays in memory. In order to maximize vectorization efficiency, arrays should be aligned to a multiple of the register SIMD width – e.g., for 512-bit instructions, their starting address should lie on a 64-byte boundary. If that is not the case, the compiler is not able to operate on the data as efficiently as if it was properly aligned, because it needs to use unaligned load/store instructions and to generate an extra "peel loop", which precedes the main loop operating on the array elements that are stored before the required alignment boundary [32].

While data alignment in Fortran can be configured individually for each array via the compiler directive `!DIR$ ATTRIBUTES ALIGN`, the Intel Fortran Compiler used in this work also provides the options `-align array32byte` and `-align array64byte`, which can be used to align all arrays (except those in `COMMON` blocks, which we do not use in this work) to 32-byte or 64-byte boundaries, respectively. Due to their simplicity, we use these compiler options in all our implementations.

In the case of arrays with two dimensions, efficient vectorization requires proper alignment not only of the first element in the arrays, but also of the first element in each of their columns (for column-major languages like Fortran). Consider again the example in Code 2.4: although vectorization is possible because the loop operates on data stored contiguously, the alignment of the array elements on which the loop operates may not be ideal depending on the parameter `N`. In this example, proper alignment is necessary for the elements of the array `rectangles` stored at positions `(1,1)`, `(1,2)` and `(1,3)`, which are accessed in the first loop iteration. That will only be possible if, in addition to the array `rectangles` being aligned, the array dimensions are defined such that the memory required to store each column is also a multiple of the register SIMD width – which can usually be accomplished by padding such arrays, if necessary [34].

2.3 Compiler Directives to Assist Auto-Vectorization

One of the most complex tasks performed by the compiler when considering whether to auto-vectorize a loop is the search for data dependencies across iterations. While compilers can often correctly detect existing dependencies, in many cases they are unable to guarantee that there are no dependencies, especially for loops with very complex codes. When that happens, they take the conservative approach and do not vectorize such loops, to avoid the risk of generating executables that compute incorrect results.

Besides checking if a loop can be safely vectorized, compilers usually also consider whether vectorization would be really beneficial for its performance [31]. For that they estimate the costs of executing the loop with scalar and with SIMD instructions, and use these estimates to decide whether to vectorize it. However, these estimates can sometimes be inaccurate, leading the compiler to not vectorize loops that would perform faster if vectorized (or inversely, to vectorize loops that would perform faster without vectorization).

In cases where developers are not satisfied with the decisions made by the compiler regarding vectorization, they can intervene and provide additional instructions, using compiler directives that aid auto-vectorization. Here are some examples of such directives supported by the Intel Fortran compiler, which should be placed immediately before the loop constructs:

- `!DIR$ NOVECTOR` directive: instructs the compiler to never vectorize the loop;
- `!DIR$ IVDEP` directive: instructs the compiler to ignore *unproven* dependencies. However, the compiler will still not vectorize the loop if it is able to find *proven* dependencies;
- `!DIR$ VECTOR ALWAYS` directive: instructs the compiler to ignore the efficiency analysis (but not the dependency analysis);
- `!DIR$ VECTOR ALIGNED` directive: can be used to assert the compiler that all arrays referenced in the loop are properly aligned;
- `!DIR$ SIMD` directive: instructs the compiler to ignore both the dependency analysis and the efficiency analysis and always vectorize the loop (provided that it is possible to do it). When using this directive, developers are responsible for guaranteeing that there are no data dependencies, as the resulting executable may compute incorrect results otherwise.

In addition, the OpenMP standard (4.0 and later versions) provides the `!$OMP SIMD` directive, which was designed to behave like the `!DIR$ SIMD` directive described above [58]. However, in our work we experienced that the OpenMP directive can achieve better results, as it is sometimes able to vectorize more complex loops than the other. Therefore, in our implementations we used the `!$OMP SIMD` variation whenever necessary.

2.3.1 Vectorizing Loops with Calls to Other Subroutines

Vectorizing a loop containing calls to different functions/subroutines can be achieved by declaring them as *SIMD-enabled* with the `!$OMP DECLARE SIMD` directive. This instructs the compiler to generate a vectorized version of the subroutine (in additional to the serial one), which is then used if the subroutine gets called from inside a vectorized loop.

Alternatively, subroutines can often be *inlined* into the loop – in that case, their executable code becomes part of the loop itself, so that it is also considered by the compiler when applying auto-vectorization to the loop. While in many cases subroutines are automatically inlined by the compiler (depending on their structure and on the compiler optimization options), developers can force the compiler to inline a specific subroutine using the directives `!DIR$ FORCEINLINE` or `!$OMP FORCEINLINE`.

In this work, we experienced much higher vectorization performances with the inlining approach, in comparison to declaring the subroutines as SIMD-enabled – inlining usually delivers better performance not only because it eliminates the overhead of subroutine calls, but also because it allows the compiler to optimize the inlined subroutine specifically for the current loop [5]. Thus, the implementations and experiments discussed in this thesis do not consider the SIMD-enabled approach, as we always use the `FORCEINLINE` directives to get subroutine calls inlined into vectorized loops (and vectorized as well).

In Section 6.2.2, we give more details about the approach we used to achieve auto-vectorization of the main solver loop both in GeoClaw and in sam(oa)^2. In addition, we also compare our implementation with other approaches used in related work in Section 6.2.3.

3 Adaptive Mesh Refinement

Scientific applications often deal with multi-scale domains where high resolutions are desired for certain regions, while lower resolutions are sufficient for other regions. Oceanic applications like the ones we consider in this thesis are a typical example: properly modeling small-scale bathymetry features along a coast may require high resolutions on the scale of meters, while the simulation domains often refer to oceans that could be as large as thousands of kilometers. Modeling entire oceans with scales of meters is clearly unnecessary, and may even be impossible depending on the computational resources available. In fact, on some regions of the ocean it is often enough to use resolutions on the scale of several kilometers, as waves very far from the coast may have wavelengths of more than 100 km [39]. However, as the waves approach the shore, higher resolutions become necessary to more accurately model their behavior.

Such applications usually benefit from implementing *adaptive mesh refinement* strategies, which allow to dynamically increase or decrease the resolution applied at specific regions, depending on the current simulation state or on other properties (e.g., geological properties) of those regions. Although managing adaptive meshes usually introduces considerable overhead due to the complex algorithms and data structures involved, they are almost always worthwhile for simulations with multi-scale domains, since they are able to substantially lower the computational costs both in time and space.

For instance, in one of the experiments we describe in Section 7.3 we performed a high-resolution simulation using an adaptive mesh whose maximum size reached approximately 34 million cells, while applying the same maximal resolution using a static mesh with uniform refinement would have required more than 17 billion cells – a reduction in the mesh size by a factor of roughly five hundred. Such drastic reductions on the number of unknowns and, as a consequence, on the computational costs are likely to compensate for any overhead (in the execution time and/or in the memory required per cell) that may have been introduced due to management of adaptive meshes.

3.1 Strategies for Adaptive Mesh Refinement

Existing PDE frameworks with adaptive mesh refinement use meshes with characteristics that can differ considerably from each other – they may be composed of cells with different geometries, may be structured or unstructured, may implement various strategies for data storage and traversal schemes, etc. As such, adaptive mesh refinement implementations may also vary considerably, depending not only on the mesh structures being considered but also on other design decisions. Therefore, although many other strategies and implementations can be found in literature, here we focus on the strategies adopted by GeoClaw and sam(oa)^2, the two PDE frameworks considered in this thesis. Note that in this chapter we do not discuss implementation details of those two frameworks, because these will be provided later in this thesis – more specifically, in Section 6.1 for GeoClaw and in Section 7.1 for sam(oa)^2.

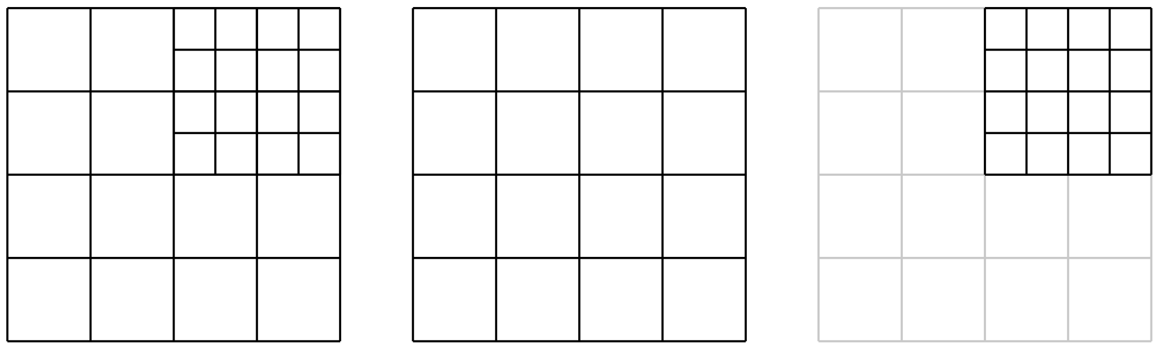

(a) Adaptive mesh **(b)** First level **(c)** Second level

Figure 3.1: Example of a multi-level hierarchical adaptive mesh with two refinement levels. The first level covers the entire domain, while patches in the second level overlap some of its regions. In a larger example, regions of the second level could have been further refined to form a third level, and so on. The complete adaptive mesh is obtained by combining all refinement levels.

3.1.1 Multi-Level Hierarchical Adaptive Meshes

Adaptive mesh refinement is often implemented by combining multiple static grids (usually referred to as *blocks* or *patches*) into a dynamic adaptive scheme. That is the case for GeoClaw, which organizes multiple patches following the Berger-Oliger-Collela [8,10] multi-level approach: a coarse grid is used for the entire simulation domain, while extra overlapping grid patches with finer refinements are generated for regions where higher resolution is desired. This process can be recursively repeated for each grid patch to achieve even higher resolutions, leading to a multi-level hierarchical structure where each refinement level is composed of grid patches with finer resolutions than the previous level. As an example, in Fig. 3.1 we show an adaptive mesh composed of two refinement levels.

An advantage of this approach is the fact that it requires relatively simple data structures and algorithms to manage adaptive mesh refinement. Also, because static grid patches are used as building blocks for the adaptive mesh, numerical algorithms can be designed similarly as they would be for a static mesh with uniform refinement. Nevertheless, managing adaptive meshes using this approach still requires careful implementation of interpolation techniques to perform refinement/coarsening of cells and to properly handle the interfaces between patches located in different refinement levels, in order to avoid introducing numerical errors in the simulation [39].

In addition, regular grid patches are also advantageous for performance reasons, especially when their data are stored as simple arrays: such data structures usually support the implementation of efficient loop-based algorithms, which in many cases can be further optimized by applying vectorization – this will be discussed further in Section 3.2.

On the other hand, this approach can often be less effective on reducing the number of unknowns in the simulation, when compared to other approaches for adaptive mesh refinement. First, because of its design with overlapping grids on the refined regions, which causes the simulation to unnecessarily store and handle unknowns in those regions more than once. And second, because it can often unnecessarily refine cells for which refinement is not required, if they are located close to other cells for which refinement has been requested. This happens because such implementations usually identify clusters of cells flagged for refinement and apply refinement to all cells in each cluster. While many frameworks like GeoClaw use a heuristic like the one presented in [7] to reduce the number of unnecessary refinements, this is still not as effective on

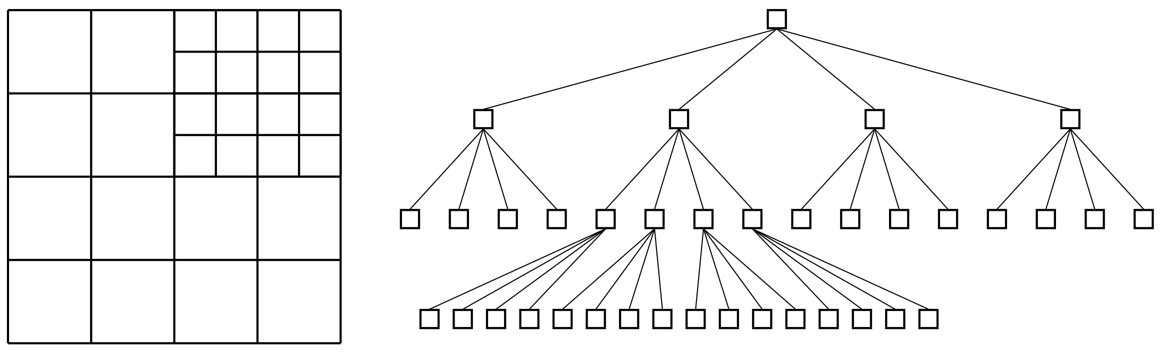

(a) Adaptive mesh **(b)** Refinement tree

Figure 3.2: Example of a tree-structured mesh based on quadtree-like refinements and its respective refinement tree. The leaves of the refinement tree correspond to the actual cells in the mesh, while non-leaf nodes correspond to cells that have been further refined. Note that the adaptive mesh in this example has exactly the same structure as the one shown in Fig. 3.1(a).

minimizing the number of unknowns as other approaches that employ finer-grained schemes.

Several other frameworks use the same or very similar strategies for adaptive mesh refinement as GeoClaw. Examples include Uintah [42, 66], SAMRAI [30], Chombo [16], AMROC [17] and FLASH [20]. For a survey on some of these and other similar frameworks, we recommend [19]. While the strategies implemented in these frameworks are based on the strategy employed by GeoClaw, a few design differences may be found in some of them. For instance, although Uintah previously used the same heuristic [7] as GeoClaw to define the regions to which refinement is applied, this has been replaced with a different one that avoids global communication and presents much better scalability on tens of thousands of cores [41].

3.1.2 Tree-structured Adaptive Meshes

The strategy for adaptive mesh refinement implemented on sam(oa)^2 uses an alternative approach based on tree-structured meshes that are generated via recursive subdivisions of the simulation domain, similarly to (or often based on) quadtrees/octrees. In Fig. 3.2 we show an example of a tree-structured adaptive mesh (and its refinement tree) based on quadtree-like refinements, i.e., rectangular cells can be recursively refined into four finer cells. This strategy leads to domain discretizations composed of non-overlapping cells with varying refinements, therefore avoiding redundant work on the refined regions. While this approach may require more complex algorithms and data structures compared to multi-level approaches, it can often be more successful at reducing the number of elements in the adaptive mesh, not only because it does not rely on overlapping cells, but also because it supports more fine-grained refinement schemes that are usually able to prevent unnecessary refinements.

Tree-structured approaches for adaptive mesh refinement are used by several frameworks, differing from each other in various implementation details. Some of them offer cell-wise adaptivity (i.e., refinement is decided individually for each cell), like e.g. sam(oa)^2, Peano [73], p4est [13], Dendro [64] and amatos [6]. On the other hand, other frameworks implement patch-based approaches, like Daino [71], WaLBerla [22], ForestClaw [12] and PeanoClaw [67]. In fact, patch-based implementations for tree-structured adaptive meshes often represent a fair balance between effectiveness of adaptive mesh refinement and performance of the numerical solvers. Therefore, replacing cell-wise adaptivity with patch-based adaptivity on tree-structured adaptive meshes has become attractive as a means to increase their efficiency. For example, in our work with sam(oa)^2 we replaced cells with regular patches in the leaves of the refinement tree, which allowed us to apply vectorization when processing each patch – this motivation is discussed further in Section 3.2 and our implementation is described in detail in Section 7.2. A similar strategy has also been developed for Peano [67, 74], where clusters of small patches with same refinement are identified and fused into larger patches on-the-fly.

Linearization via Space-Filling Curves

Many frameworks based on tree-structured adaptive meshes (e.g., sam(oa)^2, Peano, p4est) use linearizations of the mesh elements based on so-called space-filling curves for various purposes. For instance, parallelization and load balancing implementations based on space-filling curves are relatively popular, because they can provide a one-dimensional domain decomposition for multi-dimensional adaptive meshes. Furthermore, space-filling curves can also be used to eliminate the need for explicitly storing the complete refinement tree, by storing only its leaves (which correspond to the actual cells/patches in the mesh) following the order in which the curves traverse them.

In addition to the purposes described above, some frameworks like Peano and sam(oa)^2 also use space-filling curves to implement memory- and cache-efficient traversal schemes over their mesh elements. The main difference between the approaches used in these two frameworks is the geometry of the cells. On the one hand, Peano uses square or cubic cells that can be split equally in each dimension and organizes them following the Peano space-filling curve. On the other hand, sam(oa)^2 uses triangular cells that can be recursively split in half, producing meshes that can be inherently organized according to the Sierpinski space-filling curve. The main advantage of the approach used by sam(oa)^2 in comparison to Peano is that in its discretization scheme it is always possible to completely avoid hanging nodes, i.e., vertices that split cells only on one side of an edge and usually require special treatment when applying the numerical scheme. However, this comes at the cost of performing additional refinements, which increases the number of cells in the mesh. On the other hand, the Sierpinski-based adaptive meshes in sam(oa)^2 are limited to adaptivity in two spatial dimensions, while Peano supports fully adaptive simulations over three or more dimensions.

3.2 Vectorization on Adaptive Meshes

As discussed in Chapter 2, vectorization requires loop-based algorithms operating on regular data structures (most commonly, simple arrays). When dealing with applications that provide adaptive mesh refinement, that is not always the case, as considerably complex algorithms and data structures are often used. In particular, implementing vectorization on frameworks that provide cell-wise adaptive mesh refinement can be quite challenging, since in such implementations the mesh elements are rarely stored in regular data structures. For instance, sam(oa)^2 implements an element-oriented design in which cells and edges are processed individually, without access to the data of other elements in the mesh. In addition, the memory- and cache-efficient traversals in sam(oa)^2 are based on stack and stream data structures, which further complicates the implementation of vectorization approaches.

On the other hand, such complications are in general not present for frameworks that use static patches as building blocks for their adaptive scheme, as the data of each patch is usually stored in regular data structures that in many cases can be adapted to support vectorization. Still, we found that achieving efficient vectorization requires careful implementation and choice of data structures, as will be shown in Section 6.2, where we present our approach for vectorizing the numerical routines of GeoClaw.

Considering this, an interesting option for applying vectorization to tree-structured adaptive meshes with cell-wise adaptivity is to replace this scheme with a patch-based approach. That can usually be accomplished by using the leaves of the refinement tree to store regular patches instead of single cells, as done e.g. in ForestClaw and as we did in this work for sam(oa)^2 – this is discussed further in Section 7.2. By switching to a patch-based approach, such frameworks can usually store the data of each patch using regular data structures and efficiently implement vectorization on the algorithms used to process each patch.

3.3 Influence of Patch Sizes

Applications that combine adaptive mesh refinement with regular data structures (for vectorization and in general more efficient algorithms) need to be designed striving to find a balance between both strategies, since one can often invalidate the benefits of the other. This design conflict usually concerns the size of the grid patches being applied. On the one hand, small patches lead to fine-grained adaptivity that minimizes the overall computational work required, but hinders efficient algorithms and vectorization and may also increase the overhead for handling the boundaries between different patches. On the other hand, larger patches foster vectorized and more efficient loop-based algorithms, but at the same time reduce the effectiveness of adaptive mesh refinement.

As mentioned before, strategies for adaptive mesh refinement based on multi-level hierarchical adaptive meshes usually apply coarse-grained refinement schemes – this is useful to minimize the overhead necessary to handle boundaries between patches in different refinement levels, which can be considerably more expensive than handling patches in the same refinement level due to more complicated interpolation schemes required to treat hanging nodes. Nevertheless, frameworks are often able to reduce the size of the patches being used in each level by subdividing them into smaller ones. For instance, GeoClaw's current implementation splits large patches by imposing an upper limit of 60×60 cells in each patch – in practice, we observed that patches in GeoClaw generally have 900–3600 cells each. While this strategy does not solve the problem of having unnecessarily refined cells in the patches, it can be useful for improving cache efficiency and creating opportunities to apply parallelism across the multiple patches.

In contrast, frameworks that use patch-based tree-structured adaptive meshes are usually able to apply finer-grained schemes and have much smaller patches, reducing the computation work required for a given solution accuracy. For example, our recently implemented patch-based discretization in sam(oa)^2 supports patches with only four cells each, and two-dimensional simulations with Peano [67, 74] support patches with nine cells. While very small patches are probably not the best choice due to the reasons mentioned previously, supporting such a fine-grained refinement scheme is advantageous because it allows to experiment with varying patch sizes in order to find a setup that minimizes the execution time for each application.

We address these design conflicts further in the context of sam(oa)^2 in Section 7.3, where we experimentally assess how the choice of patch size influences various factors in the simulations, like the simulation throughput, the total amount of computational work performed and, as a consequence of both, the time-to-solution required by the simulations. In those experiments, we observe that minimizing the time-to-solution for a given accuracy in our simulations with sam(oa)^2 is usually accomplished by applying patches with 64–256 cells each, while using patches with 1024 cells or more is counter-productive due to considerable increases in the number of unknowns in the mesh.

As a final remark, we note that small patches in tree-structured meshes can also lead to more hanging nodes that require more complicated interpolation schemes, similarly to what happens with multi-level adaptive meshes like GeoClaw. However, hanging nodes are not an issue for sam(oa)^2 – as discussed previously, while it is not possible to completely avoid hanging nodes in adaptive meshes based on Cartesian grids, sam(oa)^2 is able to guarantee conformity of its triangular meshes by performing additional refinements in its cell-wise adaptivity scheme. In addition, the new patch-based discretization that we present in this thesis for sam(oa)^2 is also able to completely avoid hanging nodes – that is accomplished by forcing all patches in the mesh to have exactly the same structure and the same number of nodes in each of their boundaries.

4 Hyperbolic Partial Differential Equations and Finite Volume Methods

This chapter gives an overview of the mathematical concepts relevant to this work. Most of the theory described here is based upon [38, 39, 45], where the interested reader can obtain much more complete and detailed discussions. Here we cover the general theory of hyperbolic systems of partial differential equations and the numerical methods typically used for their solution. We also introduce and discuss numerical solutions for the two hyperbolic problems we use as example applications throughout this thesis: the single-layer and two-layer shallow water equations.

4.1 Notation

In this thesis we use mathematical notations similar to what can be found in related literature, especially on the three references mentioned above. For readers not used to those notations, the following notes may be useful.

In particular, we use subscripts to denote partial derivatives, e.g., $q_t \equiv \partial q/\partial t$. However, subscripts do not always indicate partial derivatives – for instance, r_p is used to denote the p-th component in a vector r. Nevertheless, in this thesis partial derivatives are always used in the context of the variables x, y or t, so it should be easy to identify from context whether the subscript indicates a partial derivative or something else. In cases where we denote a partial derivative of an unknown whose symbol already contains a subscript, we add parentheses around that symbol to make the notation clearer, e.g., $(h_1)_x \equiv \partial h_1/\partial x$.

Furthermore, we use superscripts for various purposes, like for example in $Q_i^{(n)}$, where n denotes the current simulation time step. In order to make a clear distinction from cases where superscripts are used to denote exponential operations (e.g., h^2) or domains with multiple dimensions (like $\mathbb{R}^m$), we always include parentheses in the superscripts when they are used for a non-standard purpose, as in the example shown above.

4.2 Hyperbolic Partial Differential Equations

The two frameworks considered in this work deal with hyperbolic systems of partial differential equations (PDEs), which can be used to model a wide range of phenomena involving wave propagation and transport of substances. In the simplest case, we have a one-dimensional homogeneous system with the form

$$q_t(x,t) + f(q(x,t))_x = 0, \tag{4.1}$$

where $q \in \mathbb{R}^m$ is a vector of m unknowns, and the *flux function* $f : \mathbb{R}^m \to \mathbb{R}^m$ is specific for each hyperbolic problem. Homogeneous systems like this are often called *conservation laws* [38].

Optionally, a *source term* $\psi(q,x,t) \neq 0$ can be added to the right hand side of the equations, transforming them into non-homogeneous *balance laws.*

If the flux function f is differentiable, (4.1) can be rewritten in a *quasilinear form*

$$q_t + f'(q)q_x = 0. \tag{4.2}$$

The system is then said to be hyperbolic if the Jacobian matrix $A = f'(q)$ is diagonalizable, i.e., it has real eigenvalues and a corresponding set of m linearly independent eigenvectors. Solutions to such hyperbolic systems are often obtained by rewriting (4.2) as a system of m independent linear equations, each with the same form as the so-called *advection equation*:

$$q_t + \bar{u}q_x = 0. \tag{4.3}$$

The advection equation models the passive movement of a given substance with concentration or density $q(x,t)$ being carried along with a flow of constant speed $\bar{u}$. Its solution can be shown to be

$$q(x,t) = q(x - \bar{u}t, 0), \tag{4.4}$$

meaning that the initial concentration profile $q(x,0)$ is simply advected at a constant speed $\bar{u}$ along with the flow.

It is possible to rewrite the quasilinear form (4.2) as a system of independent advection equations by computing the eigenvalues $\lambda_1, \lambda_2, \ldots, \lambda_m$ and eigenvectors $R = [r_1, r_2, \ldots, r_m]$ of A and then writing $A = R\Lambda R^{-1}$, where Λ is the diagonal matrix containing the eigenvalues, i.e., $\Lambda = diag\{\lambda_1, \lambda_2, \ldots, \lambda_m\}$. Then, by introducing $w(x,t) \equiv R^{-1}q(x,t)$ and assuming that A is constant, we can rewrite (4.2) as

$$w_t + \Lambda w_x = 0, \tag{4.5}$$

which consists of m independent advection equations $(w_p)_t + \lambda_p(w_p)_x = 0$ with $p = 1 \ldots m$. By combining their individual solutions, we obtain the solution to the system expressed as

$$q(x,t) = \sum_{p=1}^{m} r_p w_p(x,t). \tag{4.6}$$

This solution corresponds to a set of m "waves" propagating at constant speeds along so-called *characteristic curves*, which is a common property of solutions to hyperbolic systems.

As noted above, the method described assumes that the Jacobian matrix A is constant, i.e. it does not depend on x or t. If that is not the case, the transformations used are not valid and obtaining a solution usually requires more complex and usually approximate approaches such as the numerical methods we apply in this work.

4.3 Finite Volume Methods

Finite volume methods are a class of numerical methods that model the domain as a discrete grid, where each cell stores an approximation to the average value of the solution q within it. In a one-dimensional problem, the numerical solution in the ith grid cell (defined as $\mathcal{C}_i = [x_{i-1/2}, x_{i+1/2}]$) at a given time t_n is approximated as:

$$Q_i^{(n)} \approx \frac{1}{V_i} \int_{\mathcal{C}_i} q(x, t_n)\,\mathrm{d}x, \tag{4.7}$$

where $V_i = \Delta x_i = x_{i+1/2} - x_{i-1/2}$ is the "volume" of the grid cell.

In this work we concentrate on so-called *Godunov-type methods*, which update the cell-averaged solution Q based on the *fluxes* crossing the cell boundaries (edges). For that, we compute so-called *numerical fluxes*, which are approximations of the time-averaged fluxes transferred between adjacent cells during a time step. The numerical flux across the left edge of cell $\mathcal{C}_i$ is defined as

$$F_{i-1/2}^{(n)} \approx \frac{1}{\Delta t} \int_{t_n}^{t_{n+1}} f(q(x_{i-1/2}, t)) \, \mathrm{d}t, \tag{4.8}$$

where Δt is the length of the time step.

The numerical fluxes can be used to update the solution from $Q_i^{(n)}$ to $Q_i^{(n+1)}$ following

$$Q_i^{(n+1)} = Q_i^{(n)} - \frac{\Delta t}{\Delta x_i}(F_{i+1/2}^{(n)} - F_{i-1/2}^{(n)}). \tag{4.9}$$

This is often written in the *fluctuation form*

$$Q_i^{(n+1)} = Q_i^{(n)} - \frac{\Delta t}{\Delta x_i}(\mathcal{A}^+\Delta Q_{i-1/2}^{(n)} + \mathcal{A}^-\Delta Q_{i+1/2}^{(n)}). \tag{4.10}$$

Here, the *fluctuation* $\mathcal{A}^+\Delta Q_{i-1/2}^{(n)}$ describes the total effect of all waves entering or leaving the cell through its left edge, while $\mathcal{A}^-\Delta Q_{i+1/2}^{(n)}$ corresponds to the waves crossing the right edge. This is a first-order accurate method that can be extended with second-order correction terms along with limiters to create a high-resolution method. This is covered in detail in Section 4.1 of [39] and Chapter 6 of [38].

The explicit update scheme reduces the problem to computing the fluctuations $\mathcal{A}^\pm\Delta Q$, which is usually accomplished by solving the so-called *Riemann problems* that happen at the edges. These consist of the equations being solved subject to initial piecewise data at a time $t = t_n$ containing a discontinuity at one specific point $x = \bar{x}$:

$$q(x, t_n) = \begin{cases} q_\ell & \text{if } x < \bar{x} \\ q_r & \text{if } x > \bar{x}, \end{cases} \tag{4.11}$$

where q_ℓ and q_r are constant vectors that effectively define the Riemann problem. In the context of a finite volume scheme, we generally consider the Riemann problems happening at the edges. For example, for an edge located at $\bar{x} = x_{i-1/2}$, we have $q_\ell = Q_{i-1}^{(n)}$ and $q_r = Q_i^{(n)}$.

Like the solution given by (4.6), the solution to a Riemann problem describes m waves propagating at constant speeds along the characteristic curves. The numerical algorithms used to obtain solutions to Riemann problems are known as *Riemann solvers* and are generally the most complex piece of Godunov-type methods. In many cases, computing the exact solution is too computationally expensive and not worth the effort, such that approximated solvers are used instead. Even when considering approximate solvers, that is often the most time-consuming step and for that reason it is the focus of most of the discussions in this thesis involving computational performance of finite volume applications.

4.4 (Single-Layer) Shallow Water Equations

The shallow water equations are depth-averaged equations that are suitable for modeling incompressible fluids in problems where the horizontal scales (x and y dimensions) are much larger than the vertical scale (z dimension) and the vertical acceleration is negligible. This includes

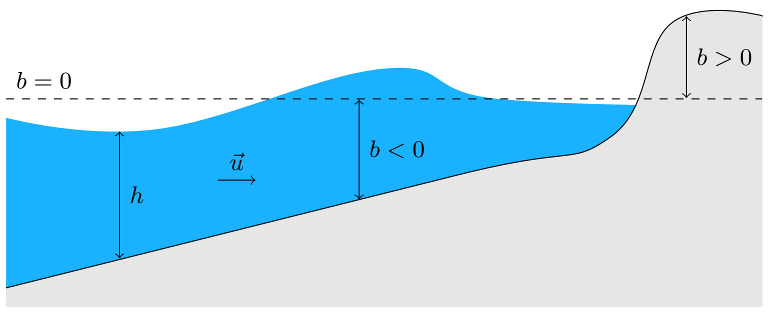


Figure 4.1: Components of the single-layer shallow water equations on a one-dimensional model of an ocean. The bathymetry data $b(x)$ is measured relatively to a reference level $b = 0$ (usually mean sea level). The water depth $h(x)$ may be equal to zero, indicating a dry state.

various problems involving wave propagation of oceanic flows such as tsunamis, because the ocean wave lengths are generally very long compared to the ocean depth [39].

This work deals with various finite volume implementations that solve two different variations of the shallow water equations. In this section we discuss the simplest variation of those, which is predominantly known simply as *shallow water equations* in literature. However, throughout this thesis we refer to it as *single-layer shallow water equations*, in order to make a clear distinction from the two-layer variation that is also studied here.

In one dimension, the single-layer shallow water equations can be written as

$$\begin{bmatrix} h \\ hu \end{bmatrix}_t + \begin{bmatrix} hu \\ hu^2 + \frac{1}{2}gh^2 \end{bmatrix}_x = \begin{bmatrix} 0 \\ -ghb_x \end{bmatrix}, \tag{4.12}$$

where $h(x,t)$ is the fluid depth; $u(x,t)$ is the vertically averaged fluid velocity in the x direction; g is the gravitational constant; and $b(x)$ is the bottom surface elevation relative to an arbitrary reference. For ocean modeling, $b(x)$ is usually relative to mean sea level such that $b < 0$ corresponds to submarine bathymetry and $b > 0$ to terrain topography. Here, the source term $S(x,t) = [0, -ghb_x]^T$ models the effect of the varying bathymetry. The source term can be extended to model other effects, such as drag and Coriolis forces, as well [39]. Fig. 4.1 illustrates all the components of (4.12) in a simplified model of an ocean.

4.4.1 The F-Wave Solver

Note that (4.12) is a balance law that can be expressed as (4.1), with

$$q(x,t) = \begin{bmatrix} q_1 \\ q_2 \end{bmatrix} = \begin{bmatrix} h \\ hu \end{bmatrix}, \quad f(q) = \begin{bmatrix} q_2 \\ q_2^2/q_1 + \frac{1}{2}gq_1^2 \end{bmatrix}, \quad \psi(q,x,t) = \begin{bmatrix} 0 \\ -gq_1b_x \end{bmatrix}. \tag{4.13}$$

The Jacobian matrix of $f(q)$ is

$$A = f'(q) = \begin{bmatrix} 0 & 1 \\ gh - u^2 & 2u \end{bmatrix}, \tag{4.14}$$

which is not constant and can therefore not be used to obtain exact solutions using the analytical approach described in Section 4.2. Nevertheless, approximate Riemann solvers often use very similar approaches, where linearization of the original problem is achieved by using constant approximations to A.

In particular, we describe the *f-wave solver* [4], which splits the *flux difference* $f(q_r) - f(q_\ell)$ as a linear combination of so-called *f-waves* $\mathcal{Z}_p$ that propagate along the characteristic curves:

$$f(q_r) - f(q_\ell) - \Delta x \psi = \sum_{p=1}^{m} \beta_p r_p \equiv \sum_{p=1}^{m} \mathcal{Z}_p. \tag{4.15}$$

The f-wave solver for the single-layer shallow water equations considered in this work uses constant approximations for the eigenvectors $r_1 = [1, s_1]^T$ and $r_2 = [1, s_2]^T$ containing the so-called *Einfeldt speeds* [21], which are defined as

$$s_1 = min(\hat{s}_1, u_\ell - \sqrt{gh_\ell}), \quad s_2 = max(\hat{s}_2, u_r + \sqrt{gh_r}), \tag{4.16}$$

where $\hat{s}_1$ and $\hat{s}_2$ are the *Roe speeds* [61]

$$\hat{s}_1 = \hat{u} - \sqrt{g\hat{h}}, \quad \hat{s}_2 = \hat{u} + \sqrt{g\hat{h}} \tag{4.17}$$

computed using the *Roe averages* of the left and right states

$$\hat{h} = \frac{h_\ell + h_r}{2}, \quad \hat{u} = \frac{u_\ell \sqrt{h_\ell} + u_r \sqrt{h_r}}{\sqrt{h_\ell} + \sqrt{h_r}}. \tag{4.18}$$

By solving (4.15) for β_1 and β_2, we can compute the f-waves $\mathcal{Z}_1$ and $\mathcal{Z}_2$, which are finally used to compute the fluctuations

$$\mathcal{A}^- \Delta Q = \sum_{p:\, s_p < 0} \mathcal{Z}_p, \quad \mathcal{A}^+ \Delta Q = \sum_{p:\, s_p > 0} \mathcal{Z}_p \tag{4.19}$$

that are used in the finite volume update scheme (4.10). In Alg. 4.1, we show the general structure of the f-wave solver implementation considered in this work.

One of the advantages of the f-wave solver is that the source terms can be easily included in its formulation creating a "well-balanced scheme" that is able to maintain steady states [4]. However, due to its simplicity the f-wave solver has a few shortcomings when compared to more advanced solvers. Notably, it does not prevent non-negativity of the fluid depth h, such that it is not able to properly handle wetting and drying of cells. To avoid such situations, solver implementations often impose "wall boundary" conditions on Riemann problems where one of the cells is dry and the other is wet – see e.g. lines 5–13 in Alg. 4.1.

4.4.2 The Augmented Riemann Solver

A more sophisticated and accurate Riemann solver that is able to properly handle wetting and drying of cells is known as *augmented Riemann solver.* Its derivation is rather complicated and is beyond the scope of this thesis, so here we will only provide a brief description of its main ideas along with an illustration of its general code structure (see Alg. 4.2). The interested reader is referred to [29] for further details.

The augmented Riemann solver decomposes the jumps at the Riemann interface into four waves:

$$\begin{bmatrix} h_r - h_\ell \\ h_r u_r - h_\ell u_\ell \\ \phi(q_r) - \phi(q_\ell) \\ b_r - b_\ell \end{bmatrix} = \sum_{p=1}^{4} \alpha_p w_p, \tag{4.20}$$

where $\phi(q) = hu^2 + \frac{1}{2}gh^2$. This decomposition gives the eigenvalues

$$\lambda_1 = u - \sqrt{gh}, \quad \lambda_2 = u + \sqrt{gh}, \quad \lambda_3 = 2u, \quad \lambda_4 = 0 \tag{4.21}$$

and the respective eigenvectors

$$r_1 = \begin{bmatrix} 1 \\ \lambda_1 \\ \lambda_1^2 \\ 0 \end{bmatrix}, \quad r_2 = \begin{bmatrix} 1 \\ \lambda_2 \\ \lambda_2^2 \\ 0 \end{bmatrix}, \quad r_3 = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}, \quad r_4 = \begin{bmatrix} gh/(\lambda_1\lambda_2) \\ 0 \\ -gh \\ 1 \end{bmatrix}. \tag{4.22}$$

Note that λ_4 and r_4 describe a stationary wave that was created by adding the jump in the bathymetry to the decomposition. By omitting the null values and using the Roe averages and Einfeldt speeds similarly as done for the f-wave solver, it is possible to convert (4.20) into

Algorithm 4.1: F-wave solver for the single-layer shallow water equations with preprocessing of dry states. To avoid inundation scenarios (which are not properly handled by the f-wave solver), we replace dry cells with wall barrier conditions. We also skip problems with dry cells on both sides, because their solution are always zero (no fluctuations).

```
Input: $h_\ell$, $(hu)_\ell$, $b_\ell$, $h_r$, $(hu)_r$ $b_r$                  // Data in left and right cells
Output: $\mathcal{A}^-\Delta Q$, $\mathcal{A}^+\Delta Q$                      // Left- and right-going fluctuations
if both left/right cells are dry then                                         // Skip completely dry problems
    $\mathcal{A}^-\Delta Q \leftarrow \vec{0}$
    $\mathcal{A}^+\Delta Q \leftarrow \vec{0}$
    Return $\mathcal{A}^-\Delta Q$, $\mathcal{A}^+\Delta Q$                     // Finish execution here
else if left cell is dry then                                                 // Simulate wall boundary on dry left cells
    $h_\ell \leftarrow h_r$
    $(hu)_\ell \leftarrow -(hu)_r$
    $b_\ell \leftarrow b_r$
else if right cell is dry then                                                // Simulate wall boundary on dry right cells
    $h_r \leftarrow h_\ell$
    $(hu)_r \leftarrow -(hu)_\ell$
    $b_r \leftarrow b_\ell$
end
$\hat{h} \leftarrow (h_\ell + h_r)/2$                                           // Compute Roe averages (4.18)
$\hat{u} \leftarrow (u_\ell\sqrt{h_\ell} + u_r\sqrt{h_r})/(\sqrt{h_\ell} + \sqrt{h_r})$
$\hat{s}_1 \leftarrow \hat{u} - \sqrt{g\hat{h}}$                                 // Compute Roe speeds (4.17)
$\hat{s}_2 \leftarrow \hat{u} + \sqrt{g\hat{h}}$
$s_1 \leftarrow min(\hat{s}_1, u_\ell - \sqrt{gh_\ell})$                          // Compute Einfeldt speeds (4.16)
$s_2 \leftarrow max(\hat{s}_2, u_r + \sqrt{gh_r})$
Solve $\begin{bmatrix} 1 & 1 \\ s_1 & s_2 \end{bmatrix} \begin{bmatrix} \beta_1 \\ \beta_2 \end{bmatrix} = f(h_r, (hu)_r) - f(h_\ell, (hu)_\ell) - \Delta x \psi$   // Compute $\beta_1$ and $\beta_2$ (4.15)
for $p \leftarrow 1$ to $2$ do                                                   // Compute fluctuations (4.19)
    if $s1 < 0$ then
        $\mathcal{A}^-\Delta Q \leftarrow \mathcal{A}^-\Delta Q + \beta_p[1, s_p]^T$
    else
        $\mathcal{A}^+\Delta Q \leftarrow \mathcal{A}^+\Delta Q + \beta_p[1, s_p]^T$
    end
end
Return $\mathcal{A}^-\Delta Q$, $\mathcal{A}^+\Delta Q$
```

$$\begin{bmatrix} h_r - h_\ell \\ h_r u_r - h_\ell u_\ell \\ \phi(q_r) - \phi(q_\ell) \end{bmatrix} = \beta_1 \begin{bmatrix} 1 \\ s_1 \\ s_1^2 \end{bmatrix} + \beta_2 \begin{bmatrix} 1 \\ s_2 \\ s_2^2 \end{bmatrix} + \beta_3 \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}, \tag{4.23}$$

which must be solved for the β_p unknowns in order to compute the fluctuations $\mathcal{A}^-\Delta Q$ and $\mathcal{A}^+\Delta Q$ that will be used in the update scheme.

In addition to the basic numerical algorithm described above, the augmented Riemann solver implementation used in this work also performs additional analyses of the wave structures, including checks for special cases such as dry states and supercritical problems. For some cases, the solver even requires one or very few iterations of Newton's method to find approximate solutions to a nonlinear system of equations. These features make the solver implementation considerably complex, with multiple if-then-else branches that can seriously influence vectorization performance. This will be further discussed in Section 6.2.

Algorithm 4.2: Augmented Riemann solver for the single-layer shallow water equations. Like the f-wave solver (Alg. 4.1), this solver also checks for dry states on both sides and skips completely dry problems. However, additional checks are performed to estimate whether a dry cell may be inundated and this is handled accordingly. Computation of the middle state is also required for computing the eigenspace decomposition. These steps (lines 4, 10 and 17) are usually performed with the same algorithm, which includes several additional if-then-else branches to check for dry states and wave structures and may require applying Newton's method. We note that this only shows the general structure of the algorithm and it is not intended to be a complete description of the full algorithm. Some steps and details have been omitted for clarity.

```
    Input: $h_\ell$, $(hu)_\ell$, $b_\ell$, $h_r$, $(hu)_r$ $b_r$          // Data in left and right cells
    Output: $\mathcal{A}^-\Delta Q$, $\mathcal{A}^+\Delta Q$                  // Left- and right-going fluctuations
1   if both left/right cells are dry then                  // Skip completely dry problems
2       Return $\mathcal{A}^\pm\Delta Q \leftarrow \vec{0}$                                 // Finish execution here
3   else if left cell is dry then
4       if water from right cell can inundate left cell then   // This may require Newton's method
5           Prepare solver for handling inundation
6       else
7           Simulate wall boundary condition on left cell     // Like done in Alg. 4.1 (lines 6–8)
8       end
9   else if right cell is dry then
10      if water from left cell can inundate right cell then   // This may require Newton's method
11          Prepare solver for handling inundation
12      else
13          Simulate wall boundary condition on right cell    // Like done in Alg. 4.1 (lines 10–12)
14      end
15  end
16  Compute Roe Averages, Roe speeds and Einfeldt speeds      // Like done in Alg. 4.1 (lines 14–19)
17  Compute middle state for Riemann interface                // This may require Newton's method
18  Compute eigenspace decomposition for waves                                // Based on (4.20)
19  Solve $\begin{bmatrix} h_r - h_\ell \\ h_r u_r - h_\ell u_\ell \\ \phi(q_r) - \phi(q_\ell) \end{bmatrix} = \beta_1 \begin{bmatrix} 1 \\ s_1 \\ s_1^2 \end{bmatrix} + \beta_2 \begin{bmatrix} 1 \\ s_2 \\ s_2^2 \end{bmatrix} + \beta_3 \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$   // Compute $\beta_1$, $\beta_2$ and $\beta_3$ (4.23)
20  Compute fluctuations $\mathcal{A}^-\Delta Q$ and $\mathcal{A}^+\Delta Q$                        // Similar to (4.19)
21  Return $\mathcal{A}^-\Delta Q$, $\mathcal{A}^+\Delta Q$
```

4.5 Two-Layer Shallow Water Equations

Although the single-layer equations are appropriate for modeling various wave propagation phenomena, such as tsunamis and dam breaks, they lack accuracy for problems where significant vertical variations in the water column can be observed. For instance, in storm surge simulations wind stress plays a crucial role and affects more the top of the water column than the bottom [48]. The single-layer equations are not able to properly model this effect, because the water momentum gets averaged vertically.

Vertical profiles can be more accurately modeled using *multi-layer shallow water equations*, which model the fluid with multiple vertical layers, possibly with different fluid densities. They can provide more accurate representations than the single-layer equations, while keeping the computational costs substantially lower than three-dimensional equations.

The simplest variation of the multi-layer equations consists of the *two-layer shallow water equations*. With them it is possible, for example, to model the ocean as composed of a shallow top layer and a deeper bottom layer, which allows more accurate representations of the wind effects. Multi-layer shallow water equations may also be useful for problems where it is possible to identify multiple layers with different characteristics. An interesting example is the Alboran Sea in the Mediterranean, where two layers of water with different densities can be distinguished: a top layer coming from the Atlantic Ocean through the Strait of Gilbratar, and a bottom layer with denser water coming from the Mediterranean into the Atlantic [43].

The system of two-layer shallow water equations in one dimension reads

$$\begin{bmatrix} h_1 \\ h_1u_1 \\ h_2 \\ h_2u_2 \end{bmatrix}_t + \begin{bmatrix} h_1u_1 \\ h_1u_1^2 + \frac{1}{2}gh_1^2 \\ h_2u_2 \\ h_2u_2^2 + \frac{1}{2}gh_2^2 \end{bmatrix}_x = \begin{bmatrix} 0 \\ -gh_1(h_2+b)_x \\ 0 \\ -rgh_2(h_1)_x - gh_2b_x \end{bmatrix}, \tag{4.24}$$

where h_1 and h_1u_1 are the conserved quantities in the top layer; and h_2 and h_2u_2 the ones in the bottom layer; also, $r \equiv \rho_1/\rho_2$ is the ratio of the densities of the fluid contained in each layer. These components are illustrated in Fig. 4.2. Note that these equations are actually very similar to two sets of single-layer shallow water equations, except for a few additional terms that model the hydrostatic pressure between the layers.

These equations can be extended to an arbitrary number of vertical layers [11], where each layer $i = 1 \dots n$ can be modeled through the equations

$$\begin{bmatrix} h_i \\ h_iu_i \end{bmatrix}_t + \begin{bmatrix} h_iu_i \\ h_iu_i^2 + \frac{1}{2}gh_i^2 \end{bmatrix}_x = \begin{bmatrix} 0 \\ -gh_i \left(\sum_{j=i+1}^{n} h_j + \sum_{j=1}^{i-1} \frac{\rho_j}{\rho_i} h_j + b \right)_x \end{bmatrix}. \tag{4.25}$$

However, in this work we deal only with the single-layer and two-layer forms described above.

4.5.1 The Two-Layer Solver

In this work we consider the Riemann solver for the two-layer shallow water equations that was proposed in [45, 46], to which we often refer simply as *two-layer solver* in this thesis. It is based on the f-wave formulation described in Section 4.4.1, but it was extended to handle dry states in each layer. The mathematical theory behind this solver is also rather complicated and will not be covered in detail here. Instead, we will only describe the general structure of its implementation, illustrated in Alg. 4.3.

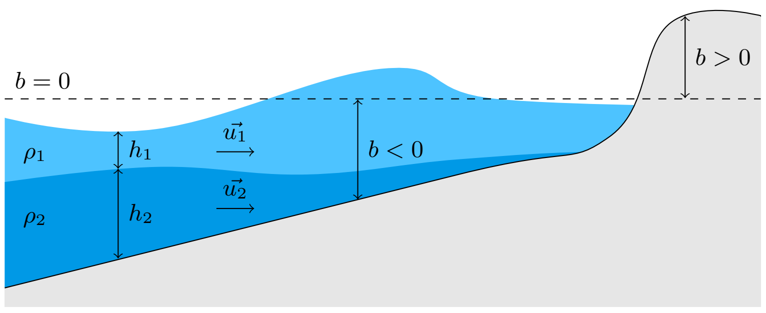


Figure 4.2: Components of the two-layer shallow water equations on a one-dimensional model of an ocean. The fluid in each layer may have a different density ρ. One or both water depths $h_1(x)$ and $h_2(x)$ may be equal to zero, meaning that the respective layer is dry. However, we usually assume that the bottom layer becomes dry before the top layer, as illustrated in the region near the shore.

As the augmented Riemann solver described in 4.4.2, this solver contains several if-then-else branches, which is usually detrimental to vectorization performance. The branches are used mainly to handle dry states that may appear on both layers and on both sides of the Riemann interface. For instance, if the two layers are dry on both sides, this is a trivial dry problem that does not require a solver (there are no fluxes crossing the interface). Also, if one of the layers is dry on both sides, this can be treated as a single-layer problem and we directly apply the augmented Riemann solver for the single-layer equations, which is by itself also heavily branched.

On the other hand, if there is water on at least one side of each layer, the full two-layer equations need to be solved. This includes additional checks for dry states and inundation of cells/layers, adding several more branches to the algorithm. After all the special cases are handled, the fluctuations $\mathcal{A}^{\pm}\Delta Q$ are computed using the f-wave formulation. More specifically, we first compute approximations to the system's eigenvalues and eigenvectors, and then solve a 6×6 system of linear equations based on (4.15).

4.6 Extension to Two-Dimensional Domains

So far we have only discussed one-dimensional hyperbolic PDEs and their respective solutions. However, we are generally interested in applications that handle problems with at least two spatial dimensions. The general form for two-dimensional hyperbolic PDEs is given by

$$q_t + f(q)_x + g(q)_y = \psi(q, x, y, t), \tag{4.26}$$

where $g(q)$ is a flux function in the y direction, analogous to $f(q)$.

The two-dimensional single-layer shallow water equations read

$$\begin{bmatrix} h \\ hu \\ hv \end{bmatrix}_t + \begin{bmatrix} hu \\ hu^2 + \frac{1}{2}gh^2 \\ huv \end{bmatrix}_x + \begin{bmatrix} hv \\ huv \\ hv^2 + \frac{1}{2}gh^2 \end{bmatrix}_y = \begin{bmatrix} 0 \\ -ghb_x \\ -ghb_y \end{bmatrix}, \tag{4.27}$$

where $u(x, y, t)$ and $v(x, y, t)$ are the vertically averaged fluid velocities in the x and y directions, respectively. Similarly, the two-layer shallow water equations in two dimensions have the form

Algorithm 4.3: Riemann solver for the two-layer shallow water equations. The solver uses several if-then-else branches to handle dry states on layers/cells. In some cases, the problem is equivalent to the single-layer shallow water equations and the augmented Riemann solver (Alg. 4.2) is used instead. When at least one cell is wet for each layer, the full two-layer solver is applied, which is able to properly handle inundation of the bottom layer. Again, here we only show the general structure of the algorithm and omit some steps and details.

```
Input: $Q_\ell$, $b_\ell$, $Q_r$, $b_r$                               // Data in left and right cells
Output: $\mathcal{A}^-\Delta Q$, $\mathcal{A}^+\Delta Q$              // Left- and right-going fluctuations
if both layers are dry in both cells then                              // Skip completely dry problems
    $\mathcal{A}^\pm\Delta Q \leftarrow \vec{0}$                       // There are no fluctuations
else if top layer is dry in both cells then
    $\mathcal{A}^\pm\Delta Q \leftarrow$ solution of single-layer equations on bottom layer   // Use augmented Riemann solver
else if bottom layer is dry in both cells then
    $\mathcal{A}^\pm\Delta Q \leftarrow$ solution of single-layer equations on top layer      // Use augmented Riemann solver
else
    // At least one cell is wet for each layer $\Rightarrow$ solve the full two-layer problem
    if both layers are dry in left cell then
        Simulate wall boundary on left cell                            // Similar to lines 6–8 in Alg. 4.1
    else if both layers are dry in right cell then
        Simulate wall boundary on right cell                           // Similar to lines 6–8 in Alg. 4.1
    else if bottom layer is dry in left cell then
        if water from right cell can inundate bottom layer in left cell then
            Prepare solver for handling inundation
        else
            Simulate wall boundary on left cell                        // Similar to lines 6–8 in Alg. 4.1
        end
    else if bottom layer is dry in right cell then
        if water from left cell can inundate bottom layer in right cell then
            Prepare solver for handling inundation
        else
            Simulate wall boundary on right cell                       // Similar to lines 6–8 in Alg. 4.1
        end
    end
    Compute eigenspace decomposition                                   // Considering inundation if necessary
    Solve $6 \times 6$ system of linear equations                      // To compute $\beta_{1\ldots6}$, similarly to (4.23)
    Compute fluctuations $\mathcal{A}^-\Delta Q$ and $\mathcal{A}^+\Delta Q$   // Similar to (4.19)
end
Return $\mathcal{A}^-\Delta Q$, $\mathcal{A}^+\Delta Q$
```

$$
\begin{bmatrix} h_1 \\ h_1u_1 \\ h_1v_1 \\ h_2 \\ h_2u_2 \\ h_2v_2 \end{bmatrix}_t +
\begin{bmatrix} h_1u_1 \\ h_1u_1^2 + \frac{1}{2}gh_1^2 \\ h_1u_1v_1 \\ h_2u_2 \\ h_2u_2 + \frac{1}{2}gh_2^2 + rgh_2h_1 \\ h_2u_2v_2 \end{bmatrix}_x +
\begin{bmatrix} h_1v_1 \\ h_1u_1v_1 \\ h_1v_1^2 + \frac{1}{2}gh_1^2 \\ h_2v_2 \\ h_2u_2v_2 \\ h_2v_2^2 + \frac{1}{2}gh_2^2 + rgh_2h_1 \end{bmatrix}_y =
\begin{bmatrix} 0 \\ -gh_1(h_2)_x - gh_1b_x \\ -gh_1(h_2)_y - gh_1b_y \\ 0 \\ rgh_1(h_2)_x - gh_2b_x \\ rgh_1(h_2)_y - gh_2b_y \end{bmatrix}. \tag{4.28}
$$

Both variations of the shallow water equations in two dimensions are rotationally invariant, which means that the solutions to the Riemann problems have the same structure regardless of the direction in which they are defined. This allows two-dimensional methods to use Riemann solvers designed for the one-dimensional equations with only a few modifications, which include applying a change of basis to the momentum vector $[hu, hv]^T$ according to the direction of the edge being considered.

A common approach used in two-dimensional Godunov-type methods on Cartesian meshes is sometimes called *donor-cell upwind method* [38]. The basic idea is to solve the Riemann problems at all four edges of a cell $\mathcal{C}_{i,j}$ and then to update its solution following

$$\begin{aligned} Q_{i,j}^{(n+1)} = Q_{i,j}^{(n)} &- \frac{\Delta t}{\Delta x}(\mathcal{A}^{+}\Delta Q_{i-1/2,j} + \mathcal{A}^{-}\Delta Q_{i+1/2,j}) \\ &- \frac{\Delta t}{\Delta y}(\mathcal{B}^{+}\Delta Q_{i,j-1/2} + \mathcal{B}^{-}\Delta Q_{i,j+1/2}), \end{aligned} \tag{4.29}$$

where $\mathcal{A}^{\pm}\Delta Q$ are the fluctuations in the x direction (left and right edges) and $\mathcal{B}^{\pm}\Delta Q$ are the fluctuations in the y direction (bottom and top edges).

The accuracy of the method described above can be improved by including *flux corrections* terms to become what is known as the *corner-transport upwind method* [38]:

$$\begin{aligned} Q_{i,j}^{(n+1)} = Q_{i,j}^{(n)} &- \frac{\Delta t}{\Delta x}(\mathcal{A}^{+}\Delta Q_{i-1/2,j} + \mathcal{A}^{-}\Delta Q_{i+1/2,j}) \\ &- \frac{\Delta t}{\Delta y}(\mathcal{B}^{+}\Delta Q_{i,j-1/2} + \mathcal{B}^{-}\Delta Q_{i,j+1/2}) \\ &- \frac{\Delta t}{\Delta x}(\widetilde{F}_{i+1/2,j} - \widetilde{F}_{i-1/2,j}) \\ &- \frac{\Delta t}{\Delta y}(\widetilde{G}_{i,j+1/2} - \widetilde{G}_{i,j-1/2}). \end{aligned} \tag{4.30}$$

The flux corrections $\widetilde{F}_{i\pm1/2,j}$ and $\widetilde{G}_{i,j\pm1/2}$ are computed by solving so-called *transverse Riemann problems* over the edges (in contrast to the *normal* Riemann problems discussed previously). They concern waves with a “transverse” component (i.e., waves that do not propagate perpendicularly to the edge), and their solutions can be obtained from the waves computed by the normal Riemann solver. By including these flux corrections, the finite volume scheme is able to properly model the effect that such waves can have on cells other than the two cells adjacent to the edge (e.g., $\mathcal{C}_{i+1,j+1}$ may be directly affected by waves coming from $\mathcal{C}_{i,j}$, even though they are not edge-adjacent).

Like in the one-dimensional case, this method is first-order accurate and can be improved into a high-resolution method by incorporating second-order correction terms and limiters. For more details on that and more complete discussions of the concepts introduced in this section, see Chapters 20–21 of [38].

5 Experimental Platforms

All performance experiments described in this work were conducted on the CoolMUC-2 [36,75] and CoolMUC-3 [37] cluster systems maintained by the Leibniz Supercomputing Center (LRZ). The former cluster is composed of 384 nodes with dual-socket Intel Xeon "Haswell" systems, while the latter consists of 148 nodes with Intel Xeon Phi "Knights Landing" (KNL) processors. An overview of the configuration of the two cluster systems is presented in Table 5.1. As we focus on vectorization performance, we point out the difference in the SIMD width of these machines: the Haswells provide AVX2 instructions (256-bit), while the KNLs provide AVX-512 instructions (512-bit). As such, the benefits from vectorization are obviously expected to be more noticeable on the KNL nodes.

In Table 5.1 we list not only the theoretical single-node peak performance of those systems, but also an estimate for the maximum performance that can be achieved in practice on them, measured in (double-precision) floating-point operations per second (Flops/s). While the theoretical peak was calculated based on the specifications of each system, the estimated peak was obtained using a benchmark proposed in [34], which we describe in Section 5.1. In our performance experiments, we often use the values obtained with this benchmark to assess how well our implementations are able to exploit the computational resources provided by each system.

In all reported experiments we used the Intel Fortran compiler 17.0.6 and double precision arithmetic. On both systems we use only one node at a time (i.e., no distributed parallelism),

Table 5.1: Specifications of the nodes in the experimental platforms.

System overview	**Dual-socket Haswells**	**KNL**
Architecture	Intel® Xeon®	Intel® Xeon Phi™
Model	E5-2697v3	7210F
Cores	2×14	64 (max. 256 threads)
Clock rate	2.60 GHz	1.30 GHz
SIMD vector width	256-bit	512-bit
L1 Instruction Cache	32 KB/core	32 KB/core
L1 Data Cache	32 KB/core	32 KB/core
L2 Cache	256 KB/core	1 MB/core
L3 Cache	35 MB	–
Memory	64 GB	96 GB + 16 GB MCDRAM
Peak bandwidth	136 GB/s	102 GB/s
Peak throughput (double)	582 GFlop/s	2 662 GFlops/s
Measured throughput (double)	156 GFlop/s	720 GFlops/s

Code 5.1: Main loops in the "helloflops3" benchmark. Adapted from [34].

```fortran
! Parameters used for arrays' dimensions and loop count
integer, parameter :: FLOPS_ARRAY_SIZE = 1024*512
integer, parameter :: MAXFLOPS_ITERS = 100000000
integer, parameter :: LOOP_COUNT = 128
! Define arrays that are 64-byte-aligned for best cache access
real :: a(FLOPS_ARRAY_SIZE)
real :: b(FLOPS_ARRAY_SIZE)
! Other variables
integer   :: i, j, k, numthreads, offset

![...] (Initialize arrays, timers, counters and variable 'numthreads')

!$OMP PARALLEL DO PRIVATE(j,k,offset)
do i=1, numthreads
    ! Each thread will work on its own array section
    offset = i * LOOP_COUNT

    ! Loop many times to get lots of calculations
    do j=1, MAXFLOPS_ITERS

        !DIR$ VECTOR ALIGNED
        do k=1, LOOP_COUNT
            a(k+offset) = 1.1 * a(k+offset) + b(k+offset)
        end do
    end do
end do
```

but we use shared-memory parallelism with OpenMP on all available cores, i.e., 28 on the dual-socket Haswells and 64 on KNLs. On KNLs we also experiment with different number of threads per core (from 1 to 4 with hyperthreading). However, in the presented results we only list the experiments with 2 threads per core (i.e., 128 in total), because this configuration achieved the best performance in most experiments (in general, slightly faster than with 3 and 4 threads per core, and considerably faster than with only 1 thread per core). Also, we always use the KNLs in cache mode, i.e., the MCDRAM memory is used as an L3 cache.

5.1 Peak Throughput Benchmark

The estimates for maximum attainable performance on each system listed in Table 5.1 have been obtained with the Fortran implementation of the "`helloflops3`" benchmark proposed in Chapter 2 of [34]. This benchmark measures the Flops/s rate in a nested loop that repeatedly computes `a(i)` $\leftarrow$ `1.1 * a(i) + b(i)` on all elements of small (cache-fitting) arrays `a` and `b` using OpenMP threading and vectorization. For illustration, we show the main loops in the benchmark in Code 5.1.

This benchmark was designed with the goal of obtaining the highest attainable performance in architectures exactly like our experimental platforms, which support multi-threading, vectorization, fused multiply-add operations, etc. In addition, it works repeatedly on small arrays that fit easily in the processor caches, so that performance is always compute-bound. Therefore, its results serve as a good estimate for the maximum performance that can be achieved in practice, which is why we use them as baseline performance for our implementations when we present our performance results in this thesis.

6 GeoClaw: Multi-Level Hierarchical Adaptive Meshes

We used the GeoClaw package [9, 28] to implement and evaluate vectorized versions of finite volume methods for the single-layer and two-layer shallow water equations, with special focus on their Riemann solvers. The finite volume implementation in GeoClaw is based on decomposing two-dimensional rectangular grids into one-dimensional slices that are processed separately. This translates naturally into lists of Riemann problems to be solved, providing an advantageous scenario for experimenting with vectorization, as we will show in this chapter.

This chapter is an extended version of one of our previous publications [25]. In addition to providing more detailed discussions and examples when describing the GeoClaw framework and our work on adding vectorization to it, here we also evaluate the performance of the vectorized f-wave solver (which was not considered in the paper) and present updated experimental results.

We start by giving an overview on the algorithms used in GeoClaw for its finite volume scheme with parallel adaptive mesh refinement in Section 6.1. We describe in detail our work and the changes in the code structure that were necessary to achieve efficient vectorization of the numerical algorithms in Section 6.2. Then, we present an experimental evaluation of the vectorized implementations in Section 6.3 and summarize our main findings in Section 6.4

6.1 GeoClaw and Clawpack

GeoClaw [9,28] is a software package that supports the implementation of finite volume methods for modeling geophysical flow problems over real topography or bathymetry data, either on Cartesian grids or on longitude-latitude grids. It is an extension of the software Clawpack [15, 47], which provides a framework for solving systems of hyperbolic PDEs in one, two or three dimensions. Geoclaw was designed to tackle various difficulties that happen on simulations involving realistic data, such as handling of dry states with the shallow water equations. It also provides performance features such as adaptive mesh refinement and shared-memory parallelism. In this section we describe the algorithms used specifically in GeoClaw, but we point out that many of the discussed details are generally valid for Clawpack as well.

Application developers can create finite volume implementations with GeoClaw by providing Riemann solvers to the specific system of equations they want to solve. The Clawpack and GeoClaw open-source repositories [14] contain several examples of applications based on various hyperbolic problems. In this work we focus on the single-layer and two-layer shallow water equations described in Sections 4.4 and 4.5.

6.1.1 Adaptive Mesh Refinement

As discussed in Chapter 3, adaptive mesh refinement is essential for large-scale simulations where high resolution is required at specific regions but not over the entire domain, with oceanic applications being a typical example. GeoClaw provides adaptivity by organizing multiple patches of logically rectangular grids using the multi-level hierarchical strategy described in Section 3.1.1. The entire simulation domain is covered by a coarse level 1 grid, whose cells may be individually flagged for refinement depending on various user-defined criteria. Regions of the level 1 grid are then refined, leading to the creation of a set of level 2 grids, with finer cells covering (and overlapping) those regions. This process can be repeated recursively up to an arbitrary number of levels until all areas of interest are covered by appropriate resolutions. For each refinement from level ℓ to $\ell + 1$, *refinement ratios* $r_x^{(\ell)}$ and $r_y^{(\ell)}$ for the horizontal and vertical directions can be defined independently, meaning that one cell in level ℓ is refined into $r_x^{(\ell)} \times r_y^{(\ell)}$ cells in level $\ell + 1$. We often use $r_x^{(\ell)} = r_y^{(\ell)}$, but that is not mandatory. As an example, consider the mesh with three refinement levels shown in Fig. 6.2 in Section 6.3.

GeoClaw performs a *regridding* procedure every few time steps, with a frequency that can also be defined by the user. It first identifies clusters of spatially-close cells that have been flagged for refinement. For that, it uses a pattern recognition heuristic [7] that tries to minimize the number of cells unnecessarily refined, but also avoids having too many small patches. Then, it updates the mesh structure accordingly, applying interpolation techniques where necessary.

Depending on the application, some issues need to be addressed during the regridding procedure, especially for maintaining steady states while interpolating the cell-averaged solutions between different grid levels. For example, in oceanic simulations, care must be taken to avoid generating spurious waves in regions where the water has not yet been perturbed. This is usually done by maintaining the same water elevation $h + b$ across multiple levels, despite possible variations in the cell-averaged bathymetry b for different refinements. However, that can be particularly problematic in cases where dry cells become wet (or vice-versa) due to those variations in the bathymetry. This is discussed in more details in Section 9 of [39]. We also show an implementation example in the context of the framework sam(oa)^2 in Section 7.1.4 of this thesis. For further discussions on the topic of implementing well-balanced adaptive mesh refinement for the shallow water equations, refer e.g. to [18].

6.1.2 Shared-Memory Parallelism

The patch-based mesh structure in GeoClaw is exploited for implementing shared-memory parallelization with OpenMP – each patch can be processed almost independently, needing only to exchange boundary data with other patches via an additional ghost layer. The data used for the ghost layer is either obtained directly from neighbor patches in the same refinement level or interpolated from the coarser grid that contains the patch, on the previous level. After exchanging boundary data, patches in the same refinement level can be trivially processed in parallel.

As mentioned in Section 3.3, an important detail in GeoClaw's implementation is that a limit is imposed to the dimensions of each patch, such that multiple smaller patches may be applied where a single large patch might otherwise be enough. In the current implementation, this limit is set to 60×60. This does not only lead to more parallel work (due to the larger number of patches), but also improves spatial locality by producing smaller patches that are more likely to fit in the cache.

6.1.3 Finite Volume Implementation

As discussed previously, each patch in the mesh is processed individually, and the data from all patches is then combined to obtain the solution over the entire simulation domain. In this section we describe the actual implementation of the finite volume scheme discussed in Chapter 4, which is applied individually to each rectangular grid. In the following, we assume that the grid being processed has m_x columns and m_y rows (including ghost layers). We also use the notation mx and my when these are referenced in the context of Fortran code.

Data structures

Given the rectangular shape of the grids, it would be natural to use two-dimensional arrays to store the cell-averaged solutions $Q_{i,j}$. An obvious approach would be to create one floating-point array with dimensions $m_x \times m_y$ for each component in q. E.g., for the single-layer shallow water equations in two dimensions there are three components in $q = [h, hu, hv]^T$, so three arrays H, HU, and HV would be enough to store the cell-averaged solutions over the entire grid. On the other hand, the two-layer equations would require six arrays for storing the quantities in $q = [h_1,\, h_1u_1,\, h_1v_1,\, h_2,\, h_2u_2,\, h_2v_2]^T$.

However, to allow an easier generalization of the numerical scheme implementation to multiple applications based on different systems of hyperbolic PDEs, GeoClaw actually stores the solution data using a single three-dimensional array Q(meqn,mx,my), where meqn is a constant that stores the number of components[1] in q. For example, for the single-layer shallow water equations, meqn$= 3$, and the user might define that the position Q(1,i,j) stores the averaged h in cell $C_{i,j}$, while Q(2,i,j) refers to hu and Q(3,i,j) to hv. We note that the core routines in GeoClaw are oblivious to the meaning of each component stored in Q, and that it is left to the application developer to decide their order and to maintain consistency throughout the application-specific code.

In addition to Q, a similar array Aux(maux,mx,my) can be defined to store a given number (maux) of other cell-specific "auxiliary" variables that are necessary for the simulation but should not be updated following the standard finite volume update scheme that is applied to Q. For instance, this can be used to store the bathymetry data $b(x, y)$ necessary for both variations of the shallow water equations.

Because this strategy for organizing the simulation data allows easy customization of the finite volume implementation to various problems with different storage requirements, we used a very similar approach for the generic programming interface that we developed for sam(oa)^2, as we discuss later in Chapter 8.

Update scheme for two-dimensional grids

When setting up a two-dimensional simulation in GeoClaw, users can select which method should be used to handle transverse waves (see Section 4.6). Namely, it is possible to choose between these three methods:

1. donor-cell upwind method;
2. corner-transport upwind method;
3. corner-transport upwind method with addition of second-order corrections.

In the following discussions, we assume that the third method is being used, requiring the solution of transverse Riemann problems and computation of second-order corrections. The other two methods can be trivially obtained by skipping steps in the algorithms.

To update a two-dimensional grid in time, GeoClaw extracts one-dimensional "slices" from it and solves all the Riemann problems in each of them. A slice corresponds to either one row or one column of the grid – see Fig. 6.1.

The algorithm used to compute the numerical fluxes leaving/entering each cell in the grid is shown in Alg. 6.1. Notice that each row slice $Q^{(row)}$ contains m_x contiguous cells

[1]The constant meqn actually refers to the number of equations in the system. However, for the class of equations considered in this work it is often assumed to be equal to the number of components in q.

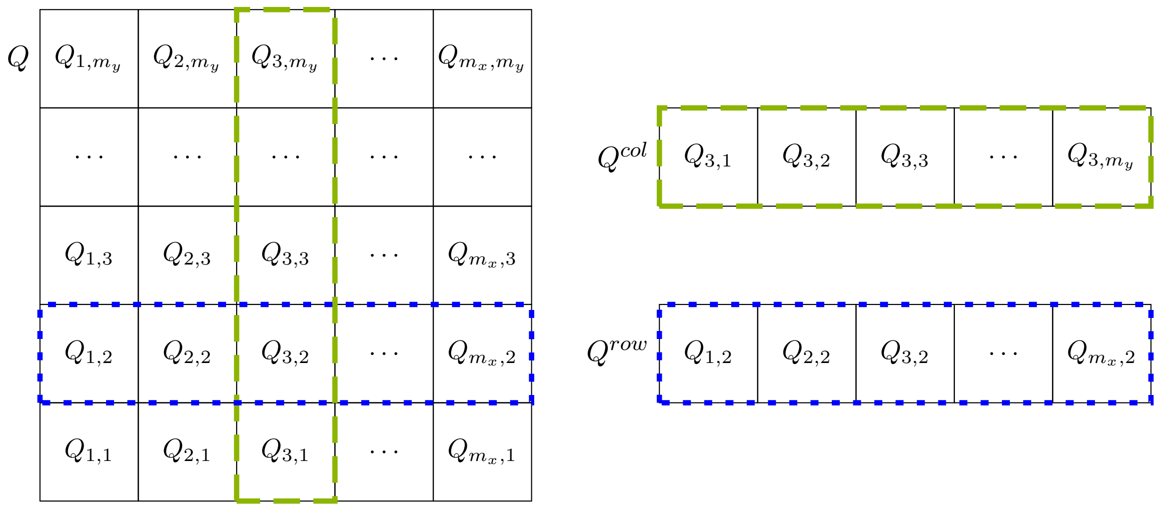

Q
Q_{1,m_y} Q_{2,m_y} Q_{3,m_y} $\cdots$ Q_{m_x,m_y}
$Q_{1,3}$ $Q_{2,3}$ $Q_{3,3}$ $\cdots$ $Q_{m_x,3}$
$Q_{1,2}$ $Q_{2,2}$ $Q_{3,2}$ $\cdots$ $Q_{m_x,2}$
$Q_{1,1}$ $Q_{2,1}$ $Q_{3,1}$ $\cdots$ $Q_{m_x,1}$
Q^{col} $Q_{3,1}$ $Q_{3,2}$ $Q_{3,3}$ $\cdots$ Q_{3,m_y}
Q^{row} $Q_{1,2}$ $Q_{2,2}$ $Q_{3,2}$ $\cdots$ $Q_{m_x,2}$

Figure 6.1: A two-dimensional grid in GeoClaw is processed column by column and then row by row. Two separate loops (one for rows and other for columns) iterate through the grid extracting one-dimensional "slices" from it. Each slice is then processed separately by a subroutine that computes the numerical fluxes crossing its edges. In this example, we show a slice composed of the third column (green/dashed contour) and another slice obtained from the second row from the bottom (blue/dotted contour).

Algorithm 6.1: Algorithm used in GeoClaw to compute the fluxes used in the finite volume update scheme (4.30). First the grid is processed column by column, with the data of each column being stored in temporary arrays $Q^{(col)}$ and $Aux^{(col)}$ that are used to compute the fluxes in the slice. For that, we use a function `computeFluxes` that returns the fluxes crossing all edges in the slice. Later, the same process is applied to all rows.

```
Input: $Q$, $Aux$                                          // 2D arrays containing cell-averaged data
Output: $Fluxes \equiv (\mathcal{A}^{\pm}, \mathcal{B}^{\pm}, \widetilde{F}, \widetilde{G})$   // 2D arrays containing numerical fluxes
$Fluxes \leftarrow 0$                                        // Initialize flux arrays
for $i \leftarrow 1$ to $m_x$ do                              // Process grid column by column
    $Q^{(col)} \leftarrow i$th column of $Q$        // Store data of column slice in temporary 1D arrays
    $Aux^{(col)} \leftarrow i$th column of $Aux$
    $Fluxes \leftarrow Fluxes +$ computeFluxes($Q^{(col)}$, $Aux^{(col)}$)   // Compute and accumulate fluxes
end
for $j \leftarrow 1$ to $m_y$ do                              // Process grid row by row
    $Q^{(row)} \leftarrow j$th row of $Q$            // Store data of row slice in temporary 1D arrays
    $Aux^{(row)} \leftarrow j$th row of $Aux$
    $Fluxes \leftarrow Fluxes +$ computeFluxes($Q^{(row)}$, $Aux^{(row)}$)   // Compute and accumulate fluxes
end
```

$Q_1^{(row)}, \dots, Q_{m_x}^{(row)}$ and $m_x - 1$ internal edges $e_1, \dots, e_{m_x-1}$. Also, each edge e_k has an associated Riemann problem with piecewise constant initial data $q_\ell = Q_k^{(row)}$, $q_r = Q_{k+1}^{(row)}$. The same idea is also applied to the grid columns. After extracting a slice from the grid, GeoClaw passes it to the subroutine `computeFluxes`, which iterates through the edges solving the respective Riemann problems and computing the output fluxes for the slice. This includes calls to the subroutines for the normal and transverse Riemann solvers and computation of second-order corrections. The computed fluxes are then accumulated into two-dimensional arrays that will be used to update the entire grid following the numerical update scheme (4.30).

Riemann solvers in GeoClaw

GeoClaw was designed in a customizable way that allows application developers to easily embed new Riemann solvers into its finite volume implementation. The subroutine `computeFluxes` calls other subroutines `rpn2` and `rpt2`, which are expected to solve the normal and transverse Riemann problems in a given one dimensional slice, respectively. To provide a new Riemann solver, developers only need to create implementations of those two subroutines that match their signatures and that return appropriate solutions to the Riemann problems. Then, it is relatively easy to switch between different solvers by managing which implementation of `rpn2` and `rpt2` is actually considered by the compiler. This is usually set up in the Makefile used to build the application.

Various Riemann solvers for a wide range of applications are available in the `Riemann` submodule of Clawpack's open source repository [14]. In this work we focus on vectorization of the three solvers that have been briefly described in Sections 4.4.1, 4.4.2 and 4.5.1. More specifically, for the single-layer shallow water equations we use the f-wave solver [4] and the augmented Riemann solver [29], while for the two-layer equations we use the two-layer solver proposed in [45, 46]. In addition to those three *normal* Riemann solvers, in this chapter we also experiment with vectorization of so-called *transverse* Riemann solvers for those equations (see Section 4.6), which compute flux corrections for two-dimensional simulations, based on the solutions of the normal Riemann problems.

6.2 Vectorization of the Numerical Routines

In this section we describe the changes that were necessary to achieve efficient vectorization of the numerical routines in GeoClaw, with particular emphasis on the Riemann solvers. These include a reorganization of the data arrays used in the finite volume scheme, the addition of compiler auto-vectorization directives to the computational loops and a few adjustments in the implementation of each solver.

6.2.1 Rearranging the Data layout

As discussed in Chapter 2, vectorization operates on sets of data stored contiguously, which is usually achieved by using the "structs of arrays" data layout (SoA). However, GeoClaw actually stores the grid data in three-dimensional arrays `Q(meqn,mx,my)` and `Aux(maux,mx,my)` (see Section 6.1.3). Since Fortran stores arrays in memory using column-major order, all cell-averaged components of q for a given cell (e.g., h, hu and hv for the single-layer shallow water equations) are stored contiguously in memory, while the averages of the same component in neighbor cells are not. Thus, these arrays actually resemble an "arrays of structs" layout (AoS), which is not suitable for vectorization.

Because efficient vectorization would require unit stride access to the same component of different cells, we made an effort to modify the data layout of those arrays. But unfortunately, modifying the array layout of the entire Clawpack software would not be practical, not only because of the software complexity, but also because such changes would affect several other simultaneous projects and applications. Clawpack and its submodules have been continuously developed as an open source project for over 20 years [47] with contributions of dozens of researchers working on separate branches, and such a huge modification would not be possible without extensive discussion within the Clawpack community and major refactoring of the code.

Instead, we decided to use the appropriate data layout only in the subroutines that compute the numerical fluxes (where we wish to apply vectorization), leaving other parts of the code unchanged. In practice, we still store the data of the two-dimensional grid in an AoS-like array `Q(meqn,mx,my)` as before, but we modified the temporary arrays used to store one-dimensional slices of the grid ($Q^{(col)}$ and $Q^{(row)}$ in Algorithm 6.1) – these slices are now stored in arrays structured as `Qcol(my,meqn)` and `Qrow(mx,meqn)`, which are suitable for vectorization over the cells/edges in each slice, since they are now stored in SoA-like order. The same approach is also applied to `Aux` and its one-dimensional slices.

This means that we now rearrange the data layout from SoA to AoS on-the-fly at every time step when extracting slices of the grid (steps 3–4 and 8–9 in Algorithm 6.1). These data movements certainly introduce an overhead in the execution time due to strided accesses to the memory. However, it should be noted that strided accesses were already performed previously when extracting column slices from the grid data. Also, the results obtained by our performance experiments presented in Section 6.3.3 show that the speedups achieved by vectorization of the numerical routines easily outweigh this overhead.

6.2.2 Effective and Efficient Compiler Auto-Vectorization

After we modified the layout of the arrays used to store the one-dimensional slices, the Intel Compiler was able to successfully auto-vectorize several loops in the subroutines that compute the numerical fluxes on the grid edges. Most of them are small loops that perform straightforward code, like accumulating the numerical fluxes for each cell or computing second-order corrections for the fluxes. For some of those loops, no further work was required to achieve compiler auto-vectorization. For others, simply annotating them with SIMD directives like `!$OMP SIMD` was enough. However, the main loops in the subroutines `rpn2` and `rpt2` (that repeatedly apply the normal and transverse Riemann solvers and are actually the most complex and time-consuming loops in the code) required slightly more complex annotations and some modifications in their code structure. In Code 6.1 we show the general structure that we used for those loops, to make them suitable for compiler auto-vectorization.

Recalling the discussion about data layout (see Section 6.2.1), notice that the arrays containing the input data for the Riemann problems are organized as SoA (see lines 22–25), which results in unit stride accesses between the same component for consecutive Riemann problems. The same can be said about the output arrays (lines 33–34), whose dimensions have also been reordered such that now the leftmost index is used to identify the Riemann problems. Thus, all the input and output arrays use data layouts that are suitable for vectorization.

Note that in addition to annotating the loop with the `!$OMP SIMD` directive, it was also necessary to declare the iteration-local variables (i.e., variables other than the arrays into which vectorization is applied) as `PRIVATE`, which effectively tells the compiler to replace these scalar variables with temporary arrays that match the SIMD width. Without this, the compiler would have been unable to vectorize the loop because of scalar assignments and data dependencies that are not allowed for vectorization. This was the main motivation for encapsulating most of the loop code into a separate subroutine `solver`, as this minimizes the number of variables that need to be declared here – local variables in external subroutines are treated as `PRIVATE` by default. Another advantage is that this provides a generic interface for achieving loop vectorization, regardless of which Riemann solver is actually implemented inside of `solver`.

We also use the compiler directive `!DIR$ FORCEINLINE` on the subroutine `solver` to make sure that it gets inlined into the loop and also vectorized. As discussed in Section 2.3.1, an alternative option to inlining would be to declare the subroutine as "SIMD-enabled" by adding the directive `!$OMP DECLARE SIMD` in its code. However, in our early experiments this approach resulted in considerably slower performance and it was discarded in favor of inlining.

Of course, whether the compiler will be able to actually vectorize this loop depends on the

Code 6.1: Example of how the main loop in the subroutine `rpn2` has been organized to allow auto-vectorization assisted by compiler directives. This example is based on the code used for the normal Riemann solvers for the single-layer shallow water equations, but very similar code structures are also used for the two-layer solver and for the transverse solvers (`rpt2`). Here, `Q` and `Aux` correspond to one-dimensional slices that have been extracted from the grid using Alg. 6.1. Note that there are no data dependencies across different loop iterations, which makes it safe to use the `!$OMP SIMD` directive.

```fortran
subroutine rpn2(Q, Aux, mx, meqn, maux, mwaves, [...], waves, speeds)
    ! Input:
    real    :: Q(mx, meqn)   ! Q data for cells in this 1D slice
    real    :: Aux(mx, maux) ! Aux data for cells in this 1D slice
    integer :: mx          ! Number of cells in this 1D slice
    integer :: meqn, maux  ! Number of components in Q and Aux, respectively
    integer :: mwaves      ! Number of waves generated by discontinuities for this PDE
    ! Output:
    real :: waves(mx-1, mwaves, meqn)  ! A set of waves for each edge
    real :: speeds(mx-1, mwaves) ! Speed of each wave
    ! Local variables:
    real :: hL,huL,hvL,bL, hR,huR,hvR,bR ! Data on Left/Right sides of the edges
    real :: w(mwaves,meqn), s(mwaves) ! Temporary storage for the Riemann solutions

    ! Loop through the edges in the slice (main loop):

    !$OMP SIMD PRIVATE(hL,huL,hvL,bL, hR,huR,hvR,bR, w,s)
    do i = 1, mx-1

        ! Copy data for this Riemann problem from Q and Aux.
        ! The left cell has index (i) and the right cell has index (i+1).
        hL  = Q(i,1)    ;  hR  = Q(i+1,1)
        huL = Q(i,2)    ;  huR = Q(i+1,2)
        hvL = Q(i,3)    ;  hvR = Q(i+1,3)
        bL  = Aux(i,1)  ;  bR  = Aux(i+1,1)

        ! Call Riemann solver for this problem (output is returned in 'w' and 's')

        !DIR$ FORCEINLINE
        call RiemannSolver(hL,huL,hvL,bL, hR,huR,hvR,bR, w, s)

        ! Copy solution ('w' and 's') to output arrays ('waves' and 'speeds')
        waves(i,:,:) = w(:,:)
        speeds(i,:)  = s(:)
    end do
end subroutine
```

implementation of the subroutine `solver`, as its code must adhere to a specific set of rules in order to be vectorizable (see Section 2.1). To achieve vectorization of our solver implementations and also to improve the vectorization performance, we made a few changes in their code. In particular, we removed “early exits” from all of them. In the original solver implementations, Riemann problems identified as “trivial” (both cells completely dry) were simply skipped, as their solution is always zero (zero-valued waves that do not affect the solution) – see the example in Code 6.2. However, skipping iterations is not possible with vectorization, because the SIMD model requires that the computation for sets of contiguous array positions advance simultaneously. In our vectorized implementations, these problems are now solved as usual and afterwards the correct solution for these problems (zero) is attributed to the output variables. An example is shown in Code 6.3. Note that this final step is necessary because the Riemann solvers may compute incorrect solutions for completely dry states, as their computations usually assume that $h_\ell > 0$ and $h_r > 0$. Because this approach uses simple post-processing of the

Code 6.2: Example implementation of `solver` for the single-layer shallow water equations with possibility of early exits, which should be avoided for vectorization. The solver implementation skips completely dry problems (problems with dry cells on both sides), immediately returning the correct solution (zero).

```fortran
subroutine solver(hL,huL,hvL,bL, hR,huR,hvR,bR, w, s)
    ! Input:
    real, intent(in) :: hL,huL,hvL,bL, hR,huR,hvR,bR ! Riemann problem data
    ! Output:
    real, intent(out) :: w(mwaves,meqn), s(mwaves) ! Waves and wave speeds
    ! Local:
    ! [...] (Other variables)

    ! Skip completely dry problems -> solution is zero
    if (hL < dryTolerance .and. hR < dryTolerance) then
        w(:,:) = 0.0
        s(:)   = 0.0
        return
    end if

    ! Compute solution of the Riemann problem ('w' and 's')
    ! [...] (Main code for the solver)

end subroutine
```

Code 6.3: Example implementation of `solver` for the single-layer shallow water equations without early exits, suitable for efficient vectorization. Note that at the end it is necessary to overwrite the content of the output arrays with the correct solution (zero), because the solver implementations usually assume that `hL` and `hR` are greater than zero and include divisions and square root computations involving those variables, which may lead to invalid solutions (`NaN`).

```fortran
subroutine solver(hL,huL,hvL,bL, hR,huR,hvR,bR, w, s)
    ! Input:
    real, intent(in) :: hL,huL,hvL,bL, hR,huR,hvR,bR ! Riemann problem data
    ! Output:
    real, intent(out) :: w(mwaves,meqn), s(mwaves) ! Waves and wave speeds
    ! Local:
    ! [...] (Other variables)

    ! Compute solution of the Riemann problem ('w' and 's')
    ! [...] (Main code for the solver)

    ! Set the correct solution for completely dry problems (zero)
    if (hL < dryTolerance .and. hR < dryTolerance) then
        w(:,:) = 0.0
        s(:)   = 0.0
    end if
end subroutine
```

output variables, it delivers better performance than introducing an extra if-then-else branch around a large chunk of code, which would force the compiler to use several additional masked operations and assignments.

Specifically for the augmented Riemann solver for the single-layer equations, we also removed the possibility of early exits from internal loops in the algorithm (such as the one used for implementing Newton's method) to avoid creating additional branches in the execution. Instead, the internal loops are now always executed for a given number of iterations, which is known beforehand by the compiler (usually only one iteration is applied).

Also the two-layer solver required additional modifications: to solve a 6×6 system of linear

Report 6.1: Excerpt from the optimization report provided by the Intel Fortran Compiler when compiling GeoClaw with the augmented Riemann solver for a KNL processor (with the compilation flag `-xMIC-AVX512`).

```
LOOP BEGIN at ../src/vec/rpn2_geoclaw.f90(118,7)
   remark #15301: OpenMP SIMD LOOP WAS VECTORIZED
   remark #15448: unmasked aligned unit stride loads: 8
   remark #15449: unmasked aligned unit stride stores: 12
   remark #15475: --- begin vector cost summary ---
   remark #15476: scalar cost: 2355
   remark #15477: vector cost: 988.870
   remark #15478: estimated potential speedup: 2.280
   remark #15486: divides: 61
   remark #15488: --- end vector cost summary ---
LOOP END
```

equations, it previously used the subroutine `dgesv` from the linear algebra library LAPACK [1]. However, the Intel Compiler was not able to inline and auto-vectorize this call to an external library, resulting in loss of vectorization performance. For that reason, we replaced this with a local implementation of the LU decomposition algorithm [69], which could then be inlined into the loop and efficiently auto-vectorized. In addition, in order to reduce the total number of execution branches, we now require the choice of method used for eigenspace decomposition to be defined in compilation time (four different methods are provided by the solver implementation, as discussed in Section 3.3 of [46]).

By checking the optimization reports provided by the Intel Fortran compiler, we confirmed that the loops annotated with `SIMD` directives have indeed been auto-vectorized, for all variations of Riemann solvers considered in this work. In Report 6.1 we show part of the report obtained when compiling GeoClaw with the augmented Riemann solver for the KNL processors – this excerpt refers to the loop shown in Code 6.1. The compiler reports that all memory accesses are unmasked, aligned and unit-strided, which is optimal for vectorization. In addition, it also estimates a potential speedup of 2.28$\times$ for this loop – however, in our performance results presented in Section 6.3 we experienced considerably higher speedups. For the other Riemann solvers and also for the Haswell architecture we obtained similar reports, all of them confirming efficient auto-vectorization of those loops.

6.2.3 Comparison with Related Work

Previous related work has also reported successful compiler auto-vectorization of the f-wave solver for the single-layer shallow water equations (e.g. [3, 44, 74]). In addition to the f-wave solver, Bader et. al. [3] were also able to vectorize the augmented Riemann solver, but intrinsic functions were necessary because the compiler was not able to auto-vectorize the loop, even after they annotated it with the C/C++ compiler directive `#pragma simd` (equivalent to the `!DIR$ SIMD` directive discussed in Section 2.3).

Before implementing the vectorization approach used in this work, we had previously published results involving auto-vectorization of those two Riemann solvers using a different approach [23], based on redesigning the solver subroutines to operate on multiple Riemann problems simultaneously, and getting the compiler to vectorize multiple loops inside them independently from each other. In addition to requiring major code modifications in the implementations, this approach was disadvantageous compared to our current one because it inhibits several kinds of compiler optimizations, and each additional vectorized loop introduces some overhead for its execution, limiting the performance improvement. In comparison to the previous approach, our current one is easier to implement, uses a single vectorized loop and achieves higher performance.

In contrast to the approaches used by those previous work that achieved vectorization of the augmented Riemann solver, the approach described here uses simple compiler auto-vectorization of the main solver loop, annotated with the `!$OMP SIMD` directive (available in the OpenMP 4.0 standard and later versions). As noted in Section 2.3, in our experience the non-OpenMP directive `!DIR$ SIMD` was not as successful as the OpenMP directive `!$OMP SIMD`, and therefore we decided to adopt the latter in our implementations. With this compiler directive, which is a relatively recent addition to the OpenMP standard, the Intel Fortran Compiler is able to successfully auto-vectorize loops that repeatedly apply the augmented Riemann solver, despite its high complexity.

Our current approach is based on the work described in [44], which reports vectorization of the f-wave and transverse Riemann solvers for the single-layer shallow water equations in GeoClaw. We extended that research by including support for vectorization of the augmented Riemann solver, as well as for the normal and transverse Riemann solvers for the two-layer shallow water equations. Although other authors [35] have worked on a GPGPU implementation of solvers for the multi-layer shallow water equations, our research is, to the best of our knowledge, the first one to report successful vectorization of Riemann solvers for multi-layer shallow water equations.

6.3 Performance Results

In this section we evaluate the performance of our vectorized implementations on the high-performance experimental platforms that we described in Chapter 5. First, we describe the simulation scenarios we used for each variation of the shallow water equations. Then, we evaluate the performance of the solver subroutines individually, comparing the vectorized and non-vectorized versions and also evaluating how efficiently they exploit the available computational resources. Afterwards, we assess the benefits of vectorization to the finite volume applications in general, particularly with respect to reducing their time-to-solution.

6.3.1 Simulation scenarios

Chile 2010 Tsunami

To test our solvers for the single-layer shallow water equations, we simulate a real tsunami event that took place in the Pacific Ocean in February 2010 and reached the coast of Chile and Peru. We show two snapshots of these simulations in Fig 6.2. In these simulations we use topography data with a resolution of $\frac{1}{6}$ degree both in latitude and longitude (roughly 18.5 km), obtained from the ETOPO2 data set made available by the National Geophysical Data Center (NGDC) [55, 56]. As initial conditions for the tsunami wave, we apply a static displacement to the water and to the bathymetry, obtained following the Okada model [57] and using data for the respective earthquake, made available by the United States Geological Survey [68]. The interested reader is referred to [39] for more details and numerical considerations regarding the simulation of the Chile 2010 Tsunami.

For the performance results presented in the following, we used three refinement levels: a level 1 grid composed of 100×100 cells and refinement factors of 6×6 and 8×8 for levels 2 and 3, respectively. This setup produces cell resolutions of roughly 67 km on the coarsest level and 1.4 km on the finest level. Our results consider a simulation of the first six hours after the earthquake that generated the tsunami. In each simulation, roughly 14.1 billion cell updates were computed.

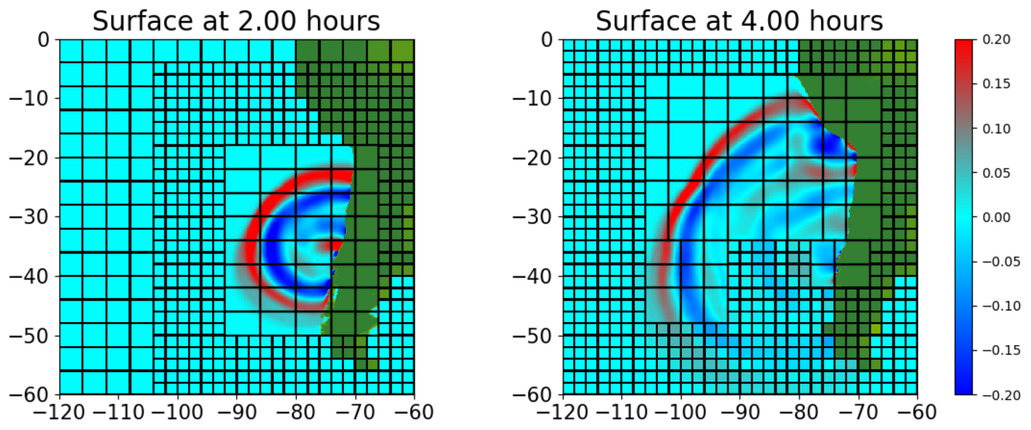


Figure 6.2: Simulation of the Chile 2010 tsunami with GeoClaw. Note that the simulation illustrated in the snapshots does not use the same resolutions as the ones used in our actual experiments. In this simulation we use a level 1 grid with 15×15 cells and refinement factors of 2×2 and 8×8 for the second and third levels, respectively. For clarity, the internal edges of the third level grids are omitted from the illustrations.

Parabolic Bowl-Shaped Lake

For the experiments with the two-layer shallow water equations, we use one of the artificial scenarios available in the GeoClaw repository [14]: the *parabolic bowl-shaped lake.* In this scenario, waves generated by a circular hump of water propagate over a bowl-shaped bathymetry, with the water being initially split into two layers 20 meters below the baseline level $b = 0$. More specifically, this scenario is composed of a square domain $[-100, 100] \times [-100, 100]$ (coordinates are given in meters), where the bathymetry and the depths of the two layers of water are defined as:

$$b(x, y) = 10^{-2} \cdot (x^2 + y^2) - 80, \tag{6.1}$$

$$h_2(x, y) = \max(0, -20 - b(x, y)), \tag{6.2}$$

$$h_1(x, y) = \begin{cases} 4e^{-10^{-2}\cdot(x^2+y^2)} - b(x, y) - h_2(x, y) & \text{if } x^2 + y^2 < 10 \\ \max(0, -b(x, y) - h_2(x, y)) & \text{if } x^2 + y^2 \geq 10. \end{cases} \tag{6.3}$$

This kind of bathymetry is often used in analytical scenarios that serve as benchmarks to evaluate the quality of numerical schemes for shallow water models, especially with respect to inundation [65]. In this case, this scenario is especially useful to assess wetting and drying of individual layers, as it contains regions in which the top layer is wet and the bottom layer is dry. For illustration, in Fig. 6.3 we show a vertical section of the initial conditions for this scenario.

Like in the experiments with the Chile 2010 Tsunami, in the two-layer simulations we also used three refinement levels, but with different refinement ratios. More specifically, we use 40×40 cells in the level 1 grid and refinement ratios of 6×6 for both layers 2 and 3. During each simulation, approximately 4.2 billion cell updates were computed.

6.3.2 Vectorization Performance

Since our goal is to evaluate the effectiveness of vectorization, initially we consider only the performance of each solver individually; the overall simulation performance will be addressed later. Thus, for the following results we considered only the times spent by the solver subroutines, and used them to compute the performance of each solver. These are given as *Riemann problems solved per second.*

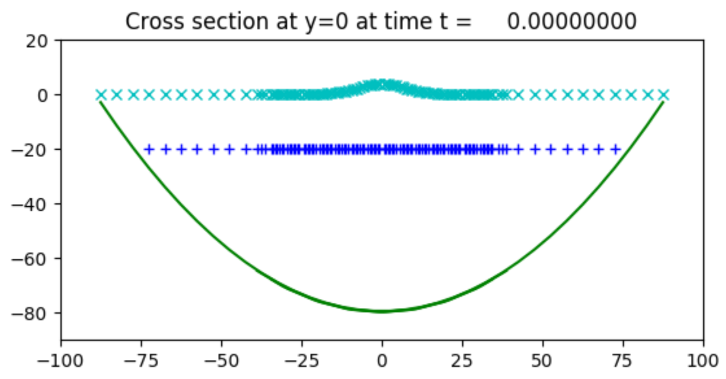


Figure 6.3: Vertical-section visualization of the "parabolic bowl-shaped lake" scenario at $t = 0$. The green line depicts the bowl-shaped bathymetry and the cross (cyan) and plus (blue) signs show the elevation of the first and second layers of water, respectively. Note that the cell concentration is much higher in the region close to the hump of water at the center due to adaptive mesh refinement.

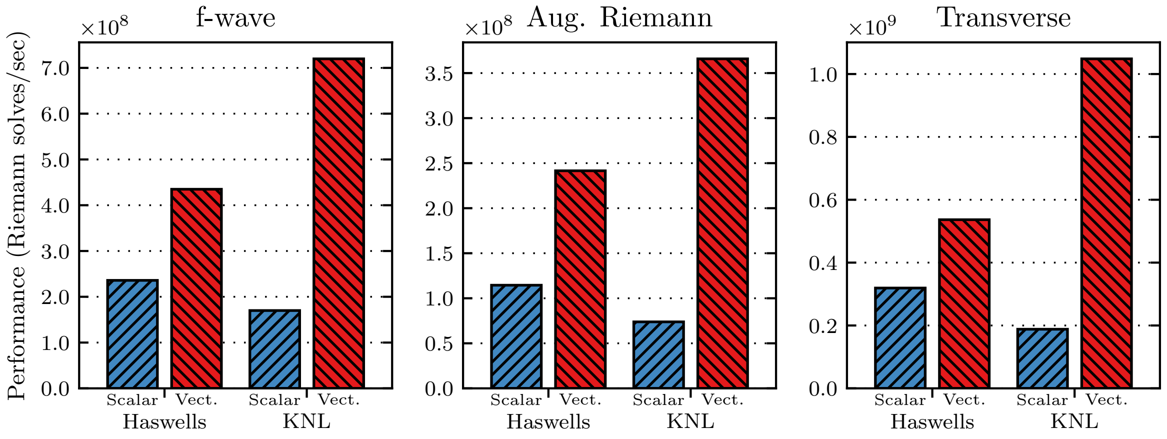


Figure 6.4: Performance of the Riemann solvers for the *single-layer* shallow water equations, before and after applying vectorization.

In Fig. 6.4 we present the performance measurements of the three Riemann solvers for the single-layer shallow water equations (f-wave, augmented Riemann and the transverse Riemann solvers). These results confirm the effectiveness of our vectorization approach on both machines. As expected, the benefits of vectorization are much more noticeable on the KNLs than on the Haswells (4.2–5.6× vs. 1.7–2.1×). It is also interesting that the augmented Riemann solver obtained slightly higher speedups than the f-wave solver, despite its higher complexity. That happens because it is much more compute-intensive than the f-wave solver, which generally leads to greater vectorization speedups.

The performance measurements for the solvers for the two-layer equations are presented in Fig. 6.5. In this case we only have one version for the normal Riemann solver, so we only distinguish between normal and transverse solvers. Again the improvements are much greater on the KNLs (2.3–2.7× vs. 1.3–1.4×). It is clear that vectorization with 256-bit SIMD instructions achieves only minor performance gains with such complex solvers. On the other hand, 512-bit instructions are able to deliver reasonable speedups, although still considerably smaller than those obtained with the single-layer solvers.

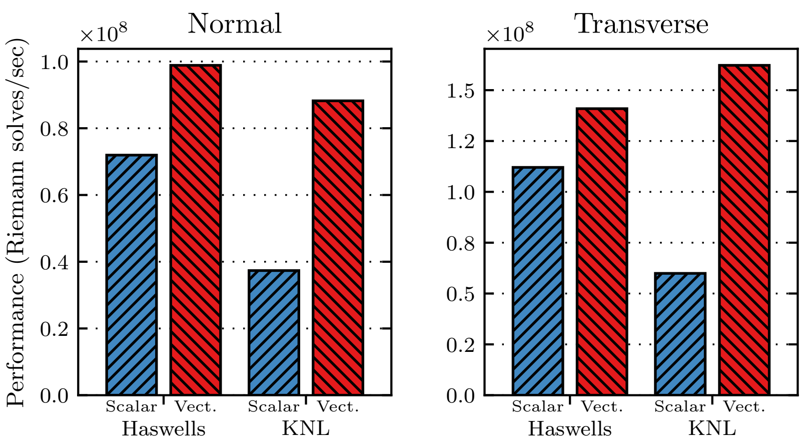


Figure 6.5: Performance of the Riemann solvers for the *two-layer* shallow water equations, before and after applying vectorization.

Table 6.1: Performance of the vectorized solvers in GeoClaw. For each solver, we list the speedup achieved by introducing vectorization and the floating-point operations throughput of the vectorized versions (in GFlops/s). The percentages in parentheses show how the achieved performances compare to the maximum attainable performance of each machine (measured with the Flop/s benchmark described in Section 5.1).

Equations	**Machine**	**Solver**	**Speedup**	**GFlop/s**
Single-layer	Haswells	f-wave	1.8×	49.7 (32%)
		Aug. Riemann	2.1×	63.7 (41%)
		Transverse	1.7×	53.8 (34%)
	KNL	f-wave	4.2×	82.2 (11%)
		Aug. Riemann	5.0×	96.5 (13%)
		Transverse	5.6×	105.1 (15%)
Two-layer	Haswells	Normal	1.4×	33.7 (22%)
		Transverse	1.3×	36.6 (23%)
	KNL	Normal	2.4×	30.0 (4%)
		Transverse	2.7×	42.1 (6%)

We summarize the speedups obtained on these experiments in Table 6.1. There we also list estimates of the floating-point operations per second rate (GFlop/s) for the vectorized solvers on each machine, obtained with the help of the PAPI interface [54]. We used PAPI to measure the average number of floating-point operations performed by each solver, considering the non-vectorized versions. We then estimated the throughput of the vectorized versions by assuming that they perform the same number of operations as the original implementations. Note that this is a conservative assumption, since the vectorized codes actually perform many additional operations, e.g. because of masked operations used to handle branches. Thus, in practice, we only consider "useful" floating-point operations, ignoring those additional operations introduced by vectorization.

Comparing the achieved throughputs with the maximum performance measured in each

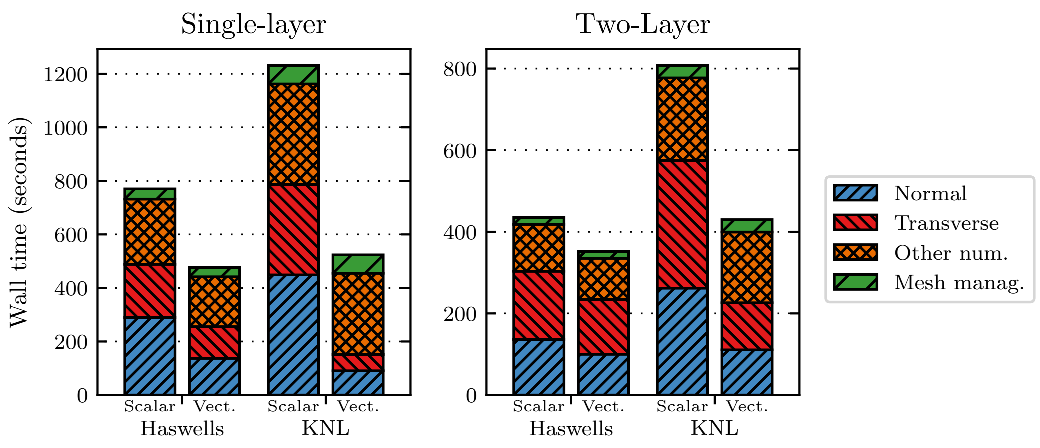


Figure 6.6: Wall time split into components for the entire simulations, before and after vectorization of the Riemann solvers.

machine (see Table 5.1), we observe that the single-layer solvers achieve roughly 32–41% of the attainable performance on the Haswells and 11–15% on KNLs. Since the solvers are much more complex than the benchmark used (including if-then-else branches, division and square-root operations, etc.), these numbers indicate high utilization of the computational resources.

A similar analysis can be made for the two-layer solvers, which achieve 22–23% and 4–6% of the attainable performance on the Haswells and the KNL, respectively. It is evident that these solvers benefit less from vectorization than the single-layer ones and perform considerably slower. This can be attributed to the even higher complexity of the two-layer solvers (in comparison with the single-layer ones), where large if-then-else branches are necessary to deal with the four times higher number of dry/wet combinations (see Section 4.5.1). Because such branches can diminish the benefits from vectorization substantially, the smaller speedups and worse performance obtained are not surprising. Also, these results indicate that the vectorized two-layer solvers might be improved by minimizing the number and size of the branches in their code and investigating this is left as a suggestion for future work.

6.3.3 Simulation Performance and Time-To-Solution

Now we consider the influence of vectorization to the entire execution time of the simulations. In Fig. 6.6 we plot the execution (wall) times spent by the simulations using the original and the vectorized solver implementations. We divide the execution time into four components: “Normal solver”, “Transverse solver”, “Other numerical routines” and “Mesh management”. The first two correspond to the Riemann solvers discussed previously. “Other numerical routines” includes all execution time necessary to update the solution values, except for the two Riemann solvers. These routines are basically composed of the steps that precede and succeed the Riemann solvers. Initially, they are responsible for fetching the cell data and reorganizing them as one-dimensional slices to be used as input for the solvers (Alg. 6.1). Afterwards, they apply second-order corrections and limiters to the Riemann solutions and use them to update the cell quantities, following (4.30). Lastly, “Mesh management” comprises the operations necessary for managing adaptivity, such as refining/coarsening cells, merging contiguous patches, etc. Note that, for the single-layer equations, here we consider only the augmented Riemann solver and omit the results obtained with the f-wave solver, since the measurements obtained in those experiments are basically the same (except for the “Normal” component).

Like the speedups computed individually for each solver, the overall speedups are significantly higher on the KNLs (2.4× and 1.8× for the single-layer and two-layer equations, respectively) than on the Haswells (1.6× and 1.2×). However, the execution times for the vectorized codes are roughly the same on both architectures, especially for the single-layer equations, even though the vectorized single-layer solvers perform much faster on KNLs. This is due to the other two components, which are clearly executed much faster on the Haswells than on the KNLs.

Particularly, the "Other numerical routines" component now consumes most of the execution time on the KNLs. As described above, this component is the one responsible for fetching the grid data from the main memory. Compared to the Riemann solvers, the number of floating-point operations performed in this step is much smaller, such that it is limited by the memory bandwidth and by synchronization of threads. It should be noted, however, that the execution times for this component did not increase on the vectorized versions, despite the overhead introduced by rearranging the array layout from AoS to SoA during every time step (as described in Section 6.2.1). In fact, there is even a small time reduction due to the auto-vectorization of some loops in this component (made possible by the new data layout). This clearly shows that the benefits of vectorization easily outweigh this overhead, especially when also considering the performance improvements of the Riemann solvers. Nevertheless, it is still expected that modifying the data layout for the entire application, although not an easy task, might be beneficial for its overall performance.

Clearly, also the "Mesh management" component is responsible for a considerable fraction of the execution time on the KNL. This happens mainly because this component does not exploit vectorization and does not scale well on hundreds of threads, as would be necessary for high performance with KNLs. As such, future efforts on optimizing these applications for the KNLs (or many-core architectures in general) should consider ways to improve scalability and parallel efficiency for this component.

6.4 Conclusions

In this chapter, we proposed and evaluated vectorized implementations of various Riemann solvers for the single-layer and two-layer shallow water equations. Compared to previous related work, the approach we use is much simpler and more portable, as it is based on auto-vectorization guided by an OpenMP compiler directive that proved itself able to successfully vectorize highly complex loops.

We experienced substantial speedups on the solver routines, especially on the modern KNL processors with 512-bit SIMD instructions. As expected, smaller but still considerable improvements were also reported for the Haswell architecture, able to perform 256-bit instructions. However, although the solvers perform considerably faster on the KNL, the overall execution time is still comparable on both machines, because other parts of the simulation code do not scale to hundreds of threads and/or do not exploit vectorization.

A component analysis of the execution times showed that the benefits of vectorization clearly compensate for the overhead introduced to make the simulation data structures suitable for SIMD instructions. It also reveals that, while the compute-intensive Riemann solvers used to be the most time-consuming routines, now most of the time is spent on other more memory-intensive components of the simulation code. These results suggest that future optimizations efforts on GeoClaw should focus mainly on those components, instead of the Riemann solvers that have already been addressed in our work.

7 Sam(oa)^2: Tree-Structured Adaptive Triangular Meshes

After achieving successful vectorization of the numerical methods in GeoClaw, we applied the same vectorization approach to our PDE framework sam(oa)^2 [49, 63]. However, sam(oa)^2 was designed for fine-grained cell-wise adaptivity, which poses an obstacle for vectorization over grid cells or edges, due to the irregular data structures used. For that reason, it was first necessary to modify the adaptive meshes in sam(oa)^2 to use a coarser-grained structure, by replacing single cells with patches of uniformly refined cells in the leaves of the refinement tree. This not only allowed effective vectorization of the numerical routines in sam(oa)^2, but also brought further increases in performance due to the more regular data structures and memory accesses and to the reductions in the size and complexity of the implicit refinement trees.

Our initial achievements in this work with sam(oa)^2 have been described in one of our publications [23]. However, since then we have completely reformulated our implementations. Most importantly, we replaced the previous vectorization approach with the much more efficient approach that we developed in GeoClaw (which was described in Section 6.2). Also, we recently modified the design of the patch-based implementation, which now matches the grid generation algorithm used in sam(oa)^2 and considerably simplifies the implementation of refinement and coarsening schemes. Therefore, most of the content in this chapter actually refers to new and so far unpublished work.

We start this chapter by introducing the basic ideas behind the framework sam(oa)^2 in Section 7.1, where we discuss its algorithms for managing adaptive meshes, numerical simulations and various HPC features. Then we describe our work on modifying the structure of its adaptive meshes and on applying vectorization to its numerical routines in Section 7.2. Later we present the results of our performance experiments in Section 7.3 and give a summary of our most important findings in Section 7.4.

7.1 Adaptive Triangular Meshes Based on Sierpinski Space-Filling Curves

Sam(oa)^2 [49,63] is a simulation framework that supports creation and management of adaptive triangular meshes for implementing high-performance finite element, finite volume and discontinuous Galerkin methods. In addition to adaptive mesh refinement, it provides various HPC features, like memory- and cache-efficient traversals over the mesh elements, multi-level parallelism and dynamic load balancing. All those features are based on organizing the mesh elements and performing domain decompositions following the Sierpinski space-filling curve that can be implicitly found in the mesh structure. Before describing our work on adding vectorization to sam(oa)^2 in Section 7.2, in this section we introduce the main ideas and algorithms used for managing the adaptive meshes and HPC features in sam(oa)^2. In particular, we discuss only

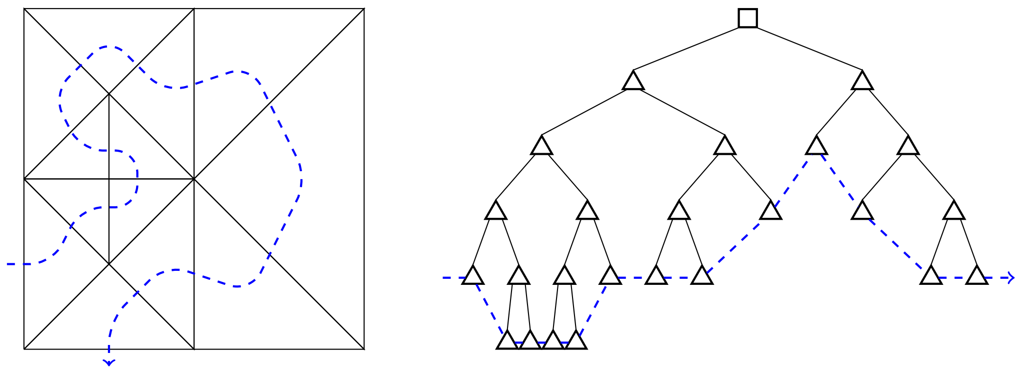

(a) Adaptive mesh **(b)** Refinement tree

Figure 7.1: Example of a small mesh generated following the newest-vertex-bisection method and its corresponding refinement tree. The blue dashed lines show the Sierpinski space-filling curve induced on the finest cells in the mesh (leaves of the refinement tree). The order given by the curve is used in sam(oa)^2 for various purposes, such as efficient mesh storage and traversals.

implementation details that are relevant for finite volume methods, as other kinds of numerical methods are beyond the scope of this thesis. For further details on sam(oa)^2 and discussions regarding implementation of other kinds of numerical methods, refer to [49, 70].

In the following, we often use the single-layer shallow water equations as an example application when discussing implementation of a finite volume scheme in sam(oa)^2. In general, extending those examples for the two-layer equations can be accomplished by performing similar operations individually for each layer and by applying the Riemann solver for the two-layer equations instead of the single-layer solver.

7.1.1 Adaptive Meshes and Memory-Efficient Traversals

Adaptive mesh refinement in sam(oa)^2 is based on the tree-structured approach that was discussed previously in Section 3.1.2. Its adaptive meshes are generated by recursive subdivision of triangular cells, following the newest-vertex-bisection method [53]. Starting with a square domain split into two isosceles right triangles, triangular cells may be individually refined by splitting them in half, with the addition of a new edge connecting the right angle vertex to the midpoint of the hypotenuse. Note that this splitting method always produces isosceles right triangles. Refinement of cells can then be recursively performed until the desired resolutions are achieved in all areas of interest. As an example, consider the small mesh shown in Fig. 7.1(a). Also, cells can be dynamically refined or coarsened following the same method after every numerical time step, depending on the requirements for resolution in each domain region. An advantage of using this method is that the resulting meshes do not require special treatment of hanging nodes, as it is always possible to obtain conforming meshes (i.e., without hanging nodes) by performing additional refinements.

Another advantage of using the newest-vertex-method for mesh generation is the fact that it always produces meshes whose elements (cells, edges and vertices) can be inherently organized following the order induced by the Sierpinski space-filling curve – illustrated by the blue dashed lines in Fig 7.1. The so-called *Sierpinski order* represents the order in which the leaves of the binary refinement tree (shown in Fig. 7.1(b)) are traversed by a depth-first search algorithm. This order provides a domain decomposition that is used in sam(oa)^2 for various purposes. In particular, it is used to organize the mesh elements linearly, such that the full refinement tree does not need to be stored explicitly, leading to low computation costs for storage and management of the adaptive mesh. Instead, it is only necessary to store the tree leaves, which effectively contain the simulation data. In the case of finite volume methods, the leaves are used to store the cell-averaged solution Q for each cell, as well as other cell-specific data required for the simulation (e.g., bathymetry for the shallow water equations).

Organizing the adaptive mesh following the Sierpinski space-filling curve also allows to iterate through all elements in the mesh in a memory- and cache-efficient way [2]. The framework implements a cache-oblivious traversal scheme based on stack and stream data structures that explores the data locality of the mesh elements, improving the rates of cache hits [70]. The complex algorithm that handles grid traversals is effectively hidden from application developers via a system of element-oriented kernels [52] that allow customization of the numerical and mesh management algorithms for various applications with different requirements [51]. In Section 7.1.4, we show some example implementations for such kernels in the context of the single-layer shallow water equations.

As a disadvantage of the strategy for adaptive mesh refinement implemented in sam(oa)^2, we mention that the Sierpinski space-filling curve is in essence a two-dimensional curve, so there is no straightforward extension of the method for more dimensions. Therefore, this adaptivity strategy is limited to refinements in two dimensions. Nevertheless, three-dimensional simulations can still be implemented by introducing a vertical domain with static refinement, as done in the work described in [50].

7.1.2 Parallelism and Dynamic Load Balancing

Sam(oa)^2 supports multiple levels of parallelism. MPI is used for parallelism in distributed memory, while OpenMP is used for shared memory. Vectorization is a recent addition by this work, and its implementation is discussed in Section 7.2. The hybrid MPI+OpenMP parallelization scheme in sam(oa)^2 is also based on the order of the mesh elements given by the Sierpinski space-filling curve, as the linear storage scheme provides a linear domain decomposition of the two-dimensional mesh. The linear list of cells in the mesh is divided into multiple *sections* with similar *loads*[1], which can be assigned to different processes/threads and processed in parallel with low communication requirements. Previous work has shown that this parallelization strategy scales well on up to 8 000 cores [52].

Because the mesh is dynamically adaptive, the number of cells in each section can increase or decrease after performing refinement and coarsening operations, leading to load imbalances that can considerably decrease the parallelization efficiency. Therefore, sam(oa)^2 performs dynamic load balancing after every remeshing procedure, attempting to properly distribute the computational load among all processes and threads. In the case of shared-memory parallelism, load redistribution can be performed by trivially repartitioning the sections such that they again have similar loads. For distributed memory, redistributing the load requires more complex schemes, as transferring cell data between different MPI processes is necessary.

To determine an optimal load distribution, sam(oa)^2 uses heuristics [59] to approximately solve the NP-complete optimization problem that strives to minimize the maximum load per section, known in literature as *chains-on-chains partitioning*. Originally developed for homogeneous systems (i.e., processors/cores with similar processing rates), the approach used in sam(oa)^2 has been recently extended to also allow proper load distribution on heterogeneous systems [23]. Furthermore, recent work [62] has developed a strategy for reactive work steal-

[1]When setting up a simulation, users can decide which definition for "load" they want to use: the load of a section can either directly reflect the number of cells in it, or be an estimate of the execution time required for its processing.

Code 7.1: Definition of the data stored for each cell in sam(oa)^2. This example is based on the single-layer shallow water equations.

```
type num_cell_data_pers
    real :: h, hu, hv ! Components of the solution q
    real :: b ! Bathymetry
end type
```

ing in distributed memory that achieves better load distributions than the heuristics for the chains-on-chains partitioning problem.

7.1.3 Simulation Data Structures

Sam(oa)^2 requires application developers to declare the data structures that will be used to store the cell-averaged quantities for each cell, which is done by declaring the data type `num_cell_data_pers` (consider the example in Code 7.1). Similar declarations are also required for the edges, which are used during the numerical time steps to store representations of the data from their adjacent cells, as well as the solutions of their respective Riemann problems. Each element in the mesh is represented by a complex data structure that stores not only the simulation data as defined by the application developer, but also many important geometrical data that is required for the meshing algorithms in sam(oa)^2 (e.g., coordinates of each vertex, connectivity information, etc.). The simulation data for the entire mesh is then stored as a linear list containing such data structures, following the Sierpinski order. This results in a storage scheme with an AoS layout that is not suitable for vectorization (see the related discussion in Section 2.2). Thus, in order to successfully apply vectorization, it was first necessary to modify this data layout, as we will describe later in Section 7.2.

7.1.4 Grid Traversals for Finite Volume Methods

New simulation codes can be developed within sam(oa)^2 by defining so-called *grid traversals*, during which kernels implemented by the application developer are applied to the mesh elements. In general, numerical simulations on adaptive meshes can be developed following the structure shown in Alg. 7.1, which uses three kinds of traversals, each with a different purpose:

1. `initialization` traversal: defines the initial state of the mesh;
2. `numerical` traversal: effectively applies a time step of the numerical method;
3. `adaptive` traversal: restructures the mesh performing refinement/coarsening of cells;

However, this is not strictly enforced and application developers have freedom to omit one or more of those traversals or to create additional ones. For instance, some applications may benefit from splitting the `numerical` traversal into two or more separate traversals.

The `initialization` and `numerical` traversals are considered *static* grid traversals, since they only operate on the data stored in the mesh elements, without altering the mesh structure. On the other hand, the `adaptive` traversal is a *dynamic* traversal that refines and coarsens cells, guarantees mesh conformity and applies dynamic load balancing.

Defining a grid traversal requires providing implementations of element-oriented *kernels*, which may vary depending on the purpose of the traversal. Each kernel is efficiently applied to the entire mesh in a cell-by-cell, edge-by-edge or vertex-by-vertex basis, depending on its purpose. In the following we describe the implementations of traversals and kernels that we use to develop finite volume simulations, with examples in the context of the single-layer shallow

Algorithm 7.1: General structure that can be used to implement numerical simulations in sam(oa)^2, based on three different kinds of traversals. This was the structure used for all implementations considered in this thesis. However, using exactly this structure is not required, as developers have flexibility to implement different logical sequences.

```
   // Mesh generation: starting with two cells, iteratively initialize and refine cells
1  Initialization traversal  // Initialize data in the two cells and flag them for refinement
2  while desired refinement was still not achieved somewhere in the mesh do
3      Adaptive traversal  // Refine/coarsen cells that have been flagged during initialization
4      Initialization traversal  // Repeat the data initialization process for the updated mesh
5  end
   // Numerical time-stepping phase
6  while simulation is not over do
7      Numerical traversal      // Apply a numerical time step, advancing the solution in time
8      Adaptive traversal   // Refine/coarsen cells flagged by the previous numerical traversal
9  end
```

Algorithm 7.2: Logical structure of the `numerical` traversal for finite volume schemes, which applies three different kernels to the mesh elements.

```
    Input: grid
    Output: grid
1   foreach cell in grid do
2       foreach edge in cell do
3           call CellToEdgeKernel(cell,edge)      // Stores copies of each cell's data in its edges
4       end
5   end
6   foreach edge in grid do
7       call SkeletonKernel(edge)                        // Solves the Riemann problem at each edge
8   end
9   foreach cell in grid do
10      call CellUpdateKernel(cell)                           // Updates the solution in each cell
11  end
```

water equations. We note that sam(oa)^2 actually defines further types of kernels that can be used to create other types of numerical algorithms, such as finite element methods. However, they are not mentioned here, since this work is only concerned with finite volume methods.

Static Grid Traversals

Static grid traversals can use various kernels that operate on the mesh elements in different ways. However, it is not necessary for a traversal to implement all existing kernels. For instance, the `initialization` traversal consists only of an `element` kernel, which is applied to all cells in the mesh. It is used to define the initial data in each cell and to flag cells for further refinement, where required. As shown in Alg. 7.1, this is usually followed by an `adaptive` traversal, and the process is repeated until all regions in the domain have been sufficiently refined.

The `numerical` traversal has a more complex structure, requiring three kernels to properly apply a time step of the finite volume scheme, as shown in Alg 7.2. In the first step, the `cell-to-edge` kernel is called for every combination of adjacent cells and edges, storing representations of the cell data in its edges. For the applications studied in this work, those representations consist of simple copies of the cell data. After this step, every edge in the mesh will have copies of the data from its two adjacent cells, which define its respective Riemann problem.

Afterwards, the `skeleton` kernel is called for each edge and typically applies a Riemann

Code 7.2: Implementation of a *skeleton kernel* that is applied to all internal edges, using a Riemann solver to solve the problem defined by the data stored in each adjacent cell (`rep1` and `rep2`) and stores the computed solutions in the form of cell updates to be applied to each adjacent cell (`update1` and `update2`).

```fortran
subroutine skeleton_kernel(edge, rep1, rep2, update1, update2, maxWaveSpeeed, [...])
    ! Input variables
    type(t_edge_data), intent(in) :: edge ! Includes data for this edge's geometry
    type(num_cell_rep), intent(in) :: rep1, rep2 ! Representations of the cells' data

    ! Output variables: numerical fluxes for each adjacent cell and max. wave speed
    type(num_cell_update), intent(out) :: update1, update2
    real, intent(out)                  :: maxWaveSpeed

    ! Local variables
    real :: normal(2) ! Vector that is normal to the edge
    real :: hL,huL,hvL,bL, hR,huR,hvR,bR ! Data in cells at each side of the edges
    real :: waves(3,3), speeds(3) ! Data of waves computed by the Riemann solver
    real :: flux_hL, flux_huL, flux_hvL ! Numerical fluxes for the 'left' cell
    real :: flux_hR, flux_huR, flux_hvR ! Numerical fluxes for the 'right' cell

    ! Extract Riemann problem data from input data structures
    normal = edge%transform_data%normal
    hL = rep1%h; huL = rep1%hu ; hvL = rep1%hv ; bL = rep1%b
    hR = rep2%h; huR = rep2%hu ; hvR = rep2%hv ; bR = rep2%b

    ! Call Riemann solver
    call RiemannSolver(normal, hL,huL,hvL,bL, hR,huR,hvR,bR,
      ↪ flux_hL,flux_huL,flux_hvL, flux_hR,flux_huR,flux_hvR, maxWaveSpeed)

    ! Copy numerical fluxes to output data structures
    update1%h = flux_hL ; update1%hu = flux_huL ; update1%hv = flux_hvL
    update2%h = flux_hR ; update2%hu = flux_huR ; update2%hv = flux_hvR
end subroutine
```

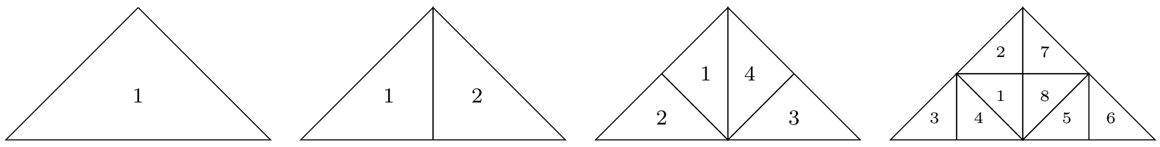
solver to its problem, computing the numerical fluxes over the edge and storing them in the edge data. We show an example implementation of the `skeleton` kernel for the internal edges in Code 7.2. Additionally, the `numerical` traversal also requires implementing a `boundary-skeleton` kernel. This kernel is applied to the edges at the domain boundary, which are only adjacent to one cell, typically simulating some kind of boundary condition, usually either reflecting or absorbing the waves that reach the domain boundary.

In the final step, the `cell-update` kernel loops through all cells and updates them using the numerical fluxes that have been computed and stored in the edges by the `skeleton` kernel. This kernel is also responsible for flagging cells for refinement or coarsening depending on the resolution required for each region. To update the cell, it typically applies an update scheme like the ones discussed in Section 4.6. However, since those assume the use of rectangular grids, it was necessary to slightly modify the update scheme to match the triangular cells used in sam(oa)^2. More specifically, we use a modified version of the donor-cell upwind method (4.29)[2] that reads

$$Q_i^{(n+1)} = Q_i^{(n)} + \frac{\Delta t}{V_i} \sum_{j \in \mathcal{N}(\mathcal{C}_i)} \mathcal{F}(Q_i^{(n)}, Q_j^{(n)}), \tag{7.1}$$

where $\mathcal{C}_i$ is the cell being updated, V_i is the area of $\mathcal{C}_i$, $\mathcal{N}(\mathcal{C}_i)$ is the set of neighbors of $\mathcal{C}_i$ and $\mathcal{F}(Q_i^{(n)}, Q_j^{(n)})$ is the numerical flux between cells $\mathcal{C}_i$ and $\mathcal{C}_j$.

[2]Currently, sam(oa)^2 does not support the corner-transport upwind method (4.30), as the element-oriented design and the use of triangular cells further complicate the computation of transverse corrections.



(a) $d = 0$ **(b)** $d = 1$ **(c)** $d = 2$ **(d)** $d = 3$

Figure 7.2: Patches with refinement depths ranging from $d = 0$ to $d = 3$. A patch with refinement depth $d > 0$ is obtained by splitting all cells in a patch with depth $d - 1$ in half. Each cell is identified with a unique number, which we use in our implementation to store the data of all cells sequentially.

Dynamic Grid Traversals

Dynamic grid traversals are used to apply adaptive mesh refinement to the simulation mesh. Most importantly, the `adaptive` traversal requires implementation of a `refine` kernel and a `coarsen` kernel, which define how cells should be refined and coarsened when necessary. The former is responsible for splitting one cell into two finer cells, and usually consists of directly copying the data from the original cell to the two new finer cells. The latter performs the inverse operation, merging two adjacent cells by interpolating their data (often, a simple averaging scheme is used).

On such kernels, some applications may require handling special cases for avoiding creating spurious waves due to the adaptive operations. In the case of shallow water equations, it is especially important to maintain the steady state $h + b = 0$ on regions where the water has not yet been perturbed. To accomplish that, it is necessary to take into account the differences in the cell-averaged bathymetry data that may occur when refining or coarsening cells. Consider Code 7.3, where we show an example of a `refine` kernel implementation that properly handles bathymetry variations and dry cells. For further details on implementing well-balanced adaptive mesh refinement for the shallow water equations, refer e.g. to [18].

7.2 Patch-Based Adaptive Mesh Refinement with Vectorization

As mentioned previously, the fine-grained cell-wise adaptivity and the element-oriented design used in sam(oa)^2 pose obstacles to vectorization over the mesh elements. Thus, to implement vectorization, it was necessary to modify the mesh structure to increase regularity of the data structures used. That was accomplished by using the leaves of the refinement tree to store uniformly refined *patches* instead of single cells as before, leading to a more coarse-grained patch-based adaptivity approach. In this section we provide details of our work performed in sam(oa)^2 in order to implement this patch-based approach and to achieve efficient vectorization of the numerical routines.

7.2.1 Uniformly Refined Patches

For the context of sam(oa)^2, we define a *patch* as a set of non-overlapping cells obtained by partitioning a right isosceles triangular cell. For that, we use the same partitioning method used for generating adaptive meshes in sam(oa)^2, i.e., the newest-vertex-bisection method [53]. The method is recursively applied to all cells in the patch up to a uniform refinement *depth d* (defined by the user at compilation time), resulting in a patch containing 2^d cells of identical size.

In Fig. 7.2 we show patches generated with refinement depths ranging from $d = 0$ to $d = 3$. A patch with $d = 0$ contains only one cell and is equivalent to the cell-wise adaptivity imple-

Code 7.3: Implementation of the `refine` kernel, which defines how new fine cells should be initialized. Note that the kernel is called twice for every cell being refined, i.e., once for each resulting cell. In the case of the shallow water equations, special care is necessary to maintain the water elevation despite the changes in bathymetry, avoiding creation of spurious waves. In particular, if a fine wet cell is derived from a coarse dry cell, it is necessary to modify its water height such that the resulting water elevation $h + b$ preserves the "water-at-rest" steady state on unperturbed regions.

```fortran
subroutine refine_kernel(src_element, dest_element, [...])
    ! Input variables
    type(t_traversal_element), intent(inout) :: src_element
    ! Output variables
    type(t_traversal_element), intent(inout) :: dest_element
    ! Local variables
    real :: src_h, src_hu, src_hv, src_bL ! Data in the original coarse cell
    real :: dest_h, dest_hu, dest_hv, dest_bL ! Data in the new fine cell
    logical :: src_was_dry, dest_is_dry ! Whether the coarse/fine cells were/are dry

    ! Extract data from original coarse cell
    src_h  = src_element%cell%data_pers%h
    src_hu = src_element%cell%data_pers%hu
    src_hv = src_element%cell%data_pers%hv
    src_b  = src_element%cell%data_pers%b

    ! Bathymetry data does not come directly from the original cell.
    ! Instead, the cell-averaged bathymetry is recomputed for the new cell.
    dest_b = getBathymetryAtCell(dest_element%transform_data)

    ! Maintain original water elevation h+b despite possible changes in bathymetry
    dest_h = src_h + src_b - dest_b

    ! Keep the water momentum from the original coarse cell
    dest_hu = src_hu ; dest_hv = src_hv

    ! Handle special cases regarding dry cells
    src_was_dry = ( src_h  < dry_tolerance )
    dest_is_dry = ( dest_h < dry_tolerance )
    if (dest_is_dry) then ! If cell is dry, its quantities should be zero
        dest_h = 0.0 ; dest_hu = 0.0 ; dest_hv = 0.0
    else if (src_was_dry) then
        ! If the original cell was dry and the new cell is wet, we need to force
        ! h + b = 0 in the new cell to avoid creating spurious waves.
        dest_h = -dest_b
    end if

    ! Store data for fine cell in the output variable
    dest_element%cell%data_pers%h  = dest_h
    dest_element%cell%data_pers%hu = dest_hu
    dest_element%cell%data_pers%hv = dest_hv
    dest_element%cell%data_pers%b  = dest_b
end subroutine
```

mentation described previously, where patches are not used. Any patch with depth $d > 0$ can be recursively generated by splitting all cells in a patch with depth $d-1$ in half. In general, we identify each cell in a patch with a unique number from 1 to 2^d, where the number of each cell is obtained according to the definition that splitting a cell C_i in a patch with depth d produces two cells identified as C_{2i-1} and C_{2i} in the resulting patch with depth $d + 1$.

In our new implementation, each leaf of the mesh refinement tree is used to store one of such patches, instead of a single cell as before. In order to reduce execution overhead, we only execute the algorithm used for patch generation once before the simulation starts, to compute and store geometry data that will be repeatedly used throughout the simulation. In this step we

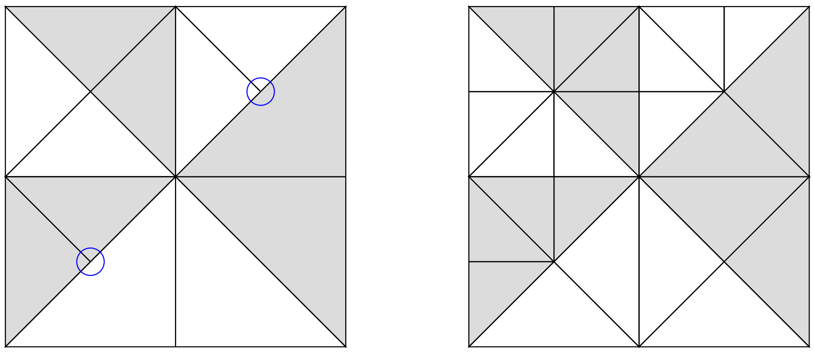
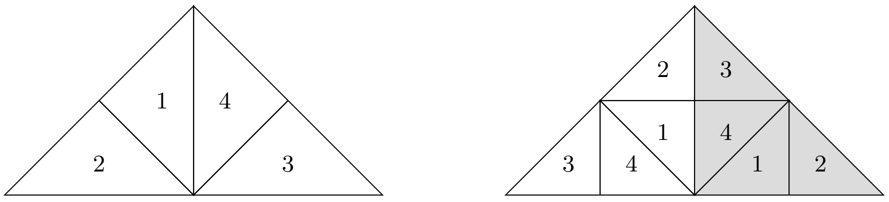

(a) $d = 1$ **(b)** $d = 2$

Figure 7.3: Examples of adaptive meshes composed of patches with refinement depths $d = 1$ and $d = 2$. To make it easier to distinguish between patches and the cells inside of them, we color entire patches with either gray or white background. Note that while (a) may contain hanging nodes (indicated by small blue circles), that is not possible in (b).



(a) Coarse patch **(b)** Two finer patches

Figure 7.4: Refinement of a coarse patch into two finer patches. Each finer patch is indicated by a different background color. Every cell in the coarse patch (a) is split into two cells to obtain two finer patches (b). When performing coarsening of patches, the reverse operation is performed, i.e., starting with the two fine patches shown in (b), a coarsening operation produces the patch shown in (a).

compute important information regarding the patches, such as the coordinates of each vertex, the direction of each edge and the neighbors of each cell. In the case of cells that share one or more edges with the patch external boundary, we also store whether they are adjacent to the left side, right side or hypotenuse of the patch, as this information will be necessary when exchanging data between neighbor patches.

7.2.2 Patch-Based Adaptive Meshes

In the new implementation of the adaptive meshes, patches are treated as the finest element in the refinement tree, corresponding to its leaves. To always guarantee conforming meshes (i.e., without hanging nodes), all patches in the mesh are generated with identical refinement depth d. For the same reason, we only allow even values for d. Consider the examples in Fig. 7.3, where we show meshes composed of patches with depths $d = 1$ and $d = 2$. Note that if the refinement depth d is odd, the patch boundary composed of its hypotenuse contains more nodes than the other two boundaries, which may lead to hanging nodes. On the other hand, if d is even, all three patch boundaries contain exactly the same number of nodes and hanging nodes are therefore not possible.

Treating patches as the finest element in the refinement tree means that it is not possible

Code 7.4: Definition of the data stored for each patch. Here, `_PATCH_NUM_CELLS` is a macro that gives the number of cells in each patch, computed as 2^d according to the patch refinement depth d chosen by the user. This example is based on the single-layer shallow water equations.

```fortran
type num_cell_data_pers
    real, dimension(_PATCH_NUM_CELLS) :: h, hu, hv ! Components of the solution q
    real, dimension(_PATCH_NUM_CELLS) :: b ! Bathymetry
end type
```

to individually refine cells inside a patch. Instead, refinement is applied to the entire patch and produces two patches with the same format and number of cells as the original one, but each with half of its size. In the process, every cell in the original patch is split into two cells to obtain the finer patches. Consider Fig. 7.4, where we show the process of refining a patch with depth $d = 2$. Note that this process is very similar to the one described in the previous section, which is recursively performed to generate the patches. The difference is that in this case the result are two patches with the same refinement depth d as the original patch, instead of a single one with depth $d + 1$. On the other hand, when performing coarsening of patches, the reverse process is applied, i.e., two neighbor patches are merged to produce a coarse patch with the same refinement depth as the two original fine patches.

7.2.3 Data Storage for each Patch

The new patches in sam(oa)^2 are effectively implemented as an additional layer of regular refinement that is managed independently from the adaptive mesh refinement strategy and from the parallelization and traversal schemes described in Section 7.1. As such, the general algorithms described previously have not been modified. Instead, the newly introduced patches are completely managed in the part of the code that can be customized by application developers to implement numerical methods for various applications, i.e., they are managed in the definition of the simulation data and in the implementation of the element-oriented kernels used in the grid traversals.

When defining the simulation data stored in each leaf of the refinement tree, it is now necessary to store data for an entire patch, instead of a single cell as before. That can be trivially performed by using arrays with the appropriate sizes, as shown in Code 7.4 (compare with Code 7.1, where this data type is defined to only store the data of a single cell). Note that the new storage scheme uses an SoA layout, in contrast to the AoS layout that was used before.

7.2.4 Patch-Based Traversals

When implementing the element-oriented kernels for the grid traversals, similar modifications are necessary. Kernels that were previously applied to a cell at a time, should now operate on all cells in the patch, and kernels that were applied to an edge are now applied to one of its three patch boundaries (i.e., to all the edges in that boundary). For example, in the `initialization` traversal, the `element` kernel now needs to initialize all cells in a given patch. Similarly, the `refine` and `coarsen` kernels of the `adaptive` traversal need to split or merge all cells in the patch taking into account their respective positions in the patch (consider again the example shown in Fig 7.4).

For the `initialization` and `adaptive` traversals, such modifications in the implementations of their element-oriented kernels can be considered trivial, as all that is required is to repeatedly apply the same operation to all cells in the patch. On the other hand, much more complex modifications were required for the `numerical` traversal, particularly because it also

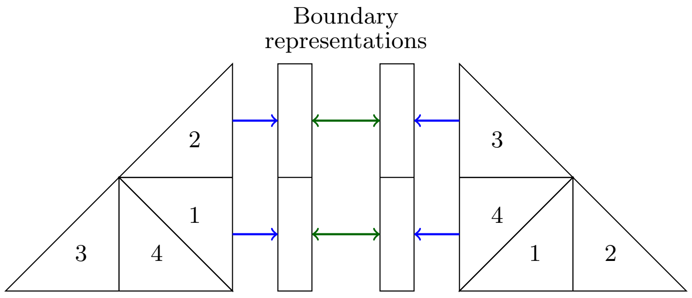


Figure 7.5: Two-step process to exchange boundary data between neighbor patches. First, the `patch-to-boundary` kernel (blue single-pointed arrows) stores data from the cells located at the patch boundary as its *boundary representation*. Later, the `skeleton` kernel (green double-pointed arrows) exchanges the data in the boundary representations of the two neighbor patches. When the process is complete, each patch has access to copies of the boundary data of its neighbor patches, so that applying the finite volume schemes to all of its cells is now possible.

performs operations on the mesh edges, which can now be adjacent to two cells that belong to different patches. In the following subsections, we describe the modifications we made to the kernels of the `numerical` traversal, in order to apply the finite volume scheme to the new patch-based adaptive mesh and to implement vectorization of the Riemann solvers.

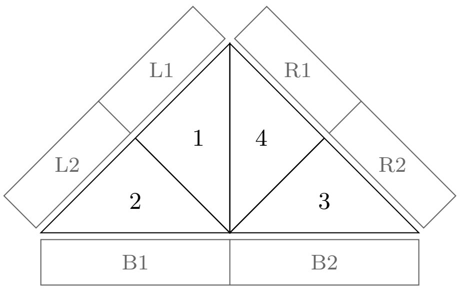
Data Exchanges Between Neighbor Patches

Applying the finite volume scheme to a single patch requires not only data from its cells, but also from cells in the boundary of its adjacent patches. Consider the two neighbor patches shown in Fig. 7.4(b) – updating cells 1 and 2 in the left patch requires data respectively from cells 3 and 4 in the right patch, and vice-versa. Therefore, before we can solve all Riemann problems in a patch, we need to get access to the data of the cells at the boundary of those neighbor patches.

Exchanging of boundary data is performed in two steps, using kernels that are analogous to the `cell-to-edge` and `skeleton` kernels described previously in Section 7.1.4. First, we replace the `cell-to-edge` kernel with a `patch-to-boundary` kernel that is applied to all patches, extracting data from the cells at the three patch boundaries and storing them as *boundary representations*, which are basically arrays containing data from all cells in each boundary. Afterwards, a modified implementation of the `skeleton` kernel acts on all boundaries between adjacent patches, effectively exchanging the boundary data between neighbor patches. The whole process is illustrated in Fig. 7.5. In this process, application developers only need to specify which data should be exchanged between neighbor patches, since sam(oa)^2 internally takes care of actually performing the data exchanges, managing communication between different processes whenever necessary.

Reorganizing the Patch Data into Temporary Arrays

After the data exchanges are complete, every patch has access to all data necessary to update all cells within it, which is then performed by the `patch-update` kernel. This kernel is analogous to the `cell-update` kernel that was applied previously, before we switched to the patch-based approach. However, the `patch-update` kernel is now also responsible for computing solutions to all Riemann problems in the patch, in addition to applying the finite volume update scheme that uses those solutions. Thus, this is the kernel where we now handle vectorization of the loop that applies the Riemann solver to all edges in the patch.



(a) Patch with its boundary representations

Q_i	Q_j
Q_1	Q_2
Q_1	Q_4
Q_1	$Q_{(L1)}$
Q_2	$Q_{(L2)}$
Q_2	$Q_{(B1)}$
Q_3	Q_4
Q_3	$Q_{(R2)}$
Q_3	$Q_{(B2)}$
Q_4	$Q_{(R1)}$

(b) Temporary arrays

Figure 7.6: Reorganization of the patch data into temporary arrays, such that each array position represents one of the edges in the patch and is used to store the input data of its respective Riemann problem. For edges at the patch boundary, one of the adjacent cells belongs to a neighbor patch. In those cases, their data is obtained from the patch boundary representations, which are used as a ghost layer. Note that Q_i and Q_j actually correspond to multiple arrays organized in an SoA data layout, although they are illustrated in (b) as a single array each.

At this point, all required data is available in contiguous arrays for the `patch-update` kernel: the data of all cells in the patch are stored in the format shown in Code 7.4 (following the cell numbering we use to identify those cells, shown in Fig 7.2); additionally, copies of the required data from cells in neighbor patches are available in the form of boundary representations, which are also stored as arrays. However, it is still not possible to directly apply vectorization to those arrays, because the order in which they are organized does not reflect the adjacency relationships between cells, which is required for identifying the Riemann problems that we need to solve.

In order to switch to a vectorization-friendly data layout, during every numerical time step our implementation reorganizes the patch data into temporary arrays such that each array position corresponds to one of the edges in the patch, storing the input data for its respective Riemann problem. More specifically, for every edge in the patch, the cell-averaged quantities Q_i and Q_j from its adjacent cells $\mathcal{C}_i$ and $\mathcal{C}_j$ are copied into these arrays. This process is illustrated in Fig. 7.6. For internal edges, the data for both sides of the edge comes directly from the cells inside the patch. For edges at the patch boundary, one of the sides actually corresponds to a cell in a neighbor patch. In these cases, the data is copied from the boundary representations, which as described previously are used to store the boundary data of the neighbor patches, effectively acting as a ghost layer.

Note that these temporary arrays are also organized as SoA, in order to allow vectorization. For example, in the case of the single-layer shallow water equations, eight arrays are used to store the cell-averaged values of h, hu, hv and b at both sides of the edges. Note also that this reorganization process involves strided accesses to memory and introduces some overhead to the algorithms, similar to the rearranging of the data layout that we implemented for GeoClaw (see Section 6.2.1). But also in this case, our performance experiments show that this overhead is negligible compared to the speedups achieved due to vectorization.

Code 7.5: Main loop that repeatedly calls the Riemann solver for the single-layer shallow water equations, annotated with an `!$OMP SIMD` directive for compiler auto-vectorization.

```
! Note: N is the number of Riemann problems stored in the temporary arrays
! Input variables: temporary arrays containing data of Riemann problems
real, dimension(N) :: hL, huL, hvL, bL ! Data from 'left' cells
real, dimension(N) :: hR, huR, hvR, bR ! Data from 'right' cells
real, dimension(N,2) :: normals ! Vectors that are normal to each edge
! Output variables: numerical fluxes for each Riemann problem and max. wave speed
real, dimension(N) :: flux_hL, flux_huL, flux_hvL
real, dimension(N) :: flux_hR, flux_huR, flux_hvR
real :: maxWaveSpeed
! Local variables
real :: waveSpeed ! Temporary storage for wave speeds

[...]

!$OMP SIMD PRIVATE(waveSpeed) REDUCTION(max: maxWaveSpeed)
do i=1,N ! Loop for all N Riemann problems
    !DIR$ FORCEINLINE
    call computeFluxes(normals(i,:), hL(i),huL(i),hvL(i),bL(i),
        ↪ hR(i),huR(i),hvR(i),bR(i), flux_hL(i),flux_huL(i),flux_hvL(i),
        ↪ flux_hR(i),flux_huR(i),flux_hvR(i), waveSpeed)
    maxWaveSpeed = max(maxWaveSpeed, waveSpeed)
end do
```

Solving the Riemann Problems with Vectorization

With the data organized in those temporary arrays, it is now possible to use vectorization to solve the Riemann problems stored in them. Since these arrays represent a list of Riemann problems to be solved (similarly to the one-dimensional grid slices in GeoClaw), we can apply exactly the same vectorization approach that was developed in Section 6.2.2 for GeoClaw. In fact, the Riemann solver implementations that we use in sam(oa)^2 have been directly extracted from GeoClaw – the only difference is that in sam(oa)^2 they have been extended to directly compute the numerical fluxes used to update the cells, instead of returning the set of waves generated by the discontinuities in the solutions.

In Code 7.5 we show an excerpt of the code with the vectorized loop used to solve the Riemann problems in a patch – note that the loop structure and the compiler directives are very similar to the ones used for GeoClaw (Code 6.1). This excerpt uses a Riemann solver to compute the numerical fluxes crossing each edge, and stores them in temporary output arrays similar to the ones that are used for the input data.

After executing this loop, our `patch-update` kernel implementation finally completes the process of updating all cells in the patch. This is performed by a trivial (also vectorized) loop that repeatedly applies the update scheme given by (7.1) to update the cell-averaged solution stored in each cell.

Cache-Efficient Implementation of the Patch-Update Kernel

In Alg. 7.3 we show a general and straightforward implementation of the `patch-update` kernel that was described above. First, all the Riemann problems in the patch are identified and their input data are organized in temporary input arrays `list`$\_Q_i$ and `list`$\_Q_j$. Then, a vectorized loop (the one shown in Code 7.5) applies a Riemann solver to all problems stored in those arrays, and stores the computed numerical fluxes are in temporary output arrays `list`$\_\mathcal{F}_i$ and `list`$\_\mathcal{F}_j$. Lastly, these lists containing the computed numerical fluxes are used to update all cells in the patch.

Algorithm 7.3: "Naive" implementation of the `patch-update` kernel that directly applies the steps described in this section. This implementation requires large temporary arrays to store the input and output data of all Riemann problems in the patch. Depending on the patch refinement depth, these arrays can grow considerably and decrease cache efficiency.

```
Input: $Q$        // Data for all cells in the patch and in the boundaries of patch neighbors
Output: $Q$                          // Updated solutions for all cells in the patch
Local: list_$Q_i$, list_$Q_j$                  // Temporary arrays for Riemann problems data
Local: list_$\mathcal{F}_i$, list_$\mathcal{F}_j$       // Temporary arrays for Riemann solutions (numerical fluxes)
1  foreach edge $(\mathcal{C}_i, \mathcal{C}_j)$ in patch do          // Copy Riemann problem data to temporary arrays
2      Insert $Q_i$ into list_$Q_i$
3      Insert $Q_j$ into list_$Q_j$
4  end
5  foreach Riemann problem $(Q_i, Q_j)$ in temporary arrays do                // Auto-vectorized loop
6      Solve the Riemann problem with initial data $(Q_i, Q_j)$
7      Store computed numerical fluxes $(\mathcal{F}_i, \mathcal{F}_j)$ in list_$\mathcal{F}_i$ and list_$\mathcal{F}_j$
8  end
9  foreach pair of fluxes $(\mathcal{F}_i, \mathcal{F}_j)$ in temporary arrays do      // Update cells in patch using (7.1)
10     $Q_i \leftarrow Q_i + \frac{\Delta t}{V_i}\mathcal{F}_i$                 // Note that this update operation will be performed
11     $Q_j \leftarrow Q_j + \frac{\Delta t}{V_j}\mathcal{F}_j$                 // three times for each cell (once for each edge)
12 end
```

Note that this implementation requires multiple arrays that need to be large enough to store input and output data for all edges in the patch, which can become considerably large depending on the application and on the patch refinement depth used. For instance, for the single-layer shallow water equations with a patch refinement depth $d = 8$, fourteen arrays (eight for input and six for output data) with 408 positions each are required, which with double-precision variables translates into roughly 45 KB per thread[3]. Although not prohibitively large, we found that this memory requirement can be enough to reduce the efficiency of cache utilization, especially when considering many-core applications and shared levels of cache.

In our actual implementation, we improve cache efficiency by applying a strategy similar to cache-blocking, which iteratively operates on small pieces or "*chunks*" of the lists that store the input data of the Riemann problems in the patch. In Alg. 7.4 we show the algorithm we use for the `patch-update` kernel, which only requires temporary arrays to be large enough to store the data for one chunk of problems at a time. The size of the chunks can be arbitrarily defined by users, so that appropriate values can be chosen for a specific computational platform. Experimentally, we observed that using chunks with 32 Riemann problems each for the single-layer and with 16 problems each for the two-layer shallow water equations delivers reasonable performance for the two experimental platforms we used for this work. Thus, these values were used for the experiments described in the following section.

7.3 Performance Results

Now we evaluate the performance of the vectorized finite volume schemes for the single-layer and two-layer shallow water equations we developed with the new patch-based implementations in sam(oa)^2. As we did for GeoClaw, we also use the experimental platforms that have been described in Chapter 5. In the following section, we describe the simulation scenarios used for

[3]Note that these calculations only consider storage of the temporary arrays used as input and output for the vectorized loop – we did not consider many other data that should ideally also fit in the cache, like the arrays that store the actual patch data and other temporary variables that are used internally by the Riemann solvers.

Algorithm 7.4: Implementation of the `patch-update` kernel that iteratively works on small pieces or "chunks" of the lists that store the input data of the Riemann problems in the patch. This implementation uses smaller temporary arrays that are likely to always fit in cache, regardless of the patch refinement depth. This is the algorithm that is actually used in the finite volume applications we developed for sam(oa)^2.

```
Input: $ChunkSize$                       // Integer: number of edges processed in each iteration
Input: $Q$      // Data for all cells in the patch and in the boundaries of patch neighbors
Output: $Q$                                    // Updated solutions for all cells in the patch
Local: list_$Q_i$, list_$Q_j$                      // Temporary arrays for problems (size: $ChunkSize$)
Local: list_$\mathcal{F}_i$, list_$\mathcal{F}_j$                      // Temporary arrays for solutions (size: $ChunkSize$)

while there are still edges to process do
    list_$Q_i \leftarrow \emptyset$, list_$Q_j \leftarrow \emptyset$, list_$\mathcal{F}_i \leftarrow \emptyset$, list_$\mathcal{F}_j \leftarrow \emptyset$            // Clear temporary arrays
    foreach edge $(\mathcal{C}_i, \mathcal{C}_j)$ in next $ChunkSize$ edges do  // Copy data of next $ChunkSize$ problems
        Insert $Q_i$ into list_$Q_i$
        Insert $Q_j$ into list_$Q_j$
    end
    foreach Riemann problem $(Q_i, Q_j)$ in temporary arrays do              // Auto-vectorized loop
        Solve the Riemann problem with initial data $(Q_i, Q_j)$
        Store computed numerical fluxes $(\mathcal{F}_i, \mathcal{F}_j)$ in list_$\mathcal{F}_i$ and list_$\mathcal{F}_j$
    end
    foreach pair of fluxes $(\mathcal{F}_i, \mathcal{F}_j)$ in temporary arrays do  // Update cells in patch using (7.1)
        $Q_i \leftarrow Q_i + \frac{\Delta t}{V_i} \mathcal{F}_i$
        $Q_j \leftarrow Q_j + \frac{\Delta t}{V_j} \mathcal{F}_j$
    end
end
```

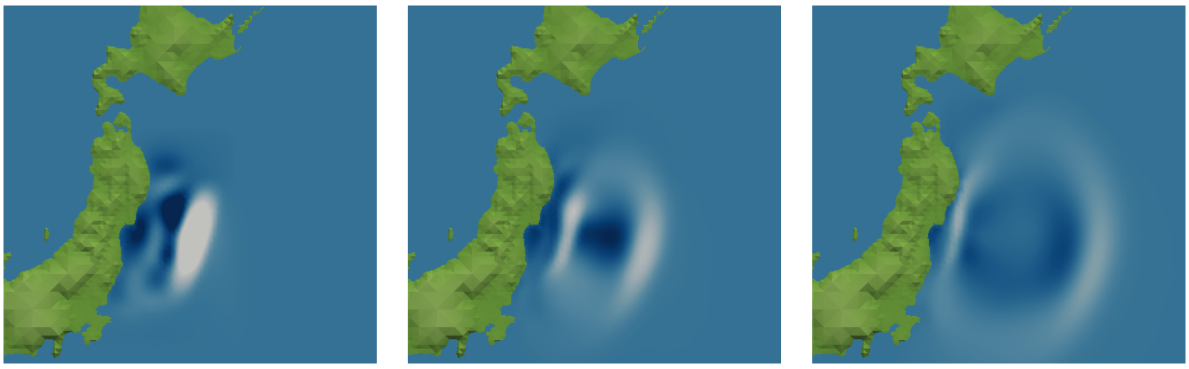


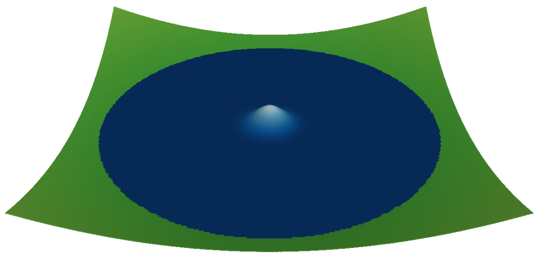
Figure 7.7: Simulation of the tsunami in Tohoku, Japan, 2011. The pictures show the tsunami wave 10, 20 and 30 minutes after the earthquake, respectively.

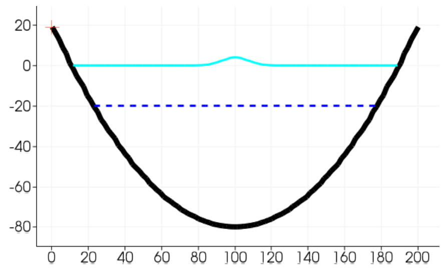
each variation of those equations. In our experimental analysis, we first evaluate the performance of vectorization by focusing only on the performance of each solver. Later, we perform a more general analysis, assessing how the new patch-based approach affects the simulation performance as a whole, and attempting to find a setup that minimizes its time-to-solution.

7.3.1 Simulation scenarios

Tohoku 2011 Tsunami

For the single-layer shallow water equations in sam(oa)^2, we simulate the tsunami event that happened near the coast of Tohoku, Japan, in March 2011. Some snapshots of the simulations


(a) 3D visualization



(b) Vertical-section visualization

Figure 7.8: Parabolic bowl-shaped lake at $t = 0$ with 3D and vertical-section visualizations. In (b), the thick black line depicts the bathymetry, while the dashed blue and solid cyan lines represent the two layers of water.

are shown in Fig. 7.7. For these simulations we use bathymetry data from the Northern Pacific and the Sea of Japan with resolutions of approximately 0.5 km in each direction, made available by the General Bathymetric Chart of the Oceans project (GEBCO_14 Grid, version 20150318) [27, 72]. Similarly as done for the simulations of the Chile 2010 Tsunami (discussed in Section 6.3), we use a static displacement for the water and bathymetry, this time based on a simulation of the Tohoku earthquake [26]. All geographical input data is handled in sam(oa)^2 using the parallel I/O library ASAGI [60].

Although the entire simulation domain covers an area of 7 000 km $\times$ 4 000 km, our adaptive mesh was able to discretize it with a maximum resolution of roughly 107 m in the area around the tsunami, while regions farther away can have cell resolutions with up to 155 km[4]. Considering all simulations, the minimum and maximum observed mesh sizes were of approx. 6.8 million to 34.2 million cells. The simulations were run for 1000 time steps and the measurements only include the regular time-stepping phase, i.e., they do not consider the time necessary for reading the input data and generating the initial mesh.

Parabolic Bowl-Shaped Lake

For the two-layer shallow water equations, we use the same artificial scenario that we used previously for GeoClaw, i.e., we simulate waves generated by a circular hump of water propagating over a parabolic bowl-shaped bathymetry (see Section 6.3.1). In Fig. 7.8, we show visualizations of this scenario at $t = 0$, now in the context of sam(oa)^2. In these simulations we used cells with sizes ranging from 2^{-27} to 2^{-11} of the computational domain, resulting in meshes with cell counts varying from approx. 0.8 million to approx. 7.2 million cells. Like in the simulations of the Tohoku tsunami, the performance measurements for this scenario also consider 1000 time steps of the regular time-stepping phase.

7.3.2 Vectorization Performance

Like we did for GeoClaw, we start our analysis by evaluating the performance of the Riemann solvers, with emphasis on the effectiveness of vectorization, and by assessing their efficiency on exploiting the computational resources. Thus, in these experiments only the subroutines that perform computations for the numerical scheme are considered (of which the Riemann solver is responsible for the majority of the execution time). For the single-layer shallow water equations, we measure the performance of the f-wave solver (Fig. 7.9) and of the augmented

[4]These resolution values are based on the length of the longest edge of each triangular cell, i.e., its hypotenuse.

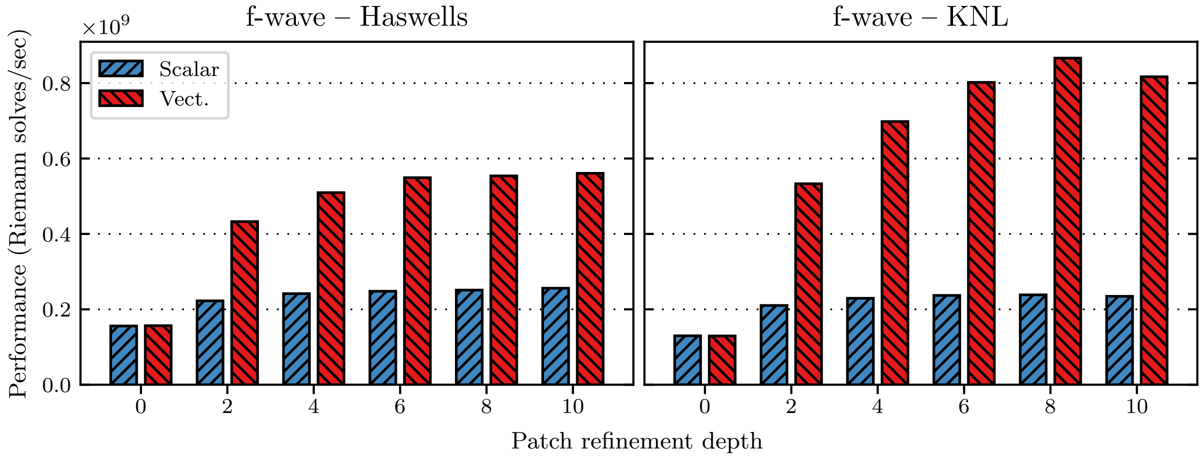


Figure 7.9: Performance of the f-wave solver for varying patch refinement depths, before and after applying vectorization.

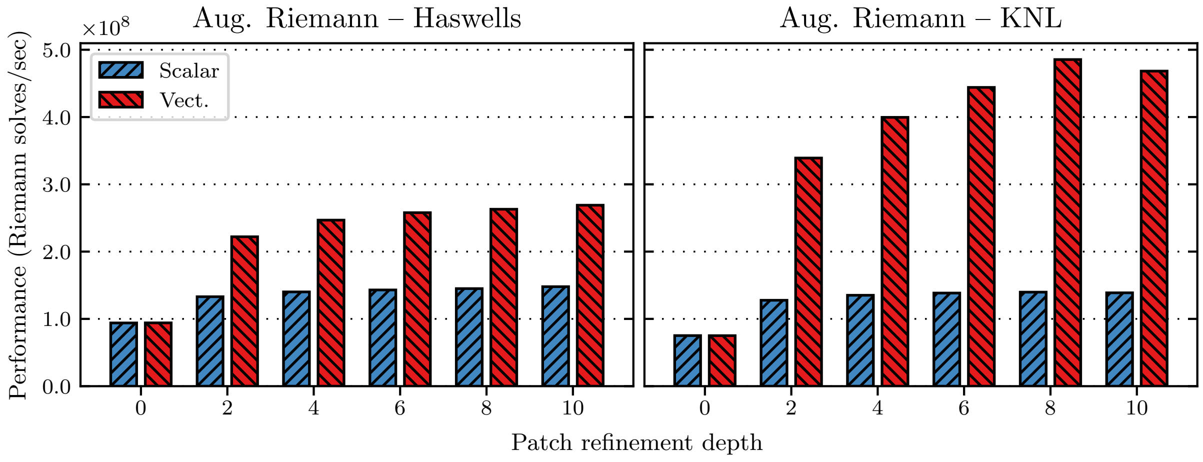


Figure 7.10: Performance of the augmented Riemann solver for varying patch refinement depths, before and after applying vectorization.

Riemann solver (Fig. 7.10). We also show the results obtained with the Riemann solver for the two-layer equations in Fig. 7.11. In these plots we use the same metric as before to measure the performance of each solver, i.e. the performances are expressed as *Riemann problems solved per second*. Note that, as discussed previously, our finite volume implementations in sam(oa)^2 use a modified version of the donor-cell upwind method, which does not apply transverse corrections in the update scheme. Thus, here we only consider *normal* Riemann solvers.

In these plots we show the performance obtained for each solver with ("Vect.") and without vectorization ("Scalar"), on adaptive meshes generated with uniform patches of varying refinement depths (d). For experiments without vectorization, vectorization was turned off by simply omitting the `!$OMP SIMD` compiler directives from the vectorized loops. We configured these simulations to always apply the same finest resolution at the wave fronts regardless of the patch refinement depth used, such that all simulations achieve the same accuracy. Thus, we note that although each increment by two in the patch refinement depth d means a fourfold increase in the number of cells in each patch (because each patch is composed of 2^d cells), the

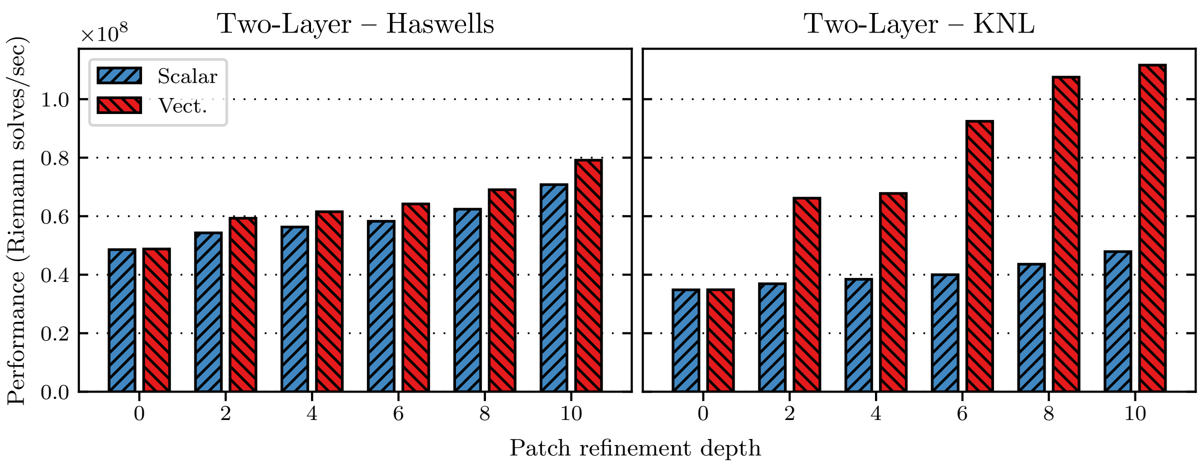


Figure 7.11: Performance of the two-layer solver for varying patch refinement depths, before and after applying vectorization.

total number of cells in the mesh is not directly proportional to the number of cells in each patch. There is, however, a tendency of greater values for d leading to more cells overall in the mesh, due to the coarser-grained adaptivity scheme. This effect is discussed further in Section 7.3.3, where we compare the mesh sizes and the resulting execution times obtained on simulations with different patch refinement depths.

In addition, we also point out that in all plots the results indicated by $d = 0$ actually refer to executions of the original applications that do not use the new patch-based approach (in contrast to executing the new patch-based implementations using $d = 0$ for the patches, which we do not consider here). In other words, these plots allow easy performance comparisons between the original implementations that perform cell-wise adaptivity ($d = 0$) and the new patch-based simulations ($d > 0$).

Even with vectorization turned off, it is possible to notice a significant increase in performance for the solvers on patch-based meshes, compared to the original implementations (up to 1.6× for Haswells and 1.9× for KNLs). That can be attributed to two factors: first, to the regular and more concise data structures that we now use, which result in improvements in the memory throughput; second, to the patch-based discretization itself, which considerably reduces the size and complexity of the implicit refinement tree, reducing the effort required to manage the adaptive mesh and leading to faster grid traversals.

When vectorization is used, we observe much more substantial speedups, with a clear pattern of larger patches (i.e., patches with greater refinement depth) delivering higher performance in almost all experiments. Compared to the experiments with vectorization turned off, the vectorized Riemann solvers for the single-layer shallow water equations achieve speedups of up to 2.2× on the Haswells and up to 3.6× on the KNLs. For the two-layer solver, vectorization gives only small speedups on the Haswells (up to 1.1×), and decent speedups on the KNLs (up to 2.5×). These results are similar to what we experienced with GeoClaw, confirming that 256-bit instructions are not enough to bring substantial performance improvements for such a complex solver.

Table 7.1 presents a summary of the speedups and performance in Flop/s achieved by the vectorized solvers in sam(oa)^2. Here we only consider the experiments with the patch refinement depths that achieve the highest performance for each solver and for each machine (either $d = 8$ or $d = 10$ in all experiments). Like we did on the experiments with GeoClaw, the Flops/s

Table 7.1: Performance of the vectorized solvers in sam(oa)^2. For each row in the table, we consider only the patch refinement depth that delivered highest performance. The speedups listed represent performance comparisons for executions with and without vectorization using that same refinement depth. The percentages in parentheses show how the achieved performances compare to the maximum attainable performance of each machine (measured with the Flop/s benchmark described in Section 5.1).

Equations	**Machine**	**Solver**	**Speedup**	**GFlop/s**
Single-layer	Haswells	f-wave	2.2×	63.3 (41%)
		Aug. Riemann	1.8×	68.7 (44%)
	KNL	f-wave	3.6×	98.9 (14%)
		Aug. Riemann	3.5×	126.8 (18%)
Two-layer	Haswells	Two-layer solver	1.1×	38.0 (24%)
	KNL	Two-layer solver	2.3×	53.6 (7%)

throughputs were estimated with the PAPI interface [54] and we used the conservative approach of considering the total number of floating-point operations performed by the non-vectorized solver as baseline for the calculations.

Comparing these performance results with the ones obtained for the same solvers with GeoClaw (see Table 6.1), we observe similar tendencies for the speedups achieved, but significantly higher throughputs for the solvers when used in sam(oa)^2, especially for the KNL architecture. The higher performance obtained by sam(oa)^2 is a consequence of it achieving more balanced load distributions for shared-memory parallelization, compared to GeoClaw. While both frameworks perform parallelization over the patches in the mesh, this delivers higher performance for sam(oa)^2 because in general its meshes are formed by much more (relatively small) patches than the meshes in GeoClaw, which are usually composed of fewer larger patches. The greater number of patches in sam(oa)^2 makes it easier to achieve balanced load distributions on tens or hundreds of threads, as is required for efficient utilization of the two many-core platforms considered in this work. On the other hand, because the meshes in GeoClaw are usually formed by much fewer patches, GeoClaw is often unable to keep all threads busy. In addition, patches in GeoClaw do not necessarily have the same size, which contributes to further load imbalances in its parallel algorithms.

7.3.3 Simulation Performance and Time-To-Solution

Now we examine how the new patch-based discretizations and the vectorization of the numerical routines affect the performance of the entire simulation, also considering the execution time necessary for handling adaptivity. In general, our goal is to minimize the time-to-solution, which does not depend solely on the performance of the Riemann solver, as we show in the following.

Compared to meshes with cell-wise adaptivity, the patch-based adaptive meshes typically require more cells and more computational work to achieve the same solution accuracy. In general, this effect is considerably more noticeable for larger patches than for smaller ones. This tendency can be easily observed in Fig. 7.12, where we plot the total number of cell updates performed in the experiments described previously.

Since the Riemann solvers are the most time-consuming step of our finite volume applications, it also makes sense to analyze which effect the value used as patch refinement depth has on the number of times they are executed. Thus, in Fig. 7.13 we plot the total number of Riemann

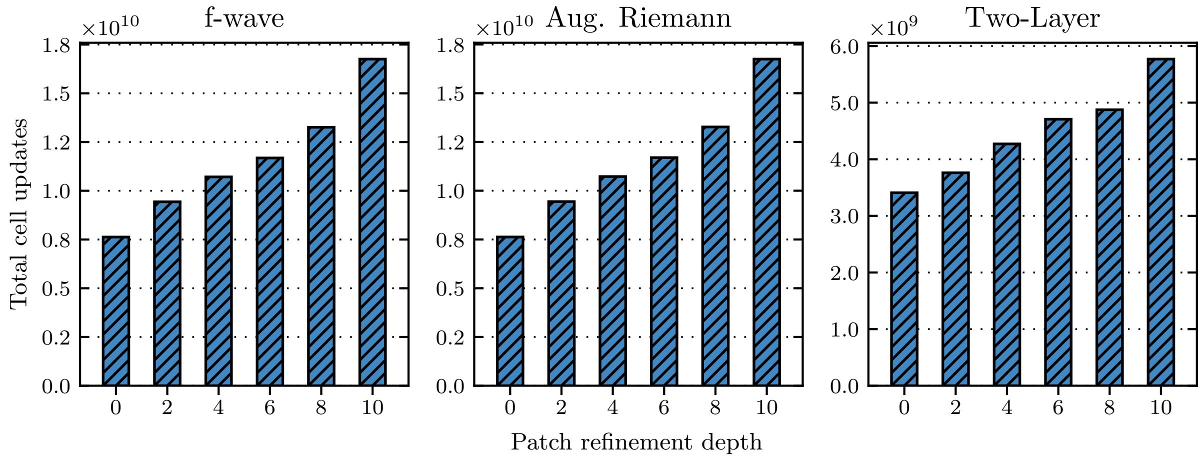


Figure 7.12: Total number of cell updates performed during the entire simulations, for different patch refinement depths.

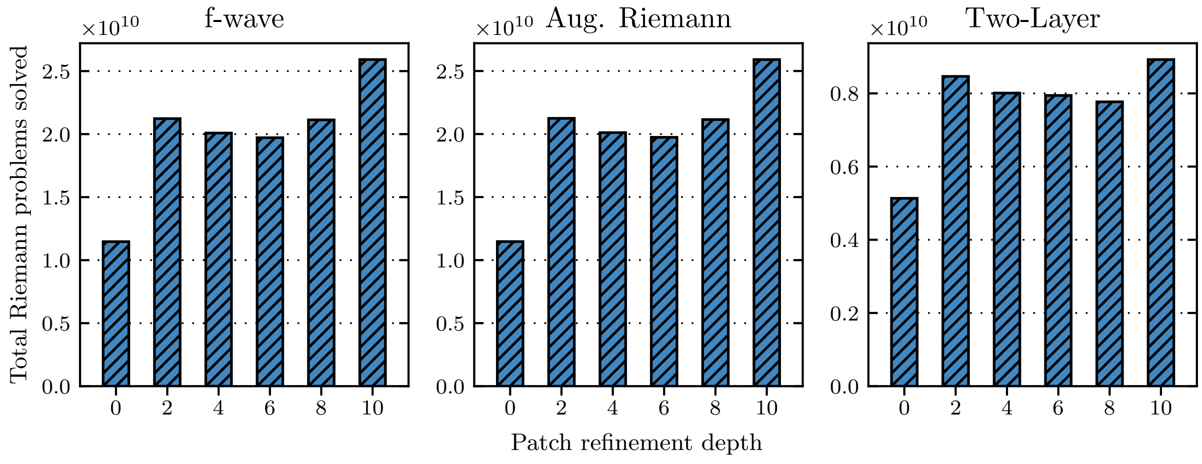


Figure 7.13: Total number of Riemann problems solved during the entire simulations, for different patch refinement depths.

problems solved during our experiments. In this case, the plots do not have a strictly increasing behavior as before, but it is still possible to notice that very large patches require solving much more Riemann problems for the same accuracy – e.g., patches with refinement depth $d = 10$ required approximately 1.7–2.3 times more Riemann solves than cell-wise adaptivity.

Note that the irregular behavior observed in these plots is due to the high proportion of external edges in small patches. As described in Section 7.2.4, these edges lie between cells located in different patches, and they are processed with the help of patch boundary representations that act as a ghost layer. Therefore, the Riemann problems on external edges are actually solved twice, once for each patch, which considerably increases the number of times the solvers are used for very small patches. E.g., consider patches generated with refinement depth $d = 2$: six of their nine edges are external edges, meaning that two thirds of all Riemann problems in the mesh are solved twice. For $d = 4$, this ratio drops to 40% and for $d = 6$ to 16.6% of the edges in the mesh. With larger patches the percentage of problems solved twice continues to drop, but at the same time the total number of edges in the mesh increases, which

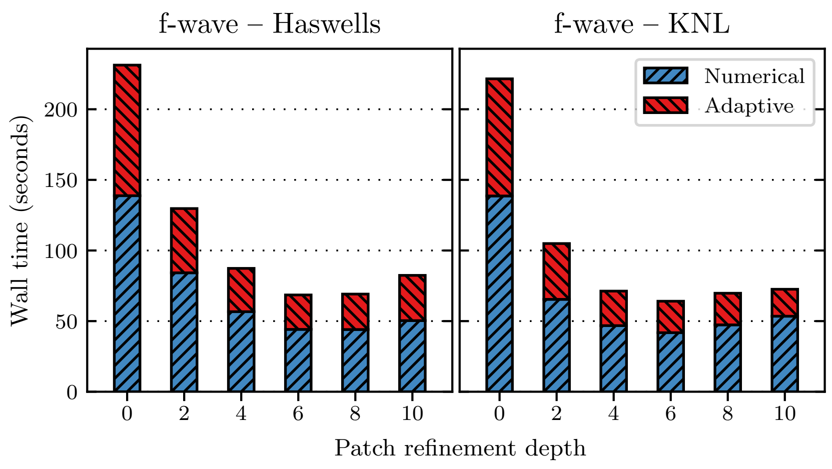


Figure 7.14: Wall time split into components for the entire simulations with the the vectorized f-wave solver.

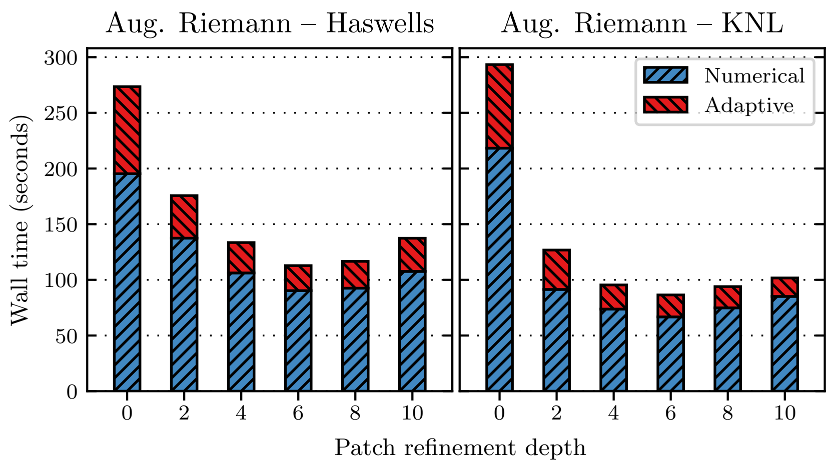


Figure 7.15: Wall time split into components for the entire simulations with the vectorized Augmented Riemann solver.

explains the curves observed in these plots.

Considering the effects that the choice of refinement depth has on the mesh size, it is clear that defining an ideal value for d with the goal of minimizing the simulation time-to-solution is not a trivial task and involves finding a trade-off between the solver throughput and the total computational work performed – two factors that usually increase with larger patches. Experimentally, we have found that using $d = 6$ or $d = 8$ as patch refinement depth minimized the time-to-solution in all our simulations – see the plots in Figs. 7.14 to 7.16, where we plot the execution wall time taken by our simulations. In those plots, we can observe improvements in the time-to-solution by factors of 1.3–3.4 for Haswells and of 2.2–6.0 for KNLs, when comparing the fastest patch-based simulations with the original implementations (which use cell-wise adaptivity and do not support vectorization, indicated in the plots as $d = 0$). Thus, using patches with refinement depths $d = 6$ or $d = 8$ (i.e., patches with 64–256 cells each) seems to be a good rule of thumb for reducing the time-to-solution of shallow water simulations in sam(oa)^2, although this may become suboptimal for other simulations in different contexts

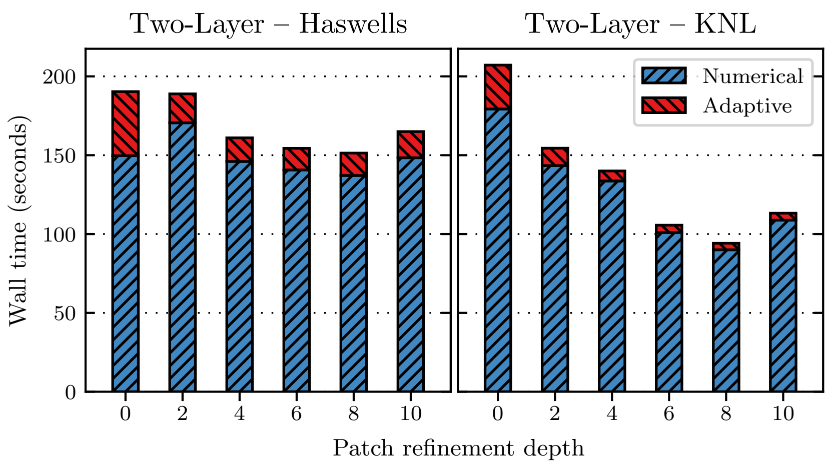


Figure 7.16: Wall time split into components for the entire simulations with the vectorized two-layer solver.

and/or different computational platforms.

Note that in Figs. 7.14 to 7.16, we split the execution time of each simulation into the time taken by the `numerical` traversals and the `adaptive` traversals. Similarly to what we experienced for the numerical routines, it is clear that the `adaptive` traversals also benefit from the patch-based discretization, even though we have not implemented vectorization in their subroutines – speedups of up to 3.8× on Haswells and 6.7× on KNLs have been observed for this component alone. Again, this is because patch-based adaptive meshes require smaller and simpler refinement trees to achieve the same resolutions as meshes with cell-wise adaptivity, which reduces the execution time for various simulations components. In addition to reducing the complexity and execution time of the Sierpinski-based grid traversals, patch-based meshes with smaller refinement trees also contribute to significantly more efficient refinement and coarsening operations, since they now work on multiple cells at once. In particular, we also observed that the component of the `adaptive` traversals that is responsible for guaranteeing mesh conformity benefits considerably from the coarser-grained adaptivity, because it needs to perform much less additional refinements in order to avoid hanging nodes in the mesh.

7.4 Conclusions

In this chapter, we described our work on modifying the adaptive meshes in sam(oa)^2 in order to support vectorization for finite volume schemes. More specifically, we replaced the fine-grained cell-wise adaptivity strategy with a coarser-grained patch-based strategy, which we used to create new implementations of finite volume schemes for the single-layer and two-layer shallow water equations. In these, we have been able to use the same implementations of the vectorized Riemann solvers that we developed for GeoClaw in Section 6.2.

In our performance experiments, we observed substantial improvements in the performance of the Riemann solvers, similar to what was observed in GeoClaw. However, we also noticed that in sam(oa)^2 the Riemann solvers achieve even higher performance, as a consequence of its more efficient shared-memory parallelization and load balancing schemes. These results also show a tendency of patches generated with greater values for the patch refinement depth d resulting in higher performances of the vectorized Riemann solvers.

However, when considering the performance of the entire simulations, we found that the relationship between patch refinement depth and time-to-solution is considerably more complex. Although larger patches (i.e., with more cells) foster vectorization and higher performance in general, they also increase the total of number of cells and edges in the adaptive meshes, considerably increasing the computational work performed during the simulations. Thus, minimizing the time-to-solution requires choosing patch sizes that find a compromise between those factors. Experimentally, we found that relatively small patches with 64–256 cells are usually a good choice, achieving reductions in the execution time of up to 83% for our simulations.

In addition to allowing vectorization, the new patch-based discretization also resulted in further speedups in various simulations components due to improved memory throughput and to reductions in the size and complexity of the implicit refinement trees. Thus, we point out that the strategy of switching from cell-wise adaptivity to patch-based meshes should be considered even for applications or computational platforms for which vectorization is not possible or efficient, since even in those cases such modifications can lead to substantial performance improvements.

8 FVM: a Generic Interface for Vectorized Finite Volume Solvers in Sam(oa)^2

As shown in the previous chapters, developing high performance applications that efficiently combine parallelization, adaptive mesh refinement and vectorization is not a trivial task, as it requires careful choice of data structures, of meshing and parallelization algorithms and of many other implementation details that may inhibit high performances. In addition, developers of scientific applications are often not HPC experts and may put less effort in performance engineering, especially in early stages of the development process, when development and improvement of the numerical models may still be the focus. As a consequence, scientific applications are often not able to fully exploit the potential of the high-performance platforms in which they are executed.

Although the kernel-based design implemented in sam(oa)^2 provides a decent abstraction layer for the complicated algorithms employed to manage adaptive mesh refinement, cache-efficient grid traversals and many other features, working with it still requires knowledge of many implementation details, especially of the data structures used in its object-oriented implementation (consider e.g. the kernel implementations shown in Codes 7.2 and 7.3). In addition, the new patch-based design introduces further complexity to the kernel implementations, since familiarity with the patch structure is now necessary.

With the goal of simplifying as much as possible the process of developing high-performance finite volume schemes with sam(oa)^2, we designed a generalization of the vectorized patch-based implementations presented in the previous chapter that can be easily customized for various systems of hyperbolic PDEs, allowing developers with no HPC expertise to benefit from the techniques developed in this thesis. That is accomplished via a programming interface that, similarly to the kernel-based design in sam(oa)^2, requires application developers to provide a set of *operators* that are applied to the mesh elements individually. However, in contrast to the previous approach, we designed the new interface such that developers do not need any knowledge of the data structures used internally by the framework. Instead, the operators that must be provided work directly with the simulation data and use only simple data structures (arrays), allowing developers to focus on implementing application-specific algorithms and decisions.

This chapter presents an extended version of our work published in [24], where we described and evaluated the new generic interface for developing high-performance finite volume methods in sam(oa)^2, which we named *FVM interface*. To demonstrate its usability we used it to create alternative implementations of the vectorized patch-based finite volume schemes for the single-layer and two-layer shallow water equations that have been developed with sam(oa)^2 and discussed in Chapter 7. In the following section we introduce the FVM interface by describing its abstraction layer and by showing a few example implementations of the so-called *FVM operators*, which are required to create new applications with it. Later, in Section 8.2, we provide an experimental performance analysis of the new applications developed with the

Code 8.1: Definition of the data stored by the FVM interface for each patch. Here, `_PATCH_NUM_CELLS` is a macro that gives the number of cells in each patch, and `_FVM_Q_SIZE` and `_FVM_AUX_SIZE` are macros that should be defined by application developers depending on their requirements for data storage.

```fortran
type num_cell_data_pers
    real :: Q(_PATCH_NUM_CELLS,_FVM_Q_SIZE)     ! Solution variables
    real :: Aux(_PATCH_NUM_CELLS,_FVM_AUX_SIZE) ! Auxiliary variables
end type
```

FVM interface, focusing especially on comparisons with their analogous applications, to assess how the additional abstraction layer provided by the interface affects the performance of the developed finite volume schemes.

8.1 FVM Interface

In this section we provide details regarding the implementation and the use of the new generic interface. In its design, we combined the concept of element-oriented kernels present in sam(oa)^2 with the GeoClaw's approach for organizing the simulation data (i.e., storing the cell quantities in arrays whose dimensions depend on the system of PDEs being considered). Those two strategies were chosen because they allow easy customization of application-specific parameters and algorithms, as will be presented in the following.

8.1.1 GeoClaw-Like Data Structures

Following the approach used in GeoClaw (see Section 6.1.3), we store the simulation data for each patch in an array with an extra dimension that is used for fitting the multiple components in the solution q. More specifically, we define an array `Q(_PATCH_NUM_CELLS,_FVM_Q_SIZE)`, where `_PATCH_NUM_CELLS` is a macro constant that stores the number of cells in each patch (computed from the patch refinement depth d as 2^d) and `_FVM_Q_SIZE` is another macro constant that refers to the number of components in q. Like in GeoClaw, we also define an additional array `Aux(_PATCH_NUM_CELLS,_FVM_AUX_SIZE)` that can be used to store a given number (defined by the macro `_FVM_AUX_SIZE`) of other cell-specific “auxiliary” variables, like the bathymetry data in the case of the shallow water equations.

In Code 8.1 we show the definition of the data stored for each patch. The advantage of this approach for data storage is that, differently from the previous one (shown in Code 7.4), this code can be reused for various system of PDEs with no modifications, as all that is required from developers to adapt it for their needs is the definition of the compiler preprocessor macros `_FVM_Q_SIZE` and `_FVM_AUX_SIZE` with appropriate values. Note also that, differently from GeoClaw's original implementation, those arrays are already organized following the SoA layout (i.e., the cell averages of any given quantity are stored contiguously for each patch), which is very important for efficient vectorization.

In the implementations of the finite volume schemes developed in this chapter with the FVM interface (to which we often refer as *FVM applications*), we define these arrays following the same order for the cell quantities as done for GeoClaw, i.e., $q = [h, hu, hv]^T$ for the single-layer equations, and $q = [h_1, h_1u_1, h_1v_1, h_2, h_2u_2, h_2v_2]^T$ for the two-layer equations. Additionally, we also use $Aux = [b]$ to store bathymetry data for both variations of the shallow water equations.

Code 8.2: Implementation of the `get-computational-domain` operator, which is called once at the execution start to get the parameters that define the simulation domain. In this example, the computational domain is being defined as $[-100, 100] \times [-100, 100]$.

```fortran
subroutine FVM_get_computational_domain(scaling, offset)
    real,                intent(out)    :: scaling
    real, dimension(2), intent(out)    :: offset

    scaling = 200.0            ! Total size of both dimensions
    offset  = [-100.0, -100.0] ! Coordinates of the domain origin
end subroutine
```

8.1.2 FVM Operators

We designed the FVM interface to transparently manage the data in the patches, such that developers do not need any familiarity with their geometry or with their data layout (shown in Code 8.1). Instead, developers only need to implement simple *FVM operators* that are applied to the mesh on a cell-by-cell or edge-by-edge basis, completely hiding the patch and mesh structures from developers and substantially simplifying the implementation process. In the FVM operators, developers define how cells should be initialized, refined, coarsened and updated, which includes providing an appropriate Riemann solver for the particular problem. In the following we describe the required FVM operators, showing examples based on the FVM application we developed for the single-layer shallow water equations.

Get-Computational-Domain Operator

This simple FVM operator is used to define the simulation's computational domain. The parameters set in this subroutine define the system of coordinates that is used to compute the coordinates of the cells' vertices that are given as input to various FVM operators. In the operator implementation, developers need to provide values for two parameters:

- `scaling`: the size of both dimensions of the computational domain;
- `offset`: coordinates of the domain origin.

After getting those values, sam(oa)^2 then defines the computational domain accordingly. For instance, to specify a computational domain as $[-100, 100] \times [-100, 100]$, developers should define `scaling=200` and `offset=[-100,-100]`, as shown in the example in Code 8.2.

Initialize-Cell Operator

With this operator, developers define the initial conditions and the initial mesh refinement for the simulation in a cell-by-cell basis. We show an example implementation for this FVM operator in Code 8.3. Given the coordinates of a cell's vertices, the operator must return the initial value of all cell quantities (`Q` and `Aux`), as well as an estimate for the initial wave speed (used for computing the size of the first time step), and a refinement flag: 1 if this cell should be further refined in the following initialization step, or 0 otherwise.

Note that although we use a patch-based discretization, the operator works with individual cells, which significantly simplifies its implementation. Internally, the FVM interface creates an abstraction layer for the patches, by calling the `initialize-cell` operator multiple times, once for each cell in the patch. This is performed inside the generic implementation of the `element` kernel for the `initialization` traversal, which handles an entire patch at a time, as shown

Code 8.3: Implementation of the `initialize-cell` operator, used to define the initial simulation conditions. This example produces a simple "radial dam-break" scenario for the single-layer shallow water equations – the initial water elevation is constant at zero ($h + b = 0$) for the entire domain except for a five-meter-radius circle at the center of the domain, where the water elevation is set to 5 meters ($h + b = 5$).

```fortran
subroutine FVM_initialize_cell(vertices,Q,Aux,wave_speed,refinement)
    ! Input variables
    real,     intent(in)  :: vertices(2,3)
    ! Output variables
    real,     intent(out) :: Q(_FVM_Q_SIZE) ! Vector of Q quantities in cell
    real,     intent(out) :: Aux(_FVM_AUX_SIZE) ! Vector of Aux quantities in cell
    real,     intent(out) :: wave_speed ! Estimate for initial wave speed
    integer,  intent(out) :: refinement ! Refinement flag (1 or 0)
    ! Local variables
    real                  :: h, hu, hv, b, center(2)

    ! Compute cell barycenter
    center = (vertices(:,1) + vertices(:,2) + vertices(:,3)) / 3.0

    ! Define initial values for cell-averaged quantities
    h = 100.0 ;  hu = 0.0 ;  hv = 0.0 ;  b  = -100.0
    if (center(1)**2 + center(2)**2 < 25.0) h = 105.0 ! Water elevation at center

    Q = [h, hu, hv] ;  Aux = [b] ! Fill output arrays
    wave_speed = sqrt(9.81*h)    ! Estimate for initial wave speed

    refinement = 0              ! Set refinement flag:
    if (h > 0.0) refinement = 1 ! Only refine over the water elevation
end subroutine
```

in Code 8.4. In general, the FVM interface creates abstraction layers like this for all element-oriented kernels in sam(oa)^2, such that the FVM operators implemented by developers only need to handle one cell or one edge at a time, while the abstraction layer internally organizes their input and output data appropriately, according to the patch and mesh structures.

Compute-Fluxes Operator

This operator is responsible for solving the PDE-specific Riemann problems at the mesh edges and for returning the computed numerical fluxes, as well as the speed of the fastest wave created by the discontinuities between the cells. This is usually performed by applying an appropriate Riemann solver for the PDE being considered and is in general the most compute-intensive and time-consuming step in the simulations. Therefore, at this step we give developers the option of having finer control of the code used to update an entire patch. For that, we give them two options for providing this operator. In the simplest case, they can implement a `single-edge` version, which only handles one edge at a time while the FVM interface handles auto-vectorization of the main solver loop, using the `!$OMP SIMD` directive on it. On the other hand, more advanced developers may choose to provide a `multi-edge` version, in which they have complete control over the loop that iterates through the list of Riemann problems, such that they may use other techniques to achieve vectorization or attempt other kinds of optimization on it. In the following, we give more details about each of those versions.

Single-edge operator: this is the simplest version of the operator, which is applied to one edge at a time, solving a single Riemann problem. In this case, the operator code provided by the developer does not need to implement vectorization, because it is completely handled by the framework. We show an example implementation for this operator in Code 8.5, and the part of

Code 8.4: Generic implementation of the `element` kernel for the `initialization` traversal, which is responsible for initializing an entire patch. Here, the FVM interface creates an abstraction layer for the data in the patches, such that developers only need to implement code for individual cells, in the `initialize-cell` operator (like in the example shown in Code 8.3).

```fortran
! Note: this is framework code that should not be modified by the developer.
! Application-specific code should be provided in FVM_initialize_cell().
subroutine element_kernel(element, [...])
    type(t_element), intent(inout) :: element ! Contains all data for the patch
    ! Local variables
    real :: Q(_FVM_Q_SIZE), Aux(_FVM_AUX_SIZE)
    real :: vertices(2,3), wave_speed(_PATCH_NUM_CELLS)
    integer :: refinement_flag(_PATCH_NUM_CELLS), i

    do i=1, _PATCH_NUM_CELLS ! Loop through all cells in the patch initializing them
        vertices = element%patch%geometry%get_vertices(i) ! Get vertices' coordinates

        ! Call initialize-cell operator provided by developer
        call FVM_initialize_cell(vertices, Q, Aux, wave_speed(i), refinement_flag(i))

        ! Copy values to cell data inside patch
        element%patch%data_pers%Q(i,:) = Q ;
        element%patch%data_pers%Aux(i,:) = Aux
    end do

    ! Compute max_wave_speed in patch
    element%patch%max_wave_speed = maxval(wave_speed(:))

    ! Refine this patch if any cell was flagged as "refine"
    if (any(refinement_flag(:) == 1)) then
        element%patch%geometry%refinement = 1
    else
        element%patch%geometry%refinement = 0
    end if
end subroutine
```

Code 8.5: Implementation of the `single-edge` version of the `compute-fluxes` operator, which is responsible for computing the solution of a single Riemann problem.

```fortran
subroutine FVM_compute_fluxes_single(normal,qL,qR,auxL,auxR,fluxL,fluxR,waveSpeed)
  real, dimension(2),            intent(in)  :: normal !Normal vector
  real, dimension(_FVM_Q_SIZE),  intent(in)  :: qL,qR
  real, dimension(_FVM_AUX_SIZE), intent(in) :: auxL,auxR
  real, dimension(_FVM_Q_SIZE),  intent(out) :: fluxL,fluxR
  real,                          intent(out) :: waveSpeed
  ! Local variables
  real :: hL,huL,hvL,bL, hR,huR,hvR,bR ! Input for Riemann solver
  real :: flux_hL,flux_huL,flux_hvL ! Output left-going fluxes
  real :: flux_hR,flux_huR,flux_hvR ! Output right-going fluxes

  !Extract data from input arrays to local variables
  hL = qL(1); huL = qL(2); hvL = qL(3); bL = auxL(1)
  hR = qR(1); huR = qR(2); hvR = qR(3); bR = auxR(1)

  ! Call Riemann solver
  !DIR$ FORCEINLINE
  call RiemannSolver(normal, hL,huL,hvL,bL, hR,huR,hvR,bR, flux_hL,flux_huL,flux_hvL,
      ↪ flux_hR,flux_huR,flux_hvR, waveSpeed)

  ! Copy computed fluxes to output arrays
  fluxL = [flux_hL, flux_huL, flux_hvL]
  fluxR = [flux_hR, flux_huR, flux_hvR]
end subroutine
```

Code 8.6: Piece of the `patch-update` kernel that repeatedly calls the `single-edge` version of the `compute-fluxes` operator. Similarly to what is done in Code 8.4, here the FVM interface creates an abstraction layer that hides the patch structure from the developers, such that they only need to handle one edge at a time.

```fortran
! Note: this is framework code that should not be modified by the developer.
! Application-specific code should be provided in FVM_compute_fluxes_single().
real, dimension(N,_FVM_Q_SIZE)   :: qL, qR     ! Data from cells to
real, dimension(N,_FVM_AUX_SIZE) :: auxL, auxR ! left/right of each edge
real, dimension(N,2) :: normals ! Vectors that are normal to each edge

! [...]

!$OMP SIMD PRIVATE(waveSpeed) REDUCTION(max: maxWaveSpeed)
do i=1,N ! Loop for all N Riemann problems
    !DIR$ FORCEINLINE
    call FVM_compute_fluxes_single(normals(i,:),qL(i,:),qR(i,:),auxL(i,:),auxR(i,:),
        ↪ fluxL(i,:),fluxR(i,:),waveSpeed)
    maxWaveSpeed = max(maxWaveSpeed, waveSpeed)
end do
```

Report 8.1: Excerpt from the optimization report provided by the Intel Fortran Compiler when compiling sam(oa)^2 and the FVM interface for a KNL processor. This excerpt refers to a compilation of an application in which the augmented Riemann solver is applied via the `single-edge` version of the `compute-fluxes` operator.

```
LOOP BEGIN at src/FVM/FVM_euler_timestep.f90(383,25)
      remark #15301: OpenMP SIMD LOOP WAS VECTORIZED
      remark #15448: unmasked aligned unit stride loads: 16
      remark #15449: unmasked aligned unit stride stores: 14
      remark #15475: --- begin vector cost summary ---
      remark #15476: scalar cost: 2864
      remark #15477: vector cost: 1065.370
      remark #15478: estimated potential speedup: 1.750
      remark #15486: divides: 61
      remark #15488: --- end vector cost summary ---
LOOP END
```

the `patch-update` kernel that repeatedly calls it in Code 8.6. While in the `patch-update` kernel the framework loops through a list with N Riemann problems extracting the input data of each problem and calling the operator with the respective data, the `single-edge compute-fluxes` operator only needs to extract the quantities from its input arrays and call the Riemann solver with the appropriate parameters. Note that the framework code already includes an `!$OMP SIMD` directive for its main loop, so that developers do not need to explicitly implement vectorization. However, successful vectorization of this loop will only be possible if the code provided for the operator does not contain obstacles for vectorization, as discussed previously in Chapter 2. Note also that in the framework code we use Fortran subarrays, which provide a useful way to reorganize the data used as input and output for the operator call, while still supporting vectorization by the Intel Fortran Compiler – as confirmed by the optimization reports provided by the compiler (see the example in Report 8.1).

Multi-edge operator: in this more advanced version of the operator, developers can provide their own optimized code for the main loop that iterates through the Riemann problems. Instead of dealing with a single Riemann problem, the multi-edge operator takes a list of problems as input, such that developers can have complete control of the loop that iterates through the list, as well as of the compiler directives for vectorization. For illustration, see Code 8.7, where we show our simple implementation of this operator for the single-layer shallow water

Code 8.7: Implementation of the `multi-edge` version of the `compute-fluxes` operator. Note that in this case, the main loop needs to be implemented inside the operator, as well as its vectorization.

```fortran
subroutine FVM_compute_fluxes_multi(normals,qL,qR,auxL,auxR,fluxL,fluxR,maxWaveSpeed)
    real, dimension(N,2),              intent(in)  :: normals !Normal vectors
    real, dimension(N,_FVM_Q_SIZE),    intent(in)  :: qL,qR
    real, dimension(N,_FVM_AUX_SIZE),  intent(in)  :: auxL,auxR
    real, dimension(N,_FVM_Q_SIZE),    intent(out) :: fluxL,fluxR
    real,                              intent(out) :: maxWaveSpeed
    ! Local variables: These should all be declared PRIVATE in the OMP SIMD directive
    real :: hL,huL,hvL,bL, hR,huR,hvR,bR ! Input for Riemann solver
    real :: flux_hL,flux_huL,flux_hvL ! Output left-going fluxes
    real :: flux_hR,flux_huR,flux_hvR ! Output right-going fluxes
    real :: waveSpeed ! Output wave speed

    !$OMP SIMD PRIVATE(normal,hL,huL,[...],waveSpeed) REDUCTION(max: maxWaveSpeed)
    do i=1,N ! Loop for all N Riemann problems
        !Extract data from input arrays to iteration-private variables
        hL = qL(i,1); huL = qL(i,2); hvL = qL(i,3); bL = auxL(i,1)
        hR = qR(i,1); huR = qR(i,2); hvR = qR(i,3); bR = auxR(i,1)
        normal = normals(i,:)

        !DIR$ FORCEINLINE
        call RiemannSolver(normal, hL,huL,hvL,bL, hR,huR,hvR,bR, flux_hL,flux_huL,
            ↪ flux_hvL, flux_hR,flux_huR,flux_hvR, waveSpeed)

        ! Copy computed fluxes to output arrays
        fluxL(i,:) = [flux_hL, flux_huL, flux_hvL]
        fluxR(i,:) = [flux_hR, flux_huR, flux_hvR]

        ! Compute max. wave speed
        maxWaveSpeed = max(maxWaveSpeed, waveSpeed)
    end do
end subroutine
```

equations, in which we also use the `!$OMP SIMD` directive to achieve auto-vectorization of the loop. Note that the implementation of this operator can actually become considerably more complex, allowing more advanced developers to exploit other techniques to achieve vectorization (intrinsic functions, etc.) or attempt other optimizations.

While the `single-edge` version certainly requires simpler implementations, the `multi-edge` version is expected to deliver higher performance, even for very similar implementations like the ones shown, because of the overhead introduced by the Fortran subarrays in the framework code that uses the `single-edge` version. This will be discussed further in in Section 8.2, where we compare the performance obtained by FVM applications developed using the `single-edge` and `multi-edge` versions of this operator, implemented as shown in the examples above.

Update-Cell Operator

This FVM operator defines how the `Q` quantities of each cell should be updated with the Riemann solutions computed by the `compute-fluxes` operator. After obtaining the numerical fluxes at each edge in the mesh, sam(oa)^2 computes the sum of the fluxes leaving/entering each cell through each of its three edges. The total flux for each cell is then used to update the cell-averaged quantities, typically applying the update scheme defined by (7.1). However, in the operator implementation developers have flexibility to modify this and may also take care of special cases that arise for the particular problem (e.g., drying or wetting of cells in simulations with the shallow water equations). In addition to updating the cell quantities, the operator should

Code 8.8: Implementation of the `update-cell` operator, which typically applies the update scheme given by (7.1), but also allows customizations, like the handling of dry cells shown here.

```
subroutine FVM_update_cell(vertices,Q,Aux,total_flux,dt,cell_volume,refinement_flag)
    real, intent(in)      :: vertices(2,3)
    real, intent(inout)   :: Q(_FVM_Q_SIZE)
    real, intent(inout)   :: Aux(_FVM_AUX_SIZE)
    real, intent(in)      :: total_flux(_FVM_Q_SIZE)
    real, intent(in)      :: dt
    real, intent(in)      :: cell_volume
    integer, intent(out)  :: refinement_flag

    ! Update cell quantities
    Q(:) = Q(:) - (dt * total_flux(:) / cell_volume)

    ! Handle cell drying: if cell becomes dry, set water level to 0 and momentum to 0
    if (Q(1) < dry_tolerance) then ! if (h < dry_tolerance)
        Q(1) = 0.0 ! h = 0
        Q(2) = 0.0 ! hu = 0
        Q(3) = 0.0 ! hv = 0
    end if

    ! Set refinement flag according to some application-specific criteria.
    ! In this example, we decide whether to refine/keep/coarsen the cell
    ! simply based on the variation in its water height:
    refinement_flag = 0
    if (abs(total_flux(1)) > refine_threshold) then
        refinement_flag = 1  ! Refine this cell
    else if (abs(total_flux(1)) < coarsen_threshold) then
        refinement_flag = -1 ! Coarsen this cell
    else
        refinement_flag = 0  ! Keep this cell
    end if
end subroutine
```

also return a refinement flag for the cell, to inform whether it should be refined (1), kept (0) or coarsened (-1) for the next time step. For illustration, consider the example shown in Code 8.8.

Split-Cell and Merge-Cells Operators

These two FVM operators control how adaptivity is performed on the cells. The `split-cell` operator, which is repeatedly called by the `refine` kernel, receives the data from a cell $\mathcal{C}^{(in)}$ as input, and returns as output the data of two finer cells $\mathcal{C}_1^{(out)}$ and $\mathcal{C}_2^{(out)}$, obtained by splitting $\mathcal{C}^{(in)}$ in half. On the other hand, the `merge-cells` operator is used by the `coarsen` kernel and performs the inverse operation. More specifically, it takes data from two neighbor cells $\mathcal{C}_1^{(in)}$ and $\mathcal{C}_2^{(in)}$ as input, and returns data for a coarse cell $\mathcal{C}^{(out)}$ that results from merging both input cells.

Both these operations are often performed by simply copying or interpolating cell quantities, but as usual we allow developers to implement further customizations. That may be useful to handle special cases depending on the PDE being solved, like the check for dry states. For instance, consider the example implementation for the `split` operator shown in Code 8.9, where, as done previously in Code 7.3, we avoid generating spurious waves by properly handling dry states when refining the cell.

8.2 Performance Results

As mentioned before, we used the FVM interface to create two FVM applications for simulations with the the single-layer and two-layer shallow water equations. In the following, we refer to

Code 8.9: Implementation of the `split-cell` operator, which defines how a cell should be split into two when performing its refinement. This operator is repeatedly applied by the `refine` kernel of the `adaptive` traversal, which is responsible for refining an entire patch.

```
subroutine FVM_split_cell(Q,Aux,vertices1,vertices2,Q_out1,Q_out2,Aux_out1,Aux_out2)
    real, intent(in)  :: Q(_FVM_Q_SIZE), Aux(_FVM_AUX_SIZE) ! Data in coarsen cell
    real, intent(in)  :: vertices1(2,3), vertices2(2,3) ! Vertices of fine cells
    real, intent(out) :: Q_out1(_FVM_Q_SIZE), Q_out2(_FVM_Q_SIZE) ! Output data for
    real, intent(out) :: Aux_out1(_FVM_AUX_SIZE),Aux_out2(_FVM_AUX_SIZE) ! fine cells
    ! Local:
    real :: h, hu, hv, b     ! Data in original coarse cell (comes from Q and Aux)
    real :: h1, hu1, hv1, b1 ! Data in first fine cell (goes to Q_out1 and Aux_out1)
    real :: h2, hu2, hv2, b2 ! Data in second fine cell (goes to Q_out2 and Aux_out2)

    ! Extract data from original coarse cell:
    h = Q(1) ; hu = Q(2)
    hv = Q(3) ; b = Aux(1)

    ! Bathymetry data does not come directly from the original cell.
    ! Instead, the cell-averaged bathymetry is recomputed for the new cell:
    b1 = getBathymetryAtCell(vertices1)
    b2 = getBathymetryAtCell(vertices2)

    ! Maintain original water elevation h+b despite possible changes in bathymetry:
    h1 = h + b - b1
    h2 = h + b - b2

    ! Keep the water momentum from the original coarse cell:
    hu1 = hu ; hv1 = hv
    hu2 = hu ; hv2 = hv

    ! Handle drying of first fine cell:
    if (h1 < dry_tolerance) then ! If cell is dry, its quantities should be zero
        h1 = 0.0 ; hu1 = 0.0 ; hv1 = 0.0
    else if (h < dry_tolerance) then ! Otherwise, if the original cell was dry,
        h1 = -b1 ! we need to force h+b=0 to avoid creating spurious waves.
    end if

    if (h2 < dry_tolerance) then ! Now do the same for second fine cell:
        h2 = 0.0 ; hu2 = 0.0 ; hv2 = 0.0
    else if (h < dry_tolerance) then
        h2 = -b2
    end if

    ! Store data for fine cells in the output arrays
    Q_split1 = [h1,hu1,hv1] ; Aux_split1 = [b1]
    Q_split2 = [h2,hu2,hv2] ; Aux_split2 = [b2]
end subroutine
```

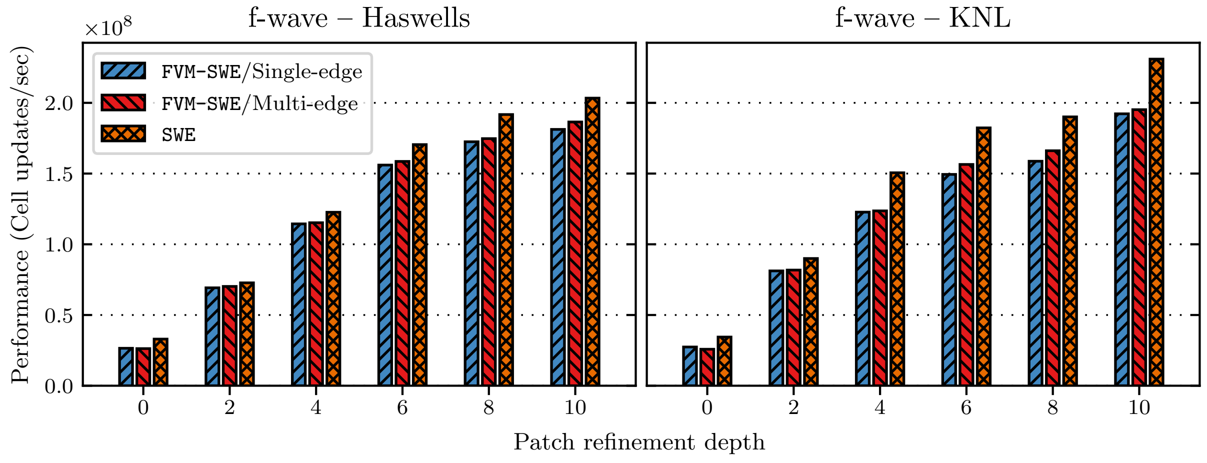


Figure 8.1: Performance of `FVM-SWE` and `SWE` using the f-wave solver.

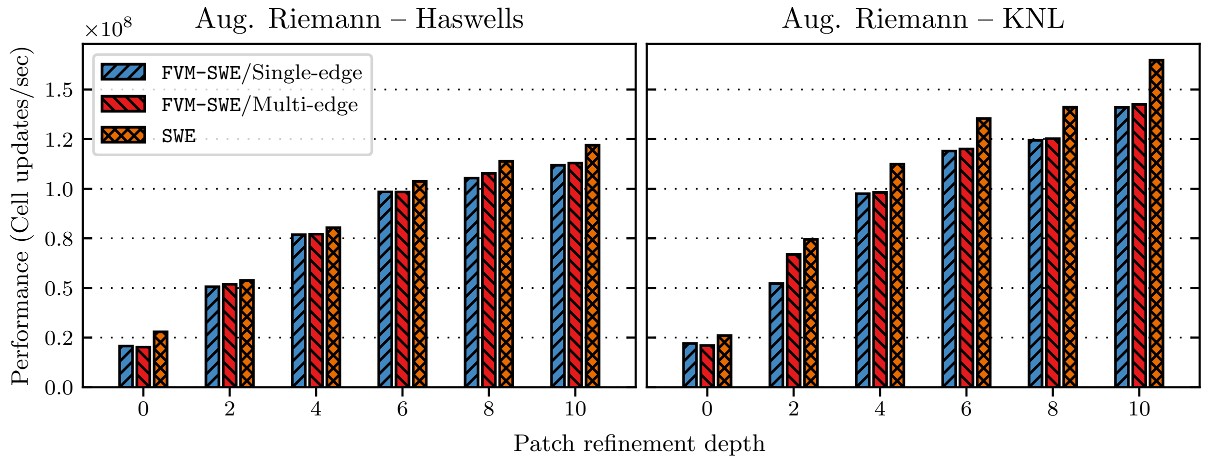


Figure 8.2: Performance of `FVM-SWE` and `SWE` using the augmented Riemann solver.

those FVM applications as `FVM-SWE` and `FVM-SWE2L`, in contrast to the original applications implemented without the FVM interface (and described in Chapter 7), to which we refer as `SWE` and `SWE2L`. While the implementations of `SWE` and `SWE2L` are considerably more complex because their codes deal directly with the data structures and algorithms used in sam(oa)^2, that may be advantageous in terms of performance, because they do not have the overhead due to additional memory management performed by the FVM interface's abstraction layer. Therefore, we will use their performance as baseline for evaluating the new applications `FVM-SWE` and `FVM-SWE2L`.

As usual, the experiments described in this section have been conducted on the experimental platforms described in Chapter 5. We also use the same simulation scenarios as in the previous chapter: the Tohoku 2011 tsunami for the single-layer and the parabolic bowl-shaped lake for the two-layer shallow water equations. For more details on these scenarios, see Section 7.3.1.

8.2.1 Simulation Performance

We start by evaluating the performance of the FVM applications `FVM-SWE` and `FVM-SWE2L` with different implementations for the `compute-fluxes` operator and comparing them with

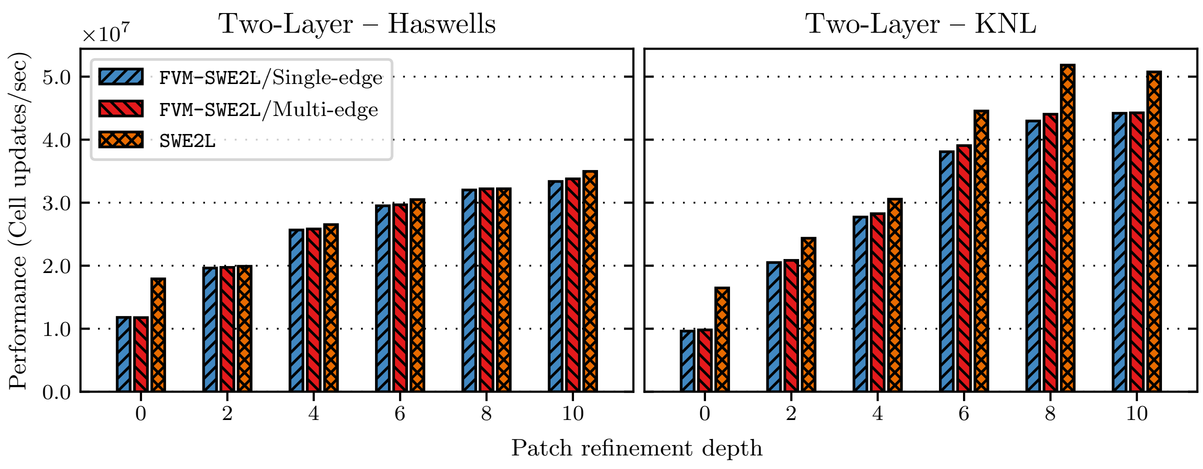


Figure 8.3: Performance of FVM-SWE2L and SWE2L.

the performance achieved by the original applications SWE and SWE2L. Note that all experiments reported here have been executed using the same configurations as the ones described in Section 7.3. We plot the performance (measured as *cell updates per second*) of the entire simulations using the vectorized f-wave (Fig. 8.1), augmented Riemann (Fig. 8.2) and two-layer (Fig. 8.3) solvers, where we compare the performance achieved by simulations implemented using the `single-edge` and `multi-edge` versions of the `compute-fluxes` operator. In addition, we also plot the performance of the original (vectorized) applications SWE and SWE2L, which we consider as the target performance for the FVM applications.

Noticeably, the codes developed using `single-edge` version for the operator perform only slightly slower than the ones that use the `multi-edge` versions (up to 3% and 4% slower on Haswells and KNLs, respectively), despite the overhead introduced due to the use of Fortran subarrays in the operator calls (as shown in Code 8.6). This not only reveals that this overhead is not so high, but also confirms that the Intel Fortran Compiler is able to efficiently handle the subarrays when vectorizing the loop. As such, this shows that the `single-edge` version of the `compute-fluxes` is able to produce applications with reasonable vectorization efficiency, while not requiring application developers to explicitly implement any vectorization approach.

Furthermore, the vectorized FVM-SWE and FVM-SWE2L implementations also achieve performances similar to their analogous applications (SWE and SWE2L) on Haswells (1–9% slower). On the other hand, on the KNLs there are more noticeable differences in performance (8–18% slower). Nevertheless, these experiments show that despite the additional memory operations performed to create the interface's layer of abstraction, the FVM applications can achieve performance comparable to applications developed directly within the complex framework code.

The only exceptions for this are the experiments performed with refinement depth $d = 0$, for which it is clear that SWE and SWE2L perform much faster than FVM-SWE and FVM-SWE2L. However, that can be easily explained by the fact that the applications SWE and SWE2L provide implementations that were specifically designed and optimized for cases where cell-wise adaptivity is used, while FVM-SWE and FVM-SWE2L always use a patch-based implementation that was designed to support any arbitrary value for d. As such, if $d = 0$, these implementations unnecessarily solve all Riemann problems in the mesh twice (because all three edges in patches with $d = 0$ are processed with the help of ghost layers, as they are all located at the patch boundary). Since we are generally only interested in simulations with $d > 0$, which deliver much higher performance and shorter time-to-solution, we did not care about optimizing the

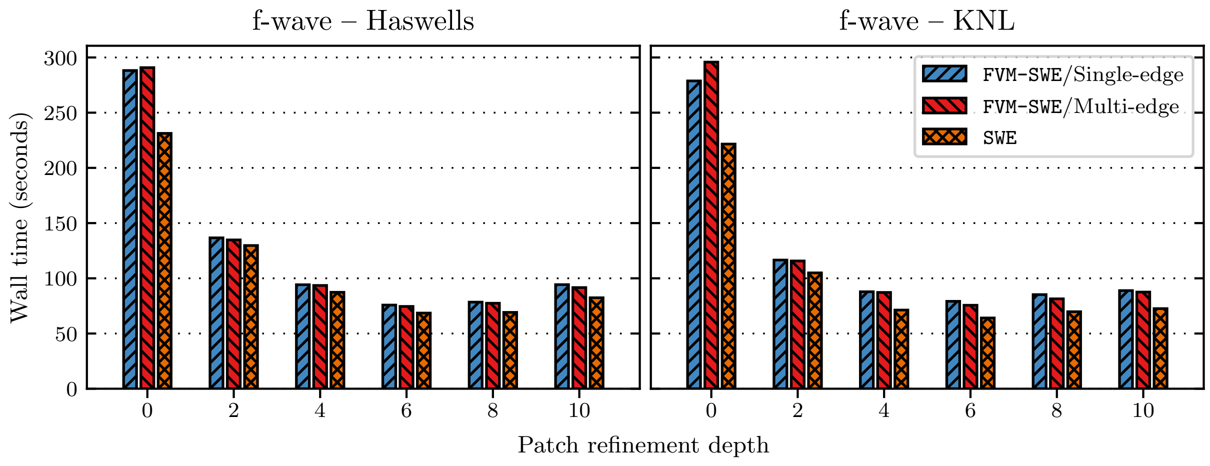


Figure 8.4: Time-to-solution of `FVM-SWE` and `SWE` using the f-wave solver.

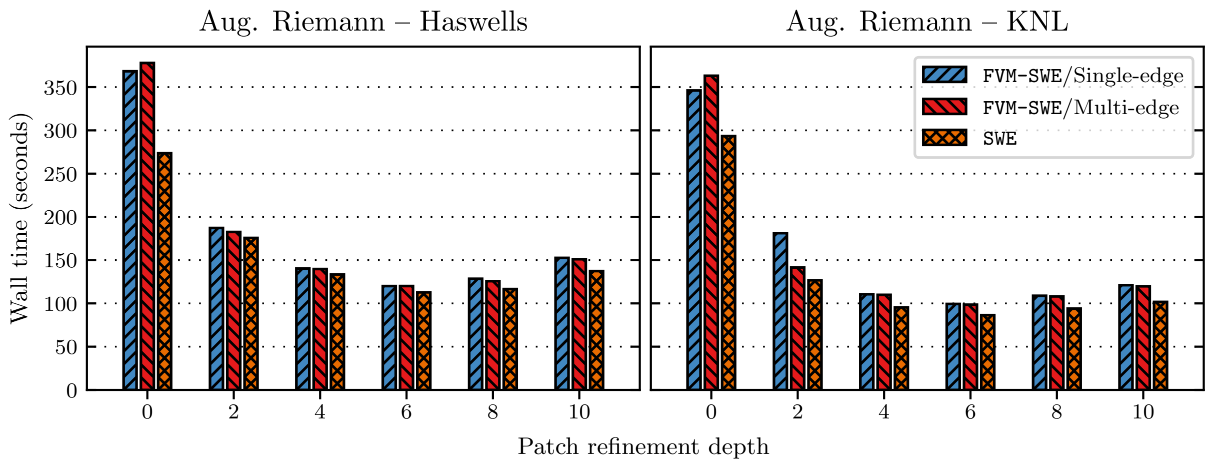


Figure 8.5: Time-to-solution of `FVM-SWE` and `SWE` using the augmented Riemann solver.

FVM interface for cell-wise adaptivity.

8.2.2 Time-to-Solution

As discussed in Chapter 7, although larger patches usually deliver higher throughputs, they may not always produce the fastest simulations, because they usually also increase the total number of cells and edges in the adaptive mesh. Thus, now we also consider the wall time taken by our experiments – see the plots in Figs. 8.4 to 8.6. As before, we can see that `FVM-SWE` and `FVM-SWE2L` behave very similarly to `SWE` and `SWE2L` , achieving only slightly longer time-to-solution in all simulations. Also, comparisons between the time taken by the FVM applications implemented with the `single-edge` and `multi-edge` versions of the `compute-fluxes` operator show again that they achieve very similar performance.

In addition, these results reinforce our previous experience that patches with 64–256 each are usually able to minimize the time-to-solution on the patch-based adaptive meshes in sam(oa)^2, since also for the applications developed with the FVM interface they achieved the fastest results.

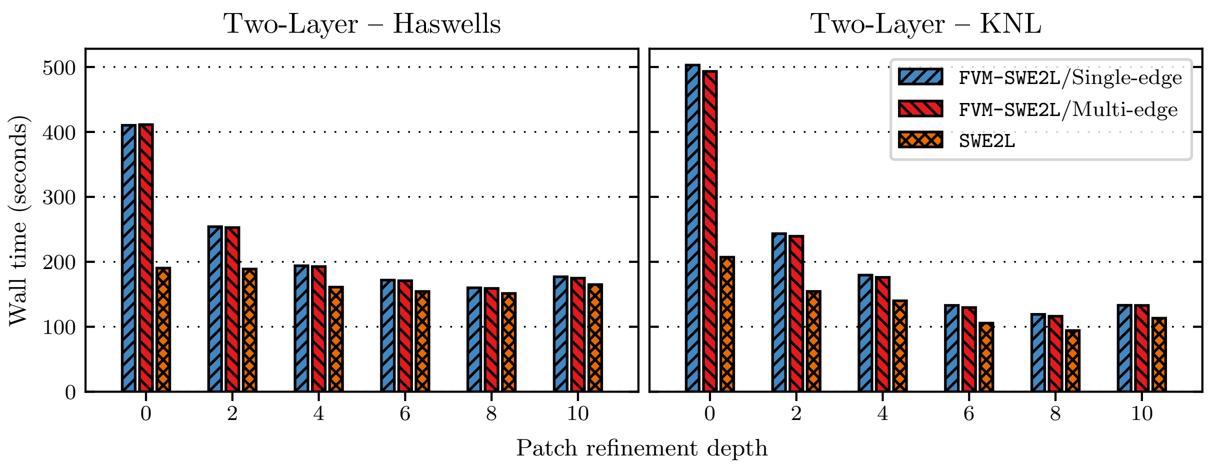


Figure 8.6: Time-to-solution of `FVM-SWE2L` and `SWE2L`.

8.3 Conclusions

In this chapter we described and evaluated the FVM interface, a programming interface that supports the creation of Godunov-type finite volume methods in sam(oa)^2. Thanks to its simple abstraction layer, application developers with no HPC expertise can create high-performance applications with adaptive mesh refinement and various HPC features (including cache-efficient traversals, multi-level parallelism and vectorization), while only needing to implement problem-specific algorithms. More experienced developers also have the option of assuming complete control over the main solver loop, such that they can manage its vectorization or attempt further optimizations on it.

The FVM interface provides a generalization of the vectorized patch-based finite volume scheme we developed for sam(oa)^2 in Chapter 7, allowing its customization for different systems of PDEs with relative ease. Assuming that a Riemann solver for the specific problem is available, the developer's work is reduced mainly to handling the solver calls and providing application-specific operators for initialization, refinement and coarsening of cells, which in many cases consist of trivial implementations. In particular, the FVM applications we implemented directly apply the same Riemann solver implementations that were used for GeoClaw in Chapter 6 and for sam(oa)^2 in Chapter 7.

Our performance experiments revealed successful vectorization of the calls to the Riemann solvers, both when vectorization was implicitly managed by the framework or explicitly managed by the developer (achieving similar performances for both cases). They also showed that the FVM applications achieve performance comparable to that of their analogous applications, which were developed directly into the complex source code of sam(oa)^2. Compared to those, the FVM applications exhibit only low to moderate overhead (1–18% slower), mainly due to the extra memory operations that are performed to create the interface's abstraction layer, which effectively hides from developers the complex algorithms and data structures managed by the underlying framework.

9 Summary

In this work we developed vectorized implementations of the finite volume schemes provided by two different PDE frameworks with support for adaptive mesh refinement: GeoClaw and sam(oa)^2. Our modifications in both frameworks led to substantial performance improvements, which were experimentally evaluated on two modern high-performance architectures and in the context of two variations of the shallow water equations. Additionally, we also developed a generalization of the vectorized patch-based finite volume scheme that was developed for sam(oa)^2, which allows easy customization of our approach to other systems of hyperbolic PDEs. In the following, we give an overview of the main specific contributions of this thesis.

9.1 Main Contributions

First, we have been able to achieve efficient compiler auto-vectorization of the numerical routines in the GeoClaw package. Most importantly, we achieved vectorization of five different Riemann solvers for two variations of the shallow water equations (three *normal* solvers and two *transverse* solvers), reporting speedups in the solver throughputs of up to 2.1× with double-precision arithmetic on machines with AVX2 SIMD instructions (Haswells) and of up to 5.6× on machines with AVX-512 SIMD instructions (KNLs). In particular, we highlight the successful vectorization of the augmented Riemann solver and of the solver for the two-layer shallow water equations, despite the high complexity of their implementations. To the best of our knowledge, no previous work has reported success with compiler auto-vectorization approaches for those two solvers.

Second, with the intention of applying vectorization to the finite volume schemes in sam(oa)^2, we replaced its cell-wise adaptive mesh refinement strategy with a patch-based discretization. This modification not only allowed vectorization, but also brought various other performance benefits. These include improvements in the memory throughput due to the regular and more concise data structures, as well as an overall reduction in the size and complexity of the implicit refinement tree, which consequently reduces the overhead for managing adaptive mesh refinement. Our performance experiments show that, even without vectorization, switching to a patch-based discretization resulted in significant speedups of up to 1.6× on Haswells and up to 1.9× on KNLs.

Third, the patch-based vectorization in sam(oa)^2 allowed us to use the same solver implementations and the same vectorization approach as used previously in GeoClaw. Vectorization of the normal Riemann solvers in sam(oa)^2 achieved speedups similar to what we experienced in GeoClaw, resulting in overall speedups in the solver throughputs of up to 3.6× on Haswells and up to 6.7× on KNLs, compared to the original implementations with cell-wise adaptivity and no vectorization.

Fourth, we performed time-to-solution analyses of the resulting finite volume schemes in sam(oa)^2, with respect to the size of the patches being applied. Since using larger patches tends to increase the total number of cells in the adaptive mesh, and as a consequence in the computational work required to achieve a given solution accuracy, it was clear since the beginning that the patch sizes should be carefully chosen in order to minimize the execution time. Thus, we experimentally evaluated the influence of different patch sizes to the solver throughputs, to the total number of cells in the mesh, to the total number of Riemann solves performed and, finally, to the resulting time-to-solution. Our analyses show that for our applications and experimental platforms, best results are achieved by relatively small patches with 64–256 cells each, for which we report up to 83% reductions in the execution times.

Fifth, we created a highly customizable generalization of the patch-based adaptive mesh refinement and vectorization approaches developed for sam(oa)^2. Customization is accomplished via a programming interface that effectively hides from application developers the complicated data structures and algorithms managed by the underlying framework, allowing developers to easily apply the high-performance finite volume schemes developed in this work to various other systems of PDEs. Our performance experiments with that interface revealed that, although the extra abstraction layer does introduce some overhead to the resulting applications, their performance is still reasonably similar (1–18% slower in the best and worst cases) to that achieved by applications developed directly into sam(oa)^2 without the new interface, thus requiring much more implementation effort. In addition, our implementations and experiments using the new interface show that it is able to achieve efficient vectorization of the numerical routines provided by application developers, even when vectorization is not explicitly handled by them.

As a final remark, we reinforce the importance of our achievements both in performance and usability in sam(oa)^2. Here we quote a piece of the concluding remarks of the previous work by Meister [49], which served as a starting point for this thesis:

> *When sam(oa)^2 is compared to other software in the field, its performance is a clear selling point. However, to become truly competitive there are still some issues left that need to be resolved. First, the implementation of scenarios in the interface is too complicated. A kernel layer that couples sam(oa)^2 with an abstract description or to a library with a fully opaque interface would be beneficial. [...]*

Taking these words into consideration, we conclude that our work brings meaningful contributions to sam(oa)^2, as we did not only substantially improve the performance of a framework that was already recognized as being highly efficient, but also considerably simplified the process of implementing new numerical solvers within it, effectively addressing one of the main shortcomings of the framework according to that previous work.

9.2 Suggestions for Future Work

Below we list a few directions that future research could follow with the intention of improving or giving continuity to the work described in this thesis:

- Reformulation of the two-layer solvers: our performance experiments both with GeoClaw and with sam(oa)^2 show that the Riemann solvers for the two-layer shallow water equations do not profit from vectorization as much as the solvers for the single-layer equations. While in this work we chose to not deviate too much from the original solver implementations, it can be expected that restructuring the code of the two-layer solvers could be highly beneficial for their vectorization performance, especially if such a reformulation is able to reduce the size and amount of execution branches.

- Improvement of parallelization efficiency in GeoClaw: our experimental results indicate that, although the parallelization strategy in GeoClaw achieves reasonable parallelization efficiency for platforms with relatively few cores, it does not scale accordingly to many-core processors like KNLs. Thus, we suggest that future research should focus on attempting to adapt GeoClaw for the next generations of high-performance architectures, which are also likely to require efficient scaling to hundreds of threads.

- Development and evaluation of other FVM applications: while the applications developed for the single-layer and two-layer shallow water equations require different Riemann solvers and differ from each other in a few other implementation details, they are still relatively similar in several aspects. Therefore, it would be interesting to have other applications developed in sam(oa)^2 with the FVM interface, preferably by developers that never had contact with the sam(oa)^2 framework – this would be useful to further demonstrate the flexibility and usability of the FVM interface.

- Extension of the FVM interface using code generation approaches: with the FVM interface we facilitated the process of implementing a new simulation in sam(oa)^2 considerably. However, this could be taken one step further by applying concepts of automatic code generation to allow application developers to deal only with higher-lever constructs when implementing the FVM operators. As an example of what can be accomplished, we mention the FEniCS project [40], which provides automatic code generation for PDE solvers in the context of finite element methods.

- Development of higher order methods in sam(oa)^2: while GeoClaw provides an implementation of the corner-transport upwind method with addition of second-order corrections, sam(oa)^2 implements only the much simpler donor-cell upwind method with no second-order corrections. A future research more focused on the numerical models could strive to implement more accurate models in sam(oa)^2, which could also include higher-order methods. Considering the modifications performed by this work, this may possibly be a less complex task by now, as the grid traversals in sam(oa)^2 are not restricted only to element-oriented stencils anymore (because of the new patch-based discretization that now allows mesh elements to have access to the data of other elements located farther away in the mesh).

- Extension to other numerical methods: although this work has focused only on finite volume methods, sam(oa)^2 actually supports other types of numerical methods, such as finite element and discontinuous Galerkin methods. Thus, future work could attempt to implement the strategies described here in the context of other methods. This may include not only applying the patch-based and vectorization approaches to those other methods, but also developing generic interfaces for them, using concepts similar to the ones used for the FVM interface.

Bibliography

[1] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, third edition, 1999.

[2] M. Bader, C. Böck, J. Schwaiger, and C. A. Vigh. Dynamically Adaptive Simulations with Minimal Memory Requirement – Solving the Shallow Water Equations Using Sierpinski Curves. *SIAM Journal on Scientific Computing*, 32(1):212–228, 2010.

[3] M. Bader, A. Breuer, W. Hölzl, and S. Rettenberger. Vectorization of an Augmented Riemann Solver for the Shallow Water Equations. In *High Performance Computing & Simulation (HPCS), 2014 International Conference on*, pages 193–201. IEEE, 2014.

[4] D. S. Bale, R. J. LeVeque, S. Mitran, and J. A. Rossmanith. A wave propagation method for conservation laws and balance laws with spatially varying flux functions. *SIAM Journal on Scientific Computing*, 24(3):955–978, 2002.

[5] P. Barry and P. Crowley. *Modern Embedded Computing: Designing Connected, Pervasive, Media-Rich Systems*. Elsevier, 2012.

[6] J. Behrens, N. Rakowsky, W. Hiller, D. Handorf, M. Luter, J. Ppke, and K. Dethloff. amatos: Parallel adaptive mesh generator for atmospheric and oceanic simulation. *Ocean Modelling*, 10(1–2):171–183, 2005. The Second International Workshop on Unstructured Mesh Numerical Modelling of Coastal, Shelf and Ocean Flows.

[7] M. Berger and I. Rigoutsos. An algorithm for point clustering and grid generation. *IEEE Transactions on Systems, Man, and Cybernetics*, 21(5):1278–1286, 1991.

[8] M. J. Berger and P. Colella. Local adaptive mesh refinement for shock hydrodynamics. *Journal of Computational Physics*, (82):64–84, 1989.

[9] M. J. Berger, D. L. George, R. J. LeVeque, and K. T. Mandli. The GeoClaw software for depth-averaged flows with adaptive refinement. *Advances in Water Resources*, 34(9):1195–1206, 2011.

[10] M. J. Berger and J. Oliger. Adaptive mesh refinement for hyperbolic partial differential equations. *Journal of Computational Physics*, (53):484–512, 1984.

[11] F. Bouchut and V. Zeitlin. A robust well-balanced scheme for multi-layer shallow water equations. *Discrete and Continuous Dynamical Systems-Series B*, 13(4):739–758, 2010.

[12] C. Burstedde, D. Calhoun, K. Mandli, and A. R. Terrel. ForestClaw: Hybrid forest-of-octrees AMR for hyperbolic conservation laws. In *Parallel Computing: Accelerating Computational Science and Engineering (CSE)*, volume 25 of *Advances in Parallel Computing*, pages 253–262, 2014.

[13] C. Burstedde, L. C. Wilcox, and O. Ghattas. `p4est`: Scalable Algorithms for Parallel Adaptive Mesh Refinement on Forests of Octrees. *SIAM Journal on Scientific Computing*, 33(3):1103–1133, 2011.

[14] Clawpack Development Team. Clawpack GitHub repository. `www.github.com/clawpack/clawpack`. Accessed in June 2019.

[15] Clawpack Development Team. Clawpack software. `www.clawpack.org`. Version 5.5.0. Accessed in June 2019.

[16] P. Colella, D. Graves, J. Johnson, H. Johansen, N. Keen, T. Ligocki, D. Martin, P. McCorquodale, D. Modiano, P. Schartz, T. Sternberg, and B. Van Straalen. Chombo Software Package for AMR Applications Design Document. `http://web.mit.edu/ehliu/Public/ehliu/chomboDesign.pdf`. Accessed in June 2019.

[17] R. Deiterding. Amroc – a generic framework for blockstructured adaptive mesh refinement in object-oriented c++. `http://amroc.sourceforge.net/`. Accessed in June 2019.

[18] R. Donat, M. C. Martí, A. Martínez-Gavara, and P. Mulet. Well-balanced adaptive mesh refinement for shallow water flows. *Journal of Computational Physics*, 257:937–953, 2014.

[19] A. Dubey, A. Almgren, J. Bell, M. Berzins, S. Brandt, G. Bryan, P. Colella, D. Graves, M. Lijewski, F. Löffler, B. O'Shea, E. Schnetter, B. V. Straalen, and K. Weide. A survey of high level frameworks in block-structured adaptive mesh refinement packages. *Journal of Parallel and Distributed Computing*, 74(12):3217–3227, 2014. Domain-Specific Languages and High-Level Frameworks for High-Performance Computing.

[20] A. Dubey, K. Antypas, M. K. Ganapathy, L. B. Reid, K. Riley, D. Sheeler, A. Siegel, and K. Weide. Extensible component-based architecture for flash, a massively parallel, multiphysics simulation code. *Parallel Computing*, 35(10-11):512–522, 2009.

[21] B. Einfeldt. On Godunov-type methods for gas dynamics. *SIAM Journal on Numerical Analysis*, 25(2):294–318, 1988.

[22] C. Feichtinger, S. Donath, H. Köstler, J. Götz, and U. Rüde. WaLBerla: HPC software design for computational engineering simulations. *Journal of Computational Science*, 2(2):105–112, 2011.

[23] C. R. Ferreira and M. Bader. Load Balancing and Patch-Based Parallel Adaptive Mesh Refinement for Tsunami Simulation on Heterogeneous Platforms Using Xeon Phi Coprocessors. In *Proceedings of the Platform for Advanced Scientific Computing Conference*. ACM, 2017.

[24] C. R. Ferreira and M. Bader. A Generic Interface for Godunov-type Finite Volume Methods on Adaptive Triangular Meshes. In *International Conference on Computational Science*, pages 402–416. Springer International Publishing, 2019.

[25] C. R. Ferreira, K. T. Mandli, and M. Bader. Vectorization of Riemann solvers for the single-and multi-layer Shallow Water Equations. In *2018 International Conference on High Performance Computing & Simulation (HPCS)*, pages 415–422. IEEE, 2018.

[26] P. Galvez, J.-P. Ampuero, L. A. Dalguer, S. N. Somala, and T. Nissen-Meyer. Dynamic earthquake rupture modelled with an unstructured 3-D spectral element method applied to the 2011 M9 Tohoku earthquake. *Geophysical Journal International*, 198(2):1222–1240, 2014.

[27] General Bathymetric Chart of the Oceans. GEBCO 30 arc-second grid. `https://www.gebco.net/data_and_products/gridded_bathymetry_data/gebco_30_second_grid/`. Accessed in June 2019.

[28] Geoclaw Development Team. Geoclaw. `www.geoclaw.org`. Accessed in June 2019.

[29] D. L. George. Augmented Riemann solvers for the shallow water equations over variable topography with steady states and inundation. *Journal of Computational Physics*, 227(6):3089–3113, 2008.

[30] B. T. Gunney and R. W. Anderson. Advances in patch-based adaptive mesh refinement scalability. *Journal of Parallel and Distributed Computing*, 89:65–84, 2016.

[31] Intel Corporation. A Guide to Vectorization with Intel C++ Compilers. `https://software.intel.com/sites/default/files/m/4/8/8/2/a/31848-CompilerAutovectorizationGuide.pdf`. Accessed in October 2019.

[32] Intel Corporation. Fortran Array Data and Arguments and Vectorization. `https://software.intel.com/en-us/articles/fortran-array-data-and-arguments-and-vectorization`. Accessed in June 2019.

[33] Intel Corporation. Programming Guidelines for Vectorization. `https://software.intel.com/en-us/fortran-compiler-developer-guide-and-reference-programming-guidelines-for-vectorization`. Accessed in June 2019.

[34] J. Jeffers and J. Reinders. *Intel Xeon Phi Coprocessor High Performance Programming.* Newnes, 2013.

[35] M. Lastra, M. J. C. Díaz, C. Ureña, and M. de la Asunción. Efficient multilayer shallow-water simulation system based on gpus. *Mathematics and Computers in Simulation*, 2017.

[36] Leibniz Supercomputing Center. LRZ: Linux-Cluster, 2019. Accessed in June 2019.

[37] Leibniz Supercomputing Center. Many-Core Cluster CoolMUC-3 at LRZ, 2019. Accessed in June 2019.

[38] R. J. LeVeque. *Finite Volume Methods for Hyperbolic Problems.* Cambridge University Press, 2002.

[39] R. J. LeVeque, D. L. George, and M. J. Berger. Tsunami modelling with adaptively refined finite volume methods. *Acta Numerica*, 20:211–289, 2011.

[40] A. Logg, K.-A. Mardal, G. N. Wells, et al. *Automated Solution of Differential Equations by the Finite Element Method.* Springer, 2012.

[41] J. Luitjens and M. Berzins. Scalable parallel regridding algorithms for block-structured adaptive mesh refinement. *Concurrency and Computation: Practice and Experience*, 23(13):1522–1537, 2011.

[42] J. P. Luitjens. *The scalability of parallel adaptive mesh refinement within Uintah.* PhD thesis, The University of Utah, 2011.

[43] J. Macías, C. Pares, and M. J. Castro. Improvement and generalization of a finite element shallow-water solver to multi-layer systems. *International journal for numerical methods in fluids*, 31(7):1037–1059, 1999.

[44] A. Malcher. Performance optimization of riemann solvers for the shallow water equations. Master's thesis, Department of Informatics, Technical University of Munich, May 2016.

[45] K. T. Mandli. *Finite Volume Methods for the Multilayer Shallow Water Equations with Applications to Storm Surges.* PhD thesis, University of Washington, 2011.

[46] K. T. Mandli. A Numerical Method for the Two Layer Shallow Water Equations with Dry States. *Ocean Modelling*, 72:80–91, 2013.

[47] K. T. Mandli, A. J. Ahmadia, M. Berger, D. Calhoun, D. L. George, Y. Hadjimichael, D. I. Ketcheson, G. I. Lemoine, and R. J. LeVeque. Clawpack: building an open source ecosystem for solving hyperbolic PDEs. *PeerJ Computer Science*, 2:e68, 2016.

[48] K. T. Mandli and C. N. Dawson. Adaptive mesh refinement for storm surge. *Ocean Modelling*, 75:36–50, 2014.

[49] O. Meister. *Sierpinski Curves for Parallel Adaptive Mesh Refinement in Finite Element and Finite Volume Methods.* PhD thesis, Technical University of Munich, 2016.

[50] O. Meister and M. Bader. 2D adaptivity for 3D problems: Parallel SPE10 reservoir simulation on dynamically adaptive prism grids. *Journal of Computational Science*, 9:101–106, 2015.

[51] O. Meister, K. Rahnema, and M. Bader. A Software Concept for Cache-Efficient Simulation on Dynamically Adaptive Structured Triangular Grids. In *PARCO*, pages 251–260, 2011.

[52] O. Meister, K. Rahnema, and M. Bader. Parallel Memory-Efficient Adaptive Mesh Refinement on Structured Triangular Meshes with Billions of Grid Cells. *ACM Transactions on Mathematical Software*, 43(3):19, 2016.

[53] W. F. Mitchell. Adaptive refinement for arbitrary finite-element spaces with hierarchical bases. *Journal of computational and applied mathematics*, 36(1):65–78, 1991.

[54] P. J. Mucci, S. Browne, C. Deane, and G. Ho. PAPI: A portable interface to hardware performance counters. In *Proceedings of the department of defense HPCMP users group conference*, volume 710, 1999.

[55] National Centers for Environmental Information. ETOPO2v2 2-minute Worldwide Bathymetry/Topography Grids. `www.ngdc.noaa.gov/mgg/global/etopo2.html`. Accessed in June 2019.

[56] National Geophysical Data Center. 2-minute gridded global relief data (ETOPO2) v2, 2006. National Geophysical Data Center, NOAA.

[57] Y. Okada. Surface deformation due to shear and tensile faults in a half-space. *Bulletin of the seismological society of America*, 75(4):1135–1154, 1985.

[58] OpenMP Committee. OpenMP 4.0 Application Program Interface. `https://www.openmp.org/wp-content/uploads/OpenMP4.0.0.pdf`. Accessed in October 2019.

[59] A. Pinar and C. Aykanat. Fast optimal load balancing algorithms for 1D partitioning. *Journal of Parallel and Distributed Computing*, 64(8):974–996, 2004.

[60] S. Rettenberger, O. Meister, M. Bader, and A.-A. Gabriel. ASAGI – A Parallel Server for Adaptive Geoinformation. In *Proceedings of the Exascale Applications and Software Conference 2016 (EASC '16)*, pages 2:1–2:9. ACM, 2016.

[61] P. L. Roe. Approximate riemann solvers, parameter vectors, and difference schemes. *Journal of computational physics*, 43(2):357–372, 1981.

[62] P. Samfass, J. Klinkenberg, and M. Bader. Hybrid MPI+ OpenMP Reactive Work Stealing in Distributed Memory in the PDE Framework sam(oa)^2. In *2018 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 337–347. IEEE, 2018.

[63] sam(oa)^2 Development Team. sam(oa)^2 – Space-Filling Curves and Adaptive Meshes for Oceanic And Other Applications. `https://gitlab.lrz.de/samoa/samoa`. Accessed in June 2019.

[64] R. S. Sampath, S. S. Adavani, H. Sundar, I. Lashuk, and G. Biros. Dendro: Parallel algorithms for multigrid and AMR methods on 2:1 balanced octrees. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, page 18. IEEE Press, 2008.

[65] W. C. Thacker. Some exact solutions to the nonlinear shallow-water wave equations. *Journal of Fluid Mechanics*, 107:499–508, 1981.

[66] University of Utah. The uintah software. `http://uintah.utah.edu/`. Accessed in June 2019.

[67] K. Unterweger, R. Wittmann, P. Neumann, T. Weinzierl, and H.-J. Bungartz. Integration of FULLSWOF2D and PeanoClaw: Adaptivity and Local Time-stepping for Complex Overland Flows. In *Recent Trends in Computational Engineering – CE2014*, volume 105 of *Lecture Notes in Computational Science and Engineering*, pages 181–195. Springer, 2015.

[68] U.S. Geological Survey. M 8.8 - offshore Bio-Bio, Chile. `https://earthquake.usgs.gov/earthquakes/eventpage/usp000h7rf`. Accessed in June 2019.

[69] W. T. Vetterling, S. A. Teukolsky, W. H. Press, and B. P. Flannery. Numerical recipes, 1989.

[70] C. A. Vigh. *Parallel Simulation of the Shallow Water Equations on Structured Dynamically Adaptive Triangular Grids.* PhD thesis, Technical University of Munich, 2012.

[71] M. Wahib, N. Maruyama, and T. Aoki. Daino: A high-level framework for parallel and efficient amr on gpus. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '16, pages 53:1–53:12. IEEE Press, 2016.

[72] P. Weatherall, K. M. Marks, M. Jakobsson, T. Schmitt, S. Tani, J. E. Arndt, M. Rovere, D. Chayes, V. Ferrini, and R. Wigley. A new digital bathymetric model of the world's oceans. *Earth and Space Science*, 2(8):331–345, 2015.

[73] T. Weinzierl. *A framework for parallel PDE solvers on multiscale adaptive Cartesian grids.* PhD thesis, Technical University of Munich, 2009.

[74] T. Weinzierl, M. Bader, K. Unterweger, and R. Wittmann. Block Fusion on Dynamically Adaptive Spacetree Grids for Shallow Water Waves. *Parallel Processing Letters*, 24(3):1441006, 2014.

[75] T. Wilde, M. Ott, A. Auweter, I. Meijer, P. Ruch, M. Hilger, S. Kühnert, and H. Huber. CooLMUC-2: A supercomputing cluster with heat recovery for adsorption cooling. In *2017 33rd Thermal Measurement, Modeling & Management Symposium (SEMI-THERM)*, pages 115–121. IEEE, 2017.