A Generic Interface for Godunov-type Finite Volume Methods on Adaptive Triangular Meshes

Chaulio R. Ferreira and Michael Bader

Department of Informatics, Technical University of Munich, Germany
chaulio.ferreira@tum.de, bader@in.tum.de

Abstract. We present and evaluate a programming interface for high performance Godunov-type finite volume applications with the framework sam(oa)^2. This interface requires application developers only to provide problem-specific implementations of a set of operators, while sam(oa)^2 transparently manages HPC features such as memory-efficient adaptive mesh refinement, parallelism in distributed and shared memory and vectorization of Riemann solvers. We focus especially on the performance of vectorization, which can be either managed by the framework (with compiler auto-vectorization of the operator calls) or directly by the developers in the operator implementation (possibly using more advanced techniques). We demonstrate the interface's performance using two example applications based on variations of the shallow water equations. Our performance results show successful vectorization using both approaches, with similar performance. They also show that the applications developed with the new interface achieve performance comparable to analogous applications developed without the new layer of abstraction.

Keywords: High performance computing · vectorization · Finite Volume methods · shallow water equations.

1 Introduction

Most modern processors used in scientific computing rely on multiple levels of parallelism to deliver high performance. In this paper, we particularly focus on vectorization (i.e., SIMD parallelism) due to the recent increases in the width of the SIMD instructions provided by high-end processors. For example, latest generations of Intel Xeon architectures provide the AVX-512 instruction set, which can operate simultaneously on 8 double-precision (or 16 single-precision) values. As SIMD parallelism relies on uniform control flows, it benefits from regular data structures and loop-oriented algorithms for effective vectorization. This poses a particular challenge to applications that exploit non-regularity.

Adaptive mesh refinement (AMR) is a key strategy for large-scale applications to efficiently compute accurate solutions, as high resolution can be selectively applied in regions of interest, or where the numerical scheme requires higher accuracy. Managing dynamic mesh refinement and coarsening on parallel platforms, however, requires complex algorithms, which makes minimizing the time-to-solution a non-trivial task. Tree-structured approaches were proven successful to realize cell-wise adaptivity, even for large-scale applications (e.g., [6, 20, 23]). They allow to minimize the invested degrees of freedom, but previous work has shown that SIMD parallelism and meshing overhead push towards combining tree-structure with uniform data structures, such as introducing uniformly refined *patches* on the leaf-level [5, 24]. Block-wise AMR approaches (e.g., [3, 2, 22]) extend refined mesh regions to uniform grid blocks – thus increasing the degrees of freedom, but allowing to stick to regular, array-based data structures. We found, however, that even then, efficient vectorization is not guaranteed, but requires careful implementation and choice of data structures [8].

In this work, we address the area of conflict between dynamic AMR and vectorization, on the development of a general Godunov-type finite volume solver in sam(oa)^2 [17], a simulation framework that provides memory- and cache-efficient traversals on dynamically adaptive triangular meshes and supports parallelism in distributed (using MPI) and shared memory (using OpenMP). We provide an easy-to-use and highly customizable interface that hides from the application developers (here referred to as *users*) all the complexity of the meshing algorithms and simplifies the creation of high-performance solvers for various systems of partial differential equations (PDEs). The use of patches for efficient vectorization is also transparent to the user, which is accomplished by a tailored concept of *operators*. Performance experiments confirm the effectiveness of vectorization for two shallow-water models developed with this interface, which perform comparably to analogous applications developed without the abstraction layer.

2 Numerical background

We consider hyperbolic PDE systems written in the general form

$$q_t + f(q)_x + g(q)_y = \psi(q, x, y), \tag{1}$$

where $q(x, y, t)$ is a vector of unknowns and the *flux functions* $f(q)$ and $g(q)$, as well as the *source term* $\psi(q, x, y)$ are specific for each PDE. In the following, we describe two PDEs that serve to demonstrate the usability and performance of sam(oa)^2. Then, we discuss a numerical approach often used to solve such problems, usually known as *Godunov-type finite volume methods*.

(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. As such, they are widely used for ocean modeling, since the ocean wave lengths are generally very long compared to the ocean depth [2, 13].

The *single-layer shallow water equations* take the form

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

where $h(x, y, t)$ is the fluid depth; $u(x, y, t)$ and $v(x, y, t)$ are the vertically averaged fluid velocities in the x and y directions, respectively; g is the gravitational constant; and $b(x, y)$ is the bottom surface elevation. In oceanic applications, b is usually relative to mean sea level and corresponds to submarine bathymetry where $b < 0$ and to terrain topography where $b > 0$. Here, the source term $\psi(x, y, t) = [0, -ghb_x, -ghb_y]^T$ models the effect of the varying topography, but may also include further terms, such as bottom friction and Coriolis forces.

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 [15]. 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 can provide more realistic representations while keeping the computational costs relatively low. Using the simplest variation of these equations, one can model the ocean with two layers: a shallow top layer over a deeper bottom layer, allowing a more accurate representation of the wind effects. The system of *two-layer shallow water equations* can be written as

$$\begin{bmatrix} h_1 \\ h_1u_1 \\ h_1v_1 \\ h_2 \\ h_2u_2 \\ h_2v_2 \end{bmatrix}_t + \begin{bmatrix} h_1u_1 \\ h_1u_1^2 + \frac{1}{2}gh_1^2 \\ h_1u_1v_1 \\ h_2u_2 \\ h_2u_2 + \frac{1}{2}gh_2^2 + rgh_2h_1 \\ h_2u_2v_2 \end{bmatrix}_x + \begin{bmatrix} h_1v_1 \\ h_1u_1v_1 \\ h_1v_1^2 + \frac{1}{2}gh_1^2 \\ h_2v_2 \\ h_2u_2v_2 \\ h_2v_2^2 + \frac{1}{2}gh_2^2 + rgh_2h_1 \end{bmatrix}_y = \begin{bmatrix} 0 \\ -(h_2)_xgh_1 - gh_1b_x \\ -(h_2)_ygh_1 - gh_1b_y \\ 0 \\ (h_2)_xrgh_1 - gh_2b_x \\ (h_2)_yrgh_1 - gh_2b_y \end{bmatrix}, \quad (3)$$

where h_1, u_1 and v_1 are the quantities in the top layer and h_2, u_2 and v_2 are in the bottom layer; and $r \equiv \rho_1/\rho_2$ is the ratio of the densities of the fluid contained in each layer. The equations can be generalized for an arbitrary number of vertical layers [4]; however, in this work we deal only with the single- and two-layer forms discussed above, using them as example applications for our framework.

Godunov-type Finite Volume Methods. To solve PDE systems as in equations (1), (2) and (3), we adopt Godunov-type finite volume methods, following the numerical approach described in [12, 13], which we slighly modify to match the triangular meshes in sam(oa)^2 (cf. [17] for details). We discretize on tree-structured adaptive triangular meshes, where the solution variables q are averaged within each cell. We represent these variables as Q_i^n, the quantities vector in cell $\mathcal{C}_i$ at time t_n. To update $\mathcal{C}_i$, we first need to approximately solve the so-called *Riemann problem* on all of its edges, i.e., to compute the *numerical fluxes* $\mathcal{F}$ that reflect the quantities being transferred between $\mathcal{C}_i$ and each of its neighbors. We can then use these numerical fluxes to update $\mathcal{C}_i$, according to

$$Q_i^{n+1} = Q_i^n + \frac{\Delta t}{V_i} \sum_{j \in \mathcal{N}(\mathcal{C}_i)} \mathcal{F}(Q_i^n, Q_j^n), \quad (4)$$

where $\mathcal{N}(\mathcal{C}_i)$ is the set of neighbors of $\mathcal{C}_i$; $\mathcal{F}(Q_i^n, Q_j^n)$ is the numerical flux betweens cells $\mathcal{C}_i$ and $\mathcal{C}_j$; V_i is the volume of $\mathcal{C}_i$; and Δt is the time step applied (computed independently for every time step according to the CFL condition).

The *Riemann solver*, i.e., the numerical algorithm used to compute the numerical fluxes, is typically the most important and complex part of the scheme. As further discussed in Section 4, users need to provide a proper Riemann solver for their specific problem. In our example applications, we used solver implementations extracted from GeoClaw [2]: the augmented Riemann solver [10] for the single-layer and the solver proposed in [14] for the two-layer equations.

3 Patch-Based Parallel Adaptive Meshes in Sam(oa)^2

Sam(oa)^2 [17] is a simulation framework that supports the creation of finite element, finite volume and discontinuous Galerkin methods using adaptive meshes generated by recursive subdivision of triangular cells, following the newest-vertex-bisection method [18]. The cells in the resulting tree-structured mesh are organized following the order induced by the Sierpinski space-filling curve – which is obtained by a depth-first traversal of the binary refinement tree. The Sierpinski order is used to store the adaptive mesh linearly without the need to explicitly store the full refinement tree, leading to low memory requirement for mesh storage and management. The order also allows to iterate through all mesh elements in a memory- and cache-efficient way [1]. Sam(oa)^2 is implemented in Fortran and features a hybrid MPI+OpenMP parallelization that is also based on the Sierpinski order induced on the cells, which are divided into contiguous sections of similar size. The sections are independent units that can be assigned to different processes/threads and processed simultaneously with low communication requirements. Previous research [17] has shown that this parallelization is very efficient and scales well on up to 8,000 cores.

To better support vectorization, we modified the mesh structure to store uniformly refined *patches* in the leaves of the refinement tree, instead of single cells [7]. This allows reorganizing the simulation data into temporary arrays that effectively define lists of Riemann problems and to which vectorization is applied, following the approach proposed in [8]. In addition to enabling vectorization, the patch-based approach also leads to further performance gains due to improved memory throughput (because of the new data layout) and to lower overhead for managing the refinement tree (because its size was considerably reduced) [7].

Each patch is obtained by newest-vertex bisection of triangular cells, up to a uniform refinement depth d, resulting in patches with 2^d cells. To guarantee conforming meshes (i.e., without hanging nodes), all patches are generated with identical depth d (whose value is set by the user at compilation time) and we only allow even values for d. Our results in Section 6 show that the best time-to-solution is achieved already for comparably small patches ($d = 6$ or $d = 8$).

Note that compared to [7] we recently changed the refinement strategy within patches: patch refinement now follows the adaptive tree-refinement strategy, which simplifies interpolation schemes for refinement and coarsening.

4 FVM Interface

To simplify the creation of new PDE solvers in sam(oa)^2, we designed a programming interface with which users can easily define implementation details that are particular to each system of PDEs. Using the abstraction layer of the new interface, users work with simple data structures (arrays) and application-specific algorithms, while sam(oa)^2 transparently manages cache-efficient grid traversals, distributed/shared-memory parallelism, adaptive mesh refinement, dynamic load balancing and vectorization. In the following, we will refer to the new programming interface as *FVM interface*, applications developed with it as *FVM applications* and subroutines provided by the users as *FVM operators*.

4.1 Data Structures Used in the FVM Interface

For a new FVM application, users first need to declare how many quantities should be stored for each cell. Following the approach used in GeoClaw [2], these are divided into two types: `Q` quantities are the actual unknowns in the system of equations and are updated at every time step following the numerical scheme, as in Equation (4); `AUX` quantities are other cell-specific values that are required for the simulation, but are either constant in time or updated according to a different scheme. E.g., in our FVM application for the single-layer shallow water equations, `Q=(h,hu,hv)` and `AUX=(b)`.

The user specifies the number of `Q` and `AUX` variables via the preprocessor macros `_FVM_Q_SIZE` and `_FVM_AUX_SIZE`. Using these values, sam(oa)^2 creates 2D arrays `Q` and `AUX` for each patch to store the respective quantities for all patch cells. The arrays are stored similarly to a "structs of arrays" layout, i.e., the cell averages of any given quantity are stored contiguously, which is very important for efficient vectorization. We designed the FVM interface to transparently manage the data in the patches, such that users do not need any knowledge of their geometry or data layout. Instead, users only need to implement simple *FVM operators* that are applied to the mesh on a cell-by-cell or edge-by-edge basis.

4.2 FVM Operators

In the FVM operators, users define how cells should be initialized, refined, coarsened and updated. This includes providing a Riemann solver for the particular problem – where we anticipate that users might want to use existing solvers (e.g., from the GeoClaw package [2]). In the following we briefly describe the interface and what is expected from the most important operators. We note that due to limited space, we omit descriptions of further operators that are mostly concerned with details of the input and output formats used by sam(oa)^2.

`InitializeCell`. With this operator, users define the initial conditions and the initial mesh refinement in a cell-by-cell basis. Given the coordinates of a cell's vertices, this operator returns the initial value of all cell quantities (`Q` and `AUX`), as well as an estimate for the initial wave speed (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.

`ComputeFluxes.` This operator is responsible for solving the PDE-specific Riemann problems at all edges in the mesh and for returning the numerical fluxes at each edge, as well as the speed of the fastest wave created by the discontinuities between the cells. This is usually the most computing-intensive and time-consuming step in the simulations and where optimization efforts achieve the best results. Thus, to give users finer control of the code used at this step, especially with respect to vectorization, we provide two options to implement this operator: a *single-edge* or a *multi-edge* operator, as described in the following.

The **single-edge operator** is applied to one edge at a time, solving a single Riemann problem. In this case, the operator code provided by the users does not need to implement vectorization, because it is completely handled by the framework. Consider the example shown in Fig. 1(a): the framework loops through a list of N Riemann problems extracting the input data of each problem and calling the operator with the respective data, while the operator only needs to extract the quantities from the input arrays and call the Riemann solver with the appropriate parameters. That loop is annotated with an `!$OMP SIMD` directive, and the operator call with a `!DIR$ FORCEINLINE` directive, to inform the compiler that we want the loop to be vectorized and the operator to be inlined and vectorized as well. In the operator call we use Fortran's subarrays, which introduce an overhead for their creation, but still support vectorization by the Intel Fortran Compiler. Note that whether the loop actually gets vectorized depends on the operator implementation, because it may contain complex operations or library calls that can inhibit auto-vectorization by the compiler.

In a **multi-edge operator** users can provide their own optimized code for the main loop. Instead of dealing with a single Riemann problem, the multi-edge operator takes a list of problems as input. It gives users complete control of the loop that iterates through the list, as well as of the compiler directives for vectorization. For illustration, see Fig. 1(b). The implementation of a multi-edge operator can become considerably more complex, allowing advanced users to exploit any technique for improved vectorization (intrinsic functions, etc.).

We created successfully vectorized FVM applications using single- and multi-edge operators both for the single- and two-layer shallow water equations. Section 6 will present and compare the performance achieved by each approach.

`UpdateCell.` This operator defines how the `Q` quantities of each cell should be updated with the Riemann solutions. After obtaining the numerical fluxes at each edge in the mesh, sam(oa)^2 computes the sum of all fluxes leaving/entering each cell through its three edges. The total flux is then used to update the cell quantities, usually following Equation (4); however, users 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 with the shallow water equations). The operator also returns a refinement flag for the cell, to inform whether it should be refined (1), kept (0) or coarsened (-1) for the next time step.

```fortran
! OPERATOR CALL (FRAMEWORK CODE, NOT TOUCHED BY THE USER):
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 computeFluxesSingle(normals(i,:),qL(i,:),qR(i,:),auxL(i,:),
        ↪ auxR(i,:),fluxL(i,:),fluxR(i,:),waveSpeed)
    maxWaveSpeed = max(maxWaveSpeed, waveSpeed)
end do
```

```fortran
! OPERATOR CODE, IMPLEMENTED BY THE USER:
subroutine computeFluxesSingle(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
  real :: hL,huL,hvL,bL, hR,huR,hvR,bR ! local variables

  !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)

  !The Riemann solver fills the output (fluxL, fluxR and waveSpeed):
  !DIR$ FORCEINLINE
  call RiemannSolver(normal,hL,huL,...,hvR,bR,fluxL,fluxR,waveSpeed)
end subroutine
```

(a) Single-edge version of the `ComputeFluxes` operator.

```fortran
! OPERATOR CALL (FRAMEWORK CODE, NOT TOUCHED BY THE USER):
! ... (Declaration of qL, qR, auxL, auxR and normals, exactly as above)

call computeFluxesMulti(normals,qL,qR,auxL,auxR,fluxL,fluxR,maxWaveSpeed)
```

```fortran
! OPERATOR CODE, IMPLEMENTED BY THE USER:
subroutine computeFluxesMulti(normals,qL,qR,auxL,auxR,fluxL,fluxR,
    ↪ maxWaveSpeed)
  real, dim(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
  real :: hL,huL,hvL,bL, hR,huR,hvR,bR, normal(2) ! local variables

  !$OMP SIMD REDUCTION(max: maxWaveSpeed) PRIVATE(normal,hL,huL,...,hvR,bR)
  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,...,hvR,bR,fluxL,fluxR,waveSpeed)
    maxWaveSpeed = max(maxWaveSpeed, waveSpeed)
  end do
end subroutine
```

(b) Multi-edge version of the `ComputeFluxes` operator.

Fig. 1: Example implementations of the single-edge (a) and multi-edge (b) versions of the `ComputeFluxes` operator for the single-layer shallow water equations, as well as the framework codes that call them. In both we assume the existence of a subroutine called `RiemannSolver` that solves a Riemann problem.

`SplitCell` and `MergeCells`. These two operators control how adaptivity is performed on the cells. `SplitCell` takes the data from a cell $\mathcal{C}^{in}$ as input, and outputs data for two finer cells $\mathcal{C}_1^{out}$ and $\mathcal{C}_2^{out}$, that result from splitting $\mathcal{C}^{in}$. `MergeCells` 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. These operations are often performed by simply copying or interpolating cell quantities, but again the users can customize the operator and handle special cases.

5 FVM Applications and Test Scenarios

We used the FVM interface to create two FVM applications that simulate tsunami wave propagation with the single- and two-layer shallow water equations discussed in Section 2. Here, we refer to them as `FVM-SWE` and `FVM-SWE2L`. They are based on two applications that already existed within sam(oa)^2 for those same systems of PDEs and were implemented directly into the framework's core (i.e., without the FVM interface). To distinguish them from the new FVM applications, we will refer to the older applications 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`.

Before presenting our experimental results, we first give more details about the simulation scenarios we used in our experiments in the following.

Tohoku Tsunami 2011. For `SWE` and `FVM-SWE`, we simulated a real tsunami event that took place near the coast of Tohoku, Japan, in 2011 – see Fig. 2. We used bathymetry data from the Northern Pacific and the Sea of Japan (GEBCO_08 Grid, version 20100927) and initial bathymetry and water displacements obtained from a simulation of the Tohoku earthquake [9]. All geographical input data was handled by the parallel I/O library ASAGI [19].

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 2852 m^2 in the area around the tsunami, while regions farther away can have cells with up to 5981 km^2. 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 `SWE2L` and `FVM-SWE2L`, we simulated waves generated by a circular hump of water propagating over a parabolic bowl-shaped bathymetry – see Fig. 3, where we show visualizations of this scenario at $t = 0$. This setup is based on an example from the GeoClaw package [2] and serves as a benchmark for the quality of numerical schemes regarding wetting/drying of

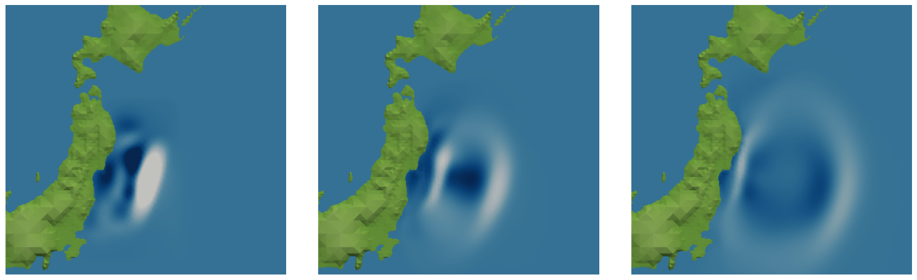

Fig. 2: Simulation of the tsunami in Tohoku, Japan, 2011. The pictures show the tsunami wave 10, 20 and 30 minutes after the earthquake, respectively.

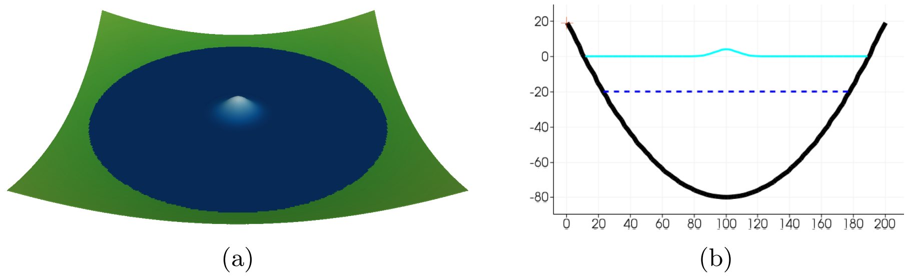

Fig. 3: Parabolic bowl-shaped lake at $t = 0$ with 3D (a) and cross-cut (b) visualizations. In (b), the thick (black) line depicts the bathymetry, while the dashed (blue) and solid (cyan) lines represent the two layers of water.

layers. We used cells with sizes ranging from 2^{-27} to 2^{-13} of the computational domain, resulting in mesh sizes from around 0.8 million to 7.2 million cells. The measured times also consider 1000 time steps of the regular time-stepping phase.

6 Performance Results

We conducted experiments on the CoolMUC2 and CoolMUC3 cluster systems hosted at the Leibniz Supercomputing Center (LRZ). CoolMUC2 contains nodes with dual-socket Haswell systems and CoolMUC3 provides nodes with Xeon Phi "Knights Landing" (KNL) processors. An overview of the system configuration of the nodes in each system is presented in Table 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 expected to be more noticeable on the KNL nodes. Table 1 lists the theoretical peak bandwidth and peak Flop/s throughput of each machine, along with measurements obtained with the STREAM benchmark [16] and with the Flop/s benchmark proposed in Chapter 2 of [11]. We use these values as estimates for the maximum performance that can be achieved in practice on those machines.

Table 1: Specifications of the nodes in the experimental platforms.

System overview	**2×Haswell**	**Knights Landing**
Architecture	Intel® Xeon®	Intel® Xeon Phi™
Model	E5-2697v3	7210F
Cores	2x14	64 (max. 256 threads)
Clock rate	2.60 GHz	1.30 GHz
SIMD vector width	256-bit	512-bit
Memory	64 GB	96 GB + 16 GB MCDRAM
Peak bandwidth	136 GB/s	102 GB/s
Measured bandwidth	106 GB/s	83 GB/s
Peak throughput (double)	582 GFlop/s	2 662 GFlops/s
Measured throughput (double)	156 GFlop/s	720 GFlops/s

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 MPI), but we use OpenMP threading on all available cores, i.e., 28 on the Haswells and 64 on the KNL. On the KNL we also experimented with different number of threads per core (from 1 to 4 with hyperthreading). However, here we only list the results with 2 threads per core (i.e., 128 in total), because this configuration achieved the best performance in most experiments. Also, we use the KNL in cache mode, i.e., the MCDRAM memory is used as an L3 cache.

Simulation Performance. We start by evaluating the performance of `FVM-SWE` and `FVM-SWE2L` with different vectorization strategies and patches with various sizes – see Fig. 4. There is a clear pattern of larger cells delivering higher performance even with vectorization turned off, which can be attributed to the improved memory throughput and the reduction of the adaptivity tree achieved by the patch-based discretizations, as mentioned in Section 3.

When vectorization is used ("Single-edge" and "Multi-edge"), we can observe speedups by factors of 1.1–1.6 on the Haswells and of 1.5–2.3 on the KNL, compared to the non-vectorized versions. Vectorization is clearly more effective for the single-layer applications and (as expected) on KNL processors, which agrees with recent results in literature [8]. The codes developed using single-edge operators perform only slightly slower than the multi-edge ones (up to 2% and 5% slower on each machine), despite of the overhead introduced due to the use of Fortran subarrays in the operator calls. 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.

The vectorized `FVM-SWE` and `FVM-SWE2L` implementations achieve performance very similar to their analogous applications (`SWE` and `SWE2L`) on Haswells (up to 6% slower), while on the KNL there is a noticeable difference in performance (up to 14% slower). Nevertheless, these experiments show that FVM applications can achieve performance comparable to other applications developed directly within the complex framework code, despite of the additional memory operations performed to create the interface's layer of abstraction.

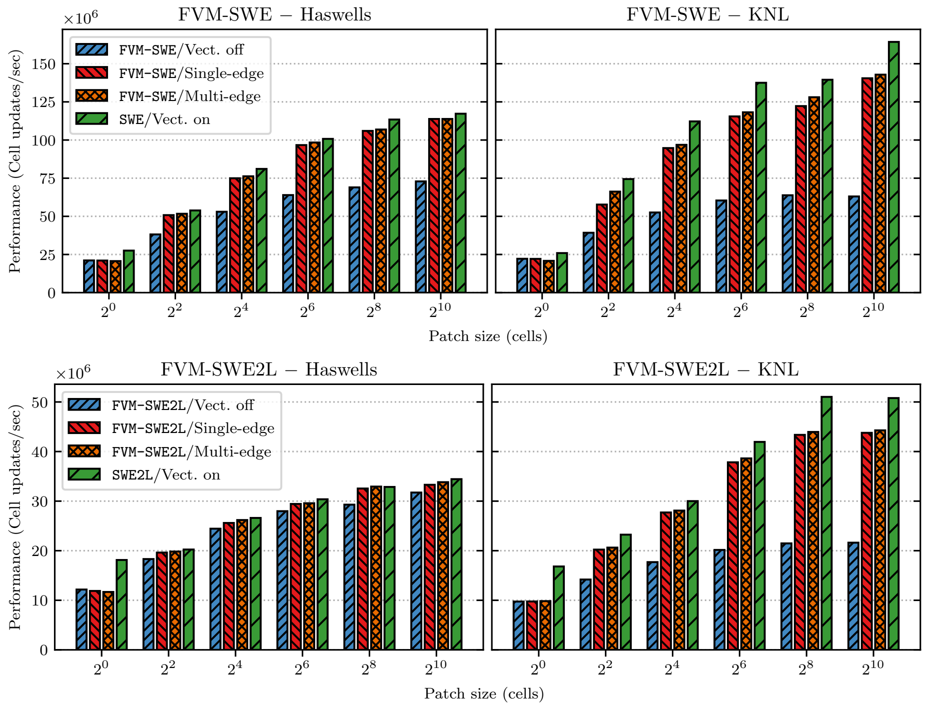

Fig. 4: Performance of the FVM applications and their analogous applications.

Time-to-Solution. Although larger patches deliver higher throughput, they also lead to meshes with more cells (due to more coarse-grained adaptivity). As such, a trade-off in the patch size must be found to minimize the time-to-solution. In Fig. 5 we compare the wall time of the "Multi-edge" implementations. These results show that patches with 64–256 cells tend to minimize the time-to-solution on both machines and both applications – despite of the higher throughput, larger patches are disadvantageous, due to the considerably increased mesh size. We also point out that by combining patch-based AMR with vectorization we have been able to reduce the total execution time by factors of 2.7–4.2, compared to cell-wise adaptivity ("trivial" patches with only one cell).

Component Analysis. In Fig. 5 we split the execution time into two components: the "Numerical" component comprises all routines performed for the numerical time step, i.e., updating the cell quantities; and "Adaptivity" is responsible for handling the remeshing performed after every time step, i.e., refining/coarsening of cells, guaranteeing mesh conformity and updating communication structures. We can observe that, while most of the time reduction happens in the numerical component, the adaptivity component also benefits considerably, because of the reduced complexity and size of the mesh refinement tree.

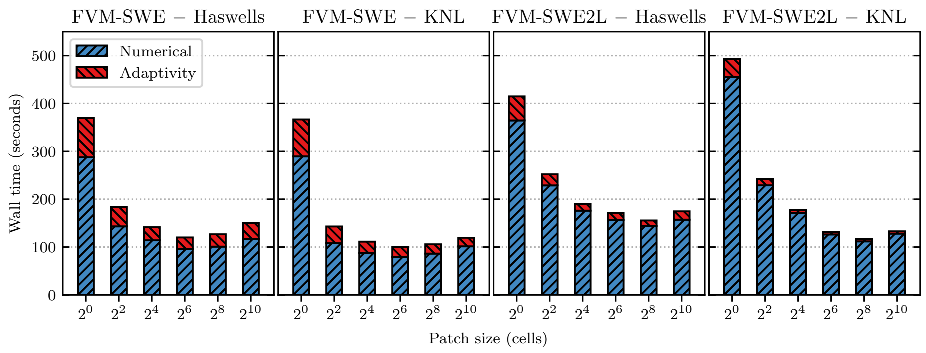

Fig. 5: Execution times of the FVM applications, split into components.

Table 2: Performance of the numerical routines of the FVM applications.

Application	Architecture	Data throughput	Flop/s throughput
`FVM-SWE`	Haswells	14.1 GB/s (13%)	66.0 GFlop/s (42%)
	KNL	20.9 GB/s (25%)	97.4 GFlop/s (14%)
`FVM-SWE2L`	Haswells	7.9 GB/s (7%)	36.3 GFlop/s (23%)
	KNL	9.5 GB/s (11%)	44.0 GFlop/s (6%)

Solver Performance. Now we evaluate the numerical routines alone, without the mesh management algorithms. We computed the data and Flop/s throughputs of the fastest run of each FVM application on each machine – see Table 2, where we also show their percentages relative to the measured peak performance of each machine. The Flop/s were measured using the library PAPI [21], and the data throughputs calculated assuming two accesses (read & write) to `Q` quantities and only one read access to `AUX` quantities for each cell updated.

Considering that the Riemann solver is much more complex than the benchmark used, the results show great utilization of the Haswell processors by the single-layer solver (42%), indicating compute-bound behavior. On the KNL the solver is also compute-bound, although it reaches only 14% of the peak throughput. We point out that the entire simulation data fits in the MCDRAM memory that is being used in cache mode, thus memory bandwidth is not an issue.

A similar analysis for the two-layer solver indicates fairly low performance, compared to the measured peak of each machine, both for Flop/s and data throughput. This happens because the two-layer solver is considerably more complex than the single-layer one, mainly due to several if-then-else branches necessary to handle special cases (such as dry cells/layers). The compiler converts these into masked operations, which causes vectorization overhead. This reveals that it may be possible to modify this solver's implementation to make it more efficient and more suitable for vectorization. However, that is beyond the scope of this paper and is left as a suggestion for future work.

7 Conclusions

We described and evaluated a programming interface that supports the creation of Godunov-type finite volume methods in sam(oa)^2. Thanks to its simple abstraction layer, users with no HPC expertise can create high performance applications with multiple levels of parallelism, only needing to provide problem-specific algorithms. Experienced users also have the option of assuming complete control of the main solver loop, such that they can manage its vectorization and/or attempt further optimizations on it.

The interface allows easy customization of our efficient finite volume solver to different systems of PDEs. Assuming that a Riemann solver for the specific problem is available, the user's work is reduced mainly to handling the solver calls and providing application-specific operators for initialization, refinement and coarsening of cells, which in most cases consist of trivial implementations. In particular, we developed two applications in which we could directly apply Riemann solver implementations from the package GeoClaw.

The underlying framework implements patch-based adaptive mesh refinement, which enables vectorization and at the same time reduces the simulation's computational costs considerably by applying relatively small patches. Our experiments revealed successful vectorization (both when it was managed by the framework and by the users), leading to substantial speedups. The performance results also showed that these applications achieve performance comparable to analogous applications developed directly into the complex source code of sam(oa)^2, with only low to moderate overhead (2–14% slower).

Acknowledgments

Chaulio R. Ferreira appreciates the support of CNPq, the Brazilian Council of Technological and Scientific Development (grant no. 234439/2014-9). Computing resources were provided by the Leibniz Supercomputing Center (project *pr85ri*).

References

1. Bader, M., Böck, C., Schwaiger, J., Vigh, C.A.: Dynamically Adaptive Simulations with Minimal Memory Requirement – Solving the Shallow Water Equations Using Sierpinski Curves. SIAM J. Sci. Comput. **32**(1), 212–228 (2010)
2. Berger, M.J., George, D.L., LeVeque, R.J., Mandli, K.T.: The GeoClaw software for depth-averaged flows with adaptive refinement. Adv. in Water Res. **34**(9), 1195–1206 (2011)
3. Berger, M.J., Oliger, J.: Adaptive mesh refinement for hyperbolic partial differential equations. J. Comp. Phys. **53**(3), 484–512 (1984)
4. Bouchut, F., Zeitlin, V., et al.: A robust well-balanced scheme for multi-layer shallow water equations. Discrete Contin. Dyn. Syst. Ser. B **13**(4), 739–758 (2010)
5. Burstedde, C., Calhoun, D., Mandli, K., Terrel, A.R.: ForestClaw: Hybrid forest-of-octrees AMR for hyperbolic conservation laws. In: Parallel Comput.: Accelerating Comput. Sci. and Eng. vol. 25, pp. 253–262 (2014)

6. Burstedde, C., Wilcox, L.C., Ghattas, O.: `p4est`: Scalable Algorithms for Parallel Adaptive Mesh Refinement on Forests of Octrees. SIAM J. Scient. Comput. **33**(3), 1103–1133 (2011)
7. Ferreira, C.R., Bader, M.: Load balancing and patch-based parallel adaptive mesh refinement for tsunami simulation on heterogeneous platforms using Xeon Phi coprocessors. In: Proc. PASC. pp. 12:1–12:12. ACM (2017)
8. Ferreira, C.R., Mandli, K.T., Bader, M.: Vectorization of Riemann solvers for the single-and multi-layer Shallow Water Equations. In: Int. Conf. High Perf. Comput. Simul. pp. 415–422. IEEE (2018)
9. Galvez, P., Ampuero, J.P., Dalguer, L.A., Somala, S.N., Nissen-Meyer, T.: Dynamic earthquake rupture modelled with an unstructured 3-D spectral element method applied to the 2011 M9 Tohoku earthquake. Geophys. J. Int. **198**(2), 1222–1240 (2014)
10. George, D.L.: Augmented Riemann solvers for the shallow water equations over variable topography with steady states and inundation. J. Comput. Phys. **227**(6), 3089–3113 (2008)
11. Jeffers, J., Reinders, J.: Intel Xeon Phi coprocessor high-performance programming. Newnes (2013)
12. LeVeque, R.J.: Finite Volume Methods for Hyperbolic Problems. Cambridge University Press (2002), www.clawpack.org/book.html
13. LeVeque, R.J., George, D.L., Berger, M.J.: Tsunami modelling with adaptively refined finite volume methods. Acta Numerica **20**, 211–289 (2011)
14. Mandli, K.T.: A Numerical Method for the Two Layer Shallow Water Equations with Dry States. Ocean Modelling **72**, 80–91 (2013)
15. Mandli, K.T., Dawson, C.N.: Adaptive mesh refinement for storm surge. Ocean Modelling **75**, 36–50 (2014)
16. McCalpin, J.D.: Memory bandwidth and machine balance in current high performance computers. IEEE TCCA Newsletter pp. 19–25 (1995)
17. Meister, O., Rahnema, K., Bader, M.: Parallel Memory-Efficient Adaptive Mesh Refinement on Structured Triangular Meshes with Billions of Grid Cells. ACM Trans. Math. Softw. **43**(3), 19 (2016)
18. Mitchell, W.F.: Adaptive refinement for arbitrary finite-element spaces with hierarchical bases. J. Comput. Appl. Math. **36**(1), 65–78 (1991)
19. Rettenberger, S., Meister, O., Bader, M., Gabriel, A.A.: ASAGI – A Parallel Server for Adaptive Geoinformation. In: Proc. EASC. pp. 2:1–2:9. ACM (2016)
20. Sampath, R.S., Adavani, S.S., Sundar, H., Lashuk, I., Biros, G.: Dendro: Parallel algorithms for multigrid and AMR methods on 2:1 balanced octrees. In: Proc. 2008 ACM/IEEE Conf. on Supercomp. pp. 18:1–18:12. IEEE Press (2008)
21. Terpstra, D., Jagode, H., You, H., Dongarra, J.: Collecting performance data with PAPI-C. In: Tools for High Performance Computing. pp. 157–173. Springer (2010)
22. Wahib, M., Maruyama, N., Aoki, T.: Daino: A high-level framework for parallel and efficient AMR on GPUs. In: Proc. Int. Conf. HPC, Networking, Storage and Analysis. pp. 53:1–53:12. IEEE Press (2016)
23. Weinzierl, T.: The Peano software – parallel, automaton-based, dynamically adaptive grid traversals. ACM Trans. Math. Softw. (2019)
24. Weinzierl, T., Bader, M., Unterweger, K., Wittmann, R.: Block Fusion on Dynamically Adaptive Spacetree Grids for Shallow Water Waves. Parallel Processing Letters **24**(3), 1441006 (2014)