Load Balancing and Patch-Based Parallel Adaptive Mesh Refinement for Tsunami Simulation on Heterogeneous Platforms Using Xeon Phi Coprocessors

Chaulio R. Ferreira

Technical University of Munich
Department of Informatics, Boltzmannstr. 3
85748 Garching, Germany
chaulio.ferreira@tum.de

Michael Bader

Technical University of Munich
Department of Informatics, Boltzmannstr. 3
85748 Garching, Germany
bader@in.tum.de

ABSTRACT

We present a patch-based approach for tsunami simulation with parallel adaptive mesh refinement on the Salomon supercomputer. The special architecture of Salomon, with two Intel Xeon CPUs (Haswell architecture) and two Intel Xeon Phi coprocessors (Knights Corner) per compute node, suggests truly heterogeneous load balancing instead of offload approaches, because host and accelerator achieve comparable performance for our simulations.

We use a tree-structured mesh refinement strategy resulting from newest-vertex bisection of triangular grid cells, but introduce small uniform grid patches into the leaves of the tree to allow vectorisation of the Finite Volume solver over grid cells. In particular, we implemented vectorised versions of the approximate Riemann solvers, exploiting Fortran's array notations where possible. While large patches increase computational performance due to vectorisation, improved memory access and reduced meshing overhead, they also increase the overall number of processed cells. Thus, a trade-off must be found regarding the patch size. We experimented with different patch sizes in a study of the time-to-solution of a simulation of the 2011 Tohoku tsunami, and found that relatively small patches with 8^2 cells resulted in the smallest execution times.

We use the Xeon Phis in *symmetric mode* and apply heterogeneous load balancing between hosts and coprocessors, identifying the relative load distribution either from on-the-fly runtime measurements or from a priori exhaustive testing. Both approaches perform better than homogeneous load balancing and better than using only the CPUs or only the Xeon Phi coprocessors in *native mode*. In all set-ups, however, the absolute speedups are impeded by the slow MPI communication between Xeon Phi coprocessors.

CCS CONCEPTS

• **Mathematics of computing** → **Mathematical software performance**; *Partial differential equations*; • **Computing methodologies** → **Massively parallel and high-performance simulations**; *Vector / streaming algorithms*; • **Applied computing** → *Earth and atmospheric sciences*;

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

PASC '17, June 26-28, 2017, Lugano, Switzerland

© 2017 Copyright held by the owner/author(s).

ACM ISBN 978-1-4503-5062-4/17/06...\$15.00

<https://doi.org/10.1145/3093172.3093237>

KEYWORDS

Parallel adaptive mesh refinement; load balancing on heterogeneous systems; Xeon Phi coprocessor; patch-based adaptivity; vectorisation; tsunami simulation

ACM Reference format:

Chaulio R. Ferreira and Michael Bader. 2017. Load Balancing and Patch-Based Parallel Adaptive Mesh Refinement for Tsunami Simulation on Heterogeneous Platforms Using Xeon Phi Coprocessors. In *Proceedings of PASC '17, Lugano, Switzerland, June 26-28, 2017*, 12 pages. <https://doi.org/10.1145/3093172.3093237>

1 INTRODUCTION

Processor technology for high performance computing (HPC) is forced towards offering parallelism on all levels. Arithmetic units offer SIMD instructions working on short vectors, such that algorithms need to support vectorisation, preferably realised via auto-vectorisation by the compiler or else via explicit vectorisation by the programmer. On general-purpose graphics processing units (GPGPUs), light-weight threads are bundled to execute the same instructions (SIMT – Single Instruction, Multiple Thread), and programmers need to consider thread divergence as a possible performance threat. Compute nodes of supercomputers start to combine different kinds of processors, including standard CPUs but also accelerator-type devices such as GPGPUs or Intel's Xeon Phi coprocessor. Hence, independent of the programming model, HPC applications need to provide efficient load balancing between heterogeneous devices, to fully exploit the available computing capacity.

In this paper, we focus on the impact of vectorisation and heterogeneous load balancing on sam(oa)² (Space-filling curves and Adaptive Meshes for Oceanic And Other Applications) [25], a framework that implements parallel adaptive mesh refinement (pAMR) for tree-structured adaptive triangular meshes. As HPC platform we focus on the Salomon supercomputer, which combines four devices per node – two regular CPUs (Intel Haswell architecture) plus two coprocessors (Intel Xeon Phi, “Knights Corner” generation). Detailed specifications of the architectures are given in Table 1. We use the coprocessors in *symmetric mode*: in our MPI+OpenMP parallelisation, we place one MPI rank on each device and treat the Xeon Phis in the same way (“symmetric”) as the host CPUs. We find this mode particularly attractive on Salomon, as compared to *offload mode* (offloading the most time-consuming kernels to the coprocessor), the symmetric mode promises to work “out of the box” and to maximize computational power, provided that the

Table 1: Specifications of the Salomon nodes with a dual-socket Haswell system and two Xeon Phi coprocessors. We list the configurations of the combined dual-socket Haswell system and of a single Xeon Phi coprocessor. A full specification of Salomon is available on <https://docs.it4i.cz/>.

System overview	Host	Xeon Phi
Architecture	Intel® Xeon®	Intel® Xeon Phi™
Model	E5-2680v3	7120P (KNC)
Cores	24	61
Clock rate	2.50 GHz	1.24 GHz
SIMD vector length	256-bit	512-bit
Peak perform. (DP)	0.48 TFlops/s	1.21 TFlops/s
Peak bandwidth	136 GB/s	352 GB/s
Memory	128 GB	16 GB

MPI application solves the heterogeneous load balancing problem, i.e., considers the differences in the performance achieved by the different devices.

sam(oa)² offers efficient dynamical grid refinement and coarsening with frequent load balancing (after each time step, if necessary). For heterogeneous load balancing, in this paper we evaluate two strategies – the first one is based on manual weighting of the relative performance of each device, while the second one uses on-the-fly runtime measurements to automatically determine these weights. As an example scenario, we simulate tsunami wave propagation based on a Finite Volume discretisation of the Shallow Water Equations – a scenario that profits from dynamically adaptive refinement and coarsening of the mesh after each (explicit) time step. We show that both heterogeneous load balancing strategies are able to achieve higher performance compared to a naive “homogeneous” load balancing that is agnostic of the performance differences. However, we also found that the achieved speedups are impeded by the slow MPI communication between Xeon Phi coprocessors.

To implement solvers for partial differential equations (PDEs) of Finite-Volume- or Finite-Element-type, sam(oa)² provides a strongly element-oriented design [25], where numerical methods are organised into kernels that implement element-local updates (consider the element-local evaluation of residuals in a Finite Element method) or edge-based computations (consider approximate Riemann solvers to compute the edge-local flux between two adjacent elements). While this concept efficiently supports fine-grained cell-wise adaptivity, it is an obstacle to vectorisation: element- and edge-based kernels are intertwined during a grid traversal, which allows to minimise the number of traversals of the entire grid (and thus minimise data transfer); however, the resulting complicated execution scheme does not allow vectorisation over successive elements. In this paper, we therefore examine a more coarse-grained, patch-based adaptivity, i.e., using patches of regularly refined cells as the finest element in the tree-structured grids. We show that patches improve memory throughput and support vectorisation in the numerical computations. The new mesh organisation also reduces the effective size of the grid, so that the traversals for performing cell refinement/coarsening and for guaranteeing grid conformity are considerably faster. On the other hand, coarse-grained adaptivity

increases the overall number of cells in dynamically adaptive simulations, such that a “sweet spot” for the patch size needs to be found in order to minimise the simulation time. In our experiments, we found that using around 8^2 cells per patch delivers the best results – this patch size reduced the time-to-solution by a factor of 2.8 on a single Xeon Phi coprocessor in native mode.

The remainder of this paper is organised as follows. We discuss related work in Section 2 and recapitulate our numerical approach and our pAMR framework in Section 3. In Section 4, we describe our new implementation that introduces regular patches and vectorisation into sam(oa)². Dynamic load balancing on heterogeneous systems is then discussed in Section 5. In Section 6 we present performance results using Xeon Phi coprocessors in both native and symmetric modes, and we conclude by summarizing our main findings in Section 7.

2 RELATED WORK

Tsunami Simulation and Mesh Adaptivity. Adaptive mesh refinement (AMR) is essential for simulations that require high resolution only at specific regions, while using this resolution throughout the entire domain would be prohibitively expensive. Tsunami simulations are a typical example: the simulation domain extends over thousands of kilometers, but resolutions down to the scale of meters might be necessary, especially at the shore. Dynamically adaptive meshes allow to handle these multiscale resolutions and to lower the simulation’s computational cost, while maintaining its accuracy [1, 16]. LeVeque et al. [21] show that even refinement up to wall structures within harbours is feasible. Popinet [33] demonstrates that adaptivity even lowers the overall complexity of simulations, as the number of cells only grows as $\Delta^{-1.4}$ with the mesh size Δ . Benefits of adaptivity were also shown for the case of storm surges [22] and overland flooding [40], which may also be modelled via the shallow water equations.

Adaptive Mesh Refinement vs. Regular Patches. The implementation of mesh adaptivity in contrast to the wish for regular data structures (and hopefully more efficient loop-based algorithms) is a classical design conflict for PDE solvers. Patch-based approaches often follow the Berger-Oliger-Collela approaches [6, 8], which use overlapping regular grid patches in a constructive bottom-up fashion – see the survey by Dubey et al. [12] on respective frameworks. Similarly, top-down approaches that combine octree-type grids with regularly refined patches as leaves of the tree have become attractive. Examples include the packages ForestClaw [9] and Peano [40], for which performance improvements via patches are especially discussed in [43]. The framework Daino offers octree AMR with patches on heterogeneous GPU platforms [42]. Performance advantages due to introducing regular data structures have also been reported for “extruded” [5] or “2.5D” [24] meshes, where 2D-adaptive grids are extended to 3D meshes via uniform refinement in the third dimension.

Load Balancing for pAMR. Our load balancing approach is based on space-filling curves – which has shown to be a highly successful approach for PDE solvers in the HPC domain (e.g., [10, 30, 35]). For triangular meshes generated via newest-vertex-bisection [28] as the ones used in sam(oa)², the use of Sierpinski

curves is a natural fit, due to the matching construction principles. Various approaches to load balancing have been explored, based on the mapping property of the curve [4], on the underlying tree structure [29], or on defining clusters for load balancing [25, 37]. As space-filling curves define a sequential order on grid cells, they turn the load balancing problem into a chains-on-chains partitioning problem [31].

Heterogeneous Load Balancing and Symmetric Mode for Xeon Phi. On heterogeneous platforms, not only the solution of the chains-on-chains partitioning problem becomes more complicated [32]. In many situations hardware properties discourage truly heterogeneous load balancing approaches, as homogeneous, accelerator-only approaches achieve almost the same performance. Both for GPGPU and for Xeon Phi platforms, the respective offload programming models often reduce the load balancing problem to ensuring that the slower host does not become a bottleneck, as the compute-intensive parts of the calculation are executed on the accelerator. Achieving benefits from heterogeneous load balancing (static or dynamic) requires dedicated load balancing strategies, as discussed in [38].

For the Xeon Phi architecture, an additional roadblock is set by communication issues that can severely slow down communication when the *symmetric mode* is used, requiring a special MPI proxy that re-routes communication through the host [34, 41]. Hence, there are only a few software packages that successfully exploit the symmetric mode of Xeon Phi – examples being [17, 18]. The Uintah framework is a notable example for truly heterogeneous load balancing for dynamical mesh refinement on GPU [19] and Xeon Phi supercomputers [26, 27] (the latter using homogeneous load balancing). Sundar and Ghattas [39] propose an approach where a so-called enclave partitioning ensures that communication between compute nodes only affects the host CPUs. They apply a hierarchical partitioning, where partitions are first created for nodes, and are then subpartitioned such that accelerator partitions are entirely enclosed within the host partition, so accelerators only communicate with their host. They demonstrate compute-bound performance and excellent scalability for a high order method, which makes it easier to hide slow communication, compared to lower order methods. They also use static adaptive meshes, which allow to spend more effort on partitioning than dynamically adaptive meshes.

3 PARALLEL TSUNAMI SIMULATION ON TREE-STRUCTURED ADAPTIVE TRIANGULAR MESHES

For tsunami modeling, we adopt an approach by LeVeque et al. [7, 21], which we slightly modify to match the use of triangular meshes in sam(oa)². Tsunami wave propagation is modelled via the 2D Shallow Water Equations (SWEs), a set of hyperbolic PDEs, given in the following form:

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

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 relative to mean sea level – corresponding to submarine bathymetry where $b < 0$ and to terrain topography where $b > 0$. The source term $S(x, y, t) = [0, -ghb_x, -ghb_y]^T$ models the effect of the varying surface depth.

Our numerical scheme closely follows the Finite Volume approach proposed in [21], but is modified to work on triangular meshes [25]. Each element stores cell-averaged values for h , hu , hv and b , respectively. Here, we represent this set of values as Q_i^n , the quantities in cell i at time t_n . For every edge lying between cells i and j , we solve the corresponding Riemann problem approximately, computing the *numerical fluxes* $\mathcal{F}^{in}(Q_i^n, Q_j^n)$, which reflect the mass and momentum transferred between elements within one time step. We then use the numerical fluxes to update the cells according to

$$Q_i^{n+1} = Q_i^n + \frac{\Delta t}{V_i} \sum_{j \in \mathcal{N}(i)} \mathcal{F}^{in}(Q_i^n, Q_j^n), \quad (2)$$

where $\mathcal{N}(i)$ is the set of neighbours of cell i , V_i is the volume of i and Δt is the length of the time step applied (computed independently for every time step according to the CFL condition [11]).

For the computation of the numerical fluxes, sam(oa)² provides implementations of different approximate Riemann solvers, such as the F-Wave [3], HLLE [13] and Augmented Riemann [15] solvers. In particular, it can directly apply the respective Riemann solvers provided with the GeoClaw package [7]. For this work, we implemented vectorised versions of these solvers, which exploit (via auto-vectorisation by the compiler) the SIMD instructions provided in the instruction set of the Intel Xeon Phi coprocessor (Knights Corner). The details for these implementations are described in Section 4.

Tree-Structured Triangular Meshes. We discretise the problem on adaptive meshes generated by recursive subdivision of triangular cells, following the newest-vertex-bisection method [28]. This method for splitting the cells results in a tree-structured mesh where the elements can be inherently organised following the order induced by the Sierpinski Space Filling Curve – which can be obtained by a depth-first traversal of the binary refinement tree. The Sierpinski order can be used to store the adaptive mesh linearly without the need to explicitly store the full refinement tree, leading to low memory requirement for grid storage and management. The order also allows to iterate through all elements in the mesh in a memory- and cache-efficient way [2]. During the grid traversal, the algorithm determines which edges are visited for the first or last time, or which edges are reused, and exploits this information for efficient stream-oriented data access. The complicated grid traversal algorithm is hidden from application developers via a system of element-oriented kernels [25].

Parallel Adaptive Mesh Refinement. sam(oa)² features a hybrid MPI+OpenMP parallelisation that is based on the Sierpinski order induced on the grid elements. The grid elements are divided into sections of similar size containing elements that are contiguous

in the Sierpinski order. These sections are implemented as independent units that can be assigned to different threads or processes and processed simultaneously with low communication requirements. Recent experiments [24, 25] have shown that this parallelisation strategy is very efficient and scales well on up to 8,000 cores.

Although the grid traversal algorithms can be parallelised, they cannot be easily vectorised, because of the irregular data access patterns induced by using cell-wise adaptivity. For effective vectorisation, it is necessary to modify the mesh structure by embedding regularly refined patches into the leaves of the refinement tree – see the detailed discussion of our patches implementation in Section 4. As demonstrated in Section 6, introducing patches also leads to improved performance by allowing more regular memory access patterns and by reducing the overhead for mesh management.

Dynamic Load Balancing. Because the number of cells in each section can increase or decrease due to dynamic adaptivity, care must be taken to guarantee proper load balance among the processes. Therefore, $\text{sam}(\text{oa})^2$ performs dynamic load balancing by redefining the sections and exchanging some of them between processes when necessary. Originally, our load balancing algorithm distributed the computational load homogeneously to all processes, but for this work we extended it to allow heterogeneous distributions, such that the relative speed of different processors could be considered. The model and algorithms used for partitioning the computational load are discussed further in Section 5.

4 PATCH-BASED ADAPTIVE MESHES

As mentioned before, the fine-grained cell-wise adaptivity used in $\text{sam}(\text{oa})^2$ poses an obstacle to vectorisation over grid cells, which impedes performance on modern architectures like the Xeon Phi. In order to have more regular access patterns, we implemented a more coarse-grained patch-based approach, using the leaves of the refinement tree to store regularly refined patches, instead of single cells. Because the patches are regular and static, it becomes possible to reorganise their data in a way suitable for vectorisation. In this section we present the details of our implementation.

Regularly Refined Patches For the context of this work, we define a *patch* as a set of non-overlapping cells obtained by partitioning a right isosceles triangular cell. More precisely, we use the uniform refinement shown in Figure 1 (a), which shows patches with 9 cells each. Our implementation supports patches with N^2 cells for arbitrary $N \in \mathbb{N}_{>0}$, with the constraint that all patches in a mesh have the same size. In Section 6, we experiment with the patch-size N and evaluate its influence on performance.

The data of all cells in a patch is stored as a struct of arrays, following the cell numbering shown in Figure 1. For tsunami simulations, each patch contains four arrays with N^2 positions H , HU , HV and B , used to store the values of water height, water momentum in x and y directions and bathymetry, respectively. For simplicity, from now on we refer to these four arrays as Q .

Data Exchanges Between Neighbour Patches. In every time step, we compute approximate solutions for the Riemann problems at all edges of the mesh and use these solutions to update the cells. To update a patch, we thus need not only the data from its cells, but also the from the adjacent cells that lie in neighbour

Figure 1: (a) Two neighbour patches with 9 cells each. The cells at the boundaries of a patch depend on data from the boundaries of the neighbour patches, e.g., cells 1, 4 and 9 in the top-left patch need data from cells 1, 2, and 5 in the bottom-right patch, and vice versa. (b) Temporary arrays storing patch data with an *edge representation* – each array position represents an edge (or Riemann problem) in the patch. For simplicity, here we only list the internal edges. The actual implementation also needs to consider the boundary edges, which involve data from neighbour patches.

patches (Figure 1 (a) shows the relationship between two neighbour patches).

Therefore, boundary information needs to be exchanged between neighbour patches before actually solving the Riemann problems and updating the cells. This is performed in two steps, supported by operators provided by $\text{sam}(\text{oa})^2$. These operators apply kernels implemented by the application programmer to the elements in the mesh. For boundary exchange we use the *cell-to-edge operator*, which is applied to all patches, and the *skeleton operator*, which is applied to all boundaries between patches.

In the first step, the *cell-to-edge operator* extracts the boundary data from all patches and stores them as boundary representations (arrays of N positions storing Q_i from cells on the boundary). In the case of the patches with 9 cells shown in Figure 1, a representation of the patches' left boundary is obtained by copying the data from cells 1, 2 and 5. Similarly, the right boundary consists of data copied from cells 1, 4 and 9; and the bottom boundary of data from cells 5, 7 and 9.

Afterwards, the *skeleton operator* acts on the boundaries exchanging the boundary data between neighbour patches. Due to the provided operators, the application programmer only needs to specify which data should be exchanged, while $\text{sam}(\text{oa})^2$ takes care of actually performing the exchanges, managing the communication between different processes whenever necessary.

```

1 | if (H > dry_tolerance) then
2 |     U = HU / H
3 |     V = HV / H
4 | else
5 |     U = 0.0
6 |     V = 0.0
7 | end if

```

(a)

```

1 | where (H(:) > dry_tolerance)
2 |     U(:) = HU(:) / H(:)
3 |     V(:) = HV(:) / H(:)
4 | elsewhere
5 |     U(:) = 0.0
6 |     V(:) = 0.0
7 | end where

```

(b)

Figure 2: Part of a Riemann solver that computes the water velocity components U and V from the values of H , HU and HV of the cells. The original serial solver is shown in (a), while the vectorised implementation that solves multiple Riemann problems at once is shown in (b).

Vectorisation to Compute the Cell Updates. After the boundary exchange is complete, the patches have all data necessary to compute the cell updates and to perform a time step. However, even though all data is available in contiguous arrays, they are organised according to the cell numbering shown in Figure 1 (a), which is not suitable for solving the Riemann problems using vectorisation.

Thus, our implementation first switches to an *edge representation*, by organising the data in temporary arrays such that each position in the arrays contains the data for a Riemann problem. More specifically, we use eight arrays (four for each cell adjacent to the edge) of size equal to the number of edges in the patch. For every edge, the cell quantities Q_i and Q_j from its adjacent cells i and j are copied into these arrays. For instance, Q_1 and Q_3 are copied into the first position in the arrays, forming the input of the Riemann problem between these cells – see Figure 1 (b).

With this data organisation, it is possible to use vectorisation to solve the Riemann problems stored in the arrays. For that, we implemented *vectorised Riemann solvers* that solve multiple problems at once using SIMD instructions whenever possible. These vectorised solvers take multiple Riemann problems as input parameters and use Fortran array operations instead of single-value operations. An example of how they are implemented is shown in Figure 2, which presents excerpts of code from the original serial solver and from the vectorised solver.

Because there is no data dependency between different indices in the arrays, we can expect the compiler auto-vectorisation to be able to use SIMD instructions for implementing the array operations. An inspection on the compiler’s optimisation report confirms this expectation and shows that all solver computations are successfully vectorised, with the only exception of a step in the Augmented Riemann solver that is indeed not suitable for vectorisation, because its loop has multiple exits.

After computing the solutions, the vectorised Riemann solver stores them in output arrays with the same organisation as the input arrays. These output arrays are then used to update the cell quantities for the next time step, according to Equation (2). This part of the algorithm accesses the data in the output arrays and in the patch’s Q arrays following a very irregular pattern that cannot be vectorised.

An important observation regarding the actual implementation is that we don’t reorganise and solve all the Riemann problems in a patch at once, but rather in small chunks, in a strategy similar to cache-blocking. This is particularly beneficial for the Xeon Phi, since its cache sizes are significantly smaller than those of other modern architectures, and attempting to process all edges at once can considerably increase the cache-miss rate. Experimentally, we observed that the vectorised solvers perform best when processing chunks of 512 bits, matching the size of the vector registers.

5 LOAD BALANCING ON HETEROGENEOUS COMPUTE NODES

When using multiple processors and coprocessors, we face the problem of load imbalance caused by the heterogeneity of the system. Although $\text{sam}(\text{oa})^2$ provides a flexible definition of the computational load [23], it by default assumes that all processors have uniform efficiency, and thus assigns the same amount of load to all processors (similar for cores). We therefore implemented a heterogeneous load balancing strategy extending the approach in [23] by taking the relative efficiency of each processor into account. In this strategy, we determine the necessary load difference between processor and coprocessor either explicitly (with processor “speeds” manually defined by the user) or implicitly (with the speeds computed on-the-fly, based on performance statistics from previous time steps).

Heterogeneous Chains-on-chains Partitioning. The load balancing performed in $\text{sam}(\text{oa})^2$ is based on a 1D partitioning following the Sierpinski order, with sequences of contiguous cells defining so-called *Sierpinski sections*. The problem of optimally distributing these sections to the different ranks is known as chains-on-chains partitioning [32]. This partitioning maps $n \in \mathbb{N}$ sections with non-negative loads $\mathcal{L} = (l_1, l_2, l_3, \dots, l_n)$ to $p \in \mathbb{N}$ target processors with strictly positive associated *execution speeds* $\mathcal{E} = (e_1, e_2, e_3, \dots, e_p)$. The objective is to find a mapping function $f : \{1, \dots, n\} \rightarrow \{1, \dots, p\}$ that preserves the order of the sections and minimizes $\max_{i \in \{1, \dots, p\}} \sum_{j \in f^{-1}(i)} \frac{l_j}{e_i}$, i.e., the maximum *execution time* among all processors. The homogeneous chains-on-chains problem is a special case where all processors have the same speed, i.e. $e_i = 1$ for all i .

Since no scalable exact algorithm is known, we use a “midpoint” heuristic based on ideas from [31]. More precisely, we define a *midpoint position* $\mathcal{P}_j$ for each section j as:

$$\mathcal{P}_j := \left(\frac{l_j}{2} + \sum_{k=1}^{j-1} l_k \right) \cdot \frac{\sum_{k=1}^p e_k}{\sum_{k=1}^n l_k} \quad (3)$$

Figure 3: Nine sections with loads $\mathcal{L} = (2, 2, 3, 3, 2, 3, 1, 2, 2)$ being distributed to four processors with execution speeds $\mathcal{E} = (30, 30, 20, 20)$. The midpoint (centre) of each section defines which processor it is assigned to.

If we represent the n sections linearly, with the size of each section j being proportional to its load l_j , then $\mathcal{P}_j$ can be interpreted as the midpoint of section j – see Figure 3. The heuristic then uses these values to distribute sections to processors following this rule: each section j is assigned to processor i such that $\sum_{k=1}^{i-1} e_k \leq \mathcal{P}_j \leq \sum_{k=1}^i e_k$. In the homogeneous case, if we set $e_i = 1$ for all $i \in 1, \dots, p$, the function f can be directly obtained from $\mathcal{P}_j$, i.e., $f(j) = \mathcal{P}_j$. If processors can have different speeds, the partitioning is computed using a trivial search based on the definition above.

For the configurations used in this work, the described algorithm achieves decent results with respect to both the quality of the partitioning and to computational efficiency. However, for simulations with a larger number of processors, it might be necessary to implement other algorithms. A more detailed analysis of the heuristic algorithm is beyond the scope of this work. An overview on various algorithms for solving the heterogeneous chains-on-chains partitioning problem can be found in [32].

Defining Processor Execution Speeds. As the second step for achieving proper load balance in heterogeneous systems we need to have appropriate values for $\mathcal{E}$, the execution speeds for each processor, such that the performance is maximized. A first and straightforward approach is to obtain these by trial-and-error: experiment with different values for a small number of time steps, choose the configuration that works best and use it for the whole simulation. The problem with this approach is that the best load distribution is highly dependent on the architectures being used and on the simulated scenario. It may also be affected by small changes in the simulation options, such as minimum or maximum grid depth, choice of Riemann solver, size of the patches, etc. Thus, any change in the simulation configuration requires performing the trial-and-error phase again.

A second and more generic strategy is to implement an automatic tuning for $\mathcal{E}$ using on-the-fly statistics for the simulation. In this work, we use a simple auto-tuning algorithm that computes the execution speeds based on the execution time measured by each processor in the previous time step. In the first time step, the same speed is assigned to all processors (homogeneous distribution). Starting on the second time step, each processor’s speed is computed as the number of cells it processed divided by its execution time in the previous time step, or $e_i := c_i/t_i$, where c_i is the number of cells and t_i is the execution time, both measured only for processor i in the previous time step.

We expect this strategy to find a stable distribution after a few iterations. On a simpler application, we could expect this distribution to be very similar to the “best” configuration found using the

Figure 4: Simulation of the tsunami in Tohoku, Japan, 2011. The pictures show the tsunami wave 10, 20 and 30 minutes after the earthquake, respectively.

trial-and-error approach (which is equivalent to an “a priori auto-tuning”). However, because of the complexity of our application, the obtained performance may not be optimal. This is mainly because the computations performed by each processor are not completely independent – there are several communication routines performed at different points of a time step computation, which affect both the performance and the time measurements. Nevertheless, this auto-tuning strategy still achieves reasonable performance, as supported by the results presented in Section 6.2.

6 PERFORMANCE RESULTS

We evaluated our implementation’s performance on the Salomon supercomputer hosted at the IT4Innovations national supercomputing centre in Ostrava, Czech Republic. Salomon contains 432 compute nodes equipped with Xeon Phi coprocessors. Each of these nodes combines two Haswell CPUs (in the following labelled as the “host”) and two Xeon Phi coprocessors (labelled as “Phis”). The Xeon Phis are connected to the host via PCI express bus 2.0 connectors, and different nodes are connected via a high-speed 7D Enhanced hypercube InfiniBand FDR56 network. For an overview of the nodes’ system specifications see Table 1 in Section 1.

As a realistic simulation scenario, we simulate the 2011 Tohoku tsunami, illustrated in Figure 4. We use 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 [14]. All geographical input data is handled by the parallel library ASAGI [36]. For compilation and execution, we used the Intel Fortran Compiler 16.0.1 and Intel MPI 5.1.2. All experiments use double precision arithmetic.

6.1 Native Mode

In this section we evaluate the influence of patches on the performance, by running simulations on a single Xeon Phi in *native mode*, i.e., using the Xeon Phi as an independent processor, responsible for the entire execution.

Performance with Static Grids. We start our performance analysis by turning off adaptivity and experimenting with static grids. This setup is advantageous for evaluating the effectiveness of vectorisation, because there is no overhead due to adaptivity and the execution time is dominated by the numeric computations.

Figure 5 plots the results of our experiments with static grids of exactly 2^{25} cells ($\approx 33 \times 10^6$) generated with patches of different sizes (2^2 , 4^2 , 8^2 , 16^2 and 32^2) and also with single cells instead of

Table 2: Total cell updates performed during 1000 time steps with dynamically adaptive grids, using various patch sizes.

Patch size	1	2 ²	4 ²	8 ²	16 ²	32 ²
Cell updates (10 ⁹)	7.62	9.51	10.90	12.20	14.60	19.07

patches (the original code with no vectorisation, labelled as “1”). We present the results using the HLLE and Augmented Riemann solvers. We omit the results for the F-Wave solver because they are very similar to the ones with the HLLE solver and do not contribute to further insights.

The metric used is “cell updates per second”, computed by dividing the total number of processed cells by the simulation wall time. The simulations were run for 1000 time steps (≈ 15 minutes following the earthquake) and the measured time only includes the regular time-stepping phase, i.e., it does not consider the time necessary for reading the input files and generating the grid.

We ran the simulations on a single Xeon Phi using 2, 3 and 4 OpenMP threads per physical core (122, 183 and 244 threads in total, respectively), since these are the usual configuration candidates for highest performance on the Xeon Phi [20]. We also ran the same experiments on the host with 24 threads (12 on each Haswell), and plotted the results along with the Xeon Phi results.

We observe that patches increase the execution speed considerably on both architectures and with both solvers, and that performance increases along with the patches. However, there seems to be a saturation on the Phi for patches larger than 16² cells, which can be attributed to the fact that the arrays used for processing patches with 32² cells or larger do not fit entirely in the Xeon Phi’s small L2 cache (512KB per core). We also observe that the Xeon Phi benefits more from the patches than the Haswells (3.9 $\times$ vs. 2.7 $\times$ in the best cases), which was expected, due to its wider vector registers (512-bit vs. 256-bit). Moreover, using hyperthreading on the Xeon Phi definitively pays off – in all experiments, best performance was achieved with the maximum number of threads (four per core).

Performance with Dynamic Grids. Next, we evaluate the framework’s performance using dynamically adaptive meshes. We configured the simulations to apply the same finest resolution at the wave fronts in all runs, regardless of patch size. Thus, all simulations present the same accuracy. In these simulations, the number of cells is not constant over time and also depends on the sizes of the patches being used, with larger patches leading to more overall cells due to loss of adaptivity. Table 2 presents the total number of cell updates performed during 1000 time steps (≈ 1 minute in these experiments – AMR allows us to use higher resolutions than static grids) and shows how using large patches considerably increases the total work performed for a fixed finest resolution.

Figure 6 plots the performance obtained in these experiments. In this case, the behaviour is similar to the static grid experiments, except that the improvements are even greater on dynamic grids (5.1 $\times$ vs. 3.9 $\times$ in the best cases). This suggests that the algorithms for managing mesh adaptivity also benefit from regular patches. We discuss this behaviour further in the following paragraph (Component Analysis).

However, element throughput alone is not the appropriate metric for choosing the best patch size. Because larger patches also lead to more cells being processed, a trade-off must be found to minimise the simulation’s time-to-solution. With that in mind, in Figure 7 we compare the wall time for the experiments of Figure 6. We can see that there is a “sweet spot” at patches with around 8² cells, which minimise the execution time for both architectures and both solvers. On the Xeon Phi, the time-to-solution was effectively improved by 2.8 $\times$ and 2.6 $\times$ respectively with the HLLE and Augmented Riemann solvers, while on the Haswells the improvements were a little more modest: 2.3 $\times$ and 1.5 $\times$, respectively. These speedups are obtained despite a 60% increase in the total number of executed cell updates due to the 8² patches.

Component Analysis. The fact that the performance improvement is greater on dynamic grids than on static grids suggests that this improvement cannot be solely attributed to the use of vectorisation for the numerical computations. This can be confirmed by splitting the average thread time according to the simulation components (Figure 8). The plots compare the time taken by each component in runs with no patches and with patches of 8² cells, and show how each component is affected by the patches. Here, “Time step” is the component responsible for solving the Riemann problems and updating the cells, as described in Sections 3 and 4. “Adaptation” is responsible for performing refinement and coarsening of the cells, and “Conformity” makes sure that the mesh does not have hanging nodes (nodes that do not belong to one of its adjacent cells) by further refining some of the cells. After this remeshing, the “Neighbours” component identifies the sections’ neighbours and updates their communication structures.

It is noticeable that the patch-based discretisation is beneficial not only to the time stepping component, but also to other time-consuming components, such as the components responsible for performing refinement/coarsening (Adaptation) and for maintaining grid conformity (Conformity). Even though the number of processed cells is greater when using patches, all relevant components have their computation time substantially reduced, not only on the Xeon Phi, but on the Haswells as well. Also, the huge improvements in the Adaptation and Conformity components show that the decrease of the refinement tree size plays a very important role on increasing the simulation performance.

6.2 Symmetric Mode

In this section we attempt to obtain the best possible performance from using all resources available in the cluster, i.e., multiple Haswell processors and Xeon Phi coprocessors. In all experiments, we placed one MPI rank on each processing unit, thus a full node contained four ranks: two on the Haswells and two on the Xeon Phis. As in the native mode, shared memory parallelism was managed by OpenMP. We used either 11 or 12 cores on each Haswell processor depending on the execution mode. For symmetric mode (using host and Xeon Phis), we experienced that reserving one core in each Haswell for managing MPI communication with the Xeon Phis delivers approx. 10 % better performance than using all cores. This is in line with the recommendations for symmetric mode using the MPI proxy, cf. [34], because one core of the host CPU is kept busy by the communication management. On the other hand, if the

Figure 5: Performance with patches of various sizes on static grids. Vectorisation successfully increases the performance on both architectures, especially for patches with 16^2 or more cells.

Figure 6: Performance with dynamic grids. The gain in performance is even greater than with static grids, showing that the algorithms for mesh adaptivity also benefit from regular patches.

Figure 7: Time-to-solution for 1000 time steps with dynamic grids. There is a “sweet spot” at patches with around 8^2 cells, which minimise the execution time for both architectures and both solvers. Even though patches larger than that achieve higher performance, they are not the best choice because they also increase the number of cells in the grid.

Figure 8: Component breakdown of the average thread time for 1000 time steps, comparing the implementations with and without patches, on both machines. Even though the number of processed cells was 60% greater when using patches with 8^2 cells, the most relevant components had their computation time substantially reduced.

Xeon Phis are not being used, we can use all Haswell cores without degrading performance. On the Xeon Phis, we used all cores with 4 threads per physical core, since this configuration delivered the best performance in the experiments performed in Section 6.1. For the same reason, all simulations reported here used patches with 8^2 cells.

In these experiments, we only used the Augmented Riemann solver, because it is the most compute-intensive solver provided by our framework, performing roughly $3\times$ more floating-point operations than the HLLC solver. Because of some issues (which will be discussed later on), MPI communication was a huge bottleneck when using the other solvers. Thus, symmetric mode was not suitable in these cases, for which maximal performance on the cluster is achieved by using only the Haswell processors.

Single-Node Performance. In Figure 9, we compare the performance obtained when using only the dual-socket host, only the Xeon Phi, and also all of them in symmetric mode. For the symmetric mode, we compare three different load balancing strategies: the naive homogeneous distribution (homog.), and heterogeneous distributions defined either explicitly before the simulation (manual) or implicitly on-the-fly (auto), as discussed in Section 5.

We can see that using the symmetric mode does pay off for a single node, when heterogeneous load balancing is considered. In the case where the execution speeds were defined manually (in these experiments, giving 36% of the load to the Haswells and 64% to the Phis), performance is more than 26% faster than using the homogeneous load balancing or only the Phi. Also, after approximately 40 time steps, the auto-tuning algorithm was able to find a stable distribution (around 30%/70%), which obtained a performance very close to the best case found out by the trial-and-error strategy (about 20% faster than the homogeneous distribution).

However, we also notice that all simulations involving Xeon Phi performed considerably slower than what could be expected – e.g., using two Xeon Phi is only 70% faster than using a single Xeon

Figure 9: Comparison of single-node performance with Haswell and Xeon Phi architectures and with symmetric mode using different load balancing strategies.

Phi. By breaking down the execution time into components (see Figure 10), we can identify components that do not scale on the Xeon Phi. While all simulation components scale quite well on two hosts (in different nodes), some components (conformity check and adaptation) do not scale at all, neither on two Xeon Phi in the same node nor on two Xeon Phi in different nodes.

The fact that the components that scale badly on two Phis depend heavily on MPI communication suggests that it might be the problem. We used the Intel MPI Benchmarks to replicate the communication pattern used in sam(oa)² and to compare the bandwidth achieved by different source/destination combinations – see Figure 11. Clearly, communication involving Xeon Phi is substantially slower, which confirms our suspicion that communication might be the bottleneck in the runs that use Xeon Phi and MPI, and explains why the performance with symmetric mode is not so great.

Figure 10: Component breakdown of the average thread time on hosts and Xeon Phi. Here, “2 hosts” and “2x1 Phi” refer to simulations run on devices in different nodes, while “2 Phi” use two Xeon Phi in the same node.

Figure 11: Results of the Intel MPI benchmarks used to measure bandwidth of MPI communication involving Haswell host processors and Xeon Phi coprocessors. The highlighted region represents message sizes responsible for more than 60% of the messages and more than 95% of the total data exchanged by sam(oa)² in our experiments. The benchmarks were run with arguments “PingPong -off_cache 1024 -iter_policy off -mem 4”. Experiments “Host-MIC” and “MIC-MIC” use (co)processors in the same node, while “Host-Host” uses two Haswell processors in different nodes.

Multi-Node Performance. Now we perform similar experiments using multiple nodes. We reproduce a weak scaling setting, where the number of cell updates performed per node is approximately the same for all runs (≈ 35 billion for 1000 time steps). The resulting plot is presented in Figure 12.

While using only the Haswells presents good scalability, all nodes that involve Xeon Phi do not scale well, due to communication overhead. Even though symmetric mode with manually adjusted weights gives the best performance for all node counts,

Figure 12: Experiments with multiple nodes comparing performance of different load balancing strategies. The values are normalized as performance per number of nodes.

the benefits of symmetric mode decrease as the number of nodes increases. For the smaller runs (up to 4 nodes), the auto-tuning approach performed closely to manual definition, but it was not successful with more nodes. On eight nodes it did not bring any significant improvement over the homogeneous distribution, and on more nodes it led to runtime errors after attempting to allocate too much load on Xeon Phi, making them run out of memory. Thus, future implementations of this approach should also be able to consider memory limitations for each machine, especially for devices with small memory like the Xeon Phi.

7 CONCLUSIONS

In this paper we described our work on optimising tsunami simulations on the Salomon supercomputer. We implemented a patch-based approach that reduced the complexity of the mesh refinement tree while increasing memory bandwidth and allowing vectorisation. With these improvements, simulation performance increased considerably on the Xeon Phi coprocessors and moderately on the Haswell processors. We found the best time-to-solution for comparably small patch sizes (8² cells per patch), which in our example led to 60% more grid cells, but achieved up to 2.8× smaller time-to-solution.

The mixture of two Haswell host CPUs and two Xeon Phi coprocessors on Salomon and their roughly comparable performance for our tsunami simulation scenario suggested to use the coprocessors in *symmetric mode*, instead of native mode or offload approaches. We therefore analysed performance in symmetric mode and showed that it achieves better performance than using only Haswells or only Xeon Phis, provided that the system heterogeneity is taken into account in the load balancing algorithm. Particularly, we used an approach that uses on-the-fly runtime measurements to automatically determine the relative load distribution. On small node counts (up to 4 nodes), this approach delivered performance very close to that achieved by manually defining the distribution based

on prior trial-and-error experiments. However, it failed to do the same for greater number of nodes.

While we conclude that symmetric mode can be successfully used on heterogeneous platforms like the Salomon supercomputer, we also note that execution is not as fast as could be expected when considering the speed of the machines individually. This is due to the inefficiency of MPI communication involving Xeon Phi coprocessors, which we were able to replicate using the Intel MPI Benchmarks. For future Xeon Phi generations (including the current Knights Landing CPUs), we expect that MPI communication will be improved and that symmetric mode (i.e. placing one MPI rank per device) will become the default programming approach for systems containing Xeon Phi (co)processors. We demonstrated that parallel adaptive mesh refinement with dynamic load balancing can be achieved on such systems by introducing the concept of an execution speed for each MPI process, which can either be manually tuned or automatically determined from the execution time.

ACKNOWLEDGMENTS

We especially thank our project partners at the IT4Innovations national supercomputing centre in Ostrava, in particular Branislav Janský, for their excellent support of our work on Salomon. The presented work was part of the project CzeBaCCA, funded by the German Federal Ministry of Education and Research (BMBF, grant no. 01DS16009B). Chaulio R. Ferreira also appreciates the support of CNPq, the Brazilian Council of Technological and Scientific Development (grant no. 234439/2014-9).

REFERENCES

- [1] Alexey Androsov, Jörn Behrens, and Sergey Danilov. 2011. Tsunami Modelling with Unstructured Grids. Interaction between Tides and Tsunami Waves. In *Computational Science and High Performance Computing IV*, Vol. 115. 191–206.
- [2] Michael Bader, Christian Böck, Johannes Schwaiger, and Csaba Attila Vigh. 2010. Dynamically Adaptive Simulations with Minimal Memory Requirement – Solving the Shallow Water Equations Using Sierpinski Curves. *SIAM Journal of Scientific Computing* 32, 1 (2010), 212–228.
- [3] Derek S. Bale, Randall J. LeVeque, Sorin Mitran, and James A. Rossmann. 2002. A wave propagation method for conservation laws and balance laws with spatially varying flux functions. *SIAM Journal on Scientific Computing* 24, 3 (2002), 955–978.
- [4] Jörn Behrens and Jens Zimmermann. 2000. Parallelizing an Unstructured Grid Generator with a Space-Filling Curve Approach. In *Euro-Par 2000 Parallel Processing (Lecture Notes in Computer Science)*, Vol. 1900. Springer Berlin Heidelberg, 815–823.
- [5] Gheorghe-Teodor Bercea, Andrew T. T. McRae, David A. Ham, Lawrence Mitchell, Florian Rathgeber, Luigi Nardi, Fabio Luporini, and Paul H. J. Kelly. 2016. A structure-exploiting numbering algorithm for finite elements on extruded meshes, and its performance evaluation in Firedrake. *Geoscientific Model Development* 9, 10 (2016), 3803–3815. <https://doi.org/10.5194/gmd-9-3803-2016>
- [6] Marsha J. Berger and Phillip Colella. 1989. Local adaptive mesh refinement for shock hydrodynamics. *Journal of Computational Physics* 82 (1989), 64–84.
- [7] Marsha J. Berger, David L. George, Randall J. LeVeque, and Kyle T. Mandli. 2011. The GeoClaw software for depth-averaged flows with adaptive refinement. *Advances in Water Resources* 34, 9 (2011), 1195–1206.
- [8] Marsha J. Berger and Joseph Oliger. 1984. Adaptive mesh refinement for hyperbolic partial differential equations. *Journal of Computational Physics* 53 (1984), 484–512.
- [9] Carsten Burstedde, Donna Calhoun, Kyle Mandli, and Andy R. Terrel. 2014. ForestClaw: Hybrid forest-of-octrees AMR for hyperbolic conservation laws. In *Parallel Computing: Accelerating Computational Science and Engineering (CSE) (Advances in Parallel Computing)*, Vol. 25. 253–262. <https://doi.org/10.3233/978-1-61499-381-0-253>
- [10] Carsten Burstedde, Lucas C. Wilcox, and Omar Ghattas. 2011. p4est: Scalable Algorithms for Parallel Adaptive Mesh Refinement on Forests of Octrees. *SIAM Journal on Scientific Computing* 33, 3 (2011), 1103–1133.
- [11] Richard Courant, Kurt Friedrichs, and Hans Lewy. 1967. On the partial difference equations of mathematical physics. *IBM journal* 11, 2 (1967), 215–234.
- [12] Anshu Dubey, Ann Almgren, John Bell, Martin Berzins, Steve Brandt, Greg Bryan, Phillip Colella, Daniel Graves, Michael Lijewski, Frank Löffler, Brian O'Shea, Erik Schnetter, Brian Van Straalen, and Klaus Weide. 2014. A survey of high level frameworks in block-structured adaptive mesh refinement packages. *J. Parallel and Distrib. Comput.* 74, 12 (2014), 3217–3227. <https://doi.org/10.1016/j.jpdc.2014.07.001> Domain-Specific Languages and High-Level Frameworks for High-Performance Computing.
- [13] Bernd Einfeldt. 1988. On Godunov-type methods for gas dynamics. *SIAM J. Numer. Anal.* 25, 2 (1988), 294–318.
- [14] Percy Galvez, Jean-Paul Ampuero, Luis A. Dalguer, Surendra N. Somala, and Tarje Nissen-Meyer. 2014. Dynamic earthquake rupture modelled with an unstructured 3-D spectral element method applied to the 2011 M9 Tohoku earthquake. *Geophysical Journal International* 198, 2 (2014), 1222–1240.
- [15] David L. George. 2008. Augmented Riemann solvers for the shallow water equations over variable topography with steady states and inundation. *J. Comput. Phys.* 227, 6 (2008), 3089–3113.
- [16] Sven Harig, Chaeroni, Widodo S. Pranowo, and Jörn Behrens. 2008. Tsunami simulations on several scales. *Ocean Dynamics* 58, 5 (2008), 429–440. <https://doi.org/10.1007/s10236-008-0162-5>
- [17] Alexander Heinecke, Roman Karlstetter, Dirk Pflüger, and Hans-Joachim Bungartz. 2015. Data Mining on Vast Datasets as a Cluster System Benchmark. *Concurrency and Computation: Practice and Experience* 28, 7 (2015), 2145–2165.
- [18] Yuta Hirokawa, Taisuke Boku, Shunsuke A. Sato, and Kazuhiro Yabana. 2016. Electron Dynamics Simulation with Time-Dependent Density Functional Theory on Large Scale Symmetric Mode Xeon Phi Cluster. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 1202–1211. <https://doi.org/10.1109/IPDPSW.2016.200>
- [19] Alan Humphrey, Daniel Sunderland, Todd Harman, and Martin Berzins. 2016. Radiative Heat Transfer Calculation on 16384 GPUs Using a Reverse Monte Carlo Ray Tracing Approach with Adaptive Mesh Refinement. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 1222–1231. <http://www.sci.utah.edu/publications/Hum2016a/ipdps-pdsec16.pdf>
- [20] James Jeffers and James Reinders. 2013. *Intel Xeon Phi coprocessor high-performance programming*. Newnes.
- [21] Randall J. LeVeque, David L. George, and Marsha J. Berger. 2011. Tsunami modelling with adaptively refined finite volume methods. *Acta Numerica* 20 (2011), 211–289.
- [22] Kyle T. Mandli and Clint N. Dawson. 2014. Adaptive mesh refinement for storm surge. *Ocean Modelling* 75 (2014), 36–50. <https://doi.org/10.1016/j.ocemod.2014.01.002>
- [23] Oliver Meister. 2016. *Sierpinski Curves for Parallel Adaptive Mesh Refinement in Finite Element and Finite Volume Methods*. Dissertation. Institut für Informatik, Technische Universität München. <https://mediatum.ub.tum.de/doc/1320149/1320149.pdf>
- [24] Oliver Meister and Michael Bader. 2015. 2D adaptivity for 3D problems: Parallel SPE10 reservoir simulation on dynamically adaptive prism grids. *Journal of Computational Science* 9 (2015), 101–106.
- [25] Oliver Meister, Kaveh Rahnama, and Michael Bader. 2016. Parallel Memory-Efficient Adaptive Mesh Refinement on Structured Triangular Meshes with Billions of Grid Cells. *ACM Transactions on Mathematical Software* 43, 3 (2016), 19.
- [26] Qingyu Meng, Alan Humphrey, John Schmidt, and Martin Berzins. 2013. Investigating Applications Portability with the Uintah DAG-based Runtime System on PetaScale Supercomputers. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC '13)*. ACM, 961–9612. <https://doi.org/10.1145/2503210.2503250>
- [27] Qingyu Meng, Alan Humphrey, John Schmidt, and Martin Berzins. 2013. Preliminary Experiences with the Uintah Framework on Intel Xeon Phi and Stampede. In *Proceedings of the Conference on Extreme Science and Engineering Discovery Environment: Gateway to Discovery (XSEDE '13)*. ACM, 48:1–48:8. <https://doi.org/10.1145/2484762.2484779>
- [28] William F. Mitchell. 1991. Adaptive refinement for arbitrary finite-element spaces with hierarchical bases. *Journal of computational and applied mathematics* 36, 1 (1991), 65–78.
- [29] William F. Mitchell. 2007. A Refinement-Tree Based Partitioning Method for Dynamic Load Balancing with Adaptively Refined Grids. *J. Parallel and Distrib. Comput.* 67, 4 (2007), 417–429.
- [30] Andreas Mueller, Michal Kopera, Simone Marras, Lucas Wilcox, Tobin Isaac, and Francis X. Giraldo. 2016. Strong scaling for numerical weather prediction at petascale with the atmospheric model NUMA. *International Journal for High-Performance Computing Applications* (2016).
- [31] Ali Pinar and Cevdet Aykanat. 2004. Fast optimal load balancing algorithms for 1D partitioning. *J. Parallel Distrib. Comput.* 64, 8 (2004), 974–996.
- [32] Ali Pinar, E. Kartal Tabak, and Cevdet Aykanat. 2008. One-dimensional partitioning for heterogeneous systems: Theory and practice. *J. Parallel and Distrib. Comput.* 68, 11 (2008), 1473–1486.

- [33] Stephane Popinet. 2012. Adaptive modelling of long-distance wave propagation and fine-scale flooding during the Tohoku tsunami. *Natural Hazards and Earth System Sciences* 12 (2012), 1213–1227.
- [34] Sreeram Potluri, Devendar Bureddy, Khaled Hamidouche, Akshay Venkatesh, Krishna Kandalla, Hari Subramoni, and Dhabaleswar K. (Dk) Panda. 2013. MVAPICH-PRISM: A Proxy-based Communication Framework Using InfiniBand and SCIF for Intel MIC Clusters. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC '13)*. ACM, Article 54, 11 pages. <https://doi.org/10.1145/2503210.2503288>
- [35] Abtin Rahimian, Ilya Lashuk, Shravan Veerapaneni, Aparna Chandramowlishwaran, Dhairya Malhotra, Logan Moon, Rahul Sampath, Aashay Shringarpure, Jeffrey Vetter, Richard Vuduc, Denis Zorin, and George Biros. 2010. Petascale Direct Numerical Simulation of Blood Flow on 200K Cores and Heterogeneous Architectures. In *Supercomputing 2010*. 1–11.
- [36] Sebastian Rettenberger, Oliver Meister, Michael Bader, and Alice-Agnes Gabriel. 2016. ASAGI – A Parallel Server for Adaptive Geoinformation. In *Proceedings of the Exascale Applications and Software Conference 2016 (EASC '16)*. ACM, 2:1–2:9. <http://delivery.acm.org/10.1145/2940000/2938618/a2-Rettenberger.pdf>
- [37] Martin Schreiber and Hans-Joachim Bungartz. 2014. Cluster-based communication and load balancing for simulations on dynamically adaptive grids. In *Proceedings of the International Conference on Computational Science (ICCS'14) (Procedia Computer Science)*, Vol. 29. Elsevier, 2241–2253.
- [38] Jie Shen, Ana Lucia Varbanescu, Yutong Lu, Peng Zou, and Henk Sips. 2016. Workload Partitioning for Accelerating Applications on Heterogeneous Platforms. *IEEE Transactions on Parallel and Distributed Systems* 27, 9 (2016), 2766–2780.
- [39] Hari Sundar and Omar Ghattas. 2015. A Nested Partitioning Algorithm for Adaptive Meshes on Heterogeneous Clusters. In *Proceedings of the 29th ACM on International Conference on Supercomputing*. ACM, 319–328.
- [40] Kristof Unterweger, Roland Wittmann, Philipp Neumann, Tobias Weinzierl, and Hans-Joachim Bungartz. 2015. Integration of FULLSWOF2D and PeanoClaw: Adaptivity and Local Time-stepping for Complex Overland Flows. In *Recent Trends in Computational Engineering – CE2014 (Lecture Notes in Computational Science and Engineering)*, Vol. 105. Springer, 181–195.
- [41] Karthikeyan Vaidyanathan, Kiran Pamnany, Dhiraj D. Kalamkar, Alexander Heinicke, Mikhail Smelyanskiy, Jongsoo Park, Daehyun Kim, Aniruddha Shet G., Bharat Kaul, B'alint Jo'o, and Pradeep Dubey. 2014. Improving Communication Performance and Scalability of Native Applications on Intel Xeon Phi Coprocessor Clusters. In *28th IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2014, Phoenix, AZ, USA, May 19-23, 2014*. <https://doi.org/10.1109/IPDPS.2014.113>
- [42] Mohamed Wahib, Naoya Maruyama, and Takayuki Aoki. 2016. 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)*. IEEE Press, 53:1–53:12. <http://dl.acm.org/citation.cfm?id=3014904.3014975>
- [43] Tobias Weinzierl, Michael Bader, Kristof Unterweger, and Roland Wittmann. 2014. Block Fusion on Dynamically Adaptive Spacetime Grids for Shallow Water Waves. *Parallel Processing Letters* 24, 3 (2014), 1441006.