Parallel Scalability of Adaptive Mesh Refinement in a Finite Difference Solution to the Shallow Water Equations

Ola Toft and Jan Christian Meyer
Department of Computer Science, NTNU
Trondheim, Norway

Abstract

The Shallow Water Equations model the fluid dynamics of deep ocean flow, and are used to simulate tides, tsunamis, and storm surges. Numerical solutions using finite difference methods are computationally expensive enough to mandate the use of large computing clusters, and the cost grows not only with the amount of fluid, but also the duration of the simulated event, and the resolution of the approximation. The benefits of increased resolution are mostly connected to regions where complex fluid interactions occur, and are not required globally for the entire simulation. In this paper, we investigate the potential for conserving computational resources by applying Adaptive Mesh Refinement to dynamically determined areas of the fluid surface. We implement adaptive mesh refinement in a MacCormack finite difference solver, develop a performance model to predict its behavior on large-scale parallel platforms, and validate its predictions experimentally on two computing clusters. We find that the solver itself has highly favorable parallel scalability, and that the addition of refined areas introduces a performance penalty due to load imbalance that is at most proportional to the refinement degree raised to the third power.

1 Introduction

The Shallow Water Equations model the fluid dynamics of flow in the deep ocean, coastal areas and rivers, and are used to simulate events such as tides, tsunami waves, and storm surges [1]. They can be solved by numerical approximation using finite difference methods, but the resulting computational workload is numerically intensive, and requires the high performance computing resources of a computational cluster or supercomputer in order to produce timely solutions for large problem sizes. Interesting effects in fluid dynamics can often occur at different scales, *e.g.* a wave may be sufficiently described by a coarse-grained simulation in the open sea, but develop complex, detailed interactions with minor features of the terrain when it makes landfall. Adaptive mesh refinement can combine coarse-grained and fine-grained solutions in particular areas of interest, thereby allowing a numerical solver program to capture smaller details without requiring the same resolution across the entire simulated domain. In a parallel computing context, variable resolution in different parts of the domain is prone to producing load balancing issues, as

This paper was presented at the NIK-2020 conference; see http://www.nik.no/.

it becomes difficult to distribute the additional time and memory requirements of refined regions evenly among all participating processors.

In this paper, we contribute an analytic performance model to quantify the impact of this effect, develop a performance proxy application that can isolate it experimentally, and evaluate its significance on two modern computing platforms. We find that the application is highly scalable, but that its performance is sensitive to situations where the simulated fluid requires greater mesh refinement in small, localized regions of the domain.

The rest of this paper is structured as follows. Section 2 describes how we discretize the Shallow Water Equations in our numerical solver program. Section 3 reviews adaptive mesh refinement, and describes choices made in our implementation. Section 4 describes how the solver program is parallelized for use on distributed-memory high performance computing systems. Section 5 develops a performance model of the program. Section 6 discusses the results of our practical experiments, Section 7 summarizes related approaches, and Section 8 concludes our study.

2 Discretization of the Shallow Water Equations

The Shallow Water Equations (SWE) are a set of partial differential equations, which model a body of fluid as a two-dimensional map of fluid surface elevation relative to a reference level. They are stated in Eqs. 1-3, where ρ is the fluid density, η is the fluid column height, u and v are the fluid's horizontal flow velocities in x and y directions, and g is the constant acceleration due to gravity.

$$\frac{\partial(\rho\eta)}{\partial t} + \frac{\partial(\rho\eta u)}{\partial x} + \frac{\partial(\rho\eta v)}{\partial y} = 0 \tag{1}$$

$$\frac{\partial(\rho\eta u)}{\partial t} + \frac{\partial}{\partial x}\left(\rho\eta u^2 + \frac{1}{2}\rho g\eta^2\right) + \frac{\partial(\rho\eta uv)}{\partial y} = 0 \tag{2}$$

$$\frac{\partial(\rho\eta v)}{\partial t} + \frac{\partial(\rho\eta uv)}{\partial x} + \frac{\partial}{\partial y}\left(\rho\eta v^2 + \frac{1}{2}\rho g\eta^2\right) = 0 \tag{3}$$

We discretize the SWE using the MacCormack method, which is an explicit, two-step predictor/corrector finite difference approximation [2]. The predictor step in Eq. 4 uses forward differences in time and space, where u is a value at coordinate i at time t, and Δx is a spatial finite difference.

$$u_i^{\overline{n+1}} = u_i^n - \Delta t\left(\frac{f(u_{i+1}^n) - f(u_i^n)}{\Delta x}\right) \tag{4}$$

The corrector step is given as Eq. 5, it averages values from the current time step and the predictor step, using backward differences and half the spatial difference.

$$u_i^{n+1} = \frac{u_i^n + u_i^{\overline{n+1}}}{2} - \frac{\Delta t}{2}\left(\frac{f(u_i^{\overline{n+1}}) - f(u_{i-1}^{\overline{n+1}})}{\Delta x}\right) \tag{5}$$

Using Eqs. 4 and 5 as templates, we obtain the MacCormack scheme for the fluid level of Eq. 1 as Eqs. 6 and 7, the velocity u of Eq. 2 as Eqs. 8 and 9, and the velocity v of Eq. 3 with a similar substitution as for u.

$$\rho\eta^{\overline{t+1}} = \rho\eta^t - \Delta t\left(\frac{\rho\eta u_{x+1} - \rho\eta u_x}{\Delta x} + \frac{\rho\eta v_{y+1} - \rho\eta v_y}{\Delta y}\right) \tag{6}$$

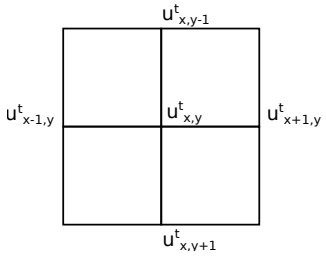


Figure 1: Five-point numerical stencil pattern of locations required to update variable $u_{x,y}^{t+1}$

$$\rho\eta^{t+1} = \frac{\rho\eta^{t} + \rho\eta^{\overline{t+1}}}{2} - \frac{\Delta t}{2}\left(\frac{\rho\eta u_{x}^{\overline{t+1}} - \rho\eta u_{x-1}^{\overline{t+1}}}{\Delta x} + \frac{\rho\eta v_{y}^{\overline{t+1}} - \rho\eta v_{y-1}^{\overline{t+1}}}{\Delta y}\right) \tag{7}$$

$$\rho\eta u^{\overline{t+1}} = \rho\eta u^{t} - \Delta t\left(\frac{du_{y,x+1} - du_{y,x}}{\Delta x} + \frac{\rho\eta uv_{y+1,x} - \rho\eta uv_{y,x}}{\Delta y}\right) \tag{8}$$

$$\rho\eta u^{t+1} = \frac{\rho\eta u^{t} + \rho\eta u^{\overline{t+1}}}{2} - \frac{\Delta t}{2}\left(\frac{du_{y,x} - du_{y,x-1}}{\Delta x} + \frac{\rho\eta uv_{y,x} - \rho\eta uv_{y-1,x}}{\Delta y}\right) \tag{9}$$

In Eqs. 8 and 9, $du_{y,x}$ is written as a shorthand notation for the expression

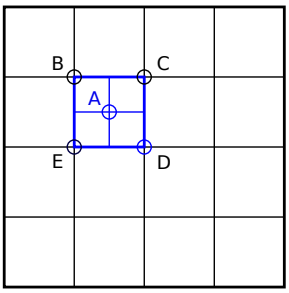
$$du_{y,x} = \rho\eta u_{y,x}^{2} + \frac{1}{2}\rho g\eta_{y,x}^{2}$$

In our numerical solver program, $\rho\eta$, $\rho\eta u$ and $\rho\eta v$ are represented as two-dimensional arrays, each with three buffers to store the values of the present time step t, the predictor step $\overline{t+1}$, and the subsequent time step $t+1$, respectively. Time integration proceeds by evaluating predictor and corrector steps for all three variables, before swapping the current time step buffers with their successors in the corrector step buffers, thus advancing simulated time by Δt. We model fluid in a rectangular area with Neumann boundary conditions, to mimic an isolated, uniform tank where waves reflect from the boundaries.

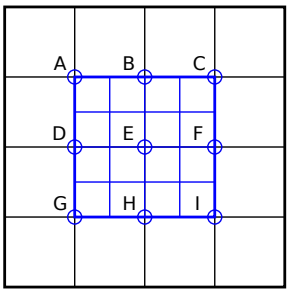
3 Adaptive Mesh Refinement

Without further development, the numerical solution in Section 2 implies that updates to each variable will depend on values from a common 5-point stencil pattern of neighboring points [3], as illustrated in Fig. 1. In order to selectively refine the spatial and temporal resolution in particular areas, we implement *mesh refinement* by overlaying a more refined grid in the region of interest, as shown in Fig. 2a. The required 5-point stencil values for point A are interpolated from the values at points B, C, D, E in the coarsely refined grid. This refinement may be applied recursively to a region, creating a hierarchy of successively refined grids, rooted in the original, most coarse-grained representation. Note that this approach implies that a single position within the problem domain may be represented multiple times in different memory locations for each resolution. Henceforth, we refer to one instance of coordinates and time/space resolution as a *point*.

In order to maintain the Courant-Friedichs-Lewy condition [4] of the root solution, solution of the spatially refined grids must also proceed at a proportionately refined rate in time, as illustrated in Fig. 3. At each level, the border values of a refined sub-grid must



(a) Point A in the refined, blue grid is derived by spatial interpolation of points B, C, D and E in the black grid.



(b) The values of points A-I in the black parent grid are replaced with values from points A-I in the blue grid.

Figure 2: Refinement and downsampling

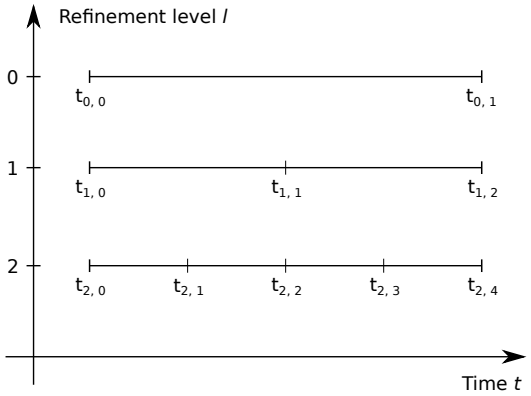


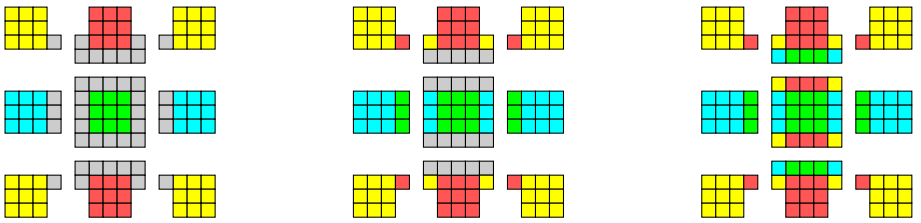
Figure 3: Recursive evolution of the solution in time for different levels of refinement. $t_{l,i}$ is the time at the beginning of time step i in the reference system of a grid at level l.

first be interpolated from its parent in the grid hierarchy. Next, a refined solution can be obtained by time integration of the refined grid, and finally, the results can substitute the values of coinciding points in the parent grid, as shown in Fig. 2b.

The objective of mesh refinement is to conserve memory and computational resources by only refining the grid structure in areas where additional precision is required, as opposed to refining the entire problem domain. Due to the fluctuations of a moving fluid, we can expect regions that are suitable candidates for grid refinement to move around in the computational domain, reflecting the dynamic behavior of wavefronts, reflections, interference patterns, and other events of particular interest. In order to account for this, we implement *adaptive* mesh refinement by introducing a parametric *regridding interval* that periodically triggers a reevaluation of the system state, and a quantitative criterion by which domain areas can be identified as candidates for grid refinement. Regridding marks all domain points that require refinement, clusters the marked points into rectangles, and replaces the previous grid hierarchy with corresponding new sub-grids. We implement the clustering step using the algorithm of Berger and Rigoutsos [5].

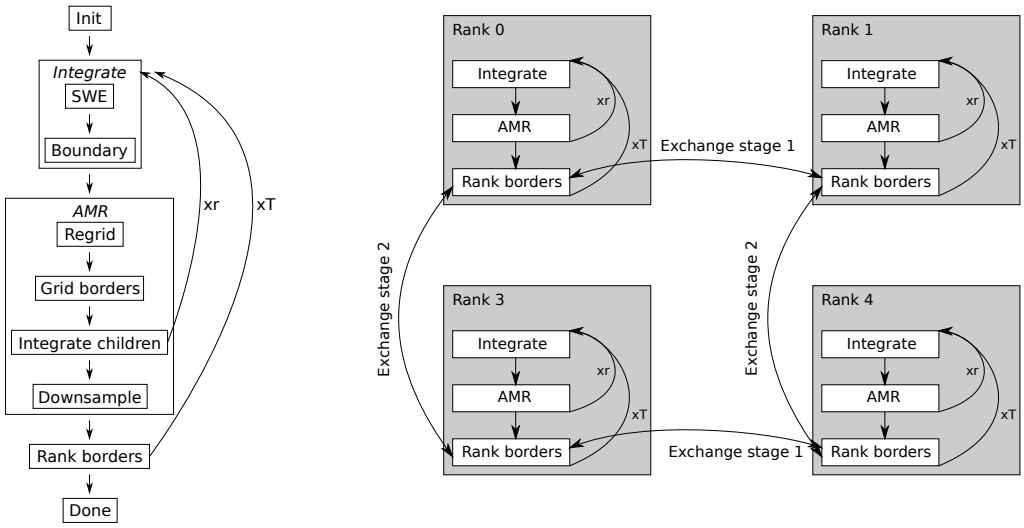
4 Program Parallelization

Since time step computation applies the numerical stencils independently to each grid point, our only sequential dependency is that each time step must be completed before the next can be started. This creates the common execution pattern known as *synchronous iterations* [6], or *bulk-synchronous parallelism* [7], where computations in a time step



(a) Initial state (b) Horizontal exchange (c) Vertical exchange

Figure 4: Staged border exchange in a Cartesian grid. Grid cells are color-coded according to the neighboring partition they originate from, grey cells represent border padding that has not yet been updated by the exchange.



(a) Computation stages per rank (b) Communication stages with 4 ranks

Figure 5: Stages of execution for each rank, and communication between 4 parallel ranks. Back edges indicate loops for each grid refinement level (xr), and for each time step (xT).

are parallel, while time integration synchronizes all participants. We implement the bulk-synchronous pattern combining message passing and multithreading techniques [8], using MPI for message passing, and OpenMP for multithreading. This combination allows us to exploit the lower resource requirements of light-weight thread parallelism at the scale of individual multi-core computing nodes, while message passing admits problems that require multiple, networked nodes. We partition the domain across distributed memory in a Cartesian topology, where nodes are ordered in a two-dimensional coordinate system and have neighbors in four directions. When the two-dimensional arrays that represent the state of the fluid are partitioned across distributed memory, the synchronization step between time steps requires a number of message passing operations, because the value of the computational stencil along the borders of one partition will depend on 1 or 2 values that are assigned to a neighboring compute node. We solve this by padding each individual partition with an additional border of 1 domain point in each direction, and explicitly copying its latest value from the neighboring partition during synchronization, using a staged border exchange as illustrated in Fig. 4.

5 Performance Model

An outline of the control flow and communication pattern in our program is shown in Fig. 5. Its three main tasks are to integrate the solution in time (*SWE*), adapt mesh resolutions accordingly (*AMR*), and exchange data with neighboring ranks (*Rank borders*). We adopt the *fundamental equation of modeling* from Barker *et al.*[9] stated in Eq. 10 as a starting point. Its *overlap* term is taken to be $T_{overlap} = 0$, as it is beyond the scope of this paper to investigate the application's potential for simultaneous computation and communication.

$$T_{total} = T_{compute} + T_{communicate} \tag{10}$$

Based on the overview in Fig. 5, we develop Eq. 10 into the application specific Eq. 11.

$$T_{total} = (T_{integrate} + T_{AMR}) + T_{exchange} \tag{11}$$

Integration Cost $T_{integrate}$

Eq. 12 further partitions the total cost of integration into the cost of developing the SWE for the interior points of the domain, and applying the boundary condition at its edges.

$$T_{integrate} = T_{SWE} + T_{boundary} \tag{12}$$

We express the SWE cost in terms of a cost W_{SWE} for each use of the numerical stencil, and assume that it is constant for all points. The aggregate number of points is the sum of all subgrid sizes, which we express using the product of the sides X_g, Y_g for each subgrid g, and we let G denote the set of all subgrids, as in Eq. 13.

$$T_{SWE} = (\sum_{g \in G} X_g \cdot Y_g) \cdot W_{SWE} \tag{13}$$

The boundary condition is only applied at the global root grid level, which gives it a cost proportional to the domain circumference, and a cost W_B per boundary point.

$$T_{boundary} = 2(X + Y) \cdot W_B \tag{14}$$

Adaptive Mesh Refinement Cost T_{AMR}

The AMR step of the control flow diagram in Fig. 5a includes recursive integration steps for each level of refinement. For performance modeling purposes we account for this cost in Eq. 13, which reduces the expression of AMR cost to the terms T_R for regridding, T_{GB} for grid border handling, and T_D for downsampling, as in Eq. 15.

$$T_{AMR} = T_R + T_{GB} + T_D \tag{15}$$

Re-evaluations of the current grid decomposition are not generally required at every solver iteration, which makes the total cost of regridding parametric in terms of a *regridding interval RI*. The aggregate cost is accumulated over a full set of subgrids as for the SWE solver cost, but we can restrict the set of subgrids to those that are parent grids in the hierarchy at some time step, written as G_p in Eq. 16. W_R denotes the time required to evaluate a point with respect to a chosen refinement criterion.

$$T_R = \frac{\sum_{g \in G_p} (X_g \cdot Y_g)}{RI} \cdot W_R \tag{16}$$

As with the global domain boundary, the cost of managing grid borders is proportional to their circumference, and a cost W_{GB} interpolating a boundary point. The total sum must, however, be aggregated from all subgrids as in Eq. 17.

$$T_{GB} = \sum_{g \in G} 2(X_g + Y_g) \cdot W_{GB} \tag{17}$$

Downsampling only applies to the set G_c of subgrids that are children of a parent grid. The subset of points in a child grid that coincide with points in its parent depends on the degree of refinement r between their respective resolutions, as expressed in Eq. 18.

$$T_D = \frac{\sum_{g \in G_c}(2 \cdot X_g \cdot Y_g)}{r^3} \cdot W_D \tag{18}$$

Communication Cost $T_{exchange}$

The total cost of communication is proportional to transmitted data volume, and inversely proportional to network bandwidth. The data volume at each rank is determined by the circumference of its assigned grid partition, which can be expressed as Eq. 19.

$$T_{exchange} = P_{RB} \cdot W_{RB} \tag{19}$$

P_{RB} is the number of grid points in a rank's circumference, and W_{RB} is the average time to exchange one point. With the root grid partitioned in a Cartesian split, individual rank circumferences differ by at most 1 point along each axis. This suggests that differences in P_{RB} must be marginal compared to its magnitude, so we expect Eq. 19 to estimate $T_{exchange}$ for all ranks. When investigating weak scaling, the experimental setup purposely increases the size of the global domain such that it creates equally sized partitions for each participating rank, regardless of the degree of parallelism. In this setting, P_{RB} will be a constant by design, and W_{RB} will depend on network connectivity between neighboring ranks only. Thus, we expect $T_{exchange}$ to be a constant overhead for practical purposes. With respect to parallel scalability, it should be noted that this assumption is inaccurate for very low degrees of parallelism. Specifically, partitioning the global domain between 2 ranks creates only 1 border, 4 ranks share only 2 borders pairwise as illustrated in Fig. 5, *etc.* Beginning with 9 ranks in a 3×3 Cartesian split, the completion of the border exchange will be bound by at least 1 rank with neighbors in all cardinal directions. We assume that further upscaling will be in increments which partition the domain similarly.

Derived Cost of an Additional Level of Refinement

Using our model expressions, we estimate the extra cost when the program decides to add a level of refinement. Eq. 13 dominates execution time in the limiting case, as it grows with the area of every subgrid in the computation. For simplicity, we estimate the refinement cost in terms of additional points to evaluate, and dispense with the hardware-dependent cost factors as they are constant at all levels in one run. The number of points p_L in a region at grid level L cannot exceed the number of points p_{L-1} obtained by refining the resolution of its entire parent grid by a factor r.

$$p_L \leq r^3 \cdot p_{L-1}$$

This is a recursive relation, so an additional level expands as

$$p_L \leq r^{3 \cdot 2} \cdot p_{L-2}$$

	Vilje	Idun
Processor	Intel Xeon E5-2670	Intel Xeon E5-2630 v2
Compiler	ICC 18.0.1	ICC 19.0.5.281
MPI	SGI MPT 2.14	Intel MPI 2018 5.288
OpenMP version	4.5	4.5
Sockets / node	2	2
Cores / socket	8	10
Memory / node	32GB	128GB

Table 1: Hardware configurations of the test platforms

or in general,

$$p_L \leq r^{3 \cdot L} p_{root}$$

The upper bound for the sum of points P_L in all refinement levels is a geometric series:

$$P_L \leq p_{root} \sum_{l}^{L} r^{3L}$$

$$P_L \leq p_{root} \frac{1 - r^{3L}}{1 - r^3}$$

The ratio of points between runs with and without an extra level of refinement becomes

$$\frac{P_L}{P_{L-1}} = \frac{p_{root} \cdot \frac{1-r^{3L}}{1-r^3}}{p_{root} \cdot \frac{1-r^{3(L-1)}}{1-r^3}} = \frac{1 - r^{3L}}{1 - r^3}$$

Accounting for arbitrarily many refinement degrees and levels, we find that

$$\lim_{L \to \infty} \frac{1 - r^{3L}}{1 - r^{3(L-1)}} = \lim_{r \to \infty} \frac{1 - r^{3L}}{1 - r^{3(L-1)}} = \frac{r^{3L}}{r^{3(L-1)}} = r^3$$

Observing that an additional level of refinement can not *reduce* the computational workload, we obtain the inequalities in Eq. 20 as lower and upper bounds of the factor P_{refine} by which run time increases with an additional level of refinement.

$$1 \leq P_{refine} \leq r^3 \tag{20}$$

6 Experimental Results and Discussion

Our experiments are carried out on two computing clusters, *Vilje* and *Idun*. Vilje is part of the Norwegian national infrastructure for scientific computing, and Idun is a local computing resource operated by NTNU. The technical specifications of their node architectures are summarized in Tab. 1.

Experimental Setup

We validate the performance model presented in Section 5 using the two fluid configurations shown in Figs. 6 and 7. Fig. 6 shows a wave initialized at the edge of the domain, which passes across it and hits the far side. Fig. 7 shows a wave initialized in the middle of the domain, which collapses and splits in opposite directions. We present three experiments using these waves, varying domain partitioning to produce specific effects.

(a) Initial state (b) State after crossing domain

Figure 6: Wave traverses the domain from the edge

(a) Initial state (b) State after split

Figure 7: Wave splits in the middle of the domain

Weak Scalability

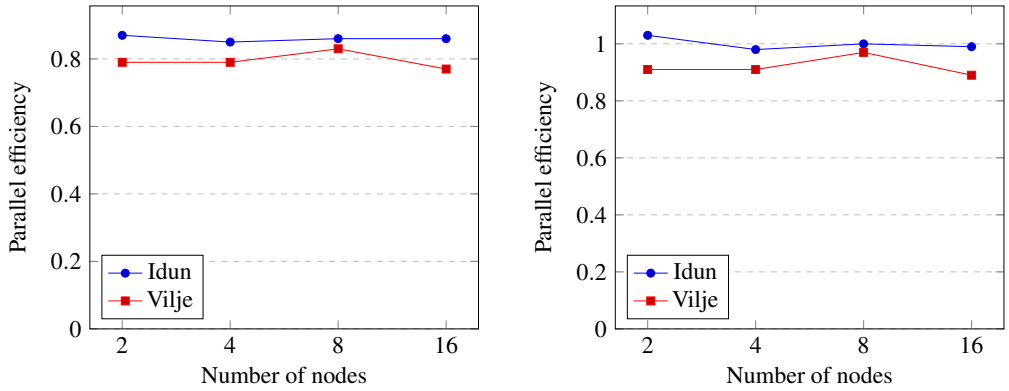
Our first experiment obtains a baseline scalability result that provides context for later variations. To keep the computational load directly proportional to the size of the computing platform, we simulate the edge wave scenario, and only partition the domain longitudinally to the wave. Each node simulates an equally sized rectangle with the same balance between wavefront and placid fluid. We maintain an identical workload for p nodes by setting global domain size to $512 \times 512p$ points, and measure *parallel efficiency* $E(p)$, as a function of p:

$$E(p) = \frac{S(p)}{p} \qquad S(p) = \frac{T_1}{T_p}$$

where T_1 and T_p are run time measurements using 1 and p nodes, respectively. The optimal value of this metric is $E(p) = 1$, reflecting that simultaneous increases in system size and workload maintain constant time. Fig. 8b shows the efficiency figures obtained from both platforms, both in terms of total time spent on computation and communication, and as measured by the cost of computation only. While there are minor variations from the theoretical optimum of a completely level curve, these can be attributed to minor fluctuations in run time conditions and timing accuracy. Noting that the curves span a range of scales from 20 through 320 parallel processing cores on Idun, and 16 through 256 on Vilje, it is evident that with an evenly partitioned workload, program performance scales practically linearly with system size across an order of magnitude difference.

Impact of Increasing Levels of Refinement

Our second experiment verifies the refinement cost bounds derived in Section 5. We isolate the effect on computation time in a series of runs using 1 node only, thus eliminating communication time. A series of edge wave scenarios capping the number of refinement levels from 2 to 4 are carried out with a doubling of the resolution at each level, *i.e.* a refinement degree $r = 2$, so that $1 \leq P_{refine} \leq 8$. Fig. 9 shows that measurements of the grid point growth caused by successive levels all lie within the predicted range.



(a) Computation and communication (b) Computation only

Figure 8: Weak scaling: Parallel efficiency relative to 1 node

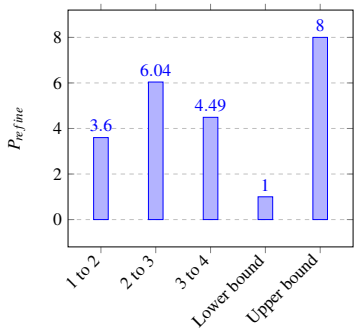


Figure 9: Problem size growth factors caused by successively adding levels of refinement

Impact of Load Balance

Our final experiment measures the effect of developing different refinement levels in domain areas assigned to different nodes. The expected effect is that nodes with a greater workload will complete each time step later than their neighbors, and since the next time step will not proceed until all participants complete the previous one, collective performance will be limited by the performance of the node with the highest load. We examine the speedup metric in strong scaling mode, *i.e.* measuring a fixed size problem over increasing node counts, so that upscaling reduces the subdomain size assigned to each node. We partition the domain in rectangular sections along both directions, and

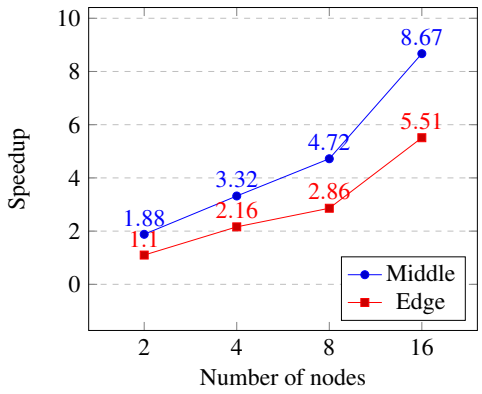


Figure 10: Strong scaling of total run time: Speedup relative to 1 node

measure run time for both the edge and middle wave scenarios. This creates a data-dependent load imbalance centered around the wavefronts, because changes in fluid level trigger regridding and refinement in the subdomains where they occur. The middle wave produces better scalability characteristics than the edge wave, because the two wavefronts of the former are distributed simultaneously across the domain, while the single wavefront of the latter places additional workload only on the nodes that handle the area it is presently traversing. Fig. 10 shows speedup relative to single-node time when distributing 6553600-point domains on 2 through 16 nodes on the Vilje platform. It is quite clear that the balance of the computational load has a substantial impact on the efficiency of the computation, as the maximal speedup figures are down to 54% and 34% of the optimal linear speedup, respectively. Part of this effect is due to Amdahl's Law [10] as we are operating in strong scaling mode, but the distinction between the two wave patterns clearly demonstrates that the dynamic distribution of the fluid has a direct impact on program performance. This suggests that adaptive mesh refinement should be applied with an acute awareness of how the fluid interacts with domain conditions.

7 Related Work

Berger [11] compares the computation time between AMR, coarse, and fine solution for four example problems. The article provides a formula for how often regridding should be performed in order to minimize the cost of the algorithm, given as a balance between the cost of integration and regridding. Cost estimates for integration and for regridding are needed to use the formula. Jameson compares lower-order AMR schemes to higher-order non-AMR schemes, and finds that the former are computationally more expensive than the latter when the goal is to reduce the error of the solution to a certain amount [12]. As the integration part suffices to show that lower-order AMR schemes have a higher computational cost than higher-order non-AMR schemes, the article omits analysis of workload and computational costs of other AMR steps. Erduran *et al.* evaluate the performance of finite volume solutions to the shallow water equations [13]. Kubatko *et al.* study the performance of two finite element methods [1], and compare the cost to finite difference methods. Sætra *et al.* implement the SWE using a finite-volume method and AMR on the GPU [14], and measure accuracy and performance for some examples.

8 Conclusions and Future Work

In this paper, we have described the design and implementation of a numerical solver with adaptive mesh refinement for the Shallow Water Equations. We have developed a performance model of the program, made predictions about three important scalability characteristics, and verified them experimentally on two high performance computing platforms. Our empirical observations show that the application exhibits close to linear scalability in weak scaling mode, given constant workload per node. We also derived upper and lower bounds for the additional workload of adaptive mesh refinement, and validated them experimentally. Finally, we observed that adaptive mesh refinement creates a load balancing issue which depends on the distribution of fluid in the simulated system, and found that it reduced parallel efficiency to 54% and 34% of the optimum in two experiments designed to produce the effect. In summary, we have found that the application is highly scalable, but that its continued efficiency at larger scales depends on an even distribution of refined areas throughout the domain. A load balancing mechanism to dynamically redistribute subgrids is an interesting direction for further research.

Acknowledgments

The experiments were performed using the NTNU IDUN/EPIC computing cluster, and on resources provided by UNINETT Sigma2 - the National Infrastructure for High Performance Computing and Data Storage in Norway.

References

[1] E. J. Kubatko, S. Bunya, C. Dawson, J. J. Westerink, and C. Mirabito, *A performance comparison of continuous and discontinuous finite element shallow water models*, Journal of Scientific Computing, Vol. 40, No. 1-3, pp. 315–339, 2009.

[2] R. MacCormack, *The effect of viscosity in hypervelocity impact cratering*, Journal of spacecraft and rockets, Vol. 40, No. 5, pp. 757–763, 2003.

[3] W. Cheney, D. Kincaid, *Numerical Mathematics and Computing*, 4th ed., Brooks/Cole publishing company, 1999

[4] R. Courant, K. Friedrichs, and H. Lewy, *On the partial difference equations of mathematical physics*, IBM journal of Research and Development, Vol. 11, No. 2, pp. 215–234, 1967.

[5] M. Berger and I. Rigoutsos, *An algorithm for point clustering and grid generation*, IEEE Transactions on Systems, Man, and Cybernetics, Vol. 21, No. 5, 1991.

[6] B. Wilkinson, M. Allen, *Parallel Programming*, 2nd ed., Pearson Prentice Hall, 2005.

[7] L. G. Valiant, *A bridging model for parallel computation*, Communications of the ACM, Vol. 33, No. 8, 1990.

[8] P. S. Pacheco, *An introduction to parallel programming*, Morgan Kaufmann Publishers, 2011.

[9] K. J. Barker, K. Davis, A. Hoisie, D. J. Kerbyson, M. Lang, S. Pakin, and J. C. Sancho, *Using Performance Modeling to Design Large-Scale Systems*, IEEE Computer, Vol. 42, No. 11, pp. 42–49, 2009

[10] G. M. Amdahl, *Validity of the single processor approach to achieving large scale computing capabilities*, Proceedings of the April 18-20, 1967, spring joint computer conference, 1967, pp. 483–485.

[11] M. J. Berger, *Adaptive mesh refinement for hyperbolic partial differential equations*, Tech. Report No. STAN-CS-KL-924, Stanford University CA, 1982

[12] L. Jameson, *AMR vs. higher order schemes*, Journal of Scientific Computing, Vol. 18, No. 1, 2003, pp. 1–24

[13] K. S. Erduran, V. Kutija, C.J.M. Hewett, *Performance of finite volume solutions to the shallow water equations with shock-capturing schemes*, Intl. Journal for Numerical Methods in Fluids, Vol. 40, No. 10, 2002, pp. 1237–1273

[14] M. L. Sætra, A. R. Brodtkorb, K.A. Lie, *Efficient GPU-implementation of adaptive mesh refinement for the shallow-water equations*, Journal of Scientific Computing, Vol. 63, No. 1, 2015, pp. 23–48