Costs and Savings of Adaptive Mesh Refinement

Ragnhild Blikberg*
Dept. of Informatics/Parallab,
University of Bergen,
NORWAY

Tor Sørevik†
Dept. of Informatics,
University of Bergen,
NORWAY

November 26, 2001

Abstract

This paper is concerned with the quality of AMRCLAW [BL97], a software package for adaptive mesh refinement (AMR) applied to conservation laws. The quality is a multi facet feature and we focus on the 3 following issues:

1. Does AMRCLAW succeed in maintaining nice properties of CLAWPACK [LeV95] such as conservation of mass and energy and correct shock speed?
2. Does the AMR create unacceptable additional error compared to a solution on a uniform fine grid?
3. What is the extra cost needed to do error estimation, flux correction and coarse grid fix up, and when does it outweigh the saving of mesh refinement?

We try to answer these questions by a mixture of analysis, experiments and modeling. For the computational cost we estimate values for the parameters involved in the complexity formulas using data from the experiments. This permits us to quite precisely compute the ratio of refinement needed to guaranty savings when using AMR.

Keywords: Adaptive Mesh Refinement, AMRCLAW, Shallow Water Equations

1 Introduction

When solving partial differential equations (PDE) numerically in real applications, the need for dense grid points may differ a lot from one region of the solution domain to another, as well as from one point in time to another for the same sub-domain. An example of this may be ocean modeling where the need for fine resolution may be larger close to the coast, a stream, or a moving obstacle than elsewhere. To attain a good accuracy on a uniform grid in such cases, it is necessary to have the finest resolution

*`http://www.ii.uib.no/~ragnhild`
†`http://www.ii.uib.no/~tors`

needed in the entire domain. This will no doubt cause more computational work to be done than necessary in those parts of the domain where a coarse grid is sufficient.

In AMR, grid cells in regions where mesh refinement is most needed are being refined. These are clustered together in grid patches which dynamically is created or changed based upon estimates of the error for the individual cells. The technique is applied recursively, such that fine grids can contain even finer grids, and the nesting continues until a given level of accuracy is attained. In this manner, one can have a fine resolution where it is needed and use a coarse resolution where this is sufficient.

However, when applying mesh refinement automatically in an adaptive fashion, great care has to be taken to make sure all cells in need for refinement are being refined. Furthermore, it is crucial that the use of non-uniform grid does not violate mathematical properties of the underlying numerical scheme such as being conservative. Dealing with these obstacles implies additional computation.

In this report we will give some indication on how well the freely available software package AMRCLAW [BL97] does in dealing with the two problems mentioned above; 1) Preserve conservation properties of the numerical scheme, and 2) Refine all cells in need for refinement. But our focus will be on the price of this additional cost and whether or not it outweighs the savings in reducing the grid points.

This analysis will be based on the results of two numerical test cases, where AMRCLAW is used on the shallow water equations (SWE). We also develop a computational model for this computation and estimate values for the different coefficients by fitting our computational results using least square approximation. The models are than used to predict the potential savings of AMRCLAW for different scenarios.

A short summary of the AMR algorithm will be presented in Section 2. Next our two test cases, a straight wave and a radial propagated wave, will be presented in Section 3, where also the experimental performance results of our two test cases using AMRCLAW on the SWE will be reported on. In Section 4 a theoretical model for the computational work of AMR will be developed, and the extra cost and savings of AMR will be studied. Finally the conclusions and thoughts about future work will be given in Section 5.

2 The AMR algorithm

In this section we will give a short summary of the AMR algorithm, developed by Berger and Oliger [BO84]. More details can be found in [BC84, BL98]. AMR solves time dependent PDE using a dynamically changing hierarchy of nested rectangular Cartesian grids with finer and finer resolution, up to a user defined maximum level of refinement NL. Each level L is a union of grid patches of a specific resolution.

Every K time step on a particular grid level, all grids except for the coarsest grid are regenerated in order to reflect changing flow features. An error estimation procedure based on Richardson extrapolation determines the cells where resolution is insufficient, by comparing the solutions obtained by taking 2 time steps on the existing grid to that obtained by taking 1 time step on a grid that is twice as coarse in each direction. Cells where the error is greater than some user defined tolerance tol, are flagged for refinement. In addition to the Richardson error estimation, cells are flagged for refinement if a spatial gradient estimate exceeds a tolerance *tolsp*.

To ensure that interesting features do not leave the refinement region over the next K time steps, a buffer zone of b cells around the cells exceeding the error tolerance is also flagged for refinement. Additional cells are also refined in order to maintain a rectangular structure of refined patches. The more cells we are willing to allow, the larger patches we will get. This is controlled by a user defined parameter *cutoff*. With a typical *cutoff* of 0.7, at least 70% of the refined cells are initially tagged for refinement. This refinement procedure is applied recursively until a maximum number *NL* of refinement.

Boundary conditions at all levels are provided by extending the grid patch by G "ghost cells" in a band around the patch. Values are assigned to the ghost cells at the start of each time step. If these values are available from grids at the same level of refinement, they are provided by a simple copy. But if there are no neighboring fine grid to supply values for the ghost cells, they are interpolated from a coarser parent grid, unless they coincide with the physical boarder of the computational domain, in which case standard boundary conditions are used. Interpolation of the coarse grid data has to be performed in time as well as in space since more time steps are taken on the fine grid than on the coarse grid, and at intermediate times there are no coarse grid values available.

A finite volume method is used to advance the solution in time on all levels of the resulting grid hierarchy. Cell averages of each variable are stored in the center of each grid cell, and are updated by a flux-differencing algorithm based on fluxes through the cell edges. CLAWPACK [LeV95] provides different schemes for time integration. In our test cases we have chosen a 2nd order scheme. For a specific time interval, all cells at level L are integrated before any cells at level $L+1$. The cells are refined by a factor R_{L+1} and integrated R_{L+1} time steps, with step length $\Delta T_{L+1} = \Delta T_L / R_{L+1}$ to catch up.

When the solutions on coarse and fine grids reach the same time, two corrections have to be made. First, for all coarse grid cells covered by fine grid cells the coarse grid data has to be replaced by the volume weighted average of the fine grid data. Second, since coarse cells and adjacent fine cells where advanced using different fluxes, a conservative flux correction is required to maintain conservation at grid interfaces.

The AMR algorithm can be written as in Algorithm 1 below. In this algorithm $R_1 = 1$.

Algorithm 1: Recursive AMR

/* *In this algorithm* Ω_L^i *represents grid patch* i *at level* L*.*
$\Omega_L = \cup_{i=1}^{P_L} \Omega_L^i$, P_L *is the number of patches at level* L */
AMR(L, Ω_L, k_L)
    for $T_L = 1, R_L$
        if ($k_L = 0$ and $L < NL$) **then** /* *Regrid* */
            Estimate_error(Ω_L) **(R)**
            Flag cells with *error* $>$ *tol* and cells in buffer zone **(R)**
            Organize cells into patches: Ω_{L+1}^i; $i = 1, \cdots, P_{L+1}$ **(R)**
        end if
        Compute_boundary_values$(\partial\Omega_L)$ **(B)**
        Advance_PDE_1_time_step(Ω_L) **(A)**
        $k_L = (k_L + 1) \bmod K$

if $(\Omega_{L+1} \neq \emptyset)$ **AMR**$(L+1, \Omega_{L+1}, k_{L+1})$
end for
Coarse_grid_update$(\Omega_{L-1} \cap \Omega_L)$ **(U)**
Conservation_fixup$(\Omega_{L-1} \cap \partial\Omega_L)$ **(F)**

The main program then looks like:

Algorithm 2: PDEsolver

$L = 1$; $k_{1:NL} = 0$; $\Omega_L =$ initial grid
for $T_L = 1, NT$
AMR(L, Ω_L, k_L)
end for

3 Numerical experiments

3.1 The test problem

Our test problem for AMRCLAW [BL97] is SWE [LeV98]. In this version of SWE we assume there is no bottom friction, diffusion or viscosity. The SWE can than be written as:

$$h_t + (hu)_x + (hv)_y = 0 \tag{1}$$

$$(hu)_t + \left(hu^2 + \frac{1}{2}gh^2\right)_x + (huv)_y = -ghB_x \tag{2}$$

$$(hv)_t + (huv)_x + \left(hv^2 + \frac{1}{2}gh^2\right)_y = -ghB_y \tag{3}$$

where $B(x, y)$ is the bottom elevation, $h(x, y, t)$ the fluid depth above this bottom, and $u(x, y, t)$ and $v(x, y, t)$ the velocity in x- and y- direction respectively.

The boundary is solid, with "no-slip" conditions, implying the following von-Neumann boundary conditions

$$\frac{\partial u}{\partial n} = 0, \quad \frac{\partial v}{\partial n} = 0, \quad \frac{\partial h}{\partial n} = 0 \qquad \text{on} \qquad \partial\Omega. \tag{4}$$

3.2 The test cases

In this section the results from two different test cases will be presented. Both take place in a square 20km $\times$ 20km basin with flat bottom topography, $B = 0$, filled with water, giving a problem domain $\Omega = [0, 20] \times [0, 20]$. The boundary conditions are given by (4). The difference in the two test cases is only initialization.

For both test cases we have compared the performance of a 1-level uniform 256×256 grid case with a 2-level grid case, resolution 128×128 at the coarsest grid, and a 3-level grid case, resolution 64×64 at the coarsest grid. For all tests we have set $R_2 = R_3 = 2$, giving the same resolution at the finest level for all grid cases. We set the other user defined parameters to $b = 2$, $G = 2$, $cutoff = 0.7$, and vary $NL \in \{2, 3\}$ and $K \in \{1, 2, 3\}$, while tol is used as the "argument" taking values: $tol = \{0.01 : 0.01 : 0.1\} = I_{tol}$. The spatial tolerance for refinement $tolsp$ is set equal to tol. In the rest of the paper, both tolerances will be referred to as tol.

Test case 1: Straight wave

In this test case the initial conditions are given by

$$u(x, y, 0) = 0, \quad v(x, y, 0) = 0, \quad h(x, y, 0) = \begin{cases} 201\text{m} & \text{for } x \leq 0.625\text{km} \\ 200\text{m} & \text{for } x > 0.625\text{km} \end{cases} \tag{5}$$

This initialization results in a simple straight wave, propagating from the left to the right where it is reflected at the boundary, returning with the same shape and velocity (One period, the wave propagating to the wall and back again, takes approximately 900s.).

In Figure 1 we display snapshots of an AMRCLAW simulation of this test case.

Test case 2: Radial propagated wave

For the second case the initial conditions are given by

$$u(x, y, 0) = 0, \quad v(x, y, 0) = 0, \quad h(x, y, 0) = \begin{cases} 201\text{m} & \text{for } x \leq 5\text{km} \ \& \ y \leq 5\text{km} \\ 200\text{m} & \text{elsewhere} \end{cases} \tag{6}$$

In this case a wave is propagating radially from one corner. When reflected at the boundaries a rather messy behavior occurs as it hits the boundary under different angles. Snapshots of this case is presented in Figure 2.

While the wave in test case 1 stays concentrated and keeps a sharp front through out the simulation, the initial sharp front in test case 2 breaks down as the wave is reflected. The difficulties are being spread all across the computational domain. This is the crucial differences in the two seemingly "almost equal" test cases, which explains the different behavior of AMRCLAW when used here.

3.3 Some general observations

First of all we observed that the very important features due to the conservative flux differencing scheme from CLAWPACK [LeV95] is preserved in the AMR scheme. These features are

- No phase displacement. (i.e. The wave moves with the correct speed.)
- No artificial oscillations near shock.
- Conservation is maintained.

Like all other schemes CLAWPACK also produces some numerical diffusion, which tends to smooth sharp shocks. This can be seen in Figure 1, where the wave gets smoother and smoother as the time increases. In the adaptive context this diffusivity seems to increase

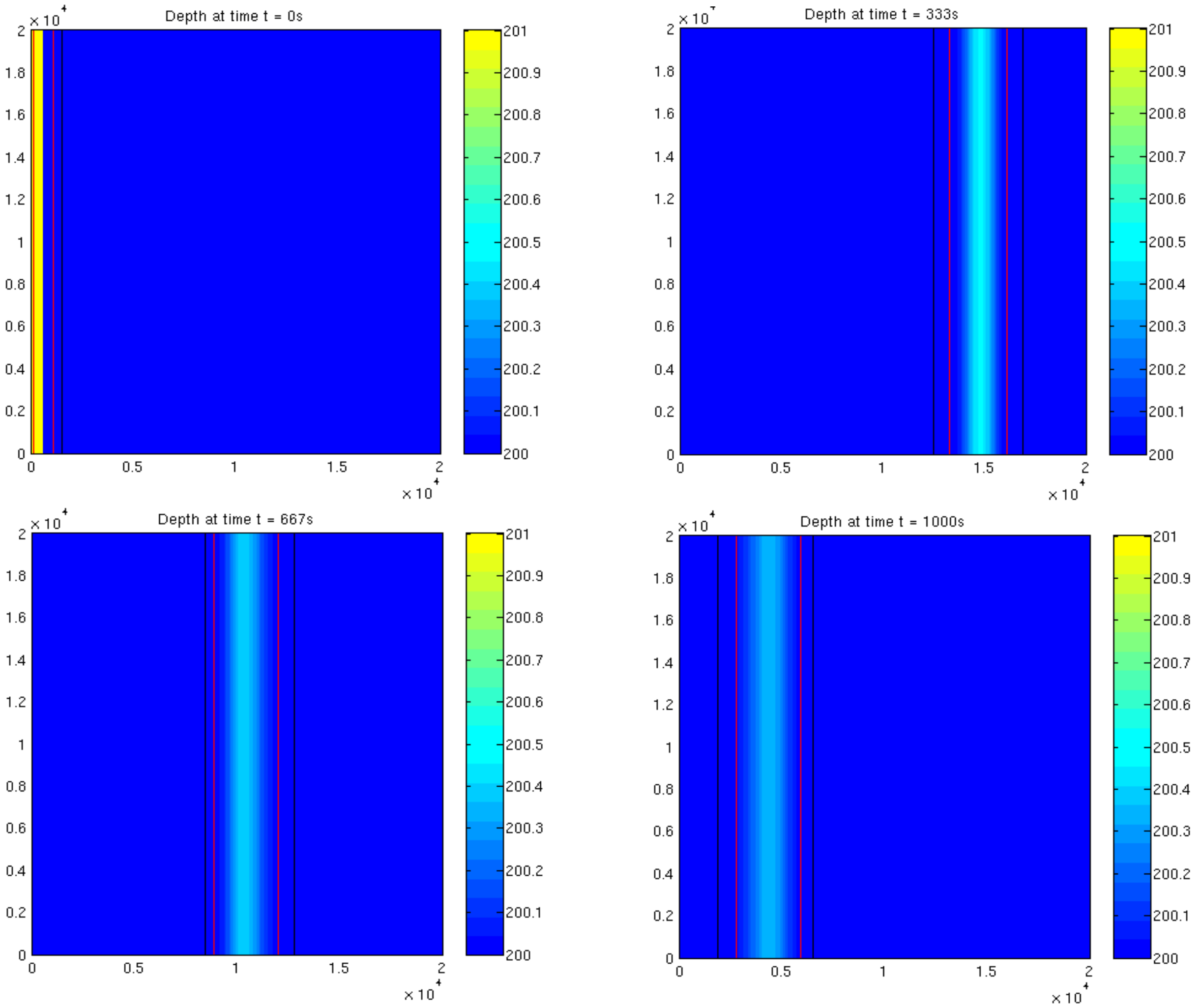

Figure 1: *Test case 1: 3-level* 64×64 *cells at coarsest grid. Initial depth and depth at* $t = 333s$, $t = 667s$ *and* $t = 1000s$ *for* $K = 1$ *and* $tol = 0.06$. *Boundaries of patches at level 2 are colored black, and boundaries of patches at level 3 are colored red.*

as *tol* increases. We believe the reason for this to be that increasing *tol* means that more cells gets refined, and a fine grid causes less diffusion than a coarse grid.

Initially the two test cases behave similar. A wave with a sharp front is propagating through the domain. In test case 1 the form of the wave is kept after reflection at boundary, while the simple form breaks down after reflection in test case 2, with the inevitable result that almost none smooth regions exist after repeated reflections. When this state is reached, the same grid level is needed in most parts of the domain. Which grid level is needed, is determined by the user provided tolerance parameter, *tol*.

For this simple test case the need for uniform grid in long running simulation, could have been predicted in advance, in which case using AMR should be avoided as it produces nothing but extra overhead. Thus, whether or not it is beneficial to use AMR for test case 2 depends on how far we are running the simulation in time. In our case non-uniform mesh made sense to use until $t = 500$s.

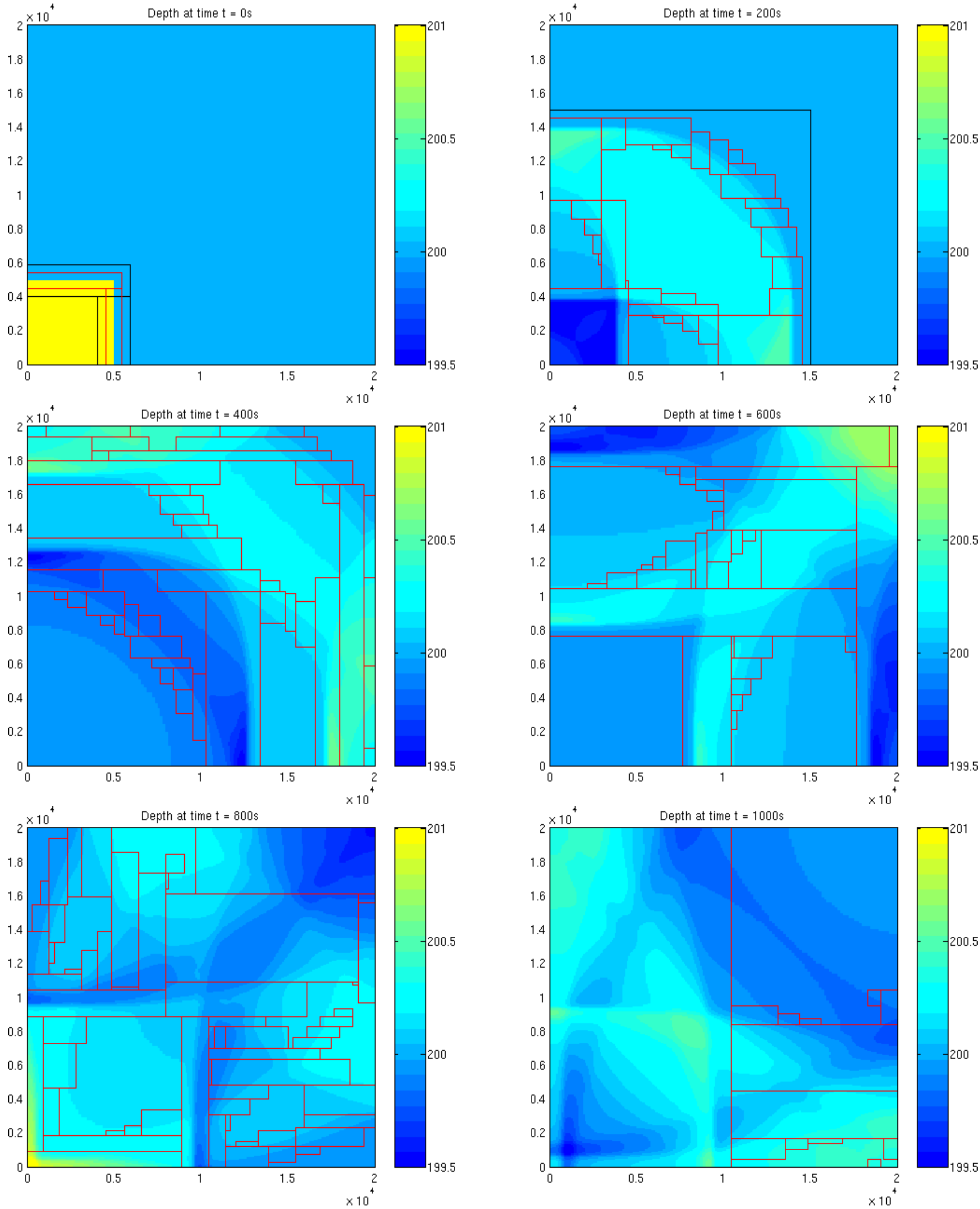

Figure 2: *Test case 2: 3-level,* 64×64 *cells at coarsest grid level. Depth at* $t = 0$, $t = 200s$, $t = 400s$, $t = 600s$, $t = 800s$ *and* $t = 1000s$ *for* $K = 1$ *and* $tol = 0.02$. *Boundaries of patches at level 2 are colored black, and boundaries of patches at level 3 are colored red.*

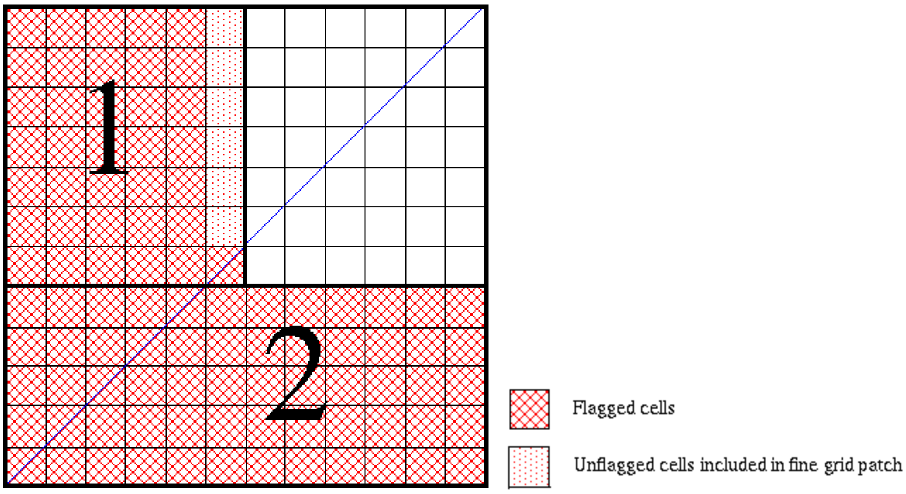

Figure 3: *Test case 2: flagged cells, unflagged cells included in fine grid patch, and two grid patches unsymmetric around the "SW-NE" diagonal.*

We also observed a special phenomenon with test case 2. Although the wave is symmetric around the "SW-NE" diagonal, the refined grid patches were not symmetric around this diagonal. This phenomenon can be seen in Figure 2, but is highlighted in Figure 3. Here the cells flagged for refinement are colored red checkered. When flagged cells are to be collected into rectangular patches, additional cells, red dotted, are added in order to avoid to many small patches. While the original flagged cells were distributed symmetrical around the "SW-NE" diagonal, the extra added cells are not, thus the pattern of the patches becomes unsymmetric around the "SW-NE" diagonal as shown in Figure 3.

As a result corresponding updates on symmetrically placed cells may not get equal corrections and thus the approximated solution shows an unsymmetry, not present in the real solution. This feature is not due to the underlying numerics, but to the way AMR adds extra cells to get rectangular patches and how the cells are grouped into patches for refinement. This will not produce a large degree of unsymmetry, but it can be enough to cause trouble when debugging if one is not aware of the phenomenon.

3.4 Results of the numerical experiments

In this section we present some numerical results for the efficiency of AMRCLAW. All tests are run on a partly loaded SGI Origin 2000, and all tests will be done for both test cases; the straight wave and the radial propagated wave. The straight wave, test case 1, is run until $t = 1000$s, while the radial propagated wave, test case 2, is not run longer than to just after the wave hits the boundaries, until $t = 500$s. Both cases are run for the six possible combinations of $NL \in \{2, 3\}$ and $K \in \{1, 2, 3\}$, and for $tol \in I_{tol}$.

Our yardstick for performance is a measure which we call *speedup*. This is defined as the CPU-time for a multilevel grid case divided by the CPU-time for the 1-level uniform fine grid case. In Figure 4 we display the speedup as a function of tol for the combination of NL and K for test case 1 and 2.

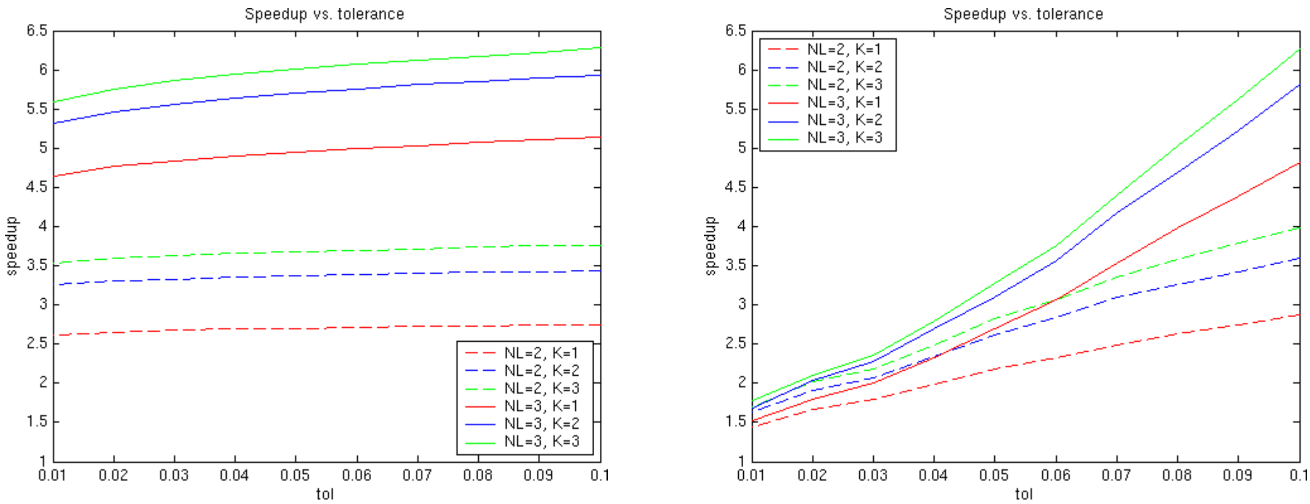

Figure 4: *Speedup versus tolerance for NL* $\in \{2, 3\}$ *and* $K \in \{1, 2, 3\}$*. Left: test case 1. Right: test case 2.*

As expected for both test cases and all possible combinations of NL and K, the speedup increases with tol. This is an obvious consequence of the fact that less cells are refined as tol increases, and fewer grid cells gives less computational work. The reason why test case 1 has a better speedup then test case 2 for the smaller values of tol, is that the number of refined grid cells is much smaller for test case 1 than for test case 2 for these choices of tol.

Not very surprisingly, one can also observe for both test cases that the speedup increases as K increases. This since the extra work related to the regridding decreases as K increases. The difference in speedup between $K = 1$ and $K = 2$ is about 1.5 times larger than the difference in speedup between $K = 2$ and $K = 3$. This comes naturally since the time spent on regridding for $K = 1$ is twice as large as for $K = 2$, and three times as large as for $K = 3$. The time spent on regridding for $K = 2$ is then $3/2$ times as large as for $K = 3$.

At the heart of the AMR algorithm is the requirement that all cells with error less than tol should be refined. Since the exact error is not known this will have to be estimated. If the estimate is smaller than the true error, cells which should be refined will not necessarily be flagged for refinement. If the error estimate is larger than the true error, too many cells may be flagged for refinement. The first situation may lead to unacceptable inaccuracy in the solution, while the second may lead to unwanted extra cost.

As an indicator of the error we have computed the maximum difference between the solution on a uniform grid and the AMRCLAW solution

$$E_{est} = \parallel h_{UNI} - h_{AMR} \parallel_\infty . \tag{7}$$

E_{est} should be interpreted as an upper bound on the additional error of using AMR as opposed to solving on a uniform fine grid. Taking tol as a measure for the acceptable error we want (7) always to be smaller than tol. In Figure 5 we have plotted E_{est} at the arrival time, against tol for the various test cases. We note that in all cases $E_{est} < tol$, thus we conclude that in our cases AMR only introduce minor additional errors.

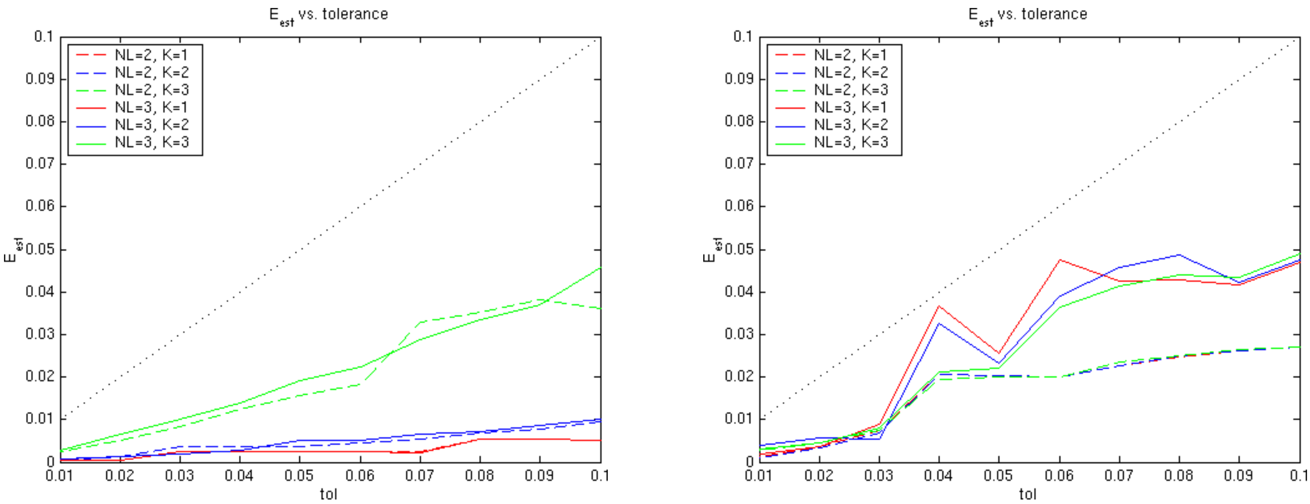

Figure 5: *E_{est} versus tolerance for $NL \in \{2, 3\}$ and $K \in \{1, 2, 3\}$. Left: test case 1. Right: test case 2 .*

For test case 1 one can observe that E_{est} is quite similar for the the 2-level and 3-level for given tol and K. Figure 5 shows something different for test case 2. For $NL = 2$ and $NL = 3$ there are only minor differences in accuracy between $K = 1, 2$ or 3. Since $K = 3$ has less extra cost and there are no large differences in accuracy for the different Ks, this value of K will be preferable.

Above we defined speedup as a measurement for the net savings in computational effort using AMRCLAW. The positive contribution to the net saving stems from the fact that using mesh refinement less grid cells are needed, but there is also a negative contribution due to extra cost in terms of regeneration of grids, updating of coarse grid, conservation fix-up, etc. To get a feeling for the size of these negative terms we define *efficiency*, which is simply the ratio between the computing time pr. grid point for a uniform fine grid and the computing time pr. grid point in AMRCLAW.

In Figure 6 the efficiency is plotted as a function of tol for our two test cases and the 6 choice of NL and K. An interesting observation is that for both test cases (except for test case 2, $K = 3$) 3-level grid appears to be more efficient than 2-level grid for a fixed value of K. As expected $K = 3$ is more efficient than $K = 2$, and the number of grid points per CPU-second slightly increases as tol decreases.

4 Modeling the computational work of AMR

The tests presented in the previous section demonstrated how the speed and accuracy vary depending on the choice of the parameters such as NL, K and tol. To better understand when the use of AMR is beneficial, and give us an idea of how much there is to gain by using AMR, we will here develop a model for the computational work of AMR.

In the sequel we assume that advancing the solution one time step has equal cost on all grid points. There is two main modification to this assumption. The first is that there is a difference between boundary points and interior points. With multiple mesh levels

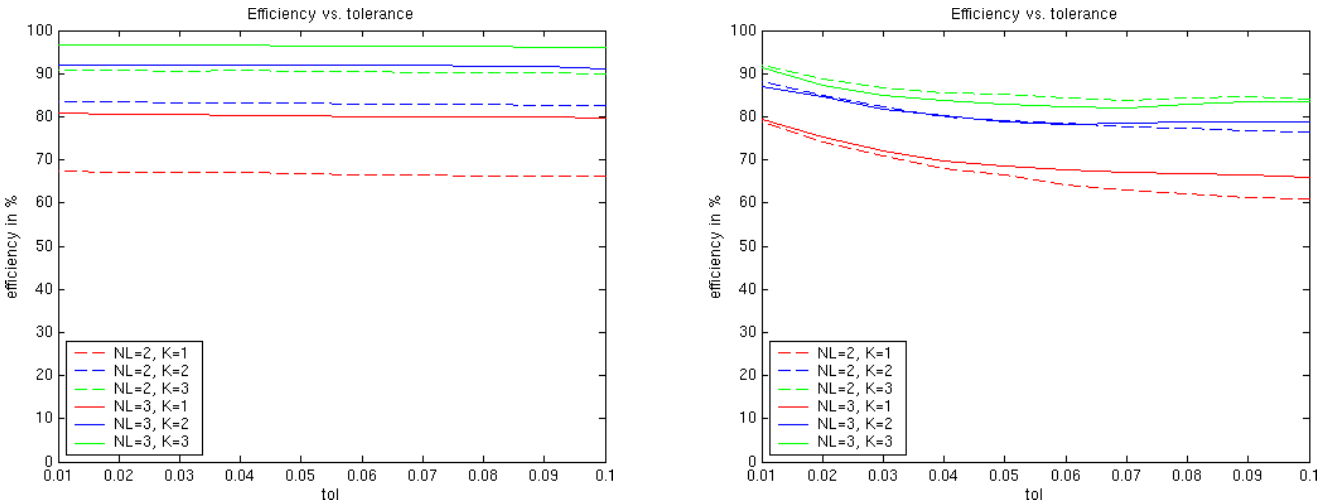

Figure 6: *CPU-time pr. grid point versus tolerance in percent compared to uniform fine grid for* $NL \in \{2, 3\}$ *and* $K \in \{1, 2, 3\}$*. Left: test case 1. Right: test case 2.*

we create artificial internal boundaries. Additional computation is sometimes needed for constructing these. This extra cost is incorporated in our model. The second is related to memory access. On todays hierarchical memory systems, the cost of performing a given sequence of numerical calculation is not just that of carrying out the floating point operation (the flops), but equally important is that of loading and storing data in memory. How this data movement effects the performance is most unclear and in practice very hard to predict as it depends not only on the problem, but system specific parameters as well. In multiuser system other users use of the system also influence on the data layout in memory. We make no attempt to model the memory behavior of the AMR algorithm compared to that of a uniform grid model, but will provide computational results which demonstrate its influence (See Figure 6).

4.1 A Theoretical model based on operation count

In this section we will develop a theoretical model for the computational work of AMR, based on operation count of the AMR algorithm as presented in Section 2. The different component included correspond to those marked **(A)**, **(R)**, **(U)** and **(F)** in Algorithm 1. The computational work of **(B)** will be ignored, since performance profiling shows that this component uses only negligible CPU-time.

We define $G_L(T_1, T_2, ..., T_L)$ to be the total number of grid points for all patches at coarse level time step T_1, 2nd level time step T_2, ..., and Lth level time step T_L. The total number of grid points when advancing from $T_1 = 1$ to $T_1 = NT$ then becomes:

$$G_L = \sum_{T_1=1}^{NT} \sum_{T_2=1}^{R_2} ... \sum_{T_L=1}^{R_L} G_L(T_1, T_2, ..., T_L). \tag{8}$$

In the following all formulas will in the general case be nested sums like (8), but for simplicity we have chosen to restrict ourself to the case $NL = 3$.

(A) Advancing the PDE in time:

For $NL = 3$ and NT number of coarse time steps, the total number of grid points used for the computation in the AMR case is

$$G_{AMR} = \sum_{T_1=1}^{NT} \left[G_1(T_1) + \sum_{T_2=1}^{R_2} \left[G_2(T_1, T_2) + \sum_{T_3=1}^{R_3} G_3(T_1, T_2, T_3) \right] \right]. \tag{9}$$

Using (8) gives

$$G_{AMR} = \sum_{L=1}^{3} G_L. \tag{10}$$

The number of total uniform fine grid points, where the density of points equals the density of the points at level 3 in the AMR case, is

$$G_{UNI} = \left[\prod_{L=2}^{3} R_L \right] \cdot \left[\prod_{L=2}^{3} R_L \right]^2 G_1 = \left[\prod_{L=2}^{3} R_L \right]^3 G_1. \tag{11}$$

In the reminder of this section we will model the additional cost incurred by the AMR routine as described in Section 4.1. **(U)** and **(C)** reflect cost associated with mesh refinements and will also be present in statically refined meshes, while **(R)** is associated with adaptivity.

For simplicity we assume $R_2 = R_3 = R$.

(R) Regeneration of grids:

Regeneration of grids represents the adaptive aspects of grid refinement, and includes error estimation, flagging and clustering of grid points. Thus in those cases where one knows in advance where the need for close grid point is, statically refined meshes should be chosen instead of adaptively refined meshes.

One should be aware of that even if the Richardson error estimation procedure compares the solutions obtained by taking 2 time steps on the existing grid to that obtained by taking 1 time step on a grid that is twice as coarse in each direction, the first time step on the existing grid is the identical time step as used in **(A)**, and that this time step is not repeated. Therefore, in practice, there is only taken 1 time step on the existing grid. The work associated with the spatial gradient estimates is very low compared to the Richardson error estimation and will be neglected in this model.

If all finer grids are regenerated at every K time step on a particular level, and we assume $NT \gg 1$, the extra number of grid points subject to computation of error estimation is given by

$$\begin{aligned} G_R &\approx \frac{1}{K} \sum_{T_1=1}^{NT} \left[\left(1 + \frac{1}{2^2}\right) G_1(T_1) + \sum_{T_2=1}^{R} \left[\left(1 + \frac{1}{2^2}\right) G_2(T_1, T_2) \right] \right] & (12) \\ &= \frac{1}{K} \frac{5}{4} \sum_{L=1}^{2} G_L. & (13) \end{aligned}$$

(U) Coarse grid update:

Where a grid is overlapped with a finer grid, its cells value should be updated by the cells values from the finer grid when these becomes available. This is done each time step on all levels $L < NL$. This update applies to all level-L cells in $\Omega_L \cap \Omega_{L+1}$. The number of these cells are $G_{L+1}(T_1, ..., T_L, R_{L+1})/R_{L+1}^2$. Assuming $R_2 = R_3 = R$, the total number of these updates in the 3-level case becomes:

$$
\begin{aligned}
G_U &= \sum_{T_1=1}^{NT} \left[\frac{1}{R^2} G_2(T_1, R) + \sum_{T_2=1}^{R} \frac{1}{R^2} G_3(T_1, T_2, R) \right] && (14)\\
&= \sum_{T_1=1}^{NT} \left[\frac{1}{R^2} \left(\frac{1}{R} \sum_{T_2=1}^{R} G_2(T_1, T_2) \right) + \sum_{T_2=1}^{R} \frac{1}{R^2} \left(\frac{1}{R} \sum_{T_3=1}^{R} G_3(T_1, T_2, T_3) \right) \right] && (15)\\
&= \frac{1}{R^3} \sum_{L=2}^{3} G_L. && (16)
\end{aligned}
$$

(F) Conservation fix-up:

To maintain conservation at the boundaries between grid levels, we must make sure that the flux out of (in to) a coarse grid cell equals the aggregated flux in to (out of) the neighboring fine grid cells. If a difference is found, this is added to coarse grid value, as a conservation fix up, which is done each time step at level $L < NL$. The number of grid cells will be the boundaries of these cells involved in coarse grid updates and thus analogously to (16) we get:

$$G_F = \frac{1}{R^3} \sum_{L=2}^{3} \partial G_L. \tag{17}$$

A model for the total computational work:

Under the reasonable assumption that all the unit operations in **(A)**, **(R)**, **(U)** and **(F)** have equal cost, we arrive at the following formula for the computational work of AMRCLAW:

$$W_{AMR} = c_{flop} G_{AMR} + c_R G_R + c_U G_U + c_F G_F \tag{18}$$

The coefficient to each term is associated with the number of arithmetic operation pr. cell for the different tasks.

The total cost of the AMR-algorithm should be compared to

$$W_{UNI} = c_{flop} G_{UNI} \tag{19}$$

the computational work when computing on a uniformed fine mesh.

4.2 Relations between the grid point counts

We notice that:

- $G_{AMR} > G_R$ whenever $K > 1$ or if $K = 1$ and $G_3 > \frac{1}{4}(G_1 + G_2)$

The simple relation

$$G_L \geq \frac{1}{R^3} G_{L+1} \tag{20}$$

implies $G_U \leq \sum_{L=1}^{2} G_L$. Strong inequality appears when not all cells at level 2 being refined. Thus we have:

- $G_R > G_U$ when $K = 1$ or when sufficiently few cells are refined at the finest level (which is when we want to apply AMR).

Finally we expect $\partial G_L < G_L$ for all L. Thus we expect the typical case to be:

$$G_{AMR} > G_R > G_U > G_F \tag{21}$$

The magnitude of $G_{AMR} - G_R$ and $G_R - G_U$ is essentially a question on whether or not the problem lends itself naturally to mesh refinement. If relatively few cells are refined at each level G_{AMR} could be an order of magnitude larger than the others.

The difference between G_U and G_F does depend not only on the problem, but also the *cutoff* parameter. If this is near 100% and only few unflagged points are included in the refinement, the algorithm might be forced to group the cells for refinement into many small patches instead of fewer but larger. This will in turn increase the ∂G_L, while G_L is slightly decreased. Thus we will see an increase in the $\partial G_L / G_L$ ratio, and thereby closing the gap between G_U and G_F.

4.3 Parameter estimations for our test cases

In our model c_{flop} is a constant, determined by the number of floating point operation needed to advance the solution in one grid point one time step. However, in practice this simple model might be obscured by memory management, incurring unequal execution time for identical floating point operations. We might hope that this "memory-access-noise" is randomly distributed such that an average c_{flop} could be used as a constant. To test whether this is the case or not we have computed the average c_{flop} for various problem sizes, $nx = ny = \{24 : 8 : 256\}$, in the 1-level uniform grid case for both test cases for $t = 1000$s. The results are plotted in Figure 7. Here we observed that, unfortunately, c_{flop} is not a constant, but seems to be a function of the problem size. As we increase the problem size c_{flops} starts to decrease, but soon reaches its minimum before it steadily increases.

For one particular problem size, $NX = NY = 200$, we observe some strange behavior. The profiling of the test runs reports severe performance loss due to L2-cache misses for this particular case. Exactly why this happens, we have not been able to explain.

The coefficient c_{flop} in (19) for $nx = ny = 256$ is for each test case found as the coefficient at the highest point in Figure 7. The results are given in Table 1.

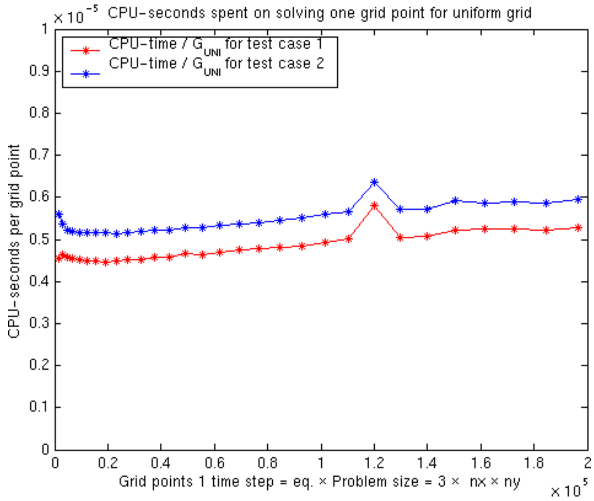

Figure 7: *Test case 1 and 2: c_{flop} for uniform grid.*

Test case	$\frac{c_{flop}}{10^{-6}}$
1	5.28
2	5.94

Table 1: *Measured c_{flop} in (19) for test case 1 and test case 2 and $NX = NY = 256$.*

Counting $G_L, L = 1, ..., NL$ in the numerical test runs described in Section 3, one can compute G_{AMR}, G_R and G_U. G_F can be found by manually including a counter in the routine doing the conservation fix-up. Measuring W_{AMR} in addition, makes is possible to do a least square approximation (LSA) to find c_{flop}, c_R, c_U and c_F in equation (18). This did, however, create a c_U coefficient which did not seem correct. Therefore we decided to measure $W_U = c_U G_U$, and compute an average c_U, before using LSA to find the remaining coefficients. We have also notices that test case 1 appears to have a better cache performance than test case 2, thus providing smaller parameters. We therefore find it necessary to operate with two sets of parameters. The results are given in Table 2.

Test case	$\frac{c_{flop}}{10^{-6}}$	$\frac{c_R}{10^{-6}}$	$\frac{c_U}{10^{-6}}$	$\frac{c_F}{10^{-6}}$
1	4.51	5.00	0.40	40.1
2	5.14	5.33	0.46	43.2

Table 2: *Approximated c_{flop}, c_R, c_U and c_F in (18) for test case 1 and test case 2, using LSA on grid point counts.*

As seen in Figure 7 and Table 1, the two test cases have different values of c_{flop} for the same problem size. This is also clearly seen in Table 2 where in every component test case 2 is more expensive than test case 1. However, what is important in Table 2 is that the relationship between the coefficients is very similar for the two test cases. Comparing

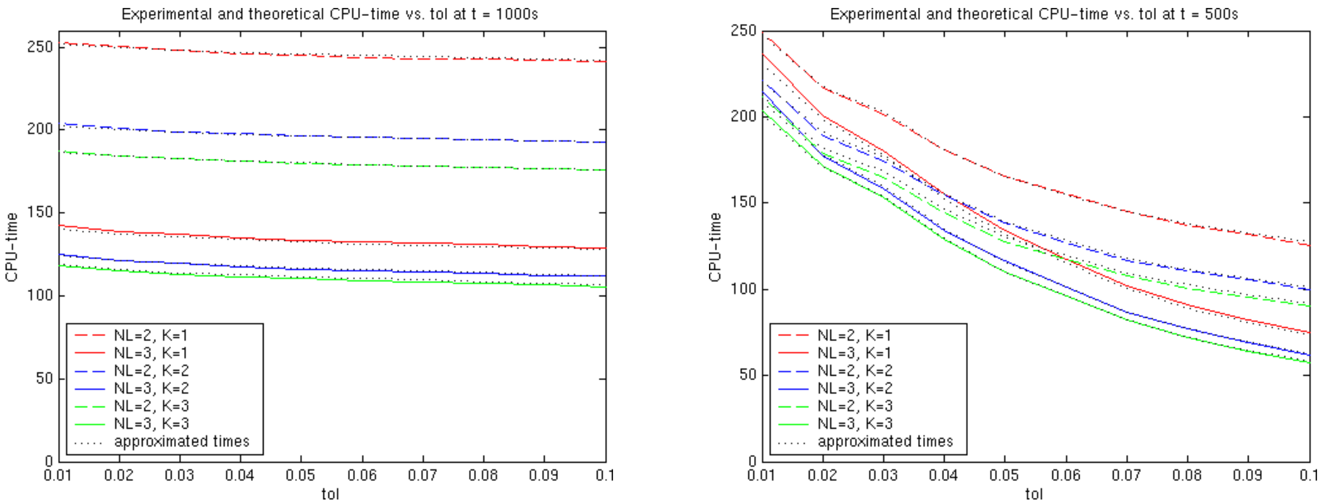

Figure 8: *Experimental and theoretical CPU-time. Left: test case 1. Right: Test case 2.*

c_{flop}, c_R, c_U and c_F, we observe that the unit cost of regeneration of grid is a little bit more expensive pr. grid point than the actual computing, the updating is cheaper and the conservation fix-up is much more expensive. We also notice that the c_{flop}s in Table 2 compare well with those attained for the lower problem sizes of the uniform case displayed in Figure 7.

Plugging the coefficients of Table 2 into (18) and comparing with the actual measurements are done in Figure 8. The figures show good consistency between the model and our experiments.

In Figure 9 and 10 we display the time consumption for the different parts of the code **(A)**, **(R)**, **(U)** and **(F)** for $K = 1$. The only difference when $K > 1$ is that the time spent on **(R)** which is reduced by a factor of K. About 90% of the time is spent on **(A)** and **(R)**, Thus if only a crude estimate is needed

$$W_{AMR} \approx \tilde{c}_{flop} G_{AMR} + \tilde{c}_R G_R \tag{22}$$

would be sufficient. The LSA of $\tilde{c}_{flop}$ and $\tilde{c}_R$ will in our case be as given in Table 3.

Test case	$\frac{\tilde{c}_{flop}}{10^{-6}}$	$\frac{\tilde{c}_R}{10^{-6}}$
1	4.95	4.68
2	6.02	5.16

Table 3: *Approximated $\tilde{c}_{flop}$ and $\tilde{c}_R$ in (22) for test case 1 and test case 2, using LSA on grid point counts.*

The verbal definition of efficiency given in 3.4 can be formalized as

$$Eff \;=\; \frac{\frac{G_{AMR}}{W_{AMR}}}{\frac{G_{UNI}}{W_{UNI}}} = \frac{G_{AMR}}{G_{UNI}} \cdot \frac{W_{UNI}}{W_{AMR}}. \tag{23}$$

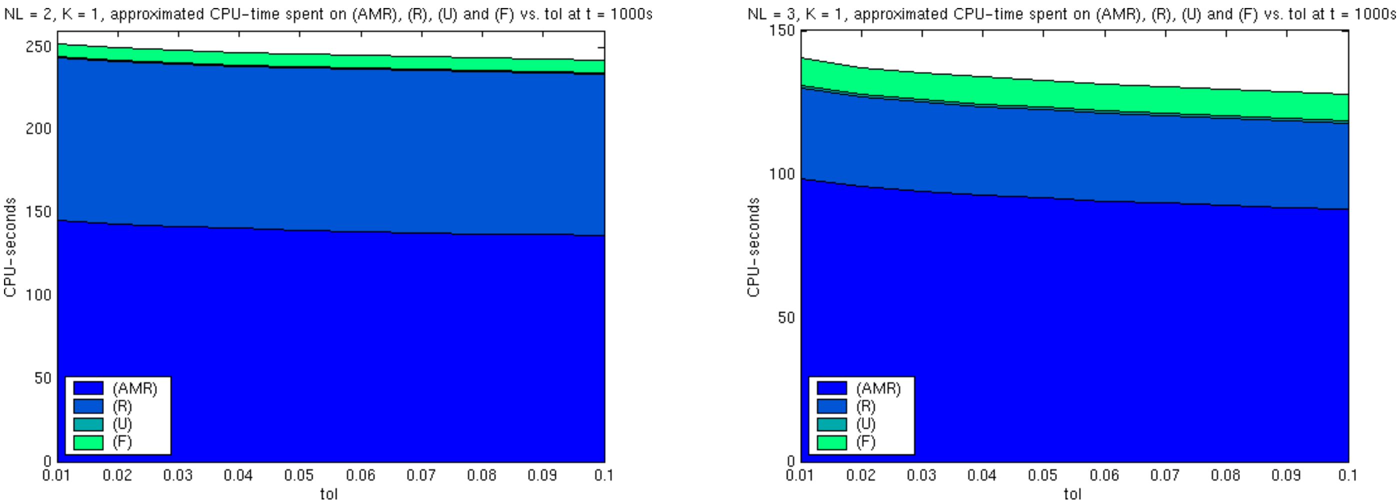

Figure 9: *Time spent on* **(A)**, **(R)**, **(U)** *and* **(F)** *at* $t = 1000s$. *Left Test case 1:* $NL = 2$, $K = 1$. *Right Test case 1:* $NL = 3$, $K = 1$.

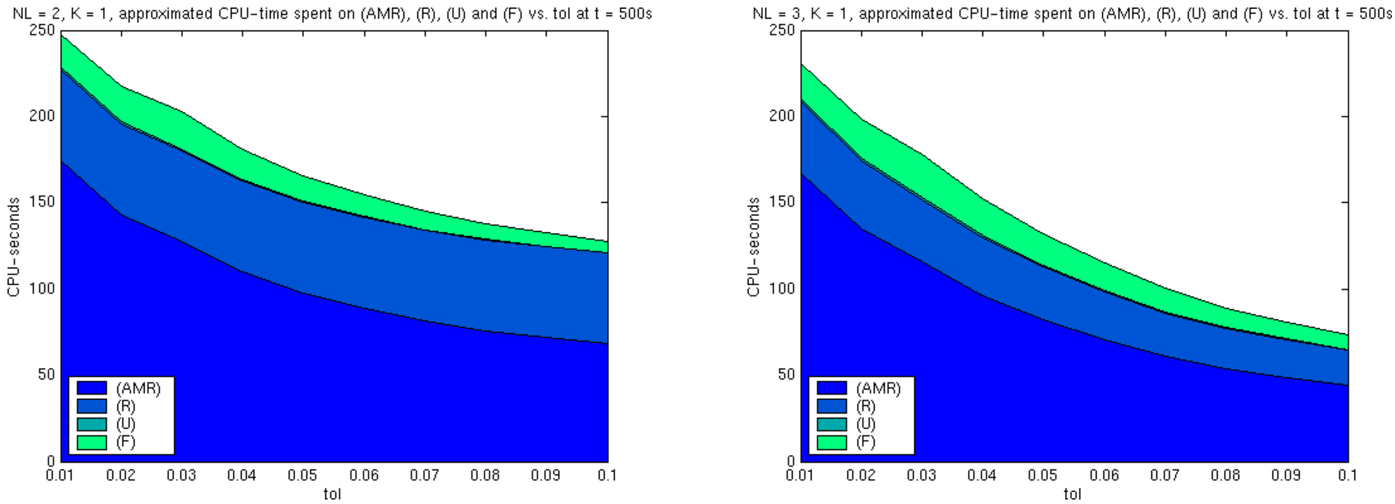

Figure 10: *Time spent on* **(A)**, **(R)**, **(U)** *and* **(F)** *at* $t = 500s$. *Left Test case 2:* $NL = 2$, $K = 1$. *Right Test case 2:* $NL = 3$, $K = 1$.

By plugging (19) and (22) into (23) we obtain

$$Eff = \frac{G_{AMR}}{G_{UNI}} \cdot \frac{c_{flop} G_{UNI}}{\tilde{c}_{flop} G_{AMR} + \tilde{c}_R G_R} = \frac{1}{\frac{\tilde{c}_{flop}}{c_{flop}} + \frac{\tilde{c}_R G_R}{c_{flop} G_{AMR}}} \approx \frac{1}{1 + \frac{G_R}{G_{AMR}}}. \tag{24}$$

The last approximation is valid if we, as in our case (see Table 1 and 3) find $c_{flop} \approx \tilde{c}_{flops} \approx \tilde{c}_R$. Since we always expect to have $G_R < G_{AMR}$ (24) shows that the additional cost of AMR is modest.

The approximated efficiency is plotted together with observed efficiency in Figure 11. The approximated efficiency is lower than the experimental efficiency, which is due to the crude approximation $c_{flop} \approx \tilde{c}_{flop} \approx \tilde{c}_R$.

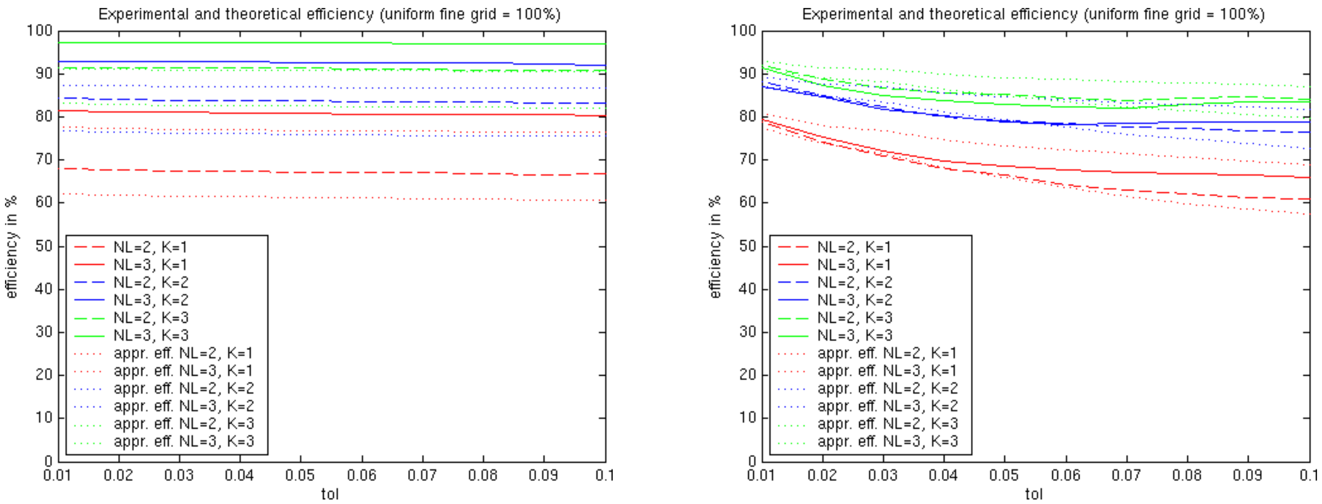

Figure 11: *Approximated and experimental efficiency in percent for* $NL \in \{2,3\}$ *and* $K \in \{1,2,3\}$. *Left: test case 1. Right: test case 2.*

Let $NL = 2$ and define p to be the ratio of refined cells at level 1 divided by the total number of cells at this level. Then requesting

$$W_{UNI} \geq W_{AMR} \tag{25}$$

and plugging in the expressions provided by the formula (22) and (19)

$$c_{flop}G_{UNI} \geq \tilde{c}_{flop}G_{AMR} + \tilde{c}_R G_R \tag{26}$$

$$c_{flop}R^3 G_1 \geq \tilde{c}_{flop}\left(G_1 + p \cdot R^3 G_1\right) + \tilde{c}_R \frac{1}{K}\frac{5}{4}G_1 \tag{27}$$

we find that p has to satisfy:

$$p \leq \frac{c_{flop}}{\tilde{c}_{flop}} - \frac{1}{R^3} - \frac{5}{4}\frac{\tilde{c}_R}{\tilde{c}_{flop}KR^3}. \tag{28}$$

Again setting $c_{flop} \approx \tilde{c}_{flop} \approx \tilde{c}_R$, gives

$$p \leq 1 - \frac{1}{R^3}\left(1 + \frac{5}{4K}\right) \tag{29}$$

The minimum of the right hand side in the above inequality is obtained for $R = 2$ and $K = 1$ implying $p \leq 0.72$.

Applying the argument recursively on more than 2 level, we find that for $K = 1$ and $R = 2$ doing further refinement in AMR-style instead of simply refine all cells, pays off when less than 70-75 % of the cells are being refined. Increasing the value of K and R will increase this percentage.

5 Conclusions and future work

In this study we have tried to assess different aspects of the performance of AMRCLAW [BL97] by combining experiments with performance modeling. The main findings of this study is:

The irregularity in the computational grid, introduced by AMR, *does not* lead to poorer memory management and corresponding lower flop-rate. The reason for this is that on a low level, where the computation is carried out, the data structure in each patch is still a regular 2-d array.

The additional computation for incorporating data from multiple grid-levels is not very expensive, while costs associated with error estimation and regridding is potentially more expensive. The later may, however, be reduced by incorporating a modest buffer zone and permitting error estimation and regridding to only take place every 3rd time step.

AMRCLAW is a successful implementation where one succeed in combining the computational desirable features of CLAWPACK [LeV95] with the computational efficiency of the AMR-technique.

The appetite for ever faster solution techniques on larger and larger problems seems however to be unlimited. Thus there is a need for further speed enhancement. The obvious direction to move in is parallelization. There are obvious limitation and possibilities to parallelize AMRCLAW. First we notice that the computation on the different grid levels have to be sequential within one coarse time step. Secondly the splitting of a grid level in multiple patches provides a natural domain decomposition for parallelism. It is not clear however how to deal with overlapping internal boundaries when processed in parallel. Moreover, as these patches usually are of different sizes a straightforward splitting of patches to processors has potentially huge load imbalance.

In earlier work [BS01] we have shown theoretically and experimentally that utilizing nested parallelism, when available, turns out to be imperative for good performance. In particular this is the case when the outer level, course grain parallelism contains relatively few tasks of uneven size, which is exactly the case for AMRCLAW. Here each grid patch on a given level constitute a task. These tasks can be processed in parallel. But there is lot of parallelism within a task too. The work associated with a task can be done by a team of threads. This problem has the kind of multilevel parallelism, discussed in [BS01]. To obtain a good load balance for this scheme we need to be able to make good a priori estimate of the work load. We hope the work reported here can contribute to construct such estimate.

Scientists at Berkeley National Laboratory [RBL+99] has already parallelized AMRCLAW with MPI [RBL+99]. So has scientist from NASA and the Courant Institute, NYU [ABA00]. These implementations do only utilize one level of parallelism. Another difference is that we plan to use OpenMP [omp97, omp00] for our implementation. That way we don't have to worry to much about a constantly shifting data access pattern.

Acknowledgments

This project has been supported by a grant from the Norwegian Supercomputing Program with computing time on Parallab's SGI/CRAY Origin 2000. We also like to thank the Parallab staff for making it possible to run full scale test on their heavily loaded system.

It is with great pleasure we acknowledge the stimulating discussions with Prof. Jörn Behrens at Munich University of Technology.

References

[ABA00] M.J. Aftosmit, Marsha J. Berger, and G. Adomavicius. A Parallel Multilevel Method for Adaptive Refined Cartesian Grids with Embedded Boundaries. In *38th Aerospace Sciences Meeting and Exhibit*, 2000.

[BC84] Marsha J. Berger and Phillip Colella. Local Adaptive Mesh Refinement for Shock Hydrodynamics. *J. Computational Physics*, 53(3):561–568, 1984.

[BL97] Marsha J. Berger and Randall J. LeVeque. AMRCLAW, Adaptive Mesh Refinement + CLAWPACK software, test version available online from. Technical report, `http://www.amath.wahsington.edu/~rjl/amrclaw`, 1997.

[BL98] Marsha J. Berger and Randall J. LeVeque. Adaptive Mesh Refinement using Wave-Propagation Algorithms for Hyperbolic Systems. *SIAM J. Numer. Anal.*, 35(6):2298–2316, 1998.

[BO84] Marsha J. Berger and Joseph Oliger. Adaptive Mesh Refinement for Hyperbolic Partial Differential Equations. *J. Comput. Phys.*, 53, 1984.

[BS01] Ragnhild Blikberg and Tor Sørevik. Nested parallelism: Allocation of threads to tasks and OpenMP implementation. *Journal of Scientific Programming*, 8(3):1–10, 2001.

[LeV95] Randall J. LeVeque. CLAWPACK, a Software Package for Conservation Laws and Hyperbolic Systems software, test version available online from. Technical report, `http://www.amath.wahsington.edu/~rjl/clawpack`, 1995.

[LeV98] Randall J. LeVeque. Balancing Source Terms and Flux Gradients in High-Resolution Godunov Methods: The Quasi-Steady Wave-propagation Algorithm. *J. Comput. Phys.*, 146(1):346–365, 1998.

[omp97] OpenMP Fortran Application Program Interface, ver 1.0. Technical report, `http://www.openmp.org/`, October 1997.

[omp00] OpenMP Fortran Application Program Interface, ver 2.0. Technical report, `http://www.openmp.org/`, November 2000.

[RBL+99] Charles A. Rendleman, Vincent E. Beckner, Mike Lijewski, William Crutchfield, and John B. Bell. Parallelization of Structured, Hierarchical Adaptive Mesh Refinement Algorithms. Technical report, April 1999.