Nested parallelism applied to AMRCLAW

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

September 1, 2002

Abstract

In earlier work [BS01] we have shown theoretically and experimentally that utilizing nested parallelism, when available in the problem, turns out to be imperative for good performance. In particular this is the case when the course-grain parallel level contains relative few tasks of uneven size. This is exactly the case for AMRCLAW [BL97], where each grid patch on a given level constitutes a task, and where all the tasks can be processed in parallel. Since there is also lots of parallelism within a task, the work associated with a task can be done by a team of threads.

In this paper a nested parallel adaptive mesh refinement algorithm will be presented. The performance of using this strategy on AMRCLAW will be compared to the performance of two 1-level parallel strategies, a coarse-grained and a fine-grained. The parallelization will be done in OpenMP [omp97, omp00].

To obtain a good load balance for this scheme it is needed to make good a priori estimate of the work load. The work reported on in [BS02] contributes to construct such estimate.

Keywords: Adaptive Mesh Refinement, AMRCLAW, nested parallelism, OpenMP

1 Introduction

Solving partial differential equations (PDE) numerically for interesting problems, is often very expensive in terms of computational effort. 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. However, to obtain a high accuracy on a uniform grid, it is necessary to have the finest resolution all over the entire domain. This will cause more computational work to be done than necessary in those parts of the domain where a coarser grid is sufficient. To avoid using the finest resolution needed in the entire domain, adaptive mesh refinement (AMR) can be used to reduce the computational costs. In AMR grid points are clustered adaptively in regions where it is most need for close grid points. Refined grids are created or existing ones removed based upon estimates of the error. In this manner, one can have a fine resolution where it is needed and use a coarse resolution where this is sufficient.

Another way of improving the performance of compute intensive problems is by parallelization. In this paper we will examine how it works to mix these two ways of performance improvement, by applying parallelism to AMRCLAW [BL97]. AMRCLAW is a freely available software package, created by Randy LeVeque and Marsha Berger. It is based on Berger's AMR

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

algorithms and -software [Ber95] and LeVeque's software Conservation LAW PACKage, CLAWPACK [LL97]. In the latest release of CLAWPACK, version 4.0 [LLBM00], AMRCLAW is included.

First two naive 1-level parallel approaches, a fine-grained and a coarse-grained, will be tried, but unfortunately without much success. Our next try will then be to test the nesting techniques presented in [BS01], implementing by explicit thread programming in OpenMP. After removing a dependency in the AMR algorithm, a nested parallel AMR algorithm will be presented. In another paper [BS02] we have examined the costs and savings of AMRCLAW [BL97], and this knowledge has been used in making good a priori estimate of the work load. The performance results of the nested parallel AMRCLAW will be compared to the performance of the 1-level parallel approaches.

A summary of the sequential AMR algorithm will be presented in Section 2. The test problem and 1-level parallel approaches of AMRCLAW are given in Section 3, while the nested parallel AMR algorithm and numerical results of applying nested parallelism to AMRCLAW are given in Section 4. In Section 5 the special workarounds needed to implement this in OpenMP, in lack of nesting directly implemented in OpenMP, are explained. Finally, the conclusions and related work will be given in Section 6.

2 Sequential AMR algorithm

In this section a short summary of the AMR algorithm developed by Berger and Oliger [BO84] will be given. More details can be found in [BC84, BL98]. AMR solves time dependent PDE using a dynamically changing hierarchy of nested rectangular Cartesian grids with successive 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, where a *patch* is a rectangular Cartesian grid, which may or may not have neighbor patches of the same resolution, but which always has a parent patch with one step coarser 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. (The first time step on the existing grid is identically to the same time step as used in the integration, and this time step is not repeated. Therefore, in practice, only 1 time step on the existing grid is taken.) 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 $tolsp$. This refinement procedure is applied recursively until a maximum level of refinement NL.

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 are also flagged for refinement. In addition, some "fill in" cells are flagged to maintain a rectangular structure of refined patches.

Boundary conditions at all levels are provided by extending the grid patch by G "ghost cells" in a band around the patch, and values are assigned to the ghost cells at the start of each time step. Details of assigning values to ghost cells will be given in Section 3.2.3.

In CLAWPACK 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 provides different schemes for time integration. In our test cases a 2nd order scheme is chosen. AMRCLAW is the part of CLAWPACK 4.0 containing the AMR algorithm, and is used in the test cases of this paper. For a specific time interval in AMRCLAW, 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 were 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. To better emphasis the parallel aspect of this algorithm, some minor modifications to the version given in [BS02] have been made. What there was referred to as "Regridding" or **(R)**, has now been split into **(R)** = **(RB)** + **(RA)** + **(RO)** to be more precise in references later. In this algorithm $R_1 = 1$.

Algorithm 1: Recursive AMR

/ In this algorithm Ω_L^i represents grid patch i at level L.*
*$\Omega_L = \cup_{i=1}^{G_L} \Omega_L^i$, G_L is the number of patches at level L. */*
Ω_0 represents the grid twice as coarse as the Ω_1 grid.

AMR(L, Ω_L, k_L)
    for $T_L = 1, R_L$
        if $(k_L = 0)$ **and** $(L < NL)$ **then** /* *Regrid* */
            Compute_boundary_values$(\partial\Omega_{L-1}, \partial\Omega_L)$ **(RB)**
            Advance_PDE_1_time_step(Ω_{L-1}, Ω_L) **(RA)**
            Estimate_error(Ω_L) **(RO)**
            Flag cells with $error > tol$ and cells in buffer zone **(RO)**
            Organize flagged cells into patches: Ω_{L+1}^i; $i = 1, \cdots, G_{L+1}$ **(RO)**
        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 as:

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

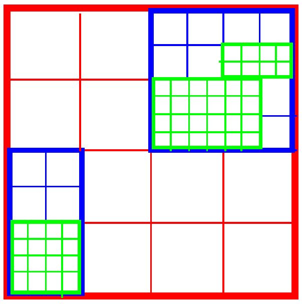

Figure 1: *Grids at 3 levels at a certain point of time.*

2.1 An example

Suppose $NL = 3$, and that the grid situation at time steps $T_1 = T, T_2 = 1, T_3 = 1$ looks as in Figure 1. The sequential procedure of solving a computational problem as the one in Figure 1 for $K = 1$ and $R_2 = R_3 = 2$, is illustrated in Figure 2, taking one line (from the left to the right) at the time, starting from the top. To make the figure as simple as possible, only the time-stepping parts, **(RA)** and **(A)**, is included. Note that in Figure 1 the size of a cell reflects the computational work involved, not its physical size.

The procedure starts with the regridding; a large time step at level 1 is taken on a grid that is twice as coarse in each direction as the existing grid. Then one time step is taken at the existing grid. Next the solution is advanced on the grid at level 1. As seen in Figure 1, there are at $T_2 = 1$ two grid patches at level 2. Before advancing the solution on these, the regridding takes place for both, taking one large time step on grids that is twice as coarse in each direction, and one time step on the existing grids. Then two time steps are taken at each level 3 grid patch. After this, the regridding at level 2 takes place again, before the solution is advanced one time step at level 2. At last two time steps are taken at each level 3 grid patch once again.

3 Parallel approaches

In our opinion the AMR algorithm lends itself naturally to SMP-programming as refined patches are created and dismissed as the computation proceeds in an a priori unpredictable way. This makes redistributing data hard and (virtually) shared memory programming more attractive. OpenMP [omp97, omp00] has therefore been chosen in the parallelization of AMRCLAW in this paper.

All test runs are done on a low loaded SGI/CRAY Origin 2000.

3.1 The test problem

Our test problem for AMRCLAW is the shallow water equations (SWE) [LeV98]. The version of SWE used here can be written as in [BS02], where it is assumed to be no bottom friction, diffusion or viscosity. The boundary is solid, with "no-slip" conditions, implying the von-Neumann boundary conditions. Our test case takes place in a square basin with flat bottom topography domain $\Omega = [0, 20] \times [0, 20]$km.

In this paper the performance of a 2-level grid, resolution 128×128 at the coarsest grid, has been studied. In addition to the maximum level of refinement parameter $NL = 2$, the parameters mentioned in Section 2 are set to $R_2 = 2$, $K = 1$, $tol = tolsp = 0.6$.

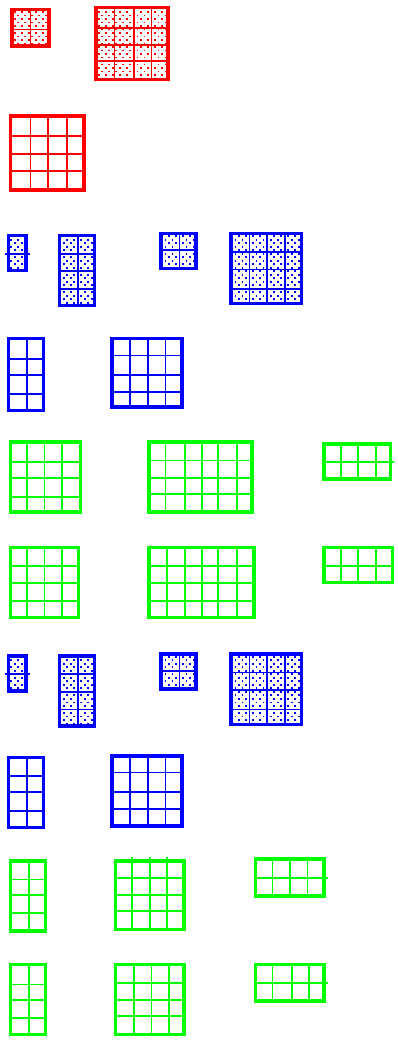

Figure 2: *The sequence for solving the subproblems of the example of Figure 1 in one coarse grid time step. The work associated with* **(A)** *is marked with non-dotted grids, and the work associated with* **(RA)** *with dotted grids.* $R_2 = R_3 = 2$ *and* $K = 1$.

The initial conditions are:

$$u(x,y,0) = 0, \quad v(x,y,0) = 0, \quad h(x,y,0) = \begin{cases} 201\text{m} & \text{for } x \leq 0.650\text{km or } y \leq 1.300\text{km} \\ 200\text{m} & \text{elsewhere} \end{cases} \tag{1}$$

where $h(x,y,t)$ is the fluid depth above the bottom, and $u(x,y,t)$ and $v(x,y,t)$ the velocity in x- and y- direction respectively. This initialization results in two straight waves, one propagating from west to east, the other from south to north. The waves will naturally cross each other immediately, and the cross-area will continue to exist, propagating from south-west to north-east. In Figure 3 snapshots of an AMRCLAW simulation of the test case is displayed.

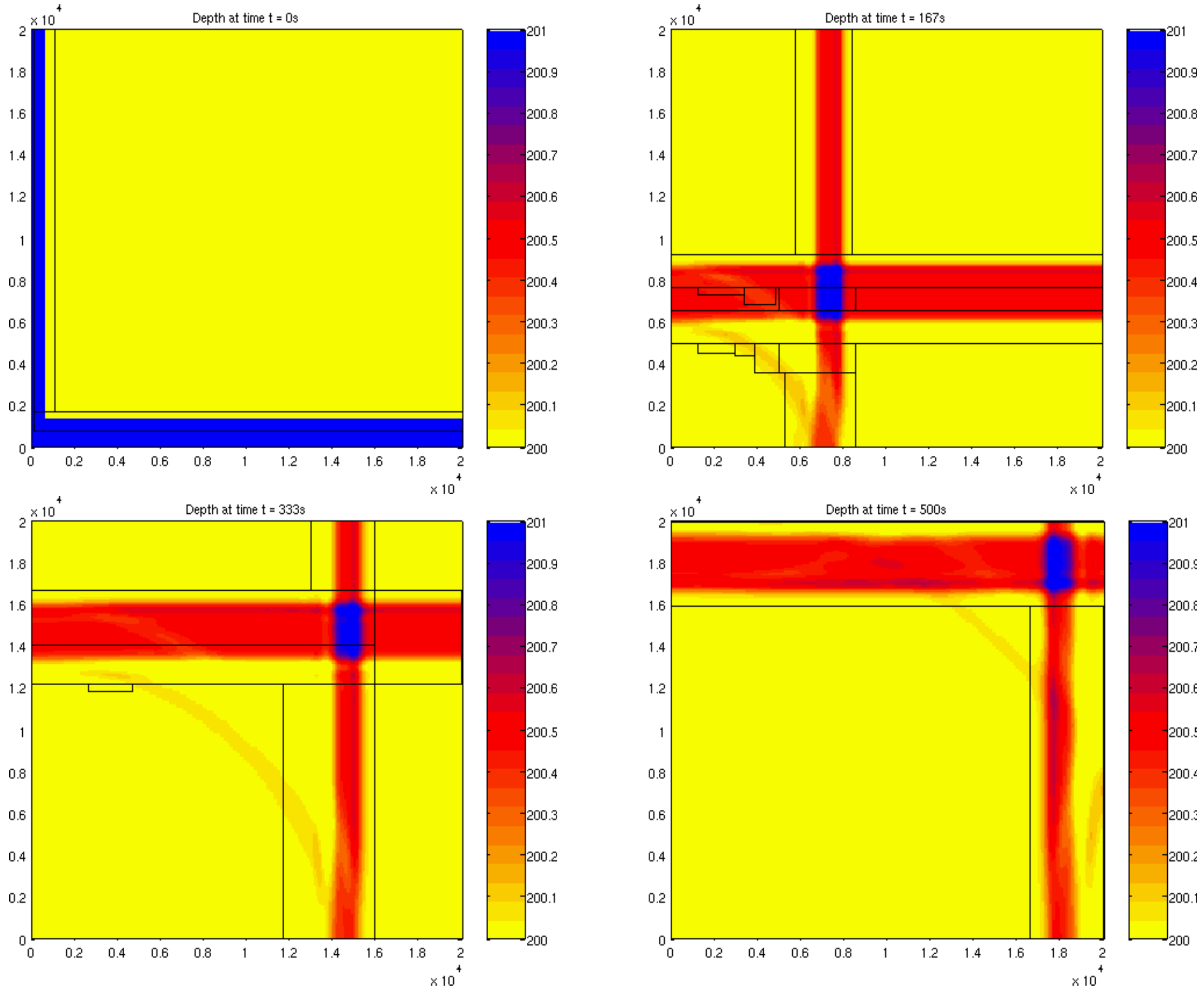

Figure 3: *Initial depth and depth at* $t = 167s$, $t = 333s$ *and* $t = 500s$, *for* $NL = 2$ *and* 128×128 *cells at coarsest grid. Boundaries for fine grid patches are marked with black lines.*

3.2 1-level parallelization of AMRCLAW

When parallelizing AMRCLAW by 1-level parallelization one can choose to either parallelize on loop-level, a fine-grained approach, or one can choose to parallelize on patch-level, a coarse-grained approach.

The work of [BS02] showed that about 90% of the time was spent on advancing the PDE in time **(A)** and on the regeneration of grids **(R)**. Therefore, in both the fine-grained and the coarse-grained approach, the effort has been put in parallelizing the part of the code involved in **(A)** and **(R)**, or more precise the time-stepping parts **(A)** and **(RA)**.

3.2.1 Fine-grained parallelization

When deciding to parallelize a large code like AMRCLAW, the first thought is to parallelize on loop-level. A fine-grained SMP parallelism on loop-level is the easiest way of parallelizing as it can be carried out by inserting directives. Consequently, it does not require many changes, if any, in the code.

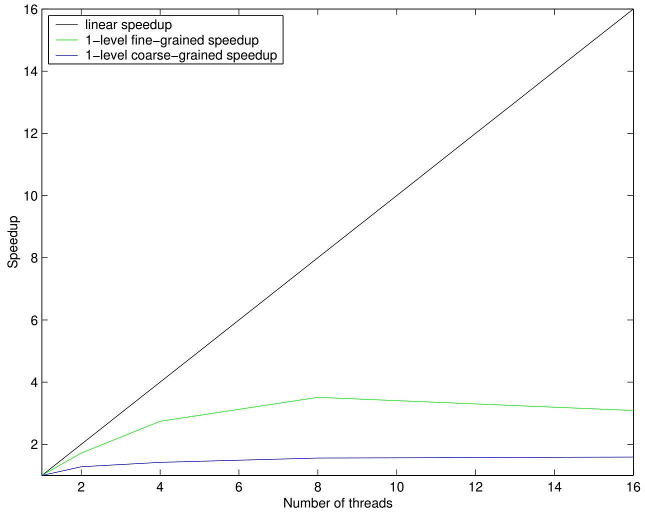

Figure 4: *Speedup obtained for 1-level fine-grained and coarse-grained parallelization of AMR-CLAW. Linear speedup is included.*

When parallelizing on loop-level it is well known that the outer loop should be preferred to get as much work as possible inside a parallel region. When inserting directives on the outer loops involved in **(A)** and **(RA)**, a problem of reduction on flux-arrays pops up. This problem can be avoided if each thread does (up to) 2 extra loop-iterations, but this requires an explicit modification of the code.

The speedup-result of the fine-grained parallelization tested on the test case described in 3.1 is shown in Figure 4. One can see that this parallelization strategy has problems with scaling. Maximum speedup of 3.5 was achieved at 8 threads. For small patches and a high number of threads the work for each thread will be so small that the parallel overhead will dominate, or there will simply not be enough work for all threads. This is probably the main reason for the poor speedup.

3.2.2 Coarse-grained parallelization

Alternatively one could try to parallelize the code in a coarse-grained manner by assigning one thread to each patch. When the number of patches is larger than the number of threads, threads can be assigned to more than one patch, executing one patch at a time. When the number of threads is larger than the number of patches, the threads not assigned to any patch are idle. A patch can only be executed by one thread. In this way, all threads assigned to one or more patches, will work in parallel.

For the regridding part, threads are spawned in front of **(RA)**. **(RB)** has not been included in the parallelization, since this part of the code is not very time consuming. The threads are joined again after **(RA)**. For the part advancing the PDE in time, threads are spawned in front of **(B)** to be able to avoid synchronization between the time steps on the finest level NL. For $L < NL$, the threads joins again right after **(A)**. For $L = NL$, the threads do all R_L time steps before joining after **(A)** in $T_L = R_L$.

As seen from Figure 4, the coarse-grained version has even poorer scaling. The reason for this is that the patches often differ a lot in size and that the number of patches frequently is less than the number of threads (For instance at level $L = 1$, where there is always only one patch.), resulting in extremely bad load balancing.

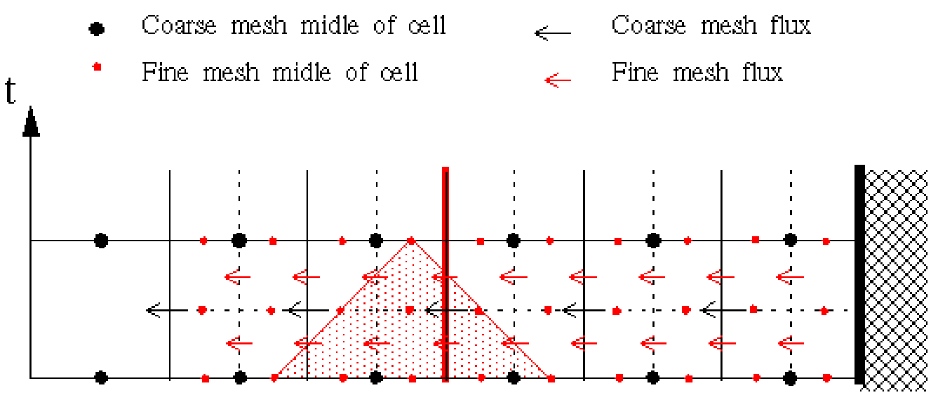

Figure 5: *Dependencies of cell values and fluxes from one time step to another.*

3.2.3 Removing dependencies

There is one dependency in the original sequential algorithm, which will affect the coarse-grained (and subsequently the nested) parallelization on level $L = NL$. This dependency can be avoided with a minor rewriting, which will be described in this section. To make the explanation simple and consistent with the parameters used elsewhere in this paper, it is in the following assumed that $R_{NL} = 2$.

Boundary values at all levels are provided by extending the actual patch by ghost cells in a band around the patch, and assigning values to the ghost cells at the start of each time step. If the ghost cells coincide with the physical boarder of the computational domain, standard boundary conditions are used. If not, values are assigned, in one of two ways:

1. If values are available from neighbor patches at the same grid level, they are provided by a simple copy.

2. If there are no neighboring patches at the same grid level, they are computed by linear interpolation in time and space from a coarser parent grid.

If 1 is applied, the extra computation in 2 can be avoided, but then a dependency between patches at the same level occurs in the computation of boundary values. AMRCLAW defines an order in which the patches should compute their boundary values, and consequently this part of the code is sequential.

Since computation of boundary values is not very time consuming, it could have been left outside of the parallel region, like in the fine-grained approach. However, for $L = NL$ this would imply a need for synchronization between every time step T_{NL}. A patch which is done with its first time step can not begin with the computation of boundary values in the second time step before the neighbor patches also are through with its first time step.

Removing the dependency and the need for synchronization by assuming that none of the patches has neighbors, and applying linear interpolation from a coarser grid everywhere in the beginning of each time step, is possible, but will create a less accurate solution.

To avoid the dependency and the need for synchronization, the alternative presented here is to integrate one extra cell in each direction in the first out of $R_{NL} = 2$ fine grid time steps. Then for the second fine grid time step, the ghost cell values just achieved from integration can be used instead of copying from neighbor patches. In this way the threads can continue directly to the next time step, without synchronizing, since the boundary values originally needed from neighbors now are available. See Figure 5, where the triangle shows which cells in the initialization of the first time step a cell depends on in the second time step.

Using this alternative on all grid patches, will create a solution which is identical to using the original procedure. It will, however, create a little bit more computational work, but we believe that the benefits in performance improvements make up for this.

4 Nested parallelization applied to AMRCLAW

The two 1-level strategies both had problems with scaling. The problems were, however, of very different nature and where the fine-grained strategy had problems the coarse-grained strategy had not and vice versa. While the fine-grained version works best for few large patches, the coarse-grained has its advantages when the patches are many and with as equal size as possible. Thus a strategy that takes advantages of the "best of both worlds" seems attractive. As the two parallelization strategies occurs on different "levels", a combined strategy has to be a "two-level" or nested parallel strategy. To test this idea we have implemented a nested approach which combines the two 1-level approaches.

Before introducing the details of nested parallelism, a few definitions have to be made. First, a problem can be decomposed into multiple *task*s, which are units of work. Dependencies between *task*s occur if the output of a task is needed as input of an other. The computation within a *task* is independent from the computations within other *task*s. Second, a *team* is defined as a group of threads collaborating in executing the work associated with a specific task.

Applying nested parallelism to AMRCLAW, the threads will be divided in teams, each team assigned to a specific grid patch, which on a given level constitutes a task. This corresponds to the coarse-grained parallelization. Next, each task can be processed in parallel on loop-level, and the threads within each team share the work associated with the respective task using fine-grained loop-level parallelization. The size of a team should reflect the workload of the task. The algorithm for distribution of work to threads will be presented in in the following subsection.

4.1 Distribution algorithm

For the nested parallel version of AMRCLAW to be successful, it needs to be flexible in dynamically distributing threads to tasks in a near optimal way, to create good load balance for a variable number of patches and variable sizes of patches.

In [BS01] we presented a distribution algorithm, which distributed P threads to N tasks, for $P \geq N$ and each task having sufficient work for at least one thread. To cover all possible occurrences of number of patches and sizes, the distribution algorithm has to be extended, to be able to tackle cases where either $P < N$ or cases where some patches are so small that more than one should be dealt with by 1 thread. This subproblem is an NP-complete bin-packing problem, and is traditionally approximated by the "sorted-best-fit algorithm.

Algorithm 2: Distribution of work to threads

/ This algorithm finds a distribution of the work of N tasks to P threads.*
The patches are divided in 2 sets, $\mathcal{L}$ ("large" tasks) and $\mathcal{S}$ ("small" tasks), where one or more threads are working on the same patch in $\mathcal{L}$, and where each thread has one or more tasks to work on in $\mathcal{S}$. How many threads are distributed to patch i is given by p_i, while the number of tasks distributed to thread i in $\mathcal{S}$ is given by t_i.
*Total number of teams is set to T. */*

$[p_{1:T}] = \textbf{Distribute}(N, P, w_{1:N})$

$W = \sum_{i=1}^{N} w_i;$

$\mathcal{L} = \{\forall i; w_i > \frac{W}{P}\};\quad N_{\mathcal{L}} = |\mathcal{L}|;\quad W_{\mathcal{L}} = \sum_{w_i \in \mathcal{L}} w_i;\quad P_{\mathcal{L}} = \text{int}\left(\frac{W_{\mathcal{L}}}{W} \cdot P\right);$

$\mathcal{S} = \{\forall i; w_i \leq \frac{W}{P}\};\quad N_{\mathcal{S}} = |\mathcal{S}|;\quad W_{\mathcal{S}} = \sum_{w_i \in \mathcal{S}} w_i;\quad P_{\mathcal{S}} = \text{int}\left(\frac{W_{\mathcal{S}}}{W} \cdot P\right);$

/* *For patches where* $i \in \mathcal{L}$: */

for $j = 1,\ N_{\mathcal{L}}$

$p_j = 1;$

end for

for $j = N_{\mathcal{L}} + 1,\ P_{\mathcal{L}}$

Find i such that $\frac{w_i}{p_i} = \min_{k \in \mathcal{L}}(\frac{w_k}{p_k});$

$p_i = p_i + 1;$

end for

/* *For patches where* $i \in \mathcal{S}$: */

Sort $\mathcal{S}$ such that $w_i \geq w_j \quad \forall i < j \quad \text{and} \quad i, j \in [1, N_{\mathcal{S}}];$

for $j = 1,\ P_{\mathcal{S}}$

$p_j = 1;$ /* *Only one thread in each team.* */

$t_j = w_j;$

end for

for $j = P_{\mathcal{S}} + 1,\ N_{\mathcal{S}};$

Find i such that $t_i = \min_{k \in \mathcal{S}}(t_k)$

$t_i = t_i + w_j;$

end for

/* *Total number of teams:* */

$T = N_{\mathcal{L}} + P_{\mathcal{S}};$

Note that for small patches (patches in $\mathcal{S}$) the work distribution algorithm degenerate to the 1-level coarse-grain parallel strategy. But this is okey, as it is the only situation where we expect it to be good: Many patches, few threads.

4.2 Nested parallel AMR algorithm

When extending the AMR algorithm to 2 levels of parallelization, the starting point is Algorithm 1 presented in Section 2.

In front of each parallel region, starting at **(RA)** and **(B)**, the distribution routine has to be called. Actually, for the error estimation this routine has to be called twice, to divide each team in two, one subteam taking one time step on the existing grid and the other subteam taking one coarse time step on the grid that is twice as coarse in each direction. Before the error estimation, the subteams and teams again joins.

In front of **(B)**, and when $L < NL$ or $T_L = 1$, the distribution routine is called again, and teams of threads are created. After executing **(B)** and before starting on **(A)** a synchronization point is needed for $L < NL$ or $T_L = 1$ to make sure that all patches have updated the ghost cells before advancing the PDE one time step further.

After **(A)** and for $L < NL$ or $T_L = 1$ all the threads need to join for several reasons. First, for $L > 1$, when level $L + 1$ has executed its R_{L+1} local time steps, and updated its parent patches, new ghost cell values have to be assigned. Second, the threads have to be redistributed before starting on level $L + 1$ to preserve load balance. For $L = NL$ and $T_L < R_L$ the teams can, thanks to the removal of dependency explained in Section 3.2.3, continue directly to the next fine grid time step without synchronizing with the other teams.

Algorithm 3 gives the nested parallel AMR algorithm, but first some constructs used in the algorithm have to be explained:

**!$rb Parallel Teams**($x_{1:N}$)	Divide the threads in N teams, x_i threads in team i.
!$rb Synchronize	No team continue to "C" (time-stepping) before all teams are done with "A" (assigning boundary values). The code in between "A" and "C" is considered as "B".

Algorithm 3: Nested Parallel Recursive AMR

```
        /* In this algorithm Ω^i_L represents grid patch i at level L.
        G_L is the number grid patches at level L, G_1 = 1.
        Ω_L = ∪_{i=1}^{G_L} Ω^i_L is a union of grid patches at level L having the same parent grid.
        Ω_0 represents the grid twice as coarse as the Ω_1 grid. */

        AMR(L, Ω_L, k_L)
              for T_L = 1, R_L
                  if (k_L = 0) and (L < NL) then /* Regrid */
                     Compute_boundary_values(∂Ω_{L-1}, ∂Ω_L)                 (RB)
                     [p_{1:T}] = Distribute(G_L, P, w_{1:G_L})
!$rb                 Parallel Teams(p_T)
                          [p_{1:T'}] = Distribute(2, p_i, [w_{i,1} = w_i, w_{i,2} = 1/(k_L^2) · w_i])
!$rb                      Parallel Teams(p_{T'})
!$omp                         Parallel
                                  Advance_PDE_1_time_step(Ω_{L-1}, Ω_L)      (RA)
!$omp                         End Parallel
!$rb                      End Parallel Teams
!$rb                 End Parallel Teams
                     Estimate_error(Ω_L)                                     (RO)
                     Flag cells with error > tol and cells in buffer zone   (RO)
                     Organize flagged cells into patches: Ω^i_{L+1}; i = 1, ··· , G_{L+1}   (RO)
                  end if
                  if (L < NL) or (T_L = 1) then
                  [p_{1:T}] = Distribute(G, P, w_{1:G})
!$rb              Parallel Teams(p_{1:T})
                  end if
!$omp                Parallel
                          Compute_boundary_values(∂Ω_L)                      (B)
                          if (L < NL) or (T_L = 1) then
!$rb                          Synchronize
                          end if
                          Advance_PDE_1_time_step(Ω_L)                       (A)
!$omp                End Parallel
                  if (L < NL) or (T_L = R_L) then
!$rb              End Parallel Teams
                  end if
                  k_L = (k_L + 1) mod K
```

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

4.3 Parallel example

The dependency graph of the sequential example in Section 2.1 is shown in Figure 6. Each edge should be associated with a team of treads, where the thickness indicates the size of a team. The size of the grids reflects the amount of work associated with **(RA)** and **(A)**, which are the time consuming parts of the parallel regions.

Figure 6 shows 3 levels of grids and 2 levels of parallelism. For each grid level the threads are divided in teams, where each team work on one grid-patch/task. For $L < NL$, each team again divides in two, each subteam doing in parallel one of the two time-steppings included in the regridding. After this all subteams and teams is synchronized. (The reason for the synchronization between **(RA)** and **(A)** is that these regions starts from two different places in the code. Letting the teams go directly from **(RA)** to **(A)** would have caused a messy code.) Then, for all L, the threads are (again) divided in teams, where each team divides the work associated with advancing the PDE one time step among each other. For $L = NL$ the last step is done twice without any synchronization. At the end of one level, all teams synchronize.

4.4 Numerical results

The nested parallel AMR algorithm presented in Section 4.2 works for any choice of NL. However, the number of levels used in the numerical tests is 2, as for the 1-level parallel tests. This has been done to keep the experiments simple and easier to analyse.

Figure 7 shows the speedup obtained for nested parallelization of AMRCLAW tested for the test case described in Section 3.1. The 2-level "work-load corrected" maximum speedup is also displayed in the figure, to give a more realistic view of maximum theoretical obtainable speedup. This was found by dividing the total amount of work over all time steps by the sum of maximum work per thread in each time step. The 1-level speedup-results from Section 3.2 is also shown in the figure. One can clearly see that the nested version has much better speedup than both the 1-level versions. The speedups increases steadily until 10 threads, before it starts to flatten out.

Speedup at 10 threads is 6.0, which is okey, but not too fantastic. Profiling shows that the time spent in the distribution routine is neglectable, and can not responsible for slowing down the performance. The problems already described when commenting on the performance of the 1-level parallel cases does not dissappear, allthough they are reduced. We like to stress that this is a hard problem to parallelize efficiently. The dynamic structure of the problem implies non-uniform data access as grid patches are created and/or deleted. Moreover the dynamic assignment of threads to tasks will most likely create a very irregular data-access pattern, and put severe stress to the bottleneck of all cache based HPC-systems; the internal moving of data.

Keeping all these difficulties in mind we find the speedup quite satisfactory. Not at least because both the standard 1-level strategies failed so badly. We therefore conclude that for these cases, nested parallelism offers much better scaling opportunities than 1-level strategy.

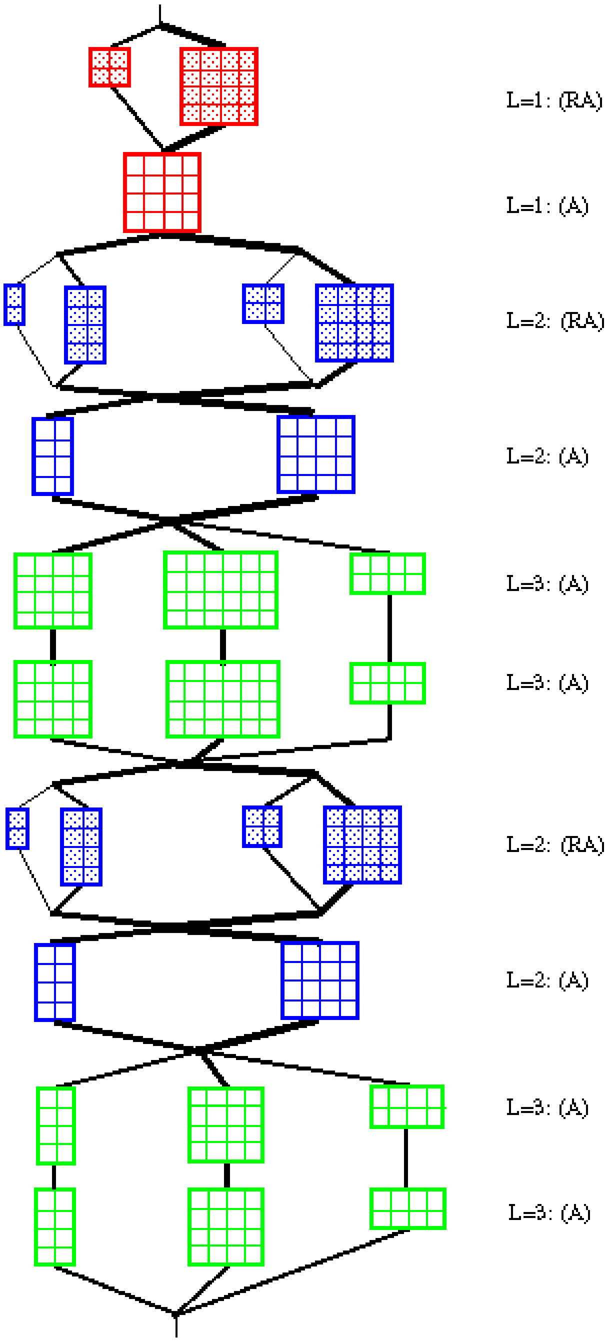

Figure 6: *Nested parallel solving of patches at level 1, 2 and 3 within one coarse grid time step (Sequential solving showed in Figure 2.).* $R_2 = R_3 = 2$ *and* $K = 1$.

5 Implementation

Since serializing nested parallelism is still compliant with the OpenMP spec., and SGI, as many vendors, has chosen to do so, the explicit thread programming techniques described in [BS01] have been used to implement nested parallelization of AMRCLAW.

During the implementation of the nested parallelization of AMRCLAW, directives appropri-

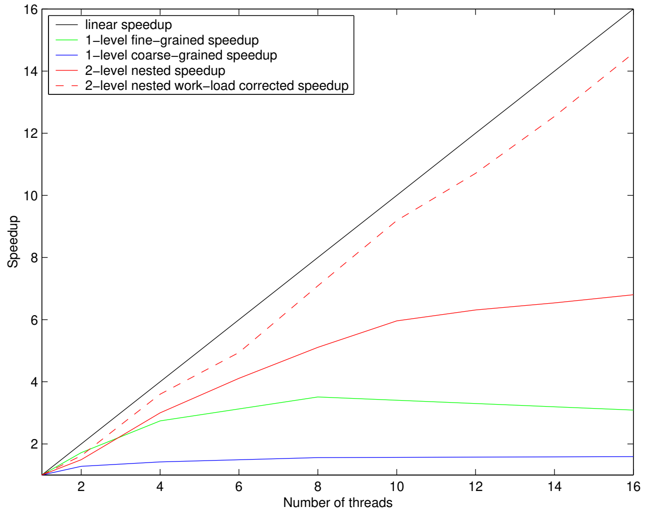

Figure 7: *Speedup obtained for 2-level nested parallelization of AMRCLAW, together with earlier performance results obtained for 1-level fine-grained and coarse-grained approaches. Linear speedup and theoretical speedup for the nested approach is included.*

ate for the team concept were strongly missed. Team-aware directives like `!$OMP TEAMPRIVATE`, corresponding to `!$OMP THREADPRIVATE`, and `!$OMP TEAMBARRIER`, corresponding to `!$OMP BARRIER`, would have saved us for extra changes in the code. Because of the lack of `!$OMP TEAMPRIVATE`, an extra dimension had to be added to the affected variables, telling which team the variable belonged to. In lack of `!$OMP TEAMBARRIER`, an extra routine had to be created to fill this need. In OpenMP version 2.0, given that the nested directives are implemented, the `!$OMP BARRIER` directive will have the function of our `!$OMP TEAMBARRIER` when appearing within a nested parallel construct.

On $T_L = 1$ level $L = NL$ none of the teams must start to update cell values before all teams have copied needed ghost cell values from neighbors. Therefore a partial synchronization like "don't start to work on C, before everyone is finished in A", is desirable. In explicit message passing this synchronization comes as a free bonus with two-sided communication. In the work presented here it was solved by writing new routines for this purpose, using logical variables to tell if "A" was done and while-loops in front of "C" to do the waiting and checking if the other teams were done with "A". In Algorithm 3 Section 4.2, this is written as the **!$rb Synchronize** construct.

6 Conclusions and related work

In [BS01] we applied nested parallelism implemented in OpenMP to problems having a static number of relative few tasks, and obtained very good scalability. The main purpose of *this* paper has been to examine the possible gain of applying nested parallelism to a problem having a dynamically changing number of tasks. AMRCLAW has a built-in multilevel nature, as refined patches of unequal size are created and dismissed dynamically, and has therefore been used in this research.

The speedup-result when applying a nesting parallel AMR algorithm to AMRCLAW was okey, but not as great as obtained for the static task problems in [BS01]. We believe that the main reason for a more moderate scalability this time, is that the dynamic structure implies non-uniform data access, which increases the internal moving of data in a cache based system like the Origin 2000. However, having these difficulties in mind and comparing with the performance of two 1-level approaches, a fine-grained and a coarse-grained, we find the speedup quite satisfactory. We therefore conclude that for the cases tested in this paper, nested parallelism offers much better scaling opportunities than 1-level parallelism.

Scientists at Berkeley National Laboratory [RBL+99] have also parallelized the AMR algorithm, but with MPI [RBL+99]. In addition to the fact that the approach presented here uses OpenMP, the Berkeley Lab uses only 1 level of parallelism, and the total computational cost is approximated as the sum of the costs of time-stepping on the grids on the finest level. According to our study of computational cost in [BS02], the cost should be approximated as the sum of the costs of time-stepping on all levels, including the regridding, and this approximation has therefore been used in this paper.

Acknowledgments

This project has been supported by a grant from the Norwegian Supercomputing Program with computing time on Parallab's SGI/CRAY Origin 2000.

It is with great pleasure the author acknowledge the stimulating discussions with Prof. Tor Sørevik at University of Bergen.

References

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

[Ber95] Marsha Berger. Adaptive Mesh Refinement for Hyperbolic Conservation Laws. `http://cs.nyu.edu/cs/faculty/berger/amrsoftware.html`, 1995.

[BL97] Marsha J. Berger and Randall J. LeVeque. AMRCLAW, Adaptive Mesh Refinement + CLAWPACK. `http://www.amath.washington.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*, 9(2):185–194, 2001.

[BS02] Ragnhild Blikberg and Tor Sørevik. Cost and Savings of Adaptive Mesh Refinement. Technical report, January 2002.

[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.

[LL97] Randall J. LeVeque and Jan Olav Langseth. CLAWPACK 3.0, a Software Package for Conservation Laws and Hyperbolic Systems. `http://www.amath.washington.edu/~rjl/clawpack`, 1997.

[LLBM00] Randall J. LeVeque, Jan Olav Langseth, Marsha J. Berger, and Sorin. Mitran. CLAWPACK 4.0, a Software Package for Conservation Laws and Hyperbolic Systems. `http://www.amath.washington.edu/claw`, 2000.

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

[omp00] OpenMP Fortran Application Program Interface, version 2.0. `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.