Construction and Application of an AMR Algorithm for Distributed Memory Computers

Ralf Deiterding

California Institute of Technology,1200 East California Blvd., Mail-Code 158-79, Pasadena, CA 91125, ralf@cacr.caltech.edu

While the parallelization of blockstructured adaptive mesh refinement techniques is relatively straight-forward on shared memory architectures, appropriate distribution strategies for the emerging generation of distributed memory machines are a topic of on-going research. In this paper, a locality-preserving domain decomposition is proposed that partitions the entire AMR hierarchy from the base level on. It is shown that the approach reduces the communication costs and simplifies the implementation. Emphasis is put on the effective parallelization of the flux correction procedure at coarse-fine boundaries, which is indispensable for conservative finite volume schemes. An easily reproducible standard benchmark and a highly resolved parallel AMR simulation of a diffracting hydrogen-oxygen detonation demonstrate the proposed strategy in practice.

1 Introduction

The adaptive mesh refinement (AMR) method after Berger and Collela [2] is widely used for adaptive simulations on logically rectangular finite volume meshes. Instead of replacing single cells by finer ones, the AMR method constructs a hierarchy of properly nested refinement grids. The striking efficiency of this algorithm, in particular for instationary supersonic gas dynamical problems, was demonstrated by Berger and her collaborators in [1].

Up to now, various reliable implementations of the AMR method for single processor computers have been developed [3, 4]. Even implementations for parallel computers with shared memory architecture have reached a stable state [1]. Parallelism is an inherent feature of the AMR algorithm and in a shared memory environment simply the numerical solution on the whole sequence of grids has to be advanced in parallel to achieve a sufficient load-balancing. The question for an efficient parallelization strategy becomes more delicate for distributed memory machines, because the costs of communication can not be neglected anymore. Due to the technical difficulties in implementing

hierarchical adaptive methods in a distributed memory environment only few parallelization efforts are documented, cf. [11, 10, 7].

This paper describes a rigorous domain decomposition approach that partitions the entire hierarchy from the base level on. By employing ghost or halo cell regions, which are synchronized whenever the algorithm applies boundary conditions, an overlap between subgrids is constructed that allows most operations of the AMR algorithm to be carried out strictly local. After a brief characterization of the employed finite volume methods in Sec. 2, we review the sequential AMR algorithm in Sec. 3. In Sec. 4, we specify the domain decomposition and discuss the necessary extensions of the previously described subroutines in parallel. Sec. 5 presents parallel AMR simulations for the Euler equations of gas dynamics obtained with our public domain code AMROC [6] on typical Linux-Beowulf-clusters.

2 Finite Volume Schemes

The Berger-Collela AMR method is a dynamic mesh adaptation approach, which is tailored especially for the adaptive numerical solution of hyperbolic conservation laws

$$\partial_t \mathbf{q}(\mathbf{x},t) + \nabla \cdot \mathbf{f}(\mathbf{q}(\mathbf{x},t)) = \mathbf{0}\,, \quad \mathbf{x} = (x_1,\ldots,x_d)^T \in \mathbb{R}^d\,, \quad t \in \mathbb{R}_0^+ \tag{1}$$

on logically rectangular finite volume (FV) meshes. For simplicity, we restrict our attention to the two-dimensional case and assume an equidistant FV discretization of the computational domain $G_0 \subset \mathbb{R}^2$ with mesh widths Δx_1, Δx_2 and a constant time step Δt. The discrete mesh points are defined by $(x_1^i, x_2^j) := \left(\left(i+\frac{1}{2}\right)\Delta x_1, s\left(j+\frac{1}{2}\right)\Delta x_2\right)$, $i,j \in \mathbb{Z}$ and $t_\kappa := \kappa \Delta t\,,\ \kappa \in \mathbb{N}_0$. In each point (x_1^i, x_2^j, t^κ) we define a discrete value $\mathbf{Q}_{ij}^\kappa$ as an approximation to the vector of state $\mathbf{q}(\mathbf{x},t)$ averaged over the control volume $[x_1^{i-1/2}, x_1^{i+1/2}) \times [x_2^{j-1/2}, x_2^{j+1/2})$. These values are updated by a time-explicit *conservative* $(2s+1)^2$-point FV scheme of the form

$$\mathcal{H}^{(\Delta t)}:\ \mathbf{Q}_{ij}^{\kappa+1} = \mathbf{Q}_{ij}^\kappa - \frac{\Delta t}{\Delta x_1}\left(\mathbf{F}^1_{i+\frac{1}{2},j} - \mathbf{F}^1_{i-\frac{1}{2},j}\right) - \frac{\Delta t}{\Delta x_2}\left(\mathbf{F}^2_{i,j+\frac{1}{2}} - \mathbf{F}^2_{i,j-\frac{1}{2}}\right) \tag{2}$$

with *numerical fluxes* given by

$$\mathbf{F}^1_{i+\frac{1}{2},j} = \mathbf{F}_1(\mathbf{Q}^\kappa_{i-s+1,j-s},\ldots,\mathbf{Q}^\kappa_{i+s,j+s}), \mathbf{F}^2_{i,j+\frac{1}{2}} = \mathbf{F}_2(\mathbf{Q}^\kappa_{i-s,j-s+1},\ldots,\mathbf{Q}^\kappa_{i+s,j+s}).$$

For vanishing boundary fluxes, scheme (2) satisfies the important discrete conservation property $\sum_{i,j}\mathbf{Q}_{jk}^{\kappa+1} = \sum_{i,j}\mathbf{Q}_{jk}^\kappa$, which is essential for the correctness of the approximation, if Eq. (1) admits discontinuous solutions, as it is the case e.g. for Euler equations. The numerical fluxes in (2) are often evaluated by solving a Riemann problem between neighboring cells approximately. In this case, a typical stability condition for $\mathcal{H}^{(\Delta t)}$ could be

$$C_{CFL} := \max_{j,k}\Big(S_{j+\frac{1}{2},k}\frac{\Delta t}{\Delta x_1}\,,\; S_{j,k+\frac{1}{2}}\frac{\Delta t}{\Delta x_2}\Big) \le 1\,, \tag{3}$$

where $S_{j+\frac{1}{2},k}$, $S_{j,k+\frac{1}{2}}$ denote the maximal signal speeds in both space directions according to the approximative solution of the Riemann problems at the cell interfaces.

3 Blockstructured Adaptive Mesh Refinement

A significant advantage of the blockstructured idea over other mesh refinement strategies is that the update operator $\mathcal{H}^{(\cdot)}$ only needs to be implemented on a single *uniform* Cartesian grid G, where s layers of auxiliary cells (ghost or halo cells) around G should be employed to define discrete boundary conditions. Cells flagged by various error indicators are clustered dynamically at run-time into non-overlapping rectangular subgrids $G_{l,m}$ that define the domain of an entire level $l = 0, \ldots, l_{\max}$ by

$$G_l := \bigcup_{m=1}^{M_l} G_{l,m}\,.$$

Refinement grids are derived recursively from coarser ones and a hierarchy of successively embedded levels is thereby constructed. All mesh widths on level l are r_l-times finer than on level $l-1$, i.e. $\Delta t_l := \Delta t_{l-1}/r_l$ and $\Delta x_{n,l} := \Delta x_{n,l-1}/r_l$ with $r_l \in \mathbb{N}, r_l \ge 2$ for $l > 0$ and $r_0 = 1$, and a time-explicit FV scheme (in principle) remains stable under a condition like (3) on all levels of the hierarchy. The recursive integration order visualized in the left sketch of Fig. 1 is an important difference to unstructured adaptation strategies and is one of the main reasons for the high efficiency of the approach.

The numerical scheme is applied on level l by calling the single-grid routines $\mathcal{H}^{(\cdot)}$ in a loop over all subgrids $G_{l,m}$. The execution of the numerical loop in `UpdateLevel()` in Alg. 1 requires the previous setting of the ghost cell values. Three types of boundary conditions have to be considered in the sequential case, see right sketch of Fig. 1. Cells outside of the root domain G_0 are used to implement physical boundary conditions. Ghost cells in G_l have a unique interior cell analogue and are set by copying the data value from the grid, where the interior cell is contained (synchronization). On the root level no further boundary conditions need to be considered, but for $l > 0$ also internal boundaries can occur. They are set by a conservative time-space interpolation from two previously calculated time steps of level $l-1$.

Beside a general data tree that stores the topology of the hierarchy, the AMR method requires at most two regular arrays assigned to each subgrid, which contain the discrete vector of state $\mathbf{Q}$ for the actual and updated time step. In the following, we denote by $\mathbf{Q}^l(t)$ and $\mathbf{Q}^l(t + \Delta t_l)$ the unions of these arrays on level l. The regularity of the input data for the numerical routines allows high performance on vector and super-scalar processors and

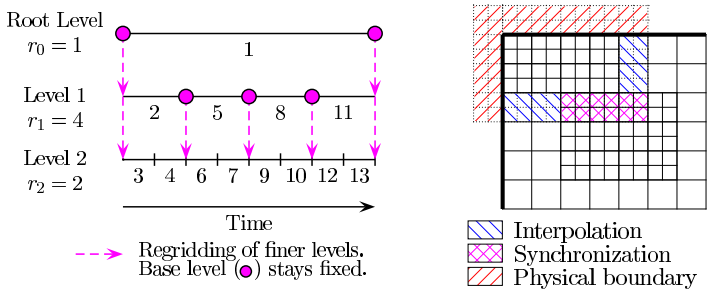


Fig. 1. Left: Recursive integration order. Right: Sources of ghost cell values.

cache optimizations. Small data arrays are effectively avoided by leaving coarse level data structures untouched, when higher level grids are created. Values of cells covered by finer subgrids are overwritten by averaged fine grid values subsequently.

3.1 Conservative Flux Correction

Replacing coarse cell values by averaged fine grid values modifies the numerical stencil on the coarse grid. In general the important property of conservation is lost. A flux correction is necessary that introduces the involved fine grid fluxes into Eq. (2). In two and three space dimensions hanging nodes additionally have to be considered. As an example we consider cell (j,k) in Fig. 2 on level l. After the numerical update on level l a correction term associated to the boundary of level $l+1$ is initialized by $\delta\mathbf{F}^{1,l+1}_{j-\frac{1}{2},k} := -\mathbf{F}^{1,l}_{j-\frac{1}{2},k}$. During the r_{l+1} update steps of level $l+1$ all necessary fine level fluxes are accumulated, i.e.

$$\delta\mathbf{F}^{1,l+1}_{j-\frac{1}{2},k} := \delta\mathbf{F}^{1,l+1}_{j-\frac{1}{2},k} + \frac{1}{r^2_{l+1}} \sum_{\nu=0}^{r_{l+1}-1} \mathbf{F}^{1,l+1}_{v+\frac{1}{2},w+\nu}(t+\mu\Delta t_{l+1}) \qquad (4)$$

with $\mu = 0,\ldots,r_{l+1}-1$. After the integration of the fine level, the correction is applied by calculating

$$\tilde{\mathbf{Q}}^{\kappa+1}_{jk} := \tilde{\mathbf{Q}}^{\kappa+1}_{jk} + \frac{\Delta t_l}{\Delta x_{1,l}}\,\delta\mathbf{F}^{1,l+1}_{j-\frac{1}{2},k}\,.$$

The edge- or face-centered flux correction terms $\delta\mathbf{F}^{n,l+1}$ have to be stored along the boundaries, where a level $l>0$ abuts the next coarser level. To avoid the usage of the numerical fluxes $\mathbf{F}^n$ on the entire level, the grid-wise application of the numerical scheme and the computation of the correction terms are effectively combined in `UpdateLevel()` in the loop over all all grids on level l.

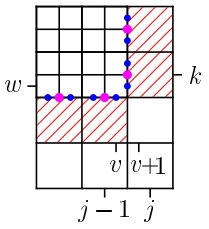


Fig. 2. Location of numerical fluxes required for flux correction. Cells to correct are shaded.

```
AdvanceLevel(l)
 Repeat r_l times
   Set ghost cells of Q^l(t)
   If time to regrid
     Regrid(l)
   UpdateLevel(l)
   If level l+1 exists
     Set ghost cells of Q^l(t + Δt_l)
     AdvanceLevel(l + 1)
     Average Q^{l+1}(t + Δt_l) onto
       Q^l(t + Δt_l)
     Correct Q^l(t + Δt_l) with δF^{n,l+1}
   t := t + Δt_l
```

Alg. 1. Recursive AMR algorithm.

```
Regrid(l)
 For ι = l_c Downto l Do
   Flag N^ι according to Q^ι(t)
   If level ι + 1 exists?
     Flag N^ι below Ğ_{ι+2}
   Flag buffer zone on N^ι
   Generate Ğ_{ι+1} from N^ι
 Ğ_l := G_l
 For ι = l To l_c Do
   CĞ_ι := G_0\Ğ_ι, Ğ_{ι+1} := Ğ_{ι+1}\CĞ_ι^1
 Recompose(l)
```

Alg. 2. Regridding procedure.

3.2 Recursive Grid Generation

The basic recursive AMR algorithm is formulated in Alg. 1. Except the regridding procedure, all operations have already been explained. New refinement grids on all higher levels are created by calling `Regrid()` from level l. Level l by itself is not modified. To consider the nesting of the level domains already in the grid generation, Alg. 2 starts at the highest refineable level l_c, where $0 \le l_c < l_{\max}$. The refinement flags are stored in grid-based integer arrays N^ι. A clustering algorithm [1] is necessary to create a new refinement $\breve{G}_{\iota+1}$ on basis of N^ι until the ratio between flagged and all cells in every new grid $\breve{G}_{\iota+1,m}$ is above a prescribed threshold $0 < \eta_{tol} < 1$.

Before the new grids $\breve{G}_{\iota+1}$ can be used to replace $G_{\iota+1}$, the proper nesting of the new refinement grids has to be enforced over the modified hierarchy. In Alg. 2 we evaluate the invalid region for level $\iota + 1$ by calculating the complement $C\breve{G}_\iota := G_0 \backslash \breve{G}_\iota$ of the next coarser level domain $\breve{G}_\iota$ in G_0 and by enlarging $C\breve{G}_\iota$ by one additional layer of cells on level ι. We denote this enlarged region by $C\breve{G}_\iota^1$. The operation $\breve{G}_{\iota+1} := \breve{G}_{\iota+1} \backslash C\breve{G}_\iota^1$ then eliminates all invalid regions from the new level domain $\breve{G}_{\iota+1}$.

The reinitialization of the hierarchy is done in the subroutine `Recompose(`l`)`, which is formulated in Alg. 3a. In particular, grid-based auxiliary data $\breve{\mathbf{Q}}(\breve{G}_\iota, t)$ is necessary to reorganize the vector of state. Cells in newly refined regions $\breve{G}_\iota \backslash G_\iota$ are initialized by interpolation, values of cells in $\breve{G}_\iota \cap G_\iota$ are copied. As interpolation requires the previous synchronized reorganization of $\mathbf{Q}^{\iota-1}(t)$, the recomposition algorithm traverses the hierarchy upwards.

4 Parallelization by Domain Decomposition

We follow a rigorous domain decomposition approach and partition the AMR hierarchy from the root level on. We assume a parallel machine with P identical nodes and split the root domain G_0 into P non-overlapping portions G_0^p,

$p = 1, \ldots, P$ by

$$G_0 = \bigcup_{p=1}^{P} G_0^p \quad \text{with} \quad G_0^p \cap G_0^q = \emptyset \ \text{ for } p \neq q \ .$$

The key idea now is that all higher level domains are required to follow the decomposition of the root level, i.e.

$$G_l^p := G_l \cap G_0^p \ . \tag{5}$$

Condition (5) can cause the splitting of a subgrid $G_{l,m}$ into multiple subgrids on different processors. Under requirement (5) we estimate the work on an arbitrary subdomain $\Omega \subset G_0$ by

$$\mathcal{W}(\Omega) = \sum_{l=0}^{l_{\max}} \left[\mathcal{N}_l(G_l \cap \Omega) \prod_{\nu=0}^{l} r_\nu \right] \ . \tag{6}$$

Herein, $\mathcal{N}_l(\cdot)$ denotes the total number of FV cells on level l in the given domain. The product in (6) is used to consider the time step refinement. A nearly equal distribution of the work necessitates

$$\mathcal{L}^p := \frac{P \cdot \mathcal{W}(G_0^p)}{\mathcal{W}(G_0)} \approx 1 \quad \text{for all } p = 1, \ldots, P \ . \tag{7}$$

A considerable advantage of the proposed decomposition strategy is the reduction of the communication costs. Together with the use of ghost cells our approach allows an almost local execution of Alg. 1. In particular, the inter-level operations interpolation and averaging remain strictly local. The only parallel operations that have to be considered additionally are the parallel ghost cell synchronization and the application of flux correction terms across processor borders. Especially `UpdateLevel()` need not be modified. Apparently, the new vector of state on each subgrid $G_{l,m}^p$ and the fluxes can be computed strictly local, but also the evaluation of the correction terms does not require communication.

4.1 Local Calculation of Flux Corrections

To illustrate this, we assume a parallel border in Fig. 2 at $j - \frac{1}{2}$. Let cell (j,k) be contained in G_l^q and let cell (v,w) be contained in G_{l+1}^p. Then the necessary correction term $\delta\mathbf{F}^{1,l+1}_{j-1/2,k}$ resides on node p, because it is assigned to the fine level. Its initialization requires the coarse grid flux $\mathbf{F}^{1,l}_{j-1/2,k}$. This flux is available on node p, because the basic AMR strategy ensures that below (v,w) an interior coarse cell $(j-1,k)$ exists having $\mathbf{F}^{1,l}_{j-1/2,k}$ as flux into a ghost cell (j,k). On the other hand, $\mathbf{F}^{1,l}_{j-1/2,k}$ is also computed on node q, where (j,k) is interior and $(j-1,k)$ is a ghost cell. As the ghost cells have been synchronized before the numerical update, the same boundary flux is calculated on both nodes, cf. Fig. 3. The fine grid fluxes $\mathbf{F}^{1,l+1}_{v+1/2,w+\nu}$ are only available on p, because no abutting interior fine grid cell exists on q. As the correction term $\delta\mathbf{F}^{1,l+1}_{j-1/2,k}$

is also stored on p the summation in (4) remains local. The only operation of the flux correction that necessarily requires communication is the final application of the correction terms as mentioned in the previous section. But the communication costs are minor, because the corrections are only necessary along lower-dimensional domains.

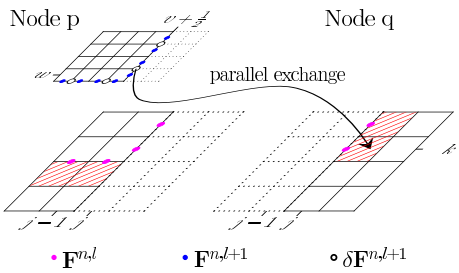


Fig. 3. Flux correction in parallel.

4.2 Parallel Grid Generation

Analogous to Alg. 1 the regridding in Alg. 2 is hardly affected by the parallelization. The flagging of cells on each level can be done locally. If a refinement criterion requires auxiliary time steps (i.e. error estimation by Richardson extrapolation, cf. [2]), additional synchronizations will be necessary, but this does not affect Alg. 2. The only operation in Alg. 2 that needs special attention is the clustering.

The clustering algorithm could be executed strictly locally on $N(G_\iota^p)$ or it could be executed on the data of the entire level $N(G_\iota)$. Usually, the results will be identical for $\eta_{tol} = 1$ only. To avoid the expensive global concatenation of all data sets $N(G_\iota^p)$ to $N(G_\iota)$, we execute the clustering algorithm strictly locally and communicate just the results $\breve{G}_{\iota+1}^p$ to obtain the global list $\breve{G}_{\iota+1} = \bigcup_p \breve{G}_{\iota+1}^p$. To consider a buffer zone of b cells before local clustering, the grid-based integer arrays N^ι are extended by b ghost cells. A parallel synchronization of these ghost cells before creating the buffer zone locally ensures the appropriate flagging of interior cells.

The main changes in the regridding procedure are in `Recompose(`l`)`. Instead of Alg. 3a we apply Alg. 3b. Due to our distribution strategy we now have to consider a complete reorganization of the entire hierarchy even for a regridding at a higher level. In particular, the whole relevant data of levels with $\iota \leq l$ have to be copied. Like the synchronization operation, these copy operations are partially local and parallel. For levels with $\iota < l$ the relevant data is $\mathbf{Q}^\iota(t)$, $\mathbf{Q}^\iota(t + \Delta t_\iota)$ and $\delta\mathbf{F}^{n,\iota}$, for level l we have to copy $\mathbf{Q}^l(t)$ and $\delta\mathbf{F}^{n,l}$. The initialization of a level with $\iota > l$ is in principle identical to Alg. 3a. Interpolation is a strictly local operation, provided that the next coarser level has already been reorganized. The copy operation is a combination of local and parallel copy.

Alg. 3b is significantly more complex than Alg. 3a, because it considers the general case of a complete parallel redistribution of the AMR hierarchy even at higher level time steps. However, in practice it usually suffices to allow this operation only on the root level. Under this simplification, Alg. 3b reduces mostly to Alg. 3a. The creation of the new load-balanced distributions G_0^p

```
Recompose(l) - sequential

For ι = l + 1 To l_c + 1 Do

  Interpolate Q^{ι-1}(t) on Q̆^ι(t)

  Copy Q^ι(t) on Q̆^ι(t)
  Set ghost cells of Q̆^ι(t)
  Q^ι(t) := Q̆^ι(t)
  G_ι := Ğ_ι
```

Alg. 3a. Sequential recomposition.

```
Recompose(l) - parallel
  Derive G_0^p from
        {G_0, ..., G_l, Ğ_{l+1}, ..., Ğ_{l_c+1}}
  For ι = 0 To l_c + 1 Do
    If ι > l
      Ğ_ι^p := Ğ_ι ∩ G_0^p
      Interpolate Q^{ι-1}(t) on Q̆^ι(t)
    else
      Ğ_ι^p := G_ι ∩ G_0^p
      If ι > 0
        Copy δF^{n,ι} onto δF̆^{n,ι}
        δF^{n,ι} := δF̆^{n,ι}
    If ι ≥ l then ν_ι = 0 else ν_ι = 1
    For ν = 0 To ν_ι Do
      Copy Q^ι(t + νΔt_ι) on Q̆^ι(t + νΔt_ι)
      Set ghost cells of Q̆^ι(t + νΔt_ι)
      Q^ι(t + νΔt_ι) := Q̆^ι(t + νΔt_ι)
    G_ι^p := Ğ_ι^p, G_ι := ⋃_p G_ι^p
```

Alg. 3b. Parallel recomposition.

then has to be considered just for the case $l = 0$ and only $\mathbf{Q}^\iota(t)$ has to be copied over processor borders.

4.3 Partitioning

It is evident, that the overall efficiency of the chosen parallelization strategy depends especially on the first step of Algorithm 3b, the partitioning algorithm. This algorithm has to meet several requirements. It must balance the estimated workload, while the parallel synchronization costs should be small. A slight change of the hierarchy should require only a moderate data redistribution. The algorithm must be fast, because it is carried out on-the-fly.

Distribution strategies based on space-filling curves seem to give an acceptable compromise between these partially competing requirements. A space-filling curve defines a continuous mapping from $[0, 1]$ onto $[0, 1]^d$ and is well suited to define an ordered sequence on the root level cells of a blockstructured domain. This sequence can easily be split into portions of equal size yielding load-balanced new distributions G_0^p. As space-filling curves are constructed recursively, they are locality-preserving by definition and naturally avoid an excessive redistribution overhead. Further on, the surface is small, which reduces the synchronization costs.

Our present implementation utilizes a partitioning algorithm based on Hilbert's space-filling curve [9]. The Figs. 4 and 6 display domain decompositions derived with this algorithm for the work estimation formula (6). Apparently, the extensions of the domain assigned to each node vary remarkably, but the workloads according to (7) always differ by less than 5%.

5 Computational Results

We use a standard test for Euler equations of a single polytropic gas to evaluate the proposed parallelization strategy within the MPI-based AMROC implementation [6]. A homogeneous circular region of high pressure and density expands in an enclosed box. After a few time steps, the initial discontinuity separates into a rapidly expanding discontinuous shock wave, a following slower contact discontinuity and a collapsing smooth rarefaction wave.

We utilize a base grid of 150×150 cells and apply a two-level refinement with the factors $r_1 = 2$ and $r_2 = 4$. About 200 root level grid time steps with $C_{CFL} \approx 0.8$ to $t_{end} = 0.5$ were computed, where the Clawpack implementation of the Wave Propagation Method [8] with the approximate Riemann solver of Roe was employed as numerical update routine. A repartitioning of the hierarchy was done only at root level time steps, cf. Sec. 4.2. A standard Linux-Beowulf-cluster of Pentium-III-1 GHz CPUs connected with Fast Ethernet (effective bandwidth ≈ 40 MB) was used for the benchmarks. Exemplary results on eight nodes are shown in Fig. 4. While the AMROC computation on one node required 152 min, the execution time decreased to remarkable 13.9 min on 16 nodes. Tab. 1 shows a breakdown of the computational time for the most important AMR operations. For one node the fractions spent in different parts of the code are in good agreements with the results in [2] and at least for a moderate number of computing nodes we achieve an acceptable parallel efficiency.

Table 1. Computational time on P nodes.

Task [%]	P=1	P=2	P=4	P=8	P=16
Update by $\mathcal{H}^{(\cdot)}$	86.6	83.4	76.7	64.1	51.9
Flux correction	1.2	1.6	3.0	7.9	10.7
Boundary setting	3.5	5.7	10.1	15.6	18.3
Recomposition	5.5	6.1	7.4	9.9	14.0
Misc.	4.9	3.2	2.8	2.5	5.1
Time [min]	**151.9**	**79.2**	**43.4**	**23.3**	**13.9**
Efficiency [%]	**100.0**	**95.9**	**87.5**	**81.5**	**68.3**

In order to demonstrate that our parallelization approach is also well suited for cutting-edge AMR simulations, we briefly present exemplary results for a two-dimensional hydrogen-oxygen detonation propagating out of a tube into unconfinement. The simulation reproduces the critical width for square tubes and is in perfect qualitative agreement with experimental results. The computation was run effectively in less than 4 days real time on a Linux-Beowulf-cluster of 48 CPUs and spent ≈ 2000 h CPU time in the update operator $\mathcal{H}^{(\cdot)}$, which was a special approximative Riemann solver for multi-component Euler equations with general equation of state. The reaction terms according to a detailed non-equilibrium reaction mechanism were incorporated numerically into $\mathcal{H}^{(\cdot)}$ with a fractional step method and required the additional solution of a *stiff* initial value problem in each FV cell. See [5] for details.

As detonation simulations require an extraordinarily high local resolution to capture the influence of the chemical kinetics correctly, the computation benefits remarkably from dynamic mesh adaptation. The graphics in Fig. 5

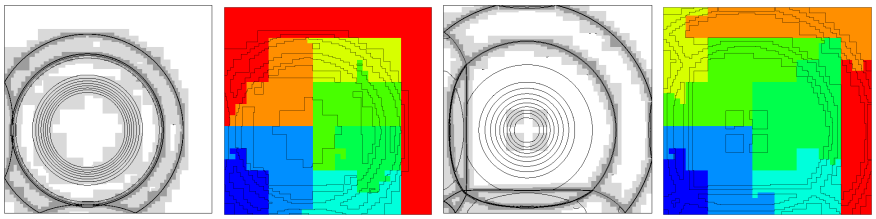
display the solution on the refinement levels 240 μs after the detonation has left the tube (730 root level time steps with $C_{CFL} \approx 0.8$, one half of the domain was simulated) and the enormous efficiency of the refinement is apparent. The base grid used 508×288 cells and four levels of refinement with $r_{1,2,3} = 2$, $r_4 = 4$, which corresponds to ≈ 150 M cells, but at the time step displayed the simulation uses less than 3.0 M cells on all levels.

6 Conclusions

We have described a locality-preserving parallelization strategy for the block-structured AMR algorithm after Berger and Colella, which is tailored especially for distributed memory machines. The approach is based on domain decomposition and reduces the communication costs. In particular, the important flux correction procedure, which can become quite complicated in a distributed memory environment, can be implemented with ease. Benchmark calculations with our MPI-based implementation AMROC show promising parallel speed-up and we were able to obtain exceptional detonation simulations with the framework on standard Linux-Beowulf-clusters, cf. [5].

References

1. J. Bell, M. Berger, J. Saltzman, and M. Welcome. Three-dimensional adaptive mesh refinement for hyperbolic conservation laws. *SIAM J. Sci. Comp.*, 15(1):127–138, 1994.
2. M. Berger and P. Colella. Local adaptive mesh refinement for shock hydrodynamics. *J. Comput. Phys.*, 82:64–84, 1988.
3. M. Berger and R. LeVeque. Adaptive mesh refinement using wave-propagation algorithms for hyperbolic systems. *SIAM J. Num. Anal.*, 35(6):2298–2316, 1998.
4. W. Crutchfield and M. L. Welcome. Object-oriented implementation of adaptive mesh refinement algorithms. *J. Scientific Programming*, 2:145–156, 1993.
5. R. Deiterding. *Parallel adaptive simulation of multi-dimensional detonation structures.* PhD thesis, Techn. Univ. Cottbus, Sep 2003.
6. R. Deiterding. AMROC - Blockstructured Adaptive Mesh Refinement in Object-oriented C++. Available at http://amroc.sourceforge.net, Oct 2003.
7. S. R. Kohn and S. B. Baden. A parallel software infrastructure for structured adaptive mesh methods. In *Proc. of the Conf. on Supercomputing '95*, Dec 1995.
8. R. J. LeVeque. Wave propagation algorithms for multidimensional hyperbolic systems. *J. Comput. Phys.*, 131(2):327–353, 1997.
9. M. Parashar and J. C. Browne. On partitioning dynamic adaptive grid hierarchies. In *Proc. 29th Annual Hawaii Int. Conf. on System Sciences*, Jan 1996.
10. M. Parashar and J. C. Browne. System engineering for high performance computing software: The HDDA/DAGH infrastructure for implementation of parallel structured AMR. In *Structured Adaptive Mesh Refinement Grid Methods*, IMA Volumes in Mathematics and its Applications. Springer, 1997.
11. C. A. Rendleman, V. E. Beckner, M. Lijewski, W. Crutchfield, and J. B. Bell. Parallelization of structured, hierarchical adaptive mesh refinement algorithms. *Computing and Visualization in Science*, 3, 2000.



After 38 time steps After 79 time steps

Fig. 4. Circular Riemann problem in an enclosed box. Isolines of density on two refinement levels (indicated by gray scales) and distribution to eight nodes (indicated by different colors).

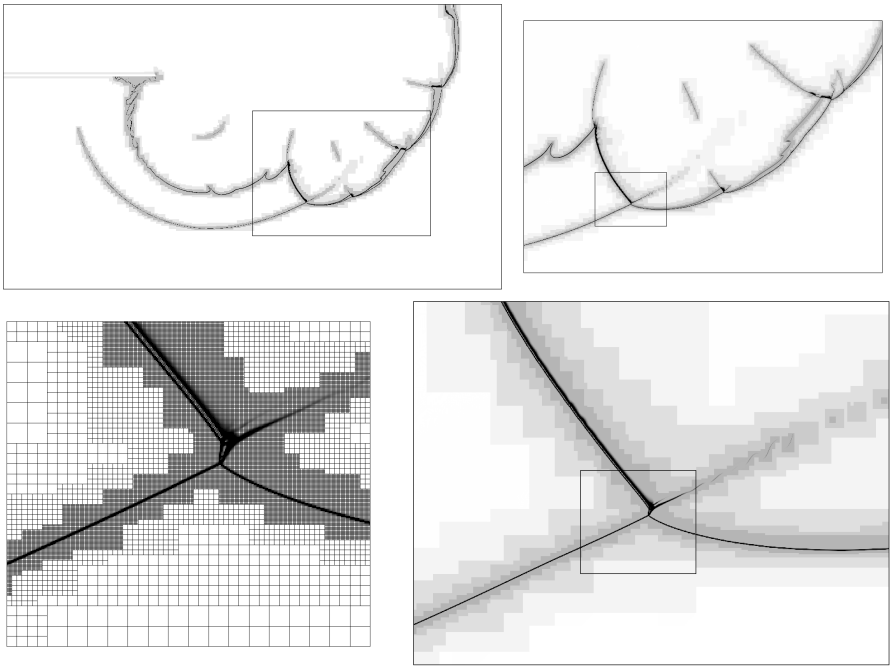


Fig. 5. (Upper four graphics.) Planar detonation diffraction. Density distribution on four refinement levels 240 μs after the detonation has left the tube. Multiple zooms are necessary to display the finite volume cells.

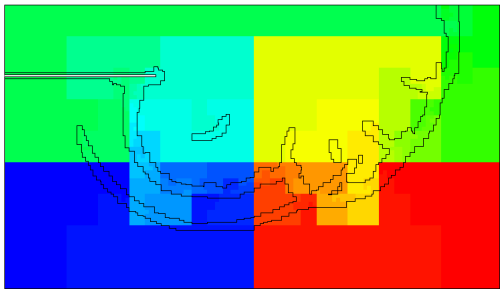


Fig. 6. Planar detonation diffraction. Distribution of computational domain to 48 nodes.