A Meta-Partitioner for run-time selection and evaluation of multiple partitioning algorithms for SAMR grid hierarchies

Henrik Johansson
Dept. of Information Technology, Scientific Computing
Uppsala University
Box 337, SE-751 05 Uppsala, Sweden
E-mail: henrik.johansson@it.uu.se

Abstract

Parallel structured adaptive mesh refinement (SAMR) methods increase the efficiency of the numerical solution to partial differential equations. These methods use an adaptive grid hierarchy to dynamically assign computational resources to areas with large solution errors. The grid hierarchy needs to be repeatedly re-partitioned and distributed over the processors but no single partitioning algorithm performs well for all hierarchies. This paper presents an extended and improved version of the Meta-Partitioner, a partitioning framework that uses the state of the application to autonomously select, configure, invoke, and evaluate partitioning algorithms during run-time. The performance of the partitioning algorithms are predicted using historical performance data for grid hierarchies similar to the current hierarchy. At each re-partitioning, a user-specified number of partitioning algorithms are selected and invoked. When multiple partitionings are constructed, the performance of each partitioning is evaluated during run-time and the best partitioning is selected. The performance evaluation shows huge improvements for the two most performance-inhibiting factors — the load imbalance and the synchronization delays. On average, the load imbalance is increased by only 11.5% and the synchronization delays by 13.6% compared to the optimal results from 768 different hybrid partitioning algorithms.

1 Introduction

Structured adaptive mesh refinement (SAMR) is employed to increase the efficiency of the numerical solution to partial differential equations in areas like computational fluid dynamics [6, 14, 25], numerical relativity [37, 39], astrophysics [9, 18, 31], and hydrodynamics [30]. In SAMR, areas in the computational domain with large solution errors are identified and overlayed with grids having a finer resolution, resulting in a dynamic and adaptive grid hierarchy.

Good parallel performance for SAMR applications requires efficient use of the underlying computer. This paper presents an extended and improved version of the Meta-Partitioner, a partitioning framework that autonomously selects, configures, invokes, and evaluates partitioning algorithms during run-time with respect to the state of the application. The Meta-Partitioner is the culmination of a larger research effort that aims to decrease the execution time of general parallel SAMR applications running on general parallel computers. Previous research has resulted in several partitioners and new theoretical models to asses the partitioning needs of SAMR applications [20–22, 24, 41, 43–45].

Previous research has also shown that no single partitioning algorithm performs well for all SAMR grid hierarchies [38, 41]. To obtain good parallel performance, the partitioning algorithm must be dynamically selected during run-time. The resulting execution time is always dependent on a combination of a number of performance-inhibiting factors like load imbalance, communication volumes, and synchronization delays. The relative importance of these performance-inhibiting factors will generally vary during the execution of the application. At one stage, the interconnect might be overloaded due to other applications executing concurrently. At another stage, the properties of the grid hierarchy can make it hard to achieve an acceptable load imbalance. Thus, the partitioning effort must be continuously focused on the most performance-inhibiting factors.

Recently, an initial implementation of the Meta-Partitioner was presented [22]. At each re-partitioning, the initial version selected the partitioning algorithm that was predicted to result in the best performance for the current grid hierarchy. An evaluation showed that the Meta-Partitioner consistently produced partitionings with a significantly better performance than the partitionings generated by the average performing algorithms. For each of the applications examined, the Meta-Partitioner also resulted in equal to or better performance than the single partitioning algorithm with the best overall performance. It was also shown that large performance improvements are possible if the performance of several alternative partitionings can be evaluated during run-time.

In this paper, an extended and improved version of the Meta-Partitioner is presented. In the new version, multiple candidate partitionings can be constructed at each re-partitioning. A re-designed and substantially faster version of the SAMR simulator [10] makes it possible to evaluate the performance of the resulting partitionings during run-time. Hence, the choice of candidate partitioning algorithms is based on predicted performance, but the selected partitioning will have the best performance among the resulting partitionings.

The rest of this paper is organized as follows. In Section 2, an overview of structured adaptive mesh refinement and partitioning approaches is presented. Section 3 contains a description of the design and implementation of the Meta-Partitioner. The experimental setup of the performance evaluation is described in Section 4 and the results are presented in Section$\tilde{5}$. Finally, Section 6 contains the conclusions.

2 SAMR and partitioning

Methods using structured adaptive mesh refinement start with a coarse base grid that covers the entire computational domain [6, 7]. During run-time, regions with large solution errors are identified and overlaid with grids having a higher resolution. If the error on a coarse grid is later found to be small, the overlayed grids can be removed. As the execution progresses, grid patches will be created, moved and deleted, resulting in a dynamic grid hierarchy.

Information is frequently exchanged between grid patches during run-time. Boundary data for a refined grid patch is typically obtained from adjacent patches or patches on the next lower level. After the solution is updated, the results are projected down from finer to coarser levels to increase the solution accuracy. Thus, data flows both over the borders of neighboring patches and between patches on different refinement levels.

Frameworks for SAMR include Paramesh [29, 33], SAMRAI [48], GrACE [35], AMROC [3, 13], Chombo [11, 12], Enzo [15, 32], and Overture [19].

2.1 Partitioning of SAMR grid hierarchies

Parallel SAMR applications present a challenging resource allocation problem when the dynamic grid hierarchy is partitioned and assigned to processors. Optimally, the invoked partitioning algorithm should simultaneously minimize all partition-related performance-inhibiting factors like load imbalance, communication volumes, synchronization delays, and data migration. However, it is unrealistic to search for the optimal partitioning [16]. Instead, the Meta-Partitioner needs to trade-off the performance-inhibiting factors in accordance with the current characteristics of the application and the computer system.

Most partitioning algorithms can be classified into one of three main categories. For *patch-based partitioners* [4, 12, 24, 26], the distribution decision is made independently for each grid patch or refinement level. A grid patch may be kept on the local processor or moved entirely to another processor. If the grid patch is large, it can be split. The main advantage of the patch-based approach is a small load imbalance. Also, depending on the implementation, re-partitioning at re-griding can be avoided. Shortcomings are high communication volumes and potentially long synchronization delays. Also, patch-based algorithms have an inability to exploit available parallelism across different levels of refinement.

Domain-based partitioners [5, 34, 38] partition the physical domain, rather than the grids themselves. The domain is partitioned along with all overlaid grids from all refinement levels. Generally, the workload of the overlaid grids is projected down to the coarse base grid, reducing the problem to the partitioning of a single grid that has a heterogeneous workload. The advantages are elimination of inter-level communication and better exploitation of all available parallelism between different levels of refinement. The main disadvantage is an intractable load imbalance for deep hierarchies that can be further amplified by even larger imbalances on the individual refinement levels. Another drawback is the occurrence of “bad cuts” that results in increased overhead costs [41].

Hybrid partitioners [27, 41] combine the patch-based and domain-based partitioning approaches to avoid their respectively shortcomings — the high communication volumes for patch-partitioners and the intractable load imbalance for domain-based partitioners. Most hybrid partitioners use a 2-step partitioning approach. The first step use domain-based techniques to generate meta-partitionings that are mapped to a group of processors. The second step uses a combination of domain and patch-based techniques to optimize the distribution of each meta-partitioning within its processor group [27].

An example of a hybrid partitioner is Nature+Fable [41], from the outset designed to form the basis of the Meta-Partitioner. Nature+Fable uses domain-based techniques to separate the unrefined parts of the coarse base grid from the refined parts. For each separate refined area, adjacent refinement levels are clustered two-by-two into bi-levels. In a bi-level, the highest refinement level is partitioned using the patch-based approach. The resulting partitioning is then projected down in a domain-based fashion onto the lower refinement level. Thus, inter-level communication is avoided inside each bi-level, while domain-based partitioning is never performed on more than two refinement levels to achieve a small load imbalance. The partitioning process in Nature+Fable is governed by a large set of parameters and each parameter setting can be regarded as a separate partitioning algorithm. Several performance evaluations of Nature+Fable have shown good results [23, 41].

Generally, the load imbalance and synchronization delays have a larger impact on the execution time than the other performance-inhibiting factors (e.g. data migration and communication volumes) [23]. Before the solution can be advanced to the next time step, all processors must have finished their computations. Thus, the processor having the largest workload will determine the computational time.

Application	Computational time (s)	Synchronization time (s)	Total time (s)
Ramp	1381.2	808.6	3035.1
ShockTurb	2618.4	562.1	4270
ConvShock	1810.7	2262.4	17102
Spheres	1843.5	1141.2	7405

Table 1: Comparison between the computational time, synchronization time, and total execution time for four example applications from the SAMR framework AMROC [3] and a domain-based partitioning algorithm. Except for the computational time and the synchronization time, the third major component of the total time is the determination of internal data exchanges between grids assigned to the same processor and external communications between grids assigned to different processors. The data origins from sixteen processor executions on the ALC parallel computer at Lawrence Livermore National Laboratory [1]. All data courtesy of Ralf Deiterding.

During run-time, processors frequently need to synchronize and exchange data with each other. Often, one of the involved processors is busy computing while the others are forced to be idle. The time spent waiting for data can be significant and is often of the same magnitude as the computational time (see Table 1). In this paper, a synchronization penalty (see Section 4.1) is computed as an approximation of the synchronization delays [42].

3 The Meta-Partitioner

The Meta-Partitioner is a partitioning framework that autonomously selects, configures, invokes, and evaluates the partitioning algorithms that are predicted to result in the best performance. The resulting partitionings are evaluated during run-time using a SAMR simulator and the partitioning with the best performance is selected. In this section, the design and implementation of the Meta-Partitioner is presented. A more detailed description is found in [21].

The partitioning algorithm selection in the Meta-Partitioner is based on the assumption that geometrically similar grid hierarchies have similar partitioning properties. A partitioning algorithm that generates a high-quality partitioning for a given grid hierarchy, probably also does so for a geometrically similar grid hierarchy. As a consequence of this assumption, historic performance data can be used to select the partitioning algorithm. If the geometric characteristics of the current grid hierarchy are matched with previously encountered hierarchies, the performance of a candidate partitioning algorithm can be predicted by its historical performance data for similar grid hierarchies.

The basis for the algorithm selection process is a large performance data base. The data base contains performance data for 768 hybrid partitioning algorithms from Nature+Fable, where each algorithm partitioned almost 1300 different grid hierarchies [23].

3.1 Implementation

For the implementation of the Meta-Partitioner, component-based software engineering (CBSE) is used. The use of CBSE simplifies modifications and expansions to the functionality of the Meta-Partitioner at the same time as it facilitates the integration of SAMR frameworks and other external components (e.g. performance monitoring tools and data base connections).

The Common Component Architecture (CCA) is used for the actual implementation [8, 28]. Developed as a community-driven CBSE initiative, CCA specifically targets the needs of high performance computing. A general, low-latency model for component inter-operability and interaction forms the basis of CCA. In CCA, a component is the basic unit of software functionality. The components interact through abstract interfaces called ports that provide access to the functionality of a component. A component can provide a port, meaning that it implements the functionality defined by the port. A component can also use ports, by performing function calls through the port to access the functionality provided by another component. During run-time, a framework manages and assembles the components and ports into an application. The framework is also responsible for the execution of the application.

A draft for a interoperability standard for SAMR frameworks has been developed by the CCA community and tested with a previous version of the Chombo framework [2]. Also, the SAMR framework GrACE is routinely used in CCA applications that are developed at the Computational Facility for Reacting Flow Science at Sandia National Laboratories [28].

The functionality of the Meta-Partitioner is divided into a number of components (see Figure 1). Because the overall design has been described in detail elsewhere [21], only parts directly involved in the algorithm selection are discussed in this paper.

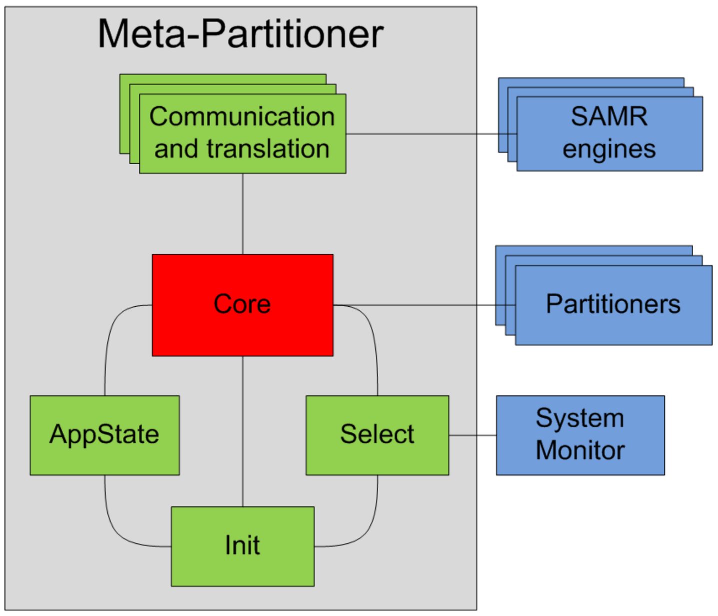


Figure 1: A Meta-Partitioner and its components. The components on the right are adapted from third-party software.

3.1.1 Characterization and matching of grid hierarchies

As previously mentioned, it is assumed that geometrically similar grid hierarchies have similar partitioning properties. When the current grid hierarchy is matched with stored grid hierarchies, historical performance data for a similar stored grid hierarchy can predict the performance for the current grid hierarchy. For the matching, it is paramount that the hierarchies are accurately characterized. For this purpose, a number of geometrical properties for the grid hierarchy are computed in the `AppState`-component (see Table 2). To make the matching more accurate, all metrics are normalized to a common interval using logistic normalization [36]. Logistic normalization employs a two-step approach. First, the metric is normalized using standard z-score normalization [17]. A value v of a metric A is normalized to v_{zero} by computing

$$v_{zero} = \frac{v - \tilde{A}}{\sigma_A}$$

where $\tilde{A}$ and σ_A are the mean and standard deviation of metric A. Next, v_{zero} is normalized to $v_{logistic}$ by computing

$$v_{logistic} = \frac{1}{1 + e^{-v_{zero}}}$$

.

After the logistic normalization, each metric is restricted to the interval $[0, 1]$ and the relative difference between values close to the mean (i.e. 0.5 after the normalization) are increased compared to z-score normalization. The initial z-score normalization is also less sensitive to outliers than many other normalization methods [17].

Metric	**Description**
`Levels`	Number of refinement levels
`Area`	Fraction of refined area compared to the base grid
`AreaLower`	Fraction of refined area compared to the next lower level
`PatchSize`	Average size of the grid patches compared to the area of the refinement level
`PatchNum`	Number of patches per area unit on the base grid
`AspectRatio`	Average aspect ratio
`StdDevSize`	Standard deviation of the PatchSize

Table 2: The metrics used by the `AppState`-component to characterize the geometrical properties of a grid hierarchy.

To match the current grid hierarchy with the stored grid hierarchies, the weighted least square method is used. Because the geometrical metrics that describe the grid hierarchy probably differ in importance, each metric is assigned an experimentally determined weight. For these experiments, the Meta-partitioner selected a partitioning algorithm for almost 1300 grid hierarchies using 2187 different weight combinations (for a total of about 2.8 million selected algorithms). Because the experiments only involved algorithms and hierarchies already present in the data base, the average performance of each weight combination could be determined without the need to actually partition the grid hierarchies. The weight combination that resulted in the best performance was selected and used in the Meta-Partitioner. Note that different selection rules (see the next section) use different weights.

3.1.2 Construction of selection rules

In the presented version of the Meta-Partitioner, a static partitioning focus is selected to concentrate the partitioning effort to the performance-inhibiting factors that generally have the largest impact on the execution time —- either the load imbalance or the synchronization delays (see Table 1). Each focus, i.e `FocusLB` and `FocusSynch`, is associated with a maximum allowed performance deviation for its main performance-inhibiting factor.

To select the partitioning algorithm, each algorithm's performance for the most similar stored grid hierarchy is compared to the best historic performance for that particular grid hierarchy. All algorithms that resulted in an equal to or better performance than maximum allowed performance deviation are selected as candidate algorithms. Next, the candidate algorithms are ordered on the basis of their historic performance for the secondary performance-inhibiting factor (i.e. synchronization for `FocusLB` and load imbalance for `FocusSynch`). The algorithm that resulted in the best performance for the secondary performance inhibiting factor is selected. Using this strategy, the partitioning effort will be concentrated to the most performance-inhibiting factor while the impact of the secondary factor is kept as low as possible. Pseudocode for the selection process is presented in Algorithm 1. An allowed performance deviation of zero will result in the selection of the algorithm with the best predicted performance for the most performance-inhibiting factor, completely disregarding the secondary performance-inhibiting factor.

Because the selected algorithm is uniquely determined by the combination of the partitioning focus and the most similar grid hierarchy, it is possible to pre-compute the algorithm selection. For each partitioning focus, the selected algorithms and the corresponding grid hierarchies are stored as rules. During run-time the algorithm selection is reduced to finding the entry in the rule that corresponds to the most similar stored grid hierarchy.

Algorithm 1 Selection of partitioning algorithm

```
FocusSynch(deviation)
1 SELECT partAlg AS candidates FROM mostSimilarAppState WHERE synch < deviation*$\mathbf{MIN}_{all}$(synch)
2 SELECT partAlg FROM candidates WHERE LB = $\mathbf{MIN}_{cand}$(LB)

FocusLB(deviation)
1 SELECT partAlg AS candidates FROM mostSimilarAppState WHERE LB < deviation*$\mathbf{MIN}_{all}$(LB)
2 SELECT partAlg FROM candidates WHERE synch = $\mathbf{MIN}_{cand}$(synch)
```

Please note the differences in the **MIN**-clauses. For step 1, **MIN** corresponds to the minimum for all partitioning algorithms. For step 2, **MIN** corresponds to the minimum for the algorithms selected during step 1. In this paper, `deviation`=1.25 for `FocusSynch` and `deviation`=1.2 for `FocusLB`

3.1.3 Run-time evaluation of multiple partitionings

During the evaluation of the initial version of the Meta-Partitioner, it was discovered that the least square matching was sensitive to perturbations. The differences in the least square sums for the most similar grid hierarchies were generally small. Even a tiny perturbation in the least square sum would probably change the matching to a new stored grid hierarchy and result in the selection of another partitioning algorithm.

To examine the performance impact of the matching process, a theoretically derived evaluation of the partitioning algorithms corresponding to the ten most similar grid hierarchies was performed [22]. The evaluation showed that the average performance of the ten algorithms was similar to or slightly worse than the performance of the algorithm that was actually selected. However, at least one of the ten partitioning algorithms would often perform significantly better than the selected partitioning algorithm. To find the best performing algorithm during run-time, all ten algorithms need to be invoked and all of the resulting partitionings must be evaluated. Because the algorithms in the original version of the Meta-Partitioner were selected on the basis of their predicted performance rather than the actual performance of the resulting partitionings, this discovery could not be used to improve the performance.

To evaluate the quality of a partitioning, a SAMR simulator has been used [10]. Rather than simulating a parallel computer, the simulator mimics the execution of the Berger-Colella SAMR algorithm [6]. At each re-partitioning, the simulator computes metrics like the arithmetical load imbalance, communication volumes, data migration, and synchronization penalty. Due to its long execution time, it is intractable to use the simulator to evaluate partitionings during run-time.

An analysis of the simulator execution time showed that the largest part was used to compute the communication volumes. By restricting the computations to the load imbalance and the synchronization penalty, the execution time could be reduced with approximately a factor of 500.

The modified simulator was expanded into a CCA component and connected to the Meta-Partitioner. Before the start of a simulation, the user can choose the number of partitioning algorithms that will be selected invoked at each re-partitioning.

Name	**Description**	**#Algorithms**
`MP1`	Most similar grid hierarchy	1
`MP1+Static`	Most similar grid hierarchy and the best static algorithm	2
`MP10`	10 most similar grid hierarchies	10
`MP10+Static`	10 most similar grid hierarchies and best static algorithm	11
`MP+Apps`	10 most similar grid hierarchies and hierarchies from all applications	10-12
`MP10+All`	10 most similar grid hierarchies, best static algorithm, and hierarchies from all applications	11-13

Table 3: The selection approaches used for the matching of the grid hierarchies. The `Apps` and `All` configurations make sure that at least one hierarchy from each of the applications in the performance data base is used, even if the similarity is low.

In the new version of the Meta-Partitioner, it is possible to use several approaches to select the corresponding partitioning algorithms. The basic option is to select the partitioning algorithms that correspond to any number of the most similar grid hierarchies. This option is called `MP`$\{$`X`$\}$, where X is the number of selected algorithms ($X = 1$ corresponds to the original implementation). A second option is to use the performance data base before the start of the execution to determine the partitioning algorithm that has the best average performance. This algorithm is always invoked together with any other algorithm(s). This option is assigned the label `static`. If the geometrical properties of the current grid hierarchies are highly unusual, the partitioning algorithms that correspond to the most similar grid hierarchies might result in bad performance. The performance might benefit if the matched grid hierarchies also include at least one hierarchy from each application present in the performance data base — even if the geometrical similarity is low. This approach uses the label Apps. Finally, different combinations of the approaches can also be used. The different approaches used in this paper are listed in Table 3.

Algorithm 2 Selection of partitioning algorithm, extended version

```
1. FocusSynch(deviation)
for each candidate algorithm do
  if (currentSynch < 0.75*bestSynch) AND (currentLB < 2*bestLB) then
    bestAlg = currentAlg; bestSynch = currentSynch; bestLB = currentLB
  else
    if (currentSynch < bestSynch) AND (currentLB < 1.5*bestLB) then
      bestAlg = currentAlg; bestSynch = currentSynch; bestLB = currentLB
    end if
  else
    if (currentSynch < 1.05*bestSynch) AND (currentLB < 0.75*bestLB) then
      bestAlg = currentAlg; bestSynch = currentSynch; bestLB = currentLB
    end if
  end if
end for

2. FocusLB(deviation)
for each candidate algorithm do
  if (currentLB < 0.75*bestLB) AND (currentSynch < 1.75*bestSynch) then
    bestAlg = currentAlg; bestSynch = currentSynch; bestLB = currentLB
  else
    if (currentLB < bestLB) AND (currentSynch < 1.2 * bestSynch) then
      bestAlg = currentAlg; bestSynch = currentSynch; bestLB = currentLB
    end if
  else
    if (currentLB < 1.05*bestLB) AND (currentSynch < 0.75*bestSynch) then
      bestAlg = currentAlg; bestSynch = currentSynch; bestLB = currentLB
    end if
  end if
end for
```

In this paper, deviation=1.25 for FocusSynch and deviation=1.2 for FocusLB

When multiple partitionings are constructed, each partitioning is evaluated during run-time by the new version of the SAMR simulator. Instead of selecting the partitioning that results in the best performance for the primary performance-inhibiting factor, more advanced selection criteria are used. For example, even if the partitioning focus is set to `FocusLB`, a marginally increased load imbalance and a substantially decreased synchronization penalty would probably result in a shorter execution time. Pseudocode for the modified selection criteria are presented in Algorithm 2. Note that the different rules use different multiples in the selection criteria due to the distribution of the performance data. The extended version replaces step 2 in the original algorithm (see Algorithm 1).

4 Experimental setup

For the evaluation of the Meta-Partitioner, four real-world applications from the Virtual Test Facility (VTF) are used. The VTF, developed at the California Institute of Technology, is a software environment for coupling solvers for compressible computational fluid dynamics (CFD) with solvers for computational solid dynamics (CSD) [14, 46]. The purpose of the VTF is to simulate highly coupled fluid-structure interaction problems. The used applications are restricted to the CFD domain of the VTF, because the CSD solver is implemented with unstructured grids and the finite element method.

Ramp simulates the reflection of a planar Mach 10 shock wave striking a 30 degree wedge. A complicated shock reflection occurs when the shock wave hits the sloping wall. The initial grid size is 480x120 grid points and the application uses three levels of refinement with refinement factors $\{2,2,4\}$. *ShockTurb* treats the interaction of two contacting gases with different densities that are subject to a shock wave. When hit by the shockwave, a Richtmyer-Meshkov instability is created. The initial grid size is 240x120 grid points and and the application uses three levels of refinement with a constant refinement factor of two. *ConvShock* simulates a Richtmyer-Meshkov instability in a spherical setting. The gaseous interface is spherical and sinusoidal in shape. The interface is disturbed by a Mach 5 spherical and converging shock wave. The initial grid size is 200x200 grid points and the application uses four levels of refinement with refinement factors $\{2,2,4,2\}$. In the *Spheres* application, a constant Mach 10 flow passes over two spheres placed inside the computational domain. The flow results in steady bow shocks over the spheres. The initial grid size is 200x160 grid points and the application uses three levels of refinement with a constant refinement factor of two.

4.1 Methodology

To evaluate the Meta-Partitioner, un-partitioned trace files for the four described applications are used [47]. The trace files contain the complete grid hierarchies from real-world executions of the applications (coordinates, sizes, refinement factors etc.). The trace files are used as input to the Meta-Partitioner, one re-partitioning at a time. The Meta-Partitioner matches the current grid hierarchy with the stored grid hierarchies in the data base. Next, either a single or multiple candidate partitioning algorithms are selected and invoked. When multiple algorithms are invoked, all resulting partitionings are evaluated during run-time using the new version of the SAMR simulator. The partitioned grid hierarchy with the best performance is stored on disk and it is later thoroughly evaluated using the original version of the simulator [40].

For the matching of the grid hierarchies, the experiments are performed using the different configurations shown in Table 3. When the performance of multiple partitionings is evaluated, the modified selection strategies described in Algorithm 2 are employed. Generally, the grid hierarchies are partitioned for 16 processors. Scalability results, where 32 processor partitionings are constructed using rules derived from performance data for 16 processor configurations, are also computed.

Only the results for the two most performance-inhibiting factors — load imbalance and synchronization — are presented. The load imbalance is defined as

$$\text{Load imbalance } (\%) = 100 * \frac{\text{Max}\{\text{processor workload}\}}{\text{Average workload}} - 100.$$

The synchronization penalty is computed as follows [42]. For each refinement level, the processors check their neighbors for the need to wait for any of them. If a processor needs to wait, the penalty is approximated by the number grid points that have to be updated by other processors before the stalled processor can resume its computations. The severity of the penalty is affected by how much work the stalled processor has left on higher refinement levels — stalling a processor with a great amount of work left is more serious than holding up a processor with little remaining work. Hence, the penalty is multiplied by the processor's remaining workload.

The performance data are compared to both the average value for all partitioning algorithms and to the best static partitioning algorithm. The best static partitioning algorithm was determined by computing the average performance for each of the 768 hybrid partitioning algorithms present in the performance data base. Disregarding the performance data for the current application, the best static partitioning algorithms were selected using the same criteria that were used to construct the selection rules (see Section 3.1.2). Note that the best static algorithm is dependent on the current partitioning focus — different partitioning focuses generally result in different partitioning algorithms.

Finally, the average execution times for the different matching configurations are also presented.

5 Results

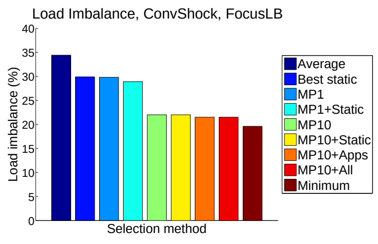
The performance results are presented separately for each partitioning focus. The results for the scalability experiments and the executions times are found at the end of the section.

The selection rules have been computed to eliminate any performance data dependencies for the current application. Hence, separate rules for each combination of application and partitioning focus has been used for the evaluation.

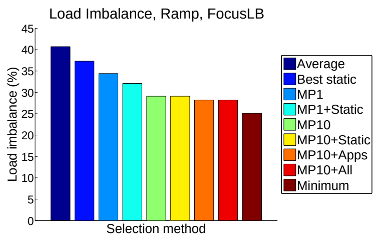
The data base includes performance data for all possible combinations of partitioning algorithm and grid hierarchy that can be encountered during the evaluation. Hence, a minimum value can be computed for each metric and application. Given the candidate partitioning algorithms and the applications used for the evaluation, it is impossible to construct partitionings with a better performance. Of course, if new partitioning algorithms are added to the Meta-Partitioner, this minimum value will no longer be valid.

For the performance analysis, the results are compared to both the average for all partitioning algorithms and to the results of the best static partitioning algorithm. If a user has access to a performance data base, he can use the data base compute the best static partitioning algorithm. Hence, emphasis should be given to the results for best static algorithm. If a user lacks access to a performance data base, it is impossible to determine the best static algorithm. Here, priority should instead be given to the average results for all algorithms.

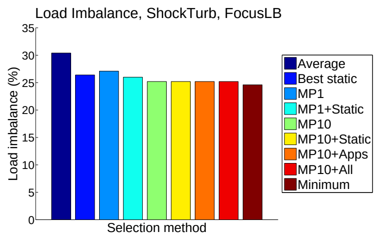
FocusLB



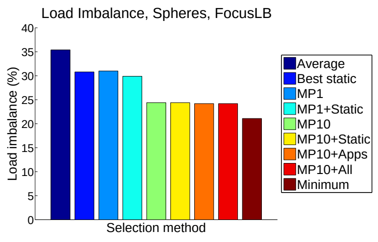
(a) ConvShock2D



(b) Ramp2D



(c) ShockTurb2D

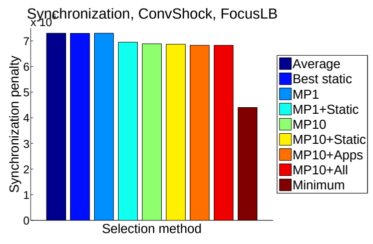


(d) Spheres2D

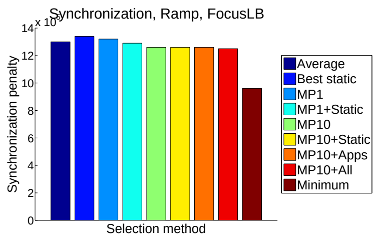
Figure 2: The load imbalance for the different configurations of the Meta-Partitioner and `FocusLB`. Note that the `MP10`-configurations result in a near optimal performance.

For all applications, `MP1`-configuration results in a smaller load imbalance than the average imbalance for all partitioning algorithms (see Figure 2). The decrease is on average 13.1%. `MP1` has an 8% smaller load imbalance than the best static partitioning algorithm for the Ramp application, while the difference between `MP1` och the best static algorithm is insignificant for the three other applications.

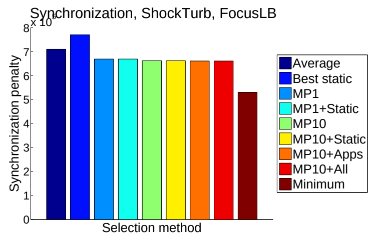
When multiple algorithms are selected, the performance of the resulting partitionings are evaluated during run-time. Hence, the performance will always be equal to or better than the performance for `MP1`. Using `MP1+Static`, the load imbalance is on average decreased by 4.8% compared to `MP1` and 5.4% compared to the best static algorithm.



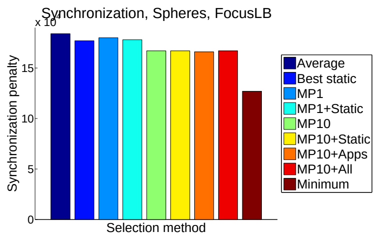
(a) ConvShock2D



(b) Ramp2D



(c) ShockTurb2D



(d) Spheres2D

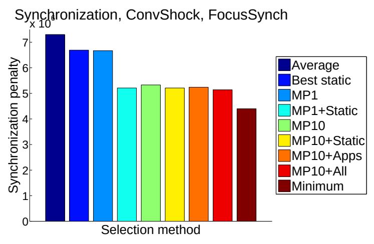
Figure 3: The synchronization penalty for the different configurations of the Meta-Partitioner and `FocusLB`. The synchronization penalty is not negatively affected by `FocusSynch`.

The load imbalance is substantially reduced when the `MP10`-configuration is used. For the three applications with the largest improvements, the load imbalance is on average decreased by 21% compared to the `MP1`-configuration. Because the load imbalance for `MP1` is already small for ShockTurb, the imbalance is decreased by only 7%. A comparison of the load imbalance for `MP10` against the minimum values shows that the load imbalance is close to the minimum for all applications. On average, the load imbalance is increased by only 11.5% from the minimum value when `MP10` is used. For the ShockTurb application, the increase is as low as 2.4%. A closer examination of the individual re-partitionings shows that `MP10` often selected the best performing partitioning algorithm among all of the 768 partitioning algorithms!

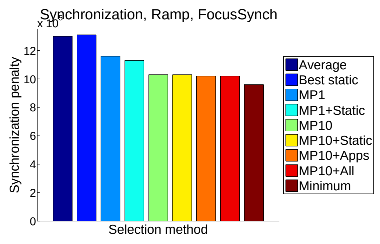
The more advanced configurations that are based on `MP10` — `MP10+Static`, `MP10+Apps`, and `MP10+All` — all result in further, but much smaller, reductions of the load imbalance. Due to the longer execution times of these configurations (see Table 4), they will probably not decrease the total execution time for the SAMR application.

The secondary performance-inhibiting factor, the synchronization penalty, is generally slightly smaller than the average penalty for all partitioning algorithms (see Figure 3). When multiple partitionings were evaluated, the new selection criteria (see Section 3.1.3 and Algorithm 2) decreased the average synchronization penalty by 4.6%. The criteria allow the selection of a partitioning with a small synchronization penalty even if the partitioning has a slightly higher load imbalance than other partitionings.

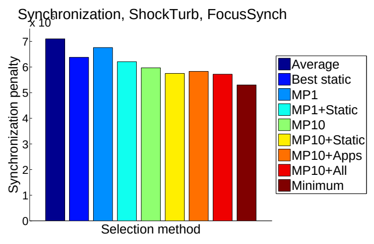
FocusSynch



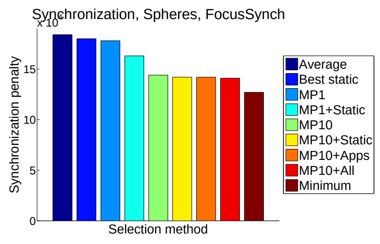
(a) ConvShock2D



(b) Ramp2D



(c) ShockTurb2D



(d) Spheres2D

Figure 4: The synchronization penalty for the different configurations of the Meta-Partitioner and `FocusSynch`. Note that the `MP10`-configurations result in a near optimal performance.

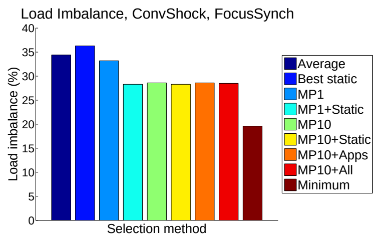
The original version of the Meta-Partitioner, `MP1`, always results in a smaller synchronization penalty than the average penalty for all partitioning algorithms (see Figure 4). On average, the decrease is 7.2%.

Compared to the best static partitioning algorithm, the synchronization penalty is approximately equal for the ConvShock and the Spheres applications. For the Ramp application, the penalty is decreased by 11.4% while it is increased by 5.9% for the ShockTurb application.

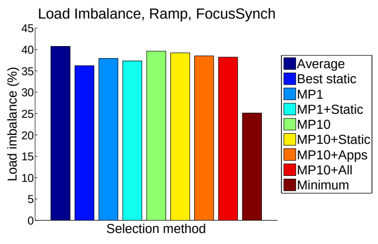
For the `MP10`-configuration, the synchronization penalty is significantly decreased for three of the four applications, on average with 16.8%. The fourth application, ShockTurb, results in a smaller decrease of 11.4% since the room for improvements are smaller. The resulting synchronization penalty is close to the minimum penalty for all applications. On average, the penalty is increased by only 13.6% from its minimum value for the 768 partitioning algorithms.

The penalty is further decreased when the more elaborate configurations based on `MP10` are used. The reductions are probably too small to result in a shorter total execution time because of the longer partitioning times.

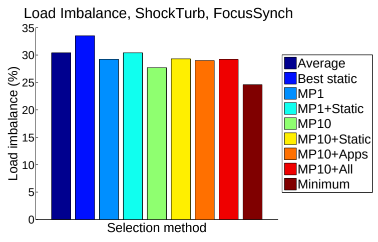
The load balance for the four applications is always equal to or smaller than the average imbalance for all algorithms (see Figure 5). The behavior observed for `FocusLB`, where the more advanced configurations based on `MP10` resulted in better performance for the secondary performance-inhibiting factor, is only repeated for the ConvShock



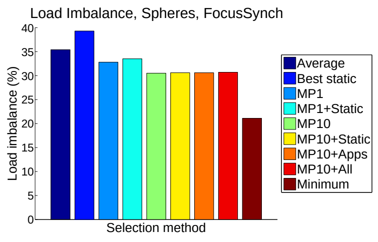
(a) ConvShock2D



(b) Ramp2D



(c) ShockTurb2D



(d) Spheres2D

Figure 5: The load imbalance for the different configurations of the Meta-Partitioner and `FocusSynch`. The load imbalance is not negatively affected by `FocusSynch`.

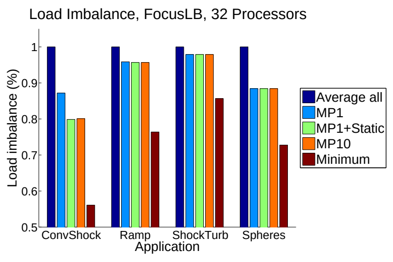
and Spheres applications. The load imbalance for two other applications, Ramp and ShockTurb, varies without a discernible pattern. For these two applications, it is probable that only a few partitionings result in a small load load imbalance, decreasing the impact of the new selection criteria (see Algorithm 2).

Scalability of the rules

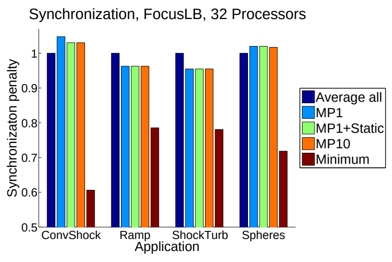
To evaluate the scalability of the selection process, rules optimized for 16 processor configurations are used to construct partitionings for 32 processor. No results for the best static partitioning algorithm are presented. These scalability experiments are only meaningful when performance data for the current processor configuration are unavailable, which makes it impossible to compute the best static algorithm. Results for the configuration `MP1+Static` are presented, but in this case the best static algorithm is derived for 16 processor configurations. All results have been normalized to the average value of the current metric.

When `FocusLB` is used, the load imbalance for all configurations is always smaller than the average load imbalance (see Figure 6a). The improvements are largest for the the ConvShock and the Spheres applications, with a decrease of 12.8% respectively 11.6%. The synchronization penalty is slightly smaller than the average penalty for Ramp and ShockTurb while it is slightly increased for ConvShock and Spheres (see Figure 6b). However, these changes are too small to be significant.

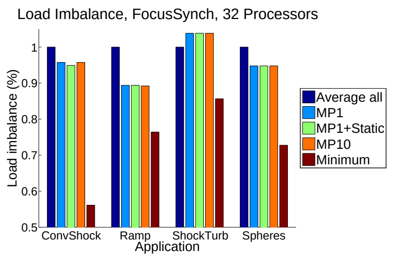
For `FocusSynch`, the synchronization penalty for three of the applications is decreased by 4.2% compared to the average for all partitioning algorithms (see Figure 6d).



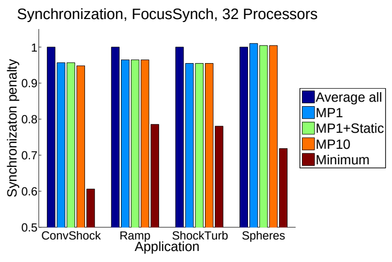
(a) Load imbalance, `FocusLB`



(b) Synchronization penalty, `FocusLB`



(c) Load imbalance, `FocusSynch`



(d) Synchronization penalty, `FocusSynch`

Figure 6: Results for 32 processors using rule originally constructed for 16 processors. The Meta-Partitioner consistently generates partitionings with a better performance than the average partitioning.

For the fourth application, Spheres, a marginal increase of 1% is observed. The results for the load imbalance is surprisingly better than the results for synchronization penalty, even though the partitioning effort is concentrated to the synchronization (see Figure 6c). For the Ramp application, the resulting load imbalance is even smaller than the corresponding load imbalance for `FocusLB`.

The more configurations that generate multiple partitionings (`MP1+Static` and `MP10`) do not generally improve the partitioning quality, regardless of partitioning focus. It is only for `FocusLB` and the ConvShock application that `MP1+Static` and `MP10` are able to produce significantly better partitionings than `MP1`. For all other cases, the performance differences between `MP1` and the more advanced configurations (`MP1+Static` and `MP10`) are insignificant.

Execution times

Both the selection of the partitioning algorithms and the construction and evaluation of multiple partitionings will add an overhead to the total execution time. To decrease the total execution time, the overhead must be smaller than the resulting reduction in simulation time for the SAMR framework. In this section, execution times for different configurations of the Meta-Partitioner are presented. All experiments where performed on a computer having a 1.73 GHz Intel T2250 Core Duo processor and 2GB of DDR2 memory. For all configurations, the execution times are computed from experiments

	ConvShock			Ramp		
	Total	MP	Part	Total	MP	Part
MP1	19.8	4.1	15.7	9.2	2.9	6.3
MP1+Static	35.2	5.6	29.6	19.4	4.9	14.1
MP10	185	28.5	156.5	95.9	21	74.9
MP10+Static	203.4	30.7	172.7	106.2	23.2	82.9
MP10+Apps	218.5	33.2	185.3	108.1	21	87.1
MP10+All	233.4	34.5	198.9	118.7	26.6	92.1

	ShockTurb			Spheres		
	Total	MP	Part	Total	MP	Part
MP1	4.5	2.2	2.3	20.6	3.9	16.7
MP1+Static	7.5	2.7	4.8	39	5.6	33.3
MP10	32.6	8.7	23.9	188.2	22.2	166
MP10+Static	35.5	9.4	26.1	206.8	24.3	182.5
MP10+Apps	39.6	11	28.6	221.2	26.8	194.4
MP10+All	42.7	11.7	31	239.6	28.9	210.7

Table 4: The execution time for the complete partitioning process (Total) is the sum of the time needed by the Meta-Partitioner to select and evaluate appropriate partitioning algorithms (MP) and the time needed to partition the grid hierarchy (Part).

performed with both partitioning focuses.

The execution time for the complete partitioning process is the sum of the time needed by the Meta-Partitioner to select and evaluate appropriate (MP) partitioning algorithms and the time needed to partition the grid hierarchy (Part).

For `MP1`, the MP-part is substantially faster than the partitioning time for three of the four application. The last application, ShockTurb, results in approximately equal execution times for the MP-part and the partitioner. The execution times between the individual applications differ because of the properties of the grid hierarchy, e.g. ConvShock and Spheres have a larger number of grid patches than the two other applications.

For `MP1+Static`, the partitioning time is approximately doubled because two partitionings are constructed at each re-partitioning. The increase in execution time for the MP-part is smaller, on average 43%.

The partitioning time for the more advanced configurations is proportional to the number of constructed partitionings because the partitioning time for a single partitioning remains approximately constant. The partitioning time varies for the `MP10+Apps` configuration, especially for the Ramp application. For `MP10+Apps`, the number of constructed partitionings can change between re-partitionings because partitioning algorithms corresponding to all applications in the data base must be selected. The MP-part grows at a slower and almost linear rate.

In the extended version of the Meta-Partitioner, all partitionings are constructed and evaluated by a single processor. However, this task is embarrassingly parallel. Hence, the partitionings can easily be both constructed and evaluated in parallel to significantly decrease the overall execution time of the Meta-Partitioner.

6 Conclusion

In this paper, a performance evaluation of an extended and improved version of the Meta-Partitioner was presented. The Meta-Partitioner is a partitioning framework that autonomously selects, configures, invokes, and evaluates partitioning algorithms for SAMR grid hierarchies during run-time. The implementation uses component-based software engineering and it is not restricted to any SAMR framework or grid format. To consistently construct high-quality partitionings, the partitioning effort is concentrated to the most performance-inhibiting factor — either the load imbalance (`FocusLB`) or the synchronization (`FocusSynch`). The performance of the candidate partitioning algorithms are predicted using historical performance data. Before execution, the user specifies the number of alternative partitionings to construct at each re-partitioning. If multiple partitionings are constructed, the performance of each partitioning is evaluated during run-time and the partitioning with the best performance is selected.

In its base configuration, `MP1`, the Meta-Partitioner selects a single partitioning algorithm. For the most performance-inhibiting factor, the `MP1` configuration consistently generates partitionings that result in better performance than the average partitioning. Compared to the best static partitioning algorithm, the results are approximately equal. For the secondary performance-inhibiting factor, the results are similar to the average performance.

The `MP10`-configuration constructs and evaluates ten partitionings. For this configuration, huge improvements for the most performance-inhibiting factor occur. The resulting performance is always close to the minimum value for each application. On average, the minimum value for the most performance-inhibiting factor is increased by only 11.5% for `FocusLB` and by 13.6% for `FocusSynch`. The more advanced methods based on the `MP10`-configuration does not result in a significantly better performance than `MP10`. The `MP10`-configuration does not result in a bad performance for the secondary performance-inhibiting factor. The only drawback of the `MP10`-configuration is the long execution time needed to construct all ten partitionings. However, the partitioning process is inherently parallel and the Meta-Partitioner can easily be modified to invoke multiple partitioners in parallel. Furthermore, configurations that construct fewer partitionings, like `MP5`, might also result in good performance.

Experiments where selection rules optimized for 16 processors were used for 32 processor configurations were also performed. The performance for both performance-inhibiting factors were generally better than the average performance for all partitioning algorithms. Unfortunately, the huge improvements for the `MP10`-configuration were not repeated. Thus, while the Meta-Partitioner still produced high-quality partitionings, the partitionings can not be expected to result in a near optimal performance.

Several tasks remain before the Meta-Partitioner can be deployed in a production environment. The Meta-Partitioner is not yet interfaced to any SAMR framework.

This task should be given a high priority. The Meta-Partitioner can also be improved in several areas. The real-world impact of performance-inhibiting factors has to be determined and the selection rules should be adjusted accordingly. Currently, the partitioning focus remains static during the execution. To change the focus during run-time, methods that measure both the partitioning needs of the application and state of the computer should be implemented [21, 44, 45]. In a longer time frame, it is desirable to add the capability to perform hierarchical partitioning for heterogeneous computational environments.

The extended version of the Meta-Partitioner significantly decreases the impact of the most performance-inhibiting factors for parallel SAMR applications. The performance of the generated partitionings are consistently close to the optimal performance for each applications in the evaluation. When interfaced with a SAMR framework, the Meta-Partitioner has the potential to significant reduce the execution time of parallel SAMR applications.

7 Acknowledgments

The author thanks Ralf Deiterding, Oak Ridge National Laboratory, for providing the application trace files and the performance data for the domain-based partitioner. The author also thanks Johan Steensland and Jaideep Ray, Sandia National Laboratories, for scientific collaboration.

References

[1] ALC linux cluster. http://www.llnl.gov/linux/alc/, Oct. 2008.

[2] Benjamin A. Allan and Jaideep Ray. The scalability impact of a component-based software engineering framework on a growing SAMR toolkit: a case study. In *Parallel Computational Fluid Dynamics*, 2005.

[3] AMROC - Blockstructured adaptive mesh refinement in object-oriented C++. http://amroc.sourceforge.net/index.htm, November 2008.

[4] Dinshaw Balsara and Charles Norton. Highly parallel structured adaptive mesh refinement using language-based approaches. *Journal of Parallel Computing*, (27):37–70, 2001.

[5] Marsha J. Berger and Shahid H. Bokhari. A partitioning strategy for nonuniform problems on multiprocessors. *IEEE Transactions on Computers*, 36(5):570–580, 1987.

[6] Marsha J. Berger and Philip Colella. Local adaptive mesh refinement for shock hydrodynamics. *Journal of Computational Physics*, 82:64–84, May 1989.

[7] Marsha J. Berger and Joseph Oliger. Adaptive mesh refinement for hyperbolic partial differential equations. *Journal of Computational Physics*, 53:484–512, Mar 1984.

[8] David E. Bernholdt et al. A Component Architecture for High-Performance Scientific Computing. *International Journal of High Performance Computing Applications*, 20(2):163–202, 2006.

[9] Greg L. Bryan. Fluids in the universe: Adaptive mesh refinement in cosmology. *Computing in Science and Engineering*, pages 46–53, Mar-Apr 1999.

[10] Sumir Chandra, Mausumi Shee, and Manish Parashar. A simulation framework for evaluating the runtime characteristics of structured adaptive mesh refinement applications. Technical Report TR-275, Center for Advanced Information Processing, Rutgers University, 2004.

[11] Chombo - Infrastructure for adaptive mesh refinement. http://seesar.lbl.gov/ANAG/chombo/, Nov. 2008.

[12] Phillip Colella, John Bell, Noel Keen, Terry Ligocki, Michael Lijewski, and Brian van Straalen. Performance and scaling of locally-structured grid methods for partial differential equations. In *SciDAC*, 2007.

[13] Ralf Deiterding. Detonation simulation with the AMROC framework. In *Forschung und wissenschaftliches Rechnen: Beitrge zum Heinz-Billing-Preis 2003*, pages 63–77. Gesellschaft fr Wiss. Datenverarbeitung, 2004.

[14] Ralf Deiterding, Raul Radovitzky, Sean P. Mauch, Ludovic Noels, Julian C. Cummings, and Daniel I. Meiron. A virtual test facility for the efficient simulation of solid material response under strong shock and detonation wave loading. *Engineering with Computers*, 22(3):325–347, 2006.

[15] Enzo homepage. http://lca.ucsd.edu/portal/software/enzo, January 2009.

[16] M. R. Garey, D. S. Johnson, and L. Stockmeyer. Some simplified NP-complete problems. In *STOC '74: Proceedings of the sixth annual ACM symposium on Theory of computing*, pages 47–63, 1974.

[17] Jiawei Han and Micheline Kamber. *Data Mining: Concepts and Techniques (The Morgan Kaufmann Series in Data Management Systems)*. Morgan Kaufmann, 2000.

[18] S. Hawley and M. Choptuic M. Boson stars driven to the brink of black hole formation. *Physical Review*, D 62:104024, 2000.

[19] William D. Henshaw and Donald W. Schwendeman. Parallel computation of three-dimensional flows using overlapping grids with adaptive mesh refinement. *Journal of Computational Physics*, 227(16):7469–7502, 2008.

[20] Henrik Johansson. *Performance Characterization and Evaluation of Parallel PDE Solvers*. Licentiate thesis, Department of Information Technology, Uppsala University, November 2006.

[21] Henrik Johansson. Design and implementation of a dynamic and adaptive meta-partitioner for parallel SAMR grid hierarchies. Report 2008-017, Department of Information Technology, Uppsala University, Sweden, 2008. Available at http://www.it.uu.se/research/reports/2008-017/.

[22] Henrik Johansson. Run-time selection of partitioning algorithms for parallel samr applications. Report 2009-005, Department of Information Technology, Uppsala University, Sweden, 2009. Available at http://www.it.uu.se/research/reports/2008-005/.

[23] Henrik Johansson and Johan Steensland. A performance characterization of load balancing algorithms for parallel SAMR applications. Report 2006-047, Department of Information Technology, Uppsala University, Sweden, 2006. Available at http://www.it.uu.se/research/reports/2006-047/.

[24] Henrik Johansson and Abbas Vakili. A patch-based partitioner for parallel SAMR applications. In *Proceedings of the International Conference on Parallel and Distributed Computing and Systems*, 2008.

[25] Samet Y. Kadioglu and Mark Sussman. Adaptive solution techniques for simulating underwater explosions and implosions. *Journal of Computational Physics*, 227(3):2083–2104, 2008.

[26] Zhiling Lan, Valerie E. Taylor, and Greg Bryan. A novel dynamic load balancing scheme for parallel systems. *Journal of Parallel and Distributed Computing*, 62:1763–1781, 2002.

[27] Zhiling Lan, Valerie E. Taylor, and Yawei Li. DistDLB: improving cosmology SAMR simulations on distributed computing systems through hierarchical load balancing. *Journal of Parallel and Distributed Computing*, 66(5):716–731, 2006.

[28] Sophia Lefantzi, Jaideep Ray, Christopher A. Kennedy, and Habib Najm. A component-based toolkit for simulating reacting flows with high order spatial discretizations on structured adaptively refined meshes. *Progress in Computational Fluid Dynamics*, 5:298–315, 2005.

[29] Peter MacNeice, Kevin Olson, Clark Mobarry, Rosalinda de Fainchtein, and Charles Packer. Paramesh: A parallel adaptive mesh refinement community toolkit. *Computer Physics Communications*, 126:330–354, 2000.

[30] Charles L. Mader and Michael L. Gittings. Modeling the 1958 Lituya Bay mega-tsunami, II. *Science of Tsunami Hazards*, 20(5):241–250, 2002.

[31] M. Norman and G. Bryan. Cosmological adaptive mesh refinement. *Numerical Astrophysics*, 1999.

[32] M. L. Norman, G. L. Bryan, R. Harkness, J. Bordner, D. Reynolds, B. O'Shea, and R. Wagner. Simulating Cosmological Evolution with Enzo. *ArXiv e-prints*, May 2007.

[33] Kevin Olson and Peter MacNeice. An overview of the PARAMESH AMR software package and some of its applications. In *Adaptive Mesh Refinement-Theory and Applications, Proceedings of the Chicago Workshop on Adaptive Mesh Refinement Methods*. Springer, 2005.

[34] Manish Parashar and James C. Browne. On partitioning dynamic adaptive grid hierarchies. In *HICSS '96: Proceedings of the 29th Hawaii International Conference on System Sciences (HICSS'96) Volume 1: Software Technology and Architecture*, page 604, 1996.

[35] Manish Parashar and James C. Browne. System engineering for high performance computing software: The HDDA/DAGH infrastructure for implementation of parallel structured adaptive mesh refinement. *IMA Volumes in Mathematics and its Applications*, 177:1–18, 2000.

[36] John C. Platt. Probabilistic outputs for support vector machines and comparisons to regularized likelihood methods. In *Advances in Large Margin Classifiers*, pages 61–74. MIT Press, 1999.

[37] Frans Pretorius and Matthew W. Choptuik. Adaptive mesh refinement for coupled elliptic-hyperbolic systems. *Journal of Computational Physics*, 218(1):246–274, 2006.

[38] Jarmo Rantakokko. *Data Partitioning Methods and Parallel Block-Oriented PDE Solvers*. PhD thesis, Uppsala University, 1998.

[39] Erik Schnetter, Scott H Hawley, and Ian Hawke. Evolutions in 3D numerical relativity using fixed mesh refinement. *Classical and Quantum Gravity*, 21(6):1465–1488, 2004.

[40] Mausumi Shee, Samip Bhavsar, and Manish Parashar. Characterizing the performance of dynamic distribution and load-balancing techniques for adaptive grid hierarchies. In *Proceedings IASTED International conference of parallel and distributed computing and systems*, 1999.

[41] Johan Steensland. *Efficent Partitioning of Dynamic Structured Grid Hierarchies*. PhD thesis, Department of Scientific Computing, Information Technology, Uppsala University, Oct. 2002.

[42] Johan Steensland. Irregular buffer-zone partitioning reducing synchronization cost in samr. In *IPDPS '05: Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05)*, page 257.2, 2005.

[43] Johan Steensland, Sumir Chandra, and Manish Parashar. An application-centric characterization of domain-based SFC partitioners for parallel SAMR. *IEEE Transactions on Parallel and Distributed Systems*, 13(12):1275–1289, Dec 2002.

[44] Johan Steensland and Jaideep Ray. A partitioner-centric model for SAMR partitioning trade-off optimization: Part I. In *Proceedings of the 4th Annual Symposium of the Los Alamos Computer Science Institute (LACSI04)*, 2003.

[45] Johan Steensland and Jaideep Ray. A partitioner-centric model for SAMR partitioning trade-off optimization: Part II. In *2004 International Conference on Parallel Processing Workshops (ICPPW'04)*, pages 231–238, 2004.

[46] The Virtual Test Facility. http://www.cacr.caltech.edu/asc/wiki, Nov. 2008.

[47] Two-dimensional AMROC mesh hierarchies. http://www.cacr.caltech.edu/asc/wiki/bin/view/Amroc/AmrSimulator, December 2008.

[48] Andrew M. Wissink, Richard D. Hornung, Scott R. Kohn, Steve S. Smith, and Noah Elliott. Large scale parallel structured AMR calculations using the SAMRAI framework. In *Supercomputing '01: Proceedings of the 2001 ACM/IEEE conference on Supercomputing (CDROM)*, 2001.