UPPSALA
UNIVERSITET

Digital Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology 618

Autonomic Management of Partitioners for SAMR Grid Hierarchies

HENRIK JOHANSSON

ACTA
UNIVERSITATIS
UPSALIENSIS
UPPSALA
2009

ISSN 1651-6214
ISBN 978-91-554-7455-3
urn:nbn:se:uu:diva-99362

Dissertation presented at Uppsala University to be publicly examined in 2446, Building 2, Polacksbacken, Lägerhyddsvägen 2, Uppsala, Friday, April 24, 2009 at 10:15 for the degree of Doctor of Philosophy. The examination will be conducted in English.

Abstract

Johansson, H. 2009. Autonomic Management of Partitioners for SAMR Grid Hierarchies. Acta Universitatis Upsaliensis. *Digital Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology* 618. 58 pp. Uppsala. ISBN 978-91-554-7455-3.

Parallel structured adaptive mesh refinement methods decrease the execution time and memory usage of partial differential equation solvers by adaptively assigning computational resources to regions with large solution errors. These methods result in a dynamic grid hierarchy. To get good parallel performance, the grid hierarchy is frequently re-partitioned and distributed over the processors. Optimally, the partitioner should minimize all performance-inhibiting factors like load imbalance, communication volumes, synchronization delays, and data migration. No single partitioner performs well for all hierarchies and parallel computers. Because the partitioning conditions change during run-time, dynamically selecting a partitioner is non-trivial.

In this thesis, we present the Meta-Partitioner: a partitioning framework that autonomously selects, configures, invokes, and evaluates partitioning algorithms during run-time. For the implementation, we use component-based software-engineering. We predict the performance of the candidate partitioning algorithms with historical performance data for grid hierarchies similar to the current hierarchy. We focus the partitioning effort on the most performance-inhibiting factors — the load imbalance and the synchronization delays. At re-partitioning, a user-specified number of partitioning algorithms is selected and invoked. The performance of each partitioning is evaluated during run-time and the best one is selected.

The performance of the selected partitioning algorithms was compared both to the average performance of 768 algorithms and the global minimum at each re-partitioning. The results showed huge improvements both for the load imbalance and the synchronization delays. Compared to the average partitioning, the load imbalance was decreased by 28.2%. The synchronization delays were decreased by 21.5%. Compared to the global optimum, the load imbalance was increased by only 11.5%. For the synchronization delays, the increase was 13.6%. Often, the Meta-Partitioner selected the best algorithm among all candidate algorithms.

Keywords: structured adaptive mesh refinement, partitioning, load balancing, SAMR

Henrik Johansson, Department of Information Technology, Division of Scientific Computing, Box 337, Uppsala University, SE-751 05. Uppsala, Sweden

ISSN 1651-6214
ISBN 978-91-554-7455-3
urn:nbn:se:uu:diva-99362 (http://urn.kb.se/resolve?urn=urn:nbn:se:uu:diva-99362)

When was the last time you did something for the first time?
-Fight Club

Choose your road. Ride it well.
-Alastair Humphreys

List of Papers

This thesis is based on the following papers, which are referred to in the text by their Roman numerals.

I Johansson, H., Steensland, J. (2006) A Performance Characterization of Load Balancing Algorithms for Parallel SAMR Applications. *Technical Report 2006-047*, Department of Information Technology, Uppsala University.

II Johansson, H. (2008) Design and Implementation of a Dynamic and Adaptive Meta-Partitioner for Parallel SAMR Grid Hierarchies. *Technical Report 2008-017*, Department of Information Technology, Uppsala University.

III Johansson, H. (2008) A Patch-based Partitioner for Parallel SAMR Applications. In *Proceedings of the 20th IASTED International Conference on Parallel and Distributed Computing and Systems*, ACTA Press[1].

IV Li, L., Norris, B., Johansson, H., Curfman McInnes, L., Ray, J. (2008) Component Infrastructure for Managing Performance Data and Runtime Adaptation of Parallel Applications. In *Proceedings of the 9th International Workshop on State-of-the-Art in Scientific and Parallel Computing.*

V Johansson, H. (2009) Run-time Selection of Partitioning Algorithms for Parallel SAMR Applications. *Technical Report 2009-007*, Department of Information Technology, Uppsala University.

VI Johansson, H. (2009) A Meta-Partitioner for Run-time Selection and Evaluation of Multiple Partitioning Algorithms for SAMR Grid Hierarchies. Submitted to *Journal of Parallel and Distributed Computing*.

Reprints were made with permission from the publishers.

[1]With kind permission of ACTA Press.

Contents

1. Introduction

Human beings have always pursued increased knowledge of the world surrounding them — from the tiny nanoscale to the vast expanses of deep space. Scientific experiments have been the most common source of new knowledge. Unfortunately, many experiments are impossible to perform because the size, time, or cost might be intractable. During the last decades, computer simulations have become a viable and cost-effective alternative to real-world scientific experiments.

As an alternative to real-world scientific experiments, we use partial differential equations (PDEs) to mathematically model the phenomena we are interested in. The solutions to the PDEs can provide the same information as the real-world experiments. Because only a few simple PDEs can be solved analytically, we approximate their solutions with computers. In other words, we have transformed our original experiment to a computer simulation.

To solve the PDEs with a computer, we transform the domain of the original problem to a discrete grid. We apply a numerical solver to the grid and compute an approximation to the solution at each grid point. In this thesis, we use structured grids.

In many simulations, features that require high resolution occupy only a small region of the grid. To increase the efficiency, we adaptively assign computational resources to those regions. We start the simulation with a coarse base grid. During the execution, we identify regions with large solution errors. To increase the accuracy, new grids with higher spatial and temporal resolution are overlaid in these regions. This method, called structured adaptive mesh refinement (SAMR), results in a dynamic and adaptive grid hierarchy where grids are created, moved, and deleted during run-time [8].

To decrease the execution time, we often execute the simulations on parallel computers. The workload is distributed over the processors and each processor computes a part of the solution. Because the grid hierarchy is dynamic, we need to repeatedly re-partition and re-distribute it to maintain good performance. At each re-partitioning, we should partition the grid hierarchy in a way that minimizes the performance-inhibiting factors like load imbalance, synchronization delays, communication volumes, and data migration. No single partitioning algorithm performs well for all grid hierarchies and all parallel computers [62]. Hence, to consistently construct high-quality partitionings, we need to select and invoke the partitioning algorithms adaptively during run-time. This is non-trivial because the conditions for how to partition the hierarchy can change dramatically between re-partitionings.

In this thesis, we present the motivation for the Meta-Partitioner, as well as its design and implementation. The Meta-Partitioner is a partitioning framework for structured adaptive grid hierarchies. At each re-partitioning, the Meta-Partitioner autonomously selects, invokes, and evaluates multiple partitioning algorithms. The Meta-Partitioner uses historical performance data to estimate the impact of the most performance-inhibiting factors for all candidate partitioning algorithms. The algorithms predicted to result in the best performance are selected and invoked. The resulting partitionings are evaluated during run-time and the partitioning with the best performance is transfered to the SAMR framework.

2. Structured Adaptive Mesh Refinement

The solution accuracy for PDE solvers based on finite differences and structured grids can be controlled with the grid resolution. Increased grid resolution generally results in higher accuracy but also in longer execution times. Often, features that require high resolution, like shocks and discontinuities, only occupy a small part of the grid; a uniform high grid resolution is then a waste of computational resources. By adaptively increasing the grid resolution in areas with large solution errors, the execution time and memory usage can be substantially decreased [8, 9].

Structured adaptive mesh refinement (SAMR) starts with a coarse and uniform base grid. At regular intervals, the local solution error is estimated and the grid is recursively refined in areas with large errors. This procedure results in a dynamic and adaptive grid hierarchy and a solution that conforms to the desired accuracy. SAMR is widely used in fields like computational fluid dynamics [8, 22, 36], numerical relativity [54, 60], astrophysics [13, 29, 43], and hydrodynamics [41]. Frameworks for SAMR include Paramesh [40, 48], SAMRAI [71], GrACE [50], AMROC [3,21], Chombo [19,20], Enzo [25,44], and Overture [30].

In the rest of this chapter, we describe SAMR in greater detail. We start with a general description of the grids, we then turn to the integration algorithm and error estimation, before we finish by describing the grid generation. The description generally follows the outline of the common Berger-Colella algorithm [8].

2.1 Grid Description

The basis for SAMR is a uniform and structured grid that covers the entire computational domain. The resolution of this base grid conforms to the lowest acceptable solution accuracy. Every subsequently refined grid must be structured and uniform. All refined grids are separate entities and are not incorporated into the base grid. A grid point can be located inside the domain of several grids, but the grids must reside on separate refinement levels. A refined grid can overlap several coarser grids as long as the refined grid is covered by the union of these coarser grids.

In the early Berger-Oliger SAMR approach, the refined grids could be rotated with respect to the underlying grids [9]. Rotated grids have an advantage when steep gradients or discontinuities are present in the solution. The grids can be aligned with the direction of the phenomena, generally reducing the size of the refined grids. The more common Berger-Colella approach only uses non-rotated grids because of the higher complexity of rotated grids, especially when conservation laws are needed [8].

In the SAMR framework Overture, overlapping grids are used [30]. A grid hierarchy with overlapping grids consists of base grids and refinement grids. The grids still need to be structured but they can also be curvilinear. Grids on the same refinement level are allowed to use different refinement factors. In contrast to the Berger-Colella approach, adjacent grids on the same refinement level must always overlap for the computations of the boundary data. Note that an overlapping grid hierarchy can contain several base grids, each having a different resolution. Overlapping grids are useful for problems with complex geometries and problems with moving or deforming boundaries.

2.2 Integration and Error Estimation

The solution on each grid can be computed individually after determining boundary conditions and initial data. The refined grids will commonly be located in the interior of the computational domain. Because interior grids can not use the boundary conditions supplied with the PDEs, it is necessary to obtain the boundary data from either adjacent grids or grids on the next lower level. The initial data are generally supplied by grids on the next lower level. Hence, all computations on the current refinement level generally must have finished before the execution can proceed to the next level [63].

To increase the solution accuracy on coarser refinement levels, the solution on higher levels is projected down to the lower levels. The update prevents the coarser level solution from becoming dispersed or dissipated. Furthermore, the update stops large solution errors from spreading and contaminating boundary conditions and initial data for refined grids. If the simulation involves flows, a correction step is performed to make sure that the flow over each grid boundary is identical for both grids.

The integration for a single time step is listed in Algorithm 1. The order of integration is illustrated with the following example. Assume two levels of refinement and that the solution is going to be advanced from time t to time $t+k$. The refinement factor in each dimension is two. The integration starts with a step of size k on the base grid (level 0). Boundary data for the refined grids on level 1 are provided by the base grid. The grids on refinement level 1 are integrated once, with a time step of size $k/2$. The level-1 solution is used as initial data for grids on level 2 and two time steps, with step size $k/4$, are performed on level 2. Thus, the integration has reached time $t+k$ on the base grid and time *t+k/2* on refinement level 1 and 2. The solution on level 2 is projected down to level 1, which is advanced to time *t+k*. Using boundary data from level 1, level 2 is advanced to time *t+k* using two steps of size *k/4*. The solution on level 2 is projected down to level 1, and the updated level-1 solution is projected down to the base grid.

Algorithm 1 Advance(l,k)

```
take one step of size $k$ on level $l$
if $l = L$ then
    return
else
    interpolate from level $l$ to $l+1$
    for $i = 1$ to 2 do
        advance($l+1$,$k/2$)
    end for
    project solution from level $l+1$ to $l$
end if
```

The solution error is estimated at regular intervals. Generally, the local truncation error and Richardson extrapolation are used for the estimation [9]. First, a large step of size $2h$ is performed. The solution from this step is compared to the solution from two regular integration steps of size h. Using Richardson extrapolation, the exact form of the truncation error can be unknown. When the solution is not smooth, an error estimated with Richardson extrapolation will be incorrect. However, it is still a useful estimation because the truncation error will probably be large near singularities [9].

An alternative to estimate the solution error is to use features in the solution. In magnetohydrodynamics, the decision on where to refine or coarsen the grid hierarchy has been based on local flows that exceed a threshold [53]. Another approach is to capture and track shocks in the solution and place refined grids over them [46].

2.3 Grid Generation

The process of constructing refined grids in areas with large solution errors is called grid generation. All grid points that have an error larger than a threshold are flagged for refinement. The refined grids must cover all flagged grid points while the grid size is restricted to minimize unnecessary computations. Also, the grid-generating algorithm must be fast because it will be invoked numerous times during a simulation. The amount of refinement — the refinement factor — is usually constant, but a number of SAMR frameworks support different refinement factors for different refinement levels.

A measure of the quality of a refined grid is its efficiency — the ratio of flagged grid points to un-flagged points. Low efficiency results in unnecessary computations, while a too high efficiency often results in a large number of small grids and high communication costs. A too high efficiency can also result in frequent regriddings, as phenomena quickly can move outside the domain of a refined grid. The optimal efficiency of a grid is application dependent.

The most widely used grid generation algorithm is the Berger-Rigoutsos algorithm (BR) [10]. The algorithm is based on signatures, a concept used in edge-detection. The signature for a d-dimensional box is computed by projecting each flagged grid point (or cell) to the d-axis and summarizing the number of flagged cells at each point on the axes. The properties of the second derivative (i.e. zero-crossings) for the signature decide when and where the grid is divided.

Several improvements have been made to the BR algorithm. To decrease fragmentation, grids are only divided if the combined efficiency of the two created grids are improved by a given amount [71]. To minimize the number of flagged grid points in the cutting plane, Rantakokko uses a slightly modified criterion to select the next grid to be divided and where to place the cut [57]. A parallel version of the BR algorithm, that also employs the improvements suggested by Rantakokko, has been developed for the SAMRAI framework [28].

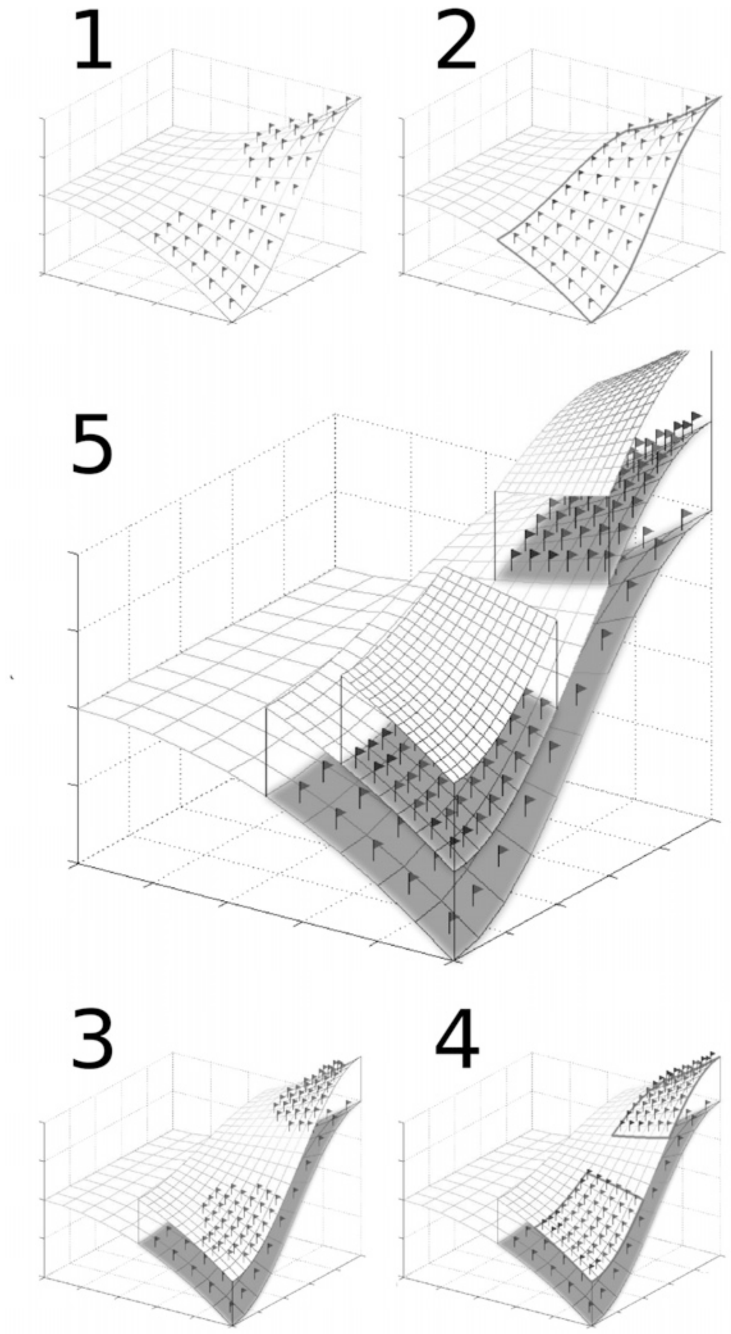


Figure 2.1: The creation of a grid hierarchy. The grids are skewed to illustrate the solution error. In step one and three, the grids are flagged for refinement. The purple boxes in step two and four shows the boundaries for the resulting refined grids. The blue flags represent grid points with small solution errors included in the refined grids to increase their efficiency. The final grid hierarchy is shown in step five.

3. Partitioning of SAMR Grid Hierarchies

The parallel efficiency of a SAMR application depends on the partitioning and distribution of the adaptive dynamic grid hierarchy. Optimally, the partitioning algorithm should construct a partitioning that minimizes the impact of all performance-inhibiting factors like load imbalance, communication, synchronization delays, and data migration. There exist several general approaches for the partitioning of SAMR grid hierarchies. The different approaches generally result in partitionings with vastly different properties and performance.

In this chapter, we first define the performance-inhibiting factors that we use to describe the quality of a partitioning. Next, we discuss the three most common partitioning approaches.

3.1 Performance-Inhibiting Factors

The parallel efficiency of a SAMR application is to a large extent determined by several performance-inhibiting factors. During execution, the impact of these performance-inhibiting factors are dependent on the partitioning algorithm, the grid hierarchy, and the current characteristics of the parallel computer. To analyze the performance, we must define these performance-inhibiting factors. When we evaluate the quality of a partitioning, we present the results for the processor that has the worst performance for the relevant performance-inhibiting factor.

Load Imbalance

The most common performance-inhibiting factor is the arithmetical load imbalance. If a processor is assigned a workload larger than the optimal workload, the execution time will increase. We define the arithmetical load imbalance as:

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

Note that we use the workload for the most overloaded processor. It is the last processor that finishes its computations that determines when the solution can be advanced to the next time step. When we present the results, we generally compute the average load imbalance for all re-partitionings.

Communication

During execution, solution data are exchanged between processors along both the borders of grids and between adjacent refinement levels (see Chapter 2.2). Communications on the same refinement level generally consists of boundary data. When data are exchanged between refinement levels, the data can consist of a combination of boundary data, initial data, and data needed to update the coarser level solution.

We use the number of transfered grid points (both sent and received) for each processor to measure the communication. We divide the communication into two categories — intra-level communication and inter-level communication. When both types of communication are present, the inter-level communication volume is usually larger than the intra-level communication volume. For intra-level communication, only solution data close to the borders of a grid are exchanged while inter-level communication frequently consists of complete grids.

The communication volumes are either presented as an average or on a per time step basis. For the average, the cumulative communication is always determined from the processor having the largest amount of communication after each re-partitioning. Note that the data for the inter-level and intra-level communication can originate from different processors.

Adjacent grids generally have to exchange data, even when they are assigned to the same processor. After each re-partitioning, the SAMR framework needs to determine both the internal data exchanges between grids assigned to the same processor and the external communications between grids assigned to different processors. Depending on the SAMR framework, the set-up time for all these data exchanges can equal the time for the actual exchanges. Unfortunately, it is impossible to measure the set-up time using off-line evaluation tools. However, reducing the number of communications probably results in a reduced set-up time.

Synchronization Delays

When two processors need to communicate data, one of them is often busy computing forcing the other processor to stall. This is called a synchronization delay. These synchronization delays can be of the same magnitude as the computational time (for example, see Table 3.1).

Synchronization delays are related to both the communication and the load imbalance. Because the number of communications affect the number of possible synchronization delays, each communication can result in a delay. The severity of a delay is related to the level-wise load imbalance. Large imbal-

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 3.1: *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 major components of the total time are determination of neighborhood relations and communication patterns. The data origin from sixteen processor executions on the ALC parallel computer at Lawrence Livermore National Laboratory [2]. All data courtesy of Ralf Deiterding, Oak Ridge National Laboratory.*

ances in the individual refinement levels increase the probability of longer delays. Note that a large load imbalance on an individual refinement levels does not necessary imply a large load imbalance for the full grid hierarchy.

We approximate the impact of the synchronization delays with a synchronization penalty [63]. The penalty is computed as follows. The processors check their neighbors on each refinement level 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.

Data Migration

Data migration results when an existing grid is re-assigned to a new processor at re-partitioning. All data associated with the re-assigned grid must be sent to its new owners. Because the data migration generally involves the transfer of complete grids, the amount of migration can be comparable to the total communication volume (i.e. inter-level and intra-level communication).

To decrease the data migration, the partitioner needs to consider the old partitioning when the new partitioning is constructed. However, other performance-inhibiting factors can suffer if a low data migration is pursued too actively. For example, if the old partitioning is kept as intact as possible, the load imbalance is increased when the newly created grids are either too few or too small to be evenly distributed. Furthermore, the communication can also be increased if the new grids are assigned to processors that does not contain any of their neighbors.

Partitioning Time

The last performance-inhibiting factor is the partitioning time — the time needed to construct the partitioning. This performance-inhibiting factor is a trade-off between the time invested in computing the partitioning and the quality of the resulting partitioning. A high-quality partitioning might justify a longer partitioning time.

To decrease the partitioning time, the partitioning can be performed in parallel. Unfortunately, parallel partitioners are still uncommon. Another approach to decrease the partitioning time is to perform the partitioning in a hierarchical fashion. A single processor partitions the grid into a small number of partitionings. Each partitioning is assigned to a group of processors. In each processor group, a single processor partitions the part of the grid assigned to the group. The last step is embarrassingly parallel. The communication patterns between the groups are determined after a partitioning has been computed for each processor group.

3.2 Partitioning Approaches

In this section, we present three common partitioning approaches; domain-based partitioning, patch-based partitioning, and hybrid partitioning.

3.2.1 Domain-Based Partitioning

A domain-based partitioner partitions the computational domain, rather than the individual grids [7, 12, 56, 58]. The domain is partitioned along with all overlaid grids from all refinement levels. Thus, overlaid grids that cover the same part of the coarse base grid are always assigned to the same processor. Generally, the workload of the overlaid grids are projected down to the coarse base grid. For the partitioning, the grid hierarchy is thus conceptually reduced to a large single grid with heterogeneous workload.

The main advantage of domain-based partitioning is the elimination of inter-level communication. Because all overlaid grids are assigned to the same processor, no communication between different refinement levels is necessary. Another consequence is that the parallelism between different levels of refinement can potentially be exploited more efficiently.

For deep or complex grid hierarchies, domain-based partitioning can result in an intractable load imbalance. The resolution of the base grid and the size of the computational stencil impose restrictions on where the base grid can be subdivided. Often, the best place for a subdivision is located between grid

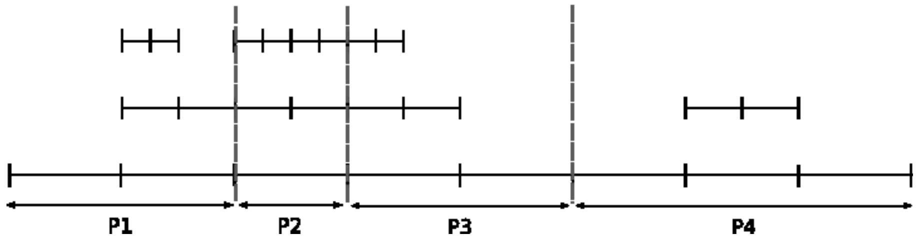


Figure 3.1: A simple domain-based partitioning of a grid hierarchy with two refinement levels and a refinement factor of two. The hierarchy is partitioned for four processors. The optimal workload is six units but P_2 is assigned seven units and P_4 is assigned five units. Also, P_4 is not assigned any work from the highest refinement level. Note that no communications between the refinement levels are necessary because all overlaid regions of the grid are assigned to the same processor.

points on the coarse base grid, forcing a sub-optimal cut. A related problem is "bad cuts" that result in many small blocks with bad aspect ratios. These blocks are created when grids are subdivided in bad locations, assigning only a fraction of a grid to one processor while the majority of the grid resides on another processor.

The domain-based partitioning approach does not generally consider the workload distribution among the refinement levels. Hence, the work assigned to one processor might only contain grids located on lower refinement levels while another processor can be assigned work mostly located on higher refinement levels. Thus, load imbalances can be further amplified by even larger imbalances on the individual refinement levels.

The most common domain-based algorithm is probably recursive coordinate bisection, originally proposed by Berger and Bokhari [7]. Using rectilinear cuts, the algorithm recursively divides the domain into two parts with approximately equal workloads. The direction of the divisions is alternated between the dimensions of the domain, i.e. between horizontal and vertical cuts for a two-dimensional domain. For SAMR applications, the placement of the cuts are restricted by the resolution of the coarse base grid. The algorithm proceeds until the number of parts equals the number of processors. The bisections make the algorithm computationally efficient and result in simple data structures that are easily mapped to a binary tree. A drawback is that the algorithm frequently creates very long and thin grids that potentially result in large amounts of intra-level communication. To create partitionings with better aspect ratios, a later version of the algorithm introduced a parametrization for the trade-off between load balance and communication [12].

Rantakokko combines partitioning techniques from structured and unstructured grids [56, 58]. The domain is initially subdivided into a large number of grids using a domain-based algorithm. For this subdivision, a number of algorithms can be used, including the previously described Berger-Bokhari algorithm [7]. Next, an unstructured graph is constructed. In the graph, the vertices correspond to grids and the edges indicate communication. The unstructured graph is partitioned with a general graph partitioning technique, like recursive spectral bisection [6, 52]. Finally, neighboring grids on the same processor are merged to reduce the overhead. The approach described by Rantakokko generally results in high-quality partitionings but also potentially long partitioning times due to the computationally intensive graph based partitioning techniques.

The domain-based partitioning algorithm in the SAMR framework GrACE follows elegantly from a linear representation of the grid hierarchy [49, 51]. Each grid is stored in a distributed and dynamic list called a Scalable distributed dynamic grid (SDDG). The computational cost for the grid is stored in the SDDG. The SDDGs are dynamically collected to form a Distributed adaptive grid hierarchy (DAGH) that represents the entire adaptive grid hierarchy. The linear representation is achieved by using inverse space-filling curves (SFC) to order the SDDGs. An inverse SFC is a recursive mapping from d-dimensional space to 1-dimensional space i.e., $N^d \to N^1$, such that each point in N^d is mapped to a unique point in N^1 [4]. Furthermore, points close together in N^d will be mapped to points close to each other in N^1.

Initially, the DAGH consists of a single SDDG that represents the coarse base grid. When a grid is refined, the created SDDG is recursively incorporated within the DAGH to replace the SDDG sub-list that corresponds to the refined region. To partition the grid hierarchy, the linear DAGH representation is divided to balance the computational workload between the processor. Due to the recursive nature of the DAGH, overlaid grid grids are always assigned to the same processor. Furthermore, the locality preserving properties of the SFC reduce the amount of intra-level communication because neighboring grids are typically assigned to the same processor.

3.2.2 Patch-Based Partitioning

For patch-based partitioners, the distribution decision is made independently for each grid or refinement level [5, 20, 35, 37, 58, 71]. A grid may be kept on the local processor or moved entirely to another processor. If a grid is large, it can be split into smaller grids.

The main advantage of the patch-based approach is a good load balance because grids can be subdivided and distributed with few restrictions. Also, depending on the implementation, re-partitioning at re-griding can be avoided because created grids can be appended to the partitioning while delete grids can be removed.

The main shortcoming of the patch-based approach is poor data locality because neighboring grids are often distributed over different processors. Bad locality results in large communication volumes and potentially long synchro-

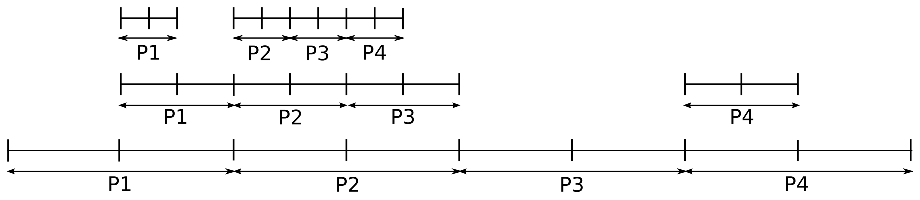


Figure 3.2: A simple patch-based partitioning of a grid hierarchy with two refinement levels and a refinement factor of two. The hierarchy is partitioned for four processors. The workload is perfectly balanced but overlaid parts of the grid are assigned to different processors. For example, P_4 has to communicate the solution on the highest refinement to P_3. After updating the solution on the first refinement level, P_3 transfers the solution to P_2.

nization delays. Also, patch-based partitioners can suffer from an inability to exploit available parallelism across different levels of refinement.

Bin-packing algorithms are often used in patch-based partitioners to distribute the grids. An example is the iterative bin-packing approach used by Balsara and Norton [5]. First, grids much larger than the average grid size are subdivided. Next, the grids are binned to the processors. The bins are then paired against each other and grids are exchanged between the bins if the the exchange reduces the load imbalance. Instead of pairing the bins in sequence, an offset between the bins is used. The offset reduces the partitioning time because it quickly brings all bins in contact with each other. Usually, only a few iterations are necessary. The drawback of this approach is that it disregards the communication costs. By only focusing on the load imbalance, the locality both on the same refinement level and across different refinement level can be poor and result in high communication costs. The partitioner were originally implemented for a shared memory computer, which decreases the impact of large communication volumes.

The SAMR framework Chombo recently changed its default load balancing algorithm from recursive coordinate bisection (see Chapter 3.2.1) to a patch-based algorithm that uses space-filling curves [19, 20]. The new algorithm reduced the communication volumes and improved the scalability. The algorithm has many similarities with our patch-based partitioner described in Paper III. Both partitioners improve the locality by using space-filling curves [59] to order the grids. The linearly ordered grids are divided into intervals with equal workloads. We assume that the Chombo algorithm only divides the grids actually causing a processor to overload, but no details about this are given. With the new algorithm, the load imbalance is kept small while only a small fraction of the grids have neighbors located on other processors.

Lan, Taylor and Bryan [37] use a patch-based approach where grids from all refinement levels are partitioned simultaneously. They use the previous partitioning as a starting point. If the load imbalance is greater than a threshold, they first move complete grids between the processors. If no grids can be moved without the recipient getting overloaded, they start a new phase. Now, grids that will cause an overload are split to exactly match the underload on the recipient. During the partitioning process, the grids are moved and divided without respect to the refinement level. Thus, a processor can theoretically be assigned grids that exclusively belong to a single refinement level. If the assigned workload is unevenly distributed between the refinement levels, some processors will probably be idle at any given moment. Processors that have most of their work on higher refinement levels will have to wait while required data from lower refinement levels are computed. When these data are ready, other processors might have finished all of their assigned work. The remedy is to partition only a single refinement level at a time, distributing it over all processors.

3.2.3 Hybrid Partitioning

Hybrid partitioners combine elements from both the domain-based and the patch-based approach. These partitioners aim to avoid the shortcomings inherent in the two other partitioning approaches — the high communication costs for the patch-based partitioners and the intractable load imbalance for the domain-based partitioners.

The most common hybrid approach is hierarchical partitioning where the domain-based approach is usually employed to divide the coarse base grid into a number of large sub-partitionings. Each sub-partitioning is then assigned to a group of processors and a partitioning is computed for each processor group.

The load balancing scheme DistLB uses a hybrid and hierarchical approach to allow for heterogeneous computing environments in cosmology simulations [38]. In the initial domain-based step, which is called the global balancing phase, the coarse base grid is partitioned and assigned to homogeneous processor groups. For each homogeneous processor group, a local balancing phase that uses the previously described patch-based approach developed by Lan, Taylor, and Bryan is performed [37]. To minimize data migration, the global balancing phase is only triggered if the computational gain is estimated to be larger than the redistribution cost.

A different approach is used in the hybrid partitioning framework Nature+Fable: The computational domain is divided into refined and unrefined regions [62]. The unrefined regions are partitioned using a custom SFC patch-based approach. In the refined regions, adjacent refinement levels are clustered two-by-two into bi-levels. The same custom SFC patch-based algorithm is applied to the highest refinement level in each bi-level. The resulting partitioning is projected down to the lower refinement level in a domain-based fashion. Because the arithmetic workload is generally concentrated to the higher refinement level in a bi-levels, a good load balance can be achieved. Furthermore, the domain-based partitioning eliminates the inter-level communication inside each bi-level while the load imbalance is kept small by clustering no more than two levels together. The partitioning process in Nature+Fable is controlled by 12 parameters and each parameter setting can be regarded as a different partitioning algorithm.

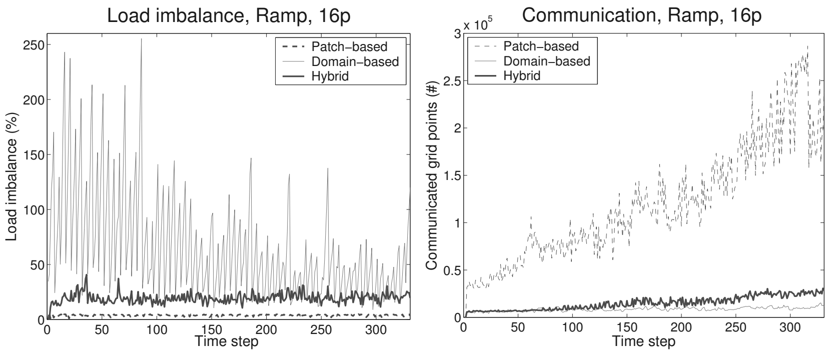
4. Dynamic Selection of Partitioning Algorithms

During the execution of parallel SAMR applications, the dynamic grid hierarchy is generally frequently re-partitioned and re-distributed over the processors. Individual partitioning algorithms produce high-quality partitionings for only certain types of grid hierarchies — no single partitioning algorithm results in high-quality partitionings for all grid hierarchies [62]. Similar observations have been made for unstructured adaptive grids [26,61,64]. The partitioning quality is also dependent on the characteristics and the current state of the parallel computer. Thus, to get the best possible performance, we need to dynamically select the partitioning algorithm with respect to the grid hierarchy as well as the parallel computer. This is non-trivial because the conditions for how to partition the hierarchy can change dramatically during run-time.

The range of possible SAMR grid hierarchies is infinite. To consistently achieve high parallel performance, we need access to a large number of complementing partitioning algorithms that together results in high-quality partitionings for a large range of grid hierarchies. Thus, we should preferably select between algorithms from all three partitioning approaches described in Chapter 3.2. The complementing characteristics of these approaches are shown in Figure 4.1. A patch-based partitioner results in small load imbalances but also in large communication volumes. A domain-based partitioner has an opposite behavior with small communication volumes and large load imbalances. The hybrid partitioner constructs partitionings with moderate load imbalances and communication volumes.

To select a suitable partitioning algorithm, we need to accurately characterize the current grid hierarchy and its partitioning needs. We also need to determine the characteristics and the current state of the parallel computer. By combining these properties, we can determine a partitioning focus that directs the partitioning effort towards the most performance-inhibiting factors. We use the partitioning focus to select the partitioning algorithm predicted to result in the highest quality partitioning. Because we base the selection on *predicted* performance, we might select a partitioning algorithm that results in poor performance. Thus, we like to evaluate the resulting partitioning before it is returned to the SAMR framework. If the performance is found to be poor, an alternative partitioning can be returned instead.

Finally, we need to implement these functions in a framework — a Meta-Partitioner. Given a grid hierarchy, the Meta-Partitioner should autonomously



(a) Load imbalance (b) Communication

Figure 4.1: Load imbalance and communication for three example partitioners. For the domain-based partitioner, we used the default partitioning algorithm in the SAMR framework AMROC [3]. The patch-based partitioner is presented in Paper III [35]. The results for the hybrid partitioner originate from the partitioning framework Nature+Fable and the performance characterization in Paper I [34]. Note the complementing behavior of the three partitioners. The example application, Ramp (from the Virtual Test Facility [68]) was partitioned for 16 processors.

select, invoke and evaluate partitioning algorithms using the guidelines described above. The framework has to be easy to expand and modify with new or improved partitioning algorithms and/or functionality. The implementation should be open to facilitate the use of arbitrary SAMR frameworks and partitioning algorithms. Furthermore, the Meta-Partitioner must be easy to use, even without prior knowledge about the SAMR application, the partitioning algorithms, and the parallel computer.

4.1 The Meta-Partitioner

In this section, we describe the general design and workflow of the Meta-Partitioner. The workflow is shown in Figure 4.2. Implementation details and performance evaluations are found in Chapter 5 and in Paper V and VI.

To simplify expansions and modifications, the Meta-Partitioner is designed using component-based software engineering (CBSE). The use of CBSE also makes it easier to integrate SAMR frameworks and other external components (e.g. performance monitoring tools and data base connections). For the implementation, we employ the Common Component Architecture (CCA), a community-based CBSE specification specifically targeted at high performance computing [11]. Existing software, e.g. SAMR frameworks, partitioners, and performance measurement tools, can easily be transformed into CCA components by the addition of a simple wrapper.

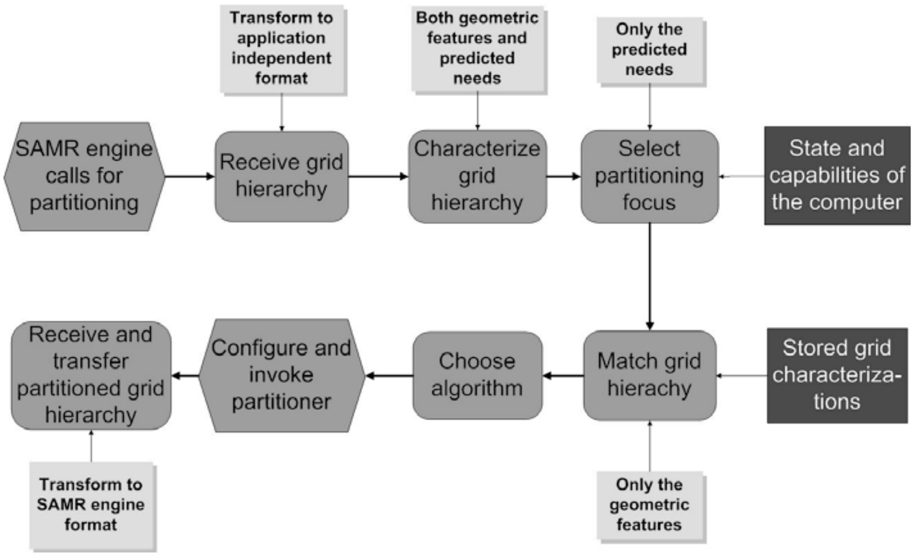

Transform to application independent format
Both geometric features and predicted needs
Only the predicted needs
SAMR engine calls for partitioning
Receive grid hierarchy
Characterize grid hierarchy
Select partitioning focus
State and capabilities of the computer
Receive and transfer partitioned grid hierarchy
Configure and invoke partitioner
Choose algorithm
Match grid hierachy
Stored grid characteriza-tions
Transform to SAMR engine format
Only the geometric features

Figure 4.2: The Meta-Partitioner workflow. The red hexagons represent tasks that are performed outside the Meta-Partitioner. The green boxes represent tasks performed by the Meta-Partitioner. The blue boxes contain input data while the yellow boxes are comments. Currently, the partitioning focus is static and determined before the start of the execution.

We base the selection of partitioning algorithms on the assumption that geometrically similar grid hierarchies have similar partitioning properties. An algorithm that results in a high-quality partitioning for a given grid hierarchy, probably also does so for geometrically similar grid hierarchies. As a consequence of this assumption, we can use historic performance data to select the partitioning algorithm. If the geometrical characteristics of the current hierarchy are matched with previously encountered hierarchies, the performance of a candidate partitioning algorithm can be predicted by its historical performance data for similar hierarchies.

We have built a data base that stores comprehensive performance data for 768 partitioning algorithms from the hybrid partitioning framework Nature+Fable [34, 62]. Each algorithm partitioned almost 1300 different grid hierarchies from four real-world SAMR applications. The data base contains all data necessary to predict the performance of a partitioning algorithm. When new partitioning algorithms become available, they must be characterized and included in the data base before they can be selected by the Meta-Partitioner.

Before the start of the execution, we select a static partitioning focus to direct the partitioning effort to one of the two performance-inhibiting factors that generally have the largest impact on the execution time — the load imbalance or the synchronization delays. Each partitioning focus, i.e `FocusLB` and `FocusSynch`, is associated with a maximum allowed performance deviation for its main performance-inhibiting factor. However, it is important to also consider the other performance-inhibiting factors when the algorithm is selected. The algorithm that results in the best performance for a certain performance-inhibiting factor will often perform poorly for all other factors.

At re-partitioning, the Meta-Partitioner receives the current grid hierarchy from a SAMR framework. The grid hierarchy is translated into an internal format. The Meta-Partitioner then computes a set of geometrical characteristics for the grid hierarchy. The characteristics for the current grid hierarchy are matched with the characteristics for all grid hierarchies stored in the performance data base. The *N* most similar stored grid hierarchies are recorded, where *N* is either selected by the user or a default value.

To select the partitioning algorithm, we use the data base to extract the performance of all eligible partitioning algorithms for the *N* most similar stored grid hierarchies. The extracted data is compared to the best recorded performance for the *N* stored grid hierarchies. For each of the *N* hierarchies, the algorithms that resulted in an equal or better performance than the maximum allowed performance deviation are selected as candidate algorithms (note that the performance deviation is dependent on the partitioning focus). 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`). For each of the *N* grid hierarchies, the candidate algorithm that resulted in the best performance for the secondary performance inhibiting factor is selected. Using this strategy, the partitioning effort is concentrated to the most performance-inhibiting factor while the impact of the secondary factor is kept as low as possible. Note that an allowed performance deviation of zero will result in the selection of the algorithm that has the best predicted performance for the most performance-inhibiting factor, completely disregarding the impact of the secondary performance-inhibiting factor. The selection of the partitioning algorithm is described in Algorithm 2.

Thus, we have selected the partitioning algorithms predicted to result in the best performance for each of the *N* most similar stored grid hierarchies. To select the algorithm that results in the best performance, we partition the current grid hierarchy with each of the *N* algorithms. The performance of the resulting partitionings are evaluated during run-time with a SAMR simulator [18]. The simulator mimics the execution of the common Berger-Colella SAMR algorithm and computes the load imbalance and synchronization penalty using the metrics described in Chapter 3.2. To select the best partitioning, we use an extended version of Algorithm 2. We do not strictly choose the partitioning that has the best performance for the primary performance-inhibiting factor — instead we allow a slight increase in the most performance-inhibiting factor if

Algorithm 2 Selection of partitioning algorithm

```
FocusSynch(deviation)
1 SELECT partAlg AS candidates FROM mostSimilarAppState WHERE
  synch < deviation*MIN_all(synch)
2 SELECT partAlg FROM candidates WHERE LB = MIN_cand(LB)

FocusLB(deviation)
1 SELECT partAlg AS candidates FROM mostSimilarAppState WHERE
  LB < deviation*MIN_all(LB)
2 SELECT partAlg FROM candidates WHERE synch = 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 candidate algorithms selected during step 1. In this thesis, `deviation`=1.25 for `FocusSynch` and `deviation`=1.2 for `FocusLB`

a large decrease is recorded for the secondary performance-inhibiting factor. The details of this algorithm are described in Paper VI.

Our motivation for invoking and evaluating more than one partitioning algorithm is the observation that the geometrical difference between the current grid hierarchy and the most similar stored grid hierarchies generally is small. A small change in one of the geometrical metrics often results in a change of the most similar stored grid hierarchy and hence also in a change of the partitioning algorithm. While most of the partitioning algorithms that correspond to the N most similar grid hierarchies are expected to result in good performance, some might result in either significantly better or worse performance. By invoking multiple algorithms, we are more likely to both generate partitionings with higher quality and to avoid bad performing algorithms.

Each of the N selected algorithms are uniquely determined by the combination of the partitioning focus and the stored grid hierarchy. Thus, it is possible to pre-compute the algorithm selection before run-time. For each combination of stored grid hierarchy and partitioning focus, the best performing algorithm is recorded. During run-time, the algorithm selection is reduced to finding the entries in a list that corresponds to the N most similar stored grid hierarchies.

For our experiments, neither the construction of multiple partitionings nor the performance evaluation add a significant overhead to the execution time (see Paper VI). Furthermore, both the partitioning and the evaluation is embarrassingly parallel and they can easily be performed in parallel if the need should arise.

To evaluate the performance of the Meta-Partitioner, we use un-partitioned trace files from four SAMR applications [69]. 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 grid hierarchy at a time. The internal functionality of the Meta-Partitioner is independent of the origins of the current grid hierarchy. After selecting, invoking and evaluating multiple partitionings, the best performing partitioning at each re-partitioning is stored on disk. The stored partitionings are later evaluated using a more comprehensive version of the SAMR simulator described above [18]. In addition to the load imbalance and the synchronization delays, the comprehensive version of the simulator also computes metrics like communication volumes and the number of grids.

Future Extensions and Improvements

The Meta-Partitioner can be expanded and improved in several ways. The most important task is to interface real-world SAMR frameworks with the Meta-Partitioner. Currently, the Meta-Partitioner is limited to the partitioning of application trace files.

The partitioning focus is static throughout the execution. To dynamically change the partitioning focus during run-time, the partitioning needs of the current grid hierarchies must be predicted. Steensland and Ray [65, 66] have proposed a number of metrics that perform this task and their integration into the Meta-Partitioner is straightforward. Furthermore, performance measurement tools that continuously monitor the state of the computer should be added. Several such tools exist, e.g. the Network Weather Service [72] and Remos [24].

In the current implementation, only hybrid partitioning algorithms from the partitioning framework Nature+Fable are used. To realize the full potential of the Meta-Partitioner, both domain-based and patch-based partitioning algorithms should be added.

To use the Meta-Partitioner, extensive knowledge about its implementation is currently needed. To be accessible to the common user, an intuitive user interface has to be constructed.

Finally, a comprehensive evaluation of the real-world impact of the performance-inhibiting factors must be performed when the Meta-Partitioner has been interfaced with a SAMR framework. This evaluation could result in tuning of the partitioning focus and the algorithm selection process.

4.2 Related Work

In this section we present work related to the Meta-Partitioner. The section is divided into two parts. First, we describe existing dynamic partitioning frameworks for both structured and unstructured grids. In the second part, we discuss the much larger and more general research field of recommender systems.

4.2.1 Dynamic Partitioning Frameworks for Grid Hierarchies

ARMaDA, developed at Rutgers University, is an adaptive partitioning framework for dynamic structured grid hierarchies [14,15,17]. The framework consists of three components; an application state characterization component, an octant-partitioner mapping policy, and a meta-partitioner component that selects and invokes the partitioning algorithm. In ARMaDA, the current grid hierarchy is characterized using three metrics. The first metric computes a computation-to-communication ratio, the second metric approximates the application dynamics (i.e. rate of change) and the last metric concerns the refinement pattern (i.e. scattered vs. localized). After normalization, each metric is converted to a bit-value (i.e. 0 or 1). The three metrics are then mapped to a octant position using a three-bit binary pattern. The resulting octant is used to select the most appropriate partitioning algorithm (currently one of four available domain-based partitioning algorithms). Several extensions to ARMaDA have been proposed and implemented. [15,16,17].

Excluding the Meta-Partitioner, ARMaDA is to our knowledge the only existing adaptive partitioning framework for structured grid hierarchies. The presented experiments show a decreased execution times compared to the single static partitioning algorithm that results in the shortest execution time. The octant approach allows for eight distinctive characterizations of the current grid hierarchy and it limits the number of partitioning algorithms to eight. To consistently achieve good parallel SAMR performance, we believe that a continuous classification space that spans over more dimensions is necessary. Such a classification space allows for both a more accurate classification of the application and the use of a larger number of complementing partitioning algorithms.

The data management toolkit Zoltan for unstructured adaptive meshes provides services for load-balancing, data movement, communication, and memory management [23]. Instead of imposing a particular data structure on the applications, Zoltan uses call-back functions to determine necessary information about the application. Several partitioners are included in the toolkit, including recursive bisection [7], SFC partitioning, and spectral and multilevel graph partitioning [61]. To change the partitioning algorithm, the user simply sets a run-time parameter. However, the choice(s) of partitioner(s) is made before run-time because Zoltan currently lacks necessary components for adaptive partitioning (e.g. characterization of the state of the application and the computer).

Zoltan has been used to examine the benefits of adaptive partitioning of unstructured meshes [64]. Six applications were adaptively partitioned by five complementing partitioners and the results were evaluated using a cost function. The adaptive partitioning performed as good as the best static partitioning algorithm for each application. Hence, the main benefit of adaptive partitioning for unstructured meshes is not an improved parallel efficiency but an elimination of an often expensive search for suitable static partitioning algorithms.

The Dynamic Resource Utilization Model (DRUM) is a resource-aware tool that extends the capabilities of Zoltan for heterogeneous and dynamic computing environments [67]. Drum maintains information about the system characteristics and monitors its current performance. When partitioning is requested, the performance data is used to determine appropriate parameters for a single partitioning algorithm. Using Drum, computational nodes can be assigned a workload that match their capabilities, even if their performance change during run-time due to the concurrent execution of other applications.

PLUM is a framework for parallel adaptive flow computations [47]. The framework includes a dynamic load balancing strategy for unstructured meshes. After re-partitioning, the partitioning is evaluated using a set of metrics for load imbalance, communication, and data migration. If the computation gain is expected to be higher than the data migration cost, the new partitioning is committed. The Meta-Partitioner can be extended to consider the data migration in a similar fashion. If the previous partitioning is stored, the SAMR simulator can compute the resulting data migration for each of the *N* partitionings without a significant overhead.

Although not designed as an adaptive partitioning framework, the widely used graph-based partitioning toolkit ParMETIS uses a similar technique as the Meta-Partitioner to select between partitioners [61]. Originally presented as the Unified Repartitioning Algorithm, the method aims to construct a partitioning that simultaneously minimizes the impact of both communication and data migration (the load imbalance is assumed to be sufficiently small). Two partitionings from complementing partitioning approaches (scratch-remap and diffusion-based) are always constructed at each re-partitioning. The two partitionings are evaluated using a parameter that describes the relative communication/migration cost and the partitioning that results in the lowest cost is selected. This approach is similar to the one used in the Meta-Partitioner where *N* partitioning algorithms are invoked and evaluated at each re-partitioning.

4.2.2 Recommender Systems

The large and active research field of recommender systems is concerned with the rating and selection of items [1]. A recommender system estimates ratings for items that have not yet been seen by a user. The estimations are usually based on ratings given by the user(s) to other items. The yet unrated item with the highest estimated rating is then recommended to the user. For the SAMR partitioning problem, the user can be seen as the grid hierarchy and the items as the available partitioning algorithms. Thus, many ideas and concepts used in recommender systems are of interest for the Meta-Partitioner.

Recommender systems are usually classified into three categories [1]. For content-based recommendations, the system will recommended items similar to items that the user have preferred in the past. A collaborative recommendation is based on item ratings by users with similar preferences as the user. Finally, there exist hybrid approaches that combine the content-based and the collaborative recommendations.

Clearly, a Meta-Partitioner can not implement content-based recommendations because these recommendations require that we have previously encountered all grid hierarchies that might arise during a simulation. However, a collaborative approach can be used in a Meta-Partitioner. The recommendations are based on items (candidate partitioning algorithms) that have previously been rated by other users (grid hierarchies) — the same methodology that we actually use in the Meta-Partitioner.

Collaborative recommendations are often grouped into two classes; memory-based and model-based. For the model-based approach, a collection of ratings are used to build a model for the recommendations. The memory-based approach computes the rating as an aggregate of the ratings given to the item by other users (usually, the M most similar users). For $M = 1$, the memory-based approach is identical to the selection approach in the Meta-Partitioner — we use the most similar user (grid hierarchy) to select the item (partitioning algorithm) that was given the highest rating by the current user (while the Meta-Partitioner initially selects N algorithms, the selection of the individual algorithms correspond to $M = 1$ because we only use ratings from a single grid hierarchy at a time). Hence, in the context of recommender systems, the Meta-Partitioner can be classified as a memory-based collaborative recommender system.

Recommender systems are most common in business applications where they are used to recommend books, music, movies, and electronics etc. to potential customers [39]. In scientific computing, recommender systems has so far seen a more limited use.

The PYTHIA-II framework aims to recommend the best-performing solver for a known class of equations and a given computer system [32]. The framework also estimates values for associated parameters like grid size, accuracy, and solution time. A collaborative model-based approach is used for the recommendation of the solver and the parameters. The framework has a web-based interface that makes it easy to add new performance data and models. In an evaluation with a limited number of users (families of PDEs) and items (solvers), the framework is generally able to recommend a good-performing solver for the specified PDE.

JAMES II is a simulation framework that can be extended to select simulation algorithms for a single problem [27]. The creators behind JAMES II argue that a framework limited to a single problem at a time can achieve better results than a more general and complex selection framework like PYTHIA-II. In its core version, JAMES II is a flexible, extensible, and modular framework for the execution of large numbers of simulations [31]. When performance data is available for the problem at hand, an algorithm selection plug-in can be attached to the framework. Before run-time, a simulation algorithm selection rule is generated from the performance data and the entries in the rule are ordered with respect to their performance. During run-time, incompatible simulation algorithms are temporarily removed from the rule and the first remaining entry in the rule (i.e. the algorithm having the best predicted performance) is selected and invoked. This selection methodology is similar to the selection process in the Meta-Partitioner — both frameworks use the collaborative memory based approach, they pre-compute the rules, and they remove incompatible algorithms (the algorithm selection in the Meta-Partitioner can be seen as a removal of the algorithms that do not correspond to the most similar stored grid hierarchies). JAMES II has been used to evaluate the performance of a limited number of graph-based partitioning algorithms for unstructured meshes [26]. We are not aware if the partitioning data have been used to adaptively select partitioning algorithms during run-time.

The performance of a numerical simulation is generally influenced by a number of continuous-valued attributes, e.g. scalars present in the PDE and parameters in the candidate solvers. Ramakrishnan and Ribbens describe a method where association rule mining is used to determine the best solver for different attributes [55]. In association rule mining, the goal is to find connections between entities that occur together, i.e. connections between features of PDE problem instances and the solver(s) that performed best for these instances. Hence, association rule mining is a model-based approach. In the case studies presented by Ramakrishnan and Ribbens, mined association rules are successfully used to select the best of two different serial solvers when up to three continuous-valued attributes are varied. For the Meta-Partitioner, we believe that it is infeasible to use association rule mining. When a rule is constructed, data are generally discarded or aggregated. Because of the similarities between many of the stored grid hierarchies and the sensibility of the algorithm selection, we are convinced that we must use all available data to consistently generate high-quality partitionings.

Vuduc, Demmel and Bilmes discuss automatic tuning of library subroutines using statistical models [70]. Libraries allow for portable high performing applications, provided that the libraries are tuned for the platform of choice. Normally, a large number of possible tuning strategies are automatically tested and the fastest implementation is selected. This search for a good performing implementation is generally very time-consuming. Vuduc, Demmel and Bilmes show how the search can be stopped early with methods from statistics. Before the start of the search, the user specifies the proximity to the best performance as well as a desired confidence. The search is terminated when a sufficiently good implementation is found (i.e. equal to or better than the desired performance) at the specified confidence. This early stopping method can be used in the Meta-Partitioner when partitioning performance data for new grid hierarchies are added to the performance data base. Instead of partitioning the new grid hierarchies with all available partitioning algorithms, we can stop the partitioning process pre-maturely to save time.

5. Contributions

In this chapter, we describe the contributions of the papers included in this thesis. We present the motivation, the design, and a suggestion for an initial implementation of the Meta-Partitioner in Paper II. This paper can been seen as the template for the Meta-Partitioner and can be used as a reference guide for the other papers in the thesis. In Paper II, we discuss the need for adaptive partitioning and the necessity to have access to a large number of complementing partitioning algorithms to consistently construct high-quality partitionings. We describe how the partitioning effort is focused on the performance-inhibiting factors with the largest impact on the partitioning quality. We present our design philosophy behind the Meta-Partitioner and how the Meta-Partitioner can be implemented with component-based software engineering. We describe the workflow of Meta-Partitioner and the resulting components separately to give a clear presentation of the algorithm selection process and the suggested implementation. The extensions to the Meta-Partitioner presented in Paper VI are not included in Paper II.

The rest of this chapter is divided into two parts. The first part lays the foundation for the Meta-Partitioner. The second part presents the implementation and performance of two versions of the Meta-Partitioner.

5.1 Foundations

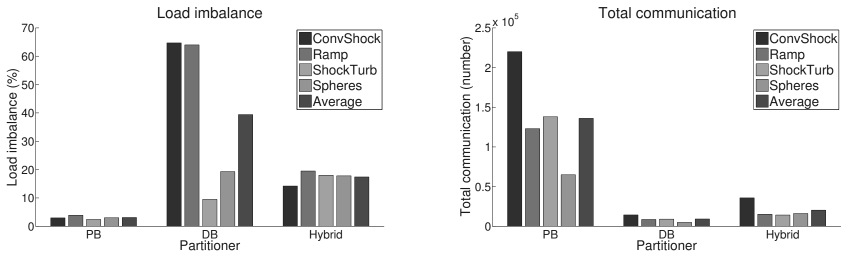
To consistently select good performing partitioning algorithms during run-time, we need performance data for all candidate partitioning algorithms. Without it, it is impossible to predict the quality of the resulting partitionings. In Paper I, we present a comprehensive performance characterization of a large number of partitioning algorithms [34]. The analysis is an extension of an early characterization that involved a small number of hybrid partitioning algorithms from Nature+Fable [33]. The goal of the initial characterization was to manually find simple relations between the algorithm, the grid hierarchy, and the performance. We could often select a hybrid partitioning algorithm to influence a specific performance metric, i.e. load imbalance, but the amount of change was inconsistent. The effects on other performance metrics, i.e. communication and synchronization delays, were generally unpredictable.

To construct more advanced algorithm selection methods, we needed a comprehensive performance characterization. Compared to the initial characterization, we substantially expanded the number of hybrid partitioning algorithms. We also complemented the characterization with performance data from the domain-based partitioner used in the SAMR framework AMROC [3, 21]. For the characterization, we used trace files from four real-world applications taken from the Virtual Test Facility [68, 22]. The *Ramp*-application simulates the reflection of a planar Mach 10 shock wave striking a 30 degree wedge. 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 that creates a Richtmyer-Meshkov instability. 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 that is created 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 initial grid size is 200x160 grid points and the application uses three levels of refinement with a constant refinement factor of two. To evaluate the partitionings, we used the SAMR simulator described in Chapter 4 [18]. The performance data were stored in a data base for easy access and evaluation.

The performance data verified the complementing characteristics of the hybrid and domain-based partitioning approaches (see Figure 4.1 and Chapter 3). The hybrid algorithms generally performed better for complex and scattered refinement patterns, while the domain-based algorithm was more successful when the refined areas were large and uniform. The characterization also proved the viability of the Meta-Partitioner — we observed large performance differences between the best performing partitioning algorithms and a random partitioning algorithm. With sufficiently many complementing algorithms, the characterization showed that there will generally be a number of partitioning algorithms that results in high-quality partitionings at each re-partitioning.

To consistently select partitioning algorithms that generates high-quality partitionings, we need to use complementing partitioning algorithms from all major partitioning approaches. To our knowledge, there does not exist any stand-alone patch-based partitioner for SAMR grid hierarchies that can be used by the Meta-Partitioner. In Paper III, we present a patch-based partitioner suitable for the Meta-Partitioner [35]. We are unaware of any comprehensive performance evaluation of a similar patch-based partitioner.

To preserve the locality and to decrease the communication volumes, the patch-based partitioner orders the grid patches on each refinement level according to an inverse SFC (see Chapter 3.2.1) [4]. For each refinement level, the partitioner divides the ordered list of grid patches into *p* parts, where *p* is the number of processors.



(a) Load imbalance (b) Total communication volume

Figure 5.1: The load imbalance and the total communication volume for the patch-based partitioner compared to the hybrid partitioner Nature+Fable [62] and the domain-based partitioner from the SAMR framework AMROC [3]. Note the small load imbalance for the patch-based partitioner compared to the two other partitioners.

The performance evaluation shows a small and stable load imbalance for all applications. On average, the load imbalance was only 3.1%. The load imbalance was also significantly smaller than the imbalance for the two other common types of partitioners. Patch-based partitioners generally suffer from high communication volumes. By using SFCs to increase the locality, the total communication volume was decreased with 17 percent. Compared to the domain-based partitioner in the SAMR framework AMROC and the hybrid partitioner Nature+Fable, the total communication volume was still 6.7 respectively 4.5 times higher for the patch-based partitioner.

The performance data base that resulted from the characterization presented in Paper I is crucial for the algorithm selection process. In paper IV, a data base interface component for the Common Component Architecture (CCA) is presented. The CCA is the component-based software engineering framework that we use for the implementation of the Meta-Partitioner. The database interface component was developed in the wider context of Computational Quality of Service (CQoS) [42, 45]. The definition of CQoS is the ability of a system to ensure that a scientific problem is solved with the best available hardware and software resources. This definition coincides with the goal of the Meta-Partitioner.

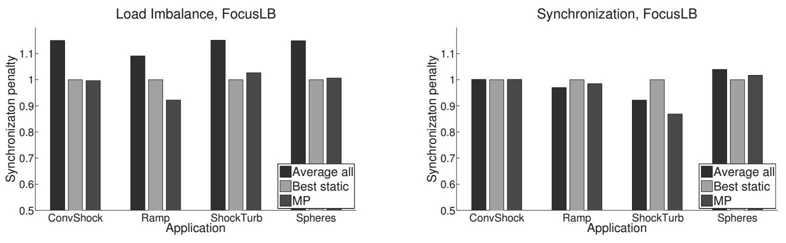
With the database interface component, the Meta-Partitioner can be expanded with the ability to access the performance data base, or any other data base, during run-time. In Paper IV, we describe how the matching of the grid hierarchies can be performed directly in the data base. If we move the matching to the data base, we will retrieve data from the data base at each re-partitioning. The data retrieval might add an overhead to the selection process but it can also facilitate new algorithm selection methods.

Substantial amounts of data used by the Meta-Partitioner during run-time are currently stored on disk. With the database interface component, much of the data can be moved to a data base for a more efficient retrieval during run-time.

We can also use the data base component to maintain a globally accessible data base that contains the characteristics of all encountered grid hierarchies, the performance data for all partitioning algorithms, and all algorithm selection rules. Before run-time, the user can download the most up-to-date selection rules and any recently added grid hierarchies. Furthermore, after the execution, the user can upload the encountered grid hierarchies to the data base. The new hierarchies can be partitioned with all available algorithms to improve the selection rules.

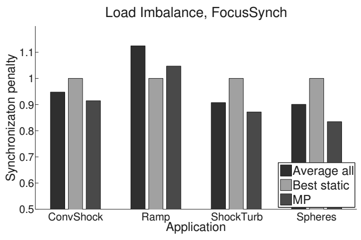
5.2 Implementation and evaluation

Our initial implementation of the Meta-Partitioner is presented in Paper V. The implementation follows the general description in Chapter 4.1 with the exception that the algorithm selection is limited to a single partitioning algorithm (N=1). The initial implementation was evaluated with the same applications used in Paper I. For both partitioning focuses (`FocusLB` and `FocusSynch`), the initial version resulted in consistent and significant improvements for the most performance-inhibiting factor compared to the average result for all partitioning algorithms. For `FocusLB`, the load imbalance was decreased by 10.9-15.5% compared to the average performing partitioning (see Figure 5.2). When `FocusSynch` was used, the reduction in the average synchronization penalty was in the interval 1.7% to 10.8% (see Figure 5.3). Compared to the static partitioning algorithms that in average resulted in the best performance, the improvements were smaller but generally noticeable. Furthermore, the Meta-Partitioner did not negatively affect the results for the secondary performance-inhibiting factor.

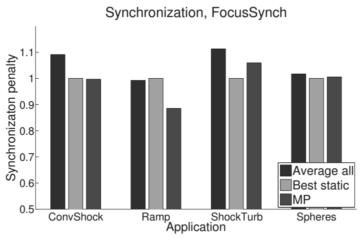


(a) Load imbalance (b) Synchronization

Figure 5.2: Results for the initial version of the Meta-Partitioner and the partitioning focus `FocusLB`. Note that the Meta-Partitioner consistently generates partitionings with a smaller load imbalance than the average imbalance for all algorithms.



(a) Load imbalance



(b) Synchronization

Figure 5.3: Results for the initial version of the Meta-Partitioner and the partitioning focus `FocusSynch`. Note that the Meta-Partitioner consistently generates partitionings with smaller synchronization penalties than the average penalty for all algorithms.

We also performed a theoretically derived analysis of the stability and accuracy of the matching process in the initial implementation. The results showed that large performance gains were possible if multiple partitioning algorithms are invoked and evaluated at each re-partitioning. While the Meta-Partitioner could easily be modified to select and invoke multiple algorithms, it was intractable to evaluate the resulting partitionings during run-time due to the long execution time of the SAMR simulator.

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

The shorter execution time of the modified simulator made it possible to evaluate the resulting partitionings during run-time. We expanded the modified simulator into a CCA component and connected the simulator to the Meta-Partitioner. The details of the extended Meta-Partitioner are described in Paper VI. Before execution, the user can choose the number of partitioning algorithms that will be selected and invoked at each re-partitioning.

The user is also presented with several options for how to select the 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 matched grid hierarchies. A second option is to use the performance data base to determine the partitioning algorithm that has the best average performance and invoke this algorithm together with any other algorithm(s). This option is labeled `'Static'`. For geometrically unusual grid hierarchies, the performance might benefit if the matched 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 is labeled `'Apps'`. Finally, different combinations of the approaches can also be used (see Table 5.1).

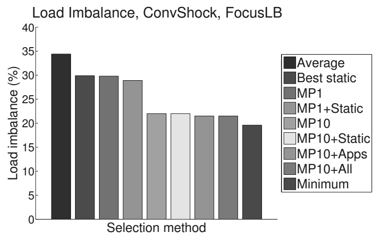
Name	**Description**	**#Algorithms**
`MP1`	Most similar grid hierarchy	1
`MP1+Static`	Most similar grid hierarchy and the best static algorithm	2
`MP10`	10 most similar grid hierarchies	10
`MP10+Static`	10 most similar grid hierarchies and best static algorithm	11
`MP+Apps`	10 most similar grid hierarchies 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 5.1: *The selection approaches used by the extended version of the Meta-Partitioner 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.*

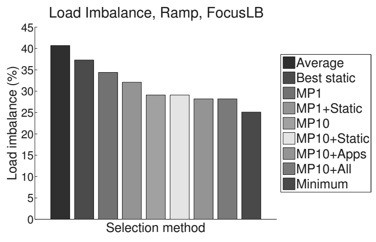
To evaluate the extended version of the Meta-Partitioner, we re-used the real-world applications from the Virtual Test Facility. Note that the performance data base includes performance data for all possible combinations of partitioning algorithm and grid hierarchy that can be encountered during the evaluation. Hence, we could compute a minimum value for each metric and application used in the evaluation.

The initial and the extended version of the Meta-Partitioner are functionally identical when a single algorithm is selected and invoked at each re-partitioning (i.e. `MP1`). Hence, the `MP1`-configuration and the initial implementation resulted in identical performance. For the `MP1+Static` configuration and `FocusLB`, the load imbalance was on average decreased by 4.8% compared to `MP1` (see Figure 5.4). `MP1+Static` and `FocusSynch` decreased the synchronization penalty with on average 10.2% compared to `MP1` (see Figure 5.5).

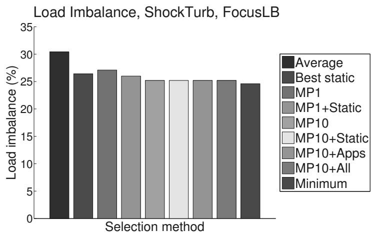
For the `MP10`-configurations, both the load imbalance and the synchronization penalty were significantly reduced compared to the `MP1`-configuration. For `FocusLB`, the load imbalance was reduced with on average 21%. The synchronization penalty was decreased with 15.5% when the `FocusSynch` was used. More importantly, these reductions brought both the load imbalance and the synchronization penalty close to their respectively minimum values. On average, the load imbalance was increased with only 11.5% compared to its minimum value. For the ShockTurb application, the increase in load imbalance was as small as 2.4%. The synchronization penalty was on average



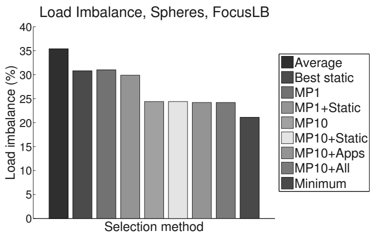
(a) ConvShock2D



(b) Ramp2D



(c) ShockTurb2D



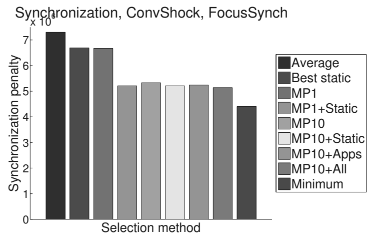
(d) Spheres2D

Figure 5.4: Load imbalance for the different configurations of the extended Meta-Partitioner and the partitioning focus `FocusLB`. Note that the configurations based on `MP10` result in a load imbalance that is close to the minimum imbalance.

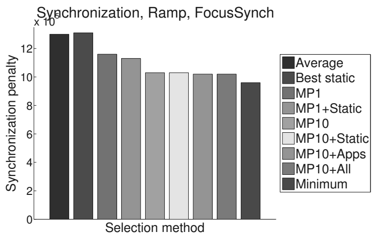
increased by 13.6% compared to its minimum value. For both partitioning focuses, the `MP10`-configuration often selected the best performing partitioning algorithm among all of the 768 partitioning algorithms!

The more advanced configurations that are based on `MP10` — `MP10+Static`, `MP10+Apps`, and `MP10+All` — all result in further, but small reductions of the load imbalance and the synchronization penalty. Due to the longer partitioning times of these configurations (see Table 5.2), they will probably not decrease the total execution time for the application.

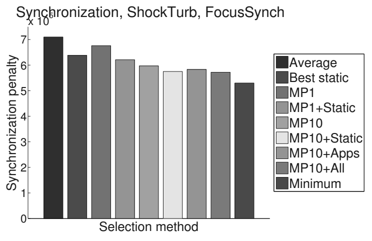
The algorithm selection and the construction and evaluation of multiple partitionings add an overhead to the execution time of the application. To decrease the execution time, the overhead must be smaller than the resulting reduction in simulation time. Generally, the most time consuming component of the Meta-Partitioner is the construction of the partitionings (see Table 5.2). The partitioning time grows proportionally to the number of constructed partitionings. The time needed to match the grid hierarchies, select one or more partitioning algorithms, and evaluate the resulting partitionings was substantially smaller than the partitioning time. Currently, all partitionings are constructed and evaluated by a single processor. However, the partitionings can easily be constructed and evaluated in parallel to significantly decrease the overhead generated by the Meta-Partitioner.



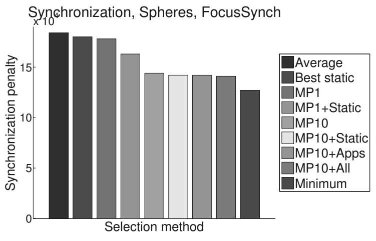
(a) ConvShock2D



(b) Ramp2D



(c) ShockTurb2D



(d) Spheres2D

Figure 5.5: The synchronization penalty for the different configurations of the extended Meta-Partitioner and the partitioning focus `FocusSynch`. Note that the configurations based on `MP10` result in a synchronization penalty that is close to the minimum penalty.

5.3 Conclusions

In this thesis, we present the Meta-Partitioner — a partitioning framework for parallel SAMR applications that autonomously selects, invokes, and evaluates multiple partitioning algorithms during run-time. The basis of the Meta-Partitioner is a comprehensive performance data base that currently contains partitioning data for 768 partitioning algorithms and over 1300 SAMR grid hierarchies. To complement the available partitioning algorithms, we developed a patch-based partitioner. The patch-based partitioner produced a smaller load imbalance than partitioners from the other partitioning approaches. We also presented a data base interface component that can extend the capabilities of future versions of the Meta-Partitioner.

The initial implementation of the Meta-Partitioner selected the partitioning algorithm that was predicted to result in the best performance with regard to the most-performance inhibiting factor. To make the Meta-Partitioner easy to modify and expand, we used component-based software engineering for the implementation. The performance evaluation showed that the initial implementation consistently generated partitionings with a better performance than the average partitioning. Furthermore, a theoretical analysis showed that large performance gains were possible if several alternative partitionings could be evaluated during run-time.

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

To evaluate multiple partitionings during run-time, we developed a substantially faster version of the SAMR simulator. We incorporated the new simulator into an extended version of the Meta-Partitioner. The performance evaluation showed that the extended Meta-Partitioner consistently generated high-quality partitionings. The resulting performance were close to the optimal performance that can be achieved with the available partitioning algorithms. Often, the Meta-Partitioner even selected the best performing partitioning algorithm among all 768 different algorithms! When the Meta-Partitioner has been interfaced with a SAMR framework, it has the potential to significantly reduce the execution time for parallel SAMR applications.

Svensk sammanfattning

Många naturvetenskapliga experiment är antingen så dyra, så stora eller så tidskrävande att de i praktiken inte går att genomföra. Den snabba utvecklingen inom datorområdet har medfört att många experiment kan simuleras med hjälp av en dator.

Innan en datorsimulering kan genomföras måste det naturvetenskapliga experimentet ersättas med en matematisk modell. Modellen innehåller ofta partiella differentialekvationer (PDE:er). Eftersom en PDE normalt inte kan lösas analytiskt måste dess lösning approximeras. Beräkningsområdet delas först upp i ett strukturerat diskret nät. Efter att en lämplig beräkningsmetod valts approximeras lösningen genom att metoden appliceras på samtliga punkter i beräkningsnätet.

I många simuleringar upptar de fenomen som kräver hög upplösning hos beräkningsnätet endast en mindre del av nätet. För att minska exekveringstiden kan upplösningen hos nätet anpassas efter lösningens egenskaper. Simuleringen startar då med ett basnät som har en låg upplösning. Genom att uppskatta det lokala beräkningsfelet kan nät med en högre upplösning läggas ovanpå de områden som har ett stort fel. Den här metoden kallas för adaptiv nätförfining[1] och den ger upphov till en hierarki av beräkningsnät. Näthierarkin förändras kontinuerligt under simuleringen allteftersom nät skapas, flyttas och tas bort.

Exekveringstiden kan minskas ytterligare om simuleringen utförs på en parallelldator. Varje processor i parallelldatorn approximerar då lösningen för en del av beräkningsnätet. Eftersom näthierarkin anpassas adaptivt efter beräkningsfelet kan hierarkin se helt olika ut vid två närliggande tidpunkter. Det här är ett stort potentiellt problem när simuleringen utförs på en parallelldator. Några processorer kommer med tiden troligen att få en för hög arbetsbelastning medan andra processorer kommer att få för en låg arbetsbelastning. Processorernas arbetsfördelning måste därför kontinuerligt uppdateras. En sådan uppdatering kallas för lastbalansering och den resulterande uppdelningen kallas för partitionering. Lastbalanseringen kompliceras av att exekveringstiden beror på fler faktorer än en jämn fördelning av beräkningsarbetet, t ex har kommunikation, datamigrering och synkronisering stor påverkan på exekveringstiden. Även parallelldatorns egenskaper och dess aktuella arbetsbelastning påverkar exekveringstiden.

[1]eng. structured adaptive mesh refinement (SAMR)

Ingen enskild lastbalanseringsalgoritm ger ett bra resultat för alla näthierarkier. För att minimera exekveringstiden bör lastbalanseringsalgoritmen därför väljas dynamiskt under simuleringen. Optimalt ska algoritmen väljas så att inverkan från samtliga faktorer som påverkar exekveringstiden minimeras. Det är ett svårt problem eftersom förutsättningarna för lastbalanseringen kan förändras kraftigt under simuleringen.

Meta-Partitioneraren

I den här avhandlingen presenteras en Meta-Partitionerare för parallelldatorsimuleringar baserade på adaptiv nätförfining. Meta-Partitioneraren väljer, exekverar och utvärderar automatiskt lastbalanseringsalgoritmer under simuleringens gång. För att förenkla framtida utvidgningar och modifieringar har komponentbaserad mjukvaruutveckling använts för implementeringen av Meta-Partitioneraren.

Eftersom exekveringstiden beror på många faktorer är det orealistiskt att hitta den absolut bästa lastbalanseringsalgoritmen för den aktuella näthierarkin. Istället fokuseras lastbalanseringen på att minimera effekten av de två faktorer som har störst påverkan på exekveringstiden, lastbalansen och synkroniseringstiden.

Valet av lastbalanseringsalgoritm baseras på att liknande näthierarkier resulterar i liknande partitioneringar när de lastbalanseras med samma algoritm. Detta beteende medför att data från tidigare lastbalanseringar kan användas för att välja algoritm för den aktuella hierarkin.

En databas med information om 768 lastbalanseringsalgoritmer utgör grunden för algoritmvalet. Varje algoritm användes för att lastbalansera nästan 1300 näthierarkier från fyra olika simuleringar. I databasen finns information om lastbalans, kommunikation och synkronisering för de olika kombinationerna av algoritm och näthierarki.

När näthierarkin behöver lastbalanseras karaktäriseras först hierarkins geometriska egenskaper. Dessa egenskaper jämförs sedan med de geometriska egenskaperna hos de näthierarkier som finns sparade i databasen. Ur databasen extraheras data som beskriver utfallet av samtliga lastbalanseringar för de X hierarkier som är mest lika den nuvarande hierarkin, där X är ett antal som bestämts av användaren. För varje hierarki väljs den algoritm ut som historiskt presterade bäst med avseende på en kombination av två de faktorer som har störst påverkan på kvalitén hos partitioneringen (lastbalansering och synkronisering).

Eftersom valet av lastbalanseringsalgoritm är baserat på historisk prestanda för liknande hierarkier så går det inte att garantera att en vald algoritm resulterar i en bra partitionering. Genom att välja flera algoritmer minskas risken att få en dålig partitionering. Samtliga algoritmer som valts ut får partitionera den aktuella hierarkin. De resulterande partitioneringarna utvärderas och partitioneringen med högst kvalité används av simuleringen.

En omfattande analys av Meta-Partitionerarens prestanda visade goda resultat. Genom att använda Meta-Partitioneraren minskades lastobalansen med i snitt 28,2% jämfört med medelvärdet för samtliga partitioneringar i analysen. För synkroniseringen var motsvarande minskning 21,5%. När resultaten jämfördes med det globala minimumvärdet för de aktuella applikationerna och algoritmerna ökade lastobalansen och synkroniseringen med endast 11,5% respektive 13,6%. För många näthierarkier valde Meta-Partitioneraren ut den bäst presterande algoritmen av totalt 768 algoritmer. Resultaten visar att Meta-Partitioneraren har potential att kraftigt minska exekveringstiden för parallelldatorsimuleringar baserade på adaptiv nätförfining.

Acknowledgments

I thank my three supervisors Jarmo Rantakokko, Michael Thuné, and Johan Steensland for giving me support, expertise and encouragement. Some people believe that three supervisors is at least one too many. I disagree. For me, the three of you have complemented each other perfectly. Jarmo, you are the calm voice in the eye of the storm. Michael, you always manage to motivate me and to put everything into perspective. Johan, your help with Nature+Fable and the art of partitioning has been invaluable. I also need to give a special thank you to Johan for photographic inspiration, proof-reading, and lots of help during my visits to Livermore.

I am grateful to Ralf Deiterding at Oak Ridge National Laboratory for both providing the application trace files and the performance data for the domain-based partitioning algorithm in AMROC.

I thank Jaideep Ray and Ben Allan at Sandia National Laboratories for valuable help with CCA. Without your help, the implementation of the Meta-Partitioner would both have been more difficult and more time-consuming. Jaideep also helped me with all practical matters concerning my four visits to Sandia National Laboratories (including taking some flak on my behalf because I visited the The Pirate Bay from a United States Government computer).

During my time as a PhD-student, I have spent considerable time in California. My visits would not have been possible without the economic support provided by Stiftelsen Anna Maria Lundins stipendiefond (Smålands Nation) and Jacob Letterstedts stiftelse (The Royal Swedish Academy of Sciences). Thank you!

To my colleagues at the Department of Information Technology, thank you! You made this a great place to work at. I will miss you all!

Finally, I thank my family and all of my friends. You know who you are. Thank you!

Bibliography

[1] Gediminas Adomavicius and Alexander Tuzhilin. Toward the next generation of recommender systems: A survey of the state-of-the-art and possible extensions. *IEEE Transactions on Knowledge and Data Engineeing*, 17(6):734–749, 2005.

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

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

[4] Tetsuo Asano, Desh Ranjan, Thomas Roos, Emo Welzl, and Peter Widmayer. Space filling curves and their use in the design of geometric data structures. *Theoretical Computer Science*, 181(1):3–15, Jul 1997.

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

[6] Stephen T. Barnard and Horst D. Simon. A fast multilevel implementation of recursive spectral bisection for partitioning unstructured problems. *Concurrency: Practice and Experience*, 6:101–107, 1994.

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

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

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

[10] Marsha J. Berger and Isidore Rigoutsos. An algorithm for point clustering and grid generation. *IEEE Transactions on Systems, Man, and Cybernetics*, 21(5):1278–1286, 1991.

[11] 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.

[12] Shahid H. Bokhari, Thomas W. Crockett, and David M. Nicol. Binary dissection: Variants & applications. Technical Report TR-97-29, Institute for Computer Applications in Science and Engineering (ICASE), 1997.

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

[14] Sumir Chandra and Manish Parashar. ARMaDA: An adaptive application-sensitive partitioning framework for SAMR applications. In *International Conference on Parallel and Distributed Computing Systems*, pages 441–446, 2002.

[15] Sumir Chandra and Manish Parashar. Towards autonomic application-sensitive partitioning for SAMR applications. *Journal of Parallel and Distributed Computing*, 65(4):519–531, 2005.

[16] Sumir Chandra, Manish Parashar, and Salim Hariri. GridARM: An autonomic runtime management framework for SAMR applications in grid environments. In *Autonomic Applications Workshop, 10th International Conference on High Performance Computing (HiPC 2003)*, pages 286–295, 2003.

[17] Sumir Chandra, Manish Parashar, Jingmei Yang, Yeliang Zhang, and Salim Hariri. Investigating autonomic runtime management strategies for SAMR applications. *International Journal of Parallel Programming*, 33(2):247–259, 2005.

[18] 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.

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

[20] 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.

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

[22] 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.

[23] Karen Devine, Erik Boman, Robert Heaphy, Bruce Hendrickson, and Courtenay Vaughan. Zoltan data management services for parallel dynamic applications. *Computing in Science and Engineering*, 4(2):90–97, 2002.

[24] Tony Dewitt, Thomas Gross, Bruce Lowekamp, Nancy Miller, Peter Steenkiste, Jaspal Subhlok, and Dean Sutherland. Remos: A resource monitoring system for network-aware applications. Technical Report CMU-CS-97-194, Carnegie Mellon School of Computer Science, 1997.

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

[26] Roland Ewald, Jan Himmelspach, and Adelinde M. Uhrmacher. A non-fragmenting partitioning algorithm for hierarchical models. In *WSC '06: Proceedings of the 38th conference on Winter simulation*, pages 848–855, 2006.

[27] Roland Ewald, Jan Himmelspach, and Adelinde M. Uhrmacher. An algorithm selection approach for simulation systems. In *PADS '08: Proceedings of the 22nd Workshop on Principles of Advanced and Distributed Simulation*, pages 91–98, 2008.

[28] Brian T.N. Gunney, Andrew M. Wissink, and David A. Hysom. Parallel clustering algorithms for structured AMR. *Journal of Parallel and Distributed Computing*, 66(11):1419 – 1430, 2006.

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

[30] 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.

[31] Jan Himmelspach and Adelinde M. Uhrmacher. Plug'n simulate. In *ANSS '07: Proceedings of the 40th Annual Simulation Symposium*, pages 137–143, 2007.

[32] Elias N. Houstis et al. Pythia-II: A knowledge/database system for managing performance data and recommending scientific software. *ACM Transactions on Mathematical Software*, 26(2):227–253, 2000.

[33] Henrik Johansson and Johan Steensland. A characterization of a hybrid and dynamic partitioner for SAMR applications. In *Proceedings of the 16th IASTED International Conference on Parallel and Distributed Computing and Systems*, 2004.

[34] 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/.

[35] 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.

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

[37] 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.

[38] 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.

[39] Greg Linden, Brent Smith, and Jeremy York. Amazon.com recommendations: Item-to-item collaborative filtering. *IEEE Internet Computing*, 7(1):76–80, 2003.

[40] 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.

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

[42] L. McInnes, J. Ray, R. Armstrong, T. Dahlgren, A. Malony, B. Norris, S. Shende, J. Kenny, and J. Steensland. Computational quality of service for scientific CCA applications: Composition, substitution, and reconfiguration. Technical Report ANL/MCS-P1326-0206, Argonne National Laboratory, Feb 2006.

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

[44] 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.

[45] Boyana Norris, Jaideep Ray, Rob Armstrong, Lois C. McInnes, David E. Bernholdt, Wael R. Elwasif, Allen D. Malony, and Sameer Shende. Computational quality of service for scientific components. In *Proceedings of the International Symposium on Component-Based Software Engineering (CBSE7)*, volume 3054 of *Lecture Notes in Computer Science*, pages 264–271, 2004.

[46] R. R. Nourgaliev and T. G. Theofanous. High-fidelity interface tracking in compressible flows: Unlimited anchored adaptive level set. *Journal of Computational Physics*, 224(2):836–866, 2007.

[47] Leonid Oliker and Rupak Biswas. Plum: Parallel load balancing for adaptive unstructured meshes. *Journal of Parallel and Distributed Computing*, 47:109–124, 1997.

[48] 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.

[49] 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.

[50] 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.

[51] Manish Parashar, James C. Browne, Carter Edwards, and Kenneth Klimkowski. A common data management infrastructure for adaptive algorithms for PDE solutions. In *Supercomputing '97: Proceedings of the 1997 ACM/IEEE conference on Supercomputing (CDROM)*, pages 1–22, 1997.

[52] A. Pothen, H. D. Simon, and L. Wang. Spectral nested dissection. Technical report, NASA Ames Research Center, 1992. RNR-92-003.

[53] Kenneth G. Powell, Philip L. Roe, Timur J. Linde, Tamas I. Gombosi, and Darren L. De Zeeuw. A solution-adaptive upwind scheme for ideal magnetohydrodynamics. *Journal of Computational Physics*, 154(2):284–309, 1999.

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

[55] Naren Ramakrishnan and Calvin J. Ribbens. Mining and visualizing recommendation spaces for elliptic PDEs with continous attributes. *ACM Transactions on Mathematical Software*, 26(2):254–273, 2000.

[56] Jarmo Rantakokko. A framework for partitioning structured grids with inhomogeneous workload. *Parallel Algorithms and Applications*, 13:135–151, 1998.

[57] Jarmo Rantakokko. An integrated decomposition and partitioning approach for irregular block-structured applications. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium*, pages 336–347, 2000.

[58] Jarmo Rantakokko. Partitioning strategies for structured multiblock grids. *Parallel Computing*, 26(12):1661–1680, 2000.

[59] Hans Sagan. *Space-filling curves*. Springer Verlag, 1994.

[60] 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.

[61] Kirk Schoegel, George Karypis, and Vipin Kumar. A unified algorithm for load-balancing adaptive scientific simulations. In *Proceedings of Supercomputing*, 2000.

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

[63] 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.

[64] Johan Steensland. The benefits of adaptive partitioning for parallel AMR application. Technical report, Sandia National Laboratories, 2008. SAND2008-4073.

[65] 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.

[66] 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.

[67] James D. Teresco, Jamal Faik, and Joseph E. Flaherty. Resource-aware scientific computation on a heterogeneous cluster. *Computing in Science and Engineering*, 7(2):40–50, 2005.

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

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

[70] Rich Vuduc, James Demmel, and Jeff Bilmes. Statistical models for automatic performance tuning. In *ICCS '01: Proceedings of the International Conference on Computational Sciences-Part I*, pages 117–126, 2001.

[71] 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.

[72] Rich Wolski, Neil T. Spring, and Jim Hayes. The network weather service: a distributed resource performance forecasting service for metacomputing. *Future Generation Computer Systems*, 15(5–6):757–768, 1999.

ACTA
UNIVERSITATIS
UPSALIENSIS
UPPSALA
2009