Design and Implementation of a dynamic and adaptive meta-partitioner for SAMR grid hierarchies

Henrik Johansson

August 20, 2007

1 Introduction

Structured adaptive mesh refinement (SAMR) is used to decrease the run-time of simulations in areas like computational fluid dynamics [3, 9], numerical relativity [8, 13], astrophysics [5, 22], and hydrodynamics [19]. Simulations based on SAMR start with a coarse and uniform grid. The grid is then recursively refined in areas where the accuracy is too low, creating a dynamic grid hierarchy that always conforms to the maximum acceptable error.

The dynamic resource allocation makes it necessary to repeatedly repartition and redistribute the grid hierarchy over the participating processors. For efficient use of SAMR on parallel computers, the partitioning process must not only take the computations and the CPU performance into account, but also all other factors that contribute to the run-time: communication volume, synchronization delays, data movement between partitions and the performance and utilization of the interconnect. Thus, to minimize the run-time, the current state of the application and the hardware must both be taken into account. This is non-trivial. The basic conditions for how to allocate hardware resources change dramatically during run-time, due to the dynamics inherent in both the applications and the computer system.

Previous research has shown that no single partitioning algorithm is the best choice for all conditions [29]. Instead, good-performing partitioning algorithms need to be dynamically selected and invoked during run-time. In this work we present the design and implementation of the *meta-partitioner* [14, 15, 31, 34], a framework that autonomously selects, configures, and invokes the best predicted partitioning algorithm with regard to the current application and computer state. To make the meta-partitioner user-friendly and to allow for easier modification and expandability, it is implemented using component-based software engineering (CBSE).

The meta-partitioner first determines a partitioning focus, computed from a set of predictive metrics that estimate the partitioning needs of the application. It then characterizes the physical properties of the grid and matches the grid against stored grid characeristics. The algorithm that has the best performance for the current combination of partitioning focus and the most similar stored grid hierarchy is selected and invoked by the meta-partitioner.

The paper is organized as follows. In Section 2 we describe SAMR and the most common partitioning techniques. The purpose and goal of the meta-partitioner is discussed in Section 3 and an overview of previous work is given in Section 4. A general discussion of the meta-partitioner design and component based software engineering is found in Section 5. The workflow of the meta-partitioner is introduced in Section 6 and the implementation and functionality of the resulting components in found Section 7. We present a number of idea for improvments in Section 8. Finally, Section 9 holds a summary together with our conclustions.

2 Background

In this section we describe structured adaptive mesh refinement (SAMR). We also present the most common algorithms for partitioning and distribution of dynamic grid hierarchies.

2.1 Structured adaptive mesh refinement

For PDE solvers based on finite differences and structured grids, solution accuracy and run-time are dictated by grid resolution. A higher resolution generally results in a higher accuracy but also in a longer run-time. Often, features requiring additional resolution, like shocks and discontinuities, only occupy a small part of the grid: a uniform and high resolution is then a waste of computational resources. By increasing the resolution in critical areas, the run-time of these PDE solvers can be decreased.

The common Berger-Colella SAMR algorithm [3] starts with a coarse structured base grid covering the entire computational domain. The resolution of the base grid conforms to the lowest acceptable accuracy of the solution. At regular intervals, the local computational error is estimated. Grid points with errors larger than a given threshold are flagged for refinement. Flagged points are clustered and overlaid with logically rectangular patches of finer, uniform resolution. For small errors, refined patches can be removed. As the execution progresses, grid patches are created, moved and deleted, resulting in a dynamic grid hierarchy.

During execution, information are frequently exchanged between grid patches. Boundary data for a refined grid patch is typically obtained from adjacent patches or patches on the next lower level, as most patches are contained in the interior of the computational domain. After integration, the results are projected down from finer to coarser levels. As refined patches use smaller time steps, updating coarser level solutions increase the accuracy. Thus, data flows both along neighboring patches and between patches on different refinement levels.

2.2 Partitioning grid hierarchies

Efficient use of parallel SAMR typically requires that the dynamic grid hierarchy is repeatedly partitioned and distributed over the participating processors. Several performance issues arise during the partitioning process. As information

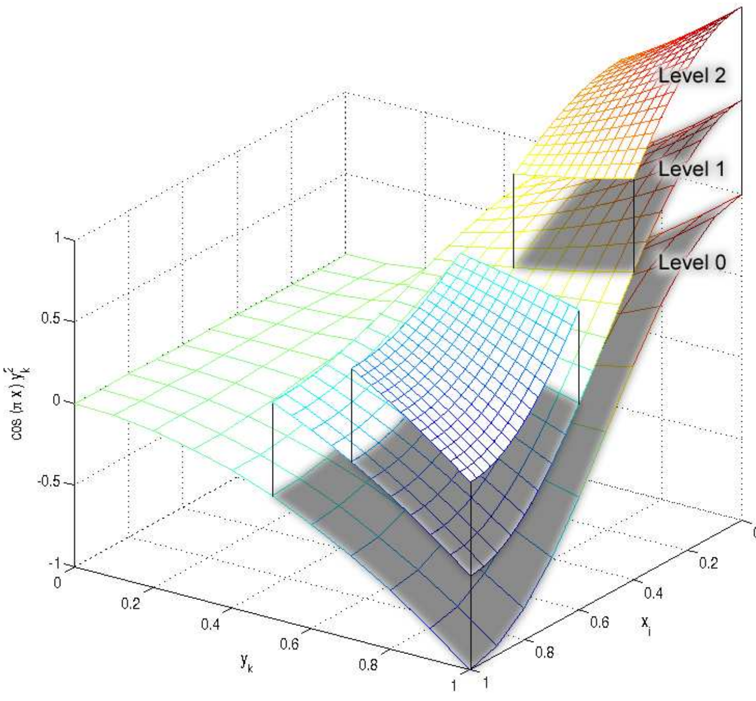


Figure 1: Example of a grid hierarchy with two levels of refinement. The grids are skewed to reflect the characteristics of a solution.

flows in the grid hierarchy, processors need to exchange data. Intra-level communication appears as grid patches are split between processors and data are exchanged along the borders. Inter-level communication can occur for overlaid patches when the solution is projected down to coarser levels and when a finer patch lacks boundary data. Both types of communication can severely inhibit parallel efficency.

A synchronization delay may occur when a processor is busy computing, while holding data needed by other processors. Until the processor has finished its computations, other processors might be unable to proceed as they lack data. Synchronization delays can be severe — the time spent waiting for data can be of the same magnitude as the actual computational time [33, ?]. The number of delays often grows as the number of processors is increased. To predict the impact of the delays, complex and time-consuming execution models are needed.

To get optimal performance, the partitioner needs to simultaneously minimize all performance inhibiting factors; data migration, load imbalance, communication volumes, and synchronization delays. Typically, it is unrealistic to search for the optimal solution [11]. Instead, the partitioner needs to trade-off the metrics in accordance with the characteristics of the application and computer. Ultimately, partition quality is determined by the resulting application execution time.

Algorithms for partitioning SAMR hierarchies can be categorized as domain-based, patch-based, or hybrid. For *patch-based partitioners* [2, 18, 27], the distribution decisions are made independently for each refinement level (or patch). The SAMR frameworks SAMRAI [37, 38] and Chombo [7] supports patch-based partitioning. *Domain-based partitioners* [24, 26, 31, 36] partition the physical domain, rather than the grids themselves. The domain is partitioned along with all contained grids on all refinement levels. Domain-based methods can be found in the AMROC [1, 9] and GrACE [25] frameworks. *Hybrid partitioners* [17, 24, 36] combine the patch-based and domain-based approaches.

2.2.1 The patch-based approach

For the patch-based approach, the most straightforward method is to divide each patch or level into p blocks, where p is the number of processors, and distribute one block to each processor. Another method is to use a bin-packing or greedy algorithm [2, 25, 38] to distribute the patches. For the partitioning to be effective, large patches may have to be divided. Regardless of the specific method, the partitioner can consider either patch-by-patch or level-by-level.

In theory, the patch-based approach results in perfect load balance. In practice, some load imbalance can be expected due to sub-optimal patch aspect ratios, integer divisions and constraints on the patch size. Partitioning can be performed incrementally, as only patches created or altered since the previous time step need to be considered for re-partitioning. However, patch-based algorithms often result in high communication volumes and communication bottlenecks. The communication volume is generally increased when a patch is subdivided into blocks to create a lower load imbalance. Communication serialization bottlenecks can occur when overlaid patches are assigned to different processors. A coarser block is typically assigned to fewer processors than a finer block. A processor owning coarser blocks will generally need to communicate with many processors having finer and overlaid blocks, creating communication bottlenecks.

2.2.2 The domain-based approach

For domain-based algorithms, only the base grid is partitioned. Initially, the workload of the refined patches are projected down onto the base grid, reducing the problem to partitioning a single grid with heterogenous workload. The minimum block size is determined by the size of the computational stencil on the base grid. As the base grid stencil corresponds to many grid points on highly refined patches, the workload of a minimum sized block can be large.

As overlaid grid blocks reside on the same processor for domain-based algorithms, inter-level communication is eliminated. A complete re-partition might be necessary when the grid hierarchy is modified. Because the computational stencil and base grid resolution impose restrictions on subdivisions of higher level patches, the load imbalance is often high for deep grid hierarchies. Furthermore, syncronization bottlenecks are common as the division of a refinement level generally results in parts with widely differing workloads. Another problem with domain-based algorithms is ”bad cuts”: many and small blocks with bad aspect ratios. These blocks occur when patches are cut in bad places, assigning only a tiny fraction of a patch to one processor while the majority of it resides on another processor.

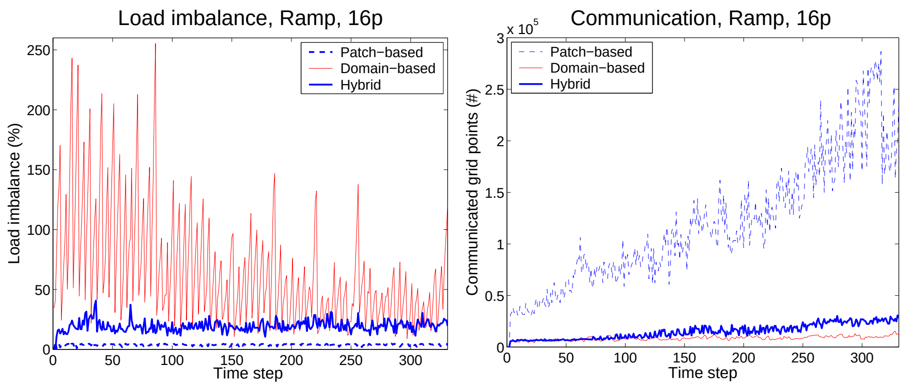
2.2.3 A hybrid approach

Both patch-based and domain-based algorithms perform well under suitable conditions, especially for simple and shallow grid hierarchies [28, 34]. Unfortunately, their shortcomings often make their performance unacceptable for deep and complex hierarchies [28, 29]. As a remedy, a hybrid approach can be used. By combining strategies from both the domain-based and the patch-based approach, it is possible to design a partitioner that performs well under a wider range of conditions.

To illustrate the hybrid approach, we describe a partitioning framework that is used by the meta-partitioner — Nature+Fable. Key concepts in Nature+Fable are separation of refined and unrefined areas of the grid and clustering of refinement levels [29]. Separation of unrefined and refined areas enables different partitioning approaches to be applied to structurally different parts of the grid hierarchy. Refinement levels are clustered into bi-levels. A bi-level consists of all patches from two adjacent levels — patches from refinement level k and the next finer level, $k+1$. If the coarser level is much larger than the finer level, the non-overlaid area of the coarser level can be removed from a bi-level. Each bi-level is partitioned with domain-based methods. Patch-based methods are used for all parts of the grid that are not included in bi-levels.

To perform well under a wide range of conditions, the partitioning process in Nature+Fable is governed by a large set of parameters. Each parameter setting corresponds to a separate partitioning algorithm.

The hybrid partitioning algorithms arising from Nature+Fable can achieve a lower load imbalance than domain-based algorithms since patches from at most two refinement levels are partitioned together. Because inter-level communication only exist between bi-levels, communication volumes are generally smaller for the hybrid algorithms than for patch-based algorithms.



(a) Load imbalance (b) Communication

Figure 2: Load imbalance and communication for the most common partitioning approaches. Note that the algorithms complement each other. The values for the domain-based and the patch-based methods are from a single partitioning algorithm, while the hybrid values are selected from over 800 algorithms [16]. The example application, Ramp, is taken from the Virtual Test Facility [35] and it was partitioned for 16 processors.

3 The need for a meta-partitioner

During the execution of a parallel SAMR application, the grid is generally repeatedly repartitioned. If we only use a single algorithm for the repartitioning, we will construct bad performing partitions for many application and computer states [14, 29]. To consistently construct good-performing partitions, a number of important issues must be addressed.

Because of the huge range of possible application and computer states, we need to have access to complementing partitioning algorithms that together perform well for all of these states. Furthermore, we must also have the capability to select a good-performing partitioning algorithm at each repartitioning.

The three types of complementing partitioning approaches (described in Section 2.2) have vastly different characteristics (see Figure 2). Using a patch-based algorithm, we can achieve a low load imbalance but at the cost of a large amount of communication. A domain-based algorithm generally results in a low amount of communication, but at the expense of a high load imbalance. The hybrid algorithms falls somewhere in between these two extremes — striking a balance between load imbalance and communications. To construct good-performing partitions for the full spectrum of possible application states, a partitioning tool needs to have access to algorithms from all three of the partitioning approaches.

For computationally intensive applications, we would prefer to give preference to algorithms that generally result in a low load imbalance. If the computer has a slow interconnect, algorithms that typically produce a low amount of communication are more attractive. Thus, before the start of a simulation, we must also consider basic and static characteristics of both the application and the computer system. These characteristics include the amount of computations to update a grid point, the storage need of a grid point, and the computation and communication capacity of the the computer system.

The current application state has a large influence on the partitioning outcome. A certain grid hierarchy can be more or less suitable for the available partitioning algorithms. One algorithm might perform well for deep grid hierarchies that consist of few grid patches, while another algorithm will achieve good results for a low number of refinement levels and many grid patches.

Finally, the current state of the computer system should be considered. If some part of the computer is overloaded or congested, the partitioning algorithm should be selected to decrease the negative performance issues arising from that component.

To address the issues described above, we have proposed the development of the meta-partitioner [14, 15, 31, 34]. The meta-partitioner autonomously selects, configures, and invokes an appropriate partitioning algorithm with regard to the current application and computer state. In the following sections, we describe the design and implementation of the meta-partitioner.

4 Initial work

The huge range of possible application states makes it necessary for the meta-partitioner to have access to a large number of complementing partitioning algorithms. To our knowledge, the only partitioner with an inherent capability of adapting to a multitude of partitioning conditions is Nature+Fable. We thus use Nature+Fable as the basis of the meta-partitioner. However, to cover the full partitioning spectrum, the meta-partitioner will also be complemented with both patch-based and domain-based partitioners.

In an early evalutation of Nature+Fable [15], we manually tried to find rules that can be used to select the partitioning algorithm. For the evaluation, we used unpartitioned trace files containing complete grid hierarchies from four SAMR applications. After Nature+Fable had partitioned the grid hierarchies, the partitioned trace files were used as input to a SAMR simulator [6]. The simulator mimics the behavior of the common Berger-Colella SAMR algorithm [3] and it computes important performance metrics like load imbalance and communication volumes. For the evaluation, we selected the four parameters that we believed had the largest impact on the partitioning outcome. For these parameters, we performed single factor experiments [] where one parameter at a time was varied. In total, eleven hybrid partitioning algorithms were evaluated. Because of parameter inter-dependencies and the relatively small number of partitioning algorithms, no rules were found.

Next, a larger performance charactierization of Nature+Fable was performed. Four advanced application from the Virtual Test Facility [9, **?**] were partitioned by approximately 800 hybrid algorithms. For each application, 8, 16, and 32 processor configurations were used. The large amount of performance data were stored in a data base. The partitions were evaluated using an expanded version of the simulator with the capability of also presenting an approximation of the synchonization penalty [30] and per-processor communication volumes. The resulting per-time step analysis of the performance can be regarded as a proof-of-concept for the need and viability of dynamic algorithm selection.

The algorithm performance data base that resulted from the larger characterization is a necessary foundation for the implementation of the MP. Without access to its comprehensive performance data, it is impossible to implement any viable method to choose good-performing algorithms.

5 Design considerations

Before implementing the meta-partitioner, several general and far-reaching design decisions must be made. These decisions are of great importance for the future use and acceptance of the meta-partitioner.

The average scientific application is steadily growing, both in size and complexity. Today, advanced computational problems often requires cooperation between several research groups, each responsible for a part of the problem. The research groups do not only need to focus on their own part, they also need to spend increasingly more time to make the different application parts work together. Thus, the meta-partitioner must be designed and implemented with careful consideration of its interactions with both its users and the SAMR software. A meta-partitioner that is restricted to a certain piece of software or adds a lot of complexity will not be used, regardless of the effectiveness of the selected algorithms.

Implementing the meta-partitioner as a stand-alone application can impose difficulties for the user. If the meta-partitioner is not tailored for the SAMR engine preferred by the user, incorporating the meta-partitioner in a simulation might need a lot of work or even be impossible. On the other hand, adapting the meta-partitioner to specific SAMR frameworks could be equally dangerous. Scientists might migrate to other SAMR engines, but migrating the meta-partitioner to a new framework would require a lot of work. Also, if the SAMR engine is modified, the meta-partitioner might cease to function properly.

A remedy to these problems is to use component-based software engineering (CBSE) to enable inter-operability between components developed independently by different research groups. To construct and execute a CBSE application, components (i.e. the meta-partitioner and a SAMR engine) are connected through well-defined interfaces to form a single executable entity. Using the component-based approach, the meta-partitioner can seamlessly be used and connected to any available SAMR engine that conforms to the used CBSE specification.

5.1 The Common Component Architecture

We use the the Common Component Architecture (CCA) [4] for the implementation of the meta-partitioner. CCA is a community based CBSE initiative, specifically targeted at the needs of parallel scientific high-performance computing. CCA presents a general, low-latency model for component inter-operability and interaction.

A component is a basic unit of software functionality. Together, components form an application. The components interact trough abstract interfaces called ports that give access to the functionality of a component. A component can provide a port, meaning that it implements the functionality expressed by the port. The component can also use ports, meaning that it make calls through the port to access the functionality provided by another component. A framework manages and assembles the components and ports into applications. The framework is also responsible for the execution of the application.

The components can be written in a number of languages and they can use different parallel programming models. Existing software can be turned into CCA components by adding a simple wrapper and a standard port. During the execution of an application, it is possible to add, remove and change components. Simulations performed in fields like climate modeling, accelerator modeling, and combustion have shown that CCA can significantly ease the development of advanced scientific applications [21] without any negative impact on performance.

In CCA, significant attention is given to computational quality of service (CQoS) [20, 23]. CQoS is the ability of a system to ensure that a scientific problem is solved with the best available hardware and software resources. This is *the* core of the meta-partitioner.

Using CCA, it is easy to use and incorporate the meta-partitioner into various SAMR engines. The SAMR engines Chombo [7] and GrACE [25] are already modified to conform to the CCA specifications.

6 The meta-partitioner workflow

To identify suitable components for the meta-partitioner, we need to consider both its internal functionality and its external interactions with both SAMR engines and computer systems. The design must also allow for easy expansion and modification when new and better tools and partitioning algorithms become available.

The components need to have carefully designed interfaces that permit internal modification without any changes to their external functionality. If we later have to modify an interface, we might need to perform cascading and time-consuming changes to many other components as well.

Below we present the workflow for the meta-partitioner (see Figure 3). The workflow is divided into separate tasks that are later transformed into components (presented in Section 7).

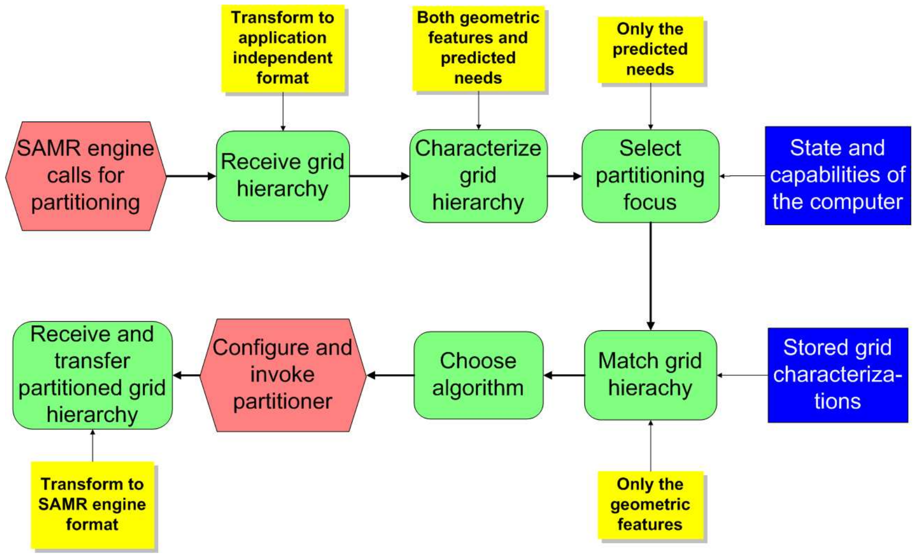

Transform to application independent format
Both geometric features and predicted needs
Only the predicted needs
SAMR engine calls for partitioning
Receive grid hierarchy
Characterize grid hierarchy
Select partitioning focus
State and capabilities of the computer
Receive and transfer partitioned grid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

Figure 3: The MP Workflow. The Hexagons represent operations performed outside the meta-partitioner. The (green) boxes with smooth corners are tasks performed by the MP. The dark grey (blue) boxes holds input data while light grey (yellow) boxes are comments

6.1 Receive the grid hierarchy

When the SAMR engine calls for re-partitioning, the current grid hierarchy is transfered to the meta-partitioner. SAMR engines can use different formats to describe the grid hierarchy. To avoid being restricted to specific SAMR engines, the grid hierarchy have to be translated into an internal grid representation.

6.2 Characterize the grid hierarchy

To select good-performing partitioning algorithms, the current state of the grid hierarchy must be throughly and accurate characterized. For this task, we compute two sets of metrics from the grid hierarchy.

First, we compute a set of predictive metrics that captures the general partitioning needs of the grid hierarchy. These metrics address issues like "is this application state likely to result in a high load imbalance?", "is the communication volume a major issue?", and "is the data migration expected to be large?". The metrics are developed by Steensland and Ray [32, ?].

Next, we use a set of metrics that are computed from the geometric features of the grid. This set of metrics captures the physical properties of the grid, i.e. "how much of the grid is refined?", "how large are the grid patches?", and "how many grid patches are present?" To allow for comparisions, all of the geometric metrics are designed to be application independent.

Together, these two sets of metrics give an accurate picture of both the physical properties and the partitioning needs of the current grid hierarchy.

6.3 Selecting a partitioning focus

Generally, no partitioning algorithm can simultaneously minimize all performance inhibiting factors (see Section 2.2 and Figure 2). Instead, the partitioning effort is focused on the single factor that is expected to have the greatest impact on the execution time.

We focus the partitioning effort on either the load balance or the synchronization delays. We do not initially consider the amount of communication or the data migration. The time needed to communicate boundary data is generally insignificant compared to the time spent on synchronization delays [35, ?]. For the data migration, we currently lack necessary performance data for the algorithm selection step. If the need arise, we can expand the focus to include both data migration and communication. To simplify both the determination and the use of the focus, it is into a number of discreet levels.

We select the inital partitioning focus by analyzing static application and computer characteristics — the computational requirements of the application and the capabilities of the computer system. During run-time, the focus is continuoously modified using a combination of the predictive characterization metrics (see Section 6.2) and the current state of the computer system. To decrease excessive focus oscillations, we use the previous focus as a starting point and limit the maximum change in focus. To simplify the algorithm selection (see Section 6.5), we restrict the focus to a discreet number of levels.

6.4 Matching the current grid hierachy

In every domain where non-random decisions are made, the decisions are ultimately based on some kind of data or knowledge. To select a partitioning algorithm, the large number of available algorithms makes it necessary to use performance data from previously encountered application states. This is true regardless of the method that we use to select the algorithm (i.e. rules, neural networks, and exhaustive searches).

For most selection methods, it is necessary to match the current application state against stored application states for which we have collected performance data. We use the stored application state that is most similar to the current state as input for the selection method.

To match application states against each other, we use the geometric characterization metrics (see Section 6.2). This set of metrics describe the physical properties of the grid. When grid hierarchies with resembling geometrical properties are partitioned by the same algorithm, we expect that the resulting partitions will have similar properties.

6.5 Selecting a partitioning algorithm

To select the partitioning algorithm, we use the stored application state that is most similiar to the current application state. For this stored state and the current partitioning focus, we select the algorithm that resulted in the best performance.

First, we evaluate all algorithms for the stored grid hierarchy with regard to the most performance inhibiting factor (e.g. the partitioning focus). We consider all good-performing algorithms as candidate algorithms. From these candidate algorithms, we select the algorithm that has the best performance with respect to the other performance inhibiting factor (e.g. load imbalance if we focus on synchronization, and synchronization if the focus is on load imbalance).

Using this method to select the algorithm, we will control the most performance inhibiting factor while the impact of the other factor is kept as low as possible. We deliberatly avoid to select the best algorithm with regard to only the most performance inhibiting factor, as this algorithm often performs badly for all other factors. This selection method is easy to expand if we extend the partitioning focus to also include communication and data migration.

The algorithm selection is only dependent on the partitioning focus and the stored application state. Because the focus is divided into discreet levels, we can precompute the algorithm selection for all combinations of focus and application states. During run-time, the algorithm selection is thus reduced to fast and simple table look up. We call each combination of focus and application for a *rule*.

6.6 Invoke and configure the partitioner

The selected partitioner (e.g. the partitioning algorithm) is configured and invoked. If the available partitioners use different formats to describe the grid hierarchy, a translation from the internal grid representation might be needed.

6.7 Transfer the partitioned grid hierarchy to the SAMR engine

The partitioned grid hierarchy is translated from the internal representation used by the MP into the format used by the SAMR engine. The grid hierarchy is then transfered to the SAMR engine, which resumes computing.

7 CCA components for the meta-partitioner

To use the full potential of CCA, we analyze the workflow (see Section 6) of meta-partitioner and divide it into a number of components. Separating the functionality of the meta-partitioner into component allows for easy expansion and modification, without any loss of efficiency.

It is important that the design of the components allows for all desired features, even if these features are not implemented in the initial version of the meta-partitioner. Below, we describe each of the proposed components. The components, and their ports, are shown in Figure 4.

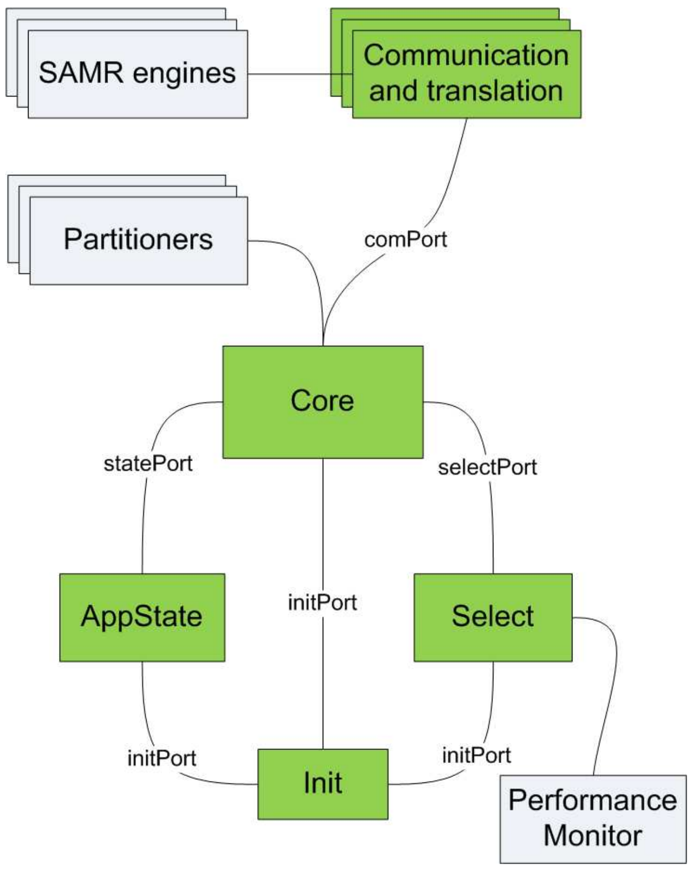


Figure 4: The CCA components. The dark (green) components are included in the meta-partitioner. The light (blue) components are developed elsewhere, but used in the meta-partitoner.

7.1 The core component (Core)

The `Core` component acts as a hub for the meta-partitioner — it is connected to all other components. The `Core` controls the execution of both the meta-partitioner and the SAMR-framework. It supplies data to the components and receives the resulting output. The implementation of the meta-partitioner does not require a `Core`-like component per se. However, it can be significantly harder to expand and modify the meta-partitioner if the separate components communicate directly with each other.

In principle, the task of the `Core` is to perform a number of function calls and to handle all necessary data transfers. The pseudocode of the `Core` is lsited in Algorithm 1. The parenthesises contain the name of component responsible for that particular task.

7.2 Communication and translation components (CoT)

The available SAMR engines can use different formats to describe the grid hierarchy. To implement an efficient and expandable meta-partitioner without

Algorithm 1 The Core component

```
CollectStaticData(Init)
ReceiveGridHierarchy(CoT)
CharaterizeHierarchy(AppState)
SelectFocus(Select)
MatchHierarchies(Select)
SelectAlgorithm(Select)
Partition(Partitioners)
ReturnGridHierarchy(CoT)
```

being restricted to a particular SAMR engine, it is necessary to translate the different grid representations into a single internal format.

As the internal format for the meta-partitioner, we use the representation developed for the DAGH/GrACE frameworks [25]. The format is listed below. For 2D application, the z-coordinate is set to -1. The `num`-entry is a unique id for each grid patch. The field `owner` corresponds to the id of the processor that is assigned to a patch.

```
Time-step owner level num size(x,y,z) start(x,y,x) stop(x,y,z)}
```

After the grid hierarchy is partitioned, the `CoT` component translates the hierachy back into the format used by the SAMR engine. The component also returns the partitioned grid hierarchy to the SAMR engine. A separate `CoT` component must generally be implemented for each SAMR engine. A similar approach is used in the load balancing tool-kit Zoltan, where callback functions translate and provide necessary mesh data [10].

To evaluate the MP without performing real simulations, we can use un-partitioned trace files derived from actual SAMR applications (similar to the evaluation described in Section 4). For these cases, the `CoT` component reads the trace file and transfers the contained grid hierarchy to the `Core` component. The partitioning functionality of the MP is independent of the source of the grid hierarchy.

All partitioners currently used by the meta-partitioner inherently support the DAGH/GrACE grid representation. If the MP incorporates partitioners that use other grid representations, the `CoT` component will be expanded to translate these hierachies to the DAGH/GrACE format.

7.3 Application state component (AppState)

Both the construction of rules for the algorithm selection and the matching of application states require that the current application state is accurately characterized. This task is performed by the`Appstate` component.

For the characterization of the grid hierarchy, the `AppState` uses two separate sets of metrics. The first set predicts the partitioning needs of the grid hierarchy. The second set characterizes the physical properties of the grid.

Predictive classification space (PCS)

To predict the partitioning needs, three predictive metrics are used to judge the...

In previous publications the PCS was known as the PCCS (Partitioner Centric Classification Space).

Because of the high complexity involved in predicting the impact of synchronization delays, it is difficult to develop an accurate synchronization metric. However, a synchronization delay can *only* occur when two processors exchange data. As a consequence, there exist a relation between synchronization delays and the amount of communication. Even though this relation is dependent on the properties of the grid hierachy, we can use the predictive communication metric to estimate the impact of synchronization delays.

The geometric classification space (GCS)

The geometric classification space (GCS) captures the geometrical properties of the grid hierarchy. These metrics can be seen as a snapshot of the physical features of the current grid hierarchy.

For each geometric metric, a separate value is computed both for each refinement level and for all levels together as an aggregate. To be comparable, all metrics are normalized with respect to either the size of the grid hierarchy or a refinement level.

Below, we present the metrics that are currently included in the GCS. Depeding on future performance evaluations, metrics can be added or removed.

Number of refinement levels The number of refinement levels is important for the matching of application states. Performance predictions for applications with different number of refinement levels are less accurate than for applications having the same number of refinement levels.

Amount of refined area This metric computes the fraction of refined area for each refinement level, normalized with the area of the base grid.

Amount of refined area with regard to next lower level This metric computes the ratio of the area of a refinement level, compared to the refined area on the next lower level. A value close to one means that we probably have sharp features in the grid hierarchy, as the size of the two refinement levels are almost equal. If the value is small, it indicates a more fuzzy refinement pattern.

Number of patches per area unit The number of patches, normalized with the area of the base grid. A large number of patches can indicate a refinement pattern that has a circular shape, making it necessesary to use many small patches to cover the curvature.

Average grid patch area with respect to refinement level area The average patch size, normalized with the area of the current refinement level.

Standard deviation of grid patch area The variation in patch size.

Average grid patch aspect ratio The aspect ratio is related to the circumference of the patches. Large aspect ratios translate into larger circumferences and possibly more communication and synchronization delays.

Note that the term “area” should be replaced with “volume” for the 3D versions of the metrics.

Normalization

Because the magnitude of the different PCS and GCS metrics can vary by several orders, it is necessary to normalize their values to a common interval. The meta-paritioner uses *z-score normalization* [12], which normalizes each metric based on the mean and the standard deviation. A value v of a metric A is normalized to v' (in the range $[0.0, 1.0]$) by computing

$$v' = \frac{v - \tilde{A}}{\sigma_A}$$

where $\tilde{A}$ and σ_A are the mean and standard deviation of metric A. The mean and standard deviation are computed from the stored grid hierachies and supplied by the `Init` component.

The z-score normalization is useful when the actual minimum and maximum values of the metric A is unknown, which makes it easier to later expand the rules. Also, z-score normalization is less sensitive to outliers than many other normalization methods [12].

7.4 Algorithm selection component (Select)

The algorithm selection component is responsible for a multitude of tasks associated with the process of selecting good-performing partitioning algorithms; the component determines the partitoning focus, it matches the current grid hierarchy against stored grid hierarchies, and it selects the partitioning algorithm that had the best performance for the most similar grid hierarchy.

The first task for the `Select` component is to modify the partitioning focus. The focus is divided into eight discreet levels, each corresponding to a rule that shares the same name (har vi skrivit det tidigare?) (see Table 1). Initially, we restrict the focus to load imbalance and synchronization delays. If necessary, the focus can later be expanded to include both communication and data migration.

Focus	**Corresponding rule**
FocusSynch 1.25	MaxSynch 125% of minSynch
FocusSynch 2	MaxSynch 200% of minSynch
FocusSynch 3	MaxSynch 300% of minSynch
FocusSynch 5	MaxSynch 500% of minSynch
FocusLB 1.2	MaxLB 120% of minLB
FocusLB 1.5	MaxLB 150% of minLB
FocusLB 2	MaxLB 200% of minLB
FocusLB 3	MaxLB 300% of minLB

Table 1: The different partitioning focuses. Note the differences between FocusSynch and FocusLB. This is due to larger variations in the performance results for the synchronization penalty compared to the load imbalance

When the partitioning focus is modified, the previous focus is used as a starting point. Depending on the relative differences between the current and the previous PCS, the partitioning focus is either kept or shifted a few steps in any direction. By using the previous focus as starting point and restriciting the maximum the rate of change, we can avoid a widly oscillating focus. Large and frequent changes to the focus can result in widely different partitions and hence large amounts of data migration and poor cache memory performance.

The current implementation of the MP does not support modification of focus during run-time. Instead, once the initial focus has been determined, it remains fixed throughout the execution.

After selecting the partitioning focus, the `Select` component matches the current grid hierarchy against the stored hierarchies. For the matching, we use the geometric metrics (computed by the `AppState` component) and the least square method []. The stored grid hierarchy that is most similar to the current hierarchy is recorded.

Generally, the workload for a SAMR application is concentrated to the highest refinement levels. Thus, an imbalance on a high refinement level is likely to have a larger impact on the execution time than an imbalance on a lower level. To giver higher precendence to levels with large workloads, the least square sum for each refinement level is assigned a weight according to the relative workload of that level. The aggregate metrics are given a weight corresponding to 25% of the total workload. This method ensures that the physical properties of levels with high workloads are given a higher priority.

During the matching step, we prefer to only compare the current hierarchy against stored hierachies that have the same number of refinement levels. We only use grid hierarchies with a differing number of refinement levels when we lack performance data for hierarchies that have the same number of refinement levels.

We assume that grid hierarchies that have similar physical properties generally also have resembling partitioning needs. During the large algorithm characterization (described in Section **??**), we stored the performance data for all combinations of application states and partitioning algorithms in a data base. For the stored grid hierarchy that is most similar to the current grid hierarchy, we use this performance data to select the best performing partitioning algorithm with regard to the current rule (see Section 6.5). The rules are listed and described in Table 1. The difference in the synchronization penalty and the load imbalance is due to a larger variation in the performance data for the synchronization penalty. Because it is possible to precompute the selection rules (see Section 6.5), we only have to do a simple table look up for the appropriate combination of application state and partitioning focus. The selected partitioning algorithm is then transfered to the `Core` component.

7.5 Initialization component (Init)

The `Init` component contains a number of important utility functions that are used both before the start of the simulation and at each repartitioning step. The functionality included in the `Init` component can generally be performed by other components. However, to keep the components as simple as possible, we have moved many of the utility-type tasks to the `Init` component.

To select an accurate partitioning focus, we need to determine static characteristics for both the computer system and the SAMR application, i.e. computational capacity and communication needs (see Sections 6.3 and **??**). This information is fixed during run-time and it is collected before the start of the execution. In the initial implementation of the MP, this information is determined before compile time. In future versions of the MP, the user will have the ablilty to supply this information before the start of the simulation. To get the highest possible accuracy, the data collection can be automated by performing several short test runs. Once determined, the characteristic data is transfered to the `Core` component.

During the grid characterization step (performed by the `Appstate` component), both the predicitive and the geometrical metrics are normalized to allow for comparisions of different grid hierarchies (see Section 7.3). For the normalization process, the mean and standard deviation from all stored hierarchies are needed. The `Init` component provides this data.

Before the `Select` component matches current grid hierarchy against stored hierachies, the `Init` component reads all stored hierarchies from file and transfers them to the `Select` component.

After the current grid hierachy has been matched against the stored hierarchies, the Init component reads the current (precomputed) rule from file and transfer it to the `Select` component.

7.6 External components

Some of the necessary functionality for the meta-partitioner are provided by third party software. Below, we briefly describe these components.

7.6.1 SAMR engines

The meta-partitioner should naturally be connected to one or more SAMR engines. To turn a SAMR engine into a CCA-component, a simple wrapper routine must be added. If the SAMR engine cannot dump and export its current grid hierarchy, such a capability must also be added. Instead of calling an internal partitioning algorithm, the SAMR should transfer the grid hierarchy to the MP and `CoT`-component. These small modifications should be easy to implement.

7.6.2 Partitioning algorithm components

In earlier work, we have shown that no single partitioning algorithm is the best choice for all application and computer states [29]. Thus, access to a collection of complementing algorithms is of great importance for the meta-partitioner. In the initial implementation of the meta-partitioner, we include patch-based hybrid partitioning algorithms (see Section 2.2). Unfortunately, we currently do not have access to a stand-alone domain-based algorithm.

We incorporate each partitioning algorithm (or partitioning framework) as an individual component. This approach makes it easy to uppdate and to add new algorithms.

7.6.3 Performance Monitoring Components

The performance of a partitioning algorithm can be significantly influenced by both the computational load of the computer system and the utilization of the interconnect. Thus, the meta-partitioner should use current performance data from the computer system and adapt the partitioning focus to the data. The basis for this task is the static characterisitc data collected before the start of the simulation by the `Init` component.

During run-time, we plan to use existing performance measurement tools, like the Network Weather Service (NWS) [39], to measure the load of the computer. NWS monitors and dynamically predicts the performance of various network and computational resources. NWS gathers readings of the instantaneous performance conditions and uses numerical models to generate forecasts of what the conditions will be for a given time frame. Currently, the system has sensors for end-to-end TCP/IP performance (bandwidth and latency), available CPU percentage, and available non-paged memory.

By comparing the current load with the static capabilities of the computer, we can modifiy the focus to decrease the impact of any overload component in the computer system.

7.7 CCA ports

The different components of the meta-partitioner communicate through a number of ports, as can be seen in Figure 4. Every component, with execption of the `Core`, have a *provides* port that gives access to the components functionality. A component only has one *provides* port, even if its functionality are used by several other components (like the `Init`-component). To access the functionality of another component, the component need to have an *uses* port. We do not describe the ports, since the description would be a repetition of the functionality of each components.

8 Future Work

In this report we have described our inital implementation of the meta-partitioner. To improve the partitioning performance, we can envision a number of modifications to the meta-partitioner.

Only hybrid- and patch-based algorithms have been incorporated into the meta-partitioner. To our knowledge, there does currently not exist any stand-alone domain-based partitioner. For certain application states, access to a domain-based partitioner could significantly improve the partitioning quality.

Currently, the partitioning focus remains fixed throughout the simulation. Allowing for modification of the focus, as described in Section 6.3 and 7.4, would tailor the partitioning effort to the needs of the current application. Also, the partitioning focus can be expanded to include the communication and data migration.

To determine an accurate partitioning focus, the current state of the computer system should be considered. When parts of the computer are overloaded, a larger effort need to made to decrease the impact of the overload. For this task, we need to incorporate an external performance measurement tool.

In the initial implementation, all individual geometric metrics are given equal weights. This is an unrealistic assumption, as some of the metrics probably have greater impact on the matching of the application states. For an accurate matching, we need to determine the weight for each geometric metric.

(Adapting SAMR-engines to the MP is not a part of Future Work, as this task is separate from the MP... Thus, nothing of SAMR engines are written in Future work)

9 Summary and conclusions

In this paper we presented the design and implementation of the meta-partitioner. The meta-partitioner is a partitioning framework for parallel SAMR applications that autonomously selects, configures, and invokes good-performing partitioning algorithms with regard to the current application and computer state.

To make the meta-partitioner user-friendly and to allow for easy modification and expandability, it is implemented using component-based software engineering. The meta-partitioner uses the Common Component Architecture specification and its implementation consists of several components. Complementing partitioning algorithms from two the three main approaches (patch-based and hybrid) are incorporated into the meta-partitioner.

To select good-performing partitioning algorithms, the MP determines a partitioning focus based on the predicted partitioning needs of the application. We characterize the current state of the application with a set of geometrical metrics that captures the physical properties of the grid hierarchy. The current application state is matched against a large number of stored states and the most similar stored state is recorded. Complete algorithm performance data for this stored application state is evaluated. The best partitioning algorithm with regard to the current partitioning foucs is selected and invoked.

By dynamically selecting and invoking good-performing partitioning algorithms, the meta-partitioner will significantly help to reduce the execution time of SAMR applications and improve their scalability.

10 Acknowledgments

The authors are grateful to Ralf Deiterding for providing the application trace files used to evaluated the different partitioning approaches, and to Jaideep Ray for valuable CCA help.

References

[1] AMROC - Blockstructured adaptive mesh refinement in object-oriented C++. http://amroc.sourceforge.net/index.htm, Oct. 2006.

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

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

[4] 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.

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

[6] 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.

[7] Chombo - Infrastructure for adaptive mesh refinement. http://seesar.lbl.gov/ANAG/chombo/, Dec. 2006.

[8] Mattew W. Choptuik. Experiences with an adaptive mesh refinement algorithm in numerical relativity. *Frontiers in Numerical Relativity*, pages 206–221, 1989.

[9] R. Deiterding, R. Radovitzky, L. Noels S. Mauch, J.C. Cummings, and D.I. Meiron. A virtual test facility for the efficient simulation of solid material response under strong shock and detonation wave loading. *To appear in Engineering with Computers*, 2006.

[10] Karen Devine et al. Design of dynamic load-balancing tools for parallel applications. In *Proceedings of the 14th international conference on Supercomputing*, 2000.

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

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

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

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

[15] Henrik Johansson and Johan Steensland. A characterization of a hybrid and dynamic partitioner for samr applications. Report 2004-009, Department of Information Technology, Uppsala University, Sweden, 2004. Available at http://www.it.uu.se/research/reports/2004-009/.

[16] 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/.

[17] Zhiling Lan, Valerie E. Taylor, and Greg Bryan. Dynamic load balancing of SAMR applications on distributed systems. In *Proceedings of 30th International Conference on Parallel Processing*, 2001.

[18] 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.

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

[20] 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.

[21] Lois Curfman McInnes et al. Parallel PDE-based simulations using the Common Component Architecture. In Are Magnus Bruaset and Aslak Tveito, editors, *Numerical Solution of PDEs on Parallel Computers*, volume 51 of *Lecture Notes in Computational Science and Engineering (LNCSE)*, pages 327–384. 2006. Invited chapter, also Argonne National Laboratory technical report ANL/MCS-P1179-0704.

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

[23] 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.

[24] Manish Parashar and James C. Browne. On partitioning dynamic adaptive grid hierarchies. In *Proceedings of the 29th Annual Hawaii International Conference on System Sciences*, 1996.

[25] Manish Parashar, James C. Browne, Carter Edwards, and Kenneth Klimkowski. A common data management infrastructure for adaptive algorithms for PDE solutions. In *Proceedings of Supercomputing*, 1997.

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

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

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

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

[30] Johan Steensland. Irregular buffer-zone partitioning reducing synchronization cost in SAMR. *International Journal of Computational Science and Engineering (IJCSE)*, 2006. Special issue, to appear.

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

[32] 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.

[33] Johan Steensland, Jaideep Ray, Henrik Johansson, and Ralf Deiterding. An improved bi-level algorithm for partitioning dynamic grid hierarchies. Technical report, Sandia National Laboratories, 2006. SAND2006-2487.

[34] Johan Steensland, Michael Thuné, Sumir Chandra, and Manish Parashar. Characterization of domain-based partitioners for parallel samr applications. In *Proceedings of the IASTED International Conference on Parallel and Distributed Computing Systems*, pages 425–430, 2000.

[35] The Virtual Test Facility. http://www.cacr.caltech.edu/asc/wiki, Oct. 2006.

[36] M. Thuné. Partitioning strategies for composite grids. *Parallel Algorithms and Applications*, 11:325–348, 1997.

[37] 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 *Proceedings of Supercomputing*, 2001.

[38] Andrew M. Wissink, David Hysom, and Richard D. Hornung. Enhancing scalability of parallel structured AMR calculations. In *Proceedings of the 17th ACM International Conference on Supercomputing*, pages 336–347, 2003.

[39] 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.