# Design and Implementation of an Adaptive Meta-Partitioner for SAMR Grid Hierarchies

Henrik Johansson

June 19, 2008

**Abstract**

In this paper we present a pilot implementation of the Meta-Partitioner, a partitioning framework that automatically selects, configures, and invokes suitable partitioning algorithms for Structured Adaptive Mesh Refinement (SAMR) applications. Efficient use of SAMR on parallel computers requires that the dynamic grid hierarchy is repeatedly repartitioned and redistributed. The partitioning process needs to consider all factors that contribute to the run-time, i.e. computational load, communication volume, synchronization delays, and data movement. There is no partitioning algorithm that performs well for all possible grid hierarchies — instead the algorithms must be selected dynamically during run-time.

At each repartitioning, the Meta-Partitioner uses performance data from previously encountered application states to select the partitioning algorithm with the best predicted performance. Before the repartition, we determine a partitioning focus to direct the partitioning effort to the performance-inhibiting factor that currently has the largest impact on the execution time.

The implementation uses component-based software engineering (CBSE) to allow for easy expansion and modification. Also, by employing CBSE it is easy to adapt existing SAMR engines for use with the Meta-Partitioner.

## 1 Introduction

Structured adaptive mesh refinement (SAMR) is used to decrease the run-time of simulations in areas like computational fluid dynamics [3, 10], numerical relativity [8, 14], astrophysics [5, 24], and hydrodynamics [20]. 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.

For efficient use of SAMR on a parallel computer, the dynamic resource allocation makes it necessary to repeatedly repartition and redistribute the grid hierarchy over the participating processors. 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 migration 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 considered. This is non-trivial because the basic conditions for how to allocate hardware resources

may change dramatically during run-time due to the dynamics inherent in both the applications and the computer system.

No single partitioning algorithm is the best choice for all conditions [31]. 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* [15, 16, 33, 37], a framework that automatically selects, configures, and invokes the best predicted partitioning algorithm with respect to the current application and computer state. For user-friendliness, easy modification, and expandability, the Meta-Partitioner 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 characteristics. The algorithm that generated 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 motivation and need for a Meta-Partitioner is discussed in Section 3 and general design considerations is the subject of Section 4. An overview of previous work is given in Section 5. The work flow of the Meta-Partitioner is described in Section 6 and the pilot implementation and functionality of the resulting software components is found in Section 7. We present a number of idea for improvements in Section 8. Finally, Section 9 contains a summary and our conclusions.

# 2 Background

In this section we describe structured adaptive mesh refinement (SAMR). We also present the most common partitioning algorithms for 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 compared to a grid with uniform resolution.

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.

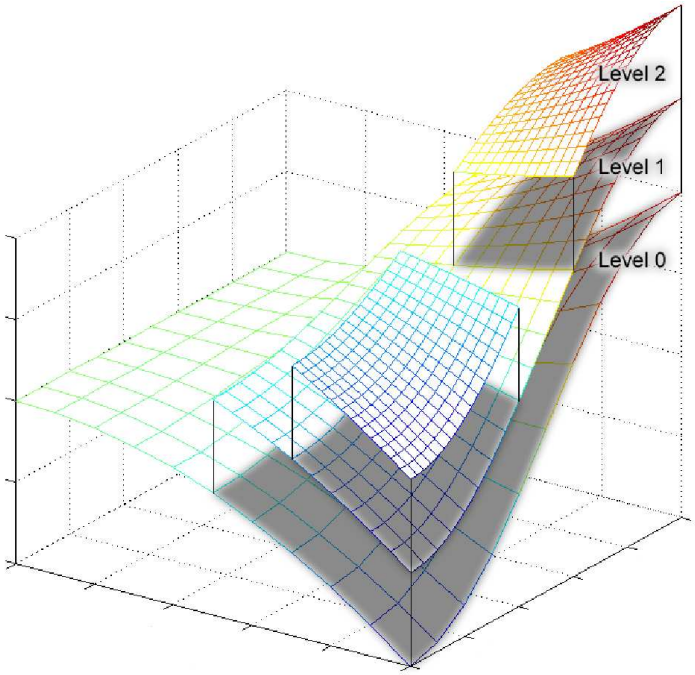



Figure 1: Example of a grid hierarchy with two levels of refinement. The grids are skewed to reflect the characteristics of a solution.

## 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 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 efficiency.

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 necessary lack data. Synchronization delays can be severe — the time spent waiting for data can be of the same magnitude as the actual computational time [36, 38]. 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 [12]. Instead, the partitioner needs to trade-off the metrics in accordance with the characteristics of the application and computer. Ultimately, the 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, 19, 29], the distribution decisions are made independently for each refinement level (or patch). The SAMR frameworks SAMRAI [40, 41] and Chombo [7] supports patch-based partitioning. *Domain-based partitioners* [26, 28, 33, 39] 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, 10] and GrACE [27] frameworks. *Hybrid partitioners* [18, 26, 39] 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, 27, 41] 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 work 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, especially inter-level communication, is generally increased when a patch is subdivided to create a lower load imbalance. Communication serialization bottlenecks can occur when overlaid patches are assigned to different processors. A coarser patch is typically assigned to fewer processors than a finer patch. A processor owning coarser patches will generally need to communicate with many processors having finer and overlaid patches, 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 heterogeneous workload. The

minimum patch 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 always reside on the same processor, 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 complex or deep grid hierarchies. Furthermore, synchronization 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 [30, 37]. Unfortunately, their shortcomings often make their performance unacceptable for deep and complex hierarchies [30, 31]. 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.

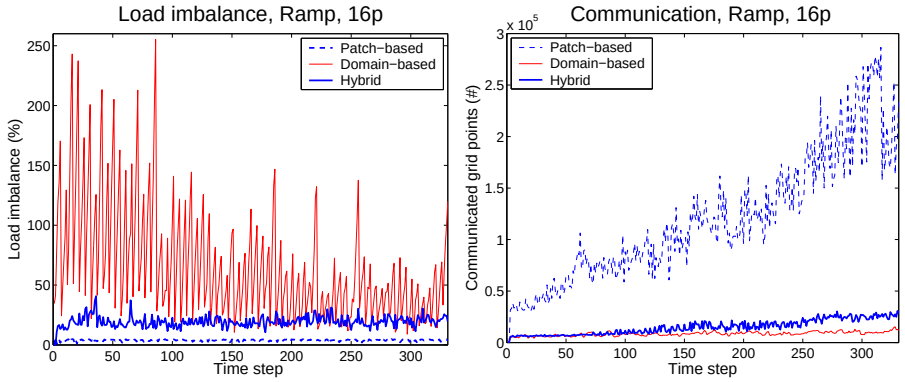
To illustrate the hybrid approach, we describe the partitioning framework that is used as a basis for the Meta-Partitioner — Nature+Fable. Key concepts in Nature+Fable are separation of refined and unrefined areas of the grid and clustering of refinement levels [31]. Separation of unrefined and refined 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 the bi-levels, communication volumes are generally smaller for the hybrid algorithms than for patch-based algorithms.

# 3 The need for a Meta-Partitioner

During the execution of a parallel SAMR application, the grid hierarchy is generally repeatedly repartitioned. Individual partitioning algorithms only produce good-performing partitions for a limited number of application and com-




(a) Load imbalance

(b) Communication

Figure 2: Load imbalance and communication for the most common partitioning approaches. Note how the algorithms complement each other. The values for the domain-based and the patch-based methods are from a single and static partitioning algorithm, while the hybrid values are selected from over 800 algorithms [17] at each time step. The example application, Ramp, is taken from the Virtual Test Facility [38] and it was partitioned for 16 processors.

puter states. Thus, if we only use a single algorithm for the repartitioning, we will construct bad performing partitions for many application and computer states [15, 31]. To consistently construct good-performing partitions that keep the run-time low, 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 application and computer states. Furthermore, we must also have the capability to select between these partitioning algorithms 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 with a more balanced relationship between the load balance and communication. 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 computational work 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 out-

come. 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 [15, 16, 33, 37]. The Meta-Partitioner automatically 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 a pilot implementation of the Meta-Partitioner.

# 4 Design considerations

Before implementing the Meta-Partitioner, several general and far-reaching design decisions must be made. These decisions are essential 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 require 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 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 user's preferred SAMR engine, incorporating the Meta-Partitioner in a simulation can be hard 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 new frameworks can be difficult and time-consuming. Also, if the SAMR framework is modified, a Meta-Partitioner dedicated to that framework might cease to function properly.

A remedy to these problems is to use component-based software engineering (CBSE) to enable inter-operability between components. To construct and execute a CBSE application, the components (i.e. the Meta-Partitioner and a SAMR engine) are connected through well-defined interfaces to form a single executable entity. With CBSE, the Meta-Partitioner can seamlessly be used and connected to any available SAMR engine that conforms to the selected CBSE specification.

# 5 Initial work

The huge range of possible application states requires that the Meta-Partitioner to have access to a large number of complementing partitioning algorithms. To our knowledge, the only SAMR partitioner able to adapt to a multitude of partitioning conditions is Nature+Fable [31]. We thus select Nature+Fable as the basis of the Meta-Partitioner. However, to cover the full partitioning spectrum, the Meta-Partitioner must also be complemented with both patch-based and domain-based partitioners.

In an early evaluation of Nature+Fable [16], we manually tried to find rules for selecting the partitioning algorithm. For the evaluation, we used unpartitioned trace files from four SAMR applications. Each trace file contains the complete grid hierarchies for a full execution of an application. After partitioning, the partitioned trace files were used as input to a SAMR simulator [6] that simulates the common Berger-Colella SAMR algorithm [3]. The simulator computes key 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 [23] where one parameter at a time was varied. In total, eleven hybrid partitioning algorithms were evaluated. The results showed that each parameter improved a specific performance metric, but large parameter inter-dependencies were also discovered. Because of the relatively small number of partitioning algorithms and the parameter inter-dependencies, no rules to select the parameters could be found.

To increase the amount of performance data, a large performance characterization of Nature+Fable was performed [17]. Trace files from four advanced application from the Virtual Test Facility [10, 38] were partitioned by approximately 800 hybrid algorithms from Nature+Fable. The large amount of performance data were stored in a data base. We evaluated the partitioned grid hierarchies using an expanded version of the simulator with the capability of also presenting an approximation of the synchronization penalty [32] and per-processor communication volumes. The per-time step analysis of the performance data shows the complementing characteristics of the partitioning algorithms (see Figure 2). The analysis can be regarded as a proof-of-concept for the need and viability of dynamic selection of the partitioning algorithm.

An algorithm performance data base, like the one resulting from the larger characterization [17], is a necessary foundation for the implementation of the Meta-Partitioner. Without access to comprehensive performance data, it is impossible to implement any viable method that in advance predicts the performance of a partitioning algorithm.

# 6 The Meta-Partitioner workflow

To identify suitable components for the Meta-Partitioner, we need to consider 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 partitioning algorithms and software (e.g. performance monitoring software) become available.

The workflow must result in components with 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).

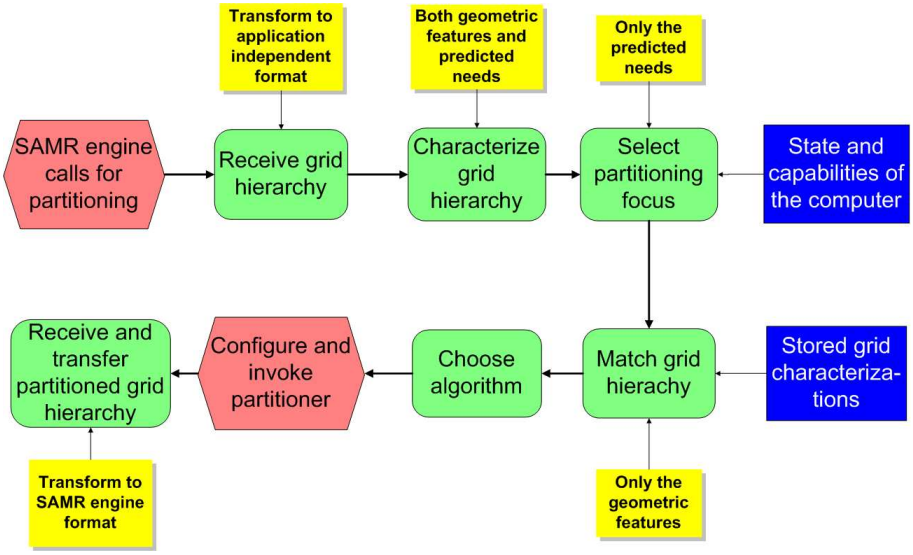



Figure 3: The Meta-Partitioner workflow. The Hexagons represent operations performed outside the Meta-Partitioner. The (green) boxes with smooth corners are tasks performed by the Meta-Partitioner. The dark grey (blue) boxes hold input data while the light grey (yellow) boxes are comments

## 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 has to be translated into an independent and internal grid representation.

## 6.2 Characterize the grid hierarchy

To select good-performing partitioning algorithms, the current state of the grid hierarchy must be thoroughly and accurately characterized. Without knowledge about the current grid hierarchy, it is impossible to in advance predict the performance of a certain partitioning algorithm. For the characterization, we propose to use two sets of complementing metrics to describe the grid hierarchy.

A set of predictive metrics allows us to capture the general partitioning priorities 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?". Metrics that capture these aspects have been developed by Steensland and Ray [34, 35].

Next, a set of descriptive metrics can be computed from the geometric features of the grid. This set of metrics capture the physical properties of the grid, e.g. "how much of the grid is refined?", "how large are the grid patches?", and "how many grid patches are present?". To allow for comparisons, all of the descriptive metrics must be application independent.

Together, the descriptive and predictive metrics give an accurate estimation of both the partitioning needs and the geometric properties 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 should be focused on the factor that is expected to have the greatest impact on the execution time.

We have identified are load imbalance, synchronization costs, communication volumes, and data migration as the main performance-inhibiting factors (see Section 2.2). Subsequently, the partitioning effort should be focused at one of these factors. To select the partitioning focus, we can use the predictive characterization metrics (see Section 6.2) and the current state of the computer.

## 6.4 Matching the current grid hierarchy

In every domain where non-random decisions are made, the decisions are ultimately based on some kind of data or previous knowledge. For the Meta-Partitioner, there are different methods that can be used to select one of the available partitioning algorithms; rules, neural networks, and exhaustive searches. Regardless of what method we choose, the decision will be based on performance data from previously encountered application states.

For many selection methods, it is necessary to match the current application state against stored application states for which we have collected performance data. The most similar stored grid hierarchy can then be used as a starting point for the algorithm selection. To match application states against each other, we can use the descriptive characterization metrics (see Section 6.2) that captures the physical properties of the grid.

## 6.5 Selecting a partitioning algorithm

A number of methods can be used to select the partitioning algorithm. All of these methods require that when grid hierarchies with similar geometrical properties are partitioned with the same algorithm, the resulting partitions will have similar properties. If this notion does not hold, it is impossible to consistently select good performing algorithms for grid hierarchies that have not previously been encountered and evaluated.

The most intuitive method to select the partitioning algorithm is to condense previously collected performance data into general rules that are applied to the current application and computer state. Thus, such a rule is an aggregation of a number of similar application and computer states. For each such aggregation

and partitioning focus, a partitioning algorithm is selected. To construct these rules, a variety of data mining techniques can be used [13].

During run-time, we match the current application state against the rules. The rule that corresponds to the current application state is selected and the contained partitioning algorithm is invoked. The main advantage of rules is a fast and straight-forward selection process. However, since the rules are based on aggregations, information is discarded and generalization are made about the properties of the grid hierarchy. Given the huge range of application states, it is extremely hard - if not impossible - to construct general rules that will perform well during the whole range of possible application and computer states.

An alternative to the rule-based approach is to perform an exhaustive search of the performance data for all stored application states. Instead of using aggregations of the application and computer states, each individual combination of application and computer state is connected to a partitioning algorithm. Since no performance data is discarded and no generalizations are made, the selected partitioning algorithm can be expected to result in better performing partitions. The disadvantage is a very large search space that can potentially slow down the algorithm selection process.

Finally, one can imagine methods that access and evaluate the performance data base during run time. Another alternative is to use some kind of neural network to select the partitioning algorithm. The main disadvantage of both these methods are their high complexity, both in terms of the implementation and in longer execution times.

### 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, translations to and from the Meta-Partitioner's internal grid representation might be needed.

### 6.7 Transfer the partitioned grid hierarchy to the SAMR engine

The partitioner transfers the partitioned grid hierarchy to the Meta-Partitioner. The grid hierarchy is translated from the internal representation into the format used by the SAMR engine. Finally, the grid hierarchy is then returned to the SAMR engine, which resumes computing.

## 7 A pilot implementation of the Meta-Partitioner

In this section we describe our pilot implementation of the Meta-Partitioner. We start with a presentation of the CBSE framework. We then describe each of the components that together constitute the Meta-Partitioner.

### 7.1 The Common Component Architecture

We use the the Common Component Architecture (CCA) as the CBSE framework for the implementation of the Meta-Partitioner) [4]. CCA is a commu-

nity based CBSE initiative, specifically aimed at the needs of parallel scientific high-performance computing. CCA presents a general, low-latency model for component inter-operability and interaction.

In CCA, a component is the 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 paradigms [9]. 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 simplify the development of advanced scientific applications without any negative impact on application performance [22].

In CCA, computational quality of service (CQoS) is getting significant attention [21, 25]. 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 exactly what we want to achieve with 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 [27] are already modified to conform to the CCA specifications.

To use the full potential of CCA, we analyze the internal workflow (see Section 6) of the Meta-Partitioner and divide the workflow into a number of components. Dividing the functionality of the Meta-Partitioner into components allow for expansion and modification, without any loss of efficiency.

The design of the components must allow for all desired features, even if these features are not included in the initial pilot implementation. Below, we describe each of the components. The components, and their ports, are shown in Figure 4.

## 7.2 The core component (Core)

The `Core` component functions as the hub for the Meta-Partitioner — it is connected to all other components and controls the execution. The `Core`-component sends data to the other components and receives their output. Though the software architecture does not explicitly require a `Core`-component, it simplifies expansion and modification. In a more linear software architecture, even small changes might cascade through several components.

The task of the `Core` is to perform a number of sequential function calls and to handle all associated data transfers. The pseudo-code of the `Core`-component is listed in Algorithm 1. The parenthesises contain the name of component responsible for each task.

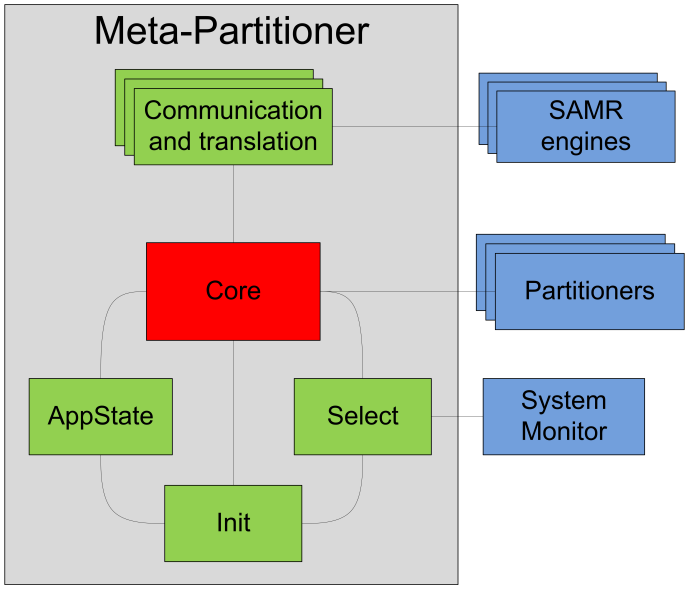



Figure 4: The CCA components used in the Meta-Partitioner. The components on the right are developed externally, but used by the meta-partitioner.

**Algorithm 1** The Core component

```
CollectStaticData(Init)
ReceiveGridHierarchy(CoT)
CharaterizeHierarchy(AppState)
SelectPartitioningFocus(Select)
MatchGridHierarchies(Select)
SelectAlgorithm(Select)
PartitionGridHierarchy(Partitioners)
ReturnGridHierarchy(CoT)
```

### 7.3 Communication and translation components (CoT)

The available SAMR engines can use different formats to describe the grid patches. To implement an efficient and expandable Meta-Partitioner without being limited to a particular SAMR engine, it is necessary to translate the different grid representations into a single internal data format. This is the main task of the `CoT` component.

We use the grid representation developed for the DAGH/GrACE frameworks as basis for the internal data format [27]. The format is listed in Algorithm 2. The `id` is a unique identifier for each grid patch. The field `owner` corresponds to the processor that is assigned to the grid patch. For 2D applications, the `z`-coordinate is set to -1. We have included the field `time-step` to enable the use of trace files for evaluation (similar to the evaluation described in Section 5).

After the grid hierarchy have been partitioned, the `CoT` component translates the grid patches back into their original format and 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

**Algorithm 2** The DAGH/GrACE data format

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

load balancing tool-kit Zoltan, where callback functions translate and provide necessary mesh data [11].

To evaluate the Meta-Partitioner without performing real-world simulations, we have made the partitioning functionality independent of the source of the grid hierarchy. For the evaluation, we use unpartitioned trace files derived from actual SAMR applications (similar to the evaluation described in Section 5). The trace file are received and transfered to the `Core` component by the `CoT` component.

All partitioners currently used by the Meta-Partitioner have native support for the internal grid representation used by the Meta-Partitioner. The `CoT`-component will provide translations to the internal grid representation for future partitioners that use other representations.

## 7.4 Application state component (AppState)

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, the `AppState` component uses the two separate sets of metrics described in Section 6.2. The first set predicts the partitioning priorities of the grid hierarchy. The second set describes the geometrical properties of the grid.

### 7.4.1 Predictive classification space

To consistently select good-performing partitioning algorithms, we need to direct the partitioning focus towards the most performance inhibiting factor. If we predict that it will be hard to evenly divide the workload between the processors, we should use a partitioning algorithm that specifically targets a low load imbalance, even at the expense of larger amounts of communication and synchronization. If the grid hierarchy is believed to results in large amounts of communication, we instead prefer to use an algorithm that gives priority to reducing the communication volumes.

To determine the most performance inhibiting factor and to select the partitioning focus (see Sections 6.3 and 7.5.1), we use a predictive classification space, PCS (in earlier publications known as the Partitioner Centric Classification Space, PCCS). Within the PCS, we use three metrics that estimate the impact of load imbalance, communication, and data migration on the partitioning quality [34, 35]. The metrics are based on key observations from worst case scenarios for both the patch-based and domain-based partitioning approaches

— what are the properties of the grid hierarchies that result in large communication volumes for patch-based algorithms and high load imbalances for domain-based algorithms?

The load balance metric captures the difficulty to evenly divided the refined part of the grid hierarchy [34]. For each connected and refined part of the grid (possibly consisting of many grid patches), we calculate both the optimal and the maximum number of processors that can be assigned to the grid part. The difference between these numbers is a good predictor of the difficulty to to evenly divide and distribute the grid part. It is generally harder to achieve a good load balance when the difference is small. For example, if the optimal number of processors is four and maximum number of processors is twenty, the partitioner has a high degree of flexibility to divide and assign the grid part to the processors. If the difference is small, the partitioner is more restricted when it comes to divide and assign the grid part to the processors.

The total communication volume is generally dominated by inter-level communication because large chunks of the grids are often transfered between refinement levels, while data on the same level are only exchanged at the borders of grid patches. The communication metric is computed using the size of overlaid grids in relation to the sizes of grids below them [34]. The result is an algebraic expression that estimates the amount of communication that can be generated when the current grid hierarchy is partitioned.

The amount of data migration is estimated by intersecting the previous grid hierarchy with the current grid hierarchy [35]. If the difference between the grid hierarchies is large, the amount of data migration can be high. However, we must also consider the direction of the change. When a grid hierarchy shrinks, parts of the grid are deleted and not migrated. Thus, the data migration metric is designed reflect this observation.

Because of the high complexity involved in predicting the impact of synchronization delays, it is difficult to develop an accurate synchronization metric inside the PCS. However, a synchronization delay can *only* occur if two processors need to exchange data. As a consequence, there is a relationship between synchronization delays and the number of communication. Even though this relation is influenced by many properties of the grid hierarchy, we use the predictive communication metric to estimate the impact of synchronization delays.

### 7.4.2 The descriptive classification space

The geometrical properties of the grid hierarchy are captured by the descriptive classification space (DCS). The metrics included in the DCS can be seen as a snapshot of the physical features of the current grid hierarchy.

For each descriptive metric, the result is computed for both 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 base grid or the size of a certain refinement level.

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

**Number of refinement levels** The number of refinement levels is important for the matching of application states. Performance predictions for ap-

plications using different numbers of refinement levels are probably 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, with regard to the refined area on the next lower level. A value close to one means that we are likely have sharp features in the grid hierarchy, as the sizes 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 might indicate a refinement pattern that covers a curved shock waves, making it necessary to use many small patches to resolve the shock wave.

**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 larger communication volumes and synchronization delays.

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

Because the magnitude of the different PCS and DCS metrics can vary by several orders, it is necessary to normalize their values to a common interval. The Meta-Partitioner uses *logistic normalization* for this task. In logistic normalization, we first normalize each metric using the mean and standard deviation based *z-score normalization* [13]. A value $v$ of a metric $A$ is normalized to $v_{zero}$ by computing

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

where $\tilde{A}$ and $\sigma_A$ are the mean and standard deviation of metric A. The mean and standard deviation are pre-computed from the stored grid hierarchies and supplied by the `Init`-component (see Section 7.6). The z-score normalization is useful when the actual minimum and maximum values of metric A is unknown, which makes it easier to later add new application data. Also, z-score normalization is less sensitive to outliers than many other normalization methods [13]. However, when many values are clustered around the mean, the z-score normalization often makes the values similar.

Having many similar values makes the matching of application states more sensitive and less accurate. To improve the matching, we use logistic normalization to increase the relative difference between values close to the mean. The

final logistic normalized value is computed by inserting $v_{zero}$ into the expression below.

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

After the logistic normalization, the values is restricted to the interval $[0, 1]$. Furthermore, values that previously were far from the mean have been compressed in a larger extent then values were close to the mean. Thus, the resolution of values close to the mean (i.e. close to 0.5) have increased.

## 7.5 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 partitioning focus, it matches the current grid hierarchy against stored grid hierarchies, and it selects the partitioning algorithm that is predicted to result in the best performance for the current grid hierarchy.

### 7.5.1 Selecting a partitioning focus

The first task for the `Select`-component is to determine the partitioning focus (see Section 6.3 for a description of the focus). In the pilot implementation, 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 [32, 38]. For the data migration, we currently lack performance data that are needed during the algorithm selection step. If necessary, we can later expand the focus to include both data migration and communication.

To simplify both the determination and the use of the partitioning focus, we employ eight discreet focus levels (see Table 1). We assign half of these focus levels to the load imbalance and the other half to the synchronization delays. The name of each level corresponds to the targeted performance-inhibiting factor. For each of the levels, we decide on a maximum allowed performance deviation from the best stored performance data. As an example, for the focus level "focusLB1.5", we allow an load imbalance that is 50 percent higher than lowest recorded load imbalance for the most similar stored grid hierarchy. The sizes of the allowed deviations was determined after careful analysis of the distribution of the performance data.

The focus will concentrate the partitioning effort onto the most performance inhibiting factor, but we will also need to control the impact of the other factors as well. We initially use the the focus to select a number of candidate partitioning algorithms that perform well with regard to the most performance inhibiting factor (i.e. equal to or better than the allowed performance deviation). The performance of these candidate algorithms is then evaluated again, but this time with regard to the other performance inhibiting factor. Using this strategy, we will select an algorithm that performs well for the most performance inhibiting factor, while the impact of the other factor are kept as low as possible. The details of the algorithm selection are found in Section 7.5.3.

The initial partitioning focus is selected by analyzing static application and computer characteristics — the computational requirements of the application and the capabilities of the computer system. The pilot implementation of the Meta-Partitioner does not currently support modification of the focus during run-time. Once an initial focus has been determined, the focus remains fixed throughout the execution.

| **Focus** | **Corresponding rule** |
|---|---|
| FocusSynch 1.25 | Synch 125% of minSynch |
| FocusSynch 2 | Synch 200% of minSynch |
| FocusSynch 3 | Synch 300% of minSynch |
| FocusSynch 5 | Synch 500% of minSynch |
| FocusLB 1.2 | LB 120% of minLB |
| FocusLB 1.5 | LB 150% of minLB |
| FocusLB 2 | LB 200% of minLB |
| FocusLB 3 | LB 300% of minLB |

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

In future versions, the partitioning focus will be continuously modified during run-time using the Predictive characterization space (see Section 6.2) and the current state of the computer system. Depending on the amount of change in the PCS and the state of the computer system, the partitioning focus is either kept or shifted a few steps in any direction. To avoid a widely oscillating focus, we use the previous focus as a starting point and restrict the maximum amount of focus change. 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.

### 7.5.2 Matching of grid hierarchies

After selecting the partitioning focus, the `Select`-component proceeds to match the current grid hierarchy against the stored grid hierarchies that was used for the extensive performance characterizations [17]. For the matching of the hierarchies, we use the descriptive characterization metrics (see Section 7.4) and the common least square method. The stored grid hierarchy that is most similar to the current hierarchy is recorded.

The workload distribution for a SAMR application is generally top heavy — the workload is concentrated to the higher refinement levels. Thus, an imbalance at a high refinement level is likely to have a larger impact on the execution time than an imbalance at a lower level. To give higher precedence to refinement 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 properties of levels with large workloads are given a higher priority.

During the matching process, we might need to compare the current grid hierarchy against stored grid hierarchies that does not have the same number

of refinement levels. For these cases, we remove the lower refinement levels on the deepest grid hierarchy to make the comparison between an equal number of levels. We do not consider the lower levels as the larger workloads on the higher refinement levels have a greater impact on the performance.

### 7.5.3 Selecting the partitioning algorithm

For the algorithm selection, we assume that grid hierarchies that have similar geometrical properties generally also have resembling partitioning properties. During extensive algorithm performance characterizations (described in Section [17]), we collected large amounts of performance data in a data base. For the stored grid hierarchy that is found to be most similar to the current grid hierarchy, we use this performance data to select the partitioning algorithm that performed best with regard to the current partitioning focus (see Section 6.5). Thus, the algorithm selection is only dependent on the current partitioning focus and a stored application state. Because the focus is divided into discreet levels, we can pre-compute the algorithm selection for all combinations of focus and application states. During run time, we only need to perform a simple table look up for the most similar stored grid hierarchy to select the partitioning algorithm. The selected algorithm is then transfered to the `Core` component.

When pre-computing the rules, we first evaluate all partitioning algorithms for each stored grid hierarchy with regard to the most performance inhibiting factor (e.g. the partitioning focus). All partitioning algorithms with a performance deviation smaller than the threshold determined by the focus are candidate algorithms (see Section 7.5.1). From these candidate algorithms, we now select the algorithm that has the best performance with respect to the other performance-inhibiting factor (i.e. load imbalance if the focus is synchronization, and synchronization if the focus is load imbalance). We call each combination of focus and application for a *rule*. SQL-like pseudo-code for the algorithm selection is presented in Algorithm 3.

**Algorithm 3** Selection of partitioning algorithm

Using focusSynch{X}
1 **SELECT** partAlg **AS** candidates **FROM** mostSimilarAppState **WHERE** synch < X***MIN**$_{all}$(synch)
2 **SELECT** partAlg **FROM** candidates **WHERE** LB = **MIN**$_{cand}$(LB)

Using focusLB{X}
1 **SELECT** partAlg **AS** candidates **FROM** mostSimilarAppState **WHERE** LB < X***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.

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 deliberately avoid selecting the algorithm with regard to only the

most performance inhibiting factor, as this algorithm often performs poorly for all other factors.

### 7.6 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 small and simple as possible, we have moved many of the utility-type tasks to the `Init` component.

During the grid characterization step (performed by the `Appstate` component), both the predictive and the descriptive metrics are normalized to allow for comparisons of different grid hierarchies (see Section 7.4). As a first step, the data is normalized with z-score method. The necessary mean and standard deviation for the all stored hierarchies are provided by the `Init` component.

Before the `Select` component matches current grid hierarchy against stored hierarchies, the `Init` component reads the DCS metrics for the stored grid hierarchies from disk. The metrics are then transfered them to the `Select` component. This task is performed only once for each simulation.

After the partitioning focus has been determined,m the `Init` component reads the corresponding precomputed rule from disk and transfers the rule to the `Select` component.

### 7.7 Partitioning algorithm component

In the pilot implementation we use Nature+Fable [31] as the partitioning algorithm component. Nature+Fable is a hybrid partitioning framework with the ability to adjust to a large number of partitioning conditions (see Section 2.2).

To incorporate Nature+Fable into the Meta-Partitioner, a simple wrapper is implemented. The only function of the wrapper is to make a simple function call to Nature+Fable.

### 7.8 CCA ports

The different components of the Meta-Partitioner communicate through a number of ports, as seen in Figure 4. Every component, with ex port that gives access to the functionality contained within the component. A component only has one *provides* port, even if its functionality are used by several other components (like the `Init`-component). The components that use functionality provided by other components all have a corresponding *uses* port. We do not describe the individual ports in this section, since such a description would be a repetition of the functionality of each components.

## 8 Future Work

In this report we have described our pilot implementation of the Meta-Partitioner. To improve the partitioning performance, we can envision a number of modifications and expansions to the Meta-Partitioner.

## 8.1 SAMR engines

The Meta-Partitioner should be connected to one or more SAMR engines. To turn a SAMR engine into a CCA-component, a wrapper routine must be added. If the SAMR engine can not export its current grid hierarchy, such a capability need to be added to the engine. Instead of calling an internal partitioning algorithm, the SAMR should transfer the grid hierarchy to the MP and `CoT`-component. These modifications should be relatively easy to implement.

## 8.2 Partitioning algorithm components

In earlier works, we have shown that no single partitioning algorithm is the best choice for all application and computer states [31]. 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 only include hybrid partitioning algorithms (see Section 2.2). We are currently adapting a patch-based partitioner for use in the Meta-Partitioner. Unfortunately, we do not currently have access to a stand-alone domain-based algorithm.

Each partitioning algorithm (or partitioning framework) will be incorporated in the Meta-Partitioner as an individual component. This approach makes it easier to update existing partitioners and to add new algorithms.

## 8.3 Determination of static application and computer data

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 7.5.1). This information should be collected before the start of the execution. Currently, the user cannot affect this data. In future versions of the Meta-Partitioner, the user should have the ability 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.

## 8.4 Performance Monitoring Components

The performance of the SAMR application 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 this data. At the basis for this task is the static characteristic data of the computer system that will be 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) [42], 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 modify the partitioning focus to decrease the impact of an overloaded part in the computer system.

### 8.5 Optimization of the descriptive classification space

The selection of good-performing algorithms is dependent on an accurate matching of the properties of both the current and all stored grid hierarchies. For this task, we use the metrics within the descriptive classification space (see Section 7.4).

Some of these metrics have a greater impact on the accuracy of the classification than others. Additional research are needed to determine the best combination of weights for the descriptive metrics. During this research, some of the metrics might be modified while others are removed. Also, it possible that we will find that the DCS needs to be expanded with new metrics.

## 9 Summary and conclusions

In this paper we presented the design and the pilot implementation of the Meta-Partitioner. The Meta-Partitioner is a partitioning framework for parallel SAMR applications that automatically 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 and its implementation consists of several components.

To select good-performing partitioning algorithms, the Meta-Partitioner 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 descriptive metrics that captures the physical properties of the grid hierarchy. The current application state is then matched against a large number of stored states and the most similar stored state is recorded. Algorithm performance data for this stored application state is evaluated. The best partitioning algorithm with regard to the current partitioning focus 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, Oak Ridge National Laboratory, for providing the application trace files used to evaluated the different partitioning approaches, and to Jaideep Ray, Sandia National Laboratories, for valuable help concerning CCA.

# References


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

[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] T. Dahlgren, T. Epperly, G. Kumfert, and J. Leek. *Babel users' guide 1.2.0.* Lawrence Livermore National Laboratory, 2006.

[10] 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. *Engineering with Computers*, 22(3-4):325–347, 2006.

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

[12] 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.

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

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

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

[16] 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/.

[17] 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/.

[18] 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.

[19] 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.

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

[21] 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.

[22] 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.

[23] Douglas C. Montgomery. *Design and Analysis of Experiments*. Wiley, 2004.

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

[25] 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.

[26] 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.

[27] 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.

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

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

[30] 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.

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

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

[33] 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.

[34] 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.

[35] 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.

[36] 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.

[37] 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.

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

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

[40] 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.

[41] 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.

[42] 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.