Run-time selection of partitioning algorithms for parallel SAMR applications

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

Abstract

Parallel structured adaptive mesh refinement methods decrease the execution time and memory requirements of partial differential equation solvers. These methods result in an adaptive and dynamic grid hierarchy that repeatedly needs to be re-partitioned and distributed over the processors. No single partitioning algorithm can consistently construct high-quality partitionings for all possible grid hierarchies. Instead, the partitioning algorithm needs to be selected during run-time. In this paper, an initial implementation of the Meta-Partitioner is presented. At each re-partitioning, the Meta-Partitioner autonomously selects, configures, and invokes the partitioning algorithm predicted to result in the best performance. To predict the performance of the partitioning algorithms, the Meta-Partitioner uses historic performance data for grid hierarchies with properties similar to the current hierarchy. The Meta-Partitioner focuses the partitioning effort on the most performance-inhibiting factor — either the load imbalance or the synchronization delays. The performance evaluation shows a small but noticeable performance increase compared to the best static algorithm. Compared to the average performance for a large number of partitioning algorithms, the Meta-Partitioner consistently generates partitionings with a significantly better performance.

1 Introduction

The efficiency of parallel PDE solvers that use structured adaptive mesh refinement (SAMR) is often degraded by low quality partitionings. The use of SAMR result in dynamic and adaptive grid hierarchies that can have vastly different characteristics. Using a single partitioning algorithm for the entire execution is sub-optimal because no single partitioning algorithm is suitable for all grid hierarchies [15, 24].

To consistently construct good performing partitionings, different partitioning algorithms must be selected and invoked during run-time with respect to the current characteristics of the application. Furthermore, both the capabilities and the state of

the parallel computer need to be considered. Optimally, the invoked partitioner should simultaneously minimize the impact of all partition-related performance-inhibiting factors like load imbalance, communication costs, synchronization delays, and data migration. Dynamic selection of the partitioning algorithm during run-time is challenging — both the state of the application and the computer system will generally change substantially during run-time.

For most application and computer states, the choice of partitioning algorithm is a trade-off between the performance-inhibiting factors [24]. A small load imbalance generally results in high communication volumes. If the goal is to decrease the communication, the load imbalance is normally increased. To consistently construct high-quality partitionings, the partitioning effort must continuously be focused on the most performance-inhibiting factors.

The work presented in this paper is part of a larger research project that aims to decrease the run-time of general SAMR applications executing on general parallel computers [12, 13, 16, 24, 26–28]. A fundamental component in this project is the Meta-Partitioner, a partitioning framework that autonomously selects the partitioning algorithm with respect to a determined partitioning focus. For the algorithm selection, it is assumed that similar application states have similar partitioning properties. This assumption enables the use of stored performance data for previously encountered applications and partitioning algorithms. Using the historical performance data, the performance of a candidate partitioning algorithm can be predicted for the current grid hierarchy.

In this paper, a performance analysis of the initial implementation of the Meta-Partitioner is presented. Using four real-world applications, the performance of the Meta-Partitioner is compared to both the static partitioning algorithm that is predicted to result in the best performance and the average result for all partitioning algorithms in the evaluation. The stability and accuracy of the algorithm selection process is examined by simulating the performance of a number of alternative partitioning algorithms.

2 Background

Structured adaptive mesh refinement (SAMR) is a method for decreasing the execution time and memory requirements of partial differential equation solvers. In SAMR, computational resources are dynamically assigned to computationally demanding regions [5, 6]. The execution starts with a coarse base grid that covers the entire computational domain and has the smallest acceptable resolution. During execution, regions with large local solution errors are continuously identified. New grids with higher spatial and temporal resolution are overlaid in these regions, enabling the computation of more accurate local solutions. This refinement process proceeds recursively. In regions where the solution error is sufficiently small, refined grids are removed. The resulting data structure is a dynamic adaptive grid hierarchy. The inherent dynamics require that the hierarchy is repeatedly re-partitioned and distributed over the processors.

During the execution of parallel SAMR applications, information flows both along the same refinement level and between different refinement levels. As grid patches are divided among processors, boundary data are exchanged between the processors.

Before computations can start on higher refinement levels, grid data must be supplied from the lower levels. Furthermore, to increase the accuracy on the lower levels, the solution data is projected down to lower refinement levels. Hence, the number of communicated grid points as well as the total size of these communications are two performance-inhibiting factors for SAMR applications.

Synchronization delays can have a much larger impact on the performance than the actual communication of the grid points [15]. Processors frequently need to synchronize and exchange data. Often, one of the involved processors is busy computing while the others have to be idle. The time spent waiting for data can be substantially times longer than the time needed to actually communicate the data. As seen in Table 1, the time waiting for data can even be as long as the actual time spent computing. In this paper, a metric called synchronization penalty is used as an approximation of these synchronization delays [25]. The computation of the synchronization penalty is described in Section 5.2.

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

Table 1: Comparison between computational time, synchronization time, and total execution time for four example applications from the SAMR framework AMROC [2] and a domain-based partitioning algorithm. Except for the computational time and the synchronization time, the third major component of the total time is the determination of internal data exchanges between grids assigned to the same processor and external communications between grids assigned to different processors. The data origins from sixteen processor executions on the ALC parallel computer at Lawrence Livermore National Laboratory [1]. All data courtesy of Ralf Deiterding.

2.1 Partitioning approaches

The parallel performance of SAMR applications is heavily dependent on the partitioning of the dynamic adaptive grid hierarchy. Optimally, the partitioning algorithm should minimize all of the partition-related performance-inhibiting factors. For the partitioning of SAMR grid hierarchies, three main approaches are identified. Most partitioning algorithms fall into one of these categories.

For *patch-based partitioners* [3, 9, 16, 17], the distribution decision is made independently for each grid patch. A grid patch may be kept on the local processor or moved entirely to another processor. If the grid patch is large, it can be split. The main advantage of the patch-based approach is a small load imbalance, both globally and on the individual refinement levels. Shortcomings in this approach are high communications cost, potentially large synchronization penalties, and an inability to exploit available parallelism across grids at different levels.

Domain-based partitioners [4, 21, 23] partition the physical domain, rather than the grids themselves. The domain is partitioned along with all overlaid grids from all refinement levels. Generally, the workload of the overlaid grids are projected down to the base grid, reducing the problem to the partitioning of a single grid that has a heterogeneous workload. The advantages are elimination of inter-level communication and better exploitation of all available parallelism between different levels of refinement. The main disadvantage is an intractable load imbalance for deep hierarchies that can be further amplified by even larger imbalances on the individual refinement levels. Another common drawback is the occurrence of "bad cuts" that results in increased overhead costs [24].

Hybrid partitioners [18, 24] combine the patch-based and domain-based partitioning approaches to avoid their respectively shortcomings. Most hybrid partitioners use a 2-step partitioning approach. The first step use domain-based techniques to generate meta-partitionings that are mapped to a group of processors. The second step uses a combination of domain and patch-based techniques to optimize the distribution of each meta-partitioning within its processor group.

The hybrid partitioning framework Nature+Fable uses domain-based techniques to separate the unrefined parts of the coarse base grid from the refined parts [24]. For each separate refined area, every two levels of refinement are clustered into a bi-level. In a bi-level, the highest refinement level is partitioned using the patch-based approach. The resulting partitioning is then projected down in a domain-based fashion onto the lower refinement level. Thus, inter-level communication is avoided inside each bi-level, while domain-based partitioning is never performed on more than two refinement levels to control the load imbalance. The partitioning process in Nature+Fable is governed by a large set of parameters where each parameter setting can be regarded as a separate partitioning algorithm.

3 Dynamic selection of partitioning algorithms

During the execution of a parallel SAMR application, the grid hierarchy is adaptively evolving to ensure that the solution error is kept sufficiently small. A partitioning algorithm that constructs high-quality partitionings during a number of time steps might not be competitive at a later stage [24]. Furthermore, the state of the parallel computer can also change during run-time as other applications are executed concurrently — imposing even higher demands on the partitioning algorithm. Thus, to consistently get good performance throughout the complete execution of the application, the partitioning algorithm must be selected dynamically during run-time.

A small initial performance characterization of hybrid partitioning algorithms originating from Nature+Fable showed that it is intractable to manually construct rules to select the partitioning algorithms [14]. It was often possible to select a hybrid partitioning algorithm that influenced a specific performance metric, i.e. load imbalance, but large parameter inter-dependencies made the amount of change inconsistent. The effects on other performance metrics, i.e. communication and synchronization costs, were generally hard or even impossible to predict. Hence, other selection methods are needed.

To design more advanced selection methods, it is assumed that geometrically similar grid hierarchies have similar partitioning properties. A partitioning algorithm that generates a high-quality partitioning for a certain grid hierarchy probably also does so for a geometrically similar grid hierarchy. If this is false, it will be virtually impossible to predict and compare the performance of different candidate partitioning algorithms during run-time. If the geometrical properties of the current grid hierarchy is accurately characterized, historic performance data from similar grid hierarchies can be used during run-time to predict the performance of a candidate partitioning algorithm.

As a basis for the algorithm selection, comprehensive performance data for each candidate partitioning algorithm are needed. Furthermore, the performance data should be collected from a wide range of different partitioning conditions. In a large performance characterization, 768 hybrid partitioning algorithms from Nature+Fable were used to partition almost 1300 different grid hierarchies [15]. The resulting performance data and the grid hierarchies were stored in a large data base for easy access and evaluation.

The current grid hierarchy is matched with the stored grid hierarchies present in the data base and the most similar hierarchy is recorded. The historical performance data for all partitioning algorithms that have partitioned the most similar stored hierarchy is extracted. Based on the extracted data, the performance of the partitioning algorithms is predicted for the current hierarchy. Because the resulting execution time is a product of a number of performance-inhibiting factors (e.g. load imbalance, synchronization, communication etc.), partitioning algorithms predicted to perform well for only a single performance-inhibiting factor should generally not be considered. Instead, a partitioning focus is determined by using the properties of both the current application state and the computer system. The partitioning algorithm with the best predicted overall performance with respect to the partitioning focus is selected, configured, and invoked.

4 Implementation of the Meta-partitioner

The Meta-Partitioner is designed and implemented to allow for dynamic selection of the partitioning algorithm during run-time [13]. Building on the ideas presented in the previous section, the Meta-Partitioner autonomously selects, configures, and invokes the partitioning algorithm that is predicted to result in the best performance.The implementation is made flexible and expandable by the use of component-based software engineering (CBSE). Also, CBSE simplifies the incorporation of existing SAMR frameworks into the Meta-Partitioner. The implementation of the Meta-Partitioner is performed using the Common Component Architecture (CCA) framework [7]. As a community based CBSE initiative, CCA is specifically aimed at the needs of parallel scientific high-performance computing. The core of CCA is 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 provide access to the functionality of a component. A component can provide a port, meaning that it implements the functionality defined by the port. A component can also use ports, by performing function calls through the port to access the func-

tionality 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.

Recently, the CCA community has given significant attention to computational quality of service (CQoS) [19, 20]. The definition of CQoS is the ability of a system to ensure that a scientific problem is solved with the best available hardware and software resources. This definition coincides with the goal of the Meta-Partitioner.

4.1 Software components

The functionality of the Meta-Partitioner is divided into a number of software components. All components are designed to function independently from any SAMR frameworks as well as any data structures that store the grid hierarchy. The components and their connections are showed in Figure 1.

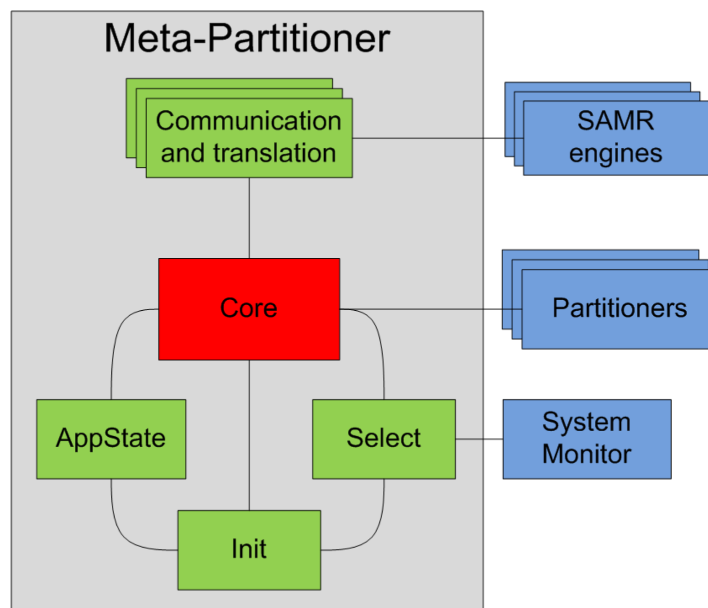


Figure 1: A conceptual Meta-Partitioner and its components. The components on the right are adapted from third-party software.

Core

The `Core`-component is the main component in the Meta-Partitioner. At each re-partitioning, the `Core` coordinates the execution of all other components. Though an implementation does not explicitly require a `Core`-component, it simplifies future expansions and modifications by providing a more homogeneous interface between the components. In practice, the `Core`-component performs a number of sequential

function calls and handles all data transfers between the components. The functionality of the `Core`-component together with the components that are responsible for each task is listed in Algorithm 1.

Algorithm 1 The functionality of the Core-component

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

Init

The `Init`-component contains a number of utility functions that are used both before the start of the execution and during each re-partitioning step, e.g. accessing both the characteristic data for the stored grid hierarchies and the algorithm selection rules. The functionality included in the `Init`-component could generally be performed by other components. However, to keep the components as small and simple as possible, a number of these utility-type tasks has been moved to the `Init`-component.

4.1.1 Communication and translation — CoT

The `CoT`-component functions as an interface between the Meta-Partitioner and any attached SAMR frameworks. At each re-partitioning, the component receives the un-partitioned grid hierarchy from the SAMR framework. After partitioning, the `CoT`-component returns the partitioned grid hierarchy to the SAMR framework. The component also translates the grid hierarchy to and from the internal grid representation used in the Meta-Partitioner.

Application state — AppState

The `AppState`-component characterizes the geometrical properties of the grid hierarchy. A number of metrics are computed from the current grid hierarchy (see Table 2). All metrics are normalized to a common interval using logistic normalization. In logistic normalization, a metric is first normalized using *z-score normalization* [11]. A value v of a metric A is normalized to v_{zero} by

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

where $\tilde{A}$ and σ_A are the mean and standard deviation of metric A. The z-score normalization is useful when the minimum and maximum values of A is unknown, making

it easier to add new data. Also, z-score normalization is less sensitive to outliers than many other normalization methods [11]. However, z-score normalization often results in similar values when many values are clustered around the mean. To increase the relative difference for the metrics, logistic normalization is used [22]. The final normalized value is

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

After the logistic normalization, each metric is restricted to the interval $[0, 1]$ and the relative difference between values close to the mean (i.e. 0.5) has been increased.

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

Table 2: The geometrical metrics computed by the `AppState`-component to characterize the grid hierarchy.

Select

The `Select`-component matches the geometrical properties of the current grid hierarchy with the stored characteristics for the grid hierarchies. For the matching, the weighted least squares method is used. Because the metrics that characterize the properties of the grid hierarchies probably differ in importance, each metric is assigned a weight. The weights are determined experimentally from almost 2200 different weight combinations. For each combination, the Meta-Partitioner selects partitioning algorithms for all grid hierarchies stored in the performance data base. Because the experiments only involve algorithms and hierarchies already present in the data base, the average performance of each weight combination can be determined without the need to actually partition the grid hierarchies. The weight combination that on average results in the best performance is computed and added to the Meta-Partitioner.

Before the partitioning algorithm is selected, a partitioning focus is determined. The purpose of the focus is to concentrate the partitioning effort to the area where it is most useful. In the presented implementation, the focus is constant during runtime and it only involves the two performance-inhibiting factors that generally have the largest impact on the execution time — load imbalance and synchronization (see Table 1 and Section 2). For each focus — `FocusLB` and `FocusSynch` — a maximum allowed performance deviation from the best stored performance data is set. As an example, for `FocusLB1_2`, a load imbalance that is 20 percent higher than the smallest

recorded imbalance for the most similar stored grid hierarchy is allowed. The amount of allowed performance deviation is determined by an analysis of the performance data distributions.

The focus concentrates the partitioning effort to the most performance-inhibiting factor, but the impact of the secondary factor should also be considered. Initially, all partitioning algorithms that historically performed well with respect to the most performance-inhibiting factor (i.e. equal to or better than the allowed performance deviation) are recorded. The historic performance of these candidate algorithms is evaluated again and ordered with respect to the secondary performance-inhibiting factor. Using this strategy, an algorithm that performed well for the most similar stored grid hierarchy and the most performance inhibiting factor will be selected, while the impact of the second factor is kept as low as possible. Pseudo-code for the algorithm selection is listed in Algorithm 2.

The selected partitioning algorithm is uniquely determined by the combination of the most similar stored grid hierarchy and the current partitioning focus. Because the partitioning focus is divided into a number of discrete levels, it is possible to pre-compute the algorithm selection for each combination of focus and stored grid hierarchy. For each partitioning focus, the selected algorithms and the corresponding grid hierarchies are stored as rules. Thus, during run-time, the algorithm selection is reduced to a simple table look-up for the rule that corresponds to the current partitioning focus.

Focus	**Corresponding rule**
FocusSynch1_25	Allowed synch 125% of minSynch
FocusSynch1_75	Allowed synch 175% of minSynch
FocusLB1_2	Allowed LB 120% of minLB
FocusLB1_5	Allowed LB 150% of minLB

Table 3: A number of sample partitioning focuses. Note the differences between `FocusSynch` and `FocusLB`. This is due to larger variations in the performance data for the synchronization penalty compared to the load imbalance. Only `FocusLB1_2` and `FocusSynch1_25` are used in this paper.

Part

One or more partitioning components, containing either a single partitioning algorithm or a complete partitioning framework, are connected to the `Core`-component. The hybrid partitioning framework Nature+Fable (see Section 2.1) is the only partitioner in the presented implementation of the Meta-Partitioner.

Algorithm 2 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 algorithms selected during step 1. In this paper, X=1.25 for `FocusSynch` and X=1.2 for `FocusLB`.

5 Experimental setup

In the performance evaluation, four real-world applications are used (see Section 5.1). For each application, the performance of the Meta-Partitioner is compared to both the best static hybrid algorithm from Nature+Fable and the average performance for all algorithms used in the evaluation.

The best static partitioning algorithm for each of the four applications was determined using the 768 hybrid partitioning algorithms in the performance data base. The average load imbalance and the average synchronization penalty were computed for each of the partitioning algorithms (see Section 5.2 for a description of the metrics). After the performance data for the current application had been excluded, the best static partitioning algorithm was selected with the same criteria that were used to construct the rules (see Section 4.1.1). Note that the best static algorithm is dependent on the current partitioning focus — a change in the partitioning focus generally results in the selection of another algorithm.

Theoretical performance results for the stability and accuracy of the matching process is also presented. To consistently select high-quality partitioning algorithms, an accurate matching of the current and the stored grid hierarchies is paramount. Often, the differences in the least square sum between the most similar grid hierarchies are small. Hence, a slight change in one of the seven metrics that characterize the geometrical properties of the grid hierarchy would almost certainly result in the selection of a different partitioning algorithm. While most of the partitioning algorithms that correspond to these similar states can be expected to perform well, some could result in a performance that is substantially better or worse than the selected partitioning algorithm. Thus, to draw valid conclusions about the performance of the Meta-Partitioner, the stability and accuracy of the matching process need to be evaluated. For these experiments, the theoretically derived performance of the ten partitioning algorithms that correspond to the ten most similar grid hierarchies is presented.

5.1 Applications

For the evaluation, four real-world applications from the Virtual Test Facility (VTF) is used. The VTF, developed at the California Institute of Technology, is a software environment for coupling solvers for compressible computational fluid dynamics (CFD) with solvers for computational solid dynamics (CSD) [10, 29]. The purpose of the VTF is to simulate highly coupled fluid-structure interaction problems. The used applications are restricted to the CFD domain of VTF, as the CSD solver is implemented with unstructured grids and the finite element method.

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

5.2 Methodology

Application execution trace files and simulations of the Berger-Colella SAMR algorithm [5] are used for the performance evaluation. A trace file completely describes the un-partitioned SAMR grid hierarchy for each time step. To obtain the trace files, real applications were executed using the SAMR framework AMROC on the ALC parallel computer at Lawrence Livermore National Laboratory (courtesy to Ralf Deiterding) [1, 30].

The partitioning results were obtained as follows. An un-partitioned grid hierarchy, originating from a trace file, is sent to the Meta-Partitioner. Once the grid hierarchy is received, the Meta-Partitioner functions exactly as if it is connected to a real-world SAMR framework. The Meta-Partitioner characterizes the current grid hierarchy, matches it with the stored grid hierarchies in the data base, and selects the partitioning algorithm that is predicted to result in the best performance with respect to the current partitioning focus (see Section 4.1.1). Only partitioning algorithms from the framework Nature+Fable are currently considered. Next, the selected partitioning algorithm is invoked and the grid hierarchy is partitioned. The partitioned grid hierarchy is returned to the component that represents the SAMR framework. Finally, the partitioned grid hierarchy is stored on disk and a new grid hierarchy is sent to the

Meta-Partitioner. The weights used during the least-square matching were always determined without the inclusion of performance data from the current application. For all experiments, the grid hierarchies were partitioned for 16 processors.

To evaluate the performance, the partitioned grid hierarchy is used as input to an SAMR simulator [8]. Rather than simulating a parallel computer, the simulator mimics the execution of the common Berger-Colella SAMR algorithm [5]. For each re-partitioning, the simulator computes metrics like the arithmetical load imbalance, communication costs, and the synchronization penalty. The computed performance metrics (see below) are independent of any computer characteristics.

For the accuracy and stability experiments, it is infeasible at each re-partitioning to invoke and evaluate the result from ten different partitioning algorithms. Instead, a slightly modified version of the Meta-Partitioner is employed. The modified version does not invoke the partitioning framework. At each re-partitioning, the modified version determines the ten most similar grid hierarchies and stores the partitioning algorithms that would have been selected for these hierarchies. Note that the data base already contains performance data for the candidate partitioning algorithms and the grid hierarchies. Thus, the would-be performance for each combination of application, time step, and partitioning algorithm can be extracted from the performance data base.

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

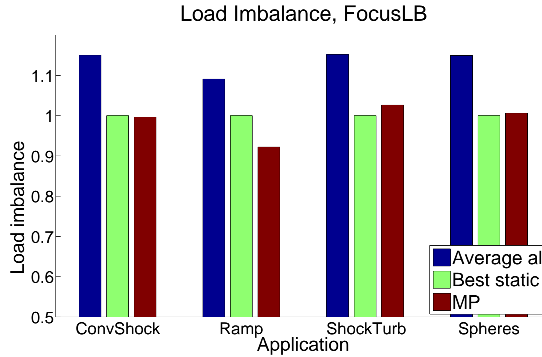
$$\text{Load imbalance (\%)} = 100 * \frac{\text{Max}\{\text{processor workload}\}}{\text{Average workload}} - 100.$$

When computing the load imbalance, the workload of the most overloaded processor is used.

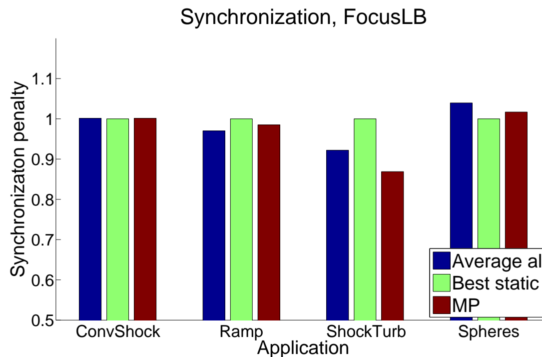
The synchronization penalty is computed as follows [25]. For each level, the processor checks their neighbors for the need to wait for any of them. If a processor needs to wait, the penalty is approximated by the number grid points that have to be updated by other processors before the stalled processor can resume its computations. The severity of the penalty is affected by how much work the stalled processor has left on higher refinement levels — stalling a processor with a great amount of work left is more serious than holding up a processor with little remaining work. Hence, the penalty is multiplied by the processor's remaining work.

6 Results

For the Meta-Partitioner to be meaningful, it is essential that it consistently selects partitioning algorithms with an equal to or better performance than both the best static partitioning algorithm and the average performance for all partitioning algorithms. During the analysis, more emphasis is given to the most performance-inhibiting factor. Thus, if `FocusSynch` is used, the synchronization penalty will take precedence over the load imbalance and vice versa. All results have been normalized with respect to the best static partitioning algorithm because this algorithm will generally result in better performance than a random algorithm. Note that it is impossible to compute the best static algorithm without access to performance data.



(a) Load imbalance



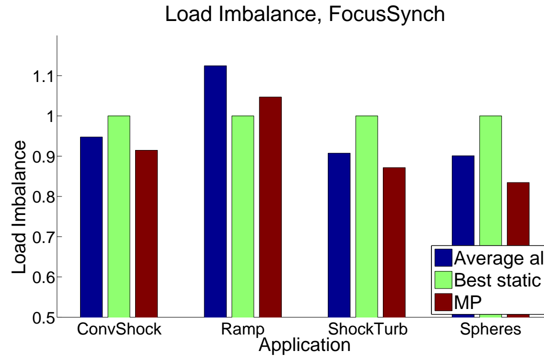
(b) Synchronization

Figure 2: Results for the Meta-Partitioner and the partitioning focus `FocusLB`. The Meta-Partitioner consistently generates partitionings with a smaller load imbalance than the average imbalance.

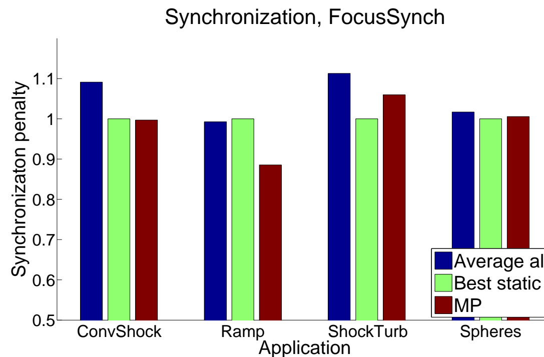
FocusLB

For `FocusLB`, the Meta-Partitioner consistently selected partitioning algorithms that resulted in a significantly smaller load imbalance than the average imbalance for all partitioning algorithms (see Figure 2). The improvements over the average load imbalance ranged from 10.9% to 15.5%. Compared to the best static partitioning algorithm, the load imbalance was decreased by 7.8% for the Ramp application. The three other applications resulted in approximately equal load imbalances for the Meta-Partitioner and the best static algorithm.

For three of the four applications, the synchronization penalty was similar to both the best static partitioning algorithm and the average for all partitioning algorithms. For the exception, ShockTurb, the Meta-Partitioner resulted in a 13.1% smaller synchronization penalty than the best static algorithm. Thus, focusing the partitioning effort on the load imbalance did not negatively affect the synchronization penalty.



(a) Load imbalance



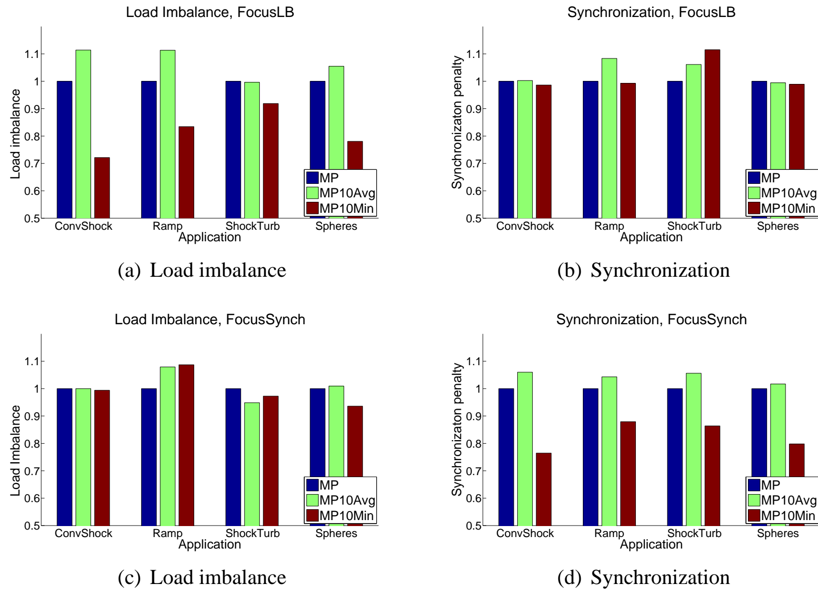
(b) Synchronization

Figure 3: Results for the Meta-Partitioner and partitioning focus `FocusSynch`. The Meta-Partitioner consistently generates partitionings with smaller synchronization penalties than the average penalty.

FocusSynch

For `FocusSynch`, the synchronization penalties resulting from the Meta-Partitioner were always smaller than the average penalty for all algorithms, ranging from 1.7% to 10.8% (see Figure 3). Comparing the results to the best static algorithm, the Meta-Partitioner resulted in a 11.5% decrease in the synchronization penalty for the Ramp applications while the penalty was increased by 6% for the ShockTurb application. The changes in the synchronization penalties for the ConvShock and Spheres applications were insignificant.

For all four applications, the load imbalance was reduced with on average 5.4% compared to the average imbalance for all algorithms. The Meta-Partitioner also produced a smaller load imbalance than the best static algorithm for three of the four algorithms, while a 5% increase was recorded for the Ramp application.



(a) Load imbalance (b) Synchronization

(c) Load imbalance (d) Synchronization

Figure 4: Results for the accuracy and stability analysis. MP1 corresponds to initial implementation and MP10 corresponds to the partitioning algorithms for the ten most similar grid hierarchies. Note the decrease in load imbalance for `FocusLB` and the smaller synchronization penalty for `FocusSynch`.

Stability and accuracy of the matching

For the most performance-inhibiting factor and both partitioning focuses, the average performance of the partitioning algorithms that correspond to the ten most similar states is consistently worse than the performance of the algorithm that is actually selected by the Meta-Partitioner (see Figure 4). The load imbalance was on average increased by 6.7% for `FocusLB` and the synchronization penalty was increased by 4.4% for `FocusSynch`. These results were expected since the ten matched grid hierarchies grow increasingly more different from the current grid hierarchy. However, the best performing partitioning algorithm among these ten grid hierarchies generally had a significantly better performance than the algorithm selected by the Meta-Partitioner. For `FocusLB`, the decrease in load imbalance ranged from 8.1% for the ShockTurb application to 27.9% for the ConvShock application. The decrease in the synchronization penalty for `FocusSynch` ranged from 13.6% for the ShockTurb to 23.5% for the ConvShock. The decrease seems proportional to the complexity of the grid hierarchy for both partitioning focuses.

For the secondary performance-inhibiting factor, the changes in performance is insignificant, especially when compared to the improvements in the most performance-inhibiting factor.

Result summary

For both partitioning focuses, the Meta-Partitioner resulted in consistent and significant improvements for the most performance-inhibiting factor compared to the average result for all partitioning algorithms. Compared to the best static partitioning algorithms, the improvements were smaller but generally still noticeable. Also, the Meta-Partitioner did not negatively affect the results for the secondary performance-inhibiting factor.

Based on both the practical experiments and the theoretical stability and accuracy evaluation, the assumption that similar grid hierarchies have similar partitioning properties is valid. For the most performance-inhibiting factor, the Meta-Partitioner always selects partitioning algorithms with better performance than the average of all algorithms. The performance is also generally equal to or better then the best static partitioning algorithm. The theoretical evaluation resulted in a slightly worse performance when more differing grid hierarchies are used to select the algorithm. The evaluation also showed that the matching of grid hierarchies is sensitive — at least one of the algorithms that correspond to the ten most similar hierarchies generally results in a significantly better performance than the algorithm that is actually invoked.

The theoretically derived analysis of the stability and accuracy of the matching process showed that large performance gains are possible if multiple partitioning algorithms are invoked and evaluated during run-time. While Meta-Partitioner can easily be modified to select and invoke multiple algorithms, it is currently intractable to evaluate the resulting partitionings during run-time due to the long execution time of the SAMR simulator.

7 Conclusions and Further work

In this paper, an initial implementation of the Meta-Partitioner for SAMR grid hierarchies was presented. At each re-partitioning, the Meta-Partitioner autonomously selects, configures, and invokes the partitioning algorithm that is predicted to result in the best performance. The implementation uses component-based software engineering and it is not restricted to any specific SAMR framework. To predict the performance of the candidate partitioning algorithms, the Meta-Partitioner uses historic performance data for grid hierarchies that are similar to the current hierarchy. The partitioning effort is focused on the performance-inhibiting factors with the largest impact on the execution time — either the load imbalance or the synchronization delays.

The performance evaluation of the Meta-Partitioner shows a small but noticeable performance increase compared to the best static algorithm. The Meta-Partitioner consistently generates partitionings with a significantly better performance compared to the average performance for a large number of partitioning algorithms. The selected partitioning algorithms did not result in a degraded performance for the secondary performance-inhibiting factor. The performance evaluation and an accuracy and stability analysis of the algorithm selection showed that similar grid hierarchies have similar partitioning properties. Finally, the performance of the Meta-Partitioner can be greatly improved if multiple algorithms are invoked and evaluated at each re-partitioning.

To use the Meta-Partitioner in real-world applications, several tasks remain. No SAMR framework have yet been interfaced to the Meta-Partitioner. While the performance evaluation showed that the Meta-Partitioner improves the performance of the most performance-inhibiting factor, real-world experiments should be performed to determine the impact on the execution time. These experiments might result in adjustments to the selection rules. The partitioning focus is currently constant during run-time. A focus that is changing in conjunction with the partitioning needs of the application and the state of the computer is desirable.

The initial implementation of the Meta-Partitioner has showed its ability to consistently reduce the impact of the most performance-inhibiting factor. Future versions of the Meta-Partitioner will certainly result in even larger reductions.

8 Acknowledgments

The author thanks Ralf Deiterding, Oak Ridge National Laboratory, for providing the application trace files. The author also thanks Johan Steensland and Jaideep Ray, Sandia National Laboratories, for valuable scientific collaboration.

References

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

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

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

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

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

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

[7] 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.

[8] 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.

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

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

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

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

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

[14] 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/.

[15] 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/.

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

[17] 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.

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

[19] 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.

[20] 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.

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

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

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

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

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

[26] 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.

[27] 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.

[28] 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.

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

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