A Performance Characterization of Load Balancing Algorithms for Parallel SAMR Applications

Henrik Johansson
Department of Information Technology
Uppsala University
Box 337, S-751 05 Uppsala, Sweden
email: henrik.johansson@it.uu.se

Johan Steensland
Advanced Software Research and Development
Sandia National Laboratories
P.O. Box 969, Livermore, CA 94550, USA
email: jsteens@sandia.gov

Abstract

We perform a comprehensive performance characterization of load balancing algorithms for parallel structured adaptive mesh refinement (SAMR) applications. Using SAMR, computational resources are dynamically concentrated to areas in need of a high accuracy. Because of the dynamic resource allocation, the workload must repeatedly be partitioned and distributed over the processors. For an efficient parallel SAMR implementation, the partitioning algorithm must be dynamically selected at run-time with regard to both the application and computer state. We characterize and compare a common partitioning algorithm and a large number of alternative partitioning algorithms. The results prove the viability of dynamic algorithm selection and show the benefits of using a large number of complementing partitioning algorithms.

1 Introduction

Structured adaptive mesh refinement (SAMR) is used to decrease the run-time of simulations in areas like computational fluid dynamics [4, 8], numerical relativity [7, 11], astrophysics [5, 17], and hydrodynamics [16]. 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. This procedure results in a dynamic and adaptive grid hierarchy that conforms to the maximum acceptable error.

Because grid patches are created, moved and deleted during run-time, the dynamic grid hierarchy is repeatedly repartitioned and redistributed over the 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 movement between partitions and the performance and utilization of the interconnect. Thus, to minimize the run-time, the current state of the application and the hardware must both be taken into account. This is non-trivial, since the basic conditions for how to allocate hardware resources change dramatically during run-time, due to the dynamics inherent in both the applications and the computer system.

In previous work we proposed the development of a meta-partitioner [12, 24, 28], which should autonomously select, configure, and invoke the best partitioning algorithm with regard to the current application and computer state. For the selection process, the meta-partitioner would have access to a large data base with thoroughly characterized partitioning algorithms.

In this paper we present such a performance characterization of SAMR load balancing algorithms. We use four real-world applications and, on a time-step basis, compare a common partitioning algorithm to a large number of alternative partitioning algorithms. The alternative algorithms can be altered with respect to the current state of the SAMR application and computer system.

The results show the benefits of having a collection of diverse partitioning algorithms to choose from. Even though the best-performing partitioning algorithm vary when the application state vary, there exist good-performing partitioning algorithms for each application and time step. As the combined range of possible application and computer states in theory is infinite, having complementing partitioning algorithms is essential. With sufficiently many complementing algorithms, there are good-performing algorithms for most states.

We see two main approaches for the meta-partitioner to select the partitioning algorithm. The first approach is to construct heuristic rules. The second approach is to use historical performance data. Regardless of approach, we will need a data base with performance data from thoroughly characterized partitioning algorithms. The performance data gathered in this work proves the viability of the metapartitioner and will form a solid basis for such a data base. Using the data base, good performing partitioning algorithms can consistently be selected.

2 Structured adaptive mesh refinement

For PDE solvers based on finite differences and structured grids, solution accuracy and run-time are higly dependent on grid resolution. A higher resolution generally results in a higher accuracy[1] 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 are a waste of computational resources. By increasing the grid grid resolution in critical areas, the run-time of these PDE solvers can be decreased

The common Berger-Colella SAMR algorithm [4] 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 treshold are flagged for refinement. Flagged points are clustered and overlaid with logically rectangular patches of finer and uniform

[1]Higher accuracy can also be achieved with higher order methods.

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 is 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 inner parts of the computational domain. Only a small fraction of the patches can generally use the physical boundary conditions. After integration, the results are projected down from finer to coarser levels. As refined patches use smaller time steps, updating coarser level solutions increases the accuracy. Thus, data flows both over the borders of neighboring patches and between patches on different refinement levels.

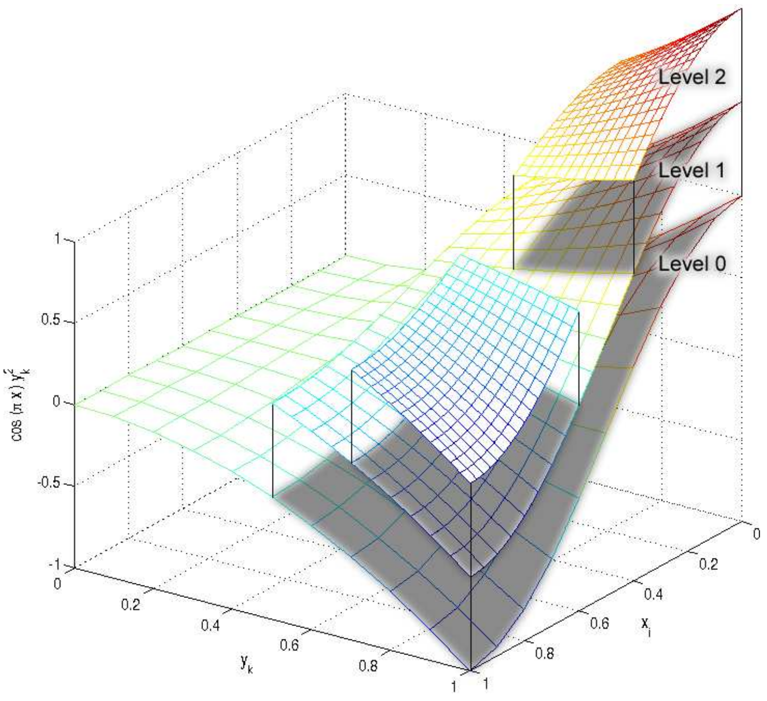


Figure 1: Example of a grid hierarchy with two levels of refinement. The grids are skewed to reflect the characterstics of a solution.

3 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 data 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 effiency.

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. Both the arithmetical load imbalance and the order in which the patches are processed can cause delays. Synchronization delays can be severe [27] and are hard to predict.

To get optimal performance, the partitioner needs to simultaneously minimize data migration, load imbalance, communication volumes, and synchronization delays. It is unrealistic to search for the optimal solution [10]. Instead, the partitioner needs to trade-off the metrics in accordance with the characteristics of the application and computer. Ultimately, partition quality is determined by the resulting application execution time.

Algorithms for partitioning SAMR hierarchies can be categorized as domain-based, patch-based, or hybrid. For *patch-based partitioners* (PB) [3, 14, 21], the distribution decisions are made independently for each patch (or refinement level). The SAMR framework SAMRAI [32, 33] fully supports PB. *Domain-based partitioners* (DB) [18, 20, 24, 30] partition the physical domain, rather than the actual grid patches. The domain is partitioned along with all contained grids on all refinement levels. Domain-based partitioners can be found in the AMROC [2, 8] and GrACE [19] frameworks. *Hybrid partitioners* (HP) [13, 18, 30] combine the patch-based and domain-based approaches. In the hybrid partitioning framework Nature+Fable [23], areas suitable for either domain-based or patch-based algorithms are separated from each other. By using different techniques for different grid parts, the partition quality can be improved.

3.1 Distributing grid patches

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 approach is to use a bin-packing algorithm [3, 19, 33] to distribute the patches. For bin-packing to be effective, large patches may have to be divided. Regardless of method, the partitioner can use either a patch-by-patch or a level-by-level approach.

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

For the domain-based approach, only the base grid is partitioned. Normally, the workload of the refined patches is projected down onto the base grid, reducing the problem to the partitioning of a single grid with heterogenous workload.

To achieve less load imbalance, the base-grid can be partitioned into more blocks than processors. Having more blocks can make it easier to find an even distribution. The minimum block size is determined by the size of the computational stencil on the base grid. Because the base grid stencil corresponds to many grid points on highly refined patches, the workload of a minimum sized block can be large.

Because overlaid grid blocks reside on the same processor for domain-based algorithms, inter-level communication is eliminated. A complete re-partition might be necessary when the grid hierarchy is modified. The computational stencil and base grid resolution impose restrictions on subdivisions of higher level patches, often resulting in high load imbalances for deep grid hierarchies. Another problem with domain-based algorithms is "bad cuts": many and small blocks with bad aspect ratios. These blocks are created 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.

Both patch-based and domain-based algorithms perform well under suitable conditions, especially for simple and shallow grid hierarchies [22, 28]. Unfortunately, their shortcomings often make their performance unacceptable for deep and complex hierarchies [22, 23]. As a remedy, a hybrid approach can be used. By combining strategies from both domain-based and patch-based algorithms, it is possible to design a partitioner that performs well under a wider range of conditions.

Key concepts in the hybrid partitioning framework used in this work (Nature+Fable [23]) are separation of refined and unrefined areas of the grid and clustering of refinement levels. 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 techniques are used for all parts of the grid that are not included in bi-levels.

Using the hybrid partitioning algorithms adopted in this work, less load imbalances than for domain-based algorithms can be achieved because patches from at most two refinement levels are partitioned together. Because inter-level communication only exist between bi-levels, communication volumes are generally smaller for the hybrid algorithms than for patch-based algorithms.

3.2 Towards the meta-partitioner

No single partitioning algorithm is the best choice for all application and computer states [23]. We therefore proposed the meta-partitioner [12, 24, 28], which autonomously selects, configures, and invokes a good-performing partitioning algorithm. The meta-partitioner has access to a variety of complementing partitioning algorithms from all approaches. At re-partitioning, the meta-partitioner evaluates the current state of the application and computer system and selects a suitable partitioning algorithm [25, 26].

We see two approaches to select the partitioning algorithm. The first approach uses heuristic rules, constructed from partitioning algorithm performance data. By combining the rules with application and computer states, a suitable partitioning algorithm can be selected.

The second approach uses historical performance data together with stored application and computer states. During run-time, the current application state is matched against historical data to find and extract the most similiar state. A partitioning algorithm is then selected based on the current computer state and historical performance data for the extracted state.

Regardless of how the meta-partitioner selects the partitioning algorithm, we need to thoroughly characterize all candidate algorithms. This work presents such a characterization. For the rule-based approach, we will need the characterization for the construction of the heuristic rules. For the historical data approach, we will use the characterizations during run-time.

For the characterization, we use four real-world applications. The applications represent a wide range of application states and they are partitoned by a large number of complementing partitioning algorithms. We analyze the performance of the algorithms and assess their strengths and weaknesses. The collected data are stored in a data base suitable for future research and for use in the meta-partitioner.

4 Applications

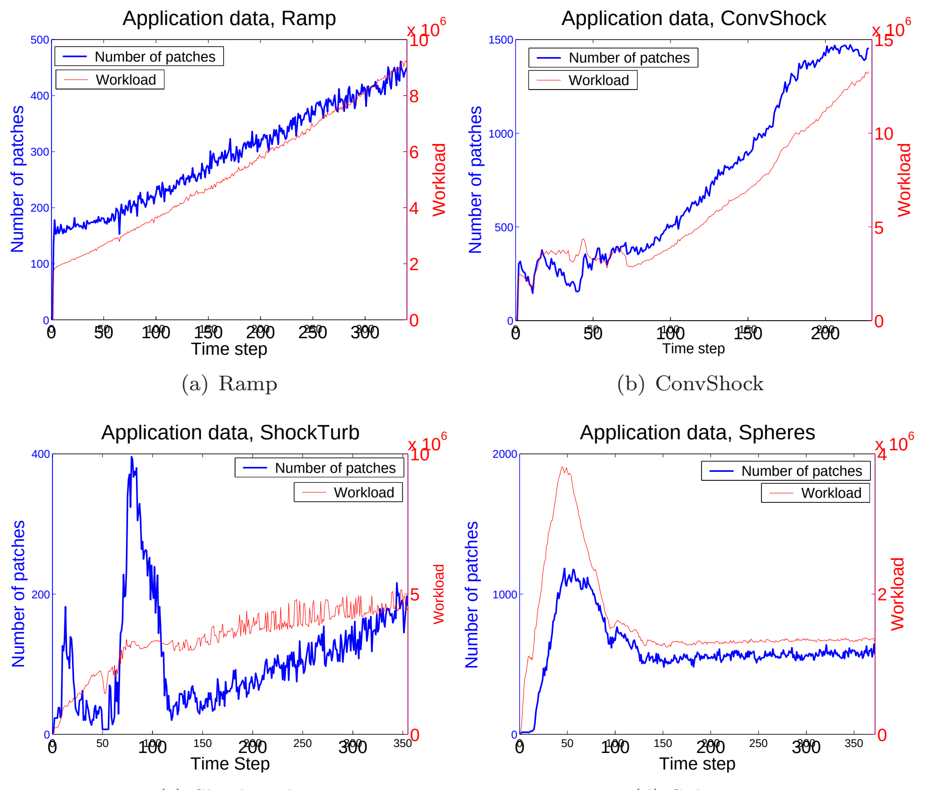
We use four applications from the Virtual Test Facility (VTF). The VTF, developed at the California Institute of Technology, is a software environment for coupling solvers for compressible computational fluid dynamics (CFD) with solvers for computational solid dynamics (CSD) [8, 29]. The purpose of VTF is to simulate highly coupled fluid-structure interaction problems. The selected applications are restricted to the CFD domain of VTF, as the CSD solver is implemented with unstructured grids and the finite element method.

For the CFD problems in VTF, the framework AMROC [2, 9] is used. AMROC is an object-oriented SAMR framework that conforms to the algorithm formulated by Berger and Colella [4]. AMROC is based on DAGH (Distributed Adaptive Grid Hierarchies), a data storage package for parallel grid hierarchies [19]. In AMROC, separate grid levels are allowed to use different degrees of refinement.

Below we present the applications used. A more comprehensive description can be found in the AMROC wiki [2]. The term "refinement factor" describes the degree of refinement with respect to the next lower refinement level. As an example, assume a 2D application with two levels of refinement and refinement factors {2,4}. The resolution on the first refinement level is twice as high as on the base grid. The second refinement level has a resolution four times higher than the first refinement level. Thus, for the example above, the maximum patch resolution is 64 times higher than for the base grid $((2 * 2) * (4 * 4))$. Because we generally also refine in time, several iterations are performed on a refined patch during one time step on the coarsest level. Using our example and assuming equal refinement in both space and time, we will perform two iterations for each patch on the first refinement level and eight iterations for the patches on the second level. The refinement factors are set by the user before the application is executed.

The metric *workload* measures the aggregate number of grid points calculated on during a time step on the coarsest level. Because of the refinement in time, a refined grid has a larger workload than the number of grid points. The workload presented in Table 4 is the aggregate workload for all patches and time steps.

A scattered refinement pattern have its patches spread over a large part of the computational domain and the patches are generally unconnected. When the pattern is unscattered, the patches are connected and located close to each other. If a pattern is sharp, the size of overlaid refined areas are roughly equal. In a fuzzy pattern, the overlaid refined area gets much smaller for each level of refinement.



(a) Ramp (b) ConvShock

(c) ShockTurb (d) Spheres

Figure 2: Workload and the number of grid patches for the four test applications.

Application	Initial grid size	Levels of refinement	Refinement factors	Max grid size	Total workload
Ramp	480x240	3	{2,2,4}	722,924	$1.78 * 10^9$
ShockTurb	240x120	3	{2,2,2}	787,076	$1.21 * 10^9$
ConvShock	200x200	4	{2,2,4,2}	695,244	$1.42 * 10^9$
Spheres	200x160	3	{2,2,2}	689,688	$0.60 * 10^9$

Table 1: Application data. The maximum grid size denotes the number of grid points at the time step when the grid was at its largest. The total workload also considers refinement in time.

4.1 Ramp — Mach reflection at a wedge

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. This problem was also used by Berger and Colella [4].

The initial grid size is 480x120 grid points and the application uses three levels of refinement with refinement factors {2,2,4} (see Table 4). The maximum number of grid points is 722,924. A density plot for time t=0.2 is shown in Figure 3.

Both the workload and the number of grid patches grow almost linearly during execution (see Figure 2a), due to the growing reflection pattern behind the shock wave. Both the incident and the reflected shockwave have a sharp and unscattered refinement pattern.

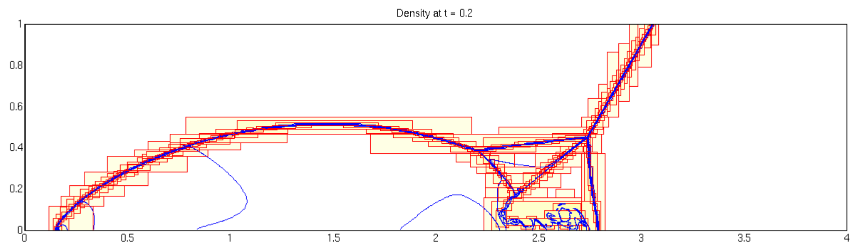


Figure 3: Density plot for Ramp at time t=0.2. The boxes correspond to grid patches.

4.2 ShockTurb — Planar Richtmyer-Meshkow instability

ShockTurb treats the interaction of two contacting gases with different densities. When the gases are subject to a shock wave, the interface between them becomes unstable and the result is called a Richtmyer-Meshkov instability. The Richtmyer-Meshkov instability finds applications in stellar evolution and supernova collapse, pressure wave interaction with flame fronts, supersonic and hypersonic combustion and in intertial confinement fusion.

In the simulation, an incident Mach 10 shock wave causes vortices along a sinusoidally perturbed gas interface (five symmetric pertubations). The geometry is rectangular and closed, except at the left-most end. The gases are air and SF_6 (sulfur and fluoride). The simulation is motivated by physical experiments [31].

The initial grid size is 240x120 grid points and and the application uses three levels of refinement with a constant refinement factor of two (see Table 4). The maximum number of grid points is 787,076. A density plot for time t=0.5 is shown in Figure 4.

Studying Figure 2c, we notice two sharp increases in the number of grid patches. The first, and smaller one, occurs at time step 15 when the shock wave hits the gas interface. Many small grid patches are created at this time. Immediately, a large portion of these smaller patches either grow slightly or are moved a little, causing them to merge.

The shock wave is reflected when it reaches the far wall. At approximately time step 70, the shock wave again passes through the interface. The area between the interface and the wall can at this time be subdivided into two parts. The part closest to the wall is subject to heavy turbulence. Even though the area is large and highly refined, only a small number of patches are needed to cover this domain as it is unscattered, homogenous and rectangular in shape. The majority of the computional work occurs in this part of the domain. The area closer to the interface is also turbulent, but not as much. Here, it is unnecessary to use patches from the highest refinement level. Since the area is scattered, many patches are created. These patches constitute the second large increase in the number of patches. Because the grid is not refined to the highest level, these patches do not significantly increase the workload. Also, the turbulence in this area quickly fades away, and the number of patches quickly decreases.

During the later portion of the execution, the number of patches is increasing as the turbulent area is slowly getting larger. However, the increase is scattered and not in need of the highest possible resolution. Thus, we again get an increase in the number of patches while the workload is largely unaffected.

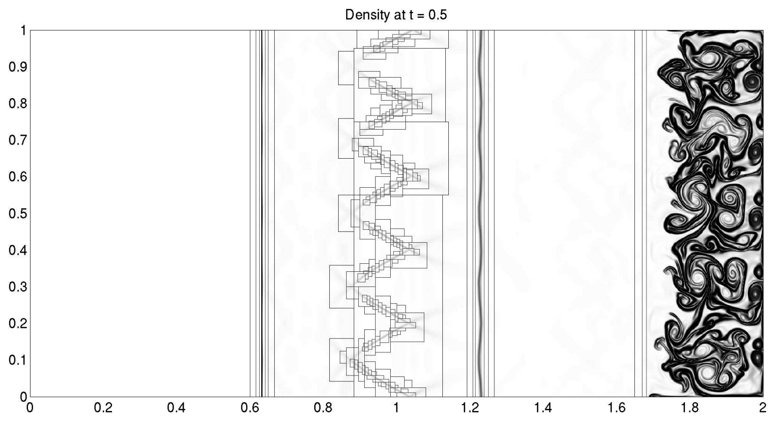


Figure 4: Density plot for ShockTurb at time t=0.5, just before the shockwave reaches the gaseous interface (at x=1) the second time. Only a small number of patches is used for the refinement of the rightmost region. The boxes correspond to grid patches.

4.3 ConvShock — Converging/diverging Richtmyer-Meshkov instability

ConvShock simulates a Richtmyer-Meshkov instability in a spherical setting. The gaseus interface is spherical and sinusoidal in shape. The interface is disturbed by a Mach 5 spherical and converging shock wave. The shockwave is reflected at the origin and drives a Richtmyer-Meshkov instability with reshock from the apex.

The initial grid size is 200x200 grid points and the application uses four levels of refinement with refinement factors {2,2,4,2} (see Table 4). The maximum number of grid points is 695,244. A density plot for time t=0.5 is shown in Figure 5.

During the first 50 time steps we see two distinct dips in the number of patches (see Figure 2). The first one is caused by the incident shockwave — as it is converging towards origo, its size is decreasing. At approximately time step 10 the shockwave hits the fluid interface, causing both turbulence and a higher level of refinement. When the shock wave continues towards origo, the turbulent area can be covered by fewer and larger blocks. At time step 50, the shock wave once again passes the interface. Since the shock wave is diverging, it continues to grow during the remainder of the execution.

The workload of the application correlates to the number of patches, except during the period from time step 30 to time step 50. During this interval, many small patches are merged, as described above. The refined area and the resulting workload is roughly constant, but the refined area is gradually covered by a lower number of patches. The refinement pattern is sharp and unscattered during the entire execution of the application.

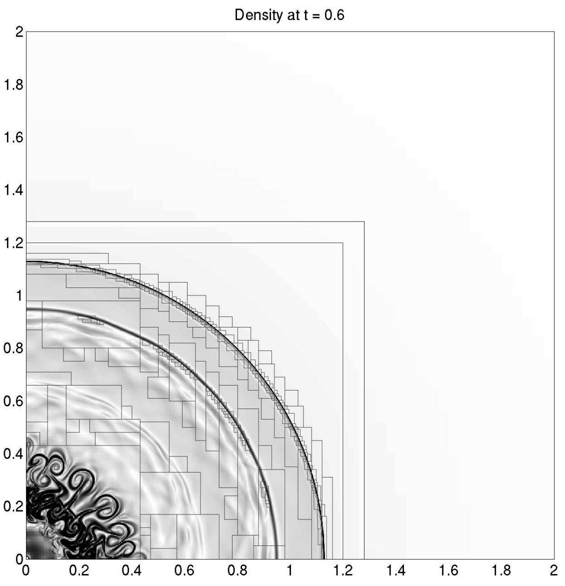


Figure 5: Density plot for ConvShock at time t=0.6 (detail). The rightmost feature is the fluid interface and slightly closer to origo is the divering shockwave. Note the heavy turbulence at orgio and the low number of grids covering it.

4.4 Spheres — Cylinders in hypersonic flow

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 cylinders. This is a realistic 4flow problem with complex boundaries.

The initial grid size is 200x160 grid points and the application uses three levels of refinement with a constant refinement factor of two (see Table 4). The maximum number of grid points is 689,688. A density plot for time t=3 is shown in Figure 6.

Both the workload and the number of patches increase sharply during the first 50 time steps (see Figure 2). They then decrease until time step 125, where they stabilize and remain constant for the duration of the execution. The increase in patches occurs when the incident flow starts to hit the two spheres. Heavy turbulence is present until bow shocks begin to form behind the spheres. The turbulence is decreased when the bow shocks become stable, attributing to the decrease in both the workload and the number of patches. When the bow shocks are fully formed and stable, the workload and the number of patches are constant. During the first 50 time steps, the refinement pattern is growing, scattered and fuzzy. When the bow shocks starts to form, the pattern gradually becomes smaller and less scattered. During the latter part of the execution, the refinement pattern is sharp and unscattered.

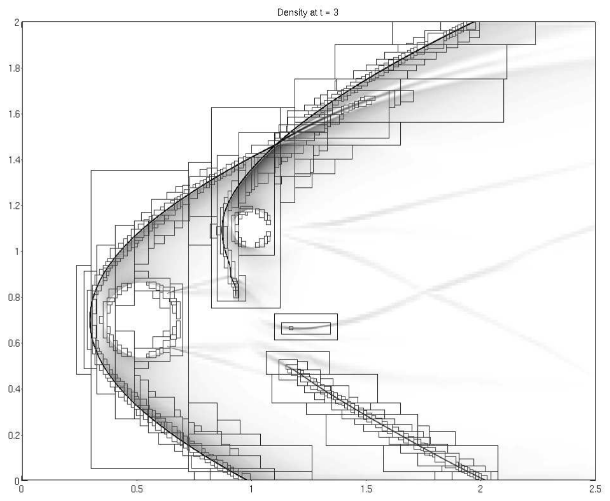


Figure 6: Density plot for Spheres at time t=3. The bow shocks are fully formed.

5 Methodology

Our ultimate goal is the meta-partitioner, which autonomously selects, configures and invokes a good-performing partitioning algorithm with respect to the current state of the application and computer. To achieve good performance, the meta-partitioner needs access to performance data from thoroughly characterized partitioning algorithms. In this work, we characterize partitioning algorithms from the domain-based and the hybrid approach. We use the domain-based partitioner from the SAMR framework AMROC and a large number of hybrid algorithms from the partitioning framework Nature+Fable [23].

AMROC uses a domain-based approach for load balancing. When the workload has been projected down onto the base grid, the grid is subdivided into smaller blocks. A Hilbert space-filling curve (SFC) dictates the orientation of the cuts, effectively creating an ordered, one-dimensional list of blocks. The list is cut into p segments of roughly equal workload. The segments are linearly mapped onto processor 1 to p. The same processor ordering is used at each re-partition. The locality preserving property of the SFC benefits the partitioning in two ways. First, adjacent blocks in the hierarchy are likely to be kept together, decreasing communication costs. Second, as the SFC dictates how the segments are mapped onto processors, small grid perurbations typically lead to small data migration costs.

Nature+Fable is a hybrid partitioning framework designed to produce high-quality partitions during a wide range of conditions. The partitioning process within Nature+Fable is governed by a large set of parameters. In this work, each parameter combination is regarded as a separate partitioning algorithm. In total, almost 900 parameter combinations were used, corresponding to as many hybrid algorithms. The main features of hybrid partitioners are described in Section 3.1.

Due to the vast number of hybrid algorithms, it is impossible to perform real-world executions for each combination of partitioning algorithm and application. Instead, we use application execution trace files and simulations of the Berger-Colella SAMR algorithm [4]. A trace file completely describes an un-partitioned grid hierarchy. The trace files were obtained from real executions on the ALC parallel computer at Lawrence Livermore National Laboratory [1]. The grid hierarchies were partitioned for 16 processors, and the partitioning algorithm was fixed during each simulation.

The partitioned grid hierarchies were used as input to an SAMR simulator [6]. Rather than simulating a parallel computer, the simulator mimics the execution of the Berger-Colella SAMR algorithm. For each time step, the simulator calculates metrics like arithmetical load imbalance and communication volume. The metrics (see Section 6) are independent of computer characteristics.

6 Performance metrics

Below we define the metrics used in this work.

Load imbalance

Arithmetic load imbalance is a common metric for judging the quality of a partition. We define load imbalance as follows:

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

Since we generally also refine in time, the workload (see Section 4) is not equal to the total grid size. We use the workload of the most loaded processor, as all processors must finish their computations before the solution can be advanced to the next time step.

Predicted computational time

This metric shows the predicted impact of load imbalance on the computational time. For each application, we divide the average real-world computational time from the ALC parallel computer with the aggregate workload. The result correponds to the computational time for a single grid point. For each time step, the derived computational time is multiplied with the workload of the most loaded processor. The outcome is a prediction of the computational time for each time step.

The real-world computation time for a grid point is determined by many factors that are out of our control, e.g. cache misses and other system effects. However, we do not use the predicted time for (and neither are we deriving it from) a small number of grid points. We use it to estimate the aggregate computational time for millions of grid points. Because of the large number of grid points, variations in computational time for individual grid points are evened out.

Configurations resulting in a low load imbalance will always perform better than ones with a higher imbalance, but the differences in computational time also depend on the characteristics of the application. For huge grid hierarchies or computationally intensive applications, the impact of the load imbalance will be large.

Application	Computational time (s)	Workload/ processor (grid points)	Time/grid point (μs)
Ramp	1381.2	$1.11 * 10^8$	12.42
ShockTurb	2618.4	$7.60 * 10^7$	34.45
ConvShock	1810.7	$8.87 * 10^7$	20.42
Spheres	1843.5	$3.73 * 10^7$	48.86

Table 2: Computational time per grid point, derived from actual executions with sixteen processors. The used computational time is the average time per processor. The workload is the sum of the gridpoints calculated on, taken over all time steps.

Number of blocks

To achieve low load imbalances, grid patches are often divided into smaller blocks. Using patches that are split into many parts generally results in larger faces between the blocks, inducing more communications. Having many blocks also result in other types of overhead, e.g. larger start-up costs, more "book-keeping" and often higher cache miss rates. For the number of blocks metric, we compute the maximum number of blocks assigned to any processor.

Number of communications

Using network performance data from the same type of network as employed in the ALC computer [15] data, we found that the time spend in communication was short. Instead of long communication times, performance was limited by synchronization delays. The time spent waiting for data was of the same magnitude as the computational time. For our applications, synchronization reductions are more benefical than decreases in communication volumes.

Using the simulator, it is impossible to measure synchronization delays without complex parallel execution models of the applications. However, each time processors need to exchange data, a synchronization delay can occur. By reducing the number of communications, the probablity of delays will be reduced. Given the many delays recorded on the ALC computer, a reduction will have substantial impact on application run-times.

Application	Computational time (s)	Synchronization time (s)
Ramp	1381.2	808.6
ShockTurb	2618.4	562.1
ConvShock	1810.7	2262.4
Spheres	1843.5	2585.1

Table 3: Comparision between computational and synchronization time. The numbers are from actual sixteen processor runs.

As a performance metric, we therefore include the total number of communications during a time step. Communications between two processors, in the *same* direction (send or receive) *and* on the same level *or* between the same levels, are assumed to be packed. Summing the packed communications yields a measurement of the total number of communications. This metric is an approximation of the communication volume, but we believe it captures the general behavior of the number of communications.

Data migration

Data migrates between processors as a consequence of repartitioning. Data is migrated from old partitions to new partitions in accordance with the new partitioning. The metric data migration is defined as the number of data points moved from one processor to another. Thus, this metric captures the partitioner's ability to consider the existing partitioning.

In this paper, we characterize almost 900 partitioning algorithms. Because data migration requires that we consider the partitionings at two consequtive time-steps, for each time-step and application it is necessary to analyze the transitions between about 900^2 partitioned grid hierarchies. The resources required to efficiently store and process this information are beyond the capacity of our current hardware and software tools. Hence — and despite the relevance — including data migration is considered future work.

7 Performance results

The trace files from the four applications were partitioned for sixten processors by the domain-based partitioner in AMROC and by almost 900 hybrid partitioning algorithms from Nature+Fable. The resulting partitions were used as input to the SAMR simulator, as described in Section 5.

For each metric *and* time step, the results for the domain-based partitioner and the best hybrid partitioning algorithm are presented. We use the best hybrid algorithm as an illustration of the performance that can be achieved with the meta-partitioner. We also present average values for each metric in Section 7.5.

7.1 Load imbalance

The domain-based algorithm produced low load imbalances for ShockTurb and, with one exception, for Spheres. In the early stage of Spheres, a small, scattered and fuzzy refinement pattern consisting of few patches (see Section 4.4) resulted in high imbalances. However, these imbalances decreased as a result of the growing refinement pattern. From time step 40 and on, a larger and sharper refinement pattern resulted in low imbalances. For ShockTurb, the majority of the computational work occurs at the rightmost part of the domain, where the refinement pattern is large, rectangular and unscattered. The domain-based algorithm is able to use this refinement pattern to produce a low load imbalance.

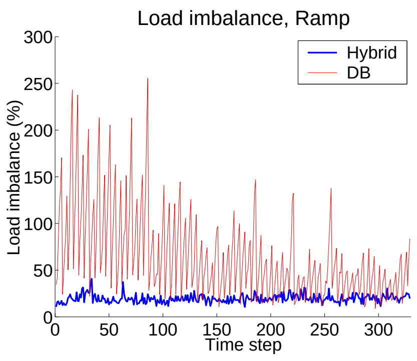
For Ramp and ConvShock, the domain-based algorithm produced wildly oscillating imbalances, generally above 50 percent and often significantly higher. Because both applications have deep grid hierarchies with sharp and relatively small refinements, these applications amplified the inherent shortcomings of domain-based algorithms (see Figure 7). However, imbalances were decreasing as a result of growing refinement patterns and an increasing number of patches. This relation between the refinement pattern and the load imbalance was similar to that observed for Spheres. For all these three applications, a decrease in load imbalances could be connected to a larger refinement pattern. We conclude that the domain-based algorithm generally produces lower imbalances for large and unscattered refinement patterns.

The hybrid algorithms generally produced low and stable load imbalances. The lowest imbalances were achieved for small (e.g. ConvShock) or scattered and fuzzy (e.g. the beginning of Spheres) refinement patterns. For the later stages of ShockTurb, where the refinement pattern is large and homogenous, the hybrid algorithms produced relatively high and oscillating load imbalances. This contrasts the low imbalances produced in the early stages, where the refinement pattern is more scattered (see Section 4.2). We conclude that the hybrid algorithms generally produce low load imbalances for sharp and scattered refinement patterns but have problems with large and homogenous patterns.

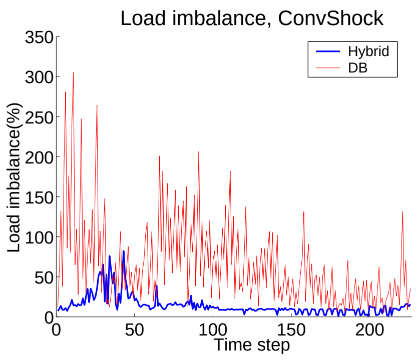
7.2 Predicted computational time

The difference in predicted computational time between the domain-based and the hybrid algorithms was large for Ramp and ConvShock (see Figure 8). The low and stable load imbalance for the hybrid algorithms resulted in superior predicted computational times. For the domain-based algorithm, the high and oscillating load imbalance caused large and fluctuating predicted computational times. For both applications, the gradual increase of workload resulted in an increase in predicted computation time as well.

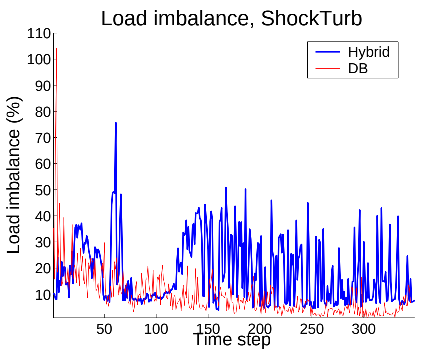
Only for ShockTurb did the domain-based algorithm achieve shorter predicted computational times than the hybrid algorithms. The high and oscillating load imbalance for the hybrid algorithms resulted in longer and fluctating computational times.



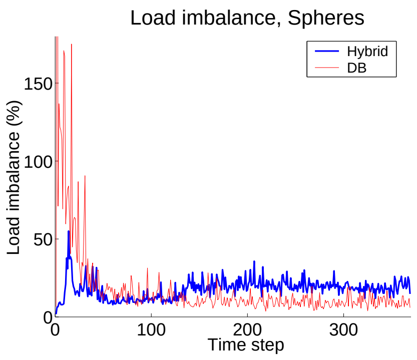
(a) Ramp



(b) ConvShock



(c) ShockTurb



(d) Spheres

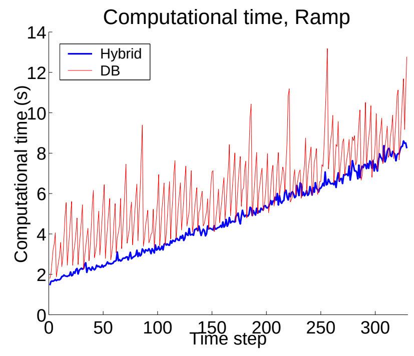
Figure 7: Load imbalance.

Even though the domain-based algorithm achieved a lower load imbalance than the hybrid algorithms for 300 out of 350 time-steps, the total predicted computational time was slightly shorter for the hybrid algorithms (see Table 4). The initial combination of high load imbalance and a large workload for the domain-based algorithm was only marginally compensated during the main part of the execution, showing the importance of targeting the load imbalance for large workloads.

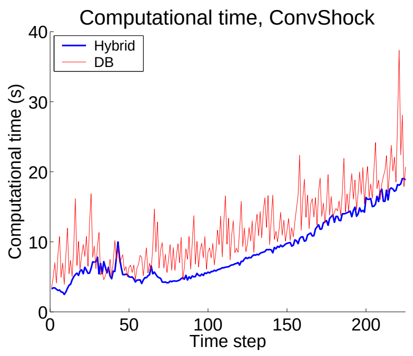
7.3 Number of blocks

The hybrid algorithms consistently produced partitions with a small number of blocks for all applications. The number of blocks closely followed the behavior of the number of patches (see Figure 2 and 9) but seemed to be unaffected by any other characteristics of the refinement pattern.

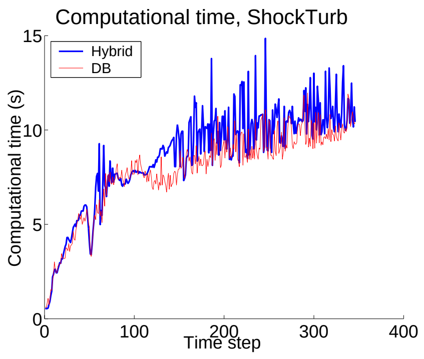
The number of blocks was larger for the domain-based algorithm than for the hybrid algorithms. The difference between the algorithms was smaller during the beginning of Spheres and for time steps 60 to 110 for ShockTurb. For both applications, the refinement patterns are scattered with small and unconnected patches. For these refinement patterns, the potential difference between the algorithms are reduced as fewer patches typically need to be subdivided to balance the load.



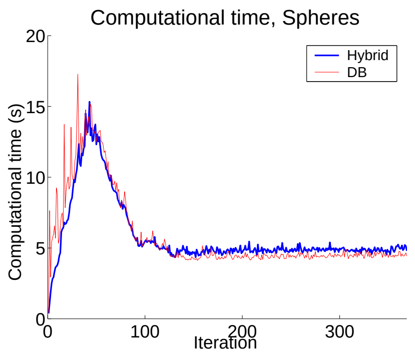
(a) Ramp



(b) ConvShock



(c) ShockTurb



(d) Spheres

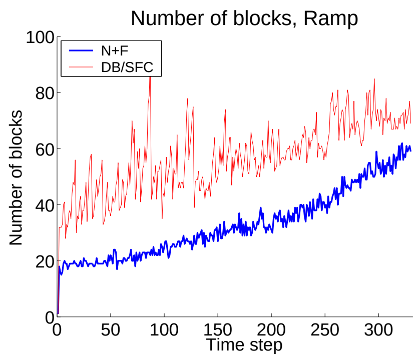
Figure 8: Computational time.

For ConvShock, a sharp increase in the number of blocks occured when the shock wave passes through the gas interface the second time (see Section 4.3). After the increase, the refinement pattern is unscattered and sharp and the patches are small. Together with the restrictions on where the patches can be subdivided (see Section 3.1), the refinement pattern and the small patches forced the domain-based algorithm to create many blocks.

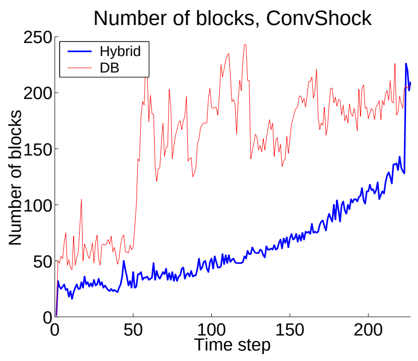
7.4 Number of communications

The hybrid algorithms resulted in small numbers of communication (see Figure 10). The only exception was Spheres, but the number of communications cannot be regarded as large. Because the hybrid algorithms created few blocks (see Figure 9), they were expected to produce less communication than the domain-based algorithm, but we were surprised to find the numbers this small. If the meta-partitioner can select the algorithms that result in these small numbers of communications, the impact on the execution time will be large.

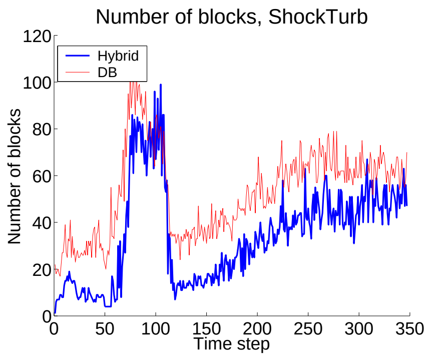
The number of communications was significantly larger for the domain-based algorithm than for the hybrid algorithms. For both types of algorithms, there is a correlation between the amount of communication and the number of blocks as fewer blocks generally resulted in less communication. The only exception was ConvShock, where the domain-based algorithm initially produced a large



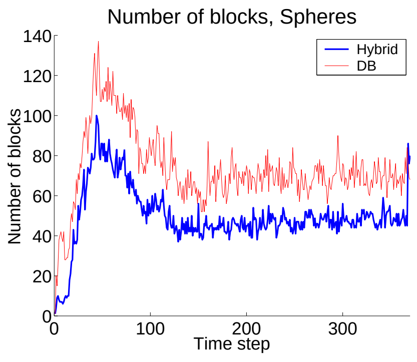
(a) Ramp



(b) ConvShock



(c) ShockTurb



(d) Spheres

Figure 9: Number of blocks.

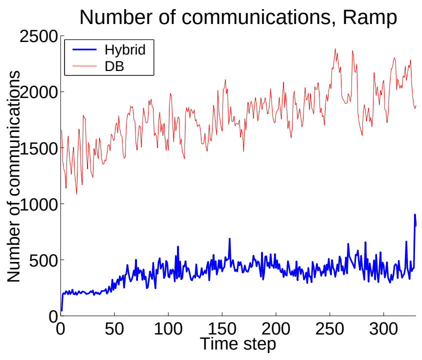
number of communications even though the number of blocks was small. In the beginning of ConvShock, the refinments is concentrated to origo and each block has many neighbors. This refinement pattern created a large number of communications. Later, the majority of the computations are performed on the divering shock wave. Since the width of the shock wave is small, the blocks only need to communicate with a small number of neighbors. However, the number of communications remained realtively stable as the smaller communication need was balanced by an increase in the number of blocks (see Figure 9).

For ShockTurb, the impact of the large number of blocks present during time steps 60 up to 110 was minimal for both appoaches. This is due to the low average workload of the patches during the interval (see Section 4.2). For small workloads, several blocks can often be assigned to a single processor, reducing the need for communication.

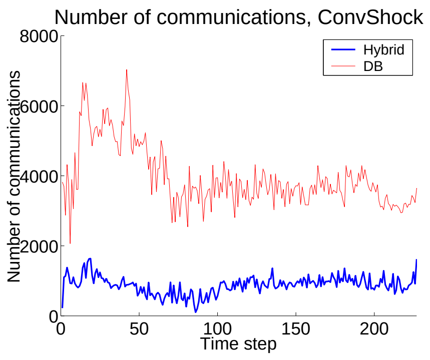
7.5 Average performance results

Above, we presented the performance results on a time step basis. In this section we discuss the average and aggregate performance of the algorithms, computed for entire executions. We also include average performance data for all hybrid algorithms, not just for the best performing algorithm.

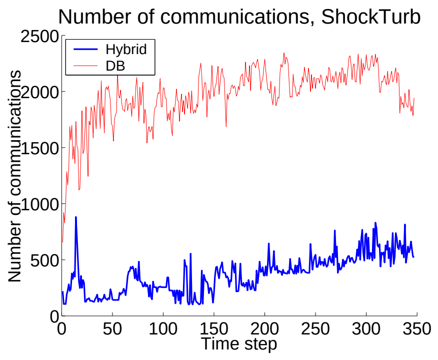
For both Ramp and ConvShock, the hybrid algorithms produced a superior



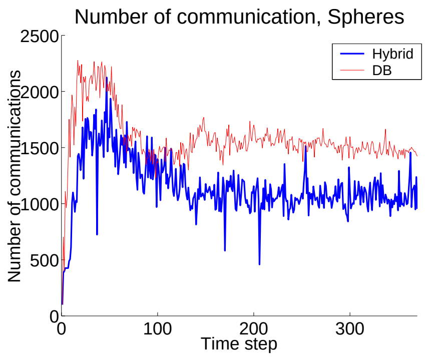
(a) Ramp



(b) ConvShock



(c) ShockTurb



(d) Spheres

Figure 10: Number of communications.

load imbalance. Here, even the average performing hybrid algorithm performed better than the domain-based algorithm. For ShockTurb, the domain-based algorithm had a significantly lower load imbalance than the best hybrid algorithm. For Spheres, the performance of best hybrid algorithm and the domain-based algorithm was roughly equal.

The average results for the predicted computational times are heavily dependent on the load imbalance. For the Spheres, the domain-based algorithm performed better in 300 out of the 350 time steps. Because of high load imbalances for large workloads, the domain-based algorithm was still outperformed by the best performing hybrid algorithms. For the other applications, the average hybrid algorithm performed better for Ramp and ConvShock while the domain-based algorithm resulted in better performance for ShockTurb.

The differences in the number of blocks were large between the algorithms. The best hybrid algorithms always constructed partitions with smaller numbers of blocks, but the domain-based algorithm outperformed the average hybrid algorithm for three of the four applications.

The average hybrid number of communications were significantly larger than the best hybrid best results. We found that only relatively few hybrid algorithms achieved results close to the best ones, while the majority performed much worse. Still, it is only for Ramp that the domain-based algorithm performed better than the average hybrid algorithm. We also found that the algorithms with

Load imbalance (%)	Min. Hybrid	Avg. Hybrid	DB
Ramp	19.5	39.5	63.7
ShockTurb	18.3	30.4	9.5
ConvShock	14.2	29.6	64.7
Spheres	17.9	37.2	18.4
Comp. time (s)	Min. Hybrid	Avg. Hybrid	DB
Ramp	1557	1825	2135
ShockTurb	3018	3321	2727
ConvShock	1958	2170	2722
Spheres	2152	2505	2162
Blocks (#)	Min. Hybrid	Avg. Hybrid	DB
Ramp	32.9	70.2	57.2
ShockTurb	35.9	65.9	52.8
ConvShock	62.3	122.7	152.8
Spheres	49.8	93.4	73.2
Communications (#)	Min. Hybrid	Avg. Hybrid	DB
Ramp	387	1809	1781
ShockTurb	370	1270	1970
ConvShock	876	2878	3954
Spheres	1153	1499	1579

Table 4: Average performance data for all metrics.

the smallest number of communications generally had a high load imbalance. The opposite was also true, a low load imbalance generally resulted in a large number of communications.

8 Summary and conclusions

To maintain good performance for parallel SAMR applications, the dynamic grid hierarchy is repeatedly partitioned and distributed over the participating processors. In this paper, we presented a comprenhensive performance characterization of a large number of partitioning algorithms for parallel SAMR applications. For the characterization, we used a domain-based algorithm from the SAMR framework AMROC and many hybrid algorithms from the partitioning framework Nature+Fable.

We found that the two partitioning methods complemented each other. The hybrid algorithms generally constructed partitions with low load imbalance, few blocks and a small number of communications. When the hybrid algorithms did not perform as well, the domain-based algorithm constructed good partitions. The domain-based algorithm always resulted in larger numbers of blocks and more communication. The hybrid algorithms performed better for scattered and sharp refinement patterns, while the domain-based algorithm was more successful when the refined areas were large and uniform.

To efficiently partition the grid hierachy, it is important to adapt the choice of partitioning algorithm to the state of the application and computer. For each metric and time step, we found a large performance difference between the best algorithm and a random algorithm. By selecting the best algorithm for each time step, large savings in run-time will be achieved. Even if a sub-optimal algorithm is selected, the resulting partitions will typically have significantly higher quality than a random partition. Thus, to consistently construct near-optimal partitions, the partitioning algoritm needs to be dynamic throughout the execution. We must at each time step have the ability to invoke any of the available candidate algorithms. With sufficiently many complementing algorithms, there will be good-performing algorithms for most states.

Unfortunately, there does not exist simple and effective translations between application and computer states and a suitable partitioner [12]. Instead, advanced partitioning algorithm selection methods need to be developed. We propose the meta-partitioner to fullfill this task. The performed experiments proves the viability of the meta-partitioner. Furthermore, the data obtained from this work is suitable to be included in the meta-partitioner and will form a valuable basis for the selection of good-performing partitioning algorithms.

9 Acknowledgements

The authors are grateful to Ralf Deiterding, California Institute of Technology, for providing the trace files and the performance data for the applications. Johan Falkenström and Henrik Lindvall provided the illustration of an AMR grid hierarchy.

References

[1] ALC linux cluster. http://www.llnl.gov/linux/alc/, Oct. 2006.

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

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

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

[8] R. Deiterding, R. Radovitzky, L. Noels S. Mauch, J.C. Cummings, and D.I. Meiron. A virtual test facility for the efficient simulation of solid material

response under strong shock and detonation wave loading. *To appear in Engineering with Computers*, 2006.

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

[10] 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.

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

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

[13] 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.

[14] 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.

[15] Jiuxing Liu et al. Performance comparision of mpi implementations over infiniband, myrinet and quadrics. In *Proceedings of Supercomputing*, 2003.

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

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

[18] 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.

[19] 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.

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

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

[22] 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.

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

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

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

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

[31] M. Vetter and R. Stuartevant. Experiments on the Richtmyer-Meshkov instability on a air/SF6 interface. *Shock Waves*, 4(5):247–252, 1995.

[32] 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.

[33] 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.