Novel views of performance data to analyze large-scale adaptive applications

Abhinav Bhatele†, Todd Gamblin†, Katherine E. Isaacs*, Brian T. N. Gunney†, Martin Schulz†,
Peer-Timo Bremer†, Bernd Hamann*

†Center for Applied Scientific Computing, Lawrence Livermore National Laboratory, Livermore, CA 94551 USA
*Department of Computer Science, University of California, Davis, CA 95616 USA
E-mail: {bhatele, tgamblin, gunneyb, schulzm, ptbremer}@llnl.gov, keisaacs@ucdavis.edu, hamann@cs.ucdavis.edu

Abstract— Performance analysis of parallel scientific codes is becoming increasingly difficult due to the rapidly growing complexity of applications and architectures. Existing tools fall short in providing intuitive views that facilitate the process of performance debugging and tuning. In this paper, we extend recent ideas of projecting and visualizing performance data for faster, more intuitive analysis of applications. We collect detailed per-level and per-phase measurements in a dynamically load-balanced, structured AMR library and relate the information back to the application’s communication structure. We show how our projections and visualizations lead to a simple diagnosis and mitigation strategy for a previously elusive scaling bottleneck in the library that is hard to detect using conventional tools. Our new insights have resulted in a 22% performance improvement for a 65,536-core run on an IBM Blue Gene/P system.

I. INTRODUCTION

As the size and complexity of modern architectures increases, performance analysis of massively parallel scientific applications becomes ever more crucial in order to scale these codes to execute efficiently. While there exist a number of tools and frameworks to collect a large variety of performance data in the form of profiles or traces, interpreting this massive amount of information is rapidly becoming the bottleneck of any analysis. One of the main reasons is that often no individual measurement contains sufficient information to detect, let alone diagnose, most problems. Instead, application developers are forced to manually collect and correlate disparate pieces of information to gain the necessary insights. This strategy is becoming infeasible especially for large core counts and adaptive applications. The former makes most standard plots incomprehensible while the latter significantly reduces the ability of users to relate individual measurements to each other.

In an application that has a static data distribution, for example a regular grid code, the relationship between MPI ranks, nodes, cores, and the application domain remains fixed throughout the execution. This makes it possible, if tedious, for an application developer to connect, for example, flop counts taken per core with a particular subset of the simulation domain. In an adaptive application, such as an adaptive mesh refinement (AMR) code, however, these relations constantly change during the execution leaving the user with few options to understand non-trivial performance problems. Further, such applications typically introduce an entirely different type of computation related to creating, maintaining, and adapting the application, which may become a scalability bottleneck in itself.

To address these challenges, new types of performance tools are necessary that can: (a) attribute performance measurements to the relevant computational phases and dynamic application data; and (b) automatically create and exploit the necessary relationships between measurements recorded on different domains. Schulz et al. [1] recently codified these concepts into the HAC model, aimed at understanding the relationships between static application domains, HPC hardware, and interprocess communication. In this paper, we expand on these concepts and focus on understanding and exploiting such relationships for dynamically decomposed application domains.

As a case study, we focus on a massively parallel AMR library and first show how even detailed measurements of various performance aspects of the application fail to highlight the root cause of the problem. We propose a new projection of data collected on the hardware domain onto the communication graph as well as a new scalable visualization of the resulting coalesced information. This provides an easily apparent and intuitive diagnosis and also suggests a simple strategy to alleviate the problem. Specifically, the new contributions described here include the following:

1) Detailed per-level and per-phase performance measurements of a massively parallel structured AMR code;
2) A new projection of the per-phase and per-core data onto the communication domain;
3) A new scalable visualization technique that combines hardware and communication data providing an intuitive diagnosis of an elusive scalability problem; and
4) A mitigation strategy which has improved the performance of an AMR library by 22% for a 65,536 core run on a Blue Gene/P system.

This paper shows that carefully measuring, attributing, and integrating various performance data can lead to simple diagnoses of previously obscure problems. As a case study we use SAMRAI [2], [3], a highly scalable structured AMR library used extensively in several large-scale computational science applications. SAMRAI displayed an elusive scalability problem which turned out to be related to the load balancing
phase of the mesh adaptation stage. It is critically important to analyze all collected data in the context of the special binary tree shaped overlay network used during this phase. As shown in Fig. 1(a) the tree shows a seemingly insignificant load imbalance which is concentrated in one of its four quadrants. However, looking at the times spent waiting to address this imbalance (Fig. 1(b)) reveals a strong correlation. Finally, including the amount of meta-information moved (Fig. 1(c)) clearly shows the problem: a linear number of elements must travel over just one edge of the tree ultimately causing a linear scaling of the code. Note that, as will be discussed in detail below, this is neither a latency (the number of hops messages travel is provably scaling as $\log$) nor a bandwidth problem (the size of messages is small), which made it difficult to diagnose with traditional techniques. These insights have allowed us to propose a mitigation strategy that has resulted in a 22% overall performance improvement for a 65,536-core run on a Blue Gene/P system. Further, the detailed analysis will enable us to redesign the load balancing algorithm to remove the scalability problem completely in the future.

II. PROJECTING DATA ACROSS DOMAINS

Schulz et al. [1] have developed a taxonomy of performance data that divides measurements into three key domains. These are the hardware domain, consisting of processors embedded in a network with some topology; the application domain, consisting of information from the application’s simulated physical domain; and the communication domain, consisting of abstract graphs with processes as nodes and communication between them represented as edges. This framework is called the HAC model (see Fig. 2).

While symptoms may show up in any of these domains, the actual root causes could lie in any other. We therefore need new techniques for visualizing and analyzing performance data that correlate symptoms to causes by projecting performance data from one domain to another in order to make correlations and root causes more clear.

The difficulty of projecting data across domains depends on how much and how frequently the relationships between domains change. For example, in a statically decomposed, structured grid application, the domain decomposition is fixed, and we can assume that per-process measurements are associated with a particular chunk of the decomposed application domain. In dynamic applications, which will get more commonplace as we move to more complex architectures and applications on our path to exascale, this is no longer the case. For example, in an AMR code the physical domain is decomposed into variable sized units, which can be moved dynamically from process to process. We must therefore take special care to track the units as they move around the system in order to detect a performance problem that arises because of particular application features in one part of the application domain.

Similarly, for a structured grid code, most communication is regular. For example, many such codes use a simple stencil-patterned ghost exchange among neighboring processes. We
can easily make assumptions about which processes communicate and how much data passes over each communication link. However, if there are many phases with very different communication patterns, we cannot attribute all bandwidth to the same algorithm. Instead we must provide a more fine-grained analysis that can distinguish phases and track many independent communication patterns.

In adaptive applications like AMR, this kind of dynamic behavior is driven by the application, its domain decomposition, and its phase structure. The behavior also changes depending on the particular problem being simulated. Performance measurement tools must therefore be able to map performance measurements pertaining to communication and computation back to entities in the application domain in order to find the root causes of performance problems.

In the remainder of this paper, we describe how we have constructed inter-domain projections that track these relationships to unscramble the mess left by adaptivity. We use these projections to create new, insightful visualizations in more intuitive domains that clearly highlight root causes of performance problems.

III. STRUCTURED ADAPTIVE MESH REFINEMENT

In this paper, we study an application domain that is complex and difficult to scale. Structured adaptive mesh refinement (SAMR) is a popular AMR technique in which the simulation domain is described by a hierarchy of successively finer mesh levels, as shown in Fig. 3. The bottom-most level spans the entire problem domain and its mesh is composed of the largest cells. Meshes in successively higher levels have increasing cell resolution but typically do not cover the entire domain. Instead, higher levels comprise a set of boxes that contain only cells in need of refinement. For data-parallel implementations the mesh is distributed by assigning one or more boxes to each process, if necessary splitting larger boxes to achieve the necessary granularity. Standard SAMR applications consist of three operations: local computation on individual boxes, coordinated data exchange between neighboring/overlapping boxes, and mesh adaptation and load balancing.

In this paper, we focus on the mesh adaptation and load balancing phases included in the AMR library. The advantage of an AMR approach is that the mesh resolution can be locally adapted to the requirements of the underlying simulations which greatly reduces the overall workload. However, to maintain this property the mesh must be constantly adapted and redistributed to ensure an evenly balanced workload. This step is by its nature a communication intensive and thus expensive operation, and typically the most difficult to scale.

Each time a mesh level should be created/adapted, the next coarser level is examined to decide the new set of boxes/cells on this level. Subsequently, these boxes must be distributed evenly among all processors to ensure a balanced computational load. Finally, the overlap and neighborhood relations between boxes must be updated to enable the data exchange during the computation stage. Since each process needs at least one box, a naive re-meshing approach will scale as \( O(P) \) for both strong and weak scaling, where \( P \) is the number of processors. Consequently, re-meshing can quickly become the dominate cost at larger core counts.

A. SAMRAI

SAMRAI [2] is a highly scalable structured AMR library used extensively in several large-scale DOE production applications. As discussed below, SAMRAI uses a parallel mesh management [4] that greatly reduces the cost of re-meshing. Nevertheless, the scaling behavior for very large core counts remains a challenge. Fig. 4 shows the current performance of adapting the highest level mesh in the LinAdv benchmark from the SAMRAI distribution, which simulates a sinusoidal wave passing through the domain. In this test case, we are using three mesh levels with a refinement ratio of two on the two Blue Gene/P systems at Argonne National Laboratory (Challenger and Intrepid). Intrepid is a 557.1 TFlop/s Blue Gene machine with 163,840 PPC450 cores and a three-dimensional torus interconnect. Challenger is a smaller, 2,048 core test machine with the same architecture and software configuration. The figure shows the timings for the entire computation (black), just the mesh adaption (dotted green), and just the load balancing phase (orange) of the adaptation stage for weak scaling. Beyond 8K cores the mesh adaptation starts to dominate and in particular the load balancing appears
problematic. This represents a serious scaling problem, as large-scale runs will not achieve good parallel efficiency if most of their time is spent re-meshing.

In the following we describe the current load balancing algorithm of SAMRAI before discussing various strategies to diagnose its apparent bottleneck.

B. Load balancing in SAMRAI

The load balancing in SAMRAI starts from a set of boxes describing the new level. These boxes are created locally on each processor from the existing coarser level without regard to load distribution and thus are likely to be unevenly distributed. In the load balancing stage SAMRAI attempts to re-distribute and/or split boxes in a way that ultimately balances the total number of grid cells given to each process. Load balancing is performed in three phases: First, the load distribution phase will compute the relative load with respect to the average on each processor and distribute boxes accordingly. Second, the mapping phase constructs the relation between pre-distribution and post-distribution boxes. Third, the overlap phase reconstructs the information about how post-distribution boxes overlap the next coarser level. All three phases operate exclusively on meta-data meaning on the extents and locations as well as the IDs of boxes rather than their actual data. Even for the largest runs the resulting messages are smaller than the latency-bandwidth product for Blue Gene/P.

Load distribution. The load distribution algorithm is based on a recursive traversal of a binary tree aimed at limiting the “distance” each box has to travel. Once the average global load has been computed with a global reduction, processes are arranged into a balanced binary tree by recursively splitting the MPI rank space. Then, each processor waits for its two children to report their respective deficit or surplus of work. In the case of a surplus this message will also contain the excess boxes. The processor integrates this work to its own load and recursively reports the aggregated deficit or surplus to its parent alongside potential excess boxes. This recursion naturally stops at the root where, by definition, all loads will balance. Note that at this stage, there can still be processes other than the root with excess boxes that have not yet been distributed to their children. In the last step the recursion is inverted: The root will send its excess boxes to its children with deficits, which recursively will integrate the extra boxes with their local excess and send the results to the respective subtrees.

This algorithm guarantees that no box takes more than \(O(\log P)\) steps along the tree and all nodes in the tree send at most two messages: One to report their aggregated load and one to potentially distribute excess boxes. Furthermore, the algorithm involves two potential MPI_waitall's: one to wait for updates from child nodes in the tree and, if necessary, a second one to receive boxes from the parent node.

Mapping computation. At some point data must be transferred from the pre-adaptation mesh to the post-adaptation version. This requires a mapping from each of the new boxes on level \(k\) to one or multiple boxes of level \(k\) and/or \(k-1\) from which to copy and/or compute the corresponding data. During the distribution phase each box is tagged with its originating process and boxes that are split inherit their origin. In the mapping phase each processor informs the corresponding originating process of the final destination of its boxes.

Computation overlap. Finally, all boxes on the new level must update their neighborhood and overlap information. This entails computing which boxes on the same level share a boundary, which boxes on higher/lower levels overlap, and which processor now owns the corresponding box. To avoid a global search for overlaps, SAMRAI uses the information from the pre-adaptation mesh combined with the mapping information of the previous phase to update the meta-data.

One or more of these three load balancing phases is responsible for the scalability problems that we see in Fig. 4. In the next section, we describe several traditional techniques aimed at diagnosing the problem, why they failed, and our new techniques to illustrate and address the root cause problem.

IV. USING STATE-OF-THE-ART PERFORMANCE ANALYSIS

A variety of tools exist that help users and library developers to gather and analyze performance data. However, most tools are restricted to analyzing the data in the context in which it was collected which is often not sufficient to diagnose complex problems. A prime example is per-process measurements which for convenience are typically collected with respect to the MPI rank space. However, especially in adaptive applications the rank space has few obvious connections to either the underlying hardware or the application domain making interpreting such data difficult. Furthermore, most tools have no or only limited support for dynamic applications, such as SAMRAI.

Following the discussion of Section III the dominant scaling problem of SAMRAI appears to be the load-balancing stage. However, given the complexity of the algorithm the root cause and thus a potential solution remains unclear. In the following we discuss the use of several common performance tools and methodologies aimed at finding the root cause and their shortcomings for this particular problem.

A. Aggregate profiles and information

The first attempts at diagnosing a performance problem are typically globally aggregated measurements, the coarsest of which are simple overall execution times as those of Fig. 4. While such plots demonstrate that a problem exists and in which portion of the code it manifests, they provide little insight into a solution. The next step is to provide a rough understanding of whether a problem is related to communication or computation. By design the load balancing algorithm guarantees a logarithmic number of hops for all messages ruling out a latency issue. Similarly, since only meta-data is transmitted during the load balancing stage the maximal message size remains below the latency-bandwidth product of the Blue Gene/P even for the largest runs, which
argues against a bandwidth limitation. This leaves potential contention on the network as a possible issue. Tools such as the communication matrix module in PN MPI [5] (which we used), TAU [6], and Vampir [7] allow one to record the complete communication matrix showing traffic between all node pairs. However, a matrix plot of the communication matrix, as shown in Fig. 5, reveals no obvious patterns. Furthermore, since the load balancing algorithm sends only few point-to-point messages contention seems unlikely.

Fig. 5. Communication matrix of 256 processes nodes running SAMRAI during the load balancing stage, color mapped according to message size.

B. Per-phase data

Going beyond aggregated results in form of global execution times or the communication matrix, profiling tools can obtain detailed information about the time spent in computation and communication on each process. Many tools offer the ability to obtain MPI profiles, including OpenSpeedShop [8], TAU [6], and Scalasca [9]. For the following experiments, we use mpiP [10], which provides information such as total time spent in MPI calls versus total application time and also the top MPI calls and their respective call sites where most of the time was spent. mpiP can be used to selectively profile a code region, and as most tools, it relies on the use of MPI_Pcontrol calls for this feature. In our case we use it to focus on the details of the three phases of the load balancing algorithm with the intent to assign blame to specific phases.

However, a common drawback of the MPI_Pcontrol mechanism is that we can only turn profiling on or off in mpiP. There is no mechanism to generate distinct profiles for different code regions within a single run without significant and complex changes to the profiler itself. We use PN MPI [11] to virtualize mpiP so that multiple code regions can be profiled at once using multiple, unmodified instances of mpiP.

Fig. 6 shows the sum of times spent by all MPI processes in the three phases of load balancing: load distribution, mapping, and overlap generation for different problem sizes. From the aggregated data it appears that the overlap generation (phase 3) is the main problem. However, as discussed in Section III load balancing is done in an asynchronous manner which may distort the results when one processor waits in a later phase for other processors to finish an earlier phase.

C. Per-core, per-phase data

Since the per-phase data of Fig. 6 is inconclusive the next step is to further refine the attribution and analyze the MPI profiles on a per-core as well as a per-phase basis. This is typically the finest scale data the tools discussed above provide. Fig. 7 shows this data with respect to the MPI rank space for a small 256 core run of SAMRAI. It is apparent that some processes indeed spend significant time in phase 1 of the load balancing thus likely causing long waits for others. However, since the rank order is not immediately related to the underlying dependencies the cause of this anomaly is still unclear. Furthermore, this graph is for a small test case which may or may not actually exhibit the same behavior as large scale runs. Indeed, as will be discussed in Section V, this plot includes several artifacts of the same magnitude as the problem we are trying to detect. Unfortunately, creating similar graphs for much larger core counts is futile as one would no longer be able to distinguish neighboring ranks for the lack of resolution.

Additional performance data can be gathered through trace and trace visualization tools, such as Vampir [7] or JumpShot [12], but the resulting data is often overwhelming in its detail and hard to interpret when looking for general patterns. A hybrid approach between full tracing and profiling is call path tracing [13], which provides and visualizes per process traces of sampled call paths. However, this technique is limited to MPI rank space and does not work well for adaptive codes.

Tallent et al. have investigated automatic discovery of scalability bottlenecks at particular phases of program execution, also based on call paths [14]. This work complements our work by providing automatic detection of the initial scalability bottleneck we noticed in the SAMRAI load balancer. Again,
though, this work only supports visualizations of how the observed data relates to the application source code, and not how it relates to application semantics.

Finally, many existing parallel performance tracing frameworks [7], [15], [9], [16], [17] attempt to visualize the behavior of large-scale parallel programs, either by visualizing communication between processes, by visualizing hardware metrics on a torus, or by examining communication traces using three-dimensional views. None of these, however, support the projection of application data into performance domains or vice versa, limiting their ability to pinpoint performance bottlenecks through the kind of correlation analysis presented here.

Overall, these state-of-the-art tools provide valuable insight into which phase is causing the scalability bottlenecks, but fall short of helping to explain why. The presented data is not tied to the application and its communication structure nor does it help track the adaptivity in the application. Data is typically presented relative to the context it was collected in and hence the interpretation of the data changes as the application progresses, making it hard to understand the underlying relationships.

V. GAINING NEW INSIGHT THROUGH PROJECTIONS

The fundamental challenge for the techniques discussed above is the tight connection of the data collected to the domain it is collected on. In this case the MPI rank space is non-intuitive and to interpret the data one must explore a more relevant context. Following the HAC model introduced in Section II, we map the data gathered into two related domains, the hardware domain, showing the impact on the physical hardware layout, and the communication domain, allowing a natural interpretation of performance data in relation to the application’s communication patterns. Incidentally, the latter also maps performance data into a “stable” domain whose semantics and interpretation are not affected by the adaptive nature of the application.

A. Projections on the hardware domain

We use the timing information collected from mpiP within a particular load balancing phase (as shown in Fig. 7) and project it onto the 3D physical torus of the Blue Gene/P hardware it was collected on. Using Boxfish, a lightweight, interactive visualization tool we have developed, we display an abstract representation of the cores, nodes, and the hardware interconnect colored according to the data of interest.

Fig. 8 shows the view generated by Boxfish when we color each node on the torus by the time processes spent in phase
Fig. 9. Phase timing data and load information visualized in the communication domain, showing the virtual tree network of the load balancing phase (512 processes on Blue Gene/P). Nodes are aggregated at deeper levels and colored by their average weight. (a) Nodes colored by times spent waiting to receive excess load. (b) Nodes colored by relative load before re-balancing. (c) The tree of (a) with arrows scaled by the number of elements sent between nodes. This highlights the heavy use of few links which leads to increased wait times at the receiving nodes.

1 of the load balancer. Apparently, the processes that spend the most time in this phase are on the third and fourth planes of the torus. Hot planes like this may indicate contention, but as discussed above, the communication matrix is sparse and messages are small, thus it is more likely that nodes on these planes are waiting.

Given the strong correlation, some connection to the scaling bottleneck seems probable which leads naturally to the analysis of the overlay network used during load balancing. Since, SAMRAI uses a recursive subdivision of the MPI rank space, the next hypothesis has been that the plane represents a sub-tree of the overlay network. To test this theory we project the same information onto a layout of the overlay network which provides a clear and intuitive explanation.

B. Projections on the communication domain

As described in detail in Section III-B, SAMRAI communicates load variations and work loads (boxes) along an overlay tree network. To understand the communication behavior, we need to go a step further and project phase timing data onto the communication domain, i.e., the load balancing tree. This ultimately allows us to connect the performance data to the critical application behavior and its communication pattern, and to overcome its adaptivity, since the structure and type of the information stays constant across load balancing steps.

We construct a pairwise communication graph among the MPI processes for the load balancing phase, which results in a binary tree. As shown in Fig. 9 we draw the tree in a hierarchical radial layout which emphasizes levels closer to the root and clusters nodes of similar behavior on lower levels. As a first step we color the nodes by the time they spend in different sub-phases of the load balancer.

Fig. 9(a) shows the virtual tree network used in the load balancing phase with each node colored by the time the corresponding MPI process spends in phase 1, i.e., load distribution. Interestingly, in this view, we see that a particular sub-tree in the virtual topology or communication graph is colored in orange/red, highlighting the processes that spend the most time in phase 1. Further, from the mpiP output, we were able to ascertain that nearly 85% of this time is spent in an MPI_Waitall, where a child is waiting to receive boxes from its parent. The problem escalates as we go further down this particular sub-tree, which is reflected in the increasing color intensity, i.e., processes farther away from the root spend longer time in this phase.

This is somewhat surprising as a plot of the initial load (Fig. 9(b)) - relative to the average - indicates no immediately obvious load imbalance to cause such drastic differences in wait times. The load appears to be randomly distributed with over- and under-loaded nodes sprinkled through the entire tree. However, a closer inspection reveals that on average three of the four subtrees on level two have 2.83, 2.87 and 3.1% excess load, respectively while the lower left tree has a 9% load deficit. This asymmetry explains the plot of Fig. 9(a); since load from three quarters of the tree all has to flow to the remaining quarter, the later lies at the end of a long dependency chain.

This leads directly to the final visualization in which we draw arrows according to the number of boxes that are sent over each link (Fig. 9(c)). To avoid confusion we only show the flow of excess boxes from parent to children nodes. From this illustration, the cause of the scaling bottleneck is directly apparent: in case of even a small load imbalance that is asymmetrically distributed, the binary tree can act as a funnel forcing a large percentage of all boxes moved to flow on a single or a small number of edges. Note that following the discussion of Section IV this is not a contention or local bandwidth issue as the size of the overall message (even on the hottest link) remains small. Instead, the problem is related to the number of boxes that must be processed for shipping.

The problem becomes worse for larger number of cores
as the maximum number of boxes to be sent on a particular edge continues to increase, as shown in Fig. 10. On 131,072 cores, we send 56 times the number of boxes that we send on 256 cores on any given edge in the tree. This explains the scalability bottleneck (attributed to the load balancing phase) that we observed in Fig. 4. While the tree network that is used for load balancing places an upper bound on the number of hops a box may travel, it may funnel load from subtrees through sparse edges near the root. This makes the algorithm susceptible to small variations in the initial distribution of load and leads to a flow problem [18] where a large number of boxes are routed through a single edge to replenish an under-utilized sub-tree.

Fig. 11(a) shows the overlay network with the wait times in phase 1 and flow information for 1024 cores, Fig. 11(b) for 2048 cores, and Fig. 11(c) for 16,384 cores. The load imbalance moves progressively further down the tree but the essential problem remains the same. Fig. 12 shows the data of Fig. 7 and indicates why any analysis based on such a small run would be misleading: The run is too small for the scaling bottleneck to dominate which results in an inconclusive picture of the flow. Note how there exist several heavily used edges in the tree of Fig. 12 not related to the fundamental problem.

To preserve the symmetry of the tree layout and provide the most direct visual link with the mental picture of a binary tree we have chosen to refine all subtrees to an equal depth. However, in cases where the problem exists far away from the root we provide an adaptively refined layout (see Fig. 13) which enables us to highlight flow problems at any level of the tree. In the adaptive layout we re-scale the angle assigned to the subtrees by their accumulated weights (wait time in this case) and refine until the variance with each subtree is below a given threshold.

VI. TURNING INSIGHT INTO OPTIMIZATION

The insight gained by interpreting the performance data in the communication domain directly points to the core problem: Scalability in the load balancing phase is restricted by a flow problem in the virtual tree topology. If a particular subtree needs to receive load from the remaining subtrees,
the corresponding traffic (in terms of meta-data for boxes) must flow through one particular edge in the load distribution phase. We understand that to ultimately eliminate this problem we must use of a different overlay network that prevents this scenario and we plan to do rewrite the load balancers in SAMRAI based on the results presented in this paper.

However, the results also lead to a series of initial steps aimed at mitigating the problem without rewriting the whole algorithm. The goal is to reduce the amount of data sent around the tree in two different ways. Part of the data sent with each box is a history of where the box has been. SAMRAI uses this in the mapping generation phase to send data back to the box’s originating process along the tree. Instead, we changed the algorithm to send the data directly to the originator instead of through the tree overlay structure which reduces the number of hops and eliminates the need for the extra data per box.

This “direct send modification” leads to a reduction in the load balancing time (as shown in Fig. 14). Compared to the old scheme, using direct sends results in a 21% performance improvement at 256 cores and 36% at 65,536 cores. This reduction in the time for load balancing leads to an improvement in the overall execution time per iteration (solving plus adaptation) by 6%.

As a second step, we target the reduction of the number of boxes being sent around by increasing the size of each box in terms of the number of cells it holds. Increasing this size has the effect of including more untagged cells in the level generated. It also reduces the choices the load balancer has when breaking up a box. The default value for the box size is (5, 5, 5) cells. We ran experiments with three larger box sizes and recorded the maximum number of boxes sent on any edge along with the timing information. Fig. 15 presents the reduction in the maximum number of boxes sent along any edge of the tree. We get better results as we continue to increase the box size. On 65,536 cores, using (7, 7, 7) boxes, roughly half the number of boxes are sent on any given link. There are 18 times fewer boxes when the box size is changed to (9, 9, 9) cells.

Changing the box size leads to a reduction in the amount of
traffic on the overlay network, which translates into a reduction of the time spent in load balancing (see Fig. 16). Compared to the default box size, using (7, 7, 7) boxes, load balancing is completed in nearly half the time on 65,536 cores. This time is decreased even further with larger boxes for large core counts. In spite of the increased computation resulting from having more cells per box, increasing the box size still leads to a reduction in overall time per iteration (spent in solving plus adaptation). This might be due to lower overheads from handling fewer boxes during the solving phase. At 65,536 cores, we get a performance benefit of more than 16% by creating slightly larger boxes. Comparing with the baseline performance, using the two optimizations together gives a performance benefit of 25% on 256 cores and nearly 22% on 65,536 cores in the overall runtime.

Fig. 15. Reduction in maximum number of boxes sent on any edge of the tree by increasing the size of each box. This reduces the amount of traffic on the overlay network which reduces the time spent in load balancing.

Fig. 16. Improvement in load balancing time by using larger, and thus creating fewer, boxes.

VII. SUMMARY

We presented a case study of performance analysis for a structured adaptive mesh refinement library to identify hard-to-detect scalability issues. We used the HAC model to project data from one performance domain to another to better attribute performance problems to their root causes. In the case of SAMRAI, we were able to exploit the relationships between different domains, particularly, application and communication, to identify scalability issues in the load balancing phase. We believe that the process outlined in this paper and the visualization techniques presented are generally applicable and especially useful for adaptive scientific codes.

We also presented some preliminary solutions to mitigate the scalability problems in SAMRAI. Our performance analysis techniques helped us to pinpoint the scalability problems of the SAMRAI load balancer in the communication domain. This enabled us to identify hot spots in the communication patterns as well as their cause, and led us to two optimization techniques. By reducing the amount of traffic and the amount of data going through these hot-spots on the overlay tree network, we demonstrated an improvement of up to 22% in overall execution time. We realize that completely eliminating the flow problem during load balancing in SAMRAI requires the use of a different overlay network that prevents this scenario and we plan to do rewrite the load balancers in SAMRAI in the future.

ACKNOWLEDGMENTS

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 (LLNL-CONF-554552).

REFERENCES


