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 for a dynamically load-balanced, structured AMR library and project per-core data collected in the hardware domain on to the application's communication topology. We show how our projections and visualizations lead to a rapid diagnosis of 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 of the AMR library 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 ensure efficient scaling of these codes. 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 for an application developer to connect, with significant effort in most cases, individual measurements – 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 may introduce an entirely different type of computation related to creating, maintaining, and adapting the simulated domain, which may become a scalability bottleneck in itself.

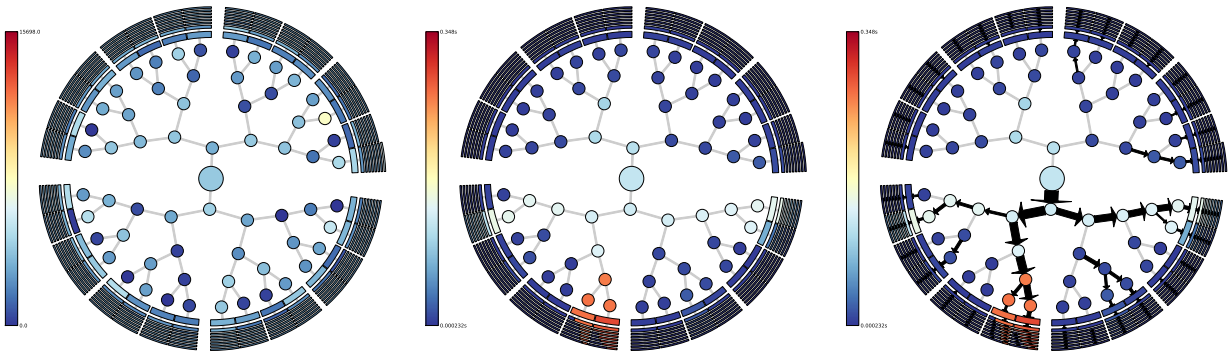
To address these challenges, new types of performance tools are necessary that can: (a) attribute performance measurements to the relevant computational phases and dynamic application data; and (b) automatically create and exploit the necessary relationships between measurements recorded on different domains. Schulz *et al.* recently codified these concepts into the HAC model [1], aimed at understanding the relationships between static **A**pplication domains, HPC **H**ardware, and inter-process **C**ommunication. In this paper, we expand on these concepts and apply them to the understanding of and exploiting such relationships for performance analysis of 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 a performance problem. We propose a new projection of data collected in the hardware domain on to 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 improves the performance of the 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

SC12, November 10-16, 2012, Salt Lake City, Utah, USA
978-1-4673-0806-9/12/$31.00



(a) (b) (c)

Fig. 1. Visualization of the binary tree shaped virtual communication topology used in SAMRAI's load balancing phase for 16,384 cores. Warmer colors (as one moves up the scale) represent higher values of load or time. (a) Nodes colored by load before load balancing: No imbalances are obvious. However, there are minor differences with the two top trees being slightly overloaded. (b) Nodes colored by times spent waiting to receive excess load. (c) The tree of (b) with arrows scaled by the number of boxes sent across, highlighting the problem. The tree structure of the SAMRAI load balancing scheme acts as a funnel with most excess being forced along a single edge causing a linear amount of processing.

used extensively in several large-scale computational science applications. An elusive scalability problem was reported in SAMRAI, known to arise from the load balancing phase of the application.

SAMRAI uses a binary tree shaped virtual topology for communication during load balancing and hence, it is important to analyze all collected data in the context of this communication structure. As shown in Fig. 1(a) the tree shows a seemingly insignificant load imbalance which is concentrated in one of its four quadrants. However, the times spent waiting to address this imbalance (Fig. 1(b)) reveal a specific pattern to the performance problem. 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.* have developed a taxonomy of performance data that divides measurements into three key domains [1]. 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).

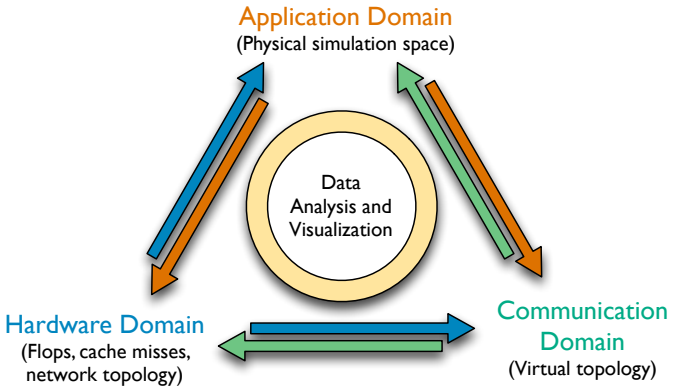


Fig. 2. The HAC model: Data used for performance analysis is divided into three domains. New visualization and analysis techniques that project data between these domains correlate problem symptoms in one domain to behavior in another, making it easier and more efficient to determine the problem origin.

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 massively parallel applications, 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 the 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 which includes load balancing.

In this paper, we focus on the mesh adaptation and load balancing phase 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 is created/adapted, the next coarser level is examined to decide the new set of boxes/cells on this

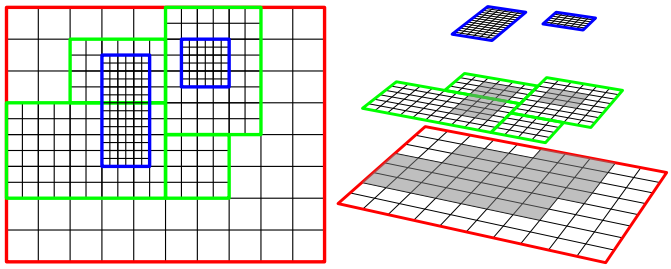


Fig. 3. A simple two-dimensional structured AMR mesh with three levels of refinement (left). The same mesh is shown as a hierarchy on the right. Cells that are refined are shown in gray.

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 $\mathcal{O}(P)$ for both strong and weak scaling, where P is the number of processors. Consequently, re-meshing can quickly become the dominant 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 that greatly reduces the cost of re-meshing [4]. Nevertheless, the scaling behavior for very large core counts remains a challenge. Fig. 4 shows the current performance of 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 and running on the Blue Gene/P system at Argonne National Laboratory (Intrepid). Intrepid is a 557.1 TFlop/s Blue Gene machine with 163,840 PPC450 cores and a three-dimensional torus interconnect.

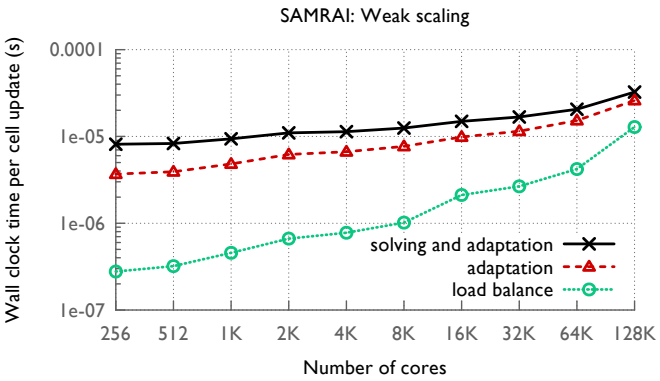


Fig. 4. Weak scaling performance of SAMRAI for the linear advection benchmark on Blue Gene/P. Mesh adaptation dominates for larger core counts where load balancing becomes problematic.

The figure shows the timings for the entire computation (black), the mesh adaptation (dashed red), and just the load

balancing phase (dotted green) of the adaptation stage for weak scaling. Beyond $8,192$ 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 in re-meshing.

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

B. Load balancing in SAMRAI

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 computes the relative load with respect to the average on each processor and distributes boxes accordingly. Second, the *mapping* phase constructs the relation between pre-distribution and post-distribution boxes. Third, the *overlap* phase reconstructs the information about the overlap of post-distribution boxes with the next coarser level. All three phases operate exclusively on meta-data i.e. 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 also contains the excess boxes. The processor integrates this work in to its own load and recursively reports the aggregated deficit or surplus to its parent along with 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 sends its excess boxes to its children with deficits, which recursively integrate the extra boxes with their local excess and send the results to the respective sub-trees.

This algorithm guarantees that no box takes more than $\mathcal{O}(logP)$ 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. Further, 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. 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 load distribution phase, each box is tagged with its originating process and boxes that are split inherit their origin. In the mapping phase, each destination processor informs the corresponding originating processor about the final destination of its boxes.

Overlap generation. 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.

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, which makes interpreting such data difficult. Also, most tools have no or limited support for dynamic applications, such as structured AMR.

Following the discussion of Section III, the dominant scaling problem of SAMRAI appears to be in the load-balancing phase. However, given the complexity of the algorithm the root cause and thus a potential solution remains unclear. In the following sections, 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 the 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 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 P^{N}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. Further, since the load balancing algorithm sends only few point-to-point messages contention seems unlikely.

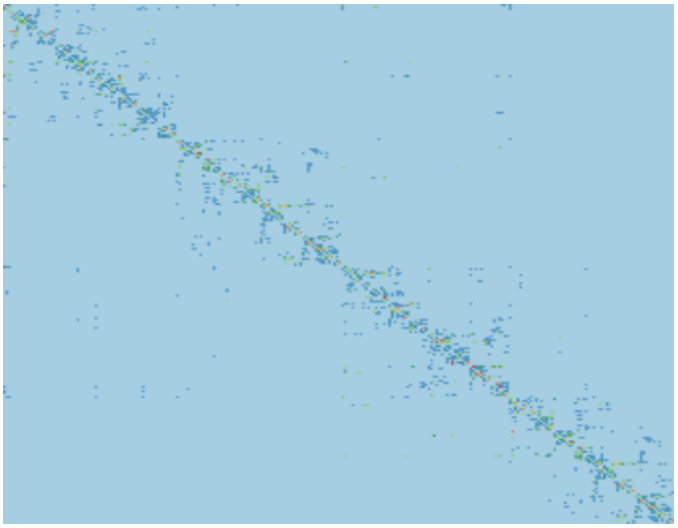


Fig. 5. Communication matrix of 256 processes during the load balancing phase of SAMRAI, color mapped by the number of bytes exchanged.

B. Per-phase data

Going beyond aggregated results in the 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 Open|SpeedShop [8], TAU [6], and Scalasca [9]. For the following experiments, we used 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 P^{N}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 core counts. 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.

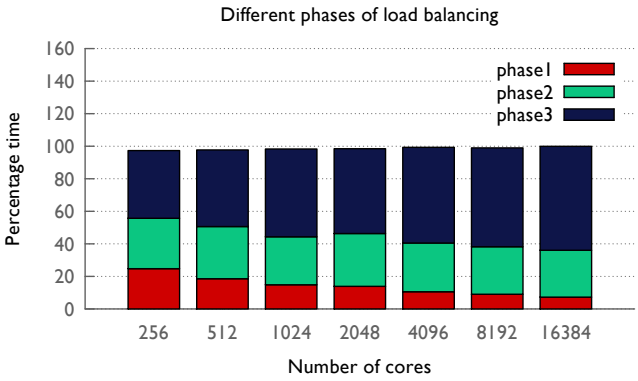


Fig. 6. Sum of times spent by all MPI processes in different load balancing sub-phases. This plot seems to indicate that phase 3, overlap generation, is the main problem. However, due to the asynchronous fashion in which SAMRAI does load balancing, processors in a later phase may be waiting on processors which have not finished an earlier one.

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 load balancing thus likely causing long waits in other phases. 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 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

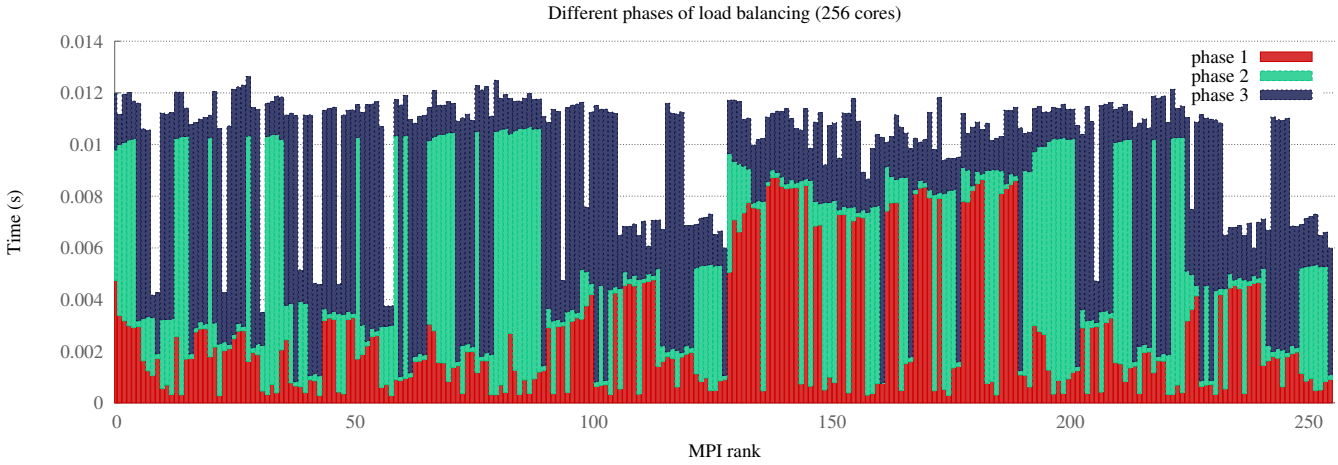


Fig. 7. Time spent in different load balancing sub-phases for 256 processes on Blue Gene/P. Some processes remain in phase 1 for a majority of time which is likely forcing others to wait. This observation is useful, but this graph does not scale to higher core counts and does not reveal the cause of the problem.

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], [9], [15], [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 in this paper.

Overall, these state of the art tools provide valuable insight by identifying the phase causing the scalability bottlenecks, but fall short of helping to explain why. The presented data is not tied to the application domain 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 with 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 projections 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 for phase 1 of load balancing (as shown in Fig. 7) and project it onto the 3D torus of Blue Gene/P, the hardware the data was collected on. Using Boxfish, a lightweight, interactive visualization tool we have developed [18], we display an abstract representation of the cores, nodes, and the hardware interconnect colored according to the data of interest.

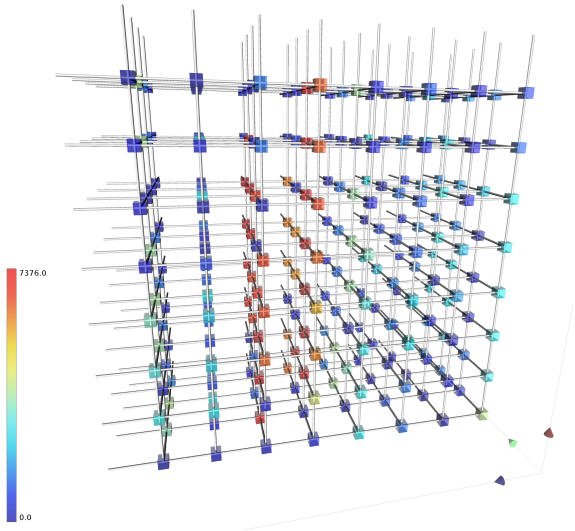
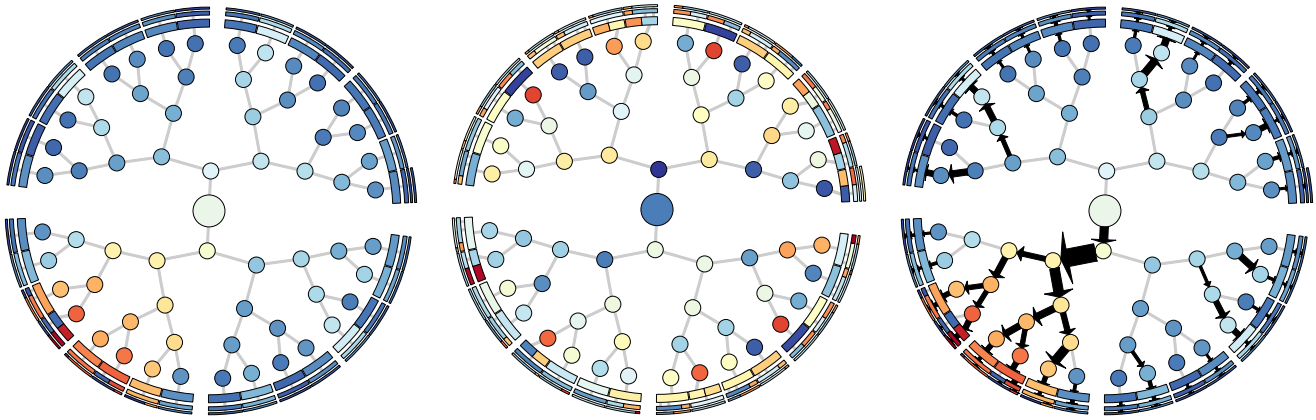


Fig. 8. Boxfish showing 256 nodes in a $8 \times 4 \times 8$ torus colored by the time spent in phase 1 (load distribution) of load balancing. We observe processes in the third and fourth planes spend the most time in this phase and infer they are waiting. This grouping suggests examining the virtual tree topology used during load balancing.

Fig. 8 shows the view generated by Boxfish when we color each node on the torus by the time processes spent in phase



(a) (b) (c)

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 the load before load 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 load balancing. It appears that 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 for other nodes.

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

B. Projections on the communication domain

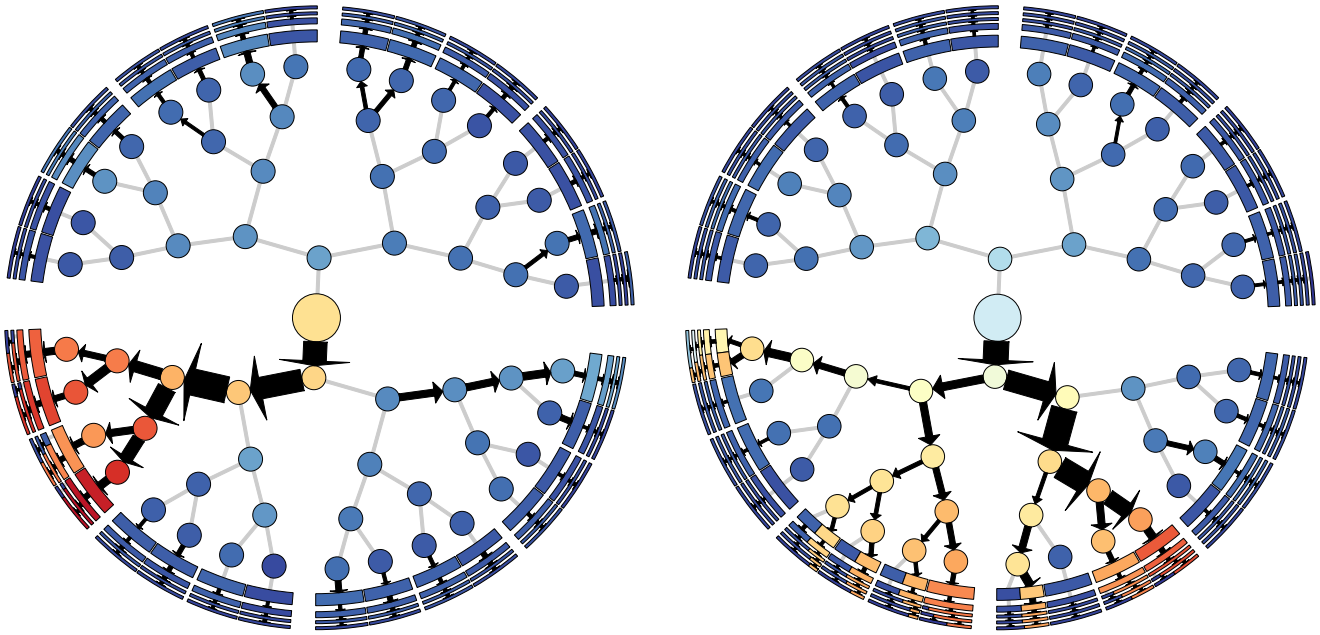
As described in detail in Section III-B, SAMRAI communicates load variations and actual loads (boxes) along a virtual binary tree topology. To understand the communication behavior, we need to go a step further and project phase timing data onto the communication domain, *i.e.*, the load balancing tree. This will ultimately allow 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. Fig. 9 shows the binary tree used by SAMRAI during the load balancing phase. 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.

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 a longer time in this phase.

This is somewhat surprising as a plot of the initial load before load balancing (Fig. 9(b)) does not indicate that the load distribution can 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 sub-trees 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 excess load from three quarters of the tree has to flow to the fourth quarter, the latter 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 downward flow of excess boxes from parents to child nodes. From this illustration, the cause of the scaling bottleneck is clear: 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 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



(a) (b)

Fig. 11. The virtual binary tree topology colored by wait times in phase 1 of load balancing for (a) 1024 cores and (b) 2048 cores. Arrow weights are proportional to boxes sent between parent and child. As core counts increase, the load imbalance moves further down the tree and large numbers of boxes are routed through few edges, resulting in a flow problem.

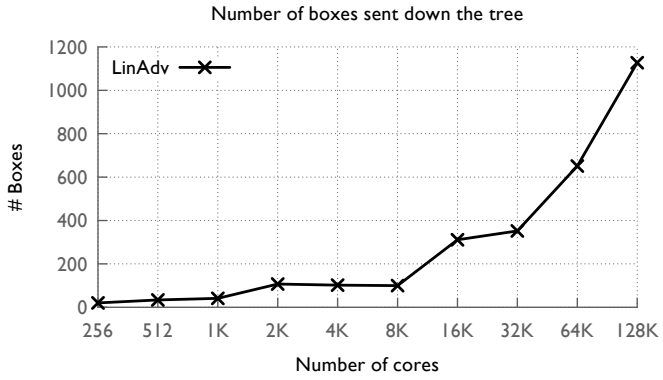


Fig. 10. Maximum number of boxes sent on any edge of the tree as a function of the number of cores (on Blue Gene/P). The virtual binary tree topology funnels a large portion of these messages through few links during load balancing causing the scalability bottleneck.

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 sub-trees 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 [19] where a large number of boxes are routed through a single edge to replenish an under-utilized sub-tree.

Fig. 11(a) shows the virtual tree topology with the wait times in phase 1 and flow information for 1024 cores, Fig. 11(b) for 2048 cores, and Fig. 1(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 chose to refine all sub-trees 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 sub-trees by their accumulated weights (wait time in this case) and refine until the variance with each sub-tree is below a given threshold. This illustrates that such visualizations of the binary tree communication topology are scalable and can be used for visualizing a large number of processes and identifying performance problems even if they are not in the first few levels of the tree.

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

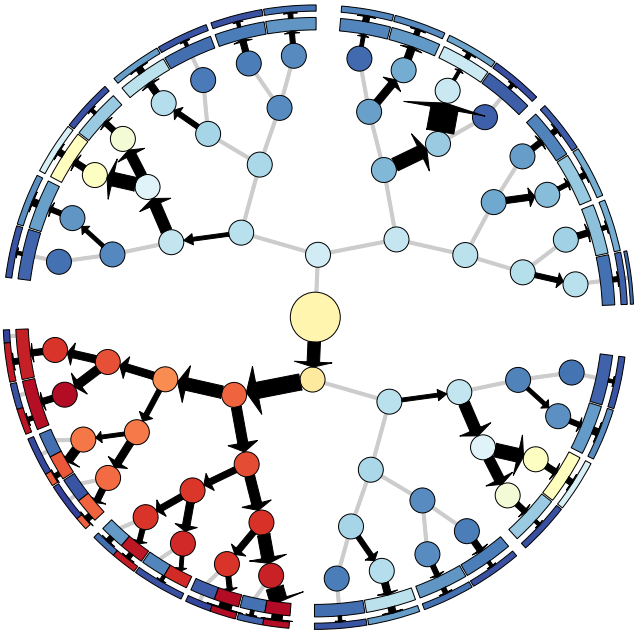


Fig. 12. The wait times for phase 1 (load distribution) and flow information for the 256 core run shown in Fig. 7. There exist several heavily used edges not directly related to the scaling bottleneck including the heaviest edge on the top right. This makes any analysis based solely on this data difficult at best and inconclusive at worst.

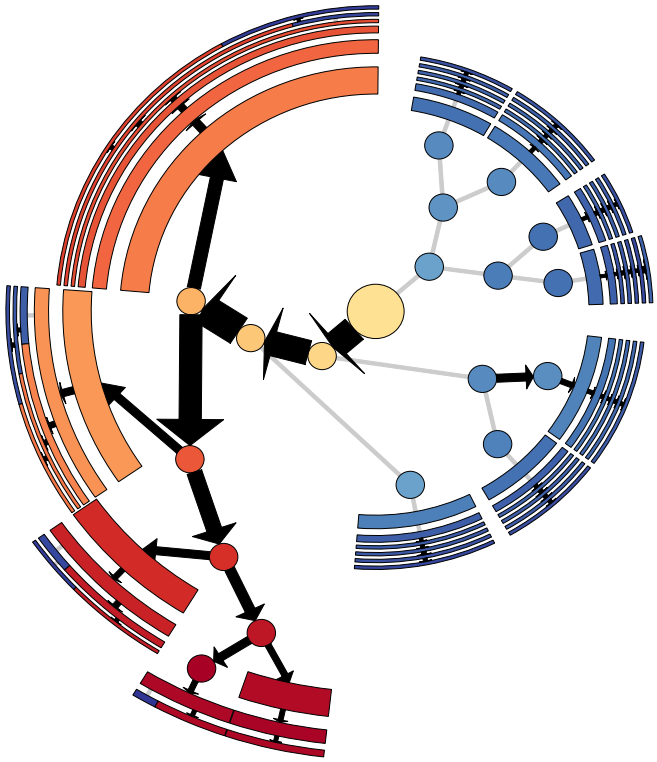


Fig. 13. An adaptively refined layout of the tree for 1024 cores. We apportion the angles based on the overall weights of the sub-tree and refine heavier trees to deeper levels. Adaptively refined layouts enable us to examine problem regions at any level of the tree.

sub-tree needs to receive load from the remaining sub-trees, 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 deploy a different virtual topology that prevents this scenario and we plan to 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. We changed the algorithm to send the data directly to the originator instead of through the binary tree 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%.

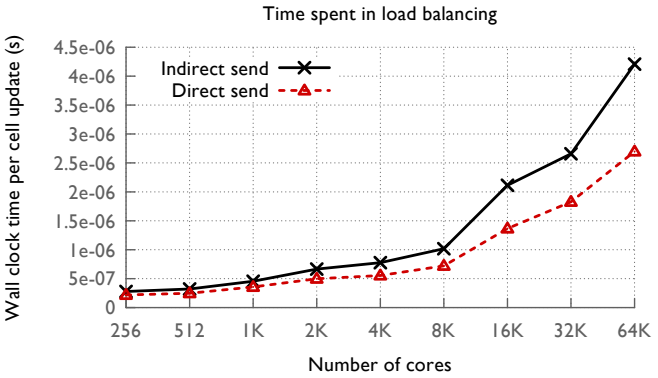


Fig. 14. Reduction in load balancing time by sending patches directly to their new destinations rather than through the binary tree.

As a second step, we target the reduction of the number of boxes being sent 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

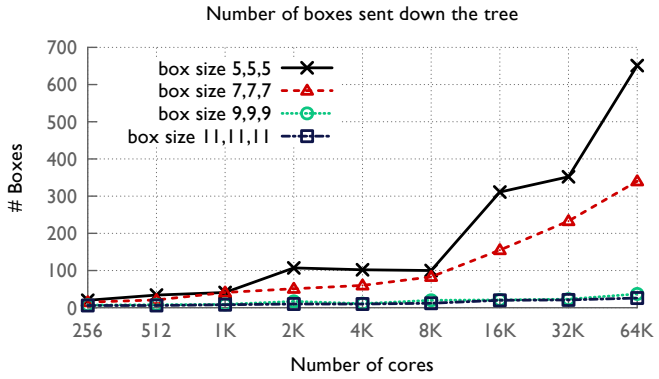


Fig. 15. Reduction in maximum number of boxes sent on any edge of the tree by increasing the size of each box.

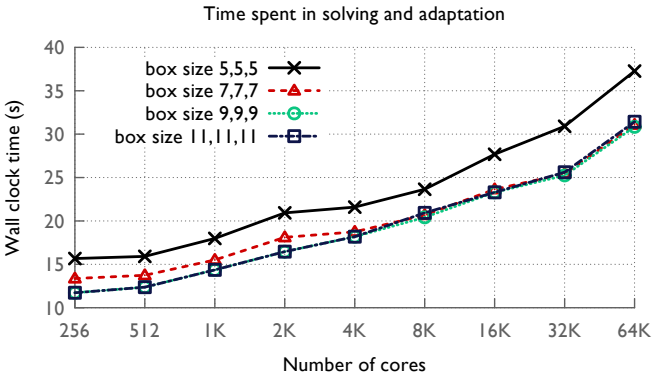


Fig. 17. Improvement in overall (solving plus adaptation) time by using larger, and thus creating fewer boxes.

traffic on the virtual binary tree topology, 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. 17).

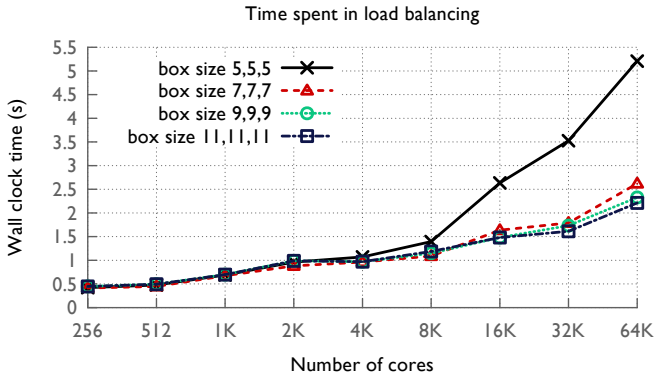


Fig. 16. Improvement in load balancing time by using larger, and thus creating fewer, boxes.

VII. Summary

In this paper, we built upon the HAC model and applied it to the understanding of and exploiting inter-domain relationships for performance analysis of dynamically decomposed applications. We focused on SAMRAI as our case study and proposed a new projection of data collected in the hardware domain on to the communication graph as well as a new scalable visualization of the resulting coalesced information. Using these projections led to a rapid diagnosis of and mitigation strategy for a previously elusive scaling bottleneck in the library that is hard to detect using conventional tools. Our new insights resulted in a 22% performance improvement for a 65,536-core run of SAMRAI on Blue Gene/P. We understand that to ultimately eliminate this problem, we must deploy a different virtual topology that prevents this scenario and we plan to rewrite the load balancers in SAMRAI based on the results presented in this paper.

We believe that the process of creating projections outlined in this paper and the visualization techniques presented are generally applicable and especially useful for adaptive scientific codes. The scalable tree visualizations can be used to identify hot-spots or scalability problems at any level of the tree. Generalizing these visualizations of the communication domain to arbitrary graphs requires feature finding to coalesce uninteresting nodes and focus on sub-domains that appear to have potential performance problems.

Acknowledgment

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). Accordingly, the U.S. government retains a non-exclusive, royalty-free license to publish or reproduce the published form of this contribution, or allow others to do so, for U.S. government purposes.

Neither the U.S. government nor Lawrence Livermore National Security, LLC (LLNS), nor any of their employees makes any warranty, expressed or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, or process disclosed, or represents that its use would not infringe privately owned rights. The views and opinions of authors expressed herein do not necessarily state or reflect those of the U.S. government or LLNS, and shall not be used for advertising or product endorsement purposes.

An award of computer time was provided by the Innovative and Novel Computational Impact on Theory and Experiment (INCITE) program. This research used resources of the Argonne Leadership Computing Facility at Argonne National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under contract DE-AC02-06CH11357.

REFERENCES

[1] M. Schulz, J. Levine, P.-T. Bremer, T. Gamblin, and V. Pascucci, "Interpreting performance data across intuitive domains," in *Parallel Processing (ICPP), 2011 International Conference on*, September 2011, pp. 206–215.

[2] R. D. Hornung and S. R. Kohn, "Managing application complexity in the samrai object-oriented framework," *Concurrency and Computation: Practice and Experience*, vol. 14, no. 5, pp. 347–368, 2002.

[3] B. T. Gunney, A. M. Wissink, and D. A. Hysom, "Parallel clustering algorithms for structured amr," *Journal of Parallel and Distributed Computing*, vol. 66, no. 11, pp. 1419 – 1430, 2006.

[4] B. T. N. Gunney, "Large-scale dynamically adaptive structured AMR," SIAM Conference on Parallel Processing for Scientific Computing, Februrary 2010, uCRL-PRES-422996.

[5] M. Schulz and B. R. de Supinski, "A Flexible and Dynamic Infrastructure for MPI Tool Interoperability," in *Proceedings of the 2006 International Conference on Parallel Processing*, Aug. 2006.

[6] R. Bell, A. Malony, and S. Shende, "ParaProf: A Portable, Extensible, and Scalable Tool for Parallel Performance Profile Analysis," in *Proceedings of the International Conference on Parallel and Distributed Computing (Euro-Par 2003)*, Aug. 2003, pp. 17–26.

[7] W. E. Nagel, A. Arnold, M. Weber, H. C. Hoppe, and K. Solchenbach, "VAMPIR: Visualization and analysis of MPI resources," *Supercomputer*, vol. 12, no. 1, pp. 69–80, 1996.

[8] M. Schulz, J. Galarowicz, D. Maghrak, W. Hachfeld, D. Montoya, and S. Cranford, "Open|speedshop: An open source infrastructure for parallel performance analysis," *Scientific Programming*, vol. 16, no. 2-3, pp. 105–121, 2008.

[9] F. Wolf, B. Wylie, E. Abraham, D. Becker, W. Frings, K. Fuerlinger, M. Geimer, M.-A. Hermanns, B. Mohr, S. Moore, and Z. Szebenyi, "Usage of the SCALASCA Toolset for Scalable Performance Analysis of Large-Scale Parallel Applications," in *Proceedings of the 2nd HLRS Parallel Tools Workshop*, Stuttgart, Germany, july 2008. [Online]. Available: http://www.cs.utk.edu/~karl/research/pubs/WOLF_2008_Scalasca.pdf

[10] J. Vetter and C. Chambreau, "mpiP: Lightweight, Scalable MPI Profiling," http://mpip.sourceforge.net.

[11] M. Schulz and B. R. de Supinski, "P^N MPI Tools: a Whole Lot Greater than the Sum of their Parts," in *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, 2007.

[12] O. Zaki, E. Lusk, W. Gropp, and D. Swider, "Toward Scalable Performance Visualization with Jumpshot," *International Journal of High Performance Computing Applications*, vol. 13, no. 3, pp. 277–288, 1999.

[13] N. Tallent, J. Mellor-Crummey, M. Franco, R. Landrum, and L. Adhianto, "Scalable Fine-grained Call Path Tracing," in *Proceedings of the International Conference on Supercomputing*, Jun. 2011.

[14] N. R. Tallent, J. M. Mellor-Crummey, L. Adhianto, M. W. Fagan, and M. Krentel, "Diagnosing performance bottlenecks in emerging petascale applications," in *Proceedings of IEEE/ACM Supercomputing '09*, Nov. 2011.

[15] K. A. Huck and A. D. Malony, "Perfexplorer: A performance data mining framework for large-scale parallel computing," in *Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, ser. SC '05. IEEE Computer Society, 2005.

[16] L. D. Rose, Y. Zhang, and D. A. Reed, "Svpablo: A multi-language performance analysis system," Sep. 1999.

[17] J. Mellor-Crummey, R. Fowler, and G. Marin, "HPCView: A tool for top-down analysis of node performance," *The Journal of Supercomputing*, vol. 23, pp. 81–101, 2002.

[18] A. G. Landge, J. A. Levine, K. E. Isaacs, A. Bhatele, T. Gamblin, M. Schulz, S. H. Langer, P.-T. Bremer, and V. Pascucci, "Visualizing network traffic to understand the performance of massively parallel simulations," in *IEEE Symposium on Information Visualization (INFOVIS'12)*, Seattle, WA, October 14-19 2012, LLNL-CONF-543359.

[19] P. Elias, A. Feinstein, and C. Shannon, "A note on the maximum flow through a network," *Information Theory, IRE Transactions on*, vol. 2, no. 4, pp. 117 –119, december 1956.