LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

LLNL-TR-523452

Simplifying Performance Analysis of Large-scale Adaptive Scientific Applications

A. Bhatele, T. Gamblin, B. T. N. Gunney, M. Schulz, P. T. Bremer

January 19, 2012

This work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.

Simplifying Performance Analysis of Large-scale Adaptive Scientific Applications

Abhinav Bhatele, Todd Gamblin, Brian T. N. Gunney, Martin Schulz, Peer-Timo Bremer
Center for Applied Scientific Computing
Lawrence Livermore National Laboratory, P. O. Box 808, Livermore, CA 94551, USA
{bhatele, tgamblin, gunneyb, schulzm, ptbremer}@llnl.gov

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 exploit a recent idea of projecting and visualizing performance data on the communication and hardware domain for faster, more intuitive analysis of applications. We leverage several performance analysis and visualization tools to showcase the discovery of scalability bottlenecks in a structured AMR library. Using novel techniques to project per-phase timing data, application data, and communication data on a communication graph, we identify a previously elusive scaling bottleneck in the library. We present solutions that mitigate this problem, resulting in 22% improvement in the performance for a 65,536-core run on an IBM Blue Gene/P system.

Categories and Subject Descriptors

C.4 [**Performance of Systems — Performance attributes**]

General Terms

Measurement, Performance

Keywords

data attribution, performance analysis, visualization, scalability, adaptivity

1. INTRODUCTION

Adaptivity is becoming ubiquitous across the high performance computing (HPC) software stack. At the application level, computational techniques such as adaptive mesh refinement (AMR) are used to reduce the cost of numerical solvers by increasing the solver resolution only for areas of the domain where higher resolution is needed [1, 2]. At a lower level, to ensure good performance, runtime libraries for adaptive codes dynamically redistribute the work generated by their refinement techniques among processors in a parallel system [3, 4]. Future exascale machines are expected to require adaptivity at even lower levels of the software stack, to distribute computational tasks among potentially heterogeneous compute resources, to balance power requirements and to adjust to hardware faults [15].

Adaptivity at any level presents a special challenge for users and developers of performance tools because it makes tying measurements to their root causes a difficult task. In a static domain decomposition, per-process performance measurements can be easily associated with specific application data. In adaptive codes such as AMR, a measurement on one process may potentially be associated with many pieces of the application data. Further, adaptive applications do not strictly follow the bulk-synchronous computational model traditionally used in HPC. Interspersed with the computation and ghost exchange phases of traditional HPC applications, an adaptive code may reorganize work using sophisticated overlay networks and communication patterns. Performance measurements must therefore be carefully attributed to particular phases of computation and specific application data, in order to clearly identify the causes of observed performance problems.

Correctly attributing performance measurements to their causes requires understanding the complex relationships between different performance domains. Schulz *et al.* have proposed techniques for understanding the relationship between statically decomposed application domains, HPC hardware and inter-process communication [17]. In this paper, we focus on understanding and exploiting such relationships for dynamically decomposed application domains. In particular, we demonstrate the utility of projecting performance data onto more intuitive domains to detect a subtle performance problem in a highly scalable AMR code.

We introduce novel performance visualization and analysis techniques and make the following key contributions:

1. We introduce methodologies to track performance in adaptive applications.
2. We use these methodologies to study scalability and performance of a structured AMR code.
3. We showcase visualization techniques that exploit the relationship between phase-based timing data and the communication domain of the application.
4. We present techniques to attribute application-specific data such as flow of AMR patches in an overlay network in the communication domain.

ICS'12, June 25–29, 2012, Venice, Italy.

Using the methodologies and visualization techniques mentioned above, we demonstrate the careful unscrambling of otherwise tangled measurements caused by adaptive systems. By attributing performance measurements directly to their causes, we are able to visualize performance measurements in the domains most intuitive to the user, which are not necessarily where they are measured. We also show that these techniques allow us to pinpoint a previously elusive scalability problem within the load balancer of SAMRAI [7, 9], a highly scalable structured AMR library used extensively in several large-scale DOE applications. We use insights gained from projecting data on the communication domain to perform targeted optimizations and to improve the performance of SAMRAI. Results for a 65,536-core run on a Blue Gene/P system show performance improvements of nearly 22%. Our techniques also provide insights that will enable us to redesign the load balancing algorithm to remove the scalability problem completely in the future.

2. PROJECTING DATA ACROSS DOMAINS

Schulz *et al.* 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, comprised of information from the application's simulated physical domain; and the communication domain, comprised of abstract graphs with processes as nodes and communication between them represented as edges. This framework is called the *HAC model* (see Figure 1).

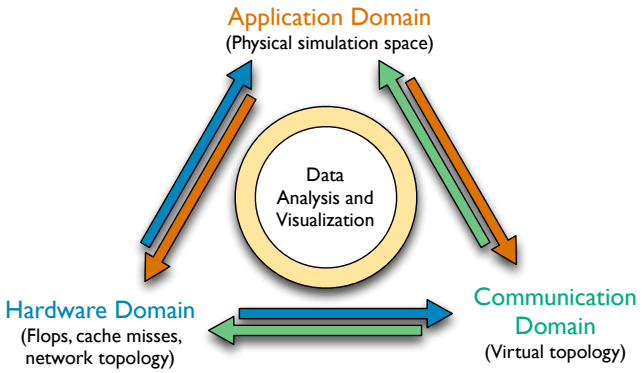

Figure 1: The HAC model

Root causes of performance problems lie in one of these domains, but their symptoms may lie in another. We propose 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 an AMR code, however, 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 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 will pass 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.

3. STRUCTURED ADAPTIVE MESH REFINEMENT

In this paper, we study an application domain that is complex and difficult to scale. *Define: What is a cell and what is a patch* Structured adaptive mesh refinement (SAMR) is a popular AMR technique. In SAMR, the mesh is a hierarchy of successively finer mesh *levels* (Figure 2 shows a SAMR mesh.) The bottom-most level spans the entire problem domain and its mesh is comprised of the largest cells. Meshes in successively higher levels have increasing cell resolution. AMR meshes are not globally refined; rather, each level is comprised of a set of patches that contain only the cells in areas needing refinement. Higher levels span smaller and smaller portions of the computational domain. For data-parallel implementations, the mesh can be partitioned by assigning one or more patches to a process. Large patches may be broken into smaller patches by the partitioning process. Standard SAMR applications consist of two operations: local computation on individual patches, and coordinated data exchange between neighboring/overlapping patches. The latter requires communication between the processes that own the patches.

SAMR introduces a great deal of complexity in parallel mesh generation, as each level must be created based on the complexity of solution features. This is a multi-step process. Whether creating the level for the first time or replacing an existing level, the SAMR library examines the solution at the next coarser level to determine which grid cells need more resolution. These cells are tagged and a reasonable set of non-overlapping boxes is computed to cover them. For parallel implementations, the boxes are partitioned among the processes and subsequently used to create the new level. Finally, the new level is populated with data from existing levels and after which the the old level is discarded.

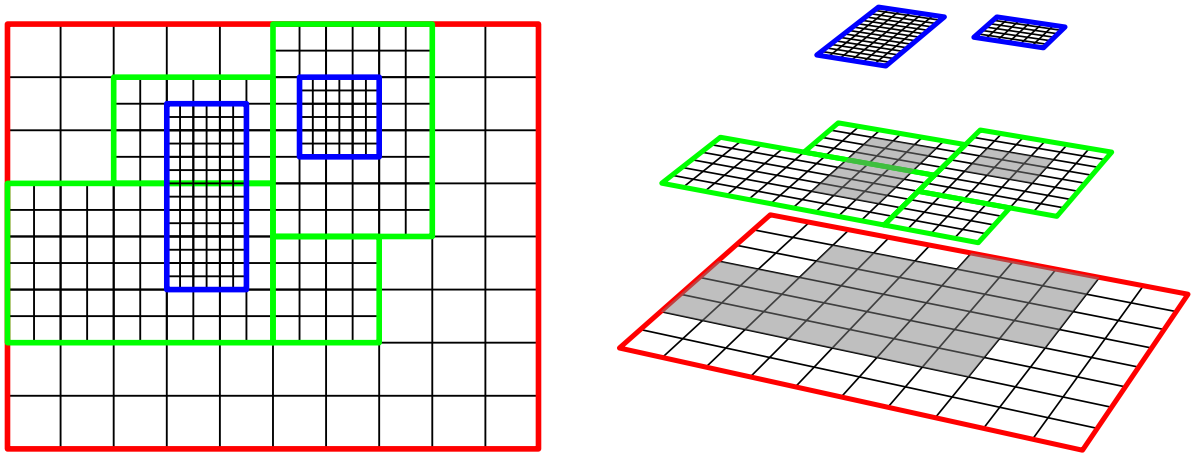

Figure 2: A simple two-dimensional structured AMR mesh with three levels of refinement (left). The same mesh is shown as a hierarchy on the right. Cells that are refined are shown in gray.

The cost of SAMR remeshing is low for sequential programs but grows quickly as a job is scaled in parallel. Conventional approaches for creating a mesh level with N boxes have at least $\mathcal{O}(N)$ complexity. In a parallel algorithm, each of the P processes must have at least one box, so the remeshing cost is at least $\mathcal{O}(P)$ for both strong and weak scaling. Conventional approaches do not parallelize easily, so their sequential versions are often used, leading to a cost that grows like $\mathcal{O}(P)$ in parallel. SAMRAI, a popular SAMR library, uses a distributed mesh management approach to parallelize remeshing to a large extent [8]. However, some significant challenges still remain. Compared to the main computation, SAMR remeshing has a high communication-to-computation ratio. Also, SAMRAI's extensive use of *asynchronous* point-to-point communication makes parallel performance difficult to analyze.

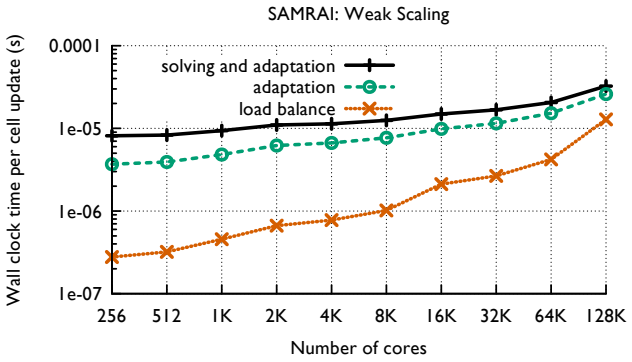

Figure 3: Weak scaling performance of SAMRAI for the linear advection benchmark on Blue Gene/P.

3.1 Current Performance

All runs in this paper use a weak scaling linear advection (LinAdv) benchmark from the SAMRAI distribution. A sinusoidal wave is simulated passing through the domain and the problem size is grown by tiling the domain. Three levels of refinements and a refinement ratio of 2 are used. After running a few time steps on the coarsest level, regeneration of level 2 (the finest level) is selected for data collection. We used the Blue Gene/P systems at Argonne National Laboratory (Challenger and Intrepid) for the runs. Intrepid is a 557.1 TFlop/s Blue Gene machine with 163,840 PPC450 cores and a three-dimensional torus interconnect.

Figure 3 presents the current scaling characteristics of the LinAdv benchmark. The top (black) line represents the overall iteration time, which includes the time for computation (solving) and remeshing (adaptation). The benchmark uses weak scaling, so perfect scaling is a flat line. The computation or solving phase is entirely local with the exception of ghost exchange, and it scales well, so we omit it here. The adaptive remeshing phase of SAMRAI (second green line) grows slower than the characteristic $\mathcal{O}(P)$ of conventional SAMR approaches. However, at 8,192 cores, remeshing starts taking more time than computing the solution. This represents a serious scaling problem, as large-scale runs will not achieve good parallel efficiency if most of their time is spent remeshing.

A significant fraction of remeshing cost beyond 8,192 cores comes from load balancing (third orange line). The load balancing step has an expected cost of $\mathcal{O}(\log P)$ but exhibits a behavior closer to $\mathcal{O}(P)$, making it an ideal candidate for further examination using our tools. The remainder of this paper describes the steps we took to diagnose its root cause. For completeness, the next section describes the load balancing algorithm used in SAMRAI.

3.2 Load Balancing in SAMRAI

Load balancing in SAMRAI has three phases: load distribution, mapping generation, and overlap update. *Load distribution* starts with a set of boxes describing the extent of the new level. These boxes represent new work load, but they are not evenly distributed. SAMRAI attempts to assign boxes to processes in a way that balances the number of grid cells given to each process. Almost always, this requires some boxes to be subdivided into smaller boxes. *Mapping generation* constructs a mapping from pre-balance boxes to post-balance boxes. This accounts for where the boxes traveled on the network as well as whether the boxes were subdivided at all. *Overlap update* uses the generated mapping to update information about overlapping regions between the

new boxes and some reference boxes in the existing hierarchy, given the changes introduced by load distribution.

We will describe the load distribution steps for a simple three-process group, drawn as a tree in Figure 4, before describing how to recursively build up the steps for bigger groups. At the start of the algorithm, each process has some load imbalance (excess or deficit load). The leaves, *child 1* and *child 2*, send the imbalance information to the parent, including any excess work. The parent places excess work from itself and its children into a separate container designated as *unassigned*. It uses the unassigned work to fill deficits where needed. If the parent has a deficit, it shifts some unassigned work to itself. For each child that has a deficit, it sends some unassigned work to that child After shifting work to fill the deficits, no unassigned work remains because the net imbalance at the parent is zero. This is true here because the parent is the root of the entire tree. For deeper trees, this will not be the case.

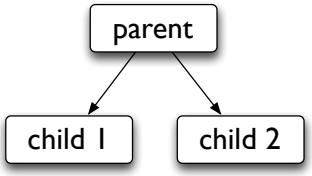

Figure 4: Load balancing tree for partitioning three processes. The parent partitions for all three nodes.

Figure 5 shows a deeper tree with seven processes. To load balance this tree, *parent 1*, *child 1* and *child 2* apply the above algorithm. Independently, *parent 2*, *child 3* and *child 4* do the same. Because parents 1 and 2 are not at the root of the tree, they may see an imbalance that represents the imbalance of their subtree. To eliminate this imbalance, they participate as children of the *grandparent* (see Figure 6) in another recursive instance of the same algorithm. For deeper trees, the grandparents rely on their parents to cancel their imbalances, and so on. The recursion stops at the root of the tree, where the imbalances always cancel out.

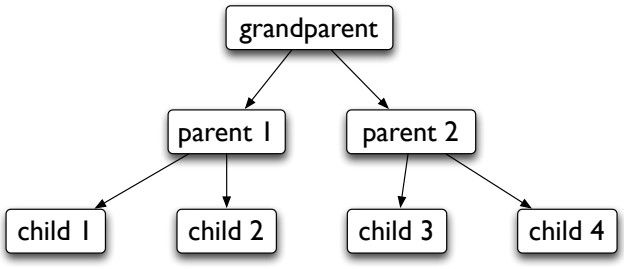

Figure 5: Load balancing tree of seven processes.

Once a process sends its excess boxes away, it loses track of them. The mapping generation phase informs each process of the final destinations of its boxes by sending this information back to the original owner along the same path the boxes took to their destinations. To each box in transit within the tree, we attach metadata describing the path it has taken. When a box is broken into smaller boxes, the smaller boxes start with the path of the broken box and build from there. Using this approach, we expect that no box will travel more than $\mathcal{O}(\log P)$ hops in the tree network. We assume that the cost of shipping metadata around the network is small compared to the cost of shipping application compute data, which is much larger, so we do not consider network bandwidth in this assessment.

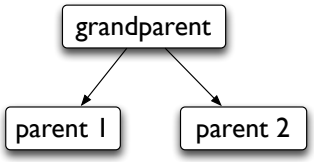

Figure 6: Tree for partitioning seven processes. The grandparent partitions for itself and the parents.

At the end of load balancing, SAMRAI must compute the overlaps between the new boxes on each level in the hierarchy. A search for the overlaps that a process's $\mathcal{O}(N/P)$ boxes may have with the $\mathcal{O}(P)$ boxes on the level would cost $\mathcal{O}((N/P)\log P) + \mathcal{O}(N \log N)$. In addition, the search requires an all-gather communication to place the existing level's N boxes in local memory to enable the search. SAMRAI can avoid this if it is given a mapping between the pre-balance boxes and the post-balance boxes. So load balancing must include informing owners of the pre-balance boxes where their original work ended up. This is the mapping generation sub-phase of load balancing.

One or more of the three load balancing sub-phases is responsible for the scalability problems that we see in the timing plot. In the next section, we describe the techniques we used to isolate the specific sub-phase and root cause.

4. PERFORMANCE VISUALIZATIONS

We now present a succession of unsuccessful and successful attempts at discovering the scalability issues in SAMRAI through visualization of performance data projected among the three HAC domains. Based on the scaling behavior of particular phases in SAMRAI, we identified in Section 3 that the problem lies within the load balancing phase. However, this is not sufficient to point us to the root cause of the problem. Hence, the rest of the paper discusses performance data and its visualization specifically for the load balancing phase. As we discuss our projections and visualizations, we also highlight various tools that were used to obtain the performance data and create the visualizations.

4.1 Unhelpful Visualizations

Timing information obtained from running SAMRAI tells us that the load balancing phase, specifically the communication in that phase, does not scale well as we increase the number of processors. Hence, we started with measuring the communication characteristics for the load balancing phase. In order to obtain this data, we use the communication matrix module in P^NMPI [16] to intercept MPI calls and record a communication matrix. We use yEd, a graph editor, to visualize the Graph Modeling Language (GML) files output by this P^NMPI module.

Figure 7 shows two visualizations of the communication graph for 256 processes during the load balancing phase of SAMRAI. The graph was obtained by profiling the LinAdv benchmark on Blue Gene/P. On the left, the communication is presented in the form of a matrix generating using `mat-`

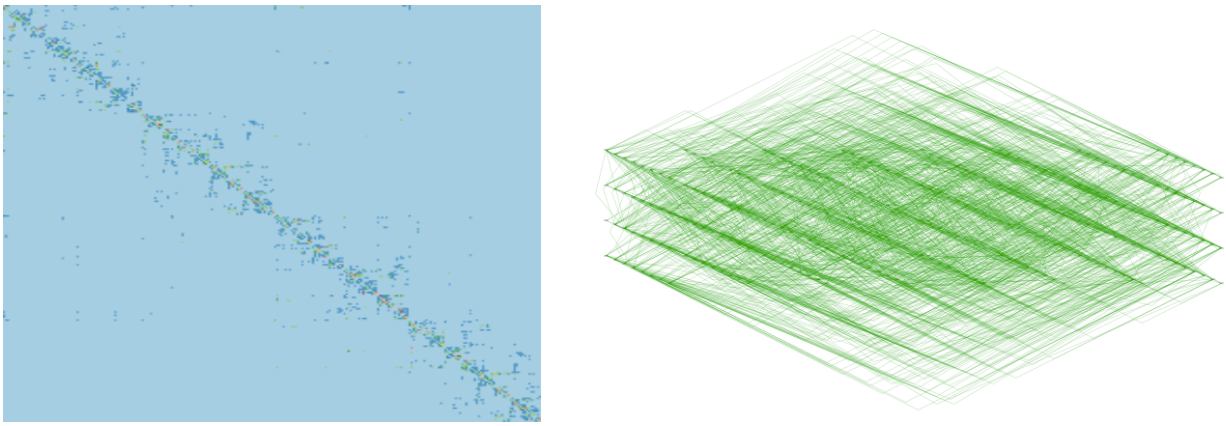

Figure 7: Communication graph of 256 processes during the load balancing phase.

`plotlib`. We can deduce that the communication is sparse but we can't see any trends that would affect the performance. On the right, the 256 processes are laid out in a different fashion and connected by edges based on the communication matrix. This visualization also does not convey any meaningful information. It can be deduced that the communication is not near-neighbor but many-to-many, however, no inferences can be drawn about its efficiency.

One hypothesis for the poor scaling performance of the load balancing phase was that the high diameter of torus networks like those on Blue Gene/P was leading to network latency problems for larger numbers of cores. For each box that is moved, the tree overlay network used by the SAMRAI load balancer guarantees a logarithmic number of *virtual* hops on the network itself. However, it does not guarantee a logarithmic number of *physical* hops on the torus network. We projected our communication domain data into the hardware domain by converting MPI ranks (endpoints) to hardware addresses, and we computed the number of torus hops required for each single link on the tree network.

Figure 8 shows a histogram with our results. Most messages sent during the load balancing phase travel fewer than four hops. Our projection thus tells us that this is not a sufficient number of hops to cause a latency problem in the network. Next, we hypothesized that too many boxes were being transferred over the hardware network between communicating pairs, causing a bandwidth problem. From our communication profiling, we were able to measure that most messages are smaller than 1200 bytes, which is below the latency-bandwidth product for Blue Gene/P. We can thus rule out network contention as the cause of the scalability problem. Since, the communication graph did not provide insights into the issue, we decided to obtain detailed timing information about the time spent in computation and communication on each process.

We used mpiP [21] to obtain profile information for individual MPI processes. It 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 by using `MPI_Pcontrol` and in our case we use this feature to focus on the details of the three sub-phases of the load balancing algorithm with the intent to assign blame to specific sub-phases (as described

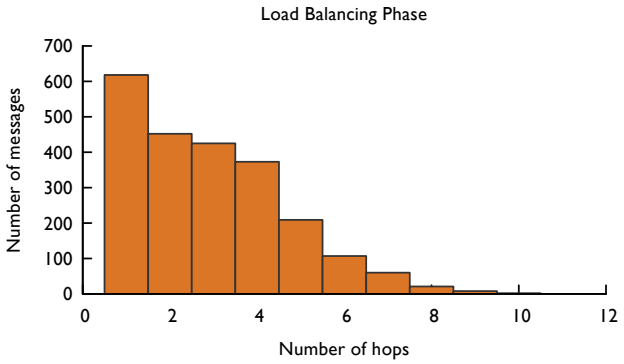

Figure 8: Histogram of the number of hops traveled by messages during the load balancing phase.

in Section 3.2).

However, 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. We use P^NMPI to virtualize mpiP so that multiple code regions can be profiled at once using multiple instances of mpiP. We implemented a P^NMPI "switch" module that uses the integer passed in the `MPI_Pcontrol` call to multiplex different instances of mpiP for different phases, yielding distinct profiles for each phase. This enabled us to obtain the time spent in the three load balancing phases outlined above.

Figure 9 shows the sum of times spent by all MPI processes in a particular load balancing sub-phase. The three sub-phases, load distribution, mapping generation, and overlap update are referred to as phase 1, 2 and 3 respectively in the rest of the paper. It appears that as we scale to more and more processors, we spend increasingly more time in phase 3. However, since each successive phase waits for the completion of the previous one, the excessive time spent in phase 3 might be a result of delays in phase 2 or even phase 1. mpiP outputs timing information for each MPI process and we next plotted the times spent by individual MPI processes in each sub-phase of load balancing on 256 cores (see Figure 10). We can see that several processes are actually spending a significant amount of the total load balancing time in phase 1 (tall red bars) rather than phase 3.

This does *not* show up in a traditional profiler because the results are presented in aggregate, and we lose information about behavior of the smaller number of slow processes in phase 1. We hypothesize that processes that have a larger fraction of the time spent in phase 2 or 3 might be waiting for processes stuck in phase 1, *i.e.*, load distribution.

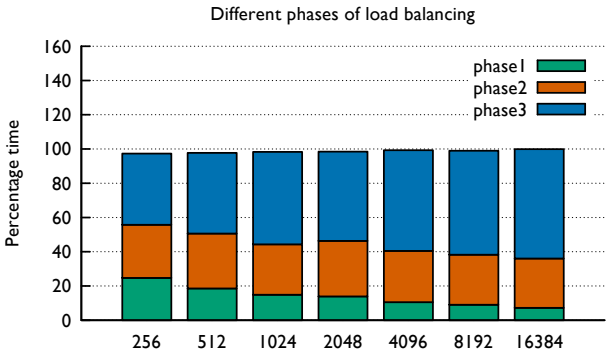

Figure 9: Sum of times spent by all MPI processes in different load balancing sub-phases.

Plotting timing information for different phases against a linear ordering of MPI processes by their ranks only gets us so far and is also unintuitive to the end user, since rank order and its mapping to compute resources is arbitrarily imposed by the MPI library and the scheduler. We must therefore map this information to another domain and visualize it there to truly understand what is going on. In particular, in the next two subsections, we project this information onto the hardware and communication domains to gain further insights into the problem.

4.2 Projections on the Hardware Domain

We use the timing information collected from mpiP (that has been presented in Figure 10) to color the nodes on the physical torus by the time spent within a particular load balancing sub-phase. We use Boxfish, a lightweight, interactive visualization tool useful for projecting performance data onto the three-dimensional torus (the hardware domain) of Blue Gene machines.

Figure 11 shows the view generated by Boxfish when we color each node on the torus by the time processes spent in phase 1 of the load balancer. We see that the processes that spend the most time in this phase are on the fourth plane of the torus. A hot plane like this can indicate contention, but we know (from the communication matrix) that messages sent in the load balance phase are very small and that the time here is most likely spent waiting.

However, the correlation stands out very strongly in the figure. We looked at the layout of the load balance tree itself, and we hypothesized that this plane was likely part of a subtree in the SAMRAI load balancer's overlay network. This led us to project our performance information back onto the SAMRAI communication domain, where we can see the load balance traffic clearly.

4.3 Projections on the Communication Domain

As described in detail in Section 3.2, SAMRAI communicates load variations and work loads (boxes) along a virtual

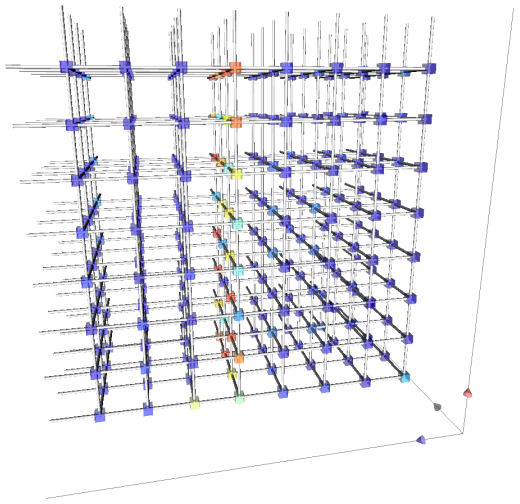

Figure 11: Boxfish showing the 256 nodes on a $8 \times 4 \times 8$ torus colored by the time spent in sub-phase 1 of load balancing.

tree network. To understand the communication behavior, we look at projections of phase timing data onto the communication domain, *i.e.*, the load balancing tree. We construct a pairwise communication graph among the MPI processes for the load balancing phase, which looks like a binary tree, and we color nodes by the time they spend in different sub-phases of the load balancer.

Figure 12 (left) 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 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.

Figure 12 (right) colors the nodes in the same tree network by the initial load on each process before load balancing starts. We see that loads of individual nodes are randomly distributed over the tree and do not appear to be correlated to the phase timings in the left diagram. However, the total loads for each of the four sub-trees give us some indication of what we'll find next. Three of the four sub-trees (in various shades of blue in the left diagram) have 2.83, 2.87 and 3.1% more load than the average whereas one (the sub-tree in red) has 9% less load than the average. This suggests that load has to flow from three overloaded sub-trees to the underloaded one to achieve load balance.

Since we established that processes in one sub-tree are waiting for their parent to send them load, we color and weight each edge by the number of boxes that the edge's child node receives from the parent. Figure 13 shows the resulting graph. We can now see a flow bottleneck that happens from node 0 to node 129 and node 129 to node 130

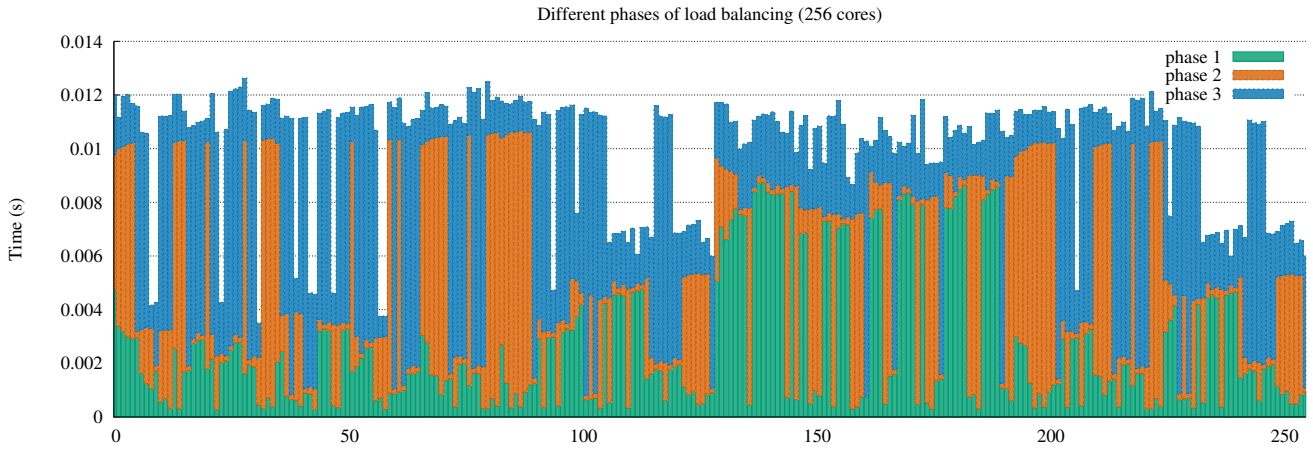

Figure 10: Time spent in different load balancing sub-phases for 256 MPI processes on Blue Gene/P.

towards the top of the slowest sub-tree. Coloring the nodes by the phase 2 or 3 timings did not present such a correlation. However, we created similar visualizations for phase 1 timings on larger number of processors and we noticed the same problem – a portion of the tree spends most of its time waiting for boxes to percolate down the tree.

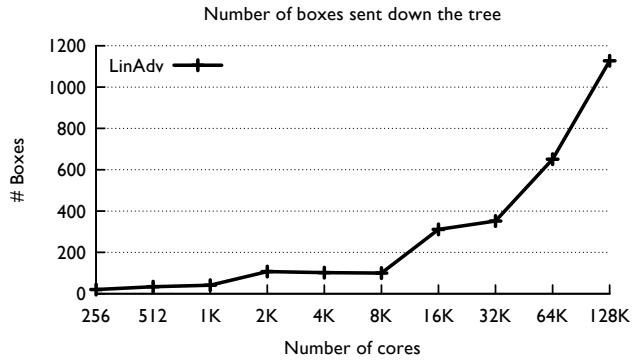

Figure 14: Maximum number of boxes sent on any edge of the tree as a function of the number of processors (on Blue Gene/P).

The problem becomes worse as we run on larger number of processors because the maximum number of boxes to be sent on a particular edge continues to increase. We plot this in Figure 14 as a function of the number of processors. 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 Figure 3. A tree network was used for load balancing to place an upper bound on the number of hops that load may have to travel on the virtual network. However, the same tree network, because it funnels load from subtrees through sparse edges near the root, is susceptible to small variations in the initial distribution of load. This leads to a flow problem [5] during load balancing where a large number of boxes have to go through a single edge to replenish an under-utilized sub-tree.

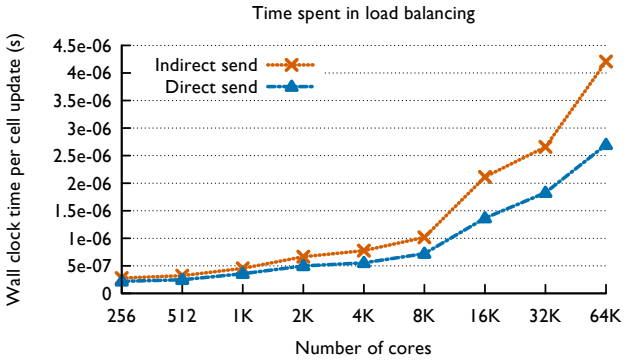

Figure 15: Reduction in load balancing time by sending patches directly to their new destinations.

5. SOLUTION AND NEW RESULTS

In this section, we present preliminary solutions to mitigate the issue observed in the load balancing phase. To reiterate, scalability in the load balancing phase is hindered by a flow problem in the virtual tree topology: a particular subtree needs to receive load from the remaining subtrees and hence all traffic (in terms of metadata for boxes) has to flow through one particular edge in the load distribution phase. We understand that eliminating this problem requires the 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.

As initial steps and proof of concept, we 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. We changed the algorithm to send the data directly to the originator instead of through the tree overlay structure, allowing us to eliminate the need for the extra data. As a consequence of not sending data along the tree, the information gets back to the originator in fewer hops (on both the tree and the physical network) and we send less data per box.

This "direct send modification" leads to a reduction in the

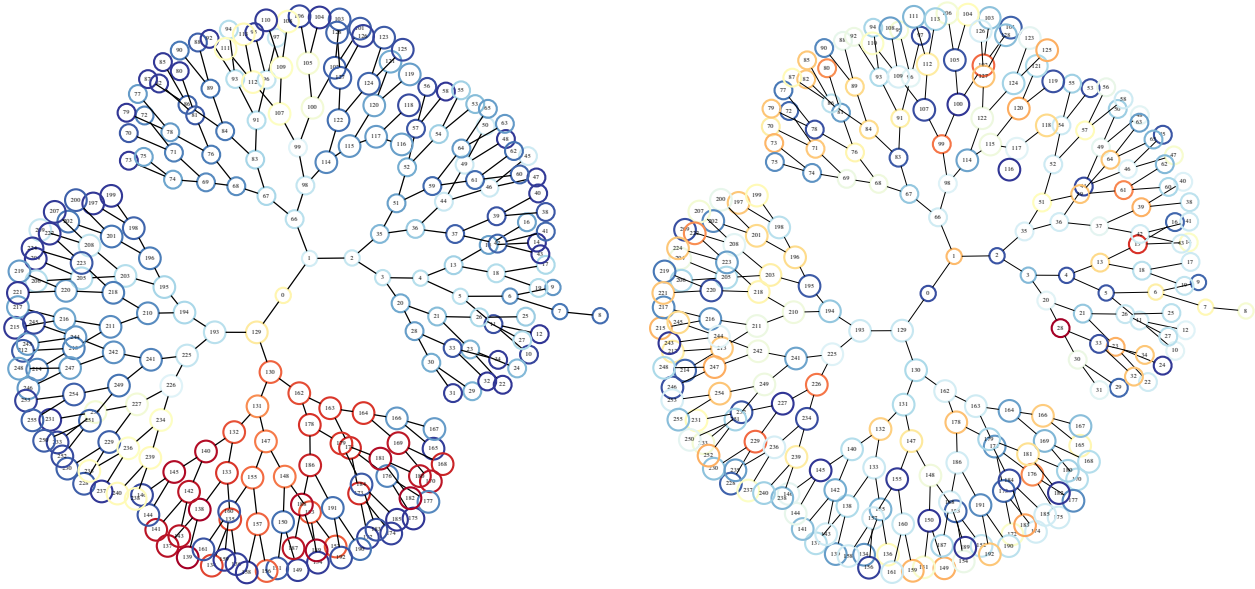

Figure 12: Phase timing data and load information visualized in the communication domain – the virtual tree network of the load balancing phase (256 processes on Blue Gene/P). On the left, the nodes are colored by the time spent in phase 1 *i.e.*, load distribution. On the right, the nodes are colored by the load in terms of the number of cells.

load balancing time (as shown in Figure 15). Compared to the old scheme, using direct sends, we get 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 sixes and recorded the maximum number of boxes sent on any edge along with the timing information. Figure 16 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. This reduces to 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 Figure 17). 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 reduces 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

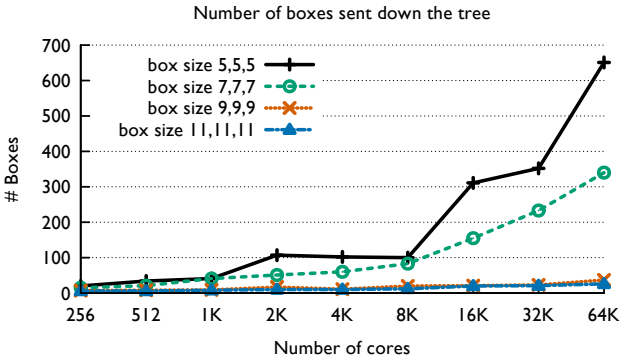

Figure 16: Reduction in maximum number of boxes sent on any edge of the tree by increasing the size of each box.

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.

6. RELATED WORK

Prior work has investigated the parallel measurement of per-process load balance data and its attribution to source code [6, 10, 18, 19]. Two of these techniques complement this work by enabling scalable parallel data collection for the type of per-process data we collect here, and by allowing code to be automatically sliced into phases and regions based on callpaths. However, these techniques do not allow the projection of application or other domain data onto abstract communication graphs to analyze the root causes of performance problems.

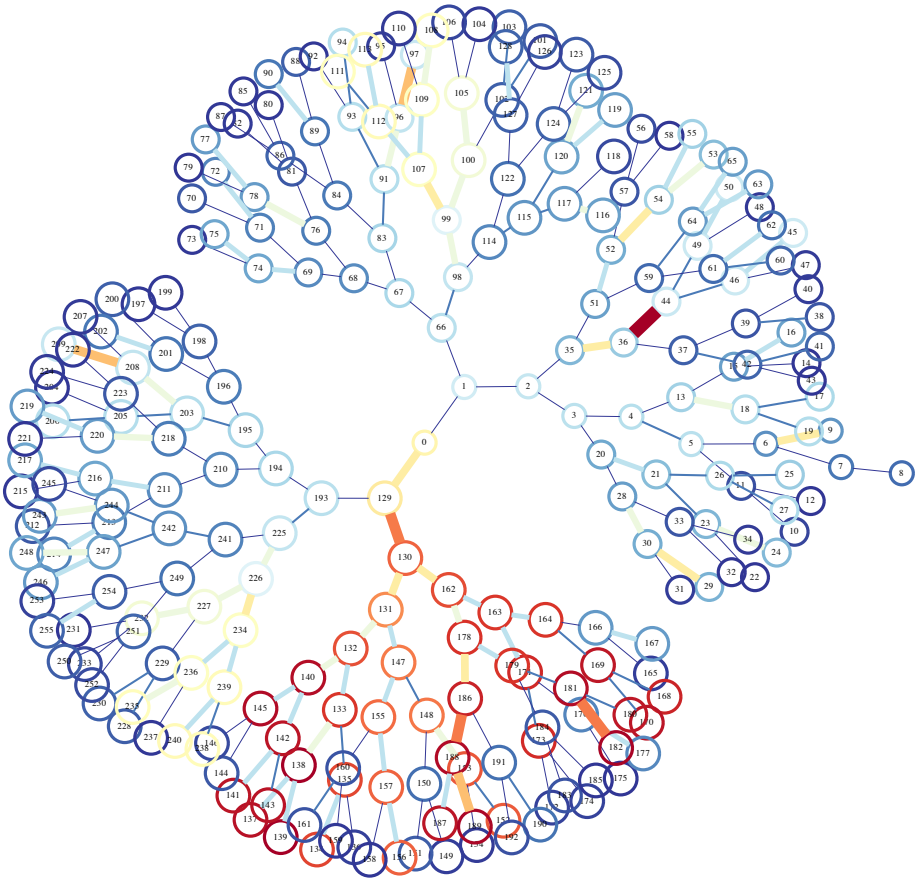

Figure 13: Phase timing data visualized in the communication domain – the virtual tree network of the load balancing phase (256 processes on Blue Gene/P). The nodes are colored by the time spent in phase 1 and the edges by the number of boxes received by a child from its parent.

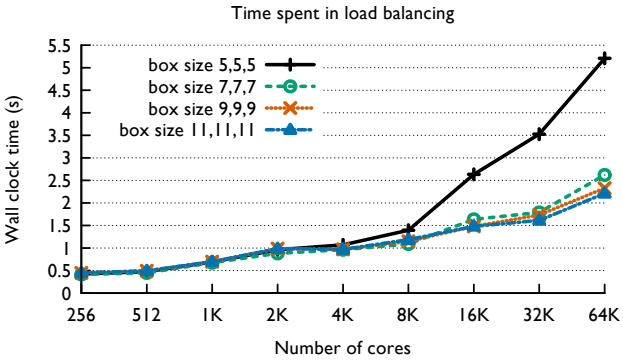

Figure 17: Improvement in load balancing time by making fewer boxes.

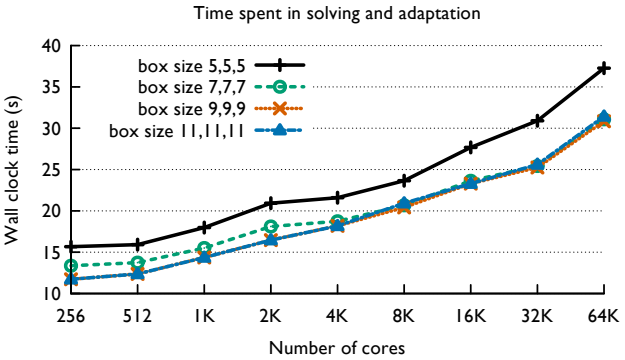

Figure 18: Improvement in overall time (solving plus adaptation) by making fewer boxes.

Tallent *et al.* have investigated automatic discovery of scalability bottlenecks at particular phases of program execution, also based on callpaths [20]. 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 [11–14, 22] 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.

7. SUMMARY

We have 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 specially 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-TR-523452).

8. REFERENCES

[1] M. J. Berger and P. Colella. Local adaptive mesh refinement for shock hydrodynamics. *J. Comput. Phys.*, 82:64–84, May 1989.

[2] M. J. Berger and J. Oliger. Adaptive mesh refinement for hyperbolic partial differential equations. *Journal of Computational Physics*, 53(3):484 – 512, 1984.

[3] R. K. Brunner and L. V. Kalé. Handling application-induced load imbalance using parallel objects. In *Parallel and Distributed Computing for Symbolic and Irregular Applications*, pages 167–181. World Scientific Publishing, 2000.

[4] U. Catalyurek, E. Boman, K. Devine, D. Bozdag, R. Heaphy, and L. Riesen. Hypergraph-based dynamic load balancing for adaptive scientific computations. In *Proc. of 21st International Parallel and Distributed Processing Symposium (IPDPS'07)*, pages 1–11. IEEE, 2007. Best Algorithms Paper Award.

[5] P. Elias, A. Feinstein, and C. Shannon. A note on the maximum flow through a network. *Information Theory, IRE Transactions on*, 2(4):117 –119, december 1956.

[6] T. Gamblin, B. R. de Supinski, M. Schulz, R. J. Fowler, and D. A. Reed. Scalable load-balance measurement for SPMD codes. In *Supercomputing 2008 (SC'08)*, pages 46–57, Austin, Texas, November 15-21 2008.

[7] B. T. Gunney, A. M. Wissink, and D. A. Hysom. Parallel clustering algorithms for structured amr. *Journal of Parallel and Distributed Computing*, 66(11):1419 – 1430, 2006.

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

[9] R. D. Hornung and S. R. Kohn. Managing application complexity in the samrai object-oriented framework. *Concurrency and Computation: Practice and Experience*, 14(5):347–368, 2002.

[10] K. Huck and J. Labarta. Detailed Load Balance Analysis of Large Scale Parallel Applications. In *International Conference on Parallel Processing (ICPP)*, San Diego, CA, USA, September 13-16 2010.

[11] 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*, SC '05. IEEE Computer Society, 2005.

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

[13] W. E. Nagel, A. Arnold, M. Weber, H. C. Hoppe, and K. Solchenbach. VAMPIR: Visualization and analysis of MPI resources. *Supercomputer*, 12(1):69–80, 1996.

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

[15] V. Sarkar. Exascale software study: Software challenges in extreme scale systems. Technical report, 2009.

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

[17] 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*, pages 206–215, September 2011.

[18] 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*, June 2011.

[19] N. R. Tallent, L. Adhianto, and J. M. Mellor-Crummey. Scalable identification of load

imbalance in parallel executions using call path profiles. In *Proceedings of IEEE/ACM Supercomputing '10*, Nov. 2010.

[20] 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.

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

[22] 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.