LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

LLNL-CONF-402967

# Lessons learned at 208K: Towards Debugging Millions of Cores

G. L. Lee, D. H. Ahn, D. C. Arnold, B. R. de Supinski,
M. Legendre, B. P. Miller, M. W. J. Schulz, B. Liblit

April 15, 2008

SuperComputing 2008
Austin, TX, United States
November 15, 2008 through November 21, 2008

## Disclaimer

# Lessons Learned at 208K: Towards Debugging Millions of Cores

Gregory L. Lee*, Dong H. Ahn*, Dorian C. Arnold†, Bronis R. de Supinski*,
Matthew Legendre†, Barton P. Miller†, Martin Schulz*, Ben Liblit†

*Lawrence Livermore National Laboratory
Computation Directorate
Livermore, CA 94550
{lee218, ahn1, bronis, schulzm}@llnl.gov

†University of Wisconsin
Computer Sciences Department
Madison, WI 53706
{darnold, legendre, bart, liblit}@cs.wisc.edu

***Abstract*—Petascale systems will present several new challenges to performance and correctness tools. Such machines may contain millions of cores, requiring that tools use scalable data structures and analysis algorithms to collect and to process application data. In addition, at such scales, each tool itself will become a large parallel application – already, debugging the full BlueGene/L (BG/L) installation at the Lawrence Livermore National Laboratory requires employing 1664 tool daemons. To reach such sizes and beyond, tools must use a scalable communication infrastructure and manage their own tool processes efficiently. Some system resources, such as the file system, may also become tool bottlenecks.**

**In this paper, we present challenges to petascale tool development, using the Stack Trace Analysis Tool (STAT) as a case study. STAT is a lightweight tool that gathers and merges stack traces from a parallel application to identify process equivalence classes. We use results gathered at thousands of tasks on an Infiniband cluster and results up to 208K processes on BG/L to identify current scalability issues as well as challenges that will be faced at the petascale. We then present implemented solutions to these challenges and show the resulting performance improvements. We also discuss future plans to meet the debugging demands of petascale machines.**

## I. Introduction

Large scale system sizes are increasing exponentially [1] and systems with millions of cores are on the horizon. We are faced with the challenge of ensuring system software and tools scale to the level required by these environments. In particular, we require scalable debugging and performance tools as users will need mechanisms to ensure the correctness of their codes and the efficient utilization of the underlying, often unique and complex, target architectures.

Recently, several projects have begun to address this problem by integrating hierarchical communication structures with online aggregation mechanisms, like MRNet [2] or Supermon [3], into their tools. On the debugger side, HP's Ladebug [4] relies on a tree of debug daemons to control large numbers of tasks and STAT (our own Stack Trace Analysis Tool) [5] uses MRNet to collect and merge stack traces from an entire parallel application. In the area of performance tools both TAU/Paraprof [6] and Open|SpeedShop [7] have demonstrated prototypes with MRNet, and TAU has also built on Supermon [8].

Using a hierarchical approach is a necessary aspect of a scalable tool design and has proven sufficient at modest system sizes of a few thousand processors. However, when applied to today's largest systems of 50,000 processors and more, we find that new bottlenecks arise and scalable tool designs require further refinements in order to achieve the necessary scalability. We anticipate that these bottlenecks will prove too great for even the most scalable of current tool designs to support petascale systems, which are projected to have more than one million cores. Instead, we must enhance those designs with new techniques gleaned from lessons learned on today's largest systems.

In this paper we categorize and describe the key scalability challenges that tools face on current and future architectures. More importantly, we provide solutions to overcome them. In particular, we identify scalability problems in tool framework startup costs, data structure representation, and tool related file I/O, such as that caused by symbol table parsing. We further highlight issues that arise from the projected move towards highly multithreaded codes. In particular, the contributions of this paper include:

- a detailed analysis of bottlenecks that prevent scaling despite the use of a highly scalable tool design;
- a classification of such bottlenecks with respect to their root cause;
- a description of the three most severe issues that prevent full scalability in today's systems along with methods to overcome them; and
- experimental results on up to 212,992 cores to validate our findings.

We perform this work using our own Stack Trace Analysis Tool (STAT) as a case study and show how we optimized its performance and scalability on two large scale systems: *Atlas*,

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-402967), and by DOE Grants DE-FG02-93ER25176 and DE-FC02-07ER25800, an Intel Graduate Fellowship, AFOSR Grant FA9550-07-1-0210, and NSF Grants CCF-0621487, CCF-0701957, and CNS-0720565.

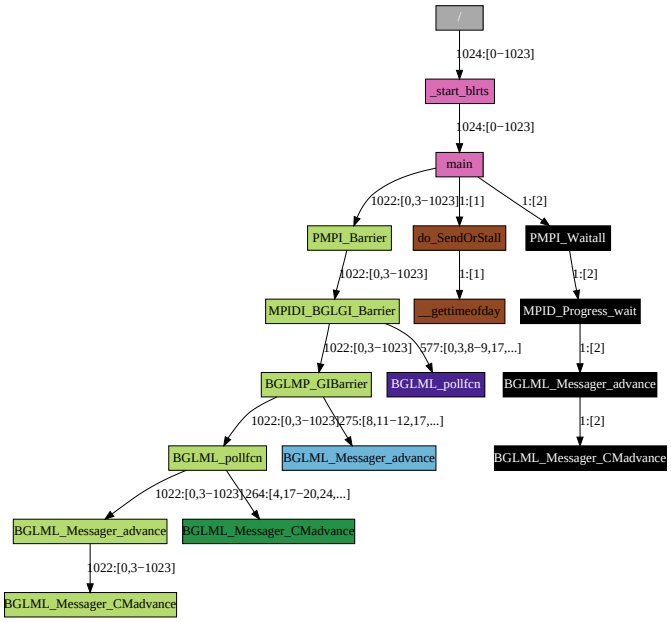



Fig. 1. Example 3D-Trace/Space/Time call graph prefix tree from STAT

an Infiniband-based Linux cluster with close to 10,000 cores, and *BlueGene/L* (*BG/L*), a custom architecture with 212,992 cores distributed in 104 racks. Importantly, the issues that we encounter are general; they are not specific to STAT or these platforms, but rather will be encountered by a wide range of tools on emerging petascale architectures.

In Section II we introduce our case study tool, STAT, and in Section III we present details of our evaluation environment. We then illustrate the three main problems identified by our study along with proposed solutions: the reduction of tool startup cost (Section IV), the need for hierarchical data structures in combination with hierarchically-structured reduction networks (Section V), and the efficient utilization of I/O resources for tool-internal analysis tasks (Section VI). In Section VII we consider upcoming issues of scalably supporting multithreaded codes before we conclude the paper with a brief review of related work in Section VIII and final remarks in Section IX.

## II. The Stack Trace Analysis Tool

The Stack Trace Analysis Tool [5] (STAT) is part of a petascale debugging strategy that uses lightweight tools on an entire parallel application to reduce the problem search space to a manageable subset of tasks, on which a heavyweight debugger can perform root cause analysis. STAT identifies process equivalence classes (groups of processes that exhibit similar behavior) by sampling stack traces over time from each task of the parallel application and merging them into a call graph prefix tree. The call graph prefix tree intuitively represents the application's behavior classes over space and time, as shown in Figure 1. These equivalence classes reduce both the search space and the number of tasks that need to be considered, so that the user can effectively apply a full-featured debugger to problems that arise at scale.

STAT was designed with scalability as the primary goal, so it uses lightweight mechanisms for gathering stack traces and the MRNet [2] tree based overlay network (TBŌN) to reduce the data and processing loads on its front end. Conceptually, STAT has three main components: the front end, the tool daemons, and the stack trace analysis routine. The front end controls the collection of stack trace samples by the tool daemons, and our stack trace analysis routine processes the collected traces. The TBŌN provides scalable communication between the STAT front end and back ends, while a custom STAT filter efficiently merges the stack traces as they propagate up the communication tree.

The development and deployment of STAT has been very successful. Our previously-published results demonstrate that the basic architecture and intelligent implementation of the filter routines support scalability to four thousand tasks [5]. Subsequent experiments have shown that it is the first parallel debugging tool to scale to tens of thousands processors while still maintaining reasonable bounds on execution time that allow for interactive usage [9]. Most other tools fail to work at such scales. Some fail due to internal or OS restrictions, and for others the execution time of even simple, individual operations grows linearly with the scale of the target application, which leads to impractical delays at the scales required for petascale architectures.

Even with an explicitly scalable design such as STAT's, our analysis shows that severe bottlenecks and challenges still remain for petascale tools. In particular, new bottlenecks arise or become more significant at 100,000 or 1,000,000 cores compared to 1,000 or 10,000. While our experiences with STAT may not exhaustively identify all petascale tool performance challenges, most tools will benefit from addressing those we have encountered.

We classify the performance challenges for petascale tools into two main groups: structural problems within the tool and inefficient interactions with the environment. Examples of tool design issues include inefficient analysis algorithms, excessive communication bandwidth requirements, and data structures that use more space than necessary when considered in the aggregate. One can directly address these problems through tool (re)design, e.g., by using different analysis algorithms or data representations. Although STAT's original design handled many of these issues, our experience, consistent with that of scaling any parallel application, demonstrates that what previously worked well at large scales does not necessarily extend an additional order or two of magnitude. For example, we show in Section V that, while space-efficient data structures are sufficient for terascale tools, petascale tools must use hierarchically-distributed data structures.

Inefficient environment interactions include actions such as tool daemon launching or access to files. These issues are more difficult to address because their sources are not entirely in the tool developer's hands. Although we can evaluate the tool's interactions with the system and implement specific solutions to minimize their impact, the best solutions may require changes

to the hardware or the system software.

In contrast to the structural problems, inefficient interactions with the environment typically are not restricted to a particular tool, but rather describe general problems that most scalable tools will face. We therefore focus on encapsulating these problems into separate reusable components with portable APIs. This decouples the tool itself from the system software, reduces its complexity, and allows easy porting of machine-dependent components, while at the same maintaining high efficiency. Further, it allows the tool to exploit future improvements in the hardware or system software seamlessly.

In the following sections we describe our experience with our initial version of STAT on two large scale platforms and analyze the performance and scalability bottlenecks we observed. We focus on a new lesson for scalable tool design, using hierarchically distributed data structures, and two issues arising from environmental interactions. For the first of the environmental issues, we implement a new tool interface to scalable resource management. We also explore some initial solutions to the second environmental challenge, which involves file system and other I/O interactions.

## III. Scalability Evaluation Environments

This section details the systems we use to explore scalability challenges of tools. We measure STAT's performance and scalability on two machines with distinctly different architectures using a simple application with a known bug. In order to explore the root cause for the observed scalability properties we measure STAT's three main phases separately: the launch time of the daemons; the daemons' local gathering and aggregation of stack traces; and the aggregation of locally-merged results to the final call graph prefix tree at the front end.

The first machine, Atlas, is a 1,152-node Linux cluster with four-way, dual-core 2.4 GHz AMD Opteron nodes connected with DDR Infiniband. On Atlas, one STAT daemon is launched per compute node and gathers stack traces from all eight MPI tasks running on that node. Any MRNet communication process employed is launched on a separate allocation of compute nodes, one per compute core. This platform allows us to explore the suitability of STAT's base design for terascale systems.

Our second machine is the BG/L installation at the Lawrence Livermore National Laboratory (LLNL). This system has 106,496 compute nodes, which are primarily dual 700 MHz PowerPC 440 chips. The BG/L architecture dictates that tools such as STAT launch their daemons onto dedicated I/O nodes. The LLNL configuration has one I/O node for every 64 compute nodes resulting in 1664 I/O nodes for the whole machine. BG/L supports two modes of operation: co-processor mode and virtual node mode. In co-processor mode, one of the cores runs an MPI task while the other core is used to offload communication. In virtual node mode, each core runs a distinct MPI task. As a direct consequence, each STAT daemon gathers stack traces from 64 processes in co-processor mode and 128 processes in virtual node mode, resulting in a significantly higher load for the STAT daemons as compared to the 8-way CMP configuration found on Atlas.

On BG/L, MRNet communication processes are spawned on one of 14 login nodes, each with two 1.6 GHz Power5 processors, which restricts the topologies that we can use. Nonetheless, we test multiple MRNet tree topologies including a flat 1-to-N topology (1-deep), a tree with a single layer of communication processes between the front end and the daemons (2-deep) and a tree with two layers of communication processes (3-deep). For the 2-deep tree, we use a fanout from the front end equal to the square root of the number of daemons or 28, whichever is less. At all scales, the 3-deep tree has a fanout from the front end equal to 4. The next level employs either 16 or 24 communication processes, depending on the job scale.

Our target application is a simple MPI ring topology test with an injected bug that causes the application to hang. Each task does an MPI_Irecv from the previous task in the ring and an MPI_Isend to the next task, followed by an MPI_Waitall and an MPI_Barrier. The injected bug causes MPI task 1 to hang before its send.

This paper presents the first performance results of tool experiments with over 100,000 application tasks. Thus, our BG/L results provide insight into the new challenges faced by petascale tools. Subsequent sections detail these insights, although one set is more indicative of the nature of running on the world's largest system. Since we share the BG/L installation with application teams, we were only able to perform a limited number of runs with all 104K compute nodes.

## IV. Scalable Resource Management

This section explores our first scalability issue arising from interactions with the system environment: tool initialization. Interactive tools must run some components, such as tool daemons, concurrently with the user application. Starting these tool daemons is an one-time cost that is easy to overlook in the design of a scalable tool, and most tools use some simple *ad hoc* mechanism based on *rsh* or *ssh*. While it may seem that any reasonable implementation is sufficient for one-time operations, application experiences at very large scales demonstrate that efficient launching strategies are essential. This is especially true for interactive tools: thirty minutes to initialize a debugger is simply unacceptable to all but the most desperate of users.

### A. Startup Costs with Initial Implementation

The initial STAT implementation relies on the daemon-spawning facilities within MRNet, which uses remote access protocols such as *ssh* or *rsh* to individually launch the daemons. On Atlas and other Linux platforms, MRNet performs all tool startup, including that of the back-end daemons. On BG/L, users cannot log in to the I/O nodes. Therefore, BG/L's own system software launches the STAT daemons, but MRNet's launching facility still launches the communication processes distributed across the BG/L login nodes. Having to use different *ad hoc* mechanisms complicates tool software maintenance. More importantly, we find that the MRNet facility does not scale well and would incur unacceptable costs for petascale systems.

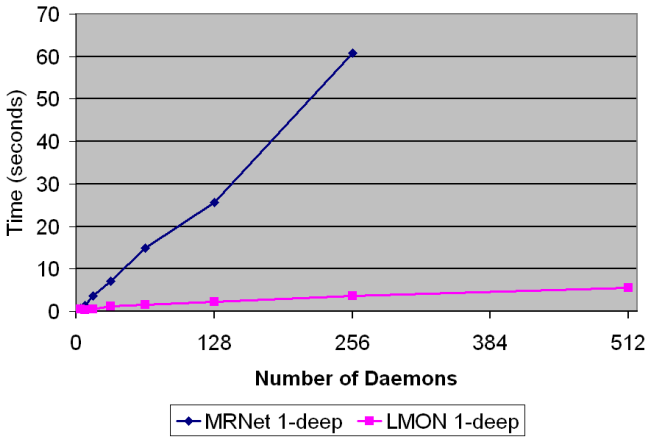



Fig. 2. STAT startup time, LaunchMON versus MRNet

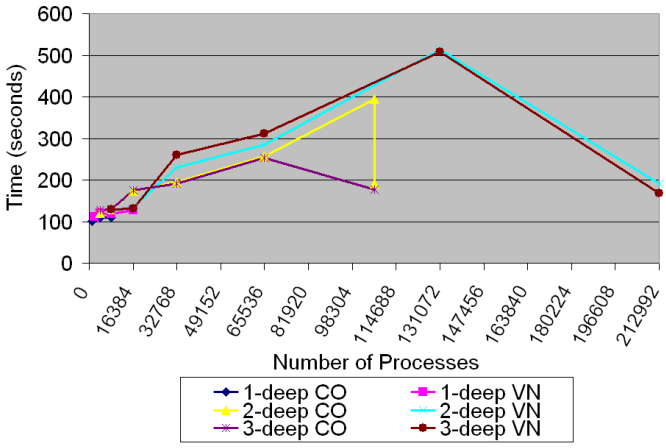



Fig. 3. STAT startup time on BG/L with various topologies

We have measured the startup time for STAT on Atlas. This includes the time required to launch all STAT daemons and MRNet communication processes and to connect all of the tool processes to the MRNet network. The MRNet line in Figure 2 shows results with a flat 1-to-N topology. Our results have a clear linear scaling trend; this is consistent with the fact that each daemon is launched sequentially. At 512 nodes, MRNet consistently fails to launch the daemons when using *rsh*. Our previous results [5] on the Thunder machine scaled beyond this point by using *ssh*. Unfortunately, Atlas does not support *ssh* on the compute nodes.

We have also measured the startup time for STAT on BG/L. As with Atlas, this includes the time to launch all STAT daemons and MRNet communication processes and to connect them all to the MRNet network. In addition, the BG/L STAT prototype only supports debugging when the application is launched under the tool's control. Hence, the startup time includes the time to launch the application. Results are shown in Figure 3. The startup time on BG/L exceeds 100 seconds even at 1024 compute nodes and exhibits linear scaling for larger runs. The majority of this time occurs during the launching of the back-end daemons and the generation of the process table by BG/L's system software. At 64K compute nodes in virtual node mode, the system software accounts for over 86% of the startup time. In addition to a scalability performance issue, the BG/L resource manager also suffered from a scalability correctness issue and caused an apparent run time failure (hang) at 208K processes. Subsequent patches by IBM were able to help alleviate both issues, leading to successful runs at 208K processes. These changes included increasing buffer sizes and removing the usage of non-scalable routines such as *strcat*, which scans the buffer for the string termination character, for data packing. The drops in startup time at the end of the curves in Figure 3 show the performance improvement, with more than a two fold speedup at 104K processes in the 2-deep CO case.

### B. Systematic, Reusable Tool and Job Startup

While *ad hoc* mechanisms like *ssh* or *rsh* to launch tool daemons and TBŌN processes work correctly on some systems, they do not provide portability. Several systems like BG/L or the Cray XT3 do not support these protocols. Rather, daemon launching must use a service that their resource management systems provide. More importantly, these *ad hoc* mechanisms rarely provide optimal performance and scalability. At extreme scales, a tool becomes a large parallel application in its own right, with thousands of debugger daemons. Even today, debugging the full BG/L installation at LLNL requires launching 1664 daemons onto the I/O nodes. Thus, petascale tools need better integration with resource management systems.

To address this challenge, we integrate STAT with LaunchMON, a general-purpose infrastructure for launching tool daemons [10]. LaunchMON implements a portable daemon-spawning mechanism that exploits scalable system services provided by the resource management software. Its use of efficient daemon-launching mechanisms improves the daemon launch and TBŌN setup time to have overheads an order of magnitude less than *ad hoc* methods. Also, decoupling the daemon-launching mechanism from the tool's core operations allows its front end to avoid excessive requests for system services such as remote shell processes.

### C. Optimized Tool Launching Performance

We compare the startup performance of STAT integrated with LaunchMON to the startup performance of STAT using MRNet's daemon launching capabilities. Figure 2 shows that STAT's startup using LaunchMON clearly scales better than serially launching the daemons using remote access protocols. In fact, STAT starts 512 daemons in 5.6 seconds, a scale at which the MRNet daemon spawning facility failed but would have taken over 2 minutes based on the clear linear scaling trend. Most of the scalability advantage comes from

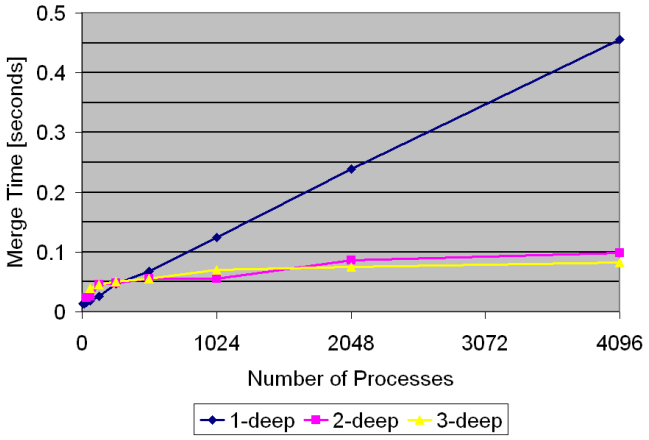



Fig. 4. STAT merge time on Atlas with various topologies

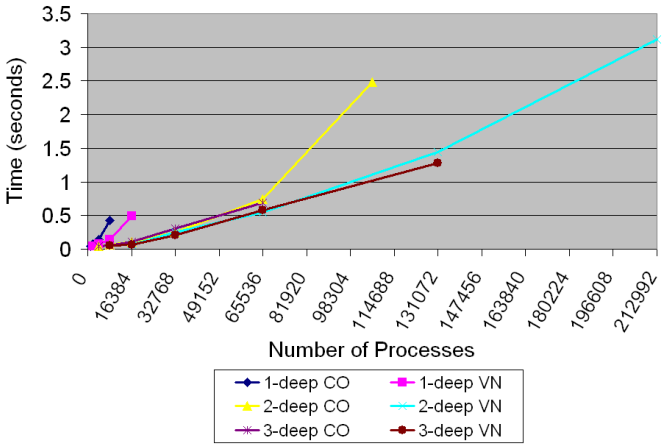



Fig. 5. STAT merge time on BG/L with various topologies

LaunchMON's ability to utilize the resource manager to bulk-launch the daemons.

## V. Scalable Data Structure Design

Most scalable tool designs deploy some kind of hierarchical, online data aggregation scheme to reduce the amount of data to be processed at the front-end node. However, as we show in this section, this aggregation alone is insufficient if not accompanied by a matching hierarchical data structure representation. Otherwise, the pressure on the network due to sending unnecessarily-large volumes of data causes significant network congestion. This problem falls into the category of structural problems and requires a careful (re)design of the tool's internal data structures.

### A. STAT Merge Times

We first encounter this bottleneck when investigating the performance of the STAT stack trace merging routine. In particular, we measure the time it takes for each STAT daemon to send its locally-merged 2D trace-space and 3D trace-space-time prefix trees through the MRNet tree until the STAT front end has a globally-merged copy of both prefix trees. For this test, we run STAT with 1-deep, 2-deep, and 3-deep topologies. In the 2-deep and 3-deep cases, we employ a balanced topology; that is, every parent process in the MRNet tree has approximately the same number of children. More specifically, for an $n$-deep tree, the maximum fanout is set to the $n^{\text{th}}$ root of the number of daemons. The scaling results in Figure 4 show that even the 1-deep tree is able to perform the merging quickly, under half a second at 4,096 tasks, although it shows a clear linear scaling trend. The 2-deep and 3-deep results, on the other hand, show significantly better scaling characteristics.

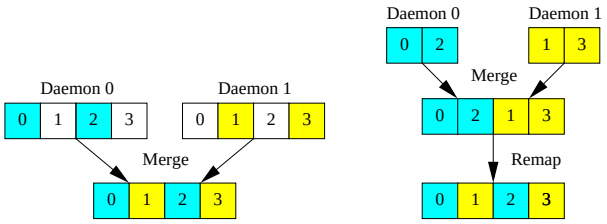
While the Atlas results show good scalability and no indication that bottlenecks exist at its modestly-large scale, our experiments on BG/L paint a different picture. The respective trace merge times can be seen in Figure 5. The 1-deep tree fails to merge the graphs at 16,384 compute nodes (256 I/O nodes), but the results up to that size clearly show the poor scaling characteristics of this flat topology. The 2-deep and 3-deep trees tested have similar performance to each other and both show linear scaling at larger scales, rather than the logarithmic scaling expected with the MRNet tree.

One of the main reasons for this unexpected behavior is the internal representation that STAT uses to identify the set of tasks that take a given call path. At all levels in the MRNet tree, we use a common representation for the stack traces that have been collected and aggregated. As a consequence, all bit vectors used to indicate task sets (e.g., for for the call graph prefix tree edge labels) are sized to fit all tasks of the entire application, even though only a subset of the bits are relevant for a given daemon or MRNet communication process. Thus, the tool unnecessarily tracks and sends many zero bits and generates a high load on the I/O nodes.

Unfortunately, limited machine access prevented us from getting complete scaling runs with the 3-deep topology, but we anticipate the results would follow the 2-deep case. In any case, the linear scaling demonstrates that the STAT implementation needs to be modified for these scales and beyond. Looking forward to petascale machines, a million cores would require a 1 megabit bit vector per edge label. This would easily saturate the network with a large daemon count as well as lead to severe memory contention on the processing nodes.

### B. Hierarchical Data Representations

To alleviate the problem caused by fixed-size bit vectors, we implement a hierarchical task list scheme that matches the hierarchical structure of the tool communication layer. With this approach each analysis node (STAT daemon or communication process) only maintains lists for tasks within its own subtree. In particular, each STAT daemon, which is a leaf in the analysis tree, only maintains lists containing all tasks from which it gathers stack traces. During the merge process in the analysis tree we then combine the task lists of all children by simple




(a) original bit vector (b) optimized bit vector

Fig. 6. Daemon 0 is debugging tasks 0 and 2, while Daemon 1 is debugging tasks 1 and 3. The original STAT bit vector (a) maintains excess bits (the white boxes). The optimized STAT bit vector (b) conserves bits, but requires the bit vector to be remapped in MPI rank order.

concatenation. However, since the mapping of nodes to STAT daemons is not guaranteed to be in MPI rank order, we need to add a remapping step at the front-end tool. For this we first collect the map information once during the setup phase and then perform a local remap during the final result rendering. Figure 6 illustrates the difference between the original and optimized implementations. Under this scheme, only the front end maintains bit vectors that spans the entire job and we never send a full bit vector over the TBŌN.

While this specific approach is limited to STAT's node lists, similar techniques can also be applied in other scenarios. As a general rule tools must avoid global views of all tasks during the processing of performance or debug information. Instead, the working set for each participant in each analysis routine should be as minimal as possible. With tree-based schemes, in particular, a reduction of the working set to application processes covered by the current subtree ensures that data structures only grow logarithmically together with the actual computation.

### C. Merge Times after Bit Vector Optimization

Figure 7 shows the merge time results from BG/L with the bit vector optimization in comparison to the original bit vector implementation. The optimized bit vector exhibits logarithmic scaling, in contrast to the original linear scaling, because of the reduction in the data volume being sent through the MRNet network. We achieve this logarithmic scaling despite limitations on the number of communication processes we could launch on BG/L and that we could not employ a fully balanced tree topology. For tools that use a TBŌN, these results clearly show the importance of designing data structures that are tailored to exploit the logarithmic properties of the network tree effectively. We note that the virtual node mode cases run faster than the co-processor mode cases at equivalent task counts because the merge performance is bound not only by the task count, but also by the number of daemons. Since we are restricted on the number of communication processes we can launch on BG/L, the fanout from the last level of communication processes in co-processor mode is double that of virtual node mode at equivalent task counts.

The optimized bit vector does incur an additional cost in the remapping step, which rearranges the bit vectors into MPI rank order. At 208K tasks, this step takes 0.66 seconds.

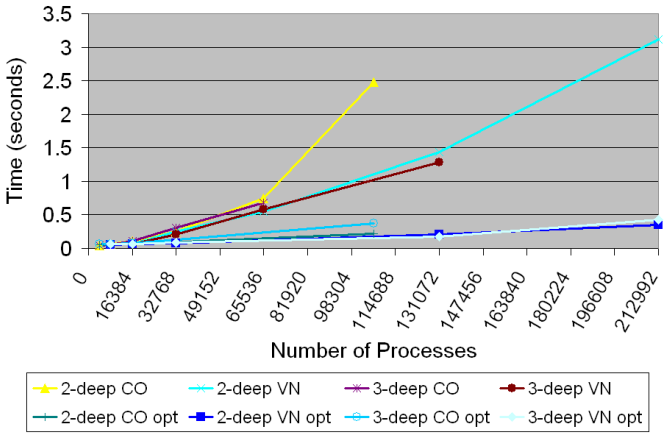



Fig. 7. Optimized bit vector STAT merge time versus original bit vector STAT merge time

## VI. File System Issues and Access to Static Information

As we scale up to larger systems, certain operations of STAT that we assumed local to each node begin manifesting scalability issues. Despite our expectation of constant overheads, the "independent" operations that each daemon performs exhibit worse than linear scaling characteristics. Most notable is the stack trace sampling. Clearly, there is a scalability bottleneck that prevents this ostensibly-independent operation from scaling optimally. Further analysis attributes it to the tool's inefficient interactions with the I/O subsystems and with the MPI implementations.

This section explores the performance problem of STAT's stack trace sampling, and presents our response: the *scalable binary relocation service* (SBRS). While we present this using STAT results, it captures general problems for contemporary scalable tool research, in particular with trends towards scalably combining reusable, serial components using a communication infrastructure [4], [7], [11].

### A. Sub-optimal Scaling Characteristics of Stack Trace Sampling

STAT uses the StackWalker API [12], a lightweight API that lets each back-end daemon take stack traces of the co-located processes on a node, or other associated processes traceable from that node. This API is intended to minimize the perturbation to the application processes and offer constant sampling overheads regardless of scale. In defiance of these expectations, our experiments reveal sub-optimal scalability in stack trace sampling, even with such a lightweight API.

On both Atlas and BG/L, we measure the local time for each daemon to gather ten stack traces from all local application processes. During this time, each daemon performs a local merge of the stack traces that it gathers, including a 2D trace-space prefix tree and a 3D trace-space-time prefix

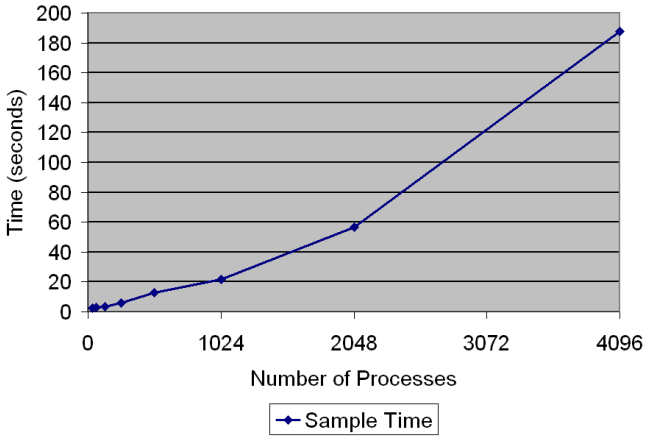



Fig. 8. STAT sampling time on Atlas with a flat 1-to-N topology

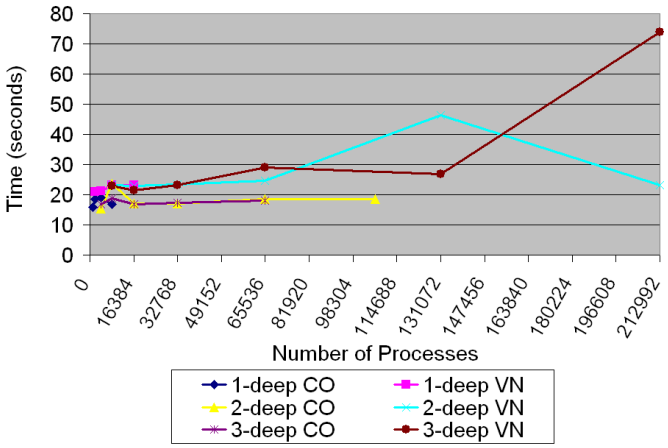



Fig. 9. STAT sampling time on BG/L with various topologies

tree. Following the common practice of our users, we stage the application executable on the network file system (NFS) mounted home directory. Figure 8 shows the Atlas results up to 4,096 MPI tasks (512 daemons) while Figure 9 shows the BG/L results up to 212,992 tasks (1,664 daemons). These scaling runs expose several scaling characteristics of the stack trace sampling performance. First, the sampling operation scales poorly on Atlas, with slightly worse than linear scaling, while it generally scales better on BG/L. Second, this operation occasionally suffers performance variations larger than 20%; specifically, the essentially-identical operation of two virtual node mode runs (2-deep VN and 3-deep VN) makes greater than a factor of two performance difference at 212,992 MPI tasks. Third, at smaller scales the stack trace sampling performs better on Atlas than on BG/L.

Among these observations, only the third has a simple explanation. We can easily attribute the third observation to an architectural difference between two systems: on BG/L, a daemon must deal with 64 processes in co-processor mode or 128 in virtual node mode, while on Atlas only 8 processes.

For the first and the second observations, we theorize that the problems may lie in the file I/O characteristics of the tool daemons on shared file systems (NFS in this case). They are the only non-local resource that the StackWalker API uses (to parse the symbol table of the binary files residing in them). Since no access coordination is provided, all participating daemons simultaneously access the binaries, thrashing the file server and/or becoming increasingly vulnerable to the current file server loads. We suspect that this problem is generally less severe on BG/L, as each daemon only has to deal with a single static executable binary, in contrast to an executable with multiple shared libraries, as is the case on Atlas.

Additionally, on BG/L a daemon does not contend for CPU time with the processes it traces, as it runs on a dedicated I/O node. In contrast, on Atlas the default behavior of an MPI task waiting for a message arrival is to spin-wait on a CPU core. When a node is fully loaded, this behavior causes CPU contention with the daemon. At large scale, there is a higher probability that a daemon encounters the processes that either spin or enter a blocking kernel service or a critical section, refusing to yield the core.

### B. I/O Subsystems and MPI Implementations

To further test our theory and to develop a general solution, we implement a scalable binary relocation service (SBRS) prototype. It is designed to scalably relocate a requested executable and its dependent shared libraries from a shared file system such as NFS to the RAM disk of participating nodes. It then automatically redirects each tool daemon's file I/O requests on the original files to the relocated versions by interposing all of its *open* calls. SBRS refers to the mounted file system table (mtab) to determine if a binary resides on a globally-shared file system. If it does, SBRS broadcasts the binary via a communication fabric that the client tool must provide. For STAT, we integrated SBRS with the LaunchMON framework such that the master back-end daemon fetches a target binary from the file system and distributes it to the rest of the daemons using LaunchMON's back-end communication API (through the Infiniband switch in the case of Atlas).

Figure 10 shows preliminary results on Atlas up to 1,024 MPI tasks, 128 back-end daemons. It illustrates that the sampling costs on the relocated binaries (red line) are now a constant of about 2 seconds regardless of scale. The SBRS overhead itself is also measured to be small, taking 0.088 seconds to relocate two main binary files, the base executable (10KB) and the MPI library (4MB), to 128 nodes. To obtain such performance, we find that we must minimize contention between SBRS and application tasks. Thus, SBRS currently sends `SIGSTOP` to all application processes and gives a grace period for them to settle before it begins the relocation. We are currently studying the MPI implementation to devise strategies that would enable better time sharing of resources between MPI tasks and the SBRS.

In this experiment, we also measure the sampling perfor-

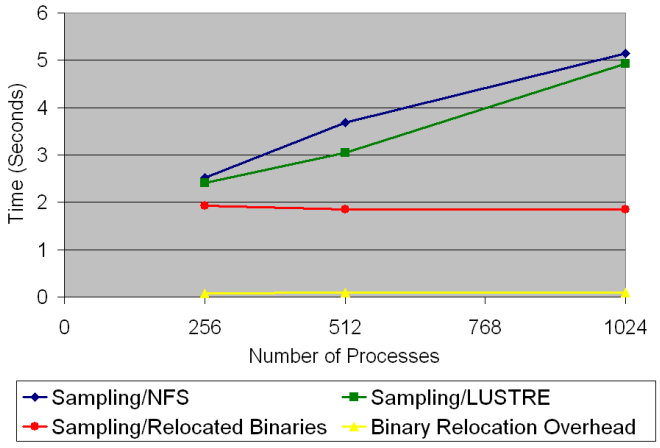



Fig. 10. STAT sampling time on Atlas with the binary relocation service prototype

mance when the binaries reside on a parallel file system, LUSTRE [13]. However, at this scale, LUSTRE offers little improvement over NFS. We note that the overall sampling performance on NFS of Figure 10 is about four times better than the original measurements shown in Figure 8. This is mainly due to a recent OS update which shifts several dependent shared libraries to faster file systems.

## VII. The Challenge Ahead: Threading

While applications on current large scale architectures mostly rely exclusively on MPI, it is expected that this will change in the near future. The increasing number of cores will put a high burden on the connection management of any MPI implementation and the expected rise of multi-core and many-core node architectures will increase the pressure on the memory per core ratio that applications can expect to work with. As a consequence of these trends, programmers will have to use threading within tasks to keep the number of MPI endpoints as well as the memory available per task reasonable.

Dealing with multithreaded applications can present additional challenges to tools. Beyond the mechanical complexities of dealing with data collection from a multithreaded application, threads can serve as a potentially-unbounded multiplier on the amount of data being collected and managed. For a tool such as STAT, an application running on 10,000 nodes with 8 threads per node presents many of the same challenges as an application running on 80,000 nodes. Many systems do not enforce a reasonable limit on the number of threads an application may spawn, so an application could spawn an arbitrary number of threads and generate an arbitrary amount of data for the tool. Of course, it is likely that such an application will need the help provided by performance analysis and debugging tools.

We plan to implement thread support in the next version of STAT. Instead of collecting a call stack from each process in an application, we plan to collect a call stack from each thread in the application. STAT will continue to associate each call stack with its process representation, rather than associate it with a new thread representation.

Collecting call stacks from multithreaded processes will generate extra work during the stack trace sampling phase (when STAT collects call stacks from each node) and during the merging phase (when STAT merges call stacks from across the application into a single representation). We expect to see only a constant slowdown per thread in stack trace sampling time, as this operation happens in parallel across all nodes. We also expect that the MRNet scalable features will only cause a logarithmic slowdown in merging time.

We do not expect to have to make significant changes to STAT's visualization mechanism to support threads. The general goal of STAT will still be to highlight the processes that may be causing bugs so that a heavyweight debugger can be judiciously deployed. Our initial thoughts are that identifying bugs in specific threads will both add complexity to the STAT implementation and visualization, and it does not present much information beyond what could be obtained from a heavyweight debugger attached to a specific process.

## VIII. Related Work

Many performance and correctness tools have been developed for parallel computing platforms, with varying levels of scalability. Full-featured debuggers such as TotalView [14] and DDT [11] have been run on thousands of processes, but typically suffer high latencies for even simple operations at these scales. The Ladebug project [4] aims at providing scalable parallel debugging and presented a prototype debugger on top of a hierarchical communication network similar to MRNet used in STAT. However, Ladebug has not been evaluated at the scales targeted in this work.

Several projects have targeted scalable MPI tracing and profiling. For instance, the mpiP [15] lightweight profiling library has been successfully run with thousands of processes. The SCALASCA project [16] performs scalable trace analysis and has been successfully run up to 16,384 processes. IBM's High Performance Computing Toolkit [17] also provides a set of tools for monitoring job performance. Tools such as CUBE [18] and VAMPIR [19] have been developed to aid in the visualization of large, parallel traces.

Several papers have discussed projects targeting BlueGene/L. Chung et al. [20] evaluated several of the aforementioned MPI performance analysis tools on BlueGene/L and identified the quantity of data collected at large scales as a potential bottleneck. Schulz et al. [21] described an implementation of the Dynamic Probe Class Library [22] targeting BlueGene/L. We previously developed STATBench [9], a tool emulation infrastructure, and evaluated STAT's scalability for BlueGene/L up to 128K processes.

## IX. Conclusion

Supercomputing experts know that what works for ten processes may not work for a hundred. Likewise, scalability to a thousand processes does not ensure good behavior as we approach the millions of cores that petascale computing

promises. Furthermore, at extreme scales, even performance and correctness tools become massively-parallel applications, with all the challenges that computing at scale brings. We have presented a detailed case study of the challenges faced in scaling the Stack Trace Analysis Tool (STAT) up to hundreds of thousands of compute nodes. Ours is the first tool-performance evaluation with over one hundred thousand application tasks, and therefore provides new insight into three key challenges faced by petascale tools.

First, we find that sequential daemon launching becomes a bottleneck at this scale. We improve both scalability and portability by eschewing *ad hoc* sequential launchers in favor of LaunchMON, a portable daemon spawner that integrates closely with native resource managers. Second, as daemons run, we find that it is critical that they avoid data structures that represent, or even reserve space to represent, a global view. Instead, we adopt a hierarchical representation that dramatically reduces data storage and transfer requirements at the fringes of the analysis tree. Third, we find that seemingly-independent operations across daemons can suffer scalability bottlenecks when accessing a shared resource, such as the file system. Our scalable binary relocation service is able to optimize the file operations and reduce file system accesses to constant time regardless of system size.

Profiling and debugging million-core applications will bring a host of new challenges, including some yet to be discovered. However, we can anticipate some scalability issues even now. The lessons we have learned with STAT will apply to future tools as well. To the extent that ours is the first reported experience with tools at this scale, we hope that our findings will light the petascale path for other tool developers to follow.

## REFERENCES


[1] "Top 500 Supercomputer Sites," http://www.top500.org/.

[2] P. C. Roth, D. C. Arnold, and B. P. Miller, "MRNet: A Software-Based Multicast/Reduction Network for Scalable Tools," in *SC '03*, Phoenix, AZ, 2003.

[3] M. Sottile and R. Minich, "Supermon: A High-Speed Cluster Monitoring System," in *Proceedings of the IEEE International Conference on Cluster Computing (Cluster)*, 2002.

[4] S. M. Balle, B. R. Brett, C. Chen, and D. LaFrance-Linden, "Extending a Traditional Debugger to Debug Massively Parallel Applications," *Journal of Parallel and Distributed Computing*, vol. 64, no. 5, pp. 617–628, 2004.

[5] D. C. Arnold, D. H. Ahn, B. R. de Supinski, G. L. Lee, B. P. Miller, and M. Schulz, "Stack Trace Analysis for Large Scale Debugging," in *The International Parallel and Distributed Processing Symposium*, Long Beach, CA, 2007.

[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] M. Schulz, J. Galarowicz, D. Maghrak, W. Hachfeld, D. Monotya, and S. Cranford, "Open|SpeedShop: An Open Source Infrastructure for Parallel Performance Analysis," *to appear in Special Issue of Scientific Programming on Large-Scale Programming Tools and Environments*, 2008.

[8] A. Nataraj, M. Sottile, A. Morrisd, A. Malony, and S. Shende, "TAUover-Supermon : Low-Overhead Online Parallel Performance Monitoring," in *Proceedings of the International Conference on Parallel and Distributed Computing (Euro-Par 2007)*, Aug. 2007, pp. 85–96.

[9] G. L. Lee, D. H. Ahn, D. C. Arnold, B. R. de Supinski, B. P. Miller, and M. Schulz, "Benchmarking the Stack Trace Analysis Tool for BlueGene/L," in *Parallel Computing: Architectures, Algorithms and Applications (Proceedings of the International Conference ParCo 2007)*, Julich/Aachen, Germany, 2007.

[10] D. H. Ahn, D. C. Arnold, B. R. de Supinski, G. L. Lee, B. P. Miller, and M. Schulz, "Overcoming Scalability Challenges for Tool Daemon Launching," in *The International Conference on Parallel Processing*, Portland, OR, 2008.

[11] Allinea Software, "Allinea DDT the Distributed Debugging Tool," http://www.allinea.com/index.php?page=48.

[12] B. R. Buck and J. K. Hollingsworth, "An API for Runtime Code Patching," *The International Journal of High Performance Computing Applications*, vol. 14, no. 4, pp. 317–329, 2000.

[13] W. Yu, R. Nononha, S. Liang, and D. K. Panda, "Benefits of High Speed Interconnects to Cluster File Systems: A Case Study with Lustre," in *The International Parallel and Distributed Processing Symposium*, Rhodes Island, Greece, 2006.

[14] TotalView Technologies, "TotalView Debugger," http://www.totalviewtech.com/productsTV.htm.

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

[16] M. Geimer, F. Wolf, B. J. N. Wylie, and B. Mohr, "Scalable Parallel Trace-Based Performance Analysis," in *Proceedings of the 13th European Parallel Virtual Machine and Message Passing Interface Conference*, Germany, 2006.

[17] IBM, "High Performance Computing Toolkit," https://domino.research.ibm.com/comm/research_projects.nsf/pages/actc.index.html.

[18] M. Geimer, B. Kuhlmann, F. Pulatova, F. Wolf, and B. J. N. Wylie, "Scalable Collation and Presentation of Call-Path Profile Data with CUBE," in *Parallel Computing: Architectures, Algorithms and Applications (Proceedings of the International Conference ParCo 2007)*, Julich/Aachen, Germany, 2007.

[19] 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.

[20] I. H. Chung, R. E. Walkup, H. F. Wen, and H. Yu, "MPI Performance Analysis Tools on BlueGene/L," in *Proceedings of the 2006 ACM/IEEE Conference on SuperComputing*, Tampa, FL, 2006.

[21] M. Schulz, D. Ahn, A. Bernat, B. R. de Supinski, S. Y. Ko, G. Lee, and B. Rountree, "Scalable dynamic binary instrumentation for Blue Gene/L," *SIGARCH Comput. Archit. News*, vol. 33, no. 5, pp. 9–14, 2005.

[22] L. DeRose, J. T. Hoover, and J. K. Hollingsworth, "The Dynamic Probe Class Library: An Infrastructure for Developing Instrumentation for Performance Tools," in *IPDPS '01: Proceedings of the 15th International Parallel & Distributed Processing Symposium*. Washington, DC, USA: IEEE Computer Society, 2001, p. 66.