Benchmarking the Stack Trace Analysis Tool for BlueGene/L

Gregory L. Lee[1]**, Dong H. Ahn**[1]**, Dorian C. Arnold**[2]**,**
Bronis R. de Supinski[1]**, Barton P. Miller**[2]**, and Martin Schulz**[1]

[1] Computation Directorate,
Lawrence Livermore National Laboratory, Livermore, California, U.S.A.
E-mail: {*lee218, ahn1, bronis, schulzm*}*@llnl.gov*

[2] Computer Sciences Department,
University of Wisconsin, Madison, Wisconsin, U.S.A.
E-mail: {*darnold, bart*}*@cs.wisc.edu*

We present STATBench, an emulator of a scalable, lightweight, and effective tool to help debug extreme-scale parallel applications, the Stack Trace Analysis Tool (STAT). STAT periodically samples stack traces from application processes and organizes the samples into a call graph prefix tree that depicts process equivalence classes based on trace similarities. We have developed STATBench which only requires limited resources and yet allows us to evaluate the feasibility of and identify potential roadblocks to deploying STAT on entire large scale systems like the 131,072 processor BlueGene/L (BG/L) at Lawrence Livermore National Laboratory.

In this paper, we describe the implementation of STATBench and show how our design strategy is generally useful for emulating tool scaling behavior. We validate STATBench's emulation of STAT by comparing execution results from STATBench with previously collected data from STAT on the same platform. We then use STATBench to emulate STAT on configurations up to the full BG/L system size – at this scale, STATBench predicts latencies below three seconds.

1 Introduction

Development of applications and tools for large scale systems is often hindered by the availability of those systems. Typically, the systems are oversubscribed and it is difficult to perform the tests needed to understand and to improve performance at large scales. This problem is particularly acute for tool development: even if the tools can be run quickly, they do not directly produce the science for which the systems are purchased. Thus, we have a critical need for strategies to predict and to optimize large scale tools based on smaller scale tests. For this reason, we have developed STATBench, an innovative emulator of STAT, the Stack Trace Analysis Tool[4].

STAT is a simple tool to help debug large scale parallel programs. It gathers and merges multiple stack traces across space, one from each of a parallel application's processes, and across time through periodic samples from each process. The resulting output is useful in characterizing the application's global behavior over time. The key goal for STAT is to provide basic debugging information efficiently at the largest scale.

From its inception, STAT was designed with scalability in mind, and was targeted at machines such as Lawrence Livermore National Laboratory's (LLNL's) BlueGene/L (BG/L), which employs 131,072 processor cores. Existing tools have come far short of

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract DE-AC52-07NA27344 (UCRL-CONF-235241).

debugging at such scale. For example, our measurements show that TotalView[1], arguably the best parallel debugger, takes over two minutes to collect and to merge stack traces from just 4096 application processes. STAT, a lightweight tool that employs a tree-based overlay network (TBŌN), merges stack traces at similar scale on Thunder, an Intel Itanium 64 cluster at LLNL, in less than a second. However, gathering the data to demonstrate this scalability was an arduous task that required significant developer time to obtain the system at this large of a scale. Gaining time on the full BG/L system is even more difficult.

As an intermediate point in STAT's evolution and to determine the feasibility of running STAT at full scale on BG/L, we designed STATBench. STATBench allows specification of various parameters that model a machine's architecture and an application's profile. This emulation runs on the actual system but generates artificial application process stack traces, which allows the emulation of many more application processes than the actual processor count of the benchmark run. Our design is general: any TBŌN-based tool could employ a similar emulation strategy to understand scaling issues. Our STATBench results demonstrate that STAT will scale well to the full BG/L size of 131,072 processors.

In the remainder of this paper, we present STATBench and explain how it accurately models the scalability of STAT. In the following section, we first review the design of STAT and discuss why this type of tool is needed for large scale debugging. We then present the design of STATBench and discuss how it validates our large scale debugging strategy and can help to identify roadblocks before final implementation and deployment in Section 3. Section 4 then compares results from STATBench to observed performance of STAT at moderate scales and presents results from scaling studies using STATBench as well as improvements for STAT derived from those results. Finally, we discuss how to generalize this approach to support optimization of any TBŌN-based tool in Section 5.

2 The Stack Trace Analysis Tool

The Stack Trace Analysis Tool[4] (STAT) is part of an overall debugging strategy for extreme-scale applications. Those familiar with using large-scale systems commonly experience that some program errors only show up beyond certain scales and that errors may be non-deterministic and difficult to reproduce. Using STAT, we have shown that stack traces can provide useful insight and that STAT overcomes the performance and functionality deficiencies that prevent previous tools from running effectively at those scales. Namely, STAT provides a scalable, lightweight approach for collecting, analyzing, and rendering the spatial and temporal information that can reduce the problem search space to a manageable subset of processes.

STAT identifies process equivalence classes, groups of processes that exhibit similar behavior, by sampling stack traces over time in each task of the parallel application. STAT processes the samples, which profile the application's behavior, to form a call graph prefix tree that intuitively represents the application's behavior classes over space and time. An example call graph prefix tree can be seen in Figure 1. Users can then apply full-featured debuggers to representatives from these behavior classes, which capture a reduced exploration space, for root cause problem analysis.

STAT is comprised of 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

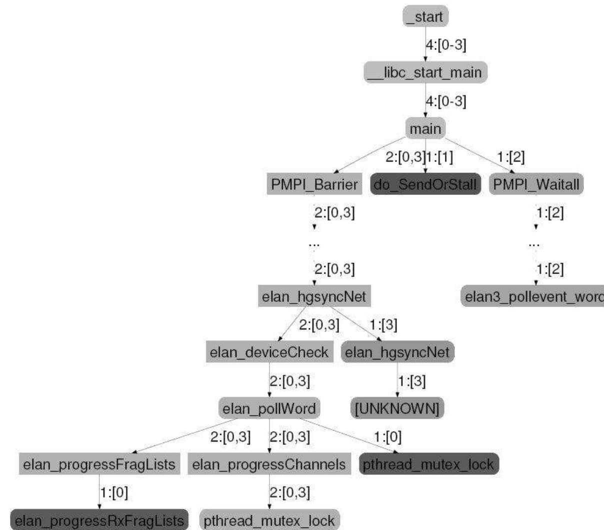


Figure 1. An example 3D-Trace/Space/Time call graph prefix tree from STAT.

front-end also color-codes and renders the resulting call graph prefix tree. The STAT front-end and back-ends use an MRNet[3] process tree for scalable communication and MRNet filters for efficiently executing the stack trace analysis routines, which parent nodes execute on input from children nodes. Back-ends merge local samples, which the TBŌN combines into a whole-program view. STAT uses the DynInst[2] library for lightweight, third-party stack tracing of unmodified applications.

3 Benchmark Implementation

STATBench inherits much of its implementation from STAT itself, using MRNet for scalable communication between the STATBench front-end and back-ends. STATBench also uses the same trace analysis algorithm as STAT to merge multiple traces into a single call prefix tree. The back-ends use this algorithm to perform local analysis and send the result through the MRNet tree to the front-end. The same algorithm is used in the MRNet filter function that is executed by the communication processes as the traces propagate through the communication tree.

STATBench does not actually gather traces from an application. Rather, the back-ends generate randomized traces controlled by several customizable parameters. These parameters include the maximum call breadth and the maximum call depth. The maximum breadth determines how many functions the generator can choose from, while the depth determines the number of function calls to generate for a given trace. STATBench prepends *__libc_start_main* and *main* function calls to the generated traces. STATBench also has a pair of parameters to control the spatial and temporal aspects of the trace generation. The spatial parameter governs how many processes each back-end emulates, equivalent to the number of processes a STAT daemon must debug. For the temporal aspect, STATBench also inputs the number of traces to generate per emulated process. With just these parameters, the generated traces can vary widely across tasks, which after spatial and temporal

merging yields a call prefix tree with a higher branching factor than is characteristic of real parallel applications. STATBench controls this variation through a parameter that specifies the number of equivalence classes to generate. Specifically, this parameter uses the task rank *modulo* the number of equivalence classes to seed the random number generator. Thus, all of the tasks within an equivalence class generate the exact same set of stack traces. These equivalence classes do not capture the hierarchical nature of the equivalence classes of real applications. However, our results, shown in Section 4, indicate that this simplification does not impact emulated performance significantly. With all of the possible parameters, STATBench can generate traces for a wide range of application profiles and can emulate various parallel computer architectures.

There is one main difference that allows STATBench to emulate larger scales than STAT, even given the same compute resources. This gain comes from the ability to utilize all of the processors on a Symmetric Multiprocessor (SMP) node's processing cores. Take, for example, a parallel machine with n-way SMP nodes, for some number n. With STAT, a single daemon is launched on each compute node and is responsible for gathering traces from all n of the application processes running on that node. STATBench, on the other hand, can launch its daemon emulators on all n of the SMP node's processing cores and have each daemon emulator generate n traces. The net effect is that STATBench only requires $1/n$ of the machine to emulate a STAT run on the full machine. Alternatively, we can use the full machine to emulate the scale of a machine that has n times as many compute nodes. The daemon emulators can also generate an arbitrary amount of traces, thus providing further insight for machines with more processing cores per compute node than the machine running STATBench.

4 Results

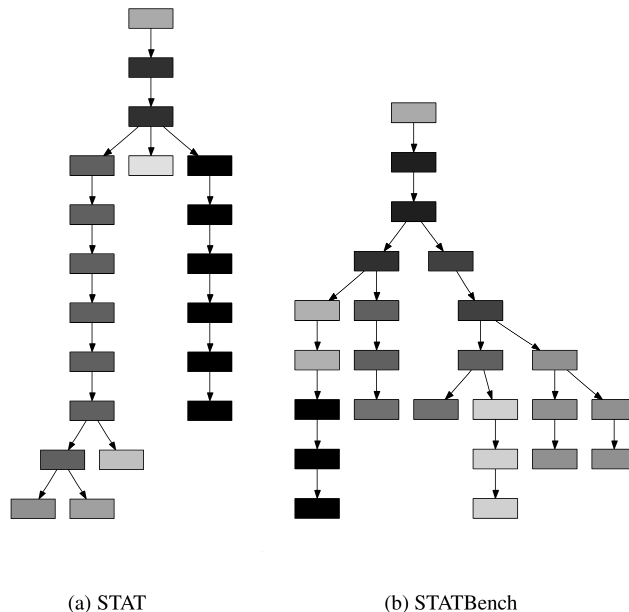
We first evaluate STATBench by comparing its performance to our STAT prototype. Next, we present the results from scaling STATBench to BG/L scales. Finally, we use those results to guide optimization of the tool's scalability.

4.1 STAT Benchmark vs. STAT

In order to evaluate STATBench's ability to model the performance of STAT, we compare results previously gathered from STAT with STATBench results. Both sets of results were gathered on Thunder, a 1024 node cluster at LLNL with four 1.4 GHz Intel Itanium2 CPUs and 8GB of memory per node. The nodes are connected by a Quadrics QsNet^{II} interconnect with a Quadrics Elan 4 network processor.

The results from STAT were gathered from debugging an MPI ring communication program with an artificially injected bug. In this application, each MPI task performs an asynchronous receive from its predecessor in the ring and an asynchronous send to its successor, followed by an MPI_Waitall that blocks pending the completion of those requests. All of the tasks then synchronize at an MPI_Barrier. A bug is introduced that causes one process to hang before its send. An example 3D-Trace/Space/Time call graph prefix tree for this program, with node and edge labels removed, can be seen in Figure 2(a)

STATBench was run with a set of parameters designed to match the STAT output from the MPI message ring program, particularly with respect to the depth, breadth, and node



(a) STAT (b) STATBench

Figure 2. Example structure (node and edge labels removed) of 3D-Trace/Space/Time call prefix trees from (a) STAT and (b) STATBench.

count of the call prefix tree. To achieve this goal, we specify three traces per simulated task, a maximum call depth past *main()* of seven, a call breadth of two, and five equivalence classes. An example output can be seen in Figure 2(b). While this output does not exactly model STAT's output with the ring program, it does provide an approximation that is conservative; it is slightly more complex than the real application with respect to call graph topology and node count.

More than the visual comparison, our goal is to model the performance of STAT. We ran STATBench at various scales, using the same MRNet topologies that were used for STAT. Specifically, we employ a 2-deep tree, with one layer of communication processes between the STATBench front-end and back-ends. At all scales, we employ a balanced tree: all parent processes have the same number of children. Furthermore, on Thunder each STAT daemon gathered traces from four application processes, hence STATBench emulates four processes per daemon. Figure 3 shows that STATBench's performance closely models STAT, taking a few hundredths of a second longer on average. These results are not too surprising as the communication tree topologies of STATBench are equivalent to those of STAT. The slight increase in time could come from the fact that the STATBench parameters chosen result in a few more call graph nodes in the output than the STAT counterpart.

4.2 Simulating STAT on BlueGene/L

Having validated STATBench's ability to model STAT on Thunder, we next emulate STAT on BG/L. BG/L is a massively parallel machine at LLNL with 65,536 compute nodes. Each

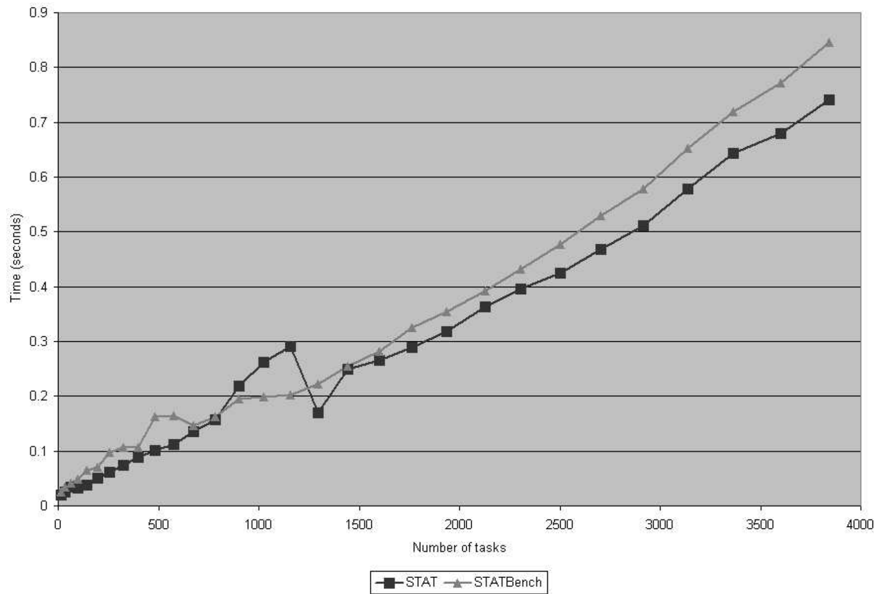


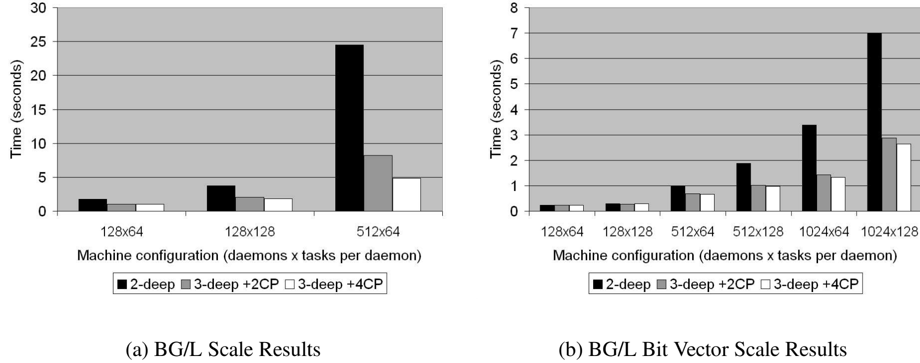
Figure 3. STAT versus STATBench on Thunder.

compute node has two PowerPC cores and runs a custom lightweight kernel (Compute Node Kernel) that does not support multi-threading and only implements a subset of the standard Linux system calls. I/O operations for the compute nodes are executed by an associated I/O node. LLNL's BG/L configuration has 1024 I/O nodes, each responsible for 64 compute nodes. The I/O nodes are also responsible for running any debugger daemons. BG/L has two modes of operation, co-processor mode and virtual node mode. In co-processor mode one of the two compute node cores runs an application task, while the other core handles communication. Virtual node mode, on the other hand, utilizes both compute node cores for application tasks, scaling up to 131,072 tasks.

4.2.1 Initial Results

Our emulation was performed on uBG/L, a single rack BG/L machine. The architecture of uBG/L is similar to BG/L, with the main difference being the compute node to I/O node ratio. Instead of a sixty-four to one ratio, uBG/L has an eight to one ratio with a total of 128 I/O nodes and 1024 compute nodes. By running STATBench on all of uBG/L's 1024 compute nodes, we are able to emulate a full scale run of STAT on BG/L, where a daemon is launched on each of BG/L's 1024 I/O nodes and debugs 64 compute nodes.

We ran STATBench on uBG/L with similar parameters that were used on Thunder. One major change was that we ran tests with both 64 and 128 tasks per daemon to emulate co-processor mode and virtual node mode respectively. We tested 2-deep and 3-deep MRNet topologies that were dictated by the machine architecture. For the 2-deep tree, the STATBench front-end, running on the uBG/L front-end node, connects to all communication processes on the I/O nodes, each communication processes in turn connects to



(a) BG/L Scale Results

(b) BG/L Bit Vector Scale Results

Figure 4. BG/L scale STATBench performance results with (a) string-based task lists and (b) bit vector task lists.

eight STATBench daemon emulators on the compute nodes. The 3-deep tree employs an additional layer of communication processes residing between the STATBench front-end process and the I/O node communication processes. We tested configurations with two and four processes in this layer, in both cases running on the four CPU uBG/L front-end node.

Using these parameters, we attempted to scale STATBench to 128, 512, and 1024 daemon emulators. The results of these tests can be seen in Figure 4(a). STATBench modestly scaled when emulating up to 512 daemons with 64 tasks per daemon, especially when using a 3-deep tree with the additional layer of four communication processes. However, STATBench was only able to run up to this scale, half of the full BG/L machine in co-processor mode, which only represents 25% of BG/L's maximum process count. At larger scales, STATBench failed because of the quantity and the size of the task lists used for the edge labels. The task lists were implemented as strings in the STAT prototype. For example, the set $\{1,3,4,5,6,9\}$ would be translated into the string "1,3-6,9". At greater scales, such a representation can grow prohibitively large, up to 75KB per edge label when emulating 32,768 tasks. Doubling this string length to 150K in order to represent twice as many tasks overloaded the communication processes, which received graphs from multiple children. Even if STAT could run at these scales with the string representation, the amount of data being sent and processed would have taken an intolerable amount of time.

4.2.2 Optimizing STAT

To overcome the scaling barrier caused by the string representation of the task lists, we instead implement each task list as a bit vector. In this implementation, one bit is allocated per task and its respective bit is set to *1* if the task is in the list, *0* otherwise. The bit vector benefits not only from requiring only one bit per task but also from merging the task lists with a simple *bitwise or* operation.

The results for using the bit vector at BG/L scales can be seen in Figure 4(b). These results show a substantial improvement over the string implementation of the task lists. At the largest successful scale with the string implementation, 512 daemons with 64 tasks each, the best time was nearly five seconds, compared to two-thirds of a second with the bit vector implementation. The emulation of the full BG/L machine in virtual node mode,

with 131,072 processes, required just over two and a half seconds to merge all of the traces. It is worth noting that going from a 2-deep tree to a 3-deep tree resulted in a substantial improvement, especially at larger scales. Although much less significant, going from a layer of 2 communication processes to 4 communication processes also had some benefit.

5 Conclusion and Future Work

We have presented STATBench, an intermediate step in the development of the Stack Trace Analysis Tool. Our evaluation of STATBench indicates that it provides accurate results in its emulation of STAT. Using STATBench, we were able to emulate scaling runs of STAT on BlueGene/L, which allowed us to identify and to fix a scalability bug in the STAT prototype implementation. With this fix, STATBench estimates that STAT will be able to merge stack traces from 131,072 tasks on BG/L in under three seconds.

Running STAT on BG/L with the same MRNet topology as our STATBench tests will require additional computational resources, particularly for the layer of 128 communication processes that were run on the I/O nodes during our STATBench tests on uBG/L. We hypothesize that STAT performance will not suffer much by reducing this layer to 64 or 32 communication processes, however this remains to be tested once STAT has been deployed on BG/L. In any case, STAT can utilize the computational resources of BG/L's 14 front-end nodes and the visualization cluster connected to BG/L.

Our design of STATBench is general for addressing the scalability of TBŌN-based tools. In such tools, a single back-end daemon is typically responsible for analyzing multiple application processes on a compute node. A benchmark for such a tool does not need to restrict itself to a single back-end daemon per compute node; rather, it can run one back-end daemon on each of a compute node's processing cores. Each daemon can then emulate any number of processes by generating artificial, yet representative tool data (i.e., stack traces in the case of STATBench). The ability of a TBŌN-based tool benchmark to run on all of a compute node's processing cores, combined with the emulation of an arbitrary number of application processes, allows for tests to be performed at larger scales than the actual tool running on the same compute resources.

References

1. TotalView Technologies, TotalView `http://www.totalviewtech.com/productsTV.htm`
2. B. R. Buck and J. K. Hollingsworth, *An API for Runtime Code Patching*, The International Journal of High Performance Computing Applications **14**, 4 (317–329)2000.
3. P. Roth, D. Arnold, and B. Miller, *MRNet: A Software-Based Multicast/Reduction Network for Scalable Tools*, Proceedings of the IEEE/ACM Supercomputing '03 **nov**, 2003 (.)
4. 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*, 21st International Parallel and Distributed Processing Symposium 2007 **mar**, 2007 (.)