LLNL-TR-537811

Program Development Tools and Infrastructures

M. Schulz

March 13, 2012

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

Program Development Tools and Infrastructures
Activities by ASC CSSE/CCE and Partners

Martin Schulz (LLNL, schulzm@llnl.gov)

Dong Ahn (LLNL), Bronis de Supinksi (LLNL), Todd Gamblin (LLNL),
Samual Gutierrez (LANL), Greg Lee (LLNL), Matthew Legrende (LLNL),
Anthony Machado (LANL), Michael Mason (LANL), David Montoya (LANL),
Mahesh Rajan (SNLs)

In collaboration with: Dorian Arnold (University of Wisconsin) Jim Galarowicz (Krell),
William Hachfeld (Krell), Tobias Hilbrich (TU-Dresden),
Jeff Hollingsworth (University of Maryland), Don Maghrak (Krell),
Bart Miller (University of Wisconsin), Matthias Mueller (TU-Dresden)

MOTIVATION

Exascale class machines will exhibit a new level of complexity: they will feature an unprecedented number of cores and threads, will most likely be heterogeneous and deeply hierarchical, and offer a range of new hardware techniques (such as speculative threading, transactional memory, programmable prefetching, and programmable accelerators), which all have to be utilized for an application to realize the full potential of the machine. Additionally, users will be faced with less memory per core, fixed total power budgets, and sharply reduced MTBFs.

At the same time, it is expected that the complexity of applications will rise sharply for exascale systems, both to implement new science possible at exascale and to exploit the new hardware features necessary to achieve exascale performance. This is particularly true for many of the NNSA codes, which are large and often highly complex integrated simulation codes that push the limits of everything in the system including language features.

To overcome these limitations and to enable users to reach exascale performance, users will expect a new generation of tools that address the bottlenecks of exascale machines, that work seamlessly with the (set of) programming models on the target machines, that scale with the machine, that provide automatic analysis capabilities, and that are flexible and modular enough to overcome the complexities and changing demands of the exascale architectures. Further, any tool must be robust enough to handle the complexity of large integrated codes while keeping the user's learning curve low.

With the ASC program, in particular the CSSE (Computational Systems and Software Engineering) and CCE (Common Compute Environment) projects, we are working towards a new generation of tools that fulfill these requirements and that provide our users as well as the larger HPC community with the necessary tools, techniques, and methodologies required to make exascale performance a reality.

PERFORMANCE ANALYSIS TOOLS

To reach exascale, performance tools will no longer be a luxury for power users, but an essential infrastructure for guiding both applications and the software stack developers to exascale. Performance tools for exascale systems must help developers identify shortcomings in the software stack as well as provide on-line performance feedback to guide runtime adaptation. As such, we require a series of tool sets that not only provide a range of advanced analysis capabilities, but that are intuitive and easy-to-use as well as available across a range of platforms.

The Open|SpeedShop (O|SS) [1] project, a joint project between LANL, LLNL, SNLs and the Krell Institute, specifically targets the ideas of ease-of-use and cross-platform availability. Installed on most major DOE (NNSA and ASCR) laboratory systems it provides users with rich set of performance analysis functionality, incl. PC sampling, call stack sampling, hardware counter experiments, as well as a wide range of tracing experiments. User can apply O|SS using simple prefix commands and can analyze the data in a comprehensive GUI.

Tools like Loba, developed at LANL, offer a different avenue: it provides a simple tool framework that attempts to automate the collection, evaluation, and application of mapping heuristics for MPI applications, an issue of rising importance on today's multi-core/multi-socket node systems. It features a range of mapping algorithms and uses profiling data to generate new MPI task placements for subsequent runs with similar communication characteristics. Experiments with several benchmarks have shown performance improvements of up to 16%.

DEBUGGING AND VERIFICATION TOOLS

The increased complexity and core counts of exascale systems will diminish the effectiveness of traditional interactive debuggers. To cope with the complexity of exascale executions, application developers will need additional tools that can help users to either automatically or semi-automatically reduce the problem to smaller core counts or to detect the problem itself.

The Stack Trace Analysis Tool (STAT) targets this problem and provides is a lightweight and highly scalable mechanism for identifying errors in code running at full scale [2]. It has been developed in close collaboration between LLNL, the University of Wisconsin, and the University of New Mexico, and works on the principle of detecting and grouping similar processes at suspicious points in a programs execution. STAT gathers stack traces across tasks and over time and merges the traces into a call graph prefix tree, from which it identifies the task equivalence classes. The tool has proven effective even at very large scales; it has demonstrated sub-second merging latencies on 212,992 tasks [3].

In addition to traditional debuggers, users will require new mechanisms that proactively check programs for correctness instead of reactively having to debug them. One example in this area is the Marmot Umpire Scalable Tool (MUST), developed at TU-Dresden in close collaboration with LANL and LLNL, which unites and extends the functionality of two existing MPI runtime error detection tools, namely Marmot [4] and Umpire [5]. Using a Generic Tool Infrastructure (GTI) [6], developed for event driven tools like MUST, as the underlying foundation, MUST implements a range of different correctness checks, each targeting different correctness aspects, and distributes them across the system for efficient online checking of the application as it executes. Violations of the use of MPI, such as potential deadlocks, resource leaks, or mismatches messages are detected and reported to the user.

Underlying Tool Infrastructures

The discussion above shows that we will need sophisticated tools to address the complexities of the target applications and systems. Each tool will itself be a highly distributed system and require substantial effort to implement and tune. On the other hand, no single tool will be able solve all problems - instead we will need the ability to create and maintain custom tools for particular problems or target platforms.

The use of generic and separable components will be key to overcoming these challenges: each functionally separable part of a tool should be implemented as its own component, which then is made available as part of a component library. Tools can assemble these components into a full end-to-end solution with minimal glue code. In the ideal case, tools may even be assembled directly from components alone using a tool construction specification (e.g., implemented as an XML file). To realize this vision, ASC teams are currently developing component infrastructures such P^{N}MPI and CBTFF.

P^{N}MPI eliminates the restriction of only being able to use a single tool layer in the MPI profiling interface (PMPI) per execution [7]. It can dynamically load and chain multiple PMPI tools into a single tool stack and then interject this complete stack between the target application and the library without changing the view for each individual tool. Additionally, a registration mechanism enables modules to offer services to other modules loaded by P^{N}MPI and thereby enables code reuse by modularizing common tasks, like datatype flattening or request tracking.

The *Component Based Tool Framework* (CBTF), jointly developed by the Krell Institute, LLNL, LANL, ORNL, and the Universities of Wisconsin and Maryland under joint NNSA/ASCR funding, provides a scalable tree-based data transport and dynamic aggregation system (CBTF-mrnet) built on top of MRNet [8]. Users of CBTF can develop individual analysis components and deploy them in CBTF through a series of component networks that run on various levels of the CBTF transport tree. Using a dataflow principle, these component networks are used to analyze performance and debugging data on the fly during its transport.

Summary

A rich program development environment, consisting of both comprehensive debugging and correctness tool support on one side and performance analysis on the other side, will be essential in reaching exascale performance. Tools like Open|SpeedShop , Loba, STAT, and MUST are examples of tools that contribute to this goal. The discussion, however, also shows that tools themselves will be complex distributed systems and we can't afford to write each from scratch. We will have to rely on component based tool infrastructures, such as P^{N}MPI and CBTF, to manage, prototype, implement, and deploy tools.

Acknowledgements

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-xxxxxx).

References

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

[2] 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.

[3] G. L. Lee, D. H. Ahn, D. C. Arnold, B. R. de Supinski, M. Legendre, B. P. Miller, M. Schulz, and B. Liblit, "Lessons learned at 208k: towards debugging millions of cores," in *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*. Piscataway, NJ, USA: IEEE Press, 2008, pp. 1–9.

[4] B. Krammer and M. S. Müller, "MPI Application Development with MARMOT," in *PARCO*, ser. John von Neumann Institute for Computing Series, vol. 33. Central Institute for Applied Mathematics, Jülich, Germany, 2005, pp. 893–900.

[5] J. S. Vetter and B. R. de Supinski, "Dynamic Software Testing of MPI Applications with Umpire," *Supercomputing, ACM/IEEE 2000 Conference*, pp. 51–51, 04-10 Nov. 2000.

[6] T. Hilbrich, M. S. Müller, B. R. de Supinski, M. Schulz, and W. E. Nagel, "GTI: A Generic Tools Infrastructure for Event Based Tools in Parallel Systems," in *To appear in IPDPS 2012: Procedings of the 26th IEEE International Parallel & Distributed Processing Symposium*, 2012.

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

[8] P. Roth, D. Arnold, and B. Miller, "MRNet: A software-based multicast/reduction network for scalable tools," in *Proceedings of IEEE/ACM Supercomputing '03*, Nov. 2003.

Program Development Tools and Infrastructures
Activities by ASC CSSE/CCE and Partners

Martin Schulz (LLNL, schulzm@llnl.gov)

Dong Ahn (LLNL), Bronis de Supinksi (LLNL), Todd Gamblin (LLNL), Samual Gutierrez (LANL), Greg Lee (LLNL), Matthew Legrende (LLNL), Anthony Machado (LANL), Michael Mason (LANL), David Montoya (LANL), Mahesh Rajan (SNLs)

In collaboration with: Dorian Arnold (University of Wisconsin) Jim Galarowicz (Krell), William Hachfeld (Krell), Tobias Hilbrich (TU-Dresden), Jeff Hollingsworth (University of Maryland), Don Maghrak (Krell), Bart Miller (University of Wisconsin), Matthias Mueller (TU-Dresden)

***Abstract*—As we move towards exascale, users will require more sophisticated tools to tackle the challenges imposed by the rising complexity of both architectures and applications. These tools must be intuitive and easy to use, present information in a concise and scalable way, and at the same time must scale to the full scale of the machine and provide a robust program development environment.**

Research teams at all three NNSA laboratories — LANL, LLNL, and SNLs — are tackling these challenges and user requirements through a wide range of tool projects targeting both current and future architectures. In the following we will describe some of these approaches from all areas of tools: we will describe the performance tools Open|SpeedShop and Loba, the debugging and correctness tools STAT and MUST, as well as tool infrastructure components, including the modular frameworks P^{N}MPI and CBTF.

I. Motivation: Working Towards a New Generation of Tools

Exascale class machines will exhibit a new level of complexity: they will feature an unprecedented number of cores and threads, will most likely be heterogeneous and deeply hierarchical, and offer a range of new hardware techniques (such as speculative threading, transactional memory, programmable prefetching, and programmable accelerators), which all have to be utilized for an application to realize the full potential of the machine. Additionally, users will be faced with less memory per core, fixed total power budgets, and sharply reduced MTBFs.

At the same time, it is expected that the complexity of applications will rise sharply for exascale systems, both to implement new science possible at exascale and to exploit the new hardware features necessary to achieve exascale performance. This is particularly true for many of the NNSA codes, which are large and often highly complex integrated simulation codes that push the limits of everything in the system including language features.

To overcome these limitations and to enable users to reach exascale performance, users will expect a new generation of tools that help address the bottlenecks of exascale machines, that work seamlessly with the (set of) programming models on the target machines, that scale with the machine, that provide the necessary automatic analysis capabilities, and that are flexible and modular enough to overcome the complexities and changing demands of the exascale architectures. Further, any tool must be robust enough to handle the complexity of large integrated codes, while keeping the user's learning curve low.

With the ASC program, in particular the CSSE (Computational Systems and Software Engineering) and CCE (Common Compute Environment) projects, we are working towards a new generation of tools that fulfill these requirements and that provide our users as well as the larger HPC community with the necessary tools, techniques, and methodologies required to make exascale performance a reality. In the following we present several examples of our activities in the areas of performance analysis tools, debugging and verification tools, as well as tool infrastructures.

II. Performance Analysis Tools

To reach exascale, performance tools will no longer be a luxury for power users, but an essential infrastructure for guiding both applications and the software stack developers to exascale. Performance tools for exascale systems must help developers identify shortcomings in the software stack as well as provide on-line performance feedback to guide runtime adaptation. As such, we require a series of tool sets that not only provide a range of advanced analysis capabilities, but that are intuitive and easy-to-use as well as available across a range of platforms. In the following we illustrate two example of performance analysis tools: Open|SpeedShop is a performance analysis tool set that is being developed as a portable and easy-to-use tool solution across all ASC platforms; and Loba, a semi-automatic tool set that helps users with MPI task placement on multi-core/multi-socket node architectures.

A. *Open|SpeedShop*

Performance analysis and optimization is a critical step in the development process of any scientific application. Efficient and easy-to-use tool support is essential to allow users to complete this task, yet current tools are often targeted for the performance analysis expert, and therefore cumbersome to use or require changes to the compilation or execution process. This makes these tools unattractive for application developers who often have limited time allocated for performance optimizations.

To overcome these problems, the ASC tri-labs and the Krell Institute, have designed Open|SpeedShop (O|SS) [1], an open source multi platform Linux performance tool that provides easy access to an application's performance profile, while not precluding more sophisticated and detailed analysis found in other tools. For this purpose it combines guided performance analysis through graphical wizards and preconfigured detailed analysis panels with low-level access to performance data through scripting interfaces and Python integration.

O|SS's performance analysis functionality includes a set of specific *Experiments* that allow the user to easily gather a variety of different performance statistics about an application. This includes Program Counter (PC) sampling, a light weight way to get an overview of application performance bottlenecks; Call Stack Sampling analysis, a technique to find hot call paths; Hardware Performance Counters, providing access to low level information such as cache or TLB misses; MPI Profiling and Tracing, enabling users to detect MPI communication bottlenecks; I/O Profiling and Tracing to study an application's I/O characteristics; and Floating Point Exception (FPE) analysis to detect floating point exceptions that can slow down applications.

The tool set offers two methods for instrumentation and data collection: an offline option that instruments the application at job start, produces unprocessed raw files at runtime, and then automatically postprocesses

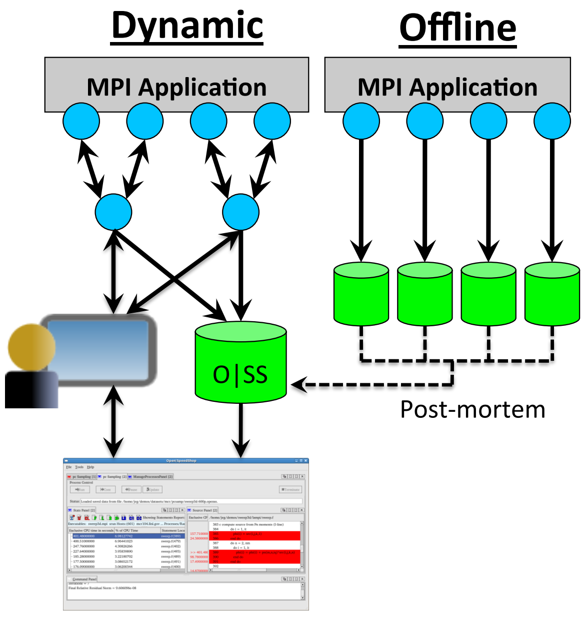
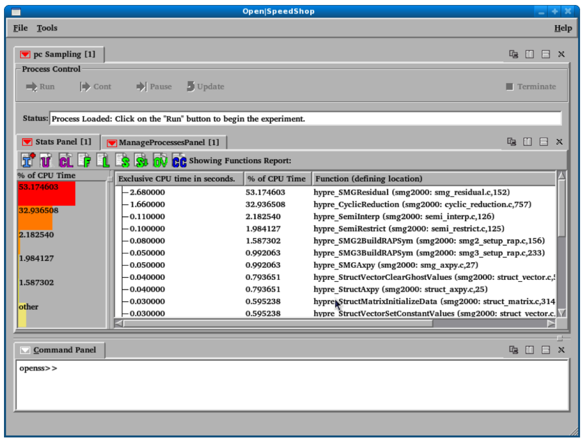


Fig. 1. The O|SS data collection architecture: offline vs. online data collection.

these files for later analysis; and an online option that collects the data using a hierarchical overlay network (Figure 1). While the first option typically is easier to install and allows for less instrumentation overhead, the second option provides advanced scalability as well as new functionality like being able to attach to already running applications. In both cases, though, the data is stored in the form of a single persistent relational database that can queried at any time after the experiment for further analysis.

Once collected, O|SS displays the data through a set of detailed reports that allow the user to easily relate the performance information back to their application source code. This information is accessible through a comprehensive GUI from a command line interface, as well as from within Python scripts. A sample screenshot of O|SS's GUI is shown in Figure 2. Additionally, the tool set includes a series of analysis techniques, including outlier detection, load balance analysis, and cross experiment comparisons. In summary, O|SS's functionality provides a comprehensive set of techniques that greatly aid in the analysis and understanding of parallel application performance.

Open|SpeedShop currently supports all major ASC platforms, including the TLCC and TLCC-2 cluster systems, BG/L and BG/P (a port to BG/Q is on the way and will be available soon), and Cray XT-4/5 and XE-6. Additionally, O|SS has been ported to a range of Linux distributions as well as SGI's Altix systems.



Status: Process Loaded: Click on the "Run" button to begin the experiment.

Showing Functions Report:

Exclusive CPU time in seconds.	% of CPU Time	Function (defining location)
2.680000	53.174603	hypre_SMGResidual (smg2000: smg_residual.c,152)
1.660000	32.936508	hypre_CyclicReduction (smg2000: cyclic_reduction.c,757)
0.110000	2.182540	hypre_SemiInterp (smg2000: semi_interp.c,126)
0.100000	1.984127	hypre_SemiRestrict (smg2000: semi_restrict.c,125)
0.080000	1.587302	hypre_SMG2BuildRAPSym (smg2000: smg2_setup_rap.c,156)
0.050000	0.992063	hypre_SMG3BuildRAPSym (smg2000: smg3_setup_rap.c,233)
0.050000	0.992063	hypre_SMGAxpy (smg2000: smg_axpy.c,27)
0.040000	0.793651	hypre_StructVectorClearGhostValues (smg2000: struct_vector.c,
0.040000	0.793651	hypre_StructAxpy (smg2000: struct_axpy.c,25)
0.030000	0.595238	hypre_StructMatrixInitializeData (smg2000: struct_matrix.c,314
0.030000	0.595238	hypre_StructVectorSetConstantValues (smg2000: struct_vector.c.

% of CPU Time
53.174603
32.936508
2.182540
1.984127
1.587302
other

```
openss>>
```

Fig. 2. The O|SS GUI showing per function statistics.

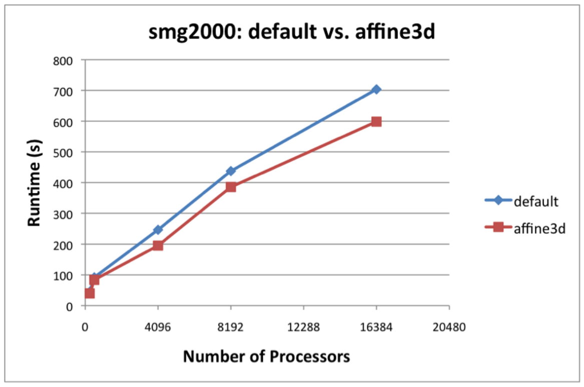


Fig. 3. Performance improvement on SMG2000 using Loba.

Within the DOE laboratories, O|SS is currently installed on all cluster platforms at the tri-labs, LLNL's BG/L and BG/P installations, LANL/SNLs Cray-XE6 installations, as well as Intrepid/Challenger at ANL, Hopper at NERSC/LBL, and Jaguar at ORNL. More information, including extensive user documentation and tutorials, is available at http://www.openspeedshop.org/.

B. Loba: Heuristics for Topology Aware Task Placement

Modern architectures often feature complex node architectures and leave users with the need to optimize node and task placements. Reordering MPI tasks to optimize for a given set of performance criteria is a relatively well-known technique in the high-performance computing community to address this problem Successfully applying this technique, however, not only requires an in-depth knowledge surrounding the applications messaging characteristics, but also requires an understanding about the target architecture and its topology. Loba is a simple tool framework that attempts to automate the collection, evaluation, and application of mapping heuristics for MPI applications. It features a range of mapping algorithms and uses profiling data to generate new MPI task placements for subsequent runs with similar communication characteristics.

Recently, LANL implemented and evaluated experimental as well as already well-established mapping techniques on Cielo, a capability-class platform for the Advanced Simulation and Computing Program. Figure 3 shows the results for one key benchmark, the SMG2000 code [2] from the ASC Purple benchmark suite [3]. This benchmark is communication intense and therefore highly sensitive to node and task placements. By using an affine-3D mapping heuristics [4], Loba was able to optimize the performance of SMG over the default performance by around 16%.

III. Debugging and Verification Tools

The increased complexity and core counts of exascale systems will diminish the effectiveness of traditional interactive debuggers. To cope with the complexity of exascale executions, application developers will need additional tools that can help users to either automatically or semi-automatically reduce the problem to smaller core counts or to detect the problem itself.

In the following we illustrate two major ASC efforts in this area tackling debugging and correctness, while specifically addressing scalability: STAT, the Stack Trace Analysis Tool, enables users to quickly identify groups of processes with similar behavior as well as individual outliers, enabling them to reduce the scale of the debugging problem; and MUST, a new tool set of MPI code verification.

A. The Stack Trace Analysis Tool (STAT)

STAT is a lightweight and highly scalable debugging tool for identifying errors in code running at full scale [5], [6]. It has been developed in close collaboration between LLNL, the University of Wisconsin, and the University of New Mexico, and works on the principle of detecting and grouping similar processes at suspicious points in a programs execution. This allows users to reduce the problem they are trying to debug to only a small and tractable number of nodes by picking representatives from each group instead of having to debug all processes at the same time. It also automatically identifies outliers, processes that cannot be grouped and/or that behave substantially different. This is often an indication of an erroneous execution and STAT can aid in quickly identifying such anomalies. STAT achieves this grouping of processes by examining the state of all

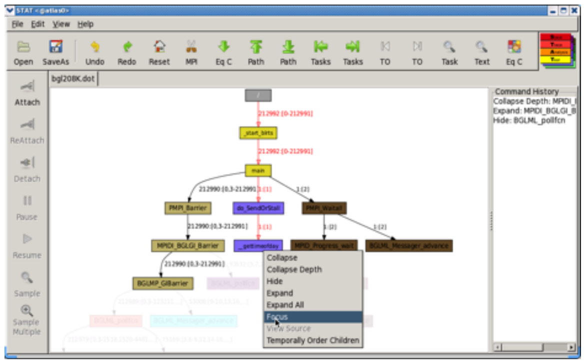


Fig. 4. The STAT GUI.

processes in a parallel program dynamically at runtime and by extracting stack traces, the calling sequence of functions that lead to the current point of execution.

STAT gathers stack traces across tasks and over time and merges the traces into a call graph prefix tree, from which it identifies the task equivalence classes. Users can then attach a traditional parallel debugger to a single representative of a class with suspicious behavior or to representatives from several classes. In either case, the technique presents the user with much less data that is targeted at the underlying problem.

STAT builds on scalable and portable tool infrastructure such as the MRNet tree-based overlay network [7]. As a result, it has been ported and deployed on many high-end computing systems including Linux clusters, BlueGene/L and P, and the Cray XT4 and 5. The tool has also proven effective even at very large scales; it has demonstrated sub-second merging latencies on 212,992 tasks [6].

STAT includes a powerful and intuitive Graphical User Interface (GUI) (see Figure 4) that allows the user to identify quickly where a bug exists in an application. The GUI automatically can perform several operations that analyze the state of the application and pinpoint potential locations of a bug. For instance, it can identify and highlight individual outliers tasks with anomalous behavior. The heuristics on which these operations are based reflect several years of debugging expertise and experiences with real world bugs. For bugs that these heuristics cannot identify, the user can manually navigate the gathered debug information within the STAT GUI. The user can then use the STAT GUI to correlate this debug information to the precise source code location or to reason about where to look for bugs using additional tools.

STAT is available on all major ASC tri-lab production platforms, which includes both the TLCC and TLCC-2 cluster systems, as well as the advanced architectures BG/L, BG/P (dawn), and Cray-XE6 (cielo). It has further been integrated into Cray's default software stack, which not only makes it available on all Cray platforms, including ORNL's Jaguar, but also makes it a prime example of transferring basic research work, which started as a student intern project, to production software with an impact beyond the ASC laboratories.

B. MUST: Scalable MPI Code Verification

The Marmot Umpire Scalable Tool (MUST) unites and extends the functionality of two existing MPI runtime error detection tools, namely Marmot [8] and Umpire [9]. Further, MUST aims at providing increased scalability along with the ability to easily add further correctness checks.

To achieve these goals, we developed the Generic Tool Infrastructure (GTI) [10] as the underlying foundation. Using GTI, the tool developer only writes the tool analyses that GTI loads, manages, and activates. Figure 5 (left) illustrates the GTI tool development process. GTI handles all infrastructure related tasks, which requires the tool developer to specify which events should trigger the analyses and the overall tool layout. GTI provides the underlying infrastructure that reads this information and generates all required code including code for wrapping, trace record creation, data transport, and analysis management. It itself relies on the P^{N}MPI , which is described in more detail in Section IV-B.

Using the abstractions of the GTI we define different types of module to implement the necessary correctness checks for MPI programs. These modules fall into three different categories: correctness checks, resource trackers, and base services. The actual correctness checks receive information about MPI calls that are issued by the application and check them for conformance with the MPI standard. The resource trackers are used to survey the creation, destruction and state of MPI resources such as requests and datatypes. They provide this information to the correctness checks. Finally, the modules for base services provide a mechanism for logging correctness errors or other types of reports as well as identifiers for MPI ranks and call locations. These identifiers refer to a certain process and thread as well as to a call stack, which can be retrieved with the StackwalkerAPI (see Section IV-A0a). Each module specifies on which other modules it depends. Based on this information the GTI is able to assemble a specific and customized MUST instance by combining the individual modules. Further, the GTI allows modules to run on additional processes or threads in order to offload tool computations from the application tasks. A TBON can be specified to run

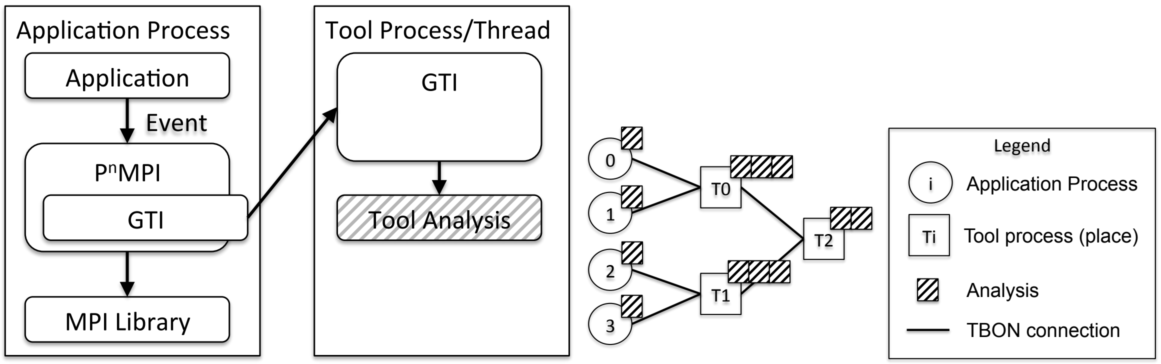


Fig. 5. GTI modular instrumentation (left) and module off-loading approach (right).

distributed and hierarchical tool analyses. This is further illustrated in Figure 5.

MUST is currently being developed at TU-Dresden in a close collaboration with LLNL and LANL. Early prototypes are available for testing with a first full release expected towards the end of the year 2012. It is currently being tested on tri-lab cluster systems.

IV. Tool Infrastructures

The discussion above shows that we will need sophisticated tools to address the complexities of the target applications and systems. Each tool will itself be a highly distributed system and require substantial effort to implement and tune. The use of generic and separable components will be key to creating a series of tool sets that can target all necessary functionalities needed from tools to reach exascale, while being maintainable and avoid re-engineering for every tool set.

In the following he highlight a series of infrastructure components that ASC is developing. We start with basic system components that can be used in a large variety of tools, followed by two larger component or module infrastructures: P^{N}MPI , a virtualization layer for the MPI profiling interface; and the Component Based Tool Framework (CBTF), a modular framework to create scalable and distributed tools.

A. Specialized Components

To aid the componentization, ASC is (co-)developing and maintaining a series of specialized components that are used in many tools, both with and beyond the tri-labs. Two of the most widely used are the Stackwalker API, an API to generate stack traces; and LaunchMON, a set of components that help launch and manage tool daemons in large scale parallel environments.

a) StackwalkerAPI: StackwalkerAPI is a tool component from the DyninstAPI toolkit, supported through collaborative efforts between the University of Wisconsin, University of Maryland and LLNL. StackwalkerAPI provides a platform-independent abstraction for collecting call stacks from running threads. It can operate in first-party or third-party modes (collecting call stacks from either its own or other processes) and over a wide range of platforms. As a tool component it is used as a common-infrastructure building block for other tools such as STAT (the Stack Trace Analysis Tools, see Section III-A), P^{N}MPI (see Section IV-B), MUST III-B, Libra [11] and CBI (Cooperative Bug Isolation) [12], [13].

In its core, StackwalkerAPI relies on a plugin mechanism (See Figure 6) that enables it to add callback routines to follow individual types of stack elements. This allows Stackwalker to be flexible and users to easily add support for new stack types and frames as architectures evolve, without changing the interface to the tool. At the same time, it uses plugins to switch between first and third party support, as well as to integrate symbol name lookup. The latter is typically done through Dyninst's SymtabAPI.

StackwalkerAPI is available as part of Dyninst from http://www.dyninst.org/ and runs on a wide variety of platforms, including most Linux distributions, BG/L and BG/P, and Cray-XE6.

b) LaunchMON: Many parallel tools, including debuggers and performance analyzers, must launch and control tool daemons. Large scale tools also often use additional middleware daemons for scalable communication. Launching and controlling these daemons are non trivial tasks that require an efficient, portable, secure solution that can be reused across a wide set of tools.

LaunchMON [14] fills this gap using a general purpose, distributed infrastructure. It is structured into four

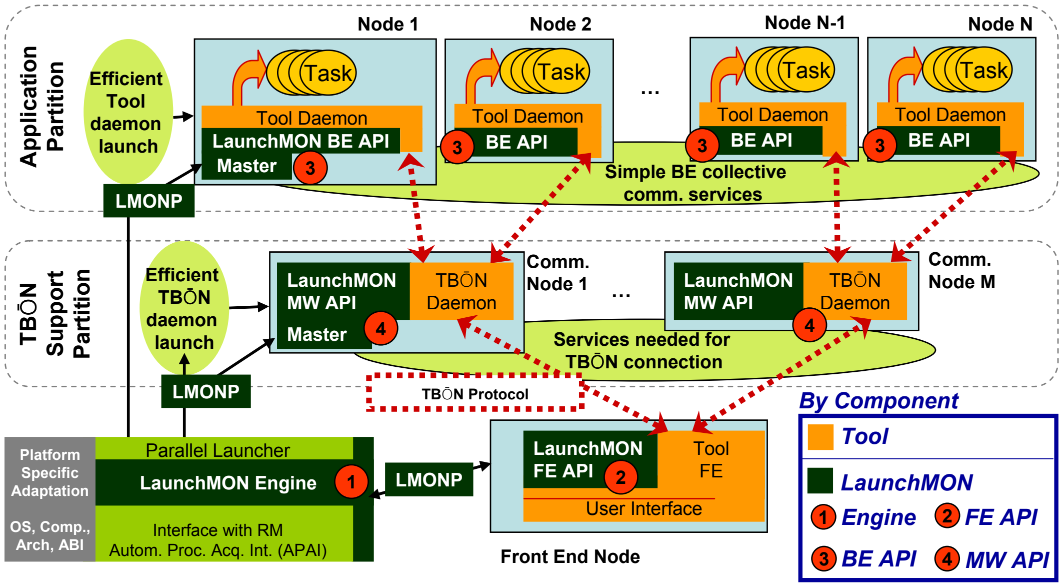


Fig. 7. Architecture of LaunchMON

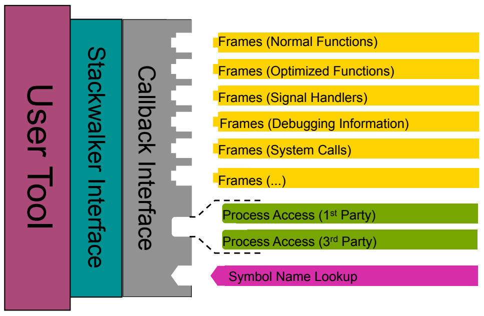

User Tool
Stackwalker Interface
Callback Interface
Frames (Normal Functions)
Frames (Optimized Functions)
Frames (Signal Handlers)
Frames (Debugging Information)
Frames (System Calls)
Frames (...)
Process Access (1st Party)
Process Access (3rd Party)
Symbol Name Lookup

Fig. 6. The Stackwalker API Plugin Infrastructure.

main components, which are shown in Figure 7: (1) the LaunchMON Engine; (2) the front-end API (LaunchMON FE API); (3) the back-end API (LaunchMON BE API); and (4) the middleware API (LaunchMON MW API). The LaunchMON FE and BE APIs provide information about application processes and scalably launch tool daemons on remote nodes. Similarly, the LaunchMON FE and MW APIs enable scalable launching and connection of tool daemons. The LaunchMON Engine interacts with the resource manager to determine when, where and how to perform the services of the other components. A compact application layer network protocol, LMONP, enables interactions between all of LaunchMON's components.

The main component, the LaunchMON Engine, directly leverages the services offered by the underlying resource manager on the target platform. Thus, it uses efficient platform specific mechanisms to launch and to manage daemons that have accepted security properties. Tool developers can control this process with a set of libraries and APIs from the front-end tool, the tool daemons running on the application nodes and, if applicable, middleware nodes. The latter two APIs include simple communication mechanisms that are useful for tool coordination.

LaunchMON is developed at LLNL and available as open source on SourceForge at http://sourceforge.net/projects/launchmon/. It currently has been ported to BG/L, BG/P, and BG/Q systems, Cray XT-4/5 and XE-6 machines, as well as Linux clusters running SLURM or OpenRTE as their resource manager. LaunchMON is used as the base component for STAT (Section III-A) and is targeted for use in CBTF (Section IV-C) and Open|SpeedShop (Section II-A).

B. P^NMPI - Virtualizing the MPI Profiling Interface

Most tools targeting MPI rely on the MPI Profiling Interface (PMPI), which allows tools to transparently intercept invocations to MPI routines and with that to establish wrappers around MPI calls to gather execution information. However, the usage of this interface is limited to a single tool. P^NMPI eliminates the restriction of a single PMPI tool layer per execution [15]. It can dynamically load and chain multiple PMPI tools into a single tool stack and then interject this complete stack between the target application and the library without

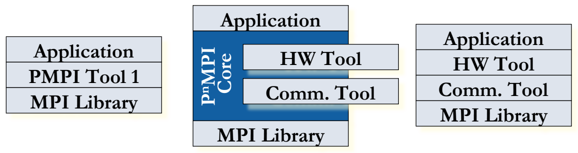


Fig. 8. Usage of the MPI profiling interface (left); combining two measurement tools using P^{N}MPI (middle); effective tool stack as seen by the application (right).

changing the view for each individual tool. It enables the user to combine arbitrary MPI tools without having to reimplement them. Figure 8 illustrates the principle behind P^{N}MPI and shows how it is possible to collect both hardware data (such as network counters) and communication data (such as MPI traces) in two independent modules concurrently.

These new capabilities can be used to compose new tools directly from existing tools or out of a library of generic services. A registration mechanism enables modules to offer services to other modules loaded by P^{N}MPI and thereby enables code reuse by modularizing common tasks, like datatype flattening or request tracking, without having to recode them in every tool. Further, a particular class of modules, *Switch Modules*, support both external steering and the multiplexing of existing tools to dynamic subsets of MPI jobs by providing the ability to dynamically switch between stacks for each intercepted MPI event. This allows users, e.g., to only forward a targeted subset of MPI calls (e.g., all calls with floating point data in the message, messages on certain communicators, or of certain size) to a profiler.

P^{N}MPI is developed at LLNL and available in open source from github at https://github.com/schulzm/PnMPI (version 1.4 scheduled for beginning of April 2012) and has been tested on a variety of Linux clusters as well as BG/P and Cray-XT systems. It forms the foundation for MUST (see Section III-B); AutomaTeD [16], an stochastical debugging tool for large scale MPI jobs; and Talanton, an asynchronous load balancing framework, currently under development at LLNL and Texas A&M University.

C. The Component-Based Tool Framework (CBTF)

The discussion above shows that we will need sophisticated tools to address the complexities of the target applications and systems. Each tool will itself be a highly distributed system and require substantial effort to implement and tune. On the other hand, no single tool will be able solve all problems - instead we will need the ability to create and maintain custom tools for particular problems or target platforms.

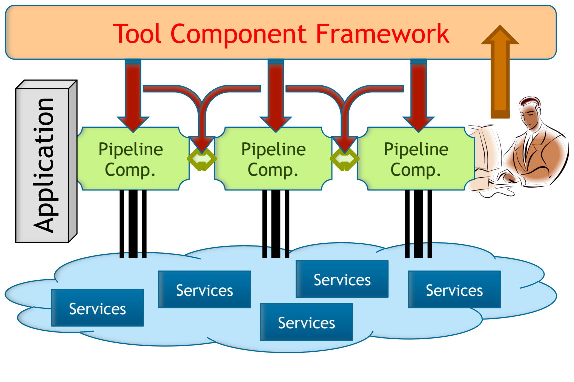


Fig. 9. The Structure of the Component-Based Tool Framework

The use of generic and separable components will be key to overcoming these challenges: each functionally separable part of a tool should be implemented as its own component, which then is made available as part of a component library. Tools can assemble these components into a full end-to-end solution with minimal glue code. In the ideal case, tools may even be assembled directly from components alone using a tool construction specification (e.g., implemented as an XML file).

To realize this vision, we are currently developing the *Component Based Tool Framework* (CBTF) [17], which is conceptually shown in Figure 9. It provides a scalable tree-based data transport and dynamic aggregation system (CBTF-mrnet) built on top of MRNet [18]. It transports data from the application from its creation point at the application level, up the MRNet tree to the client tool, where the user operate on or view the data.

Users of CBTF can develop individual analysis components and deploy them in the CBTF infrastructure through a series of component networks that run on various levels of the CBTF transport tree. Using a dataflow principle, these component networks are used to analyze performance and debugging data on the fly during its transport. These component networks allow for analysis and processing of the data items passed between the nodes of the network and provide the user of the transport mechanism the ability to filter the data as it is being transported. Filtering, in this context, means performing some type of data transformation on the data being passed to the filter and outputting the transformed data out of the filter.

Overall, CBTF will not only avoid stovepipe solutions and enable interoperability between tools, but it will also enable quick tool prototyping and the creation of custom or even application specific tools. This will allow tool providers to quickly react to new, unpredictable problems and provide users with quick and direct support without having to create specialized tools from scratch.

Work on the CBTF is done under joint DOE ASC/NNSA and ASCR funding through the project entitled "Building a Community Infrastructure for Scalable On-Line Performance Analysis Tools Around Open/SpeedShop" (ER25935). Early prototype software is available on request and the team is targeting a public release later in 2012. As a first conceptual study, the Krell Institute is re-implementing a modularized version of Open|SpeedShop (Section II-A) on top of CBTF, which will allow us to scale O|SS to next generation machines, starting with LLNL's Sequoia. In addition, the team is working on a set of administrative tools that leverage CBTF and its scalability, opening the door to supporting additional important functionality presently not covered by most tool frameworks or tool sets.

V. Looking Forward

To make exascale computing tractable, users will need sophisticated tools to maneuver the increasingly complex architecture and application space. This will include more scalable approaches for debugging and performance analysis, but will also reach into new areas such as memory efficiency analysis and optimization and power reduction. To provide these capabilities, tools themselves must face a series of challenges, be highly scalable, and be fault tolerant.

Delivering the sophisticated tools that will be required for exascale platforms will require a united effort by the tools community. The community can no longer afford to create vertically integrated stovepipe implementations; instead the community will need to establish components that can be shared across multiple tools, support the rapid development of new tools, and enable tool capabilities to be dynamically tailored in response to the system and application state and the problems at hand.

Acknowledgements

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-xxxxxx).

References

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

[2] R. Falgout and U. Yang, "hypre: a Library of High Performance Preconditioners," in *Proceedings of the International Conference on Computational Science (ICCS), Part III, LNCS vol. 2331*, Apr. 2002, pp. 632–641.

[3] Lawrence Livermore National Laboratory, "The ASCI purple benchmarks," http://www.llnl.gov/asci/platforms/purple/rfp/benchmarks, 2001.

[4] A. Bhatele, "Automating Topology Aware Mapping for Supercomputers," Ph.D. dissertation, Dept. of Computer Science, University of Illinois, August 2010, http://hdl.handle.net/2142/16578.

[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] G. L. Lee, D. H. Ahn, D. C. Arnold, B. R. de Supinski, M. Legendre, B. P. Miller, M. Schulz, and B. Liblit, "Lessons learned at 208k: towards debugging millions of cores," in *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*. Piscataway, NJ, USA: IEEE Press, 2008, pp. 1–9.

[7] 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.

[8] B. Krammer and M. S. Müller, "MPI Application Development with MARMOT," in *PARCO*, ser. John von Neumann Institute for Computing Series, vol. 33. Central Institute for Applied Mathematics, Jülich, Germany, 2005, pp. 893–900.

[9] J. S. Vetter and B. R. de Supinski, "Dynamic Software Testing of MPI Applications with Umpire," *Supercomputing, ACM/IEEE 2000 Conference*, pp. 51–51, 04-10 Nov. 2000.

[10] T. Hilbrich, M. S. Müller, B. R. de Supinski, M. Schulz, and W. E. Nagel, "GTI: A Generic Tools Infrastructure for Event Based Tools in Parallel Systems," in *To appear in IPDPS 2012: Procedings of the 26th IEEE International Parallel & Distributed Processing Symposium*, 2012.

[11] T. Gamblin, B. R. de Supinski, M. Schulz, R. J. Fowler, and D. A. Reed, "Scalable load-balance measurement for SPMD codes," in *IEEE/ACM Supercomputing (SC)*, 2008.

[12] B. Liblit, "Cooperative Bug Isolation," Ph.D. dissertation, UC Berkeley, Fall 2004.

[13] The CBI Team, "The Cooperative Bug Isolation Project," http://research.cs.wisc.edu/cbi/, 2012.

[14] 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 *Submitted to the International Conference on Parallel Processing*, Portland, OR, 2008.

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

[16] I. Laguna, T. Gamblin, B. de Supinski, S. Bagchi, G. Bronevetsky, D. Ahn, M. Schulz, and B. Rountree, ""large scale debugging of parallel tasks with automaded"," in *Proceedings of Supercomputing 2011*, 2011.

[17] The CBTF Team, "Building a Community Infrastructure for Scalable On-Line Performance Analysis Tools Around Open|SpeedShop," http://ft.ornl.gov/doku/cbtfw/start, 2012.

[18] P. Roth, D. Arnold, and B. Miller, "MRNet: A software-based multicast/reduction network for scalable tools," in *Proceedings of IEEE/ACM Supercomputing '03*, Nov. 2003.