Final Project Report – September 2012

**MOGO: Model Oriented Global Optimization
of Petascale Scientific Applications**

University of Oregon DE-SC0001777 Allen Malony malony@cs.uoregon.edu , 503-346-4407

Sponsored by the
DOE Office of Science Program Office:
Office of Advanced Scientific Computing Research

Under the
*DOE Program Announcement: Software Development Tools for
Improved Ease-of-Use of Petascale Systems - DE PS02-08ER08-19*

Contents

Report Cover Page	1
1 Summary Description of the Research Conducted	3
1.1 Products	3
1.2 Research objectives	5
2 Background: Original MOGO Approach	5
2.1 Original Proposed Research	5
2.2 Impact of Budget Reduction	6
3 Research Accomplishments - Significant results	6
3.1 Memphis: a Performance Analysis Tool for Non-Uniform Memory Patterns	6
3.1.1 Motivation	6
3.1.2 <i>Memphis</i>	7
3.1.3 Recent Work	9
3.1.4 Memphis Evaluation	10
3.1.5 Ongoing Work	11
3.1.6 Conclusion	11
3.2 Performance Analysis using TAU	11
3.2.1 PGI Accelerator Performance Measurement	11
3.2.2 TAUcuda	13
3.2.3 Library Wrapping	15
3.2.4 Other Efforts	16
3.2.5 Year 3	17
3.3 System Tuning Parameters	18
3.3.1 Paged Memory versus Big Memory	19
3.3.2 Native Torus versus TCP-Torus	19
3.3.3 Binding versus no Binding	19
3.3.4 Fusion Studies	20
3.3.5 Conclusion	20
References	21

1 Summary Description of the Research Conducted

The MOGO project was initiated under in 2008 under the DOE Program Announcemnet for Software Development Tools for Improved Ease-of-Use on Petascale systems (LAB 08-19). The MOGO team consisted of Oak Ridge National Lab, Argonne National Lab, and the University of Oregon.

The overall goal of MOGO was to attack petascale performance analysis by developing a general framework where empirical performance data could be efficiently and accurately compared with **performance expectations** at various levels of abstraction. This information could then be used to automatically identify and remediate performance problems. MOGO was based on performance models derived from application knowledge, performance experiments, and symbolic analysis.

MOGO was selected for funding by DOE, but the funded budget level was reduced to approximately 65% of the original request. This reduction was spread evenly across all three organizations. Furthermore, we were forced to reduce the scope of work on the project by one third by focusing on the near term objectives, and by reducing effort applied to more speculative research techniques outlined in the proposal.

As described later, at this level, MOGO was able to make reasonable impact on existing DOE applications and systems. New tools and techniques were developed, which, in turn, were used on important DOE applications on DOE LCF systems to show *significant* performance improvements.

1.1 Products

The MOGO project generated several products; these products included publications and presentations, software, project websites, technologies or techniques, software deployments on DOE systems, and substantial application performance improvements.

1. Impact on applications

- (a) *Memphis*-directed modifications have led to significant performance improvements of full-scale runs on ORNL's Jaguar for several applications, including a 23% improvement for dual-socket multi-threaded runs of the XGC1 fusion code, and nearly a 50% increase in throughput for single-socket multi-threaded runs of the CAM/HOMME atmosphere component of the CESM climate modeling code. These improvements have been integrated into the main applications.

2. Software (developed and deployed)

- (a) Memphis - Memphis has been deployed on ORNL Cray XT development system. Currently, it cannot be deployed directly on Jaguar or other production systems because it requires a kernel patch to the operation system, which the respective project directors are unwilling to support.
- (b) Tau¹ has been deployed on all DOE LCF systems, and many other systems throughout DOE.
 - i. TAU support for profiling PGI accelerator based directives. Released in TAU v2.20.1 on March 22, 2011:
 - ii. TAU's wrapper generator redefining library calls in calls tau_wrap.

¹TAU Performance System (R) is a federally registered trademark owned by the State of Oregon acting by and through the State Board of Higher Education on behalf of the University of Oregon.

- iii. TAU's general purpose wrapping toolkit (`tau_gen_wrapper`) supporting all three forms of wrapping: pre-processor based redirection of library calls, runtime library interposition using preloading, and linker-based instrumentation. Released in TAU v2.20.2 on May 13, 2011.
- iv. I/O wrapper library for tracking POSIX I/O using a `-optTrackIO TAU_OPTIONS` environment setting. Released in TAU v2.20.2 on May 13, 2011.
- v. CUPTI v4.1 and H2 database supported in TAU v2.21 released on November 10, 2012.
- vi. Support for tracking device memory in CUDA and tracking queue wait time in OpenCL integrated in TAU v2.21.2 released on March 26, 2012.
- vii. KTAU version 3.0 was released with support for tracepoints. This version was released on June 07, 2012.
- viii. CUDA and CUPTI 5.0 support was integrated in TAU. Along with improvements in database support for GPGPUs, TAU v2.21.3 was released on July 12, 2012.
- ix. Enhancements to the ParaProf 3D browser with support for IBM BlueGene/Q platform was released in TAU v2.21.4, released on September 18, 2012.

3. Project Websites

- (a) Memphis - <http://ft.ornl.gov>, FT Publications
- (b) Tau - <http://tau.uoregon.edu>
- (c) ZeptoOS - <http://www.zeptoos.org>, <http://bg-linux.anl-external.org/wiki>

4. Key publications (other publications are referenced in §3)

- (a) Memphis: C. McCurdy and J.S. Vetter, "Memphis: Finding and Fixing NUMA-related Performance Problems on Multicore Processors," in IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS). White Plains, NY: IEEE, 2010
- (b) Tau: A. Malony, S. Mayanglambam, S. Biersdorff, and W. Spear, "An Experimental Approach to Performance Measurement of Heterogeneous Parallel Applications using CUDA," International Conference on Supercomputing, June 2010.
- (c) ZeptoOS: K. Yoshii, H. Naik, C. Yu and P. Beckman, "Extending and Benchmarking 'Big Memory' Implementation on Blue Gene/P Linux" - International Workshop on Runtime and Operating Systems for Supercomputers (ROSS 2011), Tuscon, AZ

5. Sampling of Presentations and Training

- (a) S. Shende, "TAU Performance System", presentation at DOE ACTS Workshop, Berkeley, CA, Aug 2009, 2010, 2011, and 2012.
- (b) S. Shende, A. Knupfer, B. Wylie, M. Geimer, "Hands-On Practical Parallel Application Performance Engineering using PAPI, PerfSuite, Scalasca, Vampir and TAU," Full day tutorial at SC 2010, New Orleans, LA, 2010.
- (c) S. Shende, A. D. Malony, S. Moore, B. Wylie, "Productive Performance Engineering of Petascale Applications with POINT and VI-HPS," Full day tutorial at SC'2009.

1.2 Research objectives

The original MOGO objectives targeted the whole optimization at various levels of detail. The work at the University of Oregon has concentrated on being able to obtain more performance data at different levels. This includes GPU performance data and OS-level information, which is just now operational. An important research objective accomplished in this project is the assimilation of the different levels of performance data for optimization assessment.

In addition, the metrics used by Memphis to identify current NUMA performance problems are straightforwardly identified with manual investigation of the hardware counter data. As architectures grow more complex, additional data analysis tools and NUMA performance issues could be diagnosed automatically with the assistance of memory performance models generated by static analysis. These models, once generated, could then be compared to the data collected at runtime in order to identify these performance discrepancies.

2 Background: Original MOGO Approach

As originally proposed, the *Model-Oriented Global Optimization (MOGO)* project sought to develop a general framework where empirical performance data can be efficiently and accurately compared with **performance expectations** at various levels of abstraction. This information could then be used to automatically identify and remediate performance problems. MOGO was based on performance models derived from application knowledge, performance experiments, and symbolic analysis. The high-level MOGO architecture is shown in Figure 1.

2.1 Original Proposed Research

Given this background, the MOGO project is founded on four important claims regarding petascale performance analysis and optimization:

- Claim 1: Approaches based on scaling terascale performance measurement and analysis techniques are insufficient for petascale performance diagnosis and tuning.
- Claim 2: Optimization of petascale applications requires model-level knowledge of parallel algorithms, parallel computation, run-time execution, and parallel performance.

Figure 1: Original MOGO System Design.

- Claim 3: Performance of petascale applications and systems should be evaluated *in toto*, to understand the effects of performance interactions and identify opportunities for optimization based on whole-performance engineering.
- Claim 4: Adaptive petascale applications require performance monitoring, dynamic control, and tight coupling between the application and system at run time.

Guided by these claims, the goal of the project is to deliver the first petascale performance optimization framework based on modeling and global optimization methods. Our proposed research activities are integrative and build on our strengths in parallel performance modeling, robust performance measurement and analysis, scalable OS and run-time infrastructure, and parallel I/O systems. Specifically, we aim to:

- Specific Aim 1: Develop a petascale modeling methodology based on *performance expectations*, which incorporates knowledge of the application, computational algorithms, system properties, and execution mapping into the model.
- Specific Aim 2: Create an environment for *performance insight* based on knowledge engineering of performance problems, models, and expertise for performance remediation, augmented by powerful techniques for performance data mining, machine learning, and rule-based inference.
- Specific Aim 3: Automate identification of performance problems early in the data collection process in order to eliminate the *performance data deluge* and excessive perturbation.
- Specific Aim 4: Support adaptation of system software in response to run-time performance expectations.

2.2 Impact of Budget Reduction

MOGO was selected for funding by DOE, but the funded budget level was reduced to approximately 65% of the original request. This reduction was spread evenly across all three organizations.

Furthermore, we were forced to reduce the scope of work on the project by one third.

3 Research Accomplishments - Significant results

3.1 Memphis: a Performance Analysis Tool for Non-Uniform Memory Patterns

3.1.1 Motivation

Current projections call for each chip in an Exascale system to contain 100s to 1000s of processing cores. Already (10 cores/chip) memory limitations and performance considerations are forcing scientific application teams to consider multi-threading (i.e., OpenMP) as an alternative to "MPI-everywhere." At the same time, two recent trends in micro-processor hardware design are bringing Non-Uniform Memory Access (NUMA) to the node-level scale once associated with uniform memory access, or Symmetric Multi-Processing (SMP).

The first trend, demonstrated in Figure 2(a), is the movement towards on-chip memory controllers. While per-chip memory controllers remove contention for a shared bus, allowing for more scalable systems and increased bandwidth to *local* memory, satisfying *non-local* memory references requires trips across the chip-to-chip network.

A second trend, ever-increasing core counts, exacerbates the effect of the first by placing more pressure on shared resources, such as the memory controller and network interface. As depicted

Figure 2: Micro-processor trends.

in Figure 2(b), as the pressure increases, so too do the resources need to be increased, leading to NUMA *within* a socket.

Two performance problems are typically associated w/ NUMA:

1. Hot-spotting – due to poor initialization, memory not distributed across nodes.
2. Computation/Data-partition mismatch – memory distributed, but not appropriately.

In addition, NUMA can amplify small performance bugs, perhaps imperceptible in non-NUMA systems, and turn them into significant problems. An example is contention for locks and other shared variables: NUMA can significantly increase latency (and thus waiting time), increasing possibility of further contention.

While the problems associated with the trend towards NUMA burden programmers with yet more to worry about, there are also two pieces good news. First, a mature infrastructure already exists for handling NUMA from software level. NUMA-aware operating systems, compilers and runtime, based on years of experience with distributed shared memory platforms like SGI Origin/Altix. Second, programmers have new access to performance counters that help identify problems and their sources. NUMA performance problems are caused by references to remote data, the counters for which are naturally located in network interface. Since the network interface is on chip, the counters are easy to access and correlate with specific user code.

3.1.2 *Memphis*

Memphis is a performance tool that makes use of newly available sampling-based hardware performance monitoring extensions, along with the new performance counters described in the previous section, to enable programmers to pinpoint sources of memory system performance problems due to, or exacerbated by, NUMA.

Instruction-Based Sampling AMD introduced Instruction-Based Sampling (IBS) hardware performance monitoring extensions, similar to ProfileMe hardware found in the DEC Alpha 21264, with the Barcelona-generation Opteron. Like event-based sampling, instruction-based sampling is interrupt driven; however, interrupts are not due to counter overflow. Instead, HW periodically

interrupts, follows the next instruction through pipeline, keeping track of what happens to and because of the instruction and calling a handler upon instruction retirement.

IBS provides the following data useful for finding NUMA problems:

- Precise program counter of instruction
- Virtual address of data referenced by instruction
- Where the data came from: i.e., DRAM, another core’s cache
- Whether the agent was local or remote

Intel has since added similar functionality with their LoadLatency extensions to Precise Event-Based Sampling (PEBS) hardware counters. In both cases, the association of a remote memory access with the instruction and data address that caused the event, enables precise attribution of performance-degrading NUMA behavior to specific application code and *variables*.

Figure 3: *Memphis* runtime (a) and post-processing (b) components.

Memphis The *Memphis* tool uses IBS hardware to pinpoint NUMA problems at source, taking a data-centric approach: while sampling-based tools typically associate information with instructions, *Memphis* associates information with variables. The utility of *Memphis* is based on the following key insight: the source of NUMA problem is not necessarily where it is evidenced. For example, though the cause of a hot spot is variable initialization, problems are only evident at the variable’s use. Therefore, programmers want to know first, what variable is causing problems and, second, where (likely multiple sites).

As depicted in Figure 3 *Memphis* consists of three components:

1. Kernel module interface with IBS hardware
2. Library API to set ‘calipers’ and gather samples
3. Post-processing executable

The kernel module code is based on the *Oprofile* kernel module supplied by AMD for the *CodeAnalyst* tool. Most differences between the two modules are located in the interrupt handler:

- Filter – only interested in samples that went to Northbridge.
- User-level signaling mechanism – currently used to implement watch-point addresses.
- Per-core sample buckets – Oprofile puts samples from all threads in a single bucket.
- Fixed-sized buffers – no handler for overflow.

3.1.3 Recent Work

Since publication of the original paper describing *Memphis* [10], we have continued to extend the tool’s functionality. This section describes three recent extensions: mapping addresses to dynamically allocated variables, a port to Cray CNL [11], and work on an Eclipse-based GUI.

Allocation Instrumentation Tool Adds capability to map addresses to dynamically allocated variables. Based on a Tau tool, built on top of Program Database Toolkit from University of Oregon. Easily integrated into build process: extra step in the rule to compile F90 files in Makefile. At runtime, each dynamic allocation dumps variable-to-address-range mapping for use by post-processing tool. While our approach has several potential drawbacks – it adds overhead to each dynamic allocation and it requires access to source (i.e., cannot instrument libraries) – in practice we have found that the benefits significantly outweigh drawbacks.

Memphis on Cray Platforms Because Cray’s operating system, Compute Node Linux (CNL), is Linux-based, most components of *Memphis* work on Cray platforms without modification. The primary exception is the kernel module: several predefined kernel constants and functions are not contained in the CNL distribution, which required finding and hard-coding values into calls that set configuration registers. The kernel module port was further complicated by the black-box nature of CNL (since it is not open-source), necessitating the help of a patient Cray engineer (John Lewis) to perform first half of each iteration of the compile-install-test-modify loop.

Additionally, to make *Memphis* work on Cray platforms, we required a mechanism for making it available to jobs that want to use it. The goal of our runtime policy was to maximize the availability of Memphis for selected users, while minimizing impact of a bleeding-edge kernel module on others. The policy we arrived works as follows: the kernel module is always available on a single, dedicated node of the system. On system reboots the kernel module is installed on the dedicated node and a device entry created in `/dev`. Users that want to access Memphis have a ‘reservation’ on that node, realized in practice as a standing reservation in Moab the batch scheduler on XT systems.

While this policy means that only one node provides sample data, we have found that this is sufficient for our needs since intra-node performance is typically uniform across nodes.

Eclipse GUI Figure 4 demonstrates a new Eclipse interface to data collected by *Memphis*, improving on the original text-based user-interface. The post-processing tool outputs several files describing inefficiency points that can be cluttered and difficult to read. In the new interface, an Eclipse plugin reads the data text file and outputs an tree view of it, such that nodes in the tree can be expanded and collapsed. Additionally, when users click on the line number, the plugin brings up the source file and highlights the line where the inefficiency took place. Alternately, it brings up the assembly code for the executable file given by the user and the line that matches the address clicked on by the user would be highlighted.

Figure 4: Memphis in Eclipse.

	IBS Off, No Instrumentation	IBS On, Instrumented
Base	40.69	41.18
Mod1	36.29	36.63
Mod2	35.90	36.31

(a)

(b)

Figure 5: *Memphis* performance, measured in terms of overhead (a) and application (CESM climate code) performance improvement (b).

3.1.4 Memphis Evaluation

Figure 5 demonstrates two aspects of *Memphis* 'performance': runtime overhead and usefulness as measured by enabled application performance improvements.

Runtime Overhead. The table in Figure 5(a) shows that the runtime components of *Memphis* consistently incur only about 1% overhead, including instrumentation of most allocation statements. As described in the previous section, every allocation prints the variable name and address range

to a file, though in these runs we have removed instrumentation from allocation of communication buffers that were filling files with useless information.

Application Performance Improvements: CESM. The chart in Figure 5(b) describe the performance impact of *Memphis*-directed modifications to a single source file of CAM/HOMME code. On ORNL’s Jaguar, overall CAM throughput improves by close to 50% for large-scale runs. Since the chart shows only single-NUMA-node performance improvements, we note that 12-thread performance (two NUMA nodes) of the modified version is comparable to 6-thread performance, whereas 12-thread performance of the original code was significantly worse than 6-thread.

3.1.5 Ongoing Work

We have recently noticed a problem with IBS: secondary references to cache blocks serviced from the Northbridge are marked as L1 hits – rather than as L3 Hits, or local or remote DRAM references – albeit with extremely long latency. This behavior can lead to false negatives, where an apparent ‘fix’, indicated by lower remote reference counts, does not improve performance as expected. In response, we are currently exploring modifications to filtering mechanism in kernel module: let through long-latency L1 hits. While latency can have other causes, for example, resource contention, our hope is that ‘average case’ behavior will dominate.

3.1.6 Conclusion

Though NUMA is already a problem, and it will only get worse, there is hope. *Memphis* is a toolset that uses sampling-based hardware performance monitoring extensions to pinpoint the sources of memory performance problems. We have used *Memphis* to find and fix significant problems in several large-scale production applications, and *Memphis* is now available on Cray platforms.

3.2 Performance Analysis using TAU

With the growing emergence of heterogeneous systems as a viable petascale computing approach, the University of Oregon (UO) directed attention in the first year to measuring application accelerator performance as part of a comprehensive MOGO strategy for whole application performance optimization in a heterogeneous environment. For this purpose, UO focused its activities towards GPU accelerator performance measurement and analysis. The first target was PGI’s Accelerator compiler where we developed a library wrapping technology to interface with the PGI Accelerator runtime system [13]. The second and more difficult target was CUDA. We developed two versions of the *TAUcuda* [6, 7, 9] prototype and integrated it in the TAU framework. Our results in each area are described in UO’s progress report for Years 1, 2, and 3. The highlights are reviewed below.

3.2.1 PGI Accelerator Performance Measurement

A fundamental challenge of understanding parallel systems with accelerators is to observe and associate GPU performance information with the application code running on the CPUs. This is complicated by the fact that GPU operation can occur asynchronously from CPU activity. A performance model must reflect this concurrent, heterogeneous computation. Furthermore, mechanisms for CPU performance measurement do not directly translate to the GPU, and support within the CUDA system is necessary to observe GPU operation.

Our initial work to develop performance tools support for GPU performance measurement was with the PGI Accelerator compiler. PGI’s Accelerator compiler can parallelize loops in Fortran and C for acceleration on NVIDIA GPU using a directive-based GPU programming abstraction. It implements a runtime layer that interfaces compiled code with the CUDA execution environment accessed using the NVIDIA CUDA Software Development Kit (SDK). Our work with the PGI Ac-

celerator compiler on Linux demonstrated TAU's library wrapping capabilities. Given the runtime library interface, we can track all operations of the runtime system as seen from the host processor and associate the calls with source information where the GPU acceleration is taking places. This information includes the routine name, source file name, kernel source line number, variable names used in memory upload/download operations, and grid sizes.

The library wrapping approach used a “performance interface interposition” mechanism by appending a “_p” to runtime calls of interest to measure. This is shown in general below:

```
void __pgi_cu_module_p(void *image);
void __pgi_cu_module(void *image) {
    TAU_PROFILE('‘__pgi_cu_module’', "TAU_DEFAULT);
    __pgi_cu_module_p(image);
}
```

Calls instrumented in this way included calls for initialization, launching kernels, and upload and download. Our PGI Accelerator performance measurement tool was tested with several applications. Figure 6 shows a TAU profile display for a matrix multiplication example run with matrices of size 3000x3000. The output clearly shows the instrumented GPI runtime routines and the information to relate the calls to the source.

Figure 6: Performance measurement of PGI Accelerator matrix multiplication example.

Because the PGI Accelerator runtime system interacted with the CUDA environment in a synchronous manner, our measurement approach worked well. In general, programming of GPU devices is more complex, with multiple GPU devices, multiple streams execute concurrently, overlapped data transfer and kernel execution, and so on. More recent versions of the PGI Accelerator compiler takes this into account and we are intending to provide support for these new developments.

Figure 7: TAUcuda architecture design with TAU and TAUcuda events highlighted.

3.2.2 TAUcuda

GPU accelerators can be programmed in different ways and the CPU-GPU execution model can be more flexible. During the first year, Oregon developed a package called *TAUcuda* that enables CUDA performance to be measured and integrated with the TAU performance system. Two versions of TAUcuda were realized. An initial version was developed based on the CUDA *Event* interface [9]. Defined as part of the CUDA specification, the Event interface can be used to obtain performance data for a particular GPU stream’s execution, including each kernel’s precise timing. The performance data is measured by the CUDA runtime system and returned through the CUDA performance measurement library we developed. While we were able to demonstrate TAUcuda (v1.0) on multi-GPU applications and the HMPP GPU programming environment, two problems existed. First, the approach required the CUDA application to be instrumented with calls to the Event interface. Second, it only provided a runtime library view of performance. Although detailed information would be returned about kernel execution, it could only be interpreted in a high level application context. No access to driver-level operation was available, making precise memory transfers and kernel timing impossible to observe.

The second version of TAUcuda developed in the second year relied on an experimental, non-production version of the NVIDIA CUDA driver library (R190.36) that implements a callback API specifically for performance tool use. TAUcuda (v2.0) builds on the NVIDIA driver library and callback interface to enable the capture of CUDA performance events in profile and trace forms. There is a tighter integration of TAUcuda with the TAU performance system to allow events from the CUDA system and GPU operation to be measured and analyzed in the context of the parallel application as a whole using TAU’s sophisticated analysis tools. Figure 7 shows the architecture of the TAUcuda design and its integration with the CUDA system and TAU.

TAUcuda measurement can be used to collect CPU-GPU execution profiles and traces. In profiling mode, TAUcuda will create a profile event for any unique tuple combination defined by

Figure 8: Mean profile of a four-process execution of Linpack with GPU acceleration.

(event name, TAU context, GPU device, stream ID). Every encounter of the same tuple reuses the profile event for that tuple. Multiple metrics profile events can be calculated, including *elapsed time*, *memory transfer size*, *shared memory usage*, *registers per thread*, and *core occupancy*. TAUcuda writes out the profiles in TAU profile format when the CUDA application exits. A separate profile is created with the different profile metrics for every GPU device and stream on which GPU operation occurred. Implementing tracing in TAUcuda is more complex and the details can be found in our ICS 2010 paper [6]. The main difficulties are synchronizing the CPU and GPU timestamps, and resolving memory transfers. The goal is to show all CPU-GPU events integrated together.

Using the TAUcuda environment described briefly above (more in detail in the Year 1 report) for CUDA performance measurements on Linux systems with NVIDIA GPUs, we conducted several experiments to assess the TAUcuda profiling and tracing capabilities. The only system requirement was that the NVIDIA R190.36 Linux CUDA driver library be installed. The experiments ranged from example programs distributed with the NVIDIA CUDA SDK to scalable scientific applications run on multi-GPU clusters.

To demonstrate TAUcuda’s ability to measure the performance of scalable heterogeneous parallel applications, we ran NVIDIA’s GPU-accelerated Linpack benchmark [8]. The ParaProf profile for a four-process Linpack run on Mist is shown in Figure 8. This shows mean execution time values across the processes for all events. The corresponding Jumpshot trace is shown in Figure 9. The profile shows TAU source instrumentation of the Linpack routines and MPI communication events, in addition to the CUDA driver, CUDA runtime, and GPU and memory transfer events. The trace display emphasizes the temporal performance behavior of the Linpack execution. The yellow lines show the MPI message communication occurring between the CPU nodes. The white lines highlight the CUDA message transfers taking place.

The SHOC benchmarks provided another opportunity to demonstrate TAUcuda. We selected the SHOC Stencil2D benchmark since it contained an interesting combination of MPI, memory transfer, kernel execution, and other events to measure. Stencil2D is a standard two-dimensional nine-point stencil calculation. Shown in Figure 10 is a trace of the application running with four CPUs and four GPUs. Only the accelerated portion is shown. There is an initial data transfer, followed by a setup computation, before the main iterations proceed.

The TAUcuda implementation in Year 1 was based on an experimental driver library from NVIDIA. This library was not intended for production release. Instead, NVIDIA began development of an interface for performance tools and we worked with them in the period to re-engineer

Figure 9: Trace of a four-process execution of Linpack with GPU acceleration.

Figure 10: SHOC Stencil2D benchmark, 512 iteration on four cpus and four Tesla GPUs. The pink bars represent GPU execution and the lines show memory transfers from the Device to Host.

our tools. Leveraging the TAU wrapper generator effort (discussed below), we were able to extend the GPU measurement support to OpenCL codes.

3.2.3 Library Wrapping

The TAUcuda design approach required us to take on the challenge of wrapping a library for instrumentation where we only had access to header files. Here, we extended our wrapping tool, `tau_wrap` to instrument the CUDA runtime library. Previously, we had designed this tool to support both pre-processor based wrapping as well as wrapping calls using pre-loading of shared objects that define a given interface. While pre-processor based wrapping merely redirects calls from a package's interface to an alternative implementation, pre-loading is a more powerful mechanism to

replace all references to a given package with a wrapper library. The former requires recompilation of the caller’s code (written in C or C++). The latter can also target other languages such as Fortran and requires no changes to the application source code or the object code.

We implemented library wrapping with TAU’s source analysis technology, Program Database Toolkit (PDT). The parsed representation of a header file (e.g., `cuda.h`) contains a wealth of information about the interface. It consists of a series of vectors containing hierarchical source information such as the list of files, headers, declarations, types, structs and enumerations, return types of routines, types of arguments, and so on. For CUDA, we needed to add support to handle variable arguments in the routine by iterating over a list of pointers to the next argument. We further extended the wrapper functionality to observe the arguments of the routines. This was important to extract information about the transfer of data between the host and the GPU device.

Additionally, we developed support for interposition of wrapped libraries by preloading. Our updates to the `tau_exec` tool allows a `-loadlib` argument to pre-load the wrapped shared objects generated by `tau_wrap` along with other wrapped modules developed previously (such as I/O, memory, and communication tracking modules). The `tau_exec` tool operates on an uninstrumented or instrumented binary preloading the given set of shared objects in the address space of the application prior to execution. This work allowed us to take binary CUDA programs and dynamically link TAUcuda wrapped libraries for performance measurement

In Year 2, we added a capability to static library wrapping that enables creation of a wrapper in cases where the source of an external library cannot be modified. This may be necessary for libraries where the source is not available or for when the library is cumbersome to re-build. The approach uses TAU’s PDT to read header files of the original library and generate a new library wrapper header file with preprocessor `DEFINE` macros to change routine names to special TAU routine names. A new wrapped library is created with these instrumented TAU routines, which then calls the original library. This approach can be used for any library with a provided header. We updated our instrumentation of the PGI Accelerator runtime library with this new capability, and was applied it for I/O library wrapping in the Structural Simulation Toolkit (SST) at Sandia National Laboratory where parameters from POSIX I/O calls were tracked using a new I/O wrapping mechanism implemented in TAU.

3.2.4 Other Efforts

We also initiated three new efforts for MOGO in the second year. First, we developed online support in TAU to allow for processing of parallel profile data using scalable monitoring infrastructure [5]. This was implemented using both MRNet and MPI. The primary purpose was to allow more efficient statistics calculations at the end of execution, to perform event unification, and to reduce the performance data output and write a single structured output file. Associated with this work is our effort to define a new scalable parallel profile format for TAU.

For large numbers of threads of execution, the loading of parallel profiles in the TAU ParaProf analysis tool was slow. We are now in the process of implementing a more scalable profile format for Paraprof which makes no changes to the overall size of the performance data set. Instead, XML markers break the profile data into smaller and more manageable frames. The data representing more significant events will be held in frames which ParaProf will load and display initially. The new format, along with changes to ParaProf, will allow the analyst to quickly view and work on analyzing the profile of a large dataset early while ParaProf continues to load less significant data in the background. In addition, the format will make it easier to manage the memory footprint of ParaProf. The format is also designed in a way to allow older versions of ParaProf to load the most significant data frames for analysis.

The third new effort is the development of an approach for 3D performance visualization support

Figure 11: Choice of process allocation of S3D on the IBM BG/P has performance effects that can be visualized using MPI topology information.

in ParaProf. We have enabled visual layout and user interface design to be specified in ParaProf, from which a performance visualization is constructed. In particular, the work allows for the interpretation of MPI and system topology information in the layout of parallel performance data. The TAU measurement system was updated to capture topological metadata in a parallel profile for the IBM BG/P and Cray XE6 platforms. The approach has been tested with the Sweep3D, S3D, and GCRM applications. Figure 11 shows an example of the effects of different process allocations (default versus random) on the IBM BG/P of the S3D application. The visualization highlights the better balance of MPI_Barrier execution time. A paper is in submission to the PROPER workshop to be held in association with the Euro-Par conference [14].

3.2.5 Year 3

The University of Oregon was funded through September 14, 2012. During Year 3 of the project, we targeted three areas: kernel performance instrumentation and measurement, performance visualization enhancements, and re-engineering of the TAU performance database and data mining. With the awarding of the Vancouver project (Grant #: DE-SC0005360), the work on GPU performance measurement has now transitioned to that effort. However, the MOGO project developed how GPU performance data is integrated in the profile format. We released TAU with support for CUDA 4.1 and later CUDA 5.0 support for performance instrumentation of GPU kernels using the CUPTI API.

In the third year, we added performance instrumentation and measurement capabilities in the Linux kernel to capture OS performance data and associate this information with application

profiles. We used the Linux *tracepoints* for this effort and created a functional implementation. We released the KTAU version 3.0 software from the TAU website. This version of KTAU does not require patching the kernel source code. Rather, it uses a loadable kernel module using tracepoints to provide support for kernel performance instrumentation and measurement. We also evaluated the *kprobes* tool for kernel instrumentation. We also developed tools for merging kernel and user-space profiles. In addition to the source distribution of KTAU, a separate HPC Linux OVA file was created with built-in support for kernel instrumentation. This virtual image can be loaded in Virtual Box and simplifies the demonstration of KTAU in a complete Linux package based on a 32 bit Fedora Core OS. This updated kernel instrumentation package was released along with TAU v2.21.3 in July 2012.

We implemented a prototype of a new scalable parallel profile format for improving the efficiency of working with large profile files in ParaProf and other tools. In the area of performance database and data mining, a TAUdb prototype was created to demonstrate an alternative performance data schema that is better suited for GPU performance data and provided better hierarchical composition of trials. In particular, we added support to store and analyze GPU performance information for use in heterogeneous performance analysis. The database upload tool *perfdmf.loadtrial* was extended to support alternate mean definitions to support hybrid CPU-GPU measurement data where some kernels only execute on a subset of threads. We also re-engineered the TAU performance database and data mining tools to make them more scalable and robust.

For our performance visualization activities, we completed the prototype implementation of the 3D topology display in ParaProf and released it in the TAU version 2.21.4 distribution in September 2012. This topology display supports the IBM BlueGene/Q and Cray XE6 torus topology visualization features. This modular display, accessible through the Jython interface provides visualization capabilities that allow it to be integrated with other MOGO software.

UO will re-engineer the TAU performance database and data mining tools in the next two years to make them more scalable and robust. In particular, we will add support to store and analyze GPU performance information for use in heterogeneous performance analysis.

3.3 System Tuning Parameters

Applications live longer than machines they are originally built and optimized for. What we get as a result of this are, applications that are not appropriately fine tuned for all architectures. This situation is caused and mostly worsened due to the fact that performance diagnosis is neither easy nor very intuitive. Hence, there is a dire need for developing models to measure, diagnose and analyse complex HPC software. Our work at ANL was aimed at studying the nature of parameters that affect application performance. Particularly, we look at tuning capabilities available at the operating system level and lower layers of the software stack.

We attempt to study the parameters that are available for tweaking at a system software level. Although such studies that determined application behavior under particular environments might have been parts of other works in different contexts, this work is unique in that it aimed to:

- Be a central repository of various such results allowing one to look up application characteristics under different system environments
- Be an unbiased study purely trying to show how applications behave differently and not necessarily better or worse with respect to any particular software environment.

At ANL, our study was mostly conducted using the ZeptoOS operating system. ZeptoOS is a port of the Linux kernel for the Blue Gene/P [4] supercomputer. For a part of our study we also used Fusion Linux cluster at ANL.

At Argonne, the Blue Gene/P supercomputer uses the *Cobalt* [2] resource manager. Cobalt allows to run jobs using custom operating system kernel profiles. Hence we were able to run our applications for parameter studies using a variety of kernel configurations. An environment like Cobalt provides an unprecedented level of adaptation by providing flexibility at all levels from changing OS kernels, custom OS boot parameters to partition wide setting of application environment variables.

3.3.1 Paged Memory versus Big Memory

Linux is a virtual memory operating system. This memory management scheme provides processes isolation from each other and faster copy-on-write semantics among other things. On most modern processors, user programs run inside the virtual address space. Processes are not allowed to directly access physical addresses. The operating system creates page table entries (PTE) in memory that provide mappings from virtual to physical memory. This causes a lot of overhead. In order to overcome this problem processors implement Translation Lookaside Buffer (TLB) which are temporary caches that provide fast page translation. When a certain entry is not found in the TLB, it incurs a TLB miss which causes a new page to be brought in. When the operating system switches a process context, these entries need to be flushed.

ZeptoOS provides applications with special memory regions that are covered by larger pages; we call these regions *Big Memory* [15, 16]. Another known approach is to use *hugetlbfs* [3] regions. This scheme which uses 2 to 4 MB pages which only reduces the probability of TLB misses. Big Memory on the other hand uses pages so large that the TLB misses are eliminated while the process is scheduled in. The idea is to reserve a dedicated physical memory region at boot time and statically install TLB entries at page fault time to cover the Big Memory region [17]. The TLB entries remain in place until the process is scheduled out again. Also, the Linux exec handler transparently prepares Big Memory when a special binary is about to be loaded.

3.3.2 Native Torus versus TCP-Torus

On Blue Gene/P the default communication takes place over the default Torus-DMA. In order to emulate a cluster like system on Blue Gene/P we implemented the Torus TCP driver. Once this was done we were able to launch MPI processes on the machine like a regular cluster. Figure 13 shows that when processes were bound to individual cores and executed, the performance was comparable to that of the application on ZeptoOS bigmem with the same driver and slightly less than when run with bigmem on native network. It has been shown in our previous work [16, 17] that ZeptoOS performs within 1% of Blue Gene/P's Compute Node Kernel [4, 12].

3.3.3 Binding versus no Binding

The Blue Gene/P processor has four cores. Normally load balancing features inside the Linux kernel causes processes to migrate. Our goal was to study application behavior when processes were pinned to cores versus when the kernel was allowed to migrate them to different cores for load balancing.

We only show a few of the results to demonstrate the nature of our study. Figure 13 shows a comparison of benchmark performance when the NAS Parallel Benchmark [1] EP suite was run on ZeptoOS. The benchmark was run under four different configurations:

- **No-Binding:** Processes were launched using *mpiexec* with no special options on ZeptoOS using Torus-TCP driver and Linux paged memory.
- **Binding:** Processes were launched using *mpiexec* with *-binding cpu:cores* option on ZeptoOS using Torus-TCP driver and Linux paged memory.

Figure 12: Fusion, Suite: EP, Class: D

Figure 13: ZeptoOS Suite: EP Class: D

- **Bigmem-TCP:** Four ZCB processes were launched per node SMP mode using Bigmem and Torus-TCP driver.
- **Bigmem-Native:** Four ZCB processes were launched per node VN mode using Bigmem and native torus driver.

3.3.4 Fusion Studies

Fusion is a 320 node (8 cores per node) Intel Nahelam based Linux cluster with a InfiniBand QDR 4GB/s per link, per direction interconnect at *Laboratory Computing Resource Center* at Argonne. Since we wanted to validate our results on other platforms, we used Fusion for further studies.

Results on Fusion were significantly different when compared with the results on Blue Gene/P platform. We see in Figure 12 that for *binding* vs. *non-binding* cases, there is no noticeable difference in performance which is a stark contrast from our observations on ZeptoOS.

3.3.5 Conclusion

We observed that the same application suite on different platforms reacted differently to the same set of parameters.

The studies that we conducted at ANL help us gather an understanding about application characteristics and their performance requirements. This study is by no means exhaustive but helps us gain significant insight about settings for a select set of parameters. Data collected from the study can later be used to build mechanisms in which applications with certain observed characteristics can be launched with the most optimal set of parameters automatically.

References

- [1] D. Bailey et al. The NAS parallel benchmarks. *International Journal of High Performance Computing Applications*, 5(3):63–73, 1991.
- [2] N. Desai, R. Bradshaw, E. Lusk, and R. Butler. Component-based cluster systems software architecture a case study. In *Proceedings of the 2004 IEEE International Conference on Cluster Computing*, pages 319–326, Washington, DC, USA, 2004. IEEE Computer Society.
- [3] D. Gibson and A. Litke. libhugetlbfs. <http://sourceforge.net/projects/libhugetlbfs>.
- [4] I. journal of Research and D. staff. Overview of the ibm blue gene/p project. *IBM J. Res. Dev.*, 52:199–220, January 2008.
- [5] C. Lee, A. Malony, and A. Morris. TAUmon: Scalable Online Performance Data Analysis in TAU. In *Workshop on Productivity and Performance Tools for HPC Application Development (PROPER 2010)*, Aug. 2010.
- [6] A. Malony, S. Mayanglambam, S. Biersdorff, and W. Spear. An experimental approach to performance measurement of heterogeneous parallel applications using cuda. In *International Conference on Supercomputing*, June 2010.
- [7] A. Malony, S. Mayanglambam, L. Morin, M. Sottile, S. Bihan, S. Shende, and F. Bodin. Performance tool integration in a gpu programming environment: Experiences with tau and hmpp. In *Parallel Computing: From Multicores and GPUs to Petascale*, volume 19 of *Advances in Parallel Computing*, pages 685–692. Sept. 2009. International Conference on Parallel Computing (ParCo), Mini-Symposium on Parallel Programming Tools for Multi-core Architectures.
- [8] F. Massimilian. Accelerating linpack with cuda on heterogeneous clusters. In *Workshop on General Purpose Processing on Graphics Processing Units (GPGPU)*, pages 46–51, Mar. 2009.
- [9] S. Mayanglambam, A. Malony, and M. Sottile. Performance measurement of applications with gpu acceleration using cuda. In *Parallel Computing: From Multicores and GPUs to Petascale*, volume 19 of *Advances in Parallel Computing*, pages 341–348, Sept. 2009.
- [10] C. McCurdy and J. Vetter. Memphis: Finding and fixing numa-related performance problems on multi-core platforms. In *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2010.
- [11] C. McCurdy, J. Vetter, P. Worley, and D. Maxwell. Memphis on an xt5: Pinpointing memory performance problems on cray platforms. In *Cray User Group (CUG)*, May 2011.
- [12] 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*, 33:9–14, December 2005.
- [13] S. Shende, A. Malony, and A. Morris. Improving the Scalability of Performance Evaluation Tools. In *PARA 2010: State of the Art in Scientific and Parallel Computing*, June 2010.
- [14] W. Spear, A. Malony, C. Lee, S. Biersdorff, and S. Shende. An approach to creating performance visualizations in a parallel profile analysis tool. In *Workshop on Productivity and Performance Tools for HPC Application Development (PROPER 2011)*, Aug. 2011.

- [15] K. Yoshii, K. Iskra, H. Naik, P. Beckman, and P. C. Broekema. Characterizing the performance of ‘Big Memory’ on blue gene linux. In *Proceedings of the 2009 International Conference on Parallel Processing Workshops*, ICPPW ’09, pages 65–72, Washington, DC, USA, 2009. IEEE Computer Society.
- [16] K. Yoshii, K. Iskra, H. Naik, P. Beckman, and P. C. Broekema. Performance and scalability evaluation of ‘Big Memory’ on blue gene linux. *International Journal of High Performance Computing Applications*, May 2010. <http://hpc.sagepub.com/content/early/2010/05/11/1094342010369116>.
- [17] K. Yoshii, H. Naik, C. Yu, and P. Beckman. Extending and benchmarking the ”big memory” implementation on blue gene/p linux. In *Proceedings of the 1st International Workshop on Runtime and Operating Systems for Supercomputers*, ROSS ’11, pages 65–72, New York, NY, USA, 2011. ACM.