Improving the Scalability of Performance Evaluation Tools

Sameer Suresh Shende, Allen D. Malony, and Alan Morris

Performance Research Laboratory
Department of Computer and Information Science
University of Oregon, Eugene, OR, USA
{sameer,malony,amorris}@cs.uoregon.edu

Abstract. Performance evaluation tools play an important role in helping understand application performance, diagnose performance problems and guide tuning decisions on modern HPC systems. Tools to observe parallel performance must evolve to keep pace with the ever-increasing complexity of these systems. In this paper, we describe our experience in building novel tools and techniques in the TAU Performance System® to observe application performance effectively and efficiently at scale. It describes the extensions to TAU to contend with large data volumes associated with increasing core counts. These changes include new instrumentation choices, efficient handling of disk I/O operations in the measurement layer, and strategies for visualization of performance data at scale in TAU's analysis layer, among others. We also describe some techniques that allow us to fully characterize the performance of applications running on hundreds of thousands of cores.

Keywords: Measurement, instrumentation, analysis, performance tools.

1 Introduction

Tools for parallel performance measurement and analysis are important for evaluating the effectiveness of applications on parallel systems and investigating opportunities for optimization. Because they executes as part of the parallel program and process performance data that reflects parallel behavior, measurement and analysis techniques must evolve in their capabilities to address the complexity demands of high-end computing. Scaling in the degree of parallelism is one of the key driving requirements for next-generation applications. To address scaling concerns, performance tools can not continue with traditional techniques without considering the impacts of measurement intrusion, increased performance data size, data analysis complexity, and presentation of performance results.

This paper discusses approaches for improving the scalability of the TAU Performance System® instrumentation, measurement, and analysis tools. Our perspective looks generally at the maximizing performance evaluation return to the tool user. This starts with improving the instrumentation techniques to

K. Jónasson (Ed.): PARA 2010, Part II, LNCS 7134, pp. 441–451, 2012.
© Springer-Verlag Berlin Heidelberg 2012

select key events interest and avoid the problem of blindly generating a lot of low value performance data. Section §2 presents a few of the new techniques in TAU to support more flexible and intelligent instrumentation. Performance measurement presents a scaling challenge for tools as it places hard requirements on overhead and efficiency. Section §3 describes TAU's new measurement capabilities for addressing scale. Scalable performance analysis deals mainly with concerns of reducing large performance data into meaningful forms for the user. Recent additions to TAU performance analysis are discussed in Section §4. The paper concludes with thoughts towards future extreme-scale parallel machines.

2 Instrumentation

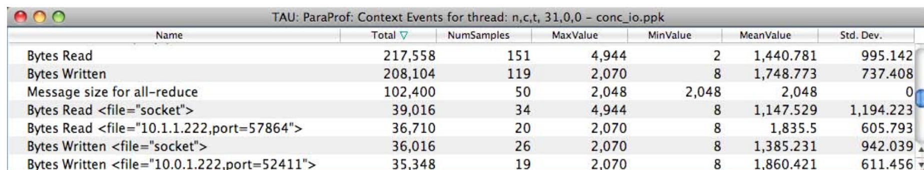
2.1 Source Instrumentation

For probe-based performance measurement systems such as TAU, instrumentation is the starting point for thinking about scalability because it is where decisions are made about what to observe. It is also where automation becomes important for tool usability. TAU has traditionally relied on a source-to-source translation tool to instrument the source code. Based on the PDT (Program Database Toolkit) [6] static analysis system, the `tau_instrumentor` [4] tool can insert instrumentation for routines, outer loops, memory, phases, and I/O operations in the source code. While source code instrumentation provides a robust and a portable mechanism for instrumentation, it does require re-compiling the application to insert the probes. While not directly affected by scaling, source instrumentation can become somewhat cumbersome in optimizing instrumentation for efficient measurement.

In the past, TAU addressed the need to re-instrument the source code by supporting runtime instrumentation (via the `tau_run` command) using binary editing capabilities of the DyninstAPI [2] package. However, dynamic instrumentation requires runtime support to be efficient at high levels of parallelism since every executable image would need to be modified. Techniques have been developed in ParaDyn to use a multicast reduction network [9] for dynamic instrumentation control, as well as in TAU to use a startup shell script that is deployed on every MPI process and then instruments and spawns an executable image prior to execution [10,7].

2.2 Binary Instrumentation

To address both usability and scalability issues, we implemented binary instrumentation in TAU using re-writing capabilities of DyninstAPI. This allows us to pre-instrument an executable instead of sending an uninstrumented executable to a large number of cores just to instrument it there. By re-writing the executable code using binary editing, probes can be inserted at routine boundaries pre-execution, saving valuable computing resources associated with spawning a DyninstAPI-based instrumentor on each node to instrument the application. The approach improves the startup time and simplifies the usage of TAU as no changes are introduced to the application source code or the build system.


TAU: ParaProf: Context Events for thread: n,c,t, 31,0,0 - conc_io.ppk

Name	Total ▽	NumSamples	MaxValue	MinValue	MeanValue	Std. Dev.
Bytes Read	217,558	151	4,944	2	1,440.781	995.142
Bytes Written	208,104	119	2,070	8	1,748.773	737.408
Message size for all-reduce	102,400	50	2,048	2,048	2,048	0
Bytes Read <file="socket">	39,016	34	4,944	8	1,147.529	1,194.223
Bytes Read <file="10.1.1.222,port=57864">	36,710	20	2,070	8	1,835.5	605.793
Bytes Written <file="socket">	36,016	26	2,070	8	1,385.231	942.039
Bytes Written <file="10.0.1.222,port=52411">	35,348	19	2,070	8	1,860.421	611.456

Fig. 1. Using a POSIX I/O interposition library, tau_exec shows the volume of I/O in an uninstrumented application

2.3 Instrumentation via Library Wrapping

While this is a solution for binary executables, shared libraries (dynamic shared objects or DSOs) used by the application cannot be re-written at present. If the source code for the DSO is unavailable, we are left with a hole in performance observation. To enable instrumentation of DSOs, we created a new tool, `tau_wrap`, to automate the generation of wrapper interposition libraries. It takes as input the PDT-parsed representation of the interface of a library (typically provided by a header file) and the name of the runtime library to be instrumented. The `tau_wrap` tool then automatically generates a wrapper interposition library by creating the source code and the build system for compiling the instrumented library. Each wrapped routine first calls the TAU measurement system and then invokes the DSO routine with the original arguments. The wrapper library may be preloaded in the address space of the application using the `tau_exec` tool that also supports tracking I/O, memory and communication operations[12]. Preloading of instrumented libraries is now supported on the IBM BG/P and Cray XT5/XE6 architectures.

The ability to enable multiple levels of instrumentation in TAU (as well as runtime measurement options) gives the user powerful control over performance observation. Figure 1 shows how the volume of read and write I/O operations can be tracked by TAU using library-level preloading of the POSIX I/O library in *tau_exec*. If callpath profiling is enabled in TAU measurement, a complete summary of all operations on individual files, sockets, or pipes along a program's callpath can be generated, as shown in Figure 2. Here we have instrumented MPI operations (through the standard MPI wrapper interposition approach). In doing so, we can now see how MPI routines like `MPI_Allreduce` invoke low-level communications functions, typically unobserved in other performance tools.

The linker is another avenue for re-directing calls for a given routine with a wrapped counterpart. For instance, using the GNU ld `--wrap routine_name` commandline flag, we can surround a routine with TAU instrumentation. However, this approach requires each wrapped routine to be specified on the link line. While onerous, this could be automated (and TAU does), but one may exceed the system limits for the length of the command line if a lot of routines are desired. Using a combination of wrapped instrumentation libraries with re-linked or re-written binaries provides complete coverage of application and system library routines without access to the source code of the application.

TAU: ParaProf: Context Events for thread: n,c,t, 31,0,0 – conc_io.ppk

Name	Total	NumSamples	MaxValue	MinValue	MeanValue	Std. Dev.
▾ .TAU application						
▾ int main(int, char **) C [{conc_allred.c} {19,1}-{74,1}]						
▸ MPI_Finalize()						
▸ MPI_Comm_split()						
▾ MPI_Allreduce()						
▸ recv()						
▸ send()						
▾ writev()						
Write Bandwidth (MB/s) <file="10.0.1.222,port=52411">		17	295.714	188.182	263.055	32.117
Bytes Written	207,000	100	2,070	2,070	2,070	0
Bytes Written <file="172.17.0.223,port=34273">	35,190	17	2,070	2,070	2,070	0
Write Bandwidth (MB/s) <file="172.17.0.222,port=42187">		17	295.714	172.5	224.456	32.332
Bytes Written <file="socket">	35,190	17	2,070	2,070	2,070	0
Write Bandwidth (MB/s) <file="172.17.0.223,port=34273">		17	258.75	147.857	226.091	29.554
Bytes Written <file="10.1.1.223,port=47660">	33,120	16	2,070	2,070	2,070	0
Bytes Written <file="10.0.1.222,port=52411">	35,190	17	2,070	2,070	2,070	0
Write Bandwidth (MB/s)		100	345	147.857	247.706	39.179
Write Bandwidth (MB/s) <file="10.1.1.222,port=57864">		16	295.714	207	256.491	29.838
Bytes Written <file="172.17.0.222,port=42187">	35,190	17	2,070	2,070	2,070	0
Bytes Written <file="10.1.1.222,port=57864">	33,120	16	2,070	2,070	2,070	0
Write Bandwidth (MB/s) <file="socket">		17	295.714	207	264.983	32.635
Write Bandwidth (MB/s) <file="10.1.1.223,port=47660">		16	345	172.5	251.922	51.998
▸ readv()						
▸ MPI_Init()						
▸ MPI_Bcast()						
▸ MPI_Barrier()						
Bytes Read	217,558	151	4,944	2	1,440.781	995.142
Bytes Read <file="/proc/net/if_inet6">	84	4	21	21	21	0
Bytes Read <file="/sys/devices/system/cpu/cpu0/topology/core_id">	2	1	2	2	2	0
Bytes Read <file="/sys/devices/system/cpu/cpu0/topology/physical_package_id">	2	1	2	2	2	0

Fig. 2. I/O statistics along a calling path reveal the internal workings of MPI library showing the extent of data transfers for each socket and file accessed by the application

2.4 Source and Compiler Instrumentation for Shared Libraries

When the source code is available for instrumentation, direct source instrumentation of static or shared libraries can be done automatically using TAU's compiler scripts (`tau_cxx.sh`, `tau_cc.sh`, and `tau_f90.sh`). The purpose of these scripts is to replace host compilers in the build system without disrupting any of the rest of the build process. TAU also supports compiler-based instrumentation where the compiler emits instrumentation code directly while creating an object file. This is supported for IBM, Intel, PGI, GNU, and Pathscale compilers at present. Source-based instrumentation involves deciphering and injecting routine names as parameters to timer calls in a copy of the original source code. While shared object instrumentation is relatively easy to implement in source-based instrumentation, it poses some unique challenges in identifying routine names for compiler-based instrumentation. Compiler-based instrumentation in statically linked code is easier to implement because the address of routines does not change during execution and is the same across all executing contexts. The address may be mapped to a name using BFD routines at any point during the execution (notably at the end of the execution).

On the other hand, dynamic shared objects, by their very nature, load position-independent object code at addresses that are assigned from an offset using a runtime loader. The same routine may be loaded at a different address in different executing contexts (ranks). Also, as the application executes, different shared objects represented by Python modules may be loaded and unloaded, and the map that correlates addresses to the routine names changes during the execution of the application. This address map (typically stored in the `/proc` file system

under Linux) cannot be queried at the end of the execution as the addresses may be re-used and shift as different shared objects are brought in and out of the executing process.

To handle instrumentation of shared objects, we examine the address ranges for the different routines after loading a shared object and determine the mapping of routines names and their addresses, dynamically in each context. This simplifies object code instrumentation in dynamic shared objects and we need only store these address mappings for shared objects that are loaded during execution. During execution, compiler-based instrumentation generates events and calls the measurement library. Events from C and Fortran languages typically map directly to their routine names. C++ events need an additional demangling step. Events from multiple layers co-exist in the executing context and performance data is generated by each context separately.

Providing robust support for selecting events to observe is important for giving optimal visibility of performance. TAU integrates several instrumentation approaches in a cohesive manner allowing a user to slice and examine the performance across multiple application layers at an arbitrary level of detail. By providing access to instrumentation hooks at multiple levels of program transformation, the user can refine the focus of instrumentation to just the relevant part while reducing the overhead by not instrumenting application constructs that may not be pertinent to a given performance experiment, thereby reducing the volume of the performance data generated.

3 Measurement

The scaling of parallel performance measurement must meet critical requirements. Most importantly, it must impact application's performance a little as possible. However the choice of what and how to measure is not that simple. Every performance measurement system will intrude on the execution. It is important then to optimize the balance between the need for performance data and the cost of obtaining it. Our goal in TAU is to provide flexible support for making optimal choices concerning measurement type and degree.

3.1 Parallel Profiling

TAU provides both parallel profiling and tracing in its measurement system. Parallel profiling characterizes the behavior of every application thread in terms of its aggregate performance metrics such as total exclusive time, inclusive time, number of calls, and child calls executed. A rich set of profiling functionality is available in TAU, including callpath profiling, phase profiling, and parameter-based profiling, that offers choices in scope and granularity of performance measurement. Although parallel profiling records minimal temporal information, it is the recommend first measurement choice in TAU because it allows significant performance characterization and runtime performance data is of a fixed size. All profiling measurements take place in a local context of execution and do not

involve synchronization or communication. This keeps it lightweight in overhead and intrusion even as the number of threads of execution scales.

However, the largest systems available now have exceeded many of the traditional means of profile data output, collection, and analysis. Tools like TAU have historically written process-local profile files. This method no longer scales to the largest systems since it creates challenges at multiple levels. It can excessively slow down the execution of the application job by creating potentially hundreds of thousands of files. The metadata operations to simply create this number of files have been shown to be a significant bottleneck [3]. After execution, the huge number of files is very difficult to manage and transfer between systems. A more subtle problem is that TAU assigns event identifiers dynamically and locally. This means that the same event can have different IDs in different threads. Event unification has typically been done in TAU in the analysis tools. Unfortunately, this requires verbose and redundant event information to be written with the profiles. Thus, not only do we end up with multiple profile files, they contain excessive information.

The TAU project has been investigating these two issues for the past year. We currently have prototype parallel implementations of *event unification* and *profile merging*. These are built from a MPI-based parallel profile analyzer that runs at the end of the application execution[15]. By using an efficient reduction layer based on a binomial heap, the unification and merging operations are implementation is portable and fast. We have tested it on over 100,000 cores on a Cray XT5 and IBM BG/P. More generally, we are looking to improve the scalability of online profile-based performance measurement. A TAU monitoring system is being implemented that uses scalable infrastructure such as MRNet to provide runtime access to parallel performance data [8,15].

Moving forward, we plan to implement various forms of on-the-fly analysis at the end of application execution, to reduce the burden on the post-mortem analysis tools, and online, to provide data reduction and feedback to the live application. For post-mortem analysis purposes, a new file format will be designed to contain multiple levels of detail and pre-computed derived data (e.g., from the runtime parallel analysis). This will allow the analysis tools the ability to read only the portions of the overall profile that they need for a given analysis or data view. In these way, we are confident that we can address the issues of large scale profile collection and analysis.

3.2 Parallel Tracing

In contrast to profiling, tracing generates a timestamped event log that shows the temporal variation of application performance. TAU traces can be merged and converted to the Vampir's [1] Open Trace Format (OTF), Scalasca's Epilog [5], Paraver [13], or Jumpshot's SLOG2 trace formats. Merging and conversion of trace files is an expensive operation at large core counts. To reduce the time for merging and conversion, and to provide more detailed event information, TAU interfaces with the Scalasca and VampirTrace libraries directly. VampirTrace provides a trace unification phase at the end of execution that requires re-writing

binary traces with updated global event identifiers. However, this can be an expensive operation at large scale.

In the near future, TAU will write OTF2 traces natively using the Score-P measurement library from the SILC[14] project. It will feature an efficient trace unification system that only re-writes global event identifier tables instead of re-writing the binary event traces. If the trace visualizer supports the OTF2 format, it will also eliminate the need to convert these large trace files from one format to another. This will improve the scalability of the tracing system.

3.3 Measuring MPI Collective Operations

As applications are re-engineered to run on ever increasing machine sizes, tracking performance of the collective operations on the basis of individual MPI communicators becomes more important. We have recently introduced tracking of MPI communicators in TAU's profiling substrate using its mapping capabilities in parameter-based profiling. TAU partitions the performance data on the basis of its communicator in a collective operation. Each communicator is identified by the list of MPI ranks that belong to it. When multiple communicators use the same set of ranks, the TAU output distinguishes each communicator based on its address. Figure 3 shows the breakdown of the average time spent in the `MPI_Allreduce` routine based on each set of communicators across all 32 ranks in an MPI application. To contend with large core counts, TAU only displays the first eight ranks in a communicator, although this depth may be altered by the user while configuring TAU's measurement library. This is shown for the `MPI_Bcast` call where all ranks participate in the broadcast operation on the `MPI_COMM_WORLD` communicator.

4 ParaProf Scalable Analysis

Scalable performance measurement only produces the performance data. It still needs to be analyzed. Analysis scalability concerns the exploration of potentially large parallel performance datasets. The TAU ParaProf parallel performance analyzer is specifically built for analysis of large scale data from the largest leadership class machines. It can easily analyze full size datasets on common desktop workstations. TAU provides a compressed, normalized, packed data format (ParaProf Packed format, `.ppk`) as a container for profile data from any supported measurement tool. This makes reading of parallel profiles significantly more efficient in ParaProf.

Analysis in ParaProf takes place in-memory for fast access and to support global aggregation and analysis views. Basic bar charts support large dataset with standard scrollbars allowing the detail for each node/thread to be seen in its own context. Additionally, we present aggregate statistics such as the mean and standard deviation. Aggregate views such as the histogram display allow a simplified view of the entire dataset in a single chart.

TAU: ParaProf: Mean Statistics – conc_allred.32p.ppk

Name △	Exclusive TIME	Inclusive TIME	Inclusive TIME / Call
MPI_Allreduce()	17,195.69	17,196.47	343.93
MPI_Allreduce() [<comm> = <ranks: 0, 8, 16, 24> <addr=0x1ab800a0>]	533.94	533.94	341.72
MPI_Allreduce() [<comm> = <ranks: 0, 8, 16, 24> <addr=0x17de0ef0>]	538.41	538.41	344.58
MPI_Allreduce() [<comm> = <ranks: 0, 8, 16, 24> <addr=0xc58d0c0>]	533.66	533.66	341.54
MPI_Allreduce() [<comm> = <ranks: 0, 8, 16, 24> <addr=0xdb64050>]	530.62	530.62	339.60
MPI_Allreduce() [<comm> = <ranks: 1, 9, 17, 25> <addr=0x1a31a070>]	532.47	532.47	340.78
MPI_Allreduce() [<comm> = <ranks: 1, 9, 17, 25> <addr=0x11ee10d0>]	531.00	531.00	339.84
MPI_Allreduce() [<comm> = <ranks: 1, 9, 17, 25> <addr=0xa29ef0>]	534.94	534.94	342.36
MPI_Allreduce() [<comm> = <ranks: 1, 9, 17, 25> <addr=0xdac0fb0>]	538.09	538.09	344.38
MPI_Allreduce() [<comm> = <ranks: 2, 10, 18, 26> <addr=0x10b36fd0>]	541.66	541.66	346.66
MPI_Allreduce() [<comm> = <ranks: 2, 10, 18, 26> <addr=0x16a0def0>]	537.91	537.91	344.26
MPI_Allreduce() [<comm> = <ranks: 2, 10, 18, 26> <addr=0x12438000>]	535.47	535.47	342.70
MPI_Allreduce() [<comm> = <ranks: 2, 10, 18, 26> <addr=0xa6bf070>]	534.12	534.12	341.84
MPI_Allreduce() [<comm> = <ranks: 3, 11, 19, 27> <addr=0x6cd70d0>]	549.62	549.62	351.76
MPI_Allreduce() [<comm> = <ranks: 3, 11, 19, 27> <addr=0x9f5ffd0>]	555.09	555.09	355.26
MPI_Allreduce() [<comm> = <ranks: 3, 11, 19, 27> <addr=0x238c070>]	550.78	550.78	352.50
MPI_Allreduce() [<comm> = <ranks: 3, 11, 19, 27> <addr=0x11146ef0>]	560.97	560.97	359.02
MPI_Allreduce() [<comm> = <ranks: 4, 12, 20, 28> <addr=0x1a83a000>]	516.84	516.84	330.78
MPI_Allreduce() [<comm> = <ranks: 4, 12, 20, 28> <addr=0x1284f070>]	523.38	523.38	334.96
MPI_Allreduce() [<comm> = <ranks: 4, 12, 20, 28> <addr=0xc939fd0>]	523.88	523.88	335.28
MPI_Allreduce() [<comm> = <ranks: 4, 12, 20, 28> <addr=0xe214ef0>]	528.34	528.34	338.14
MPI_Allreduce() [<comm> = <ranks: 5, 13, 21, 29> <addr=0x1b2c8ef0>]	519.31	519.31	332.36
MPI_Allreduce() [<comm> = <ranks: 5, 13, 21, 29> <addr=0x1194f000>]	515.12	515.12	329.68
MPI_Allreduce() [<comm> = <ranks: 5, 13, 21, 29> <addr=0x4003070>]	512.88	512.88	328.24
MPI_Allreduce() [<comm> = <ranks: 5, 13, 21, 29> <addr=0xdf9fd0>]	520.78	520.78	333.30
MPI_Allreduce() [<comm> = <ranks: 6, 14, 22, 30> <addr=0x191b6ef0>]	550.69	550.69	352.44
MPI_Allreduce() [<comm> = <ranks: 6, 14, 22, 30> <addr=0x1406b000>]	548.19	548.19	350.84
MPI_Allreduce() [<comm> = <ranks: 6, 14, 22, 30> <addr=0x4772fd0>]	554.53	554.53	354.90
MPI_Allreduce() [<comm> = <ranks: 6, 14, 22, 30> <addr=0x4003070>]	542.16	542.16	346.98
MPI_Allreduce() [<comm> = <ranks: 7, 15, 23, 31> <addr=0x2f08ef0>]	551.62	551.62	353.04
MPI_Allreduce() [<comm> = <ranks: 7, 15, 23, 31> <addr=0x15f95f50>]	546.59	546.59	349.82
MPI_Allreduce() [<comm> = <ranks: 7, 15, 23, 31> <addr=0x4772fd0>]	554.00	554.00	354.56
MPI_Allreduce() [<comm> = <ranks: 7, 15, 23, 31> <addr=0x6055000>]	549.41	549.41	351.62
MPI_Barrier()	1,610.78	1,610.78	805.39
MPI_Bcast()	2,299.25	2,299.78	2,299.78
MPI_Bcast() [<comm> = <ranks: 0, 1, 2, 3, 4, 5, 6, 7 ...> <addr=0x683f40>]	2,299.78	2,299.78	2,299.78

Fig. 3. ParaProf's shows the peformance of a collective operation partitioned by the communicator

ParaProf uses OpenGL-based 3D visualization support to enhance the interpretation of large-scale performance data. Here, millions of data elements can be visualized at once and be manipulated in real time. We provide triangle mesh displays, 3d bar plots, and scatterplots, all with width, height, depth, and color to provide 4 axes of differentiable data values. For instance, Figure 4 shows a ParaProf 3D view of the entire parallel profile for the XBEC application on 128K core of an IBM BG/P. Figure 5(left) is an example of ParaProf's new 3D communication matrix view showing the volume of point-to-point interprocessor communication between sender and receiver tasks. Although this is for a smaller execution, parallel programs larger than 2k processors will necessarily require such a 3D communications perspective.

Internally, the performance data representation in ParaProf is kept as minimally as possible. Rather than store NxM tables of performance data for each region and node, we keep sparse lists to allow for differing regions on each node. Our goal is to apply this to all visualization options where complete information is being rendered. However, it is also possible to conduct various forms of data dimensionality analysis and reduction. We have implemented several scalable analysis operations, including averaging, histogramming, and clustering.

To validate the scalability of TAU's paraprof profile browser, we synthesized a large one million core profile dataset by replicating a 32k core count dataset

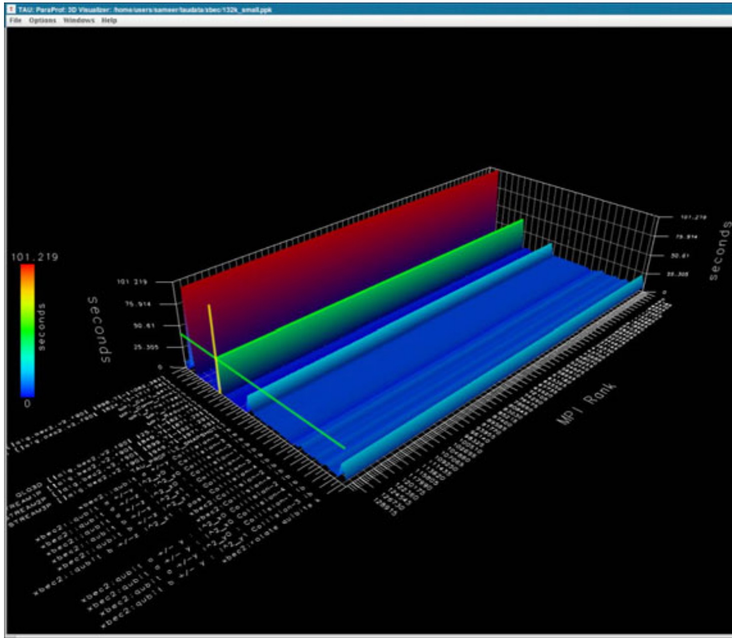


Fig. 4. ParaProf 3D browser shows the profile of a code running on 128k cores

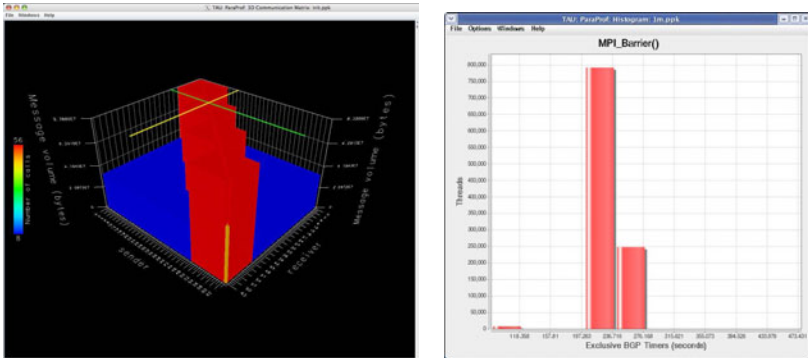


Fig. 5. Left: ParaProf's 3D communication matrix shows the volume of communication between a pair of communicating tasks. Right: ParaProf's histogram display showing the performance of `MPI_Barrier` in a synthesized 1 million core count profile dataset.

repeatedly. While it is cumbersome to scroll through a million lines representing individual MPI ranks, TAU's histogram displays are useful in highlighting the performance variation of a routine across multiple cores. Figure 5(right) shows a histogram display of the distribution of threads based on their `MPI_Barrier` execution time. The number of bins partitions the range of the chose performance metric for an event, and this can be selected by the user. Our goal here was to

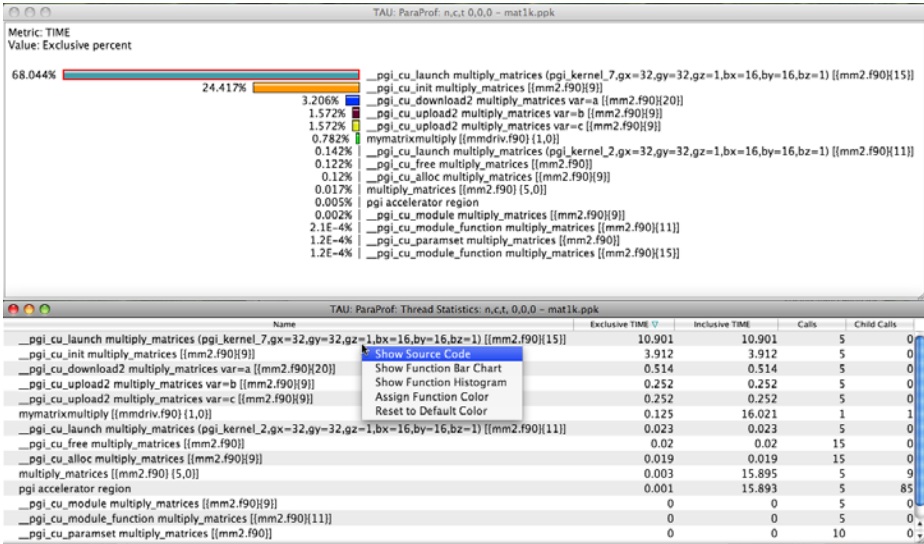


Fig. 6. ParaProf's shows the profile of a kernel executing on a GPGPU using PGI's runtime library instrumentation

ensure that the profile browsers were capable of handling large data volumes and able to handle displays of millions of cores. We do not have access to machines with a million cores at present, but such large scale machines are being built and will be available in the near future.

5 Conclusions and Future Work

Scaling will continue to be a dominant concern in high-end computing, especially as attention turns towards exascale platforms for science and engineering applications. High levels of concurrent execution on upwards of one million cores are being forecast by the community. Parallel performance tools must continue to be enhanced, re-engineered, and optimized to meet these scaling challenges in instrumentation, measurement, and analysis.

Scaling is not the only concern. Future HPC systems will likely rely on heterogeneous architectures comprised of accelerator components (GPGPU). This will require development of performance measurement and analysis infrastructure to understand parallel efficiency of the application at all levels of execution. We are working closely with compiler vendors (such as PGI and CAPS Entreprise HMPP) to target instrumentation of accelerator kernels at the runtime system level. Using weak bindings of key runtime library events, TAU can intercept and track the time spent in key events as they execute on the host. For instance, Figure 6 shows the time spent in launching individual kernels on the GPGPU as well as the time spent in transferring data from the host memory to the memory of the GPGPU. Variable names as well as source locations are shown in the profile display.

However, in general, the heterogeneous environment will dictate what is possible for performance observation. The challenge for heterogeneous performance tools will be to capture performance data at all levels of execution and integrate that information into consistent, coherent representation of performance for analysis purposes. Heterogeneity introduces issues such as asynchronous, overlapped concurrency between the CPU and accelerator devices, and potentially limited performance measurement visibility, making solutions to this challenge difficult.

References

1. Brunst, H., Kranzlmüller, D., Nagel, W.E.: Tools for Scalable Parallel Program Analysis - Vampir NG and DeWiz. In: Distributed and Parallel Systems, Cluster and Grid Computing, vol. 777 (2004)
2. Buck, B., Hollingsworth, J.: An API for Runtime Code Patching. Journal of High Performance Computing Applications 14(4), 317–329 (2000)
3. Frings, W., Wolf, F., Petkov, V.: Scalable Massively Parallel I/O to Task-Local Files. In: Proc. SC 2009 Conference (2009)
4. Geimer, M., Shende, S.S., Malony, A.D., Wolf, F.: A Generic and Configurable Source-Code Instrumentation Component. In: Allen, G., Nabrzyski, J., Seidel, E., van Albada, G.D., Dongarra, J., Sloot, P.M.A. (eds.) ICCS 2009, Part II. LNCS, vol. 5545, pp. 696–705. Springer, Heidelberg (2009)
5. Geimer, M., Wolf, F., Wylie, B., Brian, J.N., Ábrahám, E., Becker, D., Mohr, B.: The SCALASCA Performance Toolset Architecture. In: Proc. of the International Workshop on Scalable Tools for High-End Computing (STHEC), pp. 51–65 (2008)
6. Lindlan, K.A., Cuny, J., Malony, A.D., Shende, S., Mohr, B., Rivenburgh, R., Rasmussen, C.: A Tool Framework for Static and Dynamic Analysis of Object-Oriented Software with Templates. In: Proc. of SC 2000 Conference (2000)
7. Mucci, P.: Dynaprof (2010), http://www.cs.utk.edu/~mucci/dynaprof
8. Nataraj, A., Malony, A., Morris, A., Arnold, D., Miller, B.: In Search of Sweet-Spots in Parallel Performance Monitoring. In: Proc. IEEE International Conference on Cluster Computing (2008)
9. Roth, P., Arnold, D., Miller, B.: Proc. High-Performance Grid Computing Workshop, IPDPS (2004)
10. Shende, S., Malony, A., Ansell-Bell, R.: Instrumentation and Measurement Strategies for Flexible and Portable Empirical Performance Evaluation. In: Proc. Tools and Techniques for Performance Evaluation Workshop, PDPTA. CSREA, pp. 1150–1156 (2001)
11. Shende, S., Malony, A.D.: The TAU Parallel Performance System. The International Journal of High Performance Computing Applications 20(2), 287–311 (2006)
12. Shende, S., Malony, A.D., Morris, A.: Simplifying Memory, I/O, and Communication Performance Assessment using TAU. In: Proc. DoD UGC 2010 Conference. IEEE Computer Society (2010)
13. Barcelona Supercomputing Center, "Paraver" (2010), http://www.bsc.es/paraver
14. VI-HPS, "SILC" (2010), http://www.vi-hps.org/projects/silc
15. Lee, C.W., Malony, A.D., Morris, A.: TAUmon: Scalable Online Performance Data Analysis in TAU. In: Guarracino, M.R., Vivien, F., Träff, J.L., Cannatoro, M., Danelutto, M., Hast, A., Perla, F., Knüpfer, A., Di Martino, B., Alexander, M. (eds.) Euro-Par-Workshop 2010. LNCS, vol. 6586, pp. 493–499. Springer, Heidelberg (2011)