TAUg: Runtime Global Performance Data Access Using MPI

Kevin A. Huck, Allen D. Malony, Sameer Shende, and Alan Morris

Performance Research Laboratory
Department of Computer and Information Science
University of Oregon, Eugene, OR, USA
{khuck,malony,sameer,amorris}@cs.uoregon.edu
http://www.cs.uoregon.edu/research/tau

Abstract. To enable a scalable parallel application to view its global performance state, we designed and developed *TAUg*, a portable runtime framework layered on the TAU parallel performance system. TAUg leverages the MPI library to communicate between application processes, creating an abstraction of a global performance space from which profile views can be retrieved. We describe the TAUg design and implementation and show its use on two test benchmarks up to 512 processors. Overhead evaluation for the use of TAUg is included in our analysis. Future directions for improvement are discussed.

Keywords: parallel, performance, runtime, MPI, measurement.

1 Introduction

Performance measurement of parallel applications balances the need for fine-grained performance data (to understand relevant factors important for improvement) against the cost of observation (measurement overhead and its impact on performance behavior). This balance becomes more delicate as parallel systems increase in scale, especially if the scalability of the performance measurement system is poor. In practice, measurements are typically made for post-mortem analysis [1, 2], although some tools provide online monitoring[3] and analysis for purposes of performance diagnosis [4, 5] and steering [6–9]. For any performance experiment, the performance measurement system is an intimate part of the application's execution and need/cost tradeoffs must take this into account.

Scalable efficiency necessitates that performance measurements be made concurrently (in parallel threads of execution) without centralized control. The runtime *parallel performance state* can be considered to be logically a part of the application's global data space, but it must be stored distributively, local to where the measurements took place, to avoid unnecessary overhead and contention. Measurement tools for post-mortem analysis typically output the final performance state at the end of program execution. However, online tools require access the distributed performance state during execution.

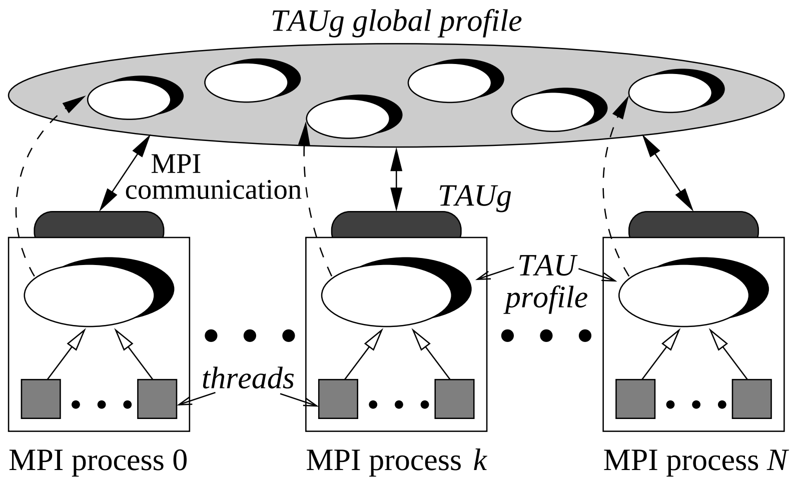

Fig. 1. TAUg System Design.

In this paper, we consider the problem of runtime support for application-level access to global parallel performance data. Our working assumption is that the importance of online performance data access is decided by the application, but will depend directly on the efficiency of the solution. It is equally important that the solution be as portable as possible, flexible enough to accommodate its use with different parallel computation scenarios, and scalable to large numbers of processes. The main challenges are in defining useful programming abstractions for coordinated performance data access, and in creating necessary infrastructure that meets portability and efficiency objectives.

We describe a solution for use with the TAU parallel performance system called *TAUg*. The TAUg design targets MPI-based applications (see §2) and utilizes MPI in its default implementation (see §3) for portability and scalability. The initial version of TAUg was tested with ASCI benchmarks sPPM and Sweep3D and a synthetic load balancing simulation. The results are reported in §4. Discussion of the TAUg approach and our future goals are discussed in §5. Related work is in §6, and §7 gives concluding remarks.

2 Design

In our approach to the TAUg system design, we first identified the desired operational abstraction, and second, considered how best to implement it with MPI. Figure 1 shows these two perspectives. The bottom part of the figure represents what TAU produces as a profile for each process. The TAU profile is an aggregation of individual thread profiles. TAUg provides the abstraction of a globally-shared performance space, the TAUg global profile. The dashed lines represent the promotion of each process profile into this space. TAUg uses MPI to create this global abstraction on behalf of the application.

2.1 Views and Communicators

In TAU, events are defined for measurement of intervals (e.g., entry and exit events for subroutines) or automic operations (e.g., memory allocation events). In TAUg, the *global performance space*, representing all events (interval and atomic events) profiled on all processes and threads, is indexed along two dimensions. The first dimension is called the *TAUg (global) performance view*, and represents a subset of the performance profile data being collected (i.e., a subset of the TAU profiled events). In our initial implementation, a view can specify only one event, whose profile gives the performance for that event measured when the event is active. The other dimension is called the *TAUg (global) performance communicator*, and represents a subset of the MPI processes in the application. The notion of the TAUg communicator is that only those processes within the communicator will share TAUg performance views, so as to minimize perturbation of the application.

2.2 Programming Interface

TAUg is designed to be a simple, lightweight mechanism for sharing TAU performance data between MPI processes. The only prerequisites for using TAUg are that the application already be using MPI and TAU. The three methods in the API are designed to be in the same style as MPI methods. These methods are callable from Fortran, C or C++.

An application programmer uses TAUg by first defining the global performance views and communicators. The method `TAU_REGISTER_VIEW` is used to specify a global performance view. This method takes as an input parameter the name of a TAU profiled event, and has an output parameter of an ID for the view. `TAU_REGISTER_VIEW` need only be called by processes that will use the view with TAUg communicators they define.

The method `TAU_REGISTER_COMMUNICATOR` is used to create a global performance communicator. It takes two input parameters; an array of process ranks in `MPI_COMM_WORLD` and the size of the array. The only output parameter is the newly created communicator ID. Because of MPI requirements when creating communicators, `TAU_REGISTER_COMMUNICATOR` must be called by all processes. The following code listing shows an example of how the `TAU_REGISTER_VIEW` and `TAU_REGISTER_COMMUNICATOR` methods would be used in C to create a global performance view of the event `calc()` and a global performance communicator containing all processes.

```
int viewID = 0, commID = 0, numprocs = 0;
TAU_REGISTER_VIEW("calc()", &viewID);
MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
int members[numprocs];
for (int i = 0 ; i < numprocs ; i++) { members[i] = i; }
TAU_REGISTER_COMMUNICATOR(members, numprocs, &commID);
```

Having created all the global performance views and communicators needed to access the global application performance, the application programmer calls the method TAU_GET_VIEW to retrieve the data. This method takes a view ID and a communicator ID as input parameters. It also takes a collective communication type as an input parameter. The idea here is to allow TAU communicators to pass profile data between themselves in different ways. The supported communication types are TAU_ALL_TO_ONE, TAU_ONE_TO_ALL and TAU_ALL_TO_ALL. If TAU_ALL_TO_ONE or TAU_ONE_TO_ALL are used, a processor rank in MPI_COMM_WORLD will represent the source or sink for the operation[1]. There are two output parameters which specify an array of doubles and the size of the array. TAU_GET_VIEW need only be called by the processes which are contained in the specified TAU global performance communicator. The following code listing shows an example of how the TAU_GET_VIEW method would be used in C.

```
double *loopTime;
int size = 0, myid = 0, sink = 0;
MPI_Comm_rank(MPI_COMM_WORLD,&myid);
TAU_GET_VIEW(viewID, commID, TAU_ALL_TO_ONE, sink,
    &loopTime, &size);
if (myid == 0) { /* do something with the result... */ }
```

In summary, this application code is requesting that all processes send performance information for the event `calc()` to the root process. The root process, for example, can thenchoose to modify the application behavior based on the running total for the specified event.

3 Implementation

TAUg is written in C++, and comprises a public C interface consisting of only the three static methods described in Section 2.2. The complete interface for the API is listed here:

```
void static TAU_REGISTER_VIEW (const char* event_name,
    int* viewID);
void static TAU_REGISTER_COMMUNICATOR (int members[],
    int size, int* commID);
void static TAU_GET_VIEW (int viewID, int commID,
    int type, int sink, double** data, int* outSize);
```

The TAU_REGISTER_VIEW method creates a new global performance view structure, and stores it internally. The new view ID is returned to the calling method. The TAU_REGISTER_COMMUNICATOR method creates new MPI group and communicator objects which contain the input process ranks, assumed to be relevant in MPI_COMM_WORLD. It then stores the MPI communicator ID and all the communicator parameters internally, and returns the new communicator ID (not to be confused with the MPI communicator type) to the calling method.

[1] If the TAU_ALL_TO_ALL type is specified, the source/sink parameter is ignored.

The `TAU_GET_VIEW` method first looks up the global performance view and communicator in the internal structures. At the same time, the code converts the source/sink process rank from `MPI_COMM_WORLD` to its rank in the global performance communicator. The method then accesses TAU to get the profile data for the global performance view. The profile data includes the inclusive and exclusive timer values, number of calls and number of subroutines (events called from this event). This data is then packaged in an MPI type structure and sent to the other processes in the global performance communicator using collective operations. Either `MPI_Allgather`, `MPI_Gather` or `MPI_Scatter` is called, depending on whether the application wants `TAU_ALL_TO_ALL`, `TAU_ALL_TO_ONE` or `TAU_ONE_TO_ALL` behavior, respectively. In the initial implementation, an array of only the exclusive timer values is returned to the user as a *view result*.

4 Experiments

4.1 Application Simulation

TAUg was integrated into a simple simulation program to demonstrate its effectiveness in dynamically load balancing an MPI application. This simulation is intended to replicate general situations where factors external to the application are affecting performance, whether it be hardware differences or other load interference on a shared system. In this experiment, the application program simulates a heterogeneous cluster of n processors, where $n/2$ of the nodes are twice as fast as the other $n/2$ processors.

Initially, each MPI process is designated an equal portion of the work to execute. After each timestep, the application code queries TAUg to get a global view of the application performance. Processes which are slower than the average are given a reduced workload, and the processes which are faster than the average are given an increased workload. This process is iterated 20 times. The application was tested with 5 configurations. Initially, an unbalanced version of the application was tested and compared to a dynamically balanced version. It soon became apparent that different lengths of performance data "decay" are necessary to detect when the load has become balanced, so that the faster nodes are not overburdened simply so that the slower nodes can catch up. Therefore, three more configurations were tested, which used only the previous 1, 2, and 4 timesteps, respectively. Using the unbalanced application as a baseline for the 32 processor simulation, the dynamically balanced simulation is 15.9% faster, and the dynamically balanced simulation which only considers the previous 1 timestep is 26.5% faster. Longer running simulations show similar speedup.

This simple example demonstrates that TAUg can be used to implement the knowledge portion of a load balancing algorithm. In general,load imbalance is reflected in performance properties (exeuction time and even more detailed behavior), but is caused by and associated with application-specific aspects (such as poor grid partitioning). TAU can be used to measure both performance and application aspects. TAUg then provides an easy-to-use interface for retrieving the information in a portable way.

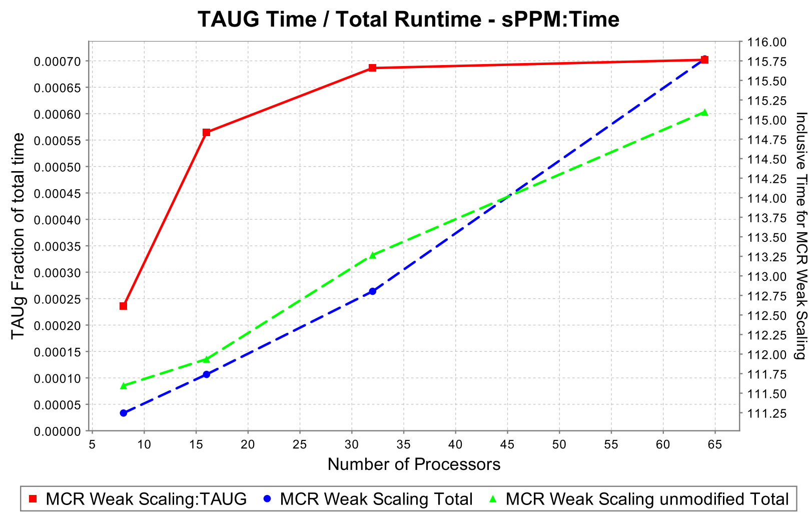

Fig. 2. Comparison of sample execution times from modified and unmodified sPPM, and fraction of time spent in TAUg. Examining the Y-axis on the right to compare total runtime measurements, the application is not significantly affected by the addition of TAUg.

4.2 Overhead and Scalability: sPPM and Sweep3D

The sPPM benchmark[10] solves a 3D gas dynamics problem on a uniform Cartesian mesh using a simplified version of the PPM (Piecewise Parabolic Method) code. We instrumented sPPM with TAU, and TAUg calls were added to get a global performance view for each of 22 subroutines in the application code, for each of 8 equal sized communicators. The sPPM benchmark iterates for 20 double timesteps, and at the end of each double timestep, sPPM was modified to request global performance data for each global performance view / communicator tuple. [2]

We ran sPPM on MCR, a 1,152 node dual P4 2.4-Ghz cluster located at Lawrence Livermore National Laboratory. Using a weak scaling test of up to 64 processors, TAUg total overhead never exceeds 0.1% of the total runtime, and the application is not significantly perturbed. Figure 2 shows the comparison of the modified and unmodified sPPM performance.

ASCI Sweep3D benchmark [11] is a solver for the 3-D, time-independent, neutron particle transport equation on an orthogonal mesh. Sweep3D was instrumented with TAU, and TAUg calls were added to get a global performance view for one of the methods in the application code and one communicator consisting of all processes. The Sweep3D benchmark iterates for 200 timesteps, and at the end of each timestep, Sweep3D was modified to request global performance data for the global performance view / communicator tuple.

[2] This resulted in 22 calls to `TAU_GET_VIEW` since only one subroutine event can be in a view in the current version.

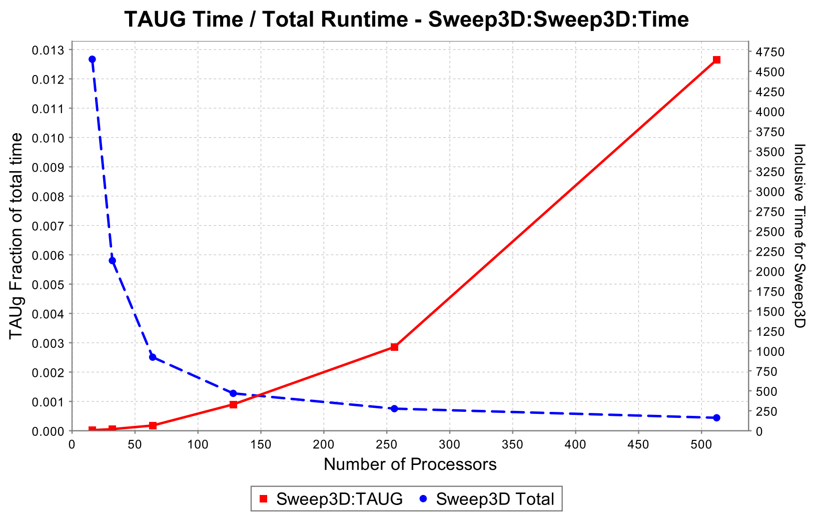

Fig. 3. Fraction of TAUg overhead as measured in Sweep3D in a strong scaling experiment.

We ran Sweep3D on ALC, a 960 node dual P4 2.4-Ghz cluster located at Lawrence Livermore National Laboratory. During a strong scaling test of up to 512 processors, TAUg total overhead never exceeded 1.3% of the total runtime, and the application was not significantly perturbed. Figure 3 shows the comparison of the modified Sweep3D performance to the TAUg overhead.

5 Discussion and Future Work

There are several issues to discuss with respect to the current TAUg system as well as areas where we are interested in improving the TAUg design and implementation. Currently, TAUg limits global access to the exclusive value of a single event for a single metric. We will add support for specifying multiple events in a TAUg view and an *all* tag for easily specifying all events. Similarly, hardware performance counter information may be useful to many applications using TAUg, such as floating point operations or cache misses. This information is currently available in TAUg, but only one metric is available at a time. TAU supports tracking multiple concurrent counters, and TAUg will be extended to support this as well. We also will allow `TAU_GET_VIEW` to be called with an array of views.

The TAUg communication patterns cover what we felt were common use cases. However, they translate into collective MPI operations in TAUg. We believe there will be value to supporting TAUg send and receive operations, to allow more pairwise performance exchange, still within a TAU communicator. This will also allow the opportunity for blocking and non-blocking communication to be used in TAUg. We will also experiment with one-sided communication in MPI-2 to reduce the effects of our current collective operation approach.

Presently, TAUg returns only the raw performance view to the application. We plan to implement TAUg helper functions to compute profile statistics that are typically offered post-mortem (e.g., mean, min, max, and standard deviation for a performance metric). One particularly useful function would take two compatible view results and calculate their difference. This would help to address a problem of calculating incremental global performance data from the last time it was viewed.

6 Related Work

TAUg has similarities to research in online performance analysis and diagnosis. Autopilot [6] uses a distributed system of sensors to collect data about an application's behavior and actuators to make modifications to application variables to change its behavior. Peridot [12] extends this concept with a distributed performance analysis system composed of agents that monitor and evaluate hierarchically-specified "performance properties" at runtime. The Distributed Performance Consultant in Paradyn [5], coupled with MRNet, provides a scalable diagnosis framework to achieve these goals. Active Harmony [13] takes one step further to include a component that automatically tunes an application's performance by adjusting application parameters.

While TAUg can be used to achieve the same purpose of performance diagnosis and online tuning, it focuses as a technology only on the problem of portable access to global performance data. In this way, its use is more general and can be applied more robustly on different platforms.

7 Conclusion

Measurement of parallel program performance is commonly done as part of a performance diagnosis and tuning cycle. However, an application may desire to query its runtime performance state to make decisions that direct how the computation will proceed. Most performance measurement systems provide little support for dynamic performance data access, much less for performance data inspection across all application processes. We developed TAUg as an abstraction layer for parallel MPI-based applications to retrieve performance views from a global performance data space. The TAUg infrastructure is built on top of the TAU performance system which generates performance profiles for each application process. TAUg uses MPI collective operations to provide access to the distributed performance data.

TAUg offers two important benefits to the application developer. First, the TAUg programming interface defines the TAU *communicator* and *view* abstractions that the developer can use to create instances specific to their runtime performance query needs. The `TAU_GET_VIEW` function will return the portion of the global performance profiles selected by the communicator and view parameters. As a result, the developer is insulated from the lower level implementation. Second, the use of MPI in TAUg's implementation affords significant portability, and the scalability of TAUg is only limited by the scalability of the local MPI implementation. Any parallel systems supporting MPI and TAU are candidates for use of TAUg.

It is true that TAUg will necessarily influence the application's operation. We provide some analysis of the overhead generated by TAUg in our benchmark tests. However, the impact of TAUg will depend directly on how the application chooses to use it. This impact is true both of its perturbation of performance as well as its ability to provide the application with performance knowledge for runtime optimization.

References

1. Shende, S., Malony, A.D.: The tau parallel performance system. The International Journal of High Performance Computing Applications (2005) *(to appear)*.
2. KOJAK: Kojak. http://www.fz-jeulick.de/zam/kojak/ (2006)
3. Wismuller, R., Trinitis, J., Ludwig, T.: Ocm – a monitoring system for interoperable tools. In: Proceedings 2nd SIGMETRICS Symposium on Parallel and Distributed Tools (SPDT'98). (1998) 1–9
4. Miller, B., Callaghan, M., Cargille, J., Hollingsworth, J., Irvin, R., Karavanic, K., Kunchithapadam, K., Newhall, T.: The paradyn parallel performance measurement tool. Computer **28**(11) (1995) 37–46
5. Roth, P., Miller, B.: On-line automated performance diagnosis on thousands of processes. In: Proceedings Proc. 11th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. (2006) 69–80
6. Ribler, R., Simitci, H., Reed, D.: The Autopilot performance-directed adaptive control system. Future Generation Computer Systems **18**(1) (2001) 175–187
7. Eisenhauer, G., Schwan, K.: An object-based infrastructure for program monitoring and steering. In: Proceedings 2nd SIGMETRICS Symposium on Parallel and Distributed Tools (SPDT'98). (1998) 10–20
8. Gu, W., et al.: Falcon: On-line monitoring and steering of large-scale parallel programs. In: Proceedings of the 5th Symposium of the Frontiers of Massively Parallel Computing. (1995) 422–429
9. Tapus, C., Chung, I.H., Hollingworth, J.: Active harmony: Towards automated performance tuning. In: SC2002: Proceedings of the 2002 ACM/IEEE conference on Supercomputing. (2002)
10. LLNL: The asci sppm benchmark code. http://www.llnl.gov/asci/purple/benchmarks/limited/sppm/ (2006)
11. LLNL: The asci sweep3d benchmark. http://www.llnl.gov/asci/purple/benchmarks/limited/sweep3d/ (2006)
12. Gerndt, M., Schmidt, A., Schulz, M., Wismuller, R.: Performance analysis for teraflop computers - a distributed automatic approach. In: Euromicro Workshop on Parallel, Distributed, and Network-based Processing (PDP), Canary Islands, Spain (2002) 23–30
13. Hollingsworth, J., Tabatabaee, V., Tiwari, A.: Active harmony. http://www.dyninst.org/harmony/ (2006)