Scalable Methods for Monitoring and Detecting Behavioral Equivalence Classes in Scientific Codes [*]

Todd Gamblin tgamblin@cs.unc.edu
Rob Fowler rjf@renci.org
Daniel A. Reed dan_reed@renci.org

Renaissance Computing Institute
University of North Carolina at Chapel Hill

Abstract

Emerging petascale systems will have many hundreds of thousands of processors, but traditional task-level tracing tools already fail to scale to much smaller systems because the I/O backbones of these systems cannot handle the peak load offered by their cores. Complete event traces of all processes are thus infeasible. To retain the benefits of detailed performance measurement while reducing volume of collected data, we developed AMPL, a general-purpose toolkit that reduces data volume using stratified sampling.

We adopt a scalable sampling strategy, since the sample size required to measure a system varies sub-linearly with process count. By grouping, or stratifying, *processes that behave similarly, we can further reduce data overhead while also providing insight into an application's behavior.*

In this paper, we describe the AMPL toolkit and we report our experiences using it on large-scale scientific applications. We show that AMPL can successfully reduce the overhead of tracing scientific applications by an order of magnitude or more, and we show that our tool scales sub-linearly, so the improvement will be more dramatic on petascale machines. Finally, we illustrate the use of AMPL to monitor applications by performance-equivalent strata, *and we show that this technique can allow for further reductions in trace data volume and traced execution time.*

1 Introduction

Processor counts in modern supercomputers are rising rapidly. Of the systems on the current Top500 list [13], over 400 are labeled as distributed-memory "clusters", up from just over 250 in 2005. The mean processor count of systems in the Top 100 has risen exponentially in the past decade. In 1997, the fastest system had just over 1,000 processors, while the current performance leader, IBM's Blue Gene/L[5], has over 200,000 cores. Only one system in the current top 100 has fewer than 1,000 processors.

Effectively monitoring highly concurrent systems is a daunting challenge. An application event trace can generate hundreds of megabytes of data for each minute of execution time, and this data needs to be stored and analyzed offline. However, the largest supercomputers are using diskless nodes. For example, Blue Gene/L at Lawrence Livermore National Laboratory has 106,496 diskless nodes for computation, but only 1,664 I/O nodes. Each I/O node is connected by gigabit ethernet to a network of 224 I/O data servers. Peak throughput of this system is around 25 GB/s [16]. A full trace from 212,992 processors could easily saturate this pathway, perturbing measurements and making the recorded trace useless.

Even if a large trace could be collected and stored efficiently, traces from petascale systems would contain far more data than could be analyzed manually. Fortunately, Amdahl's law constrains scalable applications to exhibit extremely regular behavior. A scalable performance monitoring system could exploit such regularity to remove redundancies in collected data so that its outputs would not depend on total system size. An analyst using such a system could collect just enough performance data to assess application performance, and no more.

Using simulation and *ex post facto* experiments, Mendes, *et al.* [12] showed that statistical sampling is a promising approach to the data reduction problem. It can be used to accurately estimate the global properties of a population of processes without collecting data from all of them. Sampling is particularly well-suited to large systems, since the sample size needed to measure a set of processes scales sub-linearly with the size of the set. For data with fixed variance, the sample size is constant in the limit, so sampling very large populations of processes is proportionally much less costly than measuring small ones.

[*]Part of this work was performed under the auspices of the SciDAC Performance Engineering Research Institute, grant number DE-FC02-06ER25764.

In this paper, we extend Mendes' work with infrastructure for on-line, sampled event tracing of arbitrary performance metrics gathered using on-node instrumentation. Summary data is collected dynamically and used to tune the sample size as a run progresses. We also explore the application of techniques for subdividing, or *stratifying*, a population into independently sampled behavioral equivalence classes. Stratification can provide insight into the workings of an application, as it gives the analyst a rough classification of the behavior of running processes. If the behavior within each stratum is homogeneous, the overall cost of monitoring is reduced. We have implemented these techniques in the Adaptive Monitoring and Profiling Library (AMPL), which can be linked with instrumented applications written in C, C++, or FORTRAN.

We review the statistical methods used in this paper in §2. We describe the architecture and implementation of AMPL in §3. An experimental validation of AMPL is given in §4 using sPPM[1], Chombo[3], and ADCIRC[11], three well known scientific codes. Finally, §5 discusses related work, and §6 details conclusions drawn from our results as well as plans for future work.

2 Statistical Sampling Theory

Statistical sampling has long been used in surveys and opinion polls to estimate general characteristics of populations by observing the responses of only a small subset, or sample, of the total population. Below, we review the basic principles of sampling theory, and we present their application to performance monitoring on large-scale computing systems. We also discuss stratified sampling and its role in reducing measurement overhead in scientific applications.

2.1 Estimating Mean Values

Given a set of population elements Y, sampling theory estimates the mean using only a small sample of the total population. For sample elements, $y_1, y_2, ..., y_n$, the sample mean $\bar{y}$ is an estimator of the population mean $\bar{Y}$. We would like to ensure that the value of $\bar{y}$ is within a certain error bound d of $\bar{Y}$ with some confidence. If we denote the risk of not falling within the error bound as α, then the confidence is $1 - \alpha$, yielding

$$Pr(|\bar{Y} - \bar{y}| > d) \leq \alpha. \tag{1}$$

Stated differently, z_α standard deviations of the estimator should fall within the error bound:

$$z_\alpha \sqrt{Var(\bar{y})} \leq d, \tag{2}$$

where z_α is the normal confidence interval computed from the confidence bound $1 - \alpha$. Given the variance of an estimator for the population mean, this inequality can be solved to obtain a minimum sample size, n, that will satisfy the constraints, z_α and d. For a simple random sample, we have

$$n \geq N \left[1 + N \left(\frac{d}{z_\alpha S} \right)^2 \right]^{-1} \tag{3}$$

where S is the standard deviation of the population, and N is the total population size. The estimation of mean values is described in [19, 12], so we omit further elementary derivations. However, two aspects of (3) warrant emphasis. First, (3) implies that the minimum cost of monitoring a population depends on its variance. Given the same confidence and error bounds. a population with high variance requires more sampled elements than a population with low variance. Intuitively, highly regular SPMD codes with limited data dependent behavior will benefit more from sampling than will more irregular, dynamic codes.

Second, as N increases, n approaches $(z_\alpha S/d)^2$, and the relative sampling cost n/N becomes smaller. For a fixed sample variance, the relative cost of monitoring declines as system size increases. As mentioned, sample size is constant in the limit, so sampling can be extremely beneficial for monitoring very large systems.

2.2 Sampling Performance Metrics

Formula (3) suggests that one can substantially reduce the number of processes monitored in a large parallel system, but we must modify it slightly for sampled traces. Formula (3) assumes that the granularity of sampling is in line with the events to be estimated. However, our population consists of M processes, each executing application code with embedded instrumentation. Each time control passes to an instrumentation point, some metric is measured for a performance event Y_i. Thus, the population is divided hierarchically into primary units (processes) and secondary units (events). Each process "contains" some possibly changing number of events, and when we sample a process, we receive all of its data. We must account for this when designing our sampling strategy.

A simple random sample of primary units in a partitioned population is formally called *cluster sampling*, where the primary units are "clusters" of secondary units. Here, we give a brief overview of this technique as it applies to parallel applications. A more extensive treatment of the mathematics involved can be found in [19].

We are given a parallel application running on M processes, and we want to sample it repeatedly over some time interval. The i^{th} process has N_i events per interval, such that

$$\sum_{i=1}^{M} N_i = N. \tag{4}$$

Events on each process are Y_{ij}, where $i = 1, 2, ..., M; j = 1, 2, ..., N_i$. The population mean $\bar{Y}$ is simply the mean over the values of all events:

$$\bar{Y} = \frac{1}{N} \sum_{i=1}^{M} \sum_{j=1}^{N} Y_{ij}. \tag{5}$$

We wish to estimate $\bar{Y}$ using a random sample of m processes. The counts of events collected from the sampled processes are referred to as n_i. $\bar{Y}$ can be estimated from the sample values with the *cluster sample mean*:

$$\bar{y}_c = \frac{\sum_{i=1}^{m} y_{iT}}{\sum_{i=1}^{m} n_i}, \tag{6}$$

where y_{iT} is the total of all sample values collected from the i^{th} process. The cluster mean $\bar{y}_c$ is then simply the sum of all sample values divided by the number of events sampled.

Given that $\bar{y}_c$ is an effective estimator for $\bar{Y}$, one must choose a suitable sample size to ensure statistical confidence in the estimator. To compute this, we need the variance, given by:

$$Var(\bar{y}_c) = \frac{M - m}{Mm\bar{N}^2} s_r^2, \quad s_r^2 = \frac{\sum_{i=1}^{m}(y_{iT} - \bar{y}_c n_i)^2}{m - 1} \tag{7}$$

where $\bar{N}$ is the average number of events for each process in the primary population, and s_r^2 is an estimator for the secondary population variance S^2. We can use $Var(\bar{y}_c)$ in (2) and obtain an equation for sample size as follows:

$$m = \frac{M s_r^2}{\bar{N}^2 V^2 + s_r^2}, \quad V = \left(\frac{d}{z_\alpha}\right)^2 \tag{8}$$

The only remaining unknown is N, the number of unique events. For this, we can use a straightforward estimator, $N \approx Mn/m$. We can now use equation (8) for adaptive sampling. Given an estimate for the variance of the event population, we can calculate approximately the size, m, of our next sample.

2.3 Stratified Sampling

Parallel applications often have behavioral equivalence classes among their processes, which is reflected in performance data about the application. For example, if process zero of an application reads input data, manages checkpoints and writes results, the performance profile of process zero will differ from that of the other processes. Similar situations arise from spatial and functional decompositions or master-worker paradigms.

One can exploit this property to reduce real-time monitoring overhead beyond what is possible with application-wide sampling. This is a commonly used technique in the design of political polls and sociological studies, where it may be very costly to survey every member of a population [19]. The communication cost of monitoring is the direct analog of this for large parallel applications.

Equation (3) shows that the minimum sample size is strongly correlated with variance of sampled data. Intuitively, if a process population has a high variance, and thus a large minimum sample size for confidence and error constraints, one can reduce the sampling requirement by partitioning the population into lower-variance groups.

Consider the case where there are k behavioral equivalence classes, or *strata*, in a population of N processes, with sizes $N_1, N_2, ..., N_k$; means $\bar{Y}_1, \bar{Y}_1, ..., \bar{Y}_k$; and variances $S_1^2, S_2^2, ..., S_k^2$. Assume further that in the i^{th} stratum, one uses a sample size n_i, calculated with (8). $\bar{Y}$ can be estimated as $\bar{y}_{st} = \sum_{i=1}^{k} w_i \bar{y}_i$, using the strata sample means $\bar{y}_1, \bar{y}_1, ..., \bar{y}_k$.

The weights $w_i = N_i/N$ are simply the ratios of stratum sizes to total population size, and $\bar{y}_{st}$ is the stratified sample mean. This is more efficient than $\bar{y}$ when:

$$\sum_{i=1}^{k} N_i(\bar{Y}_i - \bar{Y})^2 > \frac{1}{N} \sum_{i=1}^{k} (N - N_i) S_i^2. \tag{9}$$

In other words, when the variance between strata is significantly higher than the variance within strata, stratified sampling can reduce the number of processes that must be sampled to estimate the stratified sample means. For performance analysis, stratification gives insight to the structure of processes in a running application. The stratified sample means provide us with measures of the behavioral properties of separate groups of processes, and an engineer can use this information to assess the performance of his code.

3 The AMPL Library

The use of sampling to estimate scalar properties of populations of processes has been studied before [12]. We have built the Adaptive Monitoring and Profiling Library (AMPL), which uses the analysis described in §2 as a heuristic to sample arbitrary event traces at runtime.

AMPL collects and aggregates summary statistics from each process in a running parallel application. Using the variance of the sampled measurements, it calculates a minimum sample size as described in §2. AMPL dynamically monitors variance and it periodically updates sample size to fit the monitored data. This sampling can be performed globally, across all running processes, or the user can specify groups of processes to be sampled independently.

3.1 AMPL Architecture

The AMPL runtime is divided functionally into two components: a central client and per-process monitoring

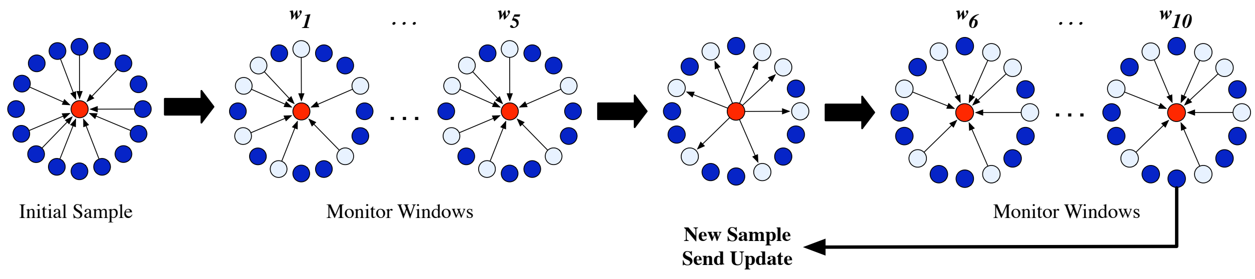


Figure 1. AMPL Runtime Sampling. Client process is at center, sampled processes are in white, and unsampled processes are dark. Arrows show communication; sample intervals are denoted by w_i.

agents. Agents selectively enable and disable an external trace library. The monitored execution is divided into a sequence of *update intervals*. Within each update interval is a sequence of data collection *windows*. The agents enable or disable collection for an entire window. They also accumulate summary data across the entire update interval and they send the data to the client at the end of the interval. The client then calculates a new sample size based on the variance of the monitored data, randomly selects a new sample set, and sends an update to monitored nodes. A monitoring agent receives this update and adopts the new sampling policy for the duration of the interval This process repeats until the monitored application's execution completes. Figure 1 shows the phases of this cycle in detail.

Interaction between the client and agents enables AMPL to adapt to changing variance in measured performance data. The user can configure which points in the code are used to determine AMPL's windows, the number of windows between updates from the client, and confidence and error bounds for the adaptive monitoring. As discussed in §2.3, these confidence and error bounds also affect the volume of collected data, giving AMPL an adaptive control to either increase accuracy or decrease trace volume and I/O overhead. Thus, traces using AMPL can be tuned to match the bandwidth restrictions of its host system.

An AMPL user can also elect to monitor subgroups of an application's processes separately. Per-group monitoring is similar to the global monitoring described here.

3.2 Modular Communication

AMPL is organized into layers. Initially, we implemented a communication layer in MPI, for close integration with the scientific codes AMPL was designed to monitor. AMPL is not tied to MPI, and we have implemented the communication layer modularly to allow for integration with other libraries and protocols. Client-to-agent sampling updates and agent-to-client data transport can be specified independently. Figure 2 shows the communication layer in

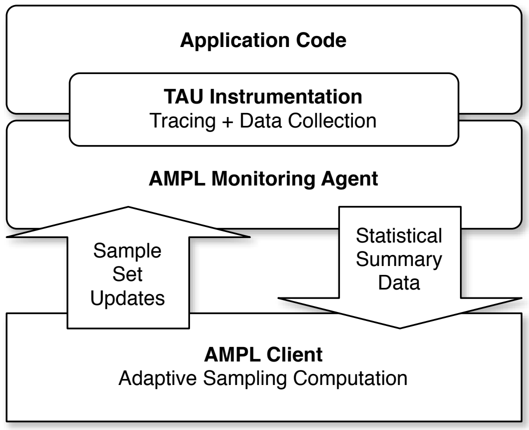


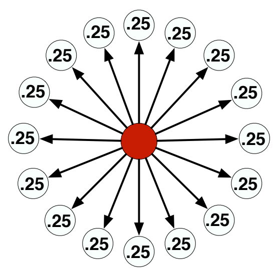
Figure 2. AMPL Software Architecture

the context of AMPLs high-level architectural design.

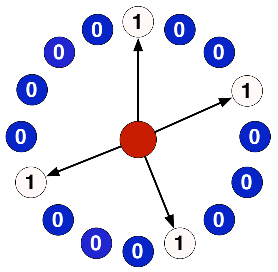
It is up to the user of AMPL to set the policy for the implementation of the random sampling of monitored processes. When the client requests that a population's sample set be updated, it only specifies the number, m, of processes in the population of M that should be monitored, not their specific ranks. The update mechanism sends to each agent a probability within $[0..1]$ that determines with what probability the agent enables data collection in its process.

We provide two standard update mechanisms. The *subset* update mechanism selects a fixed sample set of processes that will report at each window until the next update. The processes in this subset are instructed to collect data with probability 1; all other processes receive 0. This gives consistency between windows, but may accumulate sample bias if the number of windows per update interval is set too large. The *global* update policy uniformly sends m/M to each agent. Thus, in each window the expected number of agents that will collect data will be m. This makes for more random sampling at the cost of consistency. It also requires that all agents receive the update.

The desirability of each of our update policies depends



(a) Global



(b) Subset

Figure 3. Update mechanisms. Outer circles are monitored processes, labeled by probability of recording trace data. Client is shown at center.

on two factors: (a) the efficiency of the primitives available for global communication and (b) the need for multiple samples over several time windows from the same subset of the processes. To produce a simple statistical characterization of system or application behavior, global update has the advantage that its samples are truly random. However, if one desires performance data from the same nodes for a long period (*e*.g., to compute a performance profile for each sampled node), the subset update mechanism is needed. Figure 3 illustrates these policies.

3.3 Tool Integration

AMPL is written in C++, and it is designed to accept data from existing data collection tools. It provides C and C++ bindings for its external interface, and it can label performance events either by simple integer identifiers or by callpaths. Multiple performance metrics can be monitored simultaneously, so that data gathered from hardware performance counter APIs like PAPI [2] can be recorded along with timing information.

AMPL contains no measurement or tracing tools of its own. We integrated AMPL with the University of Oregon's Tuning and Analysis Utilities (TAU) [20], a widely used source-instrumentation toolkit for many languages, including C, C++, and FORTRAN. TAU uses PAPI and various timer libraries as data sources. We modified TAU's profiler to pass summary performance data to AMPL for online monitoring. The integration of AMPL with TAU required only a few hundred lines of code and slight modifications so that TAU could dynamically enable and disable tracing under AMPL's direction. Other tracing and profiling tools could be integrated with a similar level of effort.

AMPL is intended to be used on very large systems such as IBM's Blue Gene/L [5], Cray's XT3 [21] and Linux clusters [9, 6]. As such, we designed its routines to be

```
WindowsPerUpdate = 4
UpdateMechanism = Subset

EpochMarker = "TIMESTEP"

Metrics {
    "WALL_CLOCK"      Report
    "PAPI_FP_INS"     Guide
}

Group {
    Name = "Adaptive"
    Members = 0-127
    Confidence = .90
    Error = .03
}

Group {
    Name = "Static"
    SampleSize = 30
    Members = 128-255
    PinnedNodes = 128-137
}
```

Figure 4. AMPL Configuration File

called from within source-level instrumentation, as compute nodes on architectures like BlueGene do not currently support multiple processes, threads, or any other OS-level concurrency. All analyses and communication of data are driven by calls to AMPL's data-collection hooks.

3.4 Usage

To monitor an application, an analyst first compiles the application using the AMPL-enabled TAU. This automatically links the resulting executable with our library. AMPL runtime configuration and sampling parameters can be adjusted using a configuration file. See Figure 4.

This configuration file uses the `TIMESTEP` procedure to delineate sample windows. During the execution of `TIMESTEP`, summary data is collected from monitored processes. The system adaptively updates the sample size every 4 windows, based on the variance of data collected in the intervening windows. Subset sampling is used to send updates.

The user has specified two groups, each to be sampled independently. The first group, labeled `Adaptive`, consists of the first 128 processes. This group's sample size will be recalculated dynamically to yield a confidence of 90% and error of 3%, based on the variance of floating-point instruction counts. Wall-clock times of instrumented routines will be reported but not guaranteed within confidence or error bounds.

The explicit SampleSize directive causes the second group to be monitored statically. AMPL will monitor exactly 30 processes from the second 128 processes in the job. The PinnedNodes directive tells AMPL that nodes 128 through 137 should always be included in the sample set, with the remaining 20 randomly chosen from the group's members. Fine-grained control over adaptation policies for particular call sites is also provided, and this can be specified in a separate file.

4 Experimental Results

To assess the performance of the AMPL library and its efficacy in reducing monitoring overhead and data volume, we conducted a series of experiments using three well-known scientific applications. Here, we describe our tests. Our environment is covered in §4.1 - §4.2. We measure the cost of exhaustive tracing in §4.3, and in §4.4 , we verify the accuracy of AMPL's measurement using a small-scale test. In §4.5 - §4.7, we measure AMPL's overhead at larger scales. We provide results varying sampling parameters and system size. Finally, we use clustering techniques to find strata in applications, and we show how stratified sampling can be used to further reduce monitoring overhead.

4.1 Setup

Our experiments were conducted on two systems. The first is an IBM Blue Gene/L system with 2048 dual-core, 700 MHz PowerPC compute nodes. Each node has 1 GB RAM (512 MB per core). The interconnect consists of a 3-D torus network and two tree-strucutured networks. On this particular system there is one I/O node per 32 compute nodes. I/O nodes are connected via gigabit ethernet to a switch, and the switch is connected via 8 links to an 8-node file server cluster using IBM's General Parallel File System (GPFS). All our experiments were done in a file system fronted by two servers. We used IBMs xlC compilers and IBM's MPI implementation.

Our second system is a Linux cluster with 64 dual-processor, dual-core Intel Woodcrest nodes. There are a total of 256 cores, each running at 2.6 GHz. Each node has 4 GB RAM, and Infiniband 4X is the primary interconnect. The system uses NFS for the shared file system, with an Infiniband switch connected to the NFS server by four channel-bonded gigabit links. We used the Intel compilers and OpenMPI. OpenMPI was configured to use Infiniband for communication between nodes and shared memory within a node.

4.2 Applications

We used the following three scientific applications to test our library.

sPPM. ASCI sPPM [1] is a gas dynamics benchmark designed to mimic the behavior of classified codes run at Department of Energy national laboratories. sPPM is part of the ASCI Purple suite of applications, and is written in Fortran 77. The sPPM algorithm solves a 3-D gas dynamics problem on a uniform Cartesian mesh. The problem is statically divided (i.e., each node is allocated its own portion of the mesh), and this allocation does not change during execution. Thus, computational load on sPPM processes is typically well-balanced because each processor is allocated exactly the same amount of work.

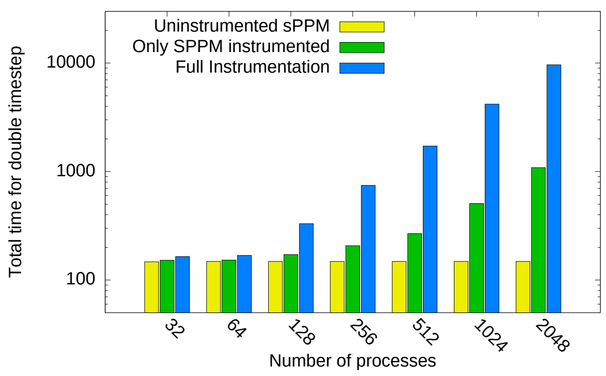
ADCIRC. The Advanced Circulation Model (ADCIRC) is a finite-element hydrodynamic model for coastal regions [11]. It is currently used in the design of levees and for predicting storm-surge inundation caused by hurricanes. It is written in Fortran 77. ADCIRC requires its input mesh to be pre-partitioned using the METIS [7] library. Static partitioning with METIS can result in load imbalances at runtime, and, as such, behavior across ADCIRC processes can be more variable than that of sPPM.

Chombo. Chombo[3] is a library for block-structured adaptive mesh refinement (AMR). It is used to solve a broad range of partial differential equations, particularly for problems involving many spatial scales or highly localized behavior. Chombo provides C++ classes and data structures for building adaptively refined grids. The Chombo package includes a Godunov solver application[4] for modeling magnetohydrodynamics in explosions. Our tests were conducted using this application and the explosion input set provided with it.

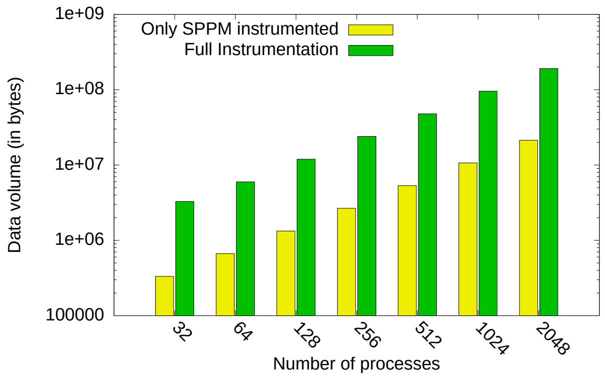
4.3 Exhaustive Monitoring

We ran several tests using sPPM on BlueGene/L to measure the costs of exhaustive tracing. First, we ran sPPM uninstrumented and unmodified for process counts from 32 to 2048. Next, to assess worst-case tracing overhead, we instrumented *all* functions in sPPM with TAU and ran the same set of tests with tracing enabled. In trace mode, TAU records timestamps for function entries and exits, as well as runtime information about MPI messages. Because performance engineers do not typically instrument every function in a code, we ran the same set of tests with only the SPPM and RUNHYD subroutines instrumented.

Figure 5(a) shows timings for each of our traced runs. It is clear from the figure that trace monitoring overhead



(a) Timings.



(b) Data volume.

Figure 5. Data volume and timing for sPPM on Blue Gene/L using varied instrumentation

scales linearly with the number of processes after 128 processes.

Figure 5(b) shows the data volume for the traced runs. As expected, data volume increases linearly with the number of monitored processes. For runs with only sPPM instrumented, approximately 11 megabytes data were produced per process, per double-timestep. For exhaustive instrumentation, each process generated 92 megabytes of data. For 2048 processes, this amounts to 183 gigabytes of data for just two timesteps of the application. Extrapolating linearly, a full two-step trace on a system the size of BlueGene/L at LLNL would consume 6 terabytes.

4.4 Sample Accuracy

AMPL uses the techniques described in §2 as a heuristic for the guided sampling of vector-valued event traces. Since we showed in §4.3 that it is impossible to collect an exhaustive trace from all nodes in a cluster without severe perturbation, we ran the verification experiments at small scale.

As before, we used TAU to instrument the `SPPM` and `RUNHYD` subroutines of sPPM. We measured the elapsed time of `SPPM`, and we used the return from `RUNHYD` to delineate windows. `RUNHYD` contains the control logic for each double-timestep that sPPM executes, so this is roughly equivalent to sampling AMPL windows every two timesteps.

We ran SPPM on 32 processes of our Woodcrest cluster with AMPL tracing enabled and with confidence and error bounds set to 90% and 8%, respectively. To avoid the extreme perturbation that occurs when the I/O system is saturated, we ran with only one active CPU per node and we recorded trace data to the local disk on each node. Instead of disabling tracing on unsampled nodes, we recorded full trace data from 32 processes, and we marked the sample set for each window of the run. This way, we know which subset of the exhaustive data would have been collected by AMPL, and we can compare the measured trace to a full trace of the application. Our exhaustive traces were 20 total timesteps long, and required a total of 29 gigabytes of disk space for all 32 processes.

Measuring trace similarity is not straightforward, so we used a generalization of the confidence measure to evaluate our sampling. We modeled each collected trace as a polyline, as per Lu and Reed in [10], with each point on the line representing the value being measured. In this case, this is the time taken by one invocation of the SPPM subroutine.

Let $p_i(t)$ be the event traces collected from each process in the system. We define the *mean trace* for M processes, $\bar{p}(t)$ to be:

$$\bar{p}(t) = \frac{1}{M} \int p_0(t) + p_1(t) + ... + p_M(t) dt$$

We define the *trace confidence*, c_{trace}, for a given run to be the percentage of time the mean trace of sampled processes, $\bar{p}_s(t)$ is within an error bound, d, of the mean trace over all processes, $\bar{p}_{exh}(t)$,

$$c_{trace} = \frac{1}{T} \int_0^T X(t) dt$$

$$X(t) = \begin{cases} 1 & \text{if } err(t) > d, \\ 0 & \text{if } err(t) \leq d. \end{cases}, \quad err(t) = \left| \frac{\bar{p}_s(t) - \bar{p}_{exh}(t)}{\bar{p}_{exh}(t)} \right|$$

where T is the total time taken by the run.

We calculated c_{trace} for the full set of 32 monitored processes and for the samples that AMPL recommended. Figure 6 shows the first two seconds of the trace. $\bar{p}_{exh}(t)$ is shown in black, with $\bar{p}_s(t)$ superimposed in gray. The shaded region shows the error bound around $\bar{p}_{exh}(t)$. Actual error is shown at bottom. For the first two seconds of

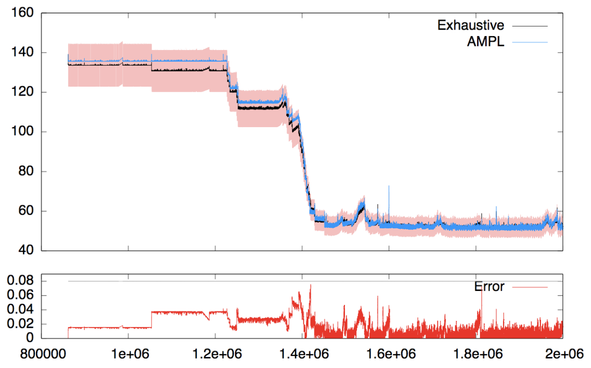


Figure 6. Mean trace (black) and sample mean trace (blue) for two seconds of a run of sPPM on a 32-node Woodcrest system.

the trace, the sampled portion is entirely within the error bound.

We measured the error for all 20 timesteps of our sPPM run, and we calculated c_{trace} to be 95.499% for our error bound of 8%. This is actually *better* than the confidence bound we specified for AMPL. We can attribute this high confidence to the fact that AMPL intentionally oversamples when it predicts very small samples (10 or fewer processs), and to sPPM's general lack of inter-node variability.

4.5 Data Volume and Runtime Overhead

We measured AMPL overhead as a function of sampling parameters. As in §4.4, we compiled all of our test applications to trace with TAU, and we enabled AMPL for all runs. In these experiments with a fixed number of processors, we varied confidence and error constraints from 90% confidence and 8% error at the lowest to exhaustive monitoring (100% confidence and 0% error). Only those processes in the sample set wrote trace data. Processes disabled by AMPL did not write trace data to disk until selected for tracing again.

For both sPPM and ADCIRC, we ran with 2048 processes on our BlueGene/L system. We ran Chombo with 128 processes on our smaller Woodcrest cluster.

Instrumentation

The routines instrumented varied from code to code, but we attempted to choose routines that would yield useful metrics for performance tuning. For sPPM, we instrumented the main timestep loop and the `SPPM` routine, and we measured elapsed time for each. Sample updates were set for every 2 time steps, and we ran a total of 20 time steps.

For ADCIRC, we added instrumentation only to the `TIME-STEP` routine and MPI calls. ADCIRC sends MPI messages frequently and its time step is much shorter than sPPM, so we set AMPL's window to 800 ADCIRC timesteps, and we used the mean time taken for calls to `MPI_Waitsome()` during each window to guide our sampling. `MPI_Waitsome()` is a good measure of load balance, as a large value indicates that a process is idle and waiting on others.

For Chombo, we instrumented the coarse timestep loop in the Godunov solver. This timestep loop is fixed-length, though the timestep routine subcycles smaller time steps when they are necessary to minimize error. Thus, the number of floating point instructions per timestep can vary. We used PAPI[2] to measure the number of floating point instructions per coarse timestep, and we set AMPL to guide the sample size using on this metric.

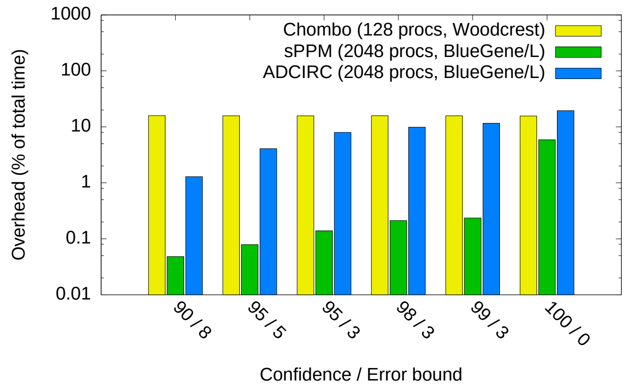
Discussion

Figures 7(a) and 7(b) show the measured time and data overhead, respectively. Total data volume scales linearly with the total processes in the system. In the presence of an I/O bottleneck, total time scales with the data volume, The experiments illustrate that AMPL is able to reduce both.

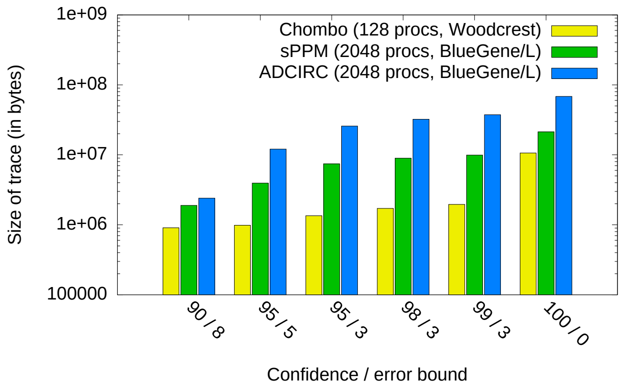
The elapsed time of the sPPM routine varies little between processes in the running application. Hence, overhead for monitoring `SPPM` with AMPL is orders of magnitude smaller than the overhead of monitoring exhaustively. For 90% confidence and 8% error, monitoring a full 2048-node run of sPPM adds only 5% to the total time of an uninstrumented run. For both 99% confidence and 3% error, and 95% confidence and 5% error, overheads were 8%. In fact, for each of these runs, all windows but the first have sample sizes of only 10 out of 2048 processes. Moreover, AMPL's initial estimate for sample size is conservative. It chooses the worst-case sample size for the requested confidence and error, which can exceed the capacity of the I/O bottleneck on BlueGene/L. If these runs were extended past 20 time windows, all of the overhead in Figure 7 would be amortized over the run.

For large runs, AMPL can reduce the amount of collected data by more than an order of magnitude. With a 90% confidence interval and 8% error tolerance, AMPL collects only 1.9 GB of performance data for 2048 processes, while sampling all processes would require over 21 GB of space. Even with a 99% confidence interval and a 3% error bound, AMPL never collects more than half as much data from sPPM as would exhaustive tracing techniques.

As sampling constraints are varied, the time overhead results for ADCIRC changes are similar to the sPPM results. Using AMPL, total time for 90% confidence and 8% error is over an order of magnitude less than that of exhaustive monitoring. Data reduction is even greater. Using AMPL with 90% confidence and 8% error, data volume for AD-



(a) Percent of total execution time.



(b) Output data volume.

Figure 7. AMPL trace overhead for three applications, varying confidence and error bounds.

CIRC is 28 times smaller than an exhaustive trace.

Data overhead for ADCIRC is higher than for sPPM because ADCIRC is more sensitive to instrumentation. An uninstrumented run with 2048 processes takes only 155 seconds, but the shortest instrumented ADCIRC run took 355 seconds. With sPPM, we did not see this degree of perturbation. In both cases, we instrumented only key routines, but ADCIRC makes more frequent MPI calls and is more sensitive to TAU's MPI wrapper library. Since sPPM makes less frequent MPI calls, its running time is less perturbed by instrumentation.

For Chombo, the Woodcrest cluster's file system was able to handle the smaller load of 128 processor traces well for all our tests, and we do not see the degree of I/O serialization that was present on our BlueGene/L runs. There was no significant variation in the runtimes of the Chombo runs with AMPL, and even running with exhaustive tracing took about the same amount of time. However, the overhead of instrumentation in Chombo was high. In general, instrumented runs took approximately 15 times longer due to the large number of MPI calls Chombo makes. This overhead could be reduced if we removed instrumentation for more frequently invoked MPI calls.

Data overhead for the 128-processor Chombo runs scales similarly to trace volume for ADCIRC and SPPM. Compared to exhaustive monitoring, we were able to reduce trace data volume by 15 times compared to an exhaustively traced run with 128 processes. As with both ADCIRC and sPPM, the sample size and data volume both climb gradually as we tighten the confidence and error bounds.

4.6 Projected Overhead at Scale

§4.5 shows that AMPL's overhead can be tuned with user-defined sampling parameters, and it illustrates that AMPL can effectively improve trace overhead on relatively small systems. However, we would like to know how AMPL will perform on even larger machines.

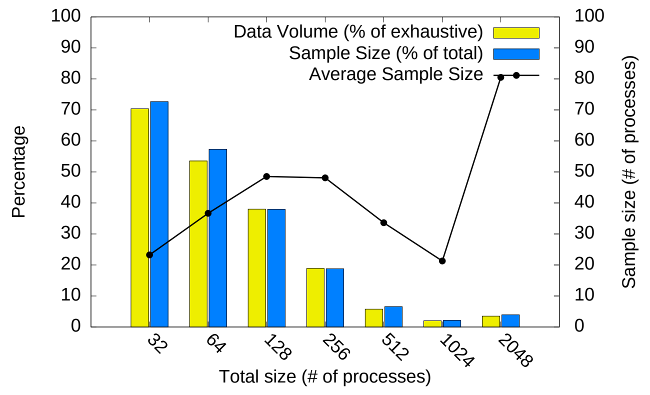
We configured ADCIRC with TAU and AMPL as in §4.5, but in these experiments we fixed the sampling parameters and varied only the process count. Figure 8(a) shows the sample size and the amount of data collected by AMPL for each run expressed as a fraction of the exhaustive case. Although the sample size increases as the system grows, the relative data volume decreases. For 2048 nodes, total volume is less than 10 percent of the exhaustive case. As mentioned in §2, sample size is constant in the limit, so we can expect this curve to level off after 2048 processes, and we can expect very large systems to require that increasingly smaller subsets of processes be sampled.

Figure 8(b) shows sample size and data volume for Chombo tracing as system size is increased. As with ADCIRC, we see that the fraction of data collected decreases for larger systems, as the minimum required sample size scales more slowly than the size of the system. For larger systems, we can thus expect to see comparatively smaller overhead and trace volume, just as we did for ADCIRC and sPPM.

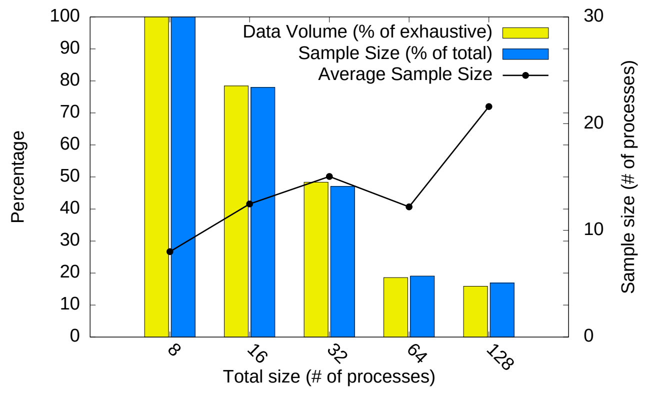
4.7 Stratification

We now turn to an examination of AMPL's performance with stratification. As discussed in §2.3, if we know which groups of processes will behave similarly, we can sample each group independently and, in theory, reduce data volume. We use simple clustering algorithms to find groups in our performance data, and we observe further reductions in data volume achieved by using these groups to stratify samples taken with AMPL runs.

Clustering algorithms find sets of elements in their input data with minimal *dissimilarity*. Here we used a well known algorithm, k-medoids[8], to analyze summary data

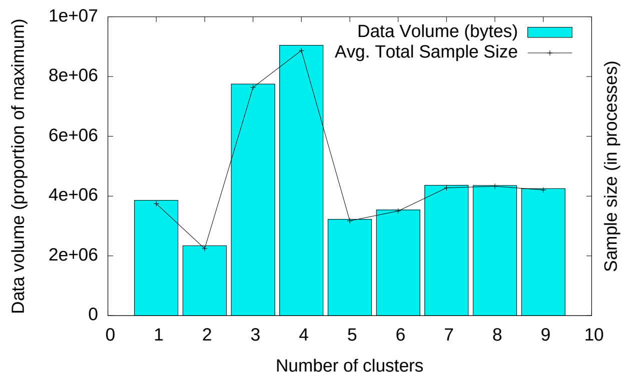


(a) Data volume for ADCIRC on BlueGene/L.

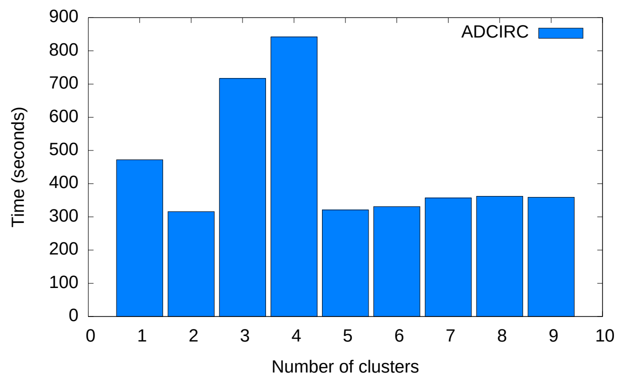


(b) Data volume for Chombo on our Woodcrest with Infiniband.

Figure 8. Scaling runs of Chombo and ADCIRC on two different machines. Note that in both instances, data volume and sample size decrease proportionally, but sample size increases absolutely.



(a) Data overhead and average total sample size.



(b) Percent total execution time.

Figure 9. Time and data volume vs. clusters for stratified ADCIRC trace, measuring MPI_Waitsome().

from ADCIRC and to subdivide the population of processes into groups.

K-medoids requires that the user have a dissimilarity metric to compare the elements being clustered, and that the user specify k, the number of clusters to find. In our experiment, the elements are ADCIRC processes. Computing dissimilarity is slightly less straightforward. We configured ADCIRC as in §4.6 and recorded summary data for all 1024 processes in the run. After each window, processes the mean time for all calls to `MPI_Waitsome()` to a local log file. The logged data thus consisted of time-varying vectors of local means. For simplicity, we used the Euclidian distance between these vectors as our similarity measure.

We ran k-medoids on our 1024-process ADCIRC data for cluster counts from 1 to 10, and we used the output to construct a stratified AMPL configuration file. We then reran ADCIRC with stratified sampling for each of these configuration files. The single-cluster case is identical to the non-stratified runs above. Figures 9(a) and 9(b) show data overhead for different cluster counts and execution time, respectively.

Figure 9(a) shows that a further 25% reduction in data volume was achieved on 1024-node runs of ADCIRC by stratifying the population into two clusters. On the BlueGene/L, we see a 37% reduction in execution time. Surprisingly, for $k = 3$ and $k = 4$, dividing the population actually causes a significant increase in both data volume and execution time, while 5-10 clusters perform similarly to the 2-cluster case. Since tracing itself can perturb a running application, it may be that we are clustering on perturbation noise for the 3 and 4-cluster cases. This is a likely culprit for the increases in overhead we see here, as our samples are chosen randomly and the balance of perturbation across nodes in the system is nondeterministic. Because we used off-line clustering we do not accurately capture this kind of dynamic behavior. This could be improved by using an online clustering algorithm and adjusting the strata dynamically.

For the remainder of the tests, our results are consistent with our expectations. With 5-clusters, the AMPL-enabled run of ADCIRC behaves similarly to the 2-cluster case. Data and time overhead of subsequent tests with more stratification gradually increase, but they do not come close to exceeding the overhead of the 2-cluster test. There is, however, a slow rise in overhead from 5 clusters and on, and this can be explained by one of the weaknesses of k-medoids clustering. K-medoids requires that the user to specify k in advance, but the user may have no idea how many equivalence classes actually exist in the data. Thus, the user can either guess, or he can run k-medoids many times before finding an optimal clustering. If k exceeds the actual number groups that exist in the data, the algorithm can begin to cluster on noise.

5 Related Work

Mendes *et al.* [12] used statistical sampling to monitor scalar properties of large systems. We have taken these methods and applied them to monitoring trace data at the application level, while Mendes measures higher-level system properties at the cluster and grid level.

Noeth *et al.* [15] have developed a scalable, lossless MPI trace framework capable of reducing data volumes of large MPI traces for very regular applications by orders of magnitude. This framework collects compressed data at runtime and aggregates it at exit time via a global trace-reduction operation. The approach does not support the collection of arbitrary numerical performance data from remote processes; only an MPI event trace is preserved. In contrast, AMPL can measure a trace of arbitrary numerical values over the course of a run, and is more suited to collecting application-level performance data and timings. We believe that the two approaches are complementary, and that Noeth's scalable MPI traces could be annotated intelligently with the sort of data AMPL collects based on information learned by AMPL at runtime. We are considering this for future work.

Roth *et al.*have developed MRNet [17] as part of the ParaDyn project [18]. MRNet uses tree-based overlay networks to aggregate performance data in very large clusters, and it has been used at Lawrence Livermore Laboratory with Blue Gene/L. MRNet allows a random sample to be taken from monitored nodes, but it does not guide the sampling or handle reductions of trace data directly. AMPL could benefit by using MRNet to restructure communication operations as reduction trees separate from the monitored application, in place of the embedded PMPI calls it currently uses. Such a modification could potentially reduce the perturbation problems we saw in our experiments with stratification.

Nikolayev *et al.* [14] have used statistical clustering to reduce data overhead in large applications by sampling only representatives from clusters detected at runtime. The approach is entirely online: clusters are generated on-the-fly based on data observed at runtime. Nikolayev's method assumes that one representative from each cluster will be enough to approximate its behavior, and it does not offer the statistical guarantees that AMPL provides. AMPL could, however, benefit from the sort of online clustering done in this work.

6 Conclusions and Future Work

We have shown that the techniques used in AMPL can reduce the data overhead and the execution time of instrumented scientific applications by over an order of magnitude on small systems. Since the overhead of our monitoring methods scales sub-linearly with the number of concurrent processes in a system, AMPL, or similar sampling frameworks, will be for monitoring petascale machines when they arrive. Further, we have shown that, in addition to estimating global, aggregate quantities across a large cluster, populations of processes can be stratified and monitored as such with our tool. This will allow for further reductions in data volume and execution time.

The ideas presented here are implemented in a library that is easily integrated with existing tools. We were able to integrate AMPL easily with the TAU performance toolkit, and we believe that our techniques are widely applicable to many other domains of monitoring.

Building on this work, we are currently developing more sophisticated performance models and low-overhead, online analysis of distributed applications. We plan to apply the techniques we have developed to other monitoring tools, and to use them to facilitate online, real-time analysis of scalable applications at the petascale and beyond.

References

[1] ASCI. The ASCI Purple sPPM benchmark code [online]. 2002. Available from: http://www.llnl.gov/asci/platforms/purple/rfp/benchmarks/limited/sppm.

[2] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. J. Mucci. A portable programming interface for performance evaluation on modern processors. *The International Journal of High Performance Computing Applications*, 14(3):189–204, Fall 2000.

[3] P. Colella, D. T. Graves, D. Modiano, D. B. Serafini, and B. v. Straalen. Chombo software package

for AMR applications. *Technical Report (Lawrence Berkeley National Laboratory)*, 2000. Available from: http://seesar.lbl.gov/anag/chombo.

[4] R. Crockett, P. Colella, R. Fisher, R. I. Klein, and C. McKee. An unsplit, cell-centered Godunov method for ideal mhd. *Journal of Computional Physics*, Vol.203:422–448, 2005.

[5] A. Gara, M. A. Blumrich, D. Chen, G. L.-T. Chiu, P. Coteus, M. E. Giampapa, R. A. Haring, P. Heidelberger, D. Hoenicke, G. V. Kopcsay, T. A. Liebsch, M. Ohmacht, B. D. Steinmacher-Burow, T. Takken, and P. Vranas. Overview of the Blue Gene/L system architecture. *IBM Journal of Research and Development*, 49(2/3), 2005.

[6] IBM. MareNostrum: A new concept in linux supercomputing [online]. February 15 2005. Available from: http://www-128.ibm.com/developerworks/library/pa-nl3-marenostrum.html.

[7] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing*, 20(1):359–392, 1999.

[8] L. Kaufman and P. J. Rousseeuw. *Finding Groups in Data: An Introduction to Cluster Analysis*. Wiley Series in Probability and Statistics. Wiley-Interscience, 2nd edition, 2005.

[9] Lawrence Livermore National Laboratory. Livermore computing resources. [online]. 2007. Available from: http://www.llnl.gov/computing/hpc/resources/OCF_resources.html.

[10] C.-d. Lu and D. A. Reed. Compact application signatures for parallel and distributed scientific codes, November, 2002 2002.

[11] R. Luettich, J. Westerink, and N. Scheffner. ADCIRC: an advanced three-dimensional circulation model for shelves coasts and estuaries, Report 1: theory and methodology of ADCIRC-2DDI and ADCIRC-3DL, 1992 1992.

[12] C. L. Mendes and D. A. Reed. Monitoring large systems via statistical sampling. *International Journal of High Performance Computing Applications*, 18(2):267–277, 2004.

[13] H. Meuer, E. Strohmaier, J. Dongarra, and S. Horst. Top500 supercomputer sites [online]. Available from: http://www.top500.org.

[14] O. Y. Nikolayev, P. C. Roth, and D. A. Reed. Real-time statistical clustering for event trace reduction. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(2):144–159, 1997.

[15] M. Noeth, F. Mueller, M. Schulz, and B. R. de Supinski. Scalable compression and replay of communication traces in massively parallel environments. In *International Parallel and Distributed Processing Symposium (IPDPS)*, March 26-30 2007.

[16] R. Ross, J. Moreira, K. Cupps, and W. Pfeiffer. Parallel I/O on the IBM Blue Gene/L system. *Blue Gene/L Consortium Quarterly Newsletter*, First Quarter, 2006. Available from: http://www-fp.mcs.anl.gov/bgconsortium/file%20system%20newsletter2.pdf.

[17] P. C. Roth, D. C. Arnold, and B. P. Miller. MRNet: A software-based multicast/reduction network for scalable tools. In *Supercomputing 2003 (SC03)*, 2003.

[18] P. C. Roth and B. P. Miller. On-line automated performance diagnosis on thousands of processors. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'06)*, New York City, 2006.

[19] R. L. Schaeffer, W. Mendenhall, and R. L. Ott. *Elementary Survey Sampling*. Wadsworth Publishing Co., Belmont, CA, 6th edition, 2006.

[20] S. Shende and A. Maloney. The TAU parallel performance system. *International Journal of High Performance Computing Applications*, 20(2):287–331, 2006.

[21] J. S. Vetter, S. R. Alam, T. H. Dunigan, Jr., M. R. Fahey, P. C. Roth, and P. H. Worley. Early evaluation of the Cray XT3. In *Proc. 20th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2006.