An Open Infrastructure for Scalable, Reconfigurable Analysis*

Bronis R. de Supinski * bronis@llnl.gov
Rob Fowler ‡ rjf@renci.org
Todd Gamblin ‡ tgamblin@cs.unc.edu
Frank Mueller † mueller@csc.ncsu.edu
Prasun Ratn † pratn@ncsu.edu
Martin Schulz * schulzm@llnl.gov

*Lawrence Livermore National Laboratory
†North Carolina State University
‡Renaissance Computing Institute, University of North Carolina at Chapel Hill

Abstract

Petascale systems will have hundreds of thousands of processor cores so their applications must be massively parallel. Effective use of petascale systems will require efficient interprocess communication through memory hierarchies and complex network topologies. Tools to collect and analyze detailed data about this communication would facilitate its optimization. However, several factors complicate tool design. First, large-scale runs on petascale systems will be a precious commodity, so scalable tools must have almost no overhead. Second, the volume of performance data from petascale runs could easily overwhelm hand analysis and, thus, tools must collect only data that is relevant to diagnosing performance problems. Analysis must be done *in-situ*, when available processing power is proportional to the data.

We describe a tool framework that overcomes these complications. Our approach allows application developers to combine existing techniques for measurement, analysis, and data aggregation to develop application-specific tools quickly. Dynamic configuration enables application developers to select exactly the measurements needed and generic components support scalable aggregation and analysis of this data with little additional effort.

1 Introduction

In recent years, the degree of concurrency in large-scale supercomputers has increased exponentially. The largest existing system, Blue Gene/L at Lawrence Livermore National Laboratory [5], has 212,992 processors, and future petascale machines are expected to have at least as many.

The performance of applications written for machines of this size will depend largely on developers' ability to manage communication through shared memory hierarchies and complex network topologies. Local performance will require application developers to share on-node resources,

*This work was supported in part by the DOE Office of Science SciDAC PERI (DE-FC02-06ER25764) and NSF grants CNS-0410203, CCF-0429653 and CAREER CCR-0237570. Part of this work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract DE-AC52-07NA27344.

while scalability will depend on efficient communication between nodes as well as eliminating impediments to parallelism such as serialization and load imbalance. System software, including performance and debugging tools, will also be constrained by these requirements. Hence, the design of tools for petascale systems must address two key challenges:

1. To address problems that only appear at extreme scale, tools must be usable on large-scale, long-running executions. Such resources will be reserved primarily for production jobs, so tools must have sufficiently low overhead to run in a production environment and must not require modification of the source code of monitored applications.
2. Since petascale runs could generate huge volumes of performance data, requiring excessive communication and storage resources, on-line data aggregation and analysis are essential.

We present a component architecture to address these challenges. On individual processes, measurement tools are combined as abstract data sources in a chain of components to transform and to analyze collected data. These components significantly reduce the volume of on-node data and then pass it to scalable aggregation components, which further reduce the collected data so its storage and eventual human analysis is feasible. Additionally, the aggregation components can perform on-line analysis. The key to our system is that these components are completely reconfigurable. Data sources can be chosen at runtime and generic transform, analysis, and aggregation components support scalable information collection from arbitrary sources.

Since our framework supports dynamic tool configuration, we allow application developers to combine precisely those components required to diagnose their particular performance problem. Tool users can monitor jobs at runtime and change tools based on their dynamic behavior if new quantities need to be measured. By measuring only what is needed and by combining gathered data on-node using transformation and analysis tools, we ensure that on-node overhead is not too steep for production jobs. We also eliminate the need to run jobs multiple times with different tools to collect different sets of performance data.

Scalable data aggregation tools factor into our framework as generic components similar to data sources and analyzers. Previous work has shown that scalable aggregation techniques such as tree-reductions [1, 12, 14] and sampling [4, 10] are essential. Rather than requiring users to rewrite interfaces to these components every time a new tool is built, our framework allows data sources to be connected to reduction networks ad-hoc, without additional programming.

The rest of this paper is organized as follows. We describe our component framework in §2. §3 and §4 detail our existing tools that fit within this model and present sample results from their use. We discuss using our framework for on-line identification of performance clusters in §5. Finally, we state our conclusions and outline future work in §6.

2 A Dynamically Reconfigurable Analysis Framework

Traditional performance analysis tools are insufficient for monitoring petascale applications. Tracing tools gather sufficient information to analyze almost all performance issues but generate prodigious data volumes, excessively perturb the monitored application and perform little or no online analysis. Conversely, profiling tools (*e.g.*, those in HPCToolkit [9], TAU [17] and Open|SpeedShop [18]) provide a compact, whole-run summary of application performance but

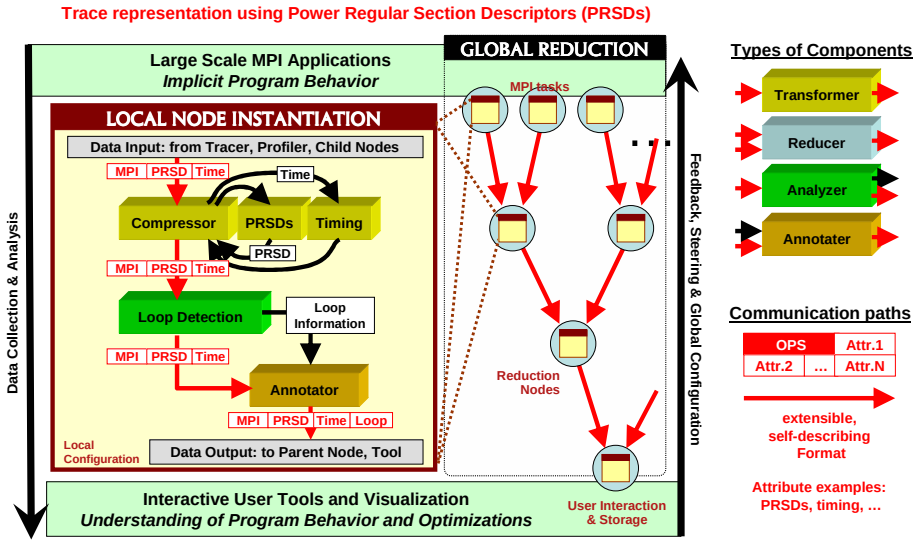


Figure 1: Interaction of Components

primarily make their aggregation decisions *a priori*, which often limits diagnosis of performance problems, particularly those that emerge over time due to evolutionary or cyclic behavior.

Our framework scalably and efficiently combines the features and advantages of both approaches while providing significant dynamic flexibility. We achieve this goal using a three-pronged approach. First, we employ efficient data compression that fundamentally limits data volumes without losing structural trace information. Second, we perform on-line analysis across a tree-based overlay network, which reduces the computational overhead of the analysis in addition to supporting our data aggregation. Most importantly, we enable the users to specify and to include only relevant analysis steps during a particular run or application phase by encapsulating all analysis routines in independent, reconfigurable components.

Our tool component architecture collects data from one or more data sources of the MPI application and then passes it through a tree-based reduction network to the front end tool, as illustrated on the right side of Figure 1. An instantiation of the analysis framework processes data at each leaf (MPI task) or internal reduction node through one or more user-provided analysis modules, as shown on the left side of the figure. We organize the data flowing between the analysis components, as well as the levels of the reduction tree, in an extensible and self describing data format that can be annotated with analysis-specific attributes (see PRSDs in Section 3).

Each component can interpret this format and transform its contents as well as change, add or remove attributes before passing the data stream to the next component. We classify the analysis components into four major operation-based groups: *Transformers* work on a single input stream and produce a single output stream; *Reducers* combine two or more input streams into a single output stream; *Analyzers* extract additional information for further processing; and *Annotators* add further annotations from analyzers or external sources to the data stream. Users can arbitrarily combine these components to target analysis precisely to their performance issues.

We decentralize module composition and data flow with local configuration modules. These mod-

ules identify required components through user input or data flow constraints, load the necessary modules and steer the data flow between them. The framework also supports dynamic global configuration from the front end or local analysis feedback, providing the flexibility to adapt quickly and autonomously to performance observations including application phases.

We implement this framework through existing infrastructure: $P^N MPI$ [15] virtualizes the PMPI profiling interface to support our dynamic tool adaptation and component interaction, and MRNet [14] provides tree-based overlay network functionality with custom reduction nodes. $P^N MPI$ provides a generic tool stack mechanism that supports dynamic configuration and control of all data sources within the application process, including non-MPI data sources, such as hardware counter measurements or application annotations as well as those that employ MPI tracing or profiling. We implement each data source as a separate PMPI module and then load them into the dynamically created MPI tool stack. Using $P^N MPI$ core services, the data is then forwarded into the analysis leaf processes and from there through the MRNet reduction nodes to the front end tool. Overall, our framework provides a flexible mechanism to tailor data collection, aggregation and analysis to large-scale performance issues.

3 ScalaTrace: Scalable Communication Trace Compression

ScalaTrace [12] is a scalable MPI tracing tool set that exploits the SPMD nature of MPI applications to extract highly compressed, full communication traces. ScalaTrace uses the PMPI interface to collect trace data, and we have integrated it as a data source module in our framework so that its traces can be dynamically analyzed by other tools. Compared to previous MPI tracing mechanisms, its traces are orders of magnitude smaller, and it often achieves near constant trace size, regardless of node count or application run length while preserving structural information and temporal event order. Thus, ScalaTrace achieves the data reduction needed for petascale analysis.

3.1 Compression Strategy

Our trace-gathering tool set intercepts MPI calls via the PMPI mechanism during application execution. $P^N MPI$ loads the tracing library and enables the tracing of MPI operations and significant parameters without recording the message content. This intra-node information (task-level) is compressed on-the-fly. We also perform inter-node compression upon application termination to obtain a single trace file that preserves structural information suitable for lossless replay. We describe the compression at a high level here; algorithmic details are available elsewhere [12].

```
for (i = 1; i < 1000; i++) {
  for (k = 1; k < 100; k++) {
    MPI_Send(...); /* send call 1 */
    MPI_Recv(...); /* recv call 1*/
  }
  MPI_Barrier(...); /* barrier call 1 */
}
```

Figure 2: Sample Code for PRSDs

We compress MPI call entries within each node on-the-fly. We use regular section descriptors (RSDs) to describe MPI events nested in a single loop in constant size [7] and power-RSDs (PRSDs) to specify recursive RSDs nested in multiple loops [8]. MPI events may occur at any level in PRSDs. For example, the tuple $RSD1 :< 100, MPI\_Send1, MPI\_Recv1 >$ denotes a loop with 100 iterations of alternating send/receive calls with identical parameters (omitted here),

and $PRSD1 :< 1000, RSD1, MPI\_Barrier1 >$ denotes 1000 invocations of that loop (RSD1) followed by a barrier. These constructs correspond to the code in Figure 2.

We perform generic and domain-specific optimizations in order to compress intra-node events efficiently. We identify calling sequences through a signature derived from a stack walk. Thus, the trace distinguishes routines by call site (*e.g.*, multiple `MPI_Send` uses). We identify request handles by a relative index into a dynamically updated handle buffer, which portably abstracts runtime-dependent data structures, (*e.g.*, for handles returned by asynchronous communication calls such as `MPI_Isend`). We aggregate iterative constructs with an indeterminate number of repetitions of a common event type, such as looping over `MPI_Waitsome` to complete n asynchronous events, into a single event that is abstracted as multiple calls.

ScalaTrace combines local traces into a single global trace upon application completion. Current work is converting this mechanism to use MRNet as the application run progresses. Our inter-node compression mechanism uses a tree-based overlay network to merge events and structures (RSD and PRSDs) of nodes step-wise and in a bottom-up fashion when events, parameters, structure and iteration counts match. This compression is essential for petascale tracing, as it avoids communication and storage requirements proportional to the job's MPI task count.

We employ further generic and domain-specific optimizations for inter-node compression (across tasks). We use location-independent encoding of communication end-points, which enables event merging, particularly for stencil-based codes. We encode sequences of task IDs as PRSDs, which allows a concise representation even for subsets of nodes. We also reorder events temporally when they originate from different nodes (with no causal relation) for a more concise representation.

3.2 Preserving Statistical Timing Information

The objective of trace analysis is to find inefficiencies in the code, *e.g.*, as indicated by load imbalance between nodes. Such analysis requires knowledge about the timing between events. Conventional trace techniques timestamp all communication events. However, our compression techniques would fail if each event included a timestamp since the times would necessarily be different. We have designed two methods to capture event timing information: a low-cost statistical recording of *delta times* (relative times between event pairs), and a variation-preserving recording scheme [13]. Both methods support constant size representations.

The first method collects aggregate statistics (maximum, minimum, average and standard deviation) of delta times and combines those statistics within PRSDs. Statistical aggregation is sufficient for approximate event replay that preserves relative timing at a coarse grain. It allows one to determine if significant load imbalances exist. However, it does not capture finer-grained inefficiencies.

The second method, which addresses the shortcomings of the first for fine grained analysis, uses size-limited histograms of delta times. Generally, a fixed number of bins suffices to gain more insight about computational imbalance. Our method divides the range of delta times associated with an event pair into k subrange bins (k constant). We increment counters corresponding to each subrange when a delta time in the subrange is observed. We have designed a dynamic algorithm based on weighted subrange partitioning to shift and to widen value ranges of bins with the objective to equalize bin frequencies. We track minimum and maximum values for each bin to provide finer-grain information suitable for trace-based performance analysis.

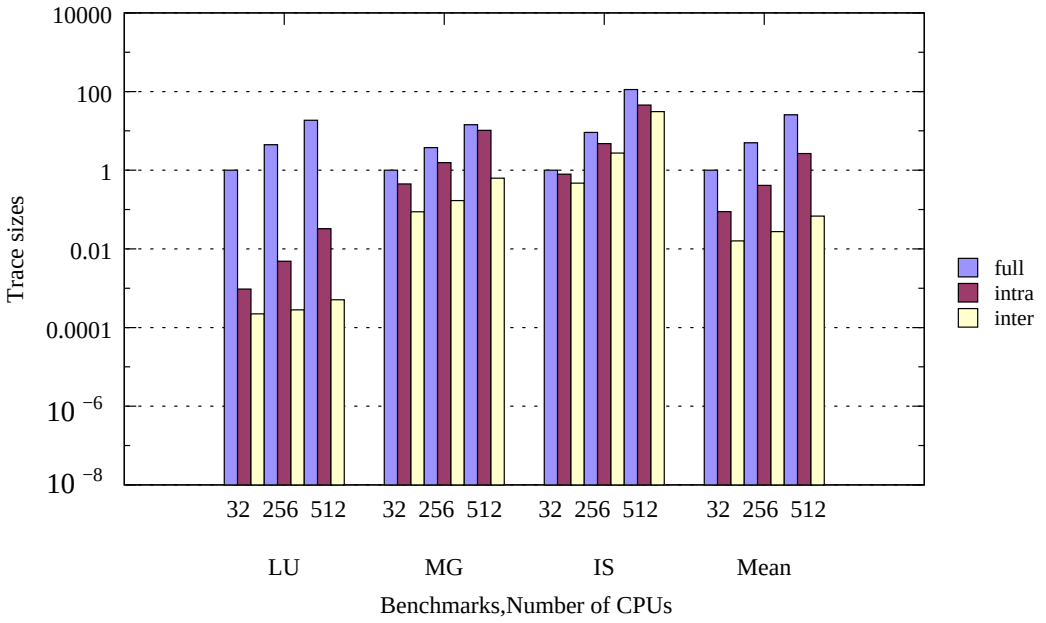


Figure 3: Representative and Mean Trace Sizes

3.3 ScalaTrace Results

ScalaTrace produces near-constant size traces for most of the NAS parallel benchmarks. Overall, our results fall into three classes based on trace size growth: near-constant size (DT, EP, LU, BT, FT), sub-linear (MG, CG) and non-scalable (IS). Figure 3 shows the normalized trace sizes for one benchmark from each class and also the geometric mean of the trace sizes across all benchmarks. The bars compare full (no compression), intra-node (i.e., local compression only) and inter-node traces. The poor IS scaling results from its use of `MPI_Alltoallv` with varying counts and displacements, thus adversely affecting the inter-node merge due to parameter mismatches. Overcoming this limitation is the subject of current research.

ScalaTrace supports generic replay of communication traces without the application code, suitable for fast network simulation. We have implemented a replay engine that interprets the compressed trace on-the-fly to issue communication calls in the same order they were issued by the application. In effect, the replay engine is a reversal of the compression algorithm. When it encounters an RSD or PRSD, it iteratively issues the communication calls recreating their structure, frequency and parameters, with random message sizes. Our structure-preserving compression enables scalable replay with very low memory requirements loosely bounded by the size of the *compressed* trace.

Using the replay engine, we conducted experiments to verify the correctness of our scalable compression for all NAS benchmarks based on aggregate wall-clock replay times. Figure 4 shows trace replay timings for three of the NAS parallel benchmarks. We achieve -9 to 14% accuracy for intra-node traces and -22 to 5% accuracy for inter-node traces (except MG). In the case of MG, the replay timing errors are higher since the communication time is significantly larger during replay than in the original application, which may be due to imbalances introduced during replay.

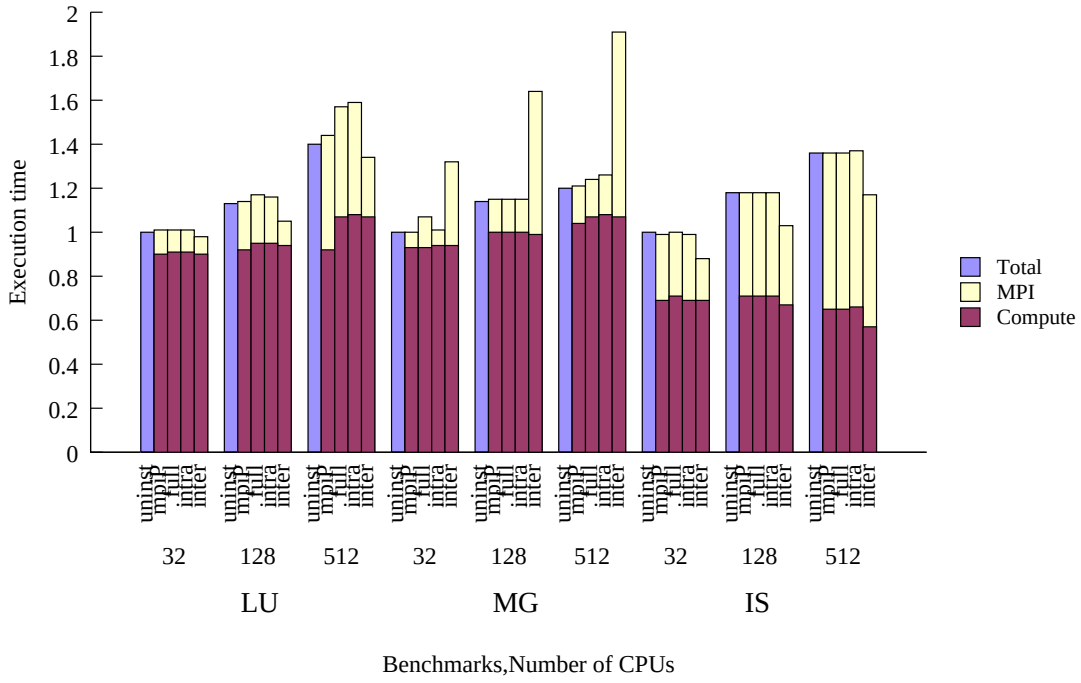


Figure 4: Representative replay timing results

4 Scalable Load-balance Monitoring

Measuring load-balance in large-scale systems is difficult because it requires that observations be recorded for all processes. However, monitoring schemes with data requirements that are linear in terms of the number of processes in the system will not scale. Compounding the problem, many scientific applications use data-dependent, adaptive algorithms and may redistribute load dynamically. To characterize the performance of these applications accurately, we must record the load for each process over time. Application developers need three pieces of information to diagnose load imbalance: *i)* where the problem occurs in the code; *ii)* on which processors the load is imbalanced; and *iii)* when the imbalance emerges during execution.

Using our framework, we have built a tool [3] for scalable, system-wide load-balance tracing using low-error compression techniques. We have developed an *Effort Model* for load-balance data, and this instantiation of the framework uses two modules in conjunction with ScalaTrace to extract and analyze model data. An *Analysis* module extracts the timing data necessary for the model from ScalaTrace's MPI event stream, and a lossy parallel wavelet compression module scalably reduces the volume of numerical load-balance data.

4.1 The Effort Model

We introduce the *Effort Model* to express load in terms of high-level application semantics. Our model measures two types of loops in SPMD applications, *progress loops* and *effort loops*, to

quantify load generated by application code. Units of progress and effort correspond to iterations of these loops. Progress units correspond to absolute headway towards some application goal. In many parallel simulations, iterations of outer time-step loops can serve as progress events. Effort units correspond to runtime events that may have variable iteration counts, such as inner loops with possibly data-dependent execution. We model per-process load as effort units.

Several factors contribute to the variable duration of effort loops. These factors include the size and complexity of the data processed, convergence rates of numerical algorithms, availability and performance of resources such as I/O, the network and even faulty nodes. Application data, in particular, can impact many aspects of the computation, including its complexity, the degree of refinement and the speed of convergence.

These two loop types lead to recurring patterns in an application's full event trace. We have devised a filter layer to generate model data by separating MPI traces into progress and effort regions. Currently, we instrument progress loops with a single call to `MPI_Pcontrol`, which is trivial for most parallel codes since the main time-step loop is easily located. Our load balance monitoring tool automatically extracts effort regions, as discussed in the following subsection.

4.2 Effort Filtering

We approximate effort with the elapsed time spent in non-communication regions of the code. We assume that communication operations delineate effort regions and that variable time in these regions is what causes processes to wait on their peers (i.e., load imbalance). At runtime, our tool slices the trace into dynamic execution regions bounded by calls to `MPI_Pcontrol` or other MPI functions likely to indicate effort regions such as `MPI_Barrier`, `MPI_Allreduce` and `MPI_Waitall`. We monitor elapsed time in each region separately, using the start and end callpaths of the slices as identifiers. Our wrapper also records time spent in MPI operations so that communication regions can be examined as well as computational effort.

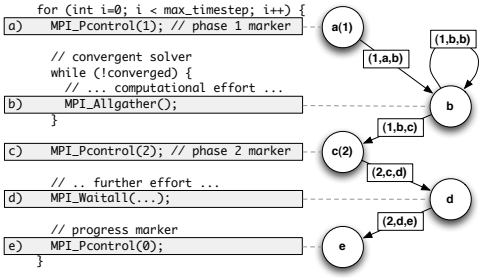


Figure 5: Effort regions in a dynamic trace.

Figure 5 uses a state machine to show how our tracer automatically extracts effort regions. The shaded MPI function call sites correspond to our machine's states. States that correspond to a phase transition are labeled with the phase's identifier in parentheses. When the tracer encounters one of these call sites, it records the effort associated with the current phase identifier, the start callpath and the end callpath. Thus, we record effort along the edges of our state machine, which the figure labels with their identifiers. Users can optionally delineate application phases (groups of effort regions) with calls to `MPI_Pcontrol(id)`, where `id` is a unique non-zero phase identifier.

As already discussed, we delineate progress events with an `MPI_Pcontrol(0)`. Each time the tracer encounters a progress event, it appends effort values for the current progress step to running lists. Thus, at the end of a run with n progress steps and m effort regions, each process has $m \times n$-element vectors of effort values. Additionally, we annotate the trace when progress events occur.

These additional trace elements allow correlation of the recorded trace with stored effort data.

4.3 Scalable Aggregation for Numerical Data

As discussed, petascale performance tools cannot naively aggregate and store data from all processes. We have designed a parallel compression algorithm to gather effort data scalably from all monitored processes, and we have integrated it as a component in our tool framework. Our algorithm uses a parallel wavelet transform to precondition data for compression. Compression is structured as a tree-based reduction over all transformed effort data. This component can be used as a generic lossy reduction technique for large volumes of numerical data.

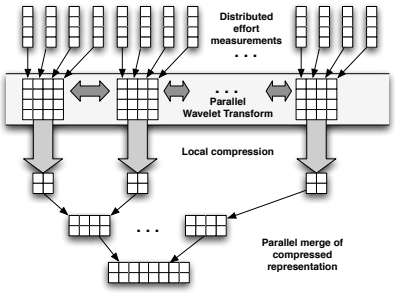


Figure 6: Parallel compression.

Figure 6 gives a high-level overview of our lossy compression architecture. We start with vectors of effort values for all progress steps distributed across the application's processes. Each of these can be considered a column in a distributed 2-dimensional matrix. Each row in the matrix represents the observed effort values from each process for a particular time-step.

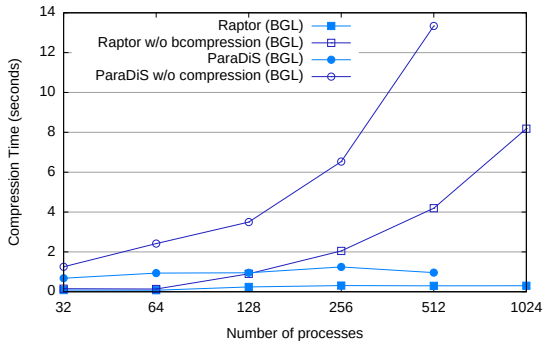
Our algorithm gathers distributed rows to fewer processes, then performs a parallel wavelet transform on the more densely-distributed data. Row consolidation may appear to reduce parallelism, but we are generally monitoring many different effort regions, each of which has its own distributed matrix. We thus balance the load by consolidating each metric on a disjoint set of processors, and by performing the wavelet transforms on each set in parallel.

After transforming the data, the algorithm thresholds the wavelet coefficients and streams them to the root of a reduction tree. As the data is streamed through the tree, it is run-length encoded, and finally the root node Huffman-encodes the remaining data. We are currently working on more sophisticated compression schemes, such as parallel EZW coding [16].

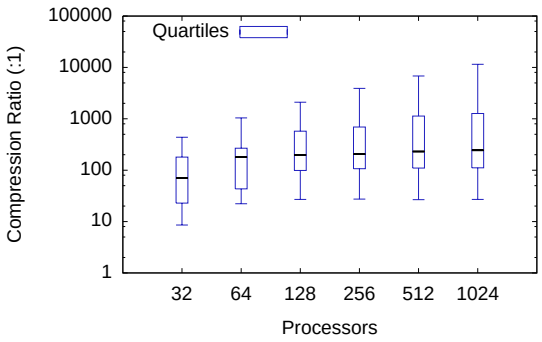
Although we describe their use for load-balance measurement here, our framework makes both our effort filter and our wavelet compression components available for use in other tools. Local effort analysis tools could use effort model data in order to identify single processor bottlenecks. Our wavelet compression algorithm has wide reuse potential as a generic, scalable aggregation tool for numerical measurements gathered from large systems.

4.4 Compression Results

We have tested our compression framework on up to 1,024 processors on an IBM BG/L system. For these tests, we used two well known simulation applications from LLNL: Raptor [6] and ParaDiS [2]. Figure 7(a) shows data collection time with our tool versus writing exhaustive data to per-process files on disk. The traces compressed were 26 time-steps long and contained between 100 and 200 effort regions. At 64 processes, exhaustive data collection saturates the I/O infrastructure and output data is serialized. Thus, the exhaustive approach scales linearly. Our compression algorithm, on the other hand, scales sub-linearly up to 1024 processes, taking near-constant exe-



(a) Compression time for traces of ParaDiS and Raptor



(b) Compression ratios for Raptor effort data.

cution time of under half a second for both applications. We achieve this performance because the parallel wavelet transform we use [11] requires only nearest-neighbor communication.

Figure 7(b) shows the distribution of compression ratios achieved over all effort regions measured for Raptor. The raw data compression ratios achieved here are impressive, and we were able to achieve considerable data reduction. Data from a 1024-process run of Raptor was compressible with a ratio of 244:1 and even a 32-processor run was compressed with a ratio of 71:1. Also, although our compression algorithm is lossy, mean-squared error rates for data reconstructed from compressed traces did not exceed 7.8%.

5 Cluster analysis

We have described tools for scalable data collection, but we have not covered extensively how our framework can be used to deploy online components for its analysis. One of our framework's major strengths is that it supports creation of new application-specific tools out of existing components. It may take developers many iterations to find the set of measurements that yield insight into a particular application's performance and the ability to combine existing methods without significant tool reengineering can speed this process.

The full performance measurement space is vast; many different quantities can be measured at different points in execution, for different regions of the code and on different processes within the same system. We are currently designing a third tool instantiation based on our framework that will find correlations across multiple dimensions in this space with cluster analysis.

The output of our scalable load-balance measurement tool is a 3-dimensional matrix that spans processes, progress steps and the full set of effort regions in application code. We are iteratively applying clustering algorithms to this data to find:

1. Which regions in the code are consuming the most time?
2. What equivalence classes exist between regions of code across different processes?
3. What metrics contribute to performance problems within these regions?

We view each of these questions as a multidimensional clustering problem on data that our framework gathers. For the first, we can cluster effort regions across processes by the amount of time

each process spends in particular effort regions across different timesteps. We can then determine if the time spent in particular regions constitutes a load imbalance by determining which clusters span only certain subsets of the full set of processes. We can use further statistical analyses and other measurement components to find correlations between slow regions on particular processes and performance metrics.

We can exploit the scalable wavelet representation outlined in §4 to perform this analysis quickly. Since a wavelet representation can be thought of as a series of incrementally higher-resolution approximations, we can perform this analysis quickly on a relatively small subset of the total data, and we can incrementally refine it if we need more detailed cross-process analysis.

6 Conclusion

We have presented a scalable tool framework suitable for petascale systems. Our framework provides a mechanism to create dynamically reconfigurable application-specific tools, and we have implemented three modules to demonstrate its versatility. First, we have created a module that enables scalable, lossless compression of MPI traces. Second, we have developed a load-balance model that characterizes parallel applications as a series of progress steps, each composed of multiple effort steps. Variability in the execution time of effort steps leads to load imbalances. We can extract model data from our MPI trace tool using a simple filter module in our framework. Third, we have developed a module for lossy compression using parallel, *in-situ* wavelet transforms to enable scalable system-wide aggregation of effort data. We have briefly discussed how to implement a new tool that scalably identifies clusters of tasks and/or code regions that exhibit similar performance. While full integration of the tracing mechanism within the context of our reconfigurable framework is on-going, our results for our existing tools demonstrate that our approach can overcome the key tool challenges for petascale systems: low overhead instrumentation, scalable data aggregation and on-line performance analysis.

References

[1] D. C. Arnold, D. H. Ahn, B. R. de Supinski, G. L. Lee, B. P. Miller, and M. Schulz. Stack trace analysis for large scale debugging. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*, Long Beach, CA, 2007.

[2] V. Bulatov, W. Cai, M. Hiratani, G. Hommes, T. Pierce, M. Tang, M. Rhee, K. Yates, and T. Arsenlis. Scalable line dynamics in ParaDiS. In *Supercomputing 2004 (SC'04)*, 2004.

[3] T. Gamblin, B. R. de Supinski, M. Schulz, R. J. Fowler, and D. A. Reed. Scalable load-balance measurement for SPMD codes. In *Submission*, 2007.

[4] T. Gamblin, R. J. Fowler, and D. A. Reed. Scalable methods for monitoring and detecting behavioral classes in scientific codes. *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*, April 14-18 2008.

[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] J. Greenough, A. Kuhl, L. Howell, A. Shestakov, U. Creach, A. Miller, E. Tarwater, A. Cook, and B. Cabot. Raptor – software and applications for BlueGene/L. In *BlueGene/L Workshop*. Lawrence Livermore National Laboratory, 2003. Available from: http://www.llnl.gov/asci/platforms/bluegene/agenda.html.

[7] P. Havlak and K. Kennedy. An implementation of interprocedural bounded regular section analysis. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):350–360, July 1991.

[8] J. Marathe, F. Mueller, T. Mohan, B. R. de Supinski, S. A. McKee, and A. Yoo. METRIC: Tracking down inefficiencies in the memory hierarchy via binary rewriting. In *International Symposium on Code Generation and Optimization*, pages 289–300, Mar. 2003.

[9] J. Mellor-Crummey. HPCToolkit: Multi-platform tools for profile-based performance analysis. In *5th International Workshop on Automatic Performance Analysis (APART)*, November 2003.

[10] 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.

[11] O. M. Nielsen and M. Hegland. Parallel performance of fast wavelet transforms. *International Journal of High Speed Computing*, 11(1):55–74, 2000.

[12] 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*, Apr. 2007.

[13] P. Ratn, F. Mueller, B. R. de Supinski, and M. Schulz. Preserving time in large-scale communication traces. In *International Conference on Supercomputing*, page (accepted), June 2008.

[14] 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.

[15] M. Schulz and B. R. de Supinski. P^NMPI tools a whole lot greater than the sum of their parts. In *Supercomputing 2007 (SC'07)*, 2007.

[16] J. M. Shapiro. Embedded image coding using zerotrees of wavelet coefficients. *IEEE Transactions on Signal Processing*, 41(12):3445–3462, December 1993. Available from: http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=258085.

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

[18] The Open|SpeedShop Team. Open|SpeedShop for Linux. Available from: http://www.openspeedshop.org.