Scalable load-balance measurement for SPMD codes

Todd Gamblin, Bronis R. de Supinski, Martin Schulz, Robert Fowler, Daniel Reed

August 5, 2008

SC08
Austin, TX, United States
November 15, 2008 through November 21, 2008

Scalable Load-Balance Measurement for SPMD Codes[*]

Todd Gamblin[*] tgamblin@cs.unc.edu
Bronis R. de Supinski[†] bronis@llnl.gov
Martin Schulz[†] schulzm@llnl.gov
Rob Fowler[*] rjf@renci.org
Daniel A. Reed[‡] daniel.reed@microsoft.com

[*]Renaissance Computing Institute, University of North Carolina at Chapel Hill
[†]Lawrence Livermore National Laboratory
[‡]Microsoft Research

ABSTRACT

Good load balance is crucial on very large parallel systems, but the most sophisticated algorithms introduce dynamic imbalances through adaptation in domain decomposition or use of adaptive solvers. To observe and diagnose imbalance, developers need system-wide, temporally-ordered measurements from full-scale runs. This potentially requires data collection from multiple code regions on all processors over the entire execution. Doing this instrumentation naively can, in combination with the application itself, exceed available I/O bandwidth and storage capacity, and can induce severe behavioral perturbations.

We present and evaluate a novel technique for scalable, low-error load balance measurement. This uses a parallel wavelet transform and other parallel encoding methods. We show that our technique collects and reconstructs system-wide measurements with low error. Compression time scales sublinearly with system size and data volume is several orders of magnitude smaller than the raw data. The overhead is low enough for online use in a production environment.

1. INTRODUCTION

In large-scale, distributed-memory parallel computers, balanced load is critical for system performance. Scientific algorithms that run on these machines typically use synchronous programming paradigms [24] in which excess computation on one process can cause all others in the system to idle. The impact of imbalance is becoming worse since the size of the largest parallel supercomputers is increasing dramatically. Six years ago, the largest machine had only 9,632 processors, while today's largest machine, IBM's Blue Gene/L [15], has 212,992. A single overloaded process can now force hundreds of thousands of other processes to wait.

[*]Part of this work was performed under the auspices of the U.S. Department of Energy, supported by the Office of Science SciDAC PERI (DE-FC02-06ER25764) and by Lawrence Livermore National Laboratory under contract DE-AC52-07NA27344 (LLNL-CONF-406045).

Measuring and diagnosing load-balance in large-scale systems is difficult because all processes must be observed. Furthermore, many scientific applications use data-dependent, adaptive algorithms and may redistribute load dynamically, so it is necessary to observe the system over time. Finally, not all regions in a parallel code may contribute to a load imbalance, so data must be collected over three dimensions: *i)* where in the code the problem occurs; *ii)* on which processors the load is imbalanced; and *iii)* when the imbalance emerges during execution. Naively recording full traces over full-scale runs is infeasible both because excessive I/O communication can perturb application and system behavior and because the data storage requirements can potentially overwhelm system capacity.

The main contribution of this paper is a novel load monitoring scheme that can reduce the volume of system-wide, time-varying measurements by two to three orders of magnitude. Our scheme uses lossy compression techniques developed for signal processing to compress two-dimensional trace data. Rank (process id) and time comprise the dimensions, and we use parallel wavelet encoding techniques to reduce communication and storage requirements enough to make on-line measurement of production runs feasible. This scheme preserves process id and time information needed for load balance monitoring. It also enables a time versus error tradeoff decision, but even at the fastest settings, the collected data is sufficient for problem diagnosis.

Part of our approach is to collect performance metrics as rates normalized to the execution of application loops that are classified as units of *progress* or *effort*. Progress loops represent steps towards some goal expressed in the application domain, *e.g.*, time steps, experiments analyzed, or transactions processed. Effort loops, or regions, are nested within progress loops. They represent the work that is performed to achieve a unit of progress. The number of iterations of an effort loop may vary over time because of adaptation in the size of local data structures, or because of the convergence properties of an iterative algorithm. The time, or other performance metrics, for executing one iteration of an effort loop may vary due to data locality issues, memory or I/O errors, and other interactions with the system architecture. Examples of variable-effort codes include iterative solvers, *e.g.*, conjugate gradient, adaptive mesh refinement, or time sub-cycling methods [10]. To measure and diagnose dynamic load balance problems, we focus on the evolution of these measures across processes.

This paper is organized as follows. In §2, we present our model for load-balance analysis to give context for measure-

SC2008 November 2008, Austin, Texas, USA 978-1-4244-2835-9/08 $25.00 ©2008 IEEE

ments made in later sections. §3 gives an brief overview of wavelet analysis and describe its fitness for our problem. In §4, we detail the architecture and implementation of our tool, and in §5, we show through extensive experimental validation that our techniques are scalable both in time and space for real-world scientific applications. We also show that our methods yield very low compression error. Finally, in §6 and §7, we place our work in the context of other performance analysis research and summarize our findings.

2. THE EFFORT MODEL

We have developed a broadly applicable model for load in large-scale scientific applications. Our model is targeted primarily at Single Program Multiple Data (SPMD) parallel applications, particularly those that use MPI [24]. As MPI is the *de-facto* standard for large-scale scientific computation, the vast majority of distributed-memory supercomputer applications could use our framework. We introduce two concepts, *progress* and *effort*, to quantify the high-level load semantics of most SPMD applications. While we have designed this model with MPI in mind, it is flexible enough to model the behavior of a broad range of scientific applications. In this section, we give an overview of our model and describe how it can be used to diagnose load imbalance.

Loops in SPMD applications can be divided into two categories:

Progress loops. Typically the outer loops in SPMD simulations, progress loops indicate absolute headway towards some domain-specific goal. For example, in climate simulations, a progress loop iteration computes the physics for some known interval of simulated time. Each iteration is a global, synchronous step toward completion, and measuring the duration of progress loops provides an estimate of how long the application will run. Even when the total number of time-steps is not known *a priori*, the evolution of progress loop performance indicates how the application's total load varies over time.

Effort loops. These are variable-time loops with possibly data-dependent execution. Nested inside of progress loops, effort loops may be part of a convergent numerical algorithm (*e.g.*, conjugate gradient methods) or they may integrate over variable-size, or adaptively-refined partitions (as in mesh-refining codes). We can measure the work required to take a progress step by counting effort iterations.

A number of factors contribute to the variable duration of a set of effort loops. These include the size and complexity of the data processed, the availability and performance of resources such as I/O, the network, and even faulty nodes. These in turn affect the performance of the enclosing progress loops. Application data can impact many aspects of the computation, including its complexity, the degree of refinement, and the speed of convergence.

Effort loops provide a basis for comparison between different progress loop iterations. The relationship between effort loops and total time for their enclosing progress loop can show how severely data dependency and intrinsic application factors affect application run time. For example, if one progress iteration takes longer than another, *and* an internal effort loop takes significantly more iterations to converge, we can attribute the problem to application data and the efficiency of the numerical algorithm. If, however, progress step performance varies but the numbers of iterations of the effort loops are unchanged, some architectural or configuration factor may be influencing performance.

We can compare progress and effort loop iterations within the same process in a running application or between processes. Intraprocess results capture the evolution of load within that process. Interprocess comparisons can be even more enlightening: they capture application load balance and any relative performance anomalies between processes.

The presence of these two loop types in application code leads to recurring patterns in an application's full event trace. Certain recurring events (or sequences of events) delimit successive iterations of both progress and effort loops, and could be monitored system-wide. Ideally, detection of these events could be automated; that is beyond the scope of this paper. In this work, we manually instrument progress events and we use elapsed time to estimate effort. This is sufficient to illustrate our model without loss of generality, as our framework could compress arbitrary effort measures in place of time.

Progress and effort are high-level, application-semantic measures. Hardware performance monitors (HPMs) can capture many architectural events [4, 22, 33]. In the long term, we intend to couple these measures to enable problem diagnosis by modeling application behavior with four categories of rates:

- The ratios of two HPM events observed over some measurement period;
- HPM events per unit time measured over an observational interval;
- HPM events per unit of application effort or progress;
- Application-specific events per unit time over the observation period;

HPM event ratios and rates, such as cache miss rates, misprediction ratios or instructions per cycle, measure the efficiency with which certain architectural devices are operating. When expressed in terms of the application-specific events (*i.e.*, progress and effort), they capture application efficiency. Other events such as instruction counts and floating point operations measure real work expended. Progress and effort rates, are the "bottom line" measurements that capture the domain-specific aspects of application performance. Correlation with HPM data provides deeper understanding of performance. If we find that progress is slowing, we can consult effort data to determine which code regions may be causing the problem. Correlating the architectural and application-specific rates supports root cause diagnosis.

3. WAVELET ANALYSIS

To collect system-wide effort data scalably, we need a compact representation and a scalable aggregation method for the data. Wavelet analysis has become a prominent, if not the dominant, method for data reduction in fields as diverse as signal processing, digital imaging [1, 2], and sensor networks [34]. A *wavelet transform* expands a function in the spatial domain to a function of orthogonal polynomials in $L^2(\mathbb{R})$. It is a particularly interesting operator because wavelet expansions require very few terms to approximate most functions.

The theoretical underpinnings of wavelet analysis [11, 35] have deep roots in functional analysis, and are beyond the

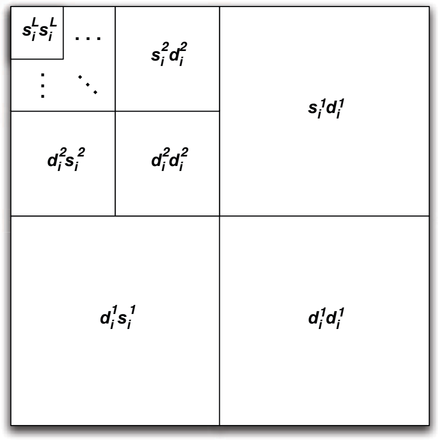


Figure 1: Multiscale decomposition for our level L 2-D wavelet transform.

scope of this paper. We are concerned primarily with those properties of the wavelet transform that lend themselves to load balance measurement as described above. Here, we provide a brief overview of the discrete wavelet transform, as well as a discussion of its suitability for scalable load measurement.

The *discrete wavelet transform* is an operation that converts a set of N samples, $s_0 \ldots s_{N-1}$ into two sets of $N/2$ coefficients. Recursive applications of the transform are expressed with *levels*, and we denote the inputs to the transform as level 0, or $s_0^0 \ldots s_{N-1}^0$. At level l, each sample s_i^{l-1} is converted to coefficients $s_0^l \ldots s_{n-1}^l$ and $d_0^l \ldots d_{n-1}^l$ according to the recurrence relations:

$$s_i^l = \sum_{d=0}^{D-1} a_d s_{\langle 2i+d \rangle_{2n}}^{l-1} \qquad d_i^l = \sum_{d=0}^{D-1} b_d s_{\langle 2i+d \rangle_{2n}}^{l-1} \tag{1}$$

where $n = N/2^l$, $i = 0 \ldots n-1$, a_d and b_d are coefficients for low- and high-pass wavelet filters, respectively, and D is the number of coefficients in the wavelet filters. $\langle x \rangle_m$ is the modulus function defined so that $\langle x \rangle_m \in [0, m-1]$ for all $x \in \mathbb{Z}$. The key observation here is that s_i^l contain low-frequency information from the input samples, while d_i^l represent high-frequency details. If the transform is applied recursively L times, we call this a *level L* transform, and the transformed data will have L scales.

D is the width of the filters (a and b) used, and this depends on which wavelets are selected as the basis functions of the transform. In this work, we use the Cohen-Daubechies-Favreau 9/7 (CDF 9/7) wavelets [11]. The high-pass and low-pass CDF 9/7 filters contain 9 and 7 real-valued elements, respectively. They have been shown to work well for lossy compression of graphical images and are in wide use as part of the JPEG-2000 standard [1] for image compression.

We have selected a wavelet transform for two reasons. First, scalable parallel algorithms for the discrete wavelet transform are known [25]. Second, certain properties of the transform are particularly useful for load-balance monitoring. Wavelet coefficients are sparse and well-suited to compression. Further, the wavelet representation is locality-preserving. Unlike the output of traditional global transforms, (*e.g.*, the Fast Fourier Transform and the Discrete Cosine Transform), each wavelet coefficient contains both locality information *and* frequency information from the original set of samples, which is essential for the purpose of our analyses. The coefficients represent data at a particular frequency, or *scale* (this is analogous to the level above), as well as a particular *position* in the original samples. The wavelet transform is thus *multiscale* in nature, in that s_i^l from successively deeper levels of the transform represent progressively larger-scale information from the original data. These low-frequency, large-scale subbands provide a very compact approximation of the original data, enabling speedy analysis. Such approximations can be selectively refined with high-frequency information if more detail is needed.

In this paper, we use multiple two-dimensional, parallel wavelet transforms to analyze measurements taken at runtime. The two-dimensional transform is a series of one-dimensional transforms applied alternately to the rows then the columns of a matrix. Here, the row dimension is progress steps, and the column dimension is the ranks of processes in the parallel application. Values in each matrix represent effort generated by some region in the code. Figure 1 shows the resulting decomposition. The lower right quadrant of the matrix contains the high frequency data from both dimensions, the upper right and lower left quadrants contain high-frequency data in one dimension and low-frequency data in another, and the low-frequency information in the upper left quadrant is recursively transformed L times.

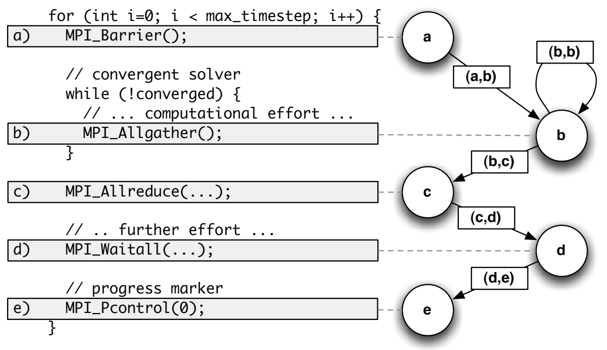
4. A FRAMEWORK FOR SCALABLE LOAD MEASUREMENT

We have designed and implemented a framework to scalably measure and compress effort model data. Our design consists of two major components. First, we have devised a scheme for extracting effort model data from MPI events. Our scheme monitors events as they occur, and determines effort regions automatically. Second, we have designed a scalable aggregation method using a parallel wavelet transform and parallel compression techniques.

We employ a general-purpose tool design. We implement our model data extraction techniques using the PMPI interface, and we map MPI events to distributed effort matrices that can be quickly aggregated. While this paper focuses on load imbalance, our techniques are applicable to a much broader range of problems. The effort model can be applied to a wide range of scientific applications, and it provides application-semantic information that is generally useful for performance analysis. Our aggregation tool is decoupled from the effort model; it simply compresses generic numerical data. It could easily support runtime monitoring for many other performance tools, such as profilers or internet monitoring tools. It could also be used to efficiently collect and visualize evolving algorithm/application data.

4.1 Effort Filter Layer

We have designed a filter layer to automatically extract effort data from MPI applications. Since the filter layer is implemented using the PMPI profiling interface, it is a link-level library. We currently require the user to instrument a single progress loop in her application, but once this is added, the user need only link against our library to take advantage of our data collection tools. Work to automatically detect progress loops is currently in progress.



```
    for (int i=0; i < max_timestep; i++) {
a)    MPI_Barrier();

      // convergent solver
      while (!converged) {
        // ... computational effort ...
b)      MPI_Allgather();
      }
c)    MPI_Allreduce(...);

      // .. further effort ...
d)    MPI_Waitall(...);

      // progress marker
e)    MPI_Pcontrol(0);
    }
```

Figure 2: Dynamic identification of effort regions.

To measure computational effort, we record the elapsed time spent in the non-communication regions of the code. We assume that effort regions are delineated by *split operations*, or globally synchronous communication constructs. Split operations, which are the routines where processes wait on their peers, include not only collectives such as `MPI_Allgather()`, `MPI_Barrier()` and `MPI_Reduce()`, but also `MPI_Waitall()`. `MPI_Waitall()` frequently ends phases in physical simulations where nodes must block while waiting for synchronous communication to complete. We allow for customization of the exact set of calls considered split operations to suit the application.

Figure 2 illustrates the effort filter layer with a state machine. In the figure, the shaded `MPI_Pcontrol()` and split operation call sites correspond to states. When control passes over these call sites, our state machine transitions along a (start callpath, end callpath) edge. The tracer also adds the elapsed time to the effort associated with this edge. Thus, we record effort along the edges of our state machine, labeled in the figure by their identifiers. At runtime, we monitor elapsed time for each dynamic effort region separately, using the start and end callpaths as identifiers. The framework also records time spent *inside* split operations as a separate measure of communication effort. We use the publicly available DynStackwalker API [27] to look up callpaths.

Effort data is recorded at the end of each progress iteration. Our filter appends effort values for all regions in the current progress step to per-region vectors. Thus, at the end of a run with n progress steps and m effort regions, each process has m n-element vectors of effort values. Since effort values are keyed by their dynamic callpaths, the user can later correlate effort expended at runtime with specific regions in the source code.

Currently, the user must call `MPI_Pcontrol(0)` to mark progress events at runtime. The applications we examined required the addition of only a single call in the main time step loop, and these loops were generally easy to locate in the source code without prior knowledge. As mentioned, we hope to automatically detect progress loops from MPI traces in the long term.

To divide the effort space into phases, the user of our tool may insert additional `MPI_Pcontrol(id)` calls with unique integer identifiers. When a call to `MPI_Pcontrol(id)` is made with a non-zero parameter, our tool marks this as a phase shift and records the parameter as the phase identifier. Effort is recorded separately for each phase so that the user can view the behavior of each phase independently. Phase markers are entirely optional.

4.2 Parallel Compression Algorithm

As discussed in §1, it is not feasible for a scalable trace framework to aggregate and to store data from all processes in large systems naively. We therefore designed a scalable, parallel compression algorithm to gather effort data from all processes in a parallel application. Our algorithm aggressively targets the I/O bottleneck of current large systems by using parallel wavelet compression to reduce data size. We make use of *all* processes in large systems to perform compression fast enough for real-time monitoring at scale.

We base our parallel transform on that of Nielsen, *et al.* [25], although our data distribution is slightly different. At the end of a trace, each process in the distributed application has a vector of effort measurements (one for each progress step) for each effort region. Each of these vectors can be considered a row in a 2-dimensional, distributed matrix. For transforms within rows of this matrix, the data is entirely local, but transforms within columns are distributed. For good performance, we need at least $D/2$ (half the width of the wavelet filter) rows per process. This ensures that only nearest neighbor communication is necessary between processes. Further, for a level L transform, the number of rows per process should be large enough that it can be recursively halved L times and still not shrink below $D/2$. To ensure this, we consolidate rows before performing the transform. For a system with P processes, our algorithm regroups the distributed matrix into P/S local sets of S rows. Figure 3 shows how this would look for $S = 4$. We then perform the parallel transform on the consolidated matrix.

After transforming each matrix, our algorithm encodes the transformed coefficients using the Embedded Zerotree Wavelet (EZW) coding [31]. We chose this encoding for two reasons. First, it parallelizes well [3, 19]. The data layout for zerotree coding corresponds to the organization of transformed wavelet coefficients, and encoding is entirely local to each process. Second, it supports efficient space/accuracy tradeoffs. The bits output in EZW coding are ordered by significance. Each pass of the encoder tests wavelet coefficients against a successively smaller threshold and outputs bits indicating whether the coefficients were larger or smaller than the thresholds. The first few passes of EZW-coded data are typically very compact, and they contain the most significant bits of the largest coefficients in the output. We can thus obtain a good approximation by reading a very small amount of EZW data. Examining more detailed passes refines the quality of the approximation at a cost of higher data volume. In our framework, the number of EZW passes is customizable, allowing the user to moderate this tradeoff.

In the final stage of compression, we take local EZW-coded passes, run-length encode them, and then merge the run-length encoded buffers in a parallel reduction. Each internal node of the reduction tree receives encoded buffers from its children, splices them together without decompressing, and sends the resulting merged buffer to its parent. Splicing is done by joining runs of matching symbols at either end of the encoded buffer. We aggregate this compressed data into a single buffer at the root of the reduction tree, and we Huffman-encode [17] the full buffer.

The consolidation step of our algorithm sacrifices parallelism for increased locality and reduced communication cost. However, we typically have many effort matrices to transform, and we can perform S concurrent transforms. If

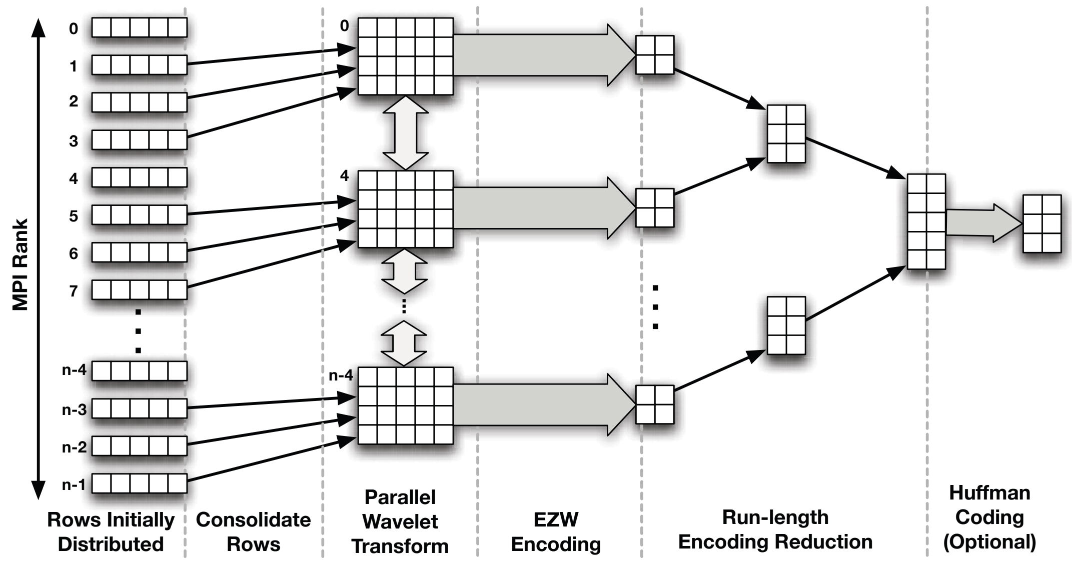

MPI Rank
0
1
2
3
4
5
6
7
n-4
n-3
n-2
n-1
0
4
n-4
Rows Initially Distributed
Consolidate Rows
Parallel Wavelet Transform
EZW Encoding
Run-length Encoding Reduction
Huffman Coding (Optional)

Figure 3: Parallel compression architecture.

```
DISTRIBUTE-WORK($P, S$)
 1  $comm$ = MPI-COMM-SPLIT($WORLD, rank \ \% \ S, 0$)
 2  $v = effortVectors.first$
 3  while $v <= effortVectors.last$
 4  do $set \leftarrow 0$
 5      while $set < S$ and $v \leq effortVectors.last$
 6      do $base = (rank \ div \ S) * S$
 7          if $rank \ \% \ S = set$
 8             then $i = 1$
 9                  while $i < S$
10                  do START-RECV-FROM($comm, base + i$)
11                      $i \leftarrow i + 1$
12
13             else START-SEND-TO($comm, v, base$)
14
15          $set \leftarrow set + 1$
16          $v = effortVectors.next$
17
18      FINISH-SENDS-AND-RECEIVES($comm$)
19      DO-COMPRESSION($comm, localRows$)
```

Figure 4: Data consolidation algorithm.

we have S or more effort regions, we can distribute this work over *all* processes in the system. Figure 4 gives pseudocode for our algorithm. We first split the system into S separate sets of ranks, each with its own local communicator, using a call to `MPI_Comm_split()`. After this, the code behaves as S separate parallel encoders executing simultaneously. Each program has P/S processes, with ranks 0 to P/S. These ranks map to modulo sets in the entire system's rank space.

Within each modulo set, each process with id *rank* sends its first local effort vector to the process with id 0. It sends its next vector to the process with id 1, and so on until S vectors have been sent. These sends consolidate data for S effort matrices, after which S simultaneous instances of our compression algorithm are performed. This entire process is repeated until all effort matrices have been encoded.

4.3 Trace Reconstruction

Our compression tool produces a compact representation of system-wide, temporally-ordered load balance data. Once stored, we can decompress and reconstruct the data for analysis or for display in a visualization tool. The reconstruction process is simply the inverse of the compression process. We Huffman-decode, run-length decode, EZW-decode, and apply the inverse wavelet transform to the compressed data. This decompression is independent for each effort region recorded at runtime, allowing focused data exploration.

The wavelet representation is particularly useful for load-balance modeling because it preserves spatial information. Since the transform records both scale information and spatial information, features represented by coefficients in the wavelet domain are reconstructed at their original rank and progress step when the inverse transform is applied. Wavelets are thus particularly useful for representing outliers, large spikes, and aperiodic data.

Our compression scheme is lossy for several reasons. The double-precision floating point values used in the CDF 9/7 wavelet transform introduce rounding error. The double-precision output is then scaled and converted to 64-bit integers for EZW coding, which introduces quantization error. Finally, the EZW output stream is truncated after a certain number of passes, giving variable approximation error. Our results show that the amount of error introduced by rounding and quantization is modest. Error introduced by truncation of the EZW stream can be greater, but wavelet compression and EZW coding have unique properties that make this error less problematic. Most importantly, the wavelet transform typically produces very few term approximations, so a large number of coefficients in the transformed data are zero or near-zero to begin with. Second, we filter out only *less* significant data by truncating an EZW stream. Higher-magnitude coefficients in the wavelet domain correspond to more significant features in the original data, so even under severe compression, wavelets yield a reconstruction that is qualitatively similar to the original data.

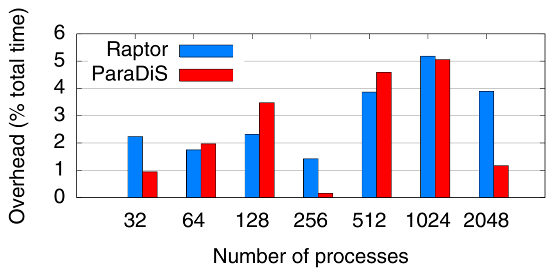


Figure 5: Tool overhead for Raptor and ParaDiS.

5. EXPERIMENTAL RESULTS

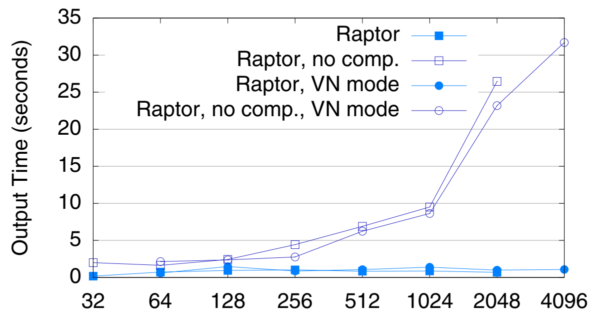
We conducted experiments with our tool on two systems. The first is an IBM BlueGene/L (BG/L) system with 2048 dual-core, 700 MHz PowerPC compute nodes. Each node has 1 GB RAM (512 MB per core). A 3-D torus network and two tree-strucutured networks are available for communication between processes. We used IBM's xlC and gcc compilers along with IBM's MPI implementation. In our tests, we used both of BG/L's modes of execution: *coprocessor mode*, with the second core on each node dedicated to communication; and *virtual node* (VN) mode, in which both cores perform computation and communication. The second system is a Linux cluster with 66 dual-processor, dual-core Intel Woodcrest nodes. The 264 cores run at 2.6 GHz. Each node has 4 GB RAM, and Infiniband 4X is the primary interconnect. We used the Intel compilers and OpenMPI, with Infiniband used for inter-node communication and shared memory used for intra-node communication.

We use two well-known scientific applications for our tests, ParaDiS and Raptor, both of which scale to 16,000 or more processes [20]. ParaDiS [6], models the dynamics of dislocation lines in crystals as a network of *nodes*, or discretized points, and *arms*, which connect nodes in the dislocation network. ParaDiS uses an adaptive load balancer at runtime to keep load across this network even. The dislocations are divided into bounded regions called *domains*. Each process computes on one domain, with the load balancer periodically redistributing dislocations between domains. Raptor [16], is an Eulerian Adaptive Mesh Refinement (AMR) code that simulates complex fluid dynamics using the Godunov finite difference method. Like ParaDiS, Raptor is known to have variable amounts of per-process computation. However, whereas the underlying physics (the time evolution of dislocations) is the cause of imbalance in ParaDiS, load imbalance in Raptor arises from mesh refinement.

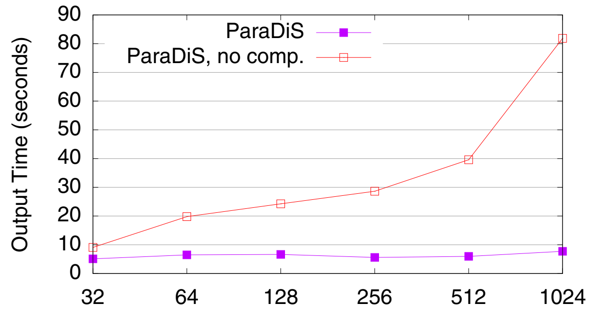
5.1 Merge Time

We performed scaling runs of Raptor and ParaDiS on our clusters and measured the time consumed by our compression algorithm. This includes transform, encoding, and file write time. For comparison, we performed an identical set of runs in which we dumped exhaustive data to disk. For these experiments, the exhaustive dump is done *after* the consolidation of rows. That is, each leaf node in our run-length encoding reduction tree dumps its partition of the distributed matrix to a file. Thus, our exhaustive dump takes advantage of parallel I/O if it is available.

For both applications, we ran our simulations for 1024 time-steps. During compression, we allowed the wavelet transform to recur as deeply as possible, and we set the



(a) Raptor on Blue Gene/L



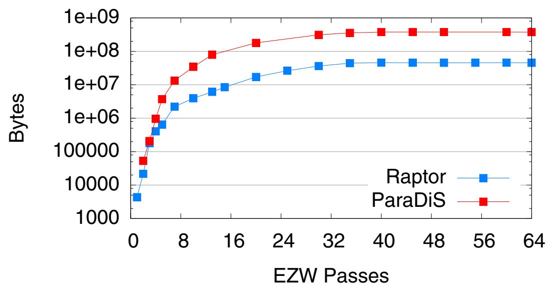
(b) ParaDiS on Blue Gene/L

Figure 6: Compression and I/O times for 1024-timestep traces.

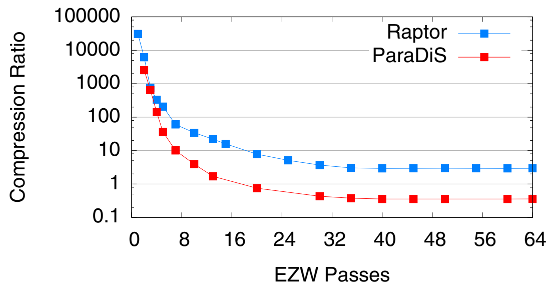
initial phase of our algorithm to consolidate to 128 rows per process. We truncated the output to 4 EZW passes. For each run, we recorded total run time, time to write out raw data, and time to compress and write compressed data.

Figure 5 shows the overhead of our tool in terms of percent increase in application runtime, not including compression. For both Raptor and ParaDiS, overhead was always 5% or less compared to an uninstrumented run of the same application. This is sufficient to measure production codes without severe perturbation.

Figure 6 shows the time required to write out load balance data at the end of a run, with and without our tool. For both ParaDiS and Raptor, as we increase the system size, the load on the I/O system also increases, and the time to write uncompressed data increases modestly until the I/O system is saturated. For Raptor, the I/O system saturates at around 1024 processes, while for ParaDiS it starts slightly earlier, at 512 processes. In both cases, once the BG/L I/O system is saturated, write performance degrades significantly. Alternatively, our compression algorithm achieves fairly constant performance across the task range. The time is primarily consumed by transforming and encoding the data, and our compressed files do not saturate the I/O system. Our approach scales well since our transform algorithm requires only nearest-neighbor communication, and the EZW encoding algorithm is entirely local. In fact, this implementation of the wavelet transform achieves near-perfect speedup [25]. In the limit, the run-length encoding phase is linear in terms of the *compressed* data size, but this data does not come close to saturating our machine's I/O system. Since our machine's I/O system is proportionally small, we expect our scheme to keep data volume manageable for most large high-performance I/O systems.



(a) Compressed size vs encoded passes.



(b) Compression ratio vs encoded passes.

Figure 7: Varying EZW passes.

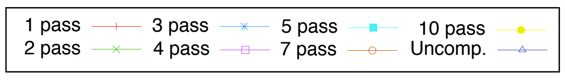
5.2 Data Volume

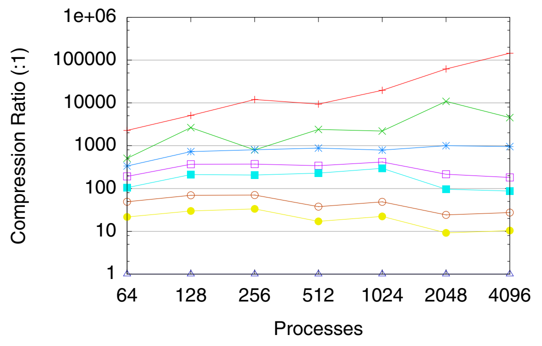
In this section, we quantify the volume of data produced by our data collection algorithm. We characterize the degree to which users can trade off accuracy for improved compression, and we compare the total volume of data produced by our algorithm to the size of uncompressed output. For these runs, we used a maximum of 128 rows per process, and we allowed the wavelet transform to recur up to level 5, depending on the dimensions of the matrix to be compressed.

As mentioned, the EZW compression algorithm allows the user to trade off error and data volume by adjusting the number of EZW passes encoded. To characterize this tradeoff, we held system size constant and ran our tool varying the number of passes. Figures 7(a) and 7(b) show compressed data sizes and compression ratios from 1024-process, 1024-timestep runs of raptor. We see that the first few EZW passes do not consume a large amount of space when compressed. For one pass, the compression ratio is over 10,000:1 and up to 5 passes, it is 100:1 or greater. The compressed size then increases until around 35 passes, beyond which the total size does not increase significantly.

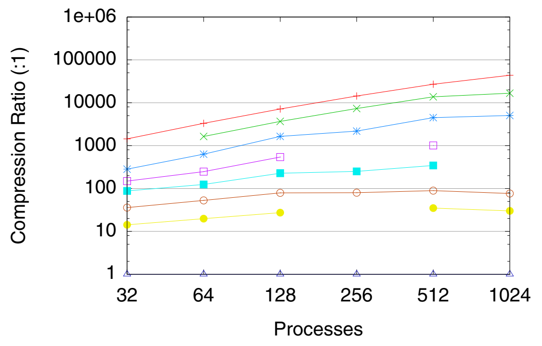
We next conducted runs to measure total data volume and compression ratio, varying system size and number of EZW passes. For these runs, we ran Raptor for 1024 progress steps and compressed data from its 16 effort regions. We also ran ParaDiS for 1024 progress steps and compressed data from its 120 effort regions. Figure 8 shows these results.

Figures 8(a) and 8(b) show raw compression ratios for ParaDiS and Raptor. Recording only the first pass of the EZW-encoded output, we can achieve compression ratios of over three orders of magnitude for both codes. Recording five passes gives compression ratios close to or above two orders of magnitude, and the user of our algorithm could choose fewer passes to obtain compression ratios in between 100:1 and 1000:1. Accordingly, Figure 8(c) shows that our compressed representation is always well below the size of the uncompressed data.

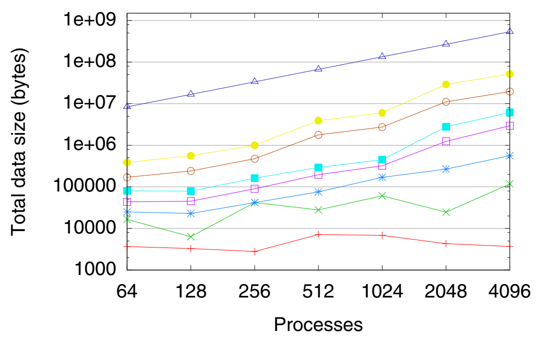




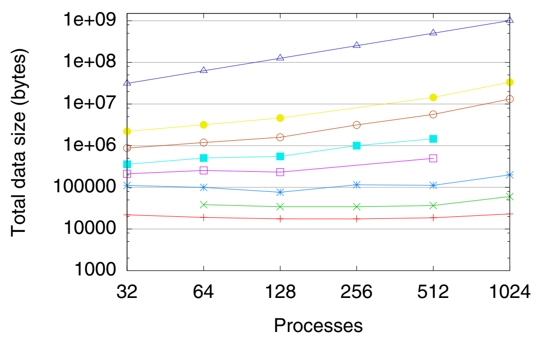
(a) Compression ratio vs. processes (Raptor).



(b) Compression ratio vs. processes (ParaDiS).

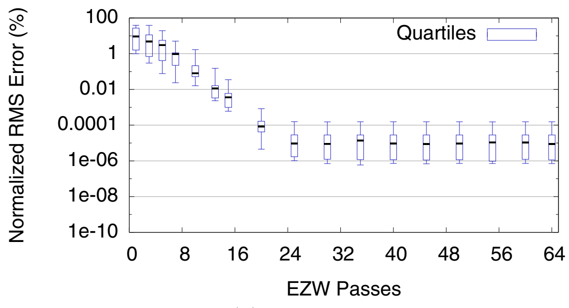


(c) Total compressed size vs. processes (Raptor).

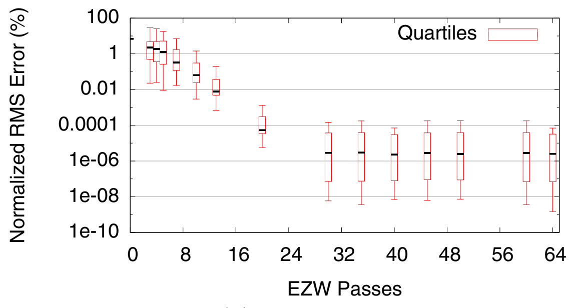


(d) Total compressed size vs. processes (ParaDiS).

Figure 8: Total data volume and compression ratios.



(a) Raptor



(b) ParaDiS.

Figure 9: Error vs. encoded EZW passes.

As discussed in §1, the collective output capability of all nodes in modern supercomputers far exceeds the available I/O bandwidth. Compression ratios in the 100:1-1000:1 regime enable system-wide data collection without perturbation. Figure 6 validates this, as it clearly shows that our algorithm is bounded only by the local time to transform and to encode, while dumping exhaustive data is bound by the capacity of the I/O system.

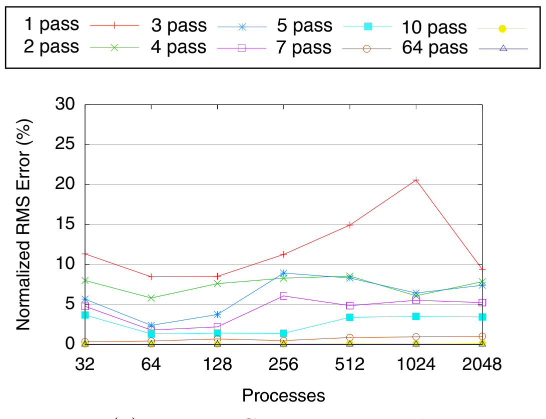
5.3 Reconstruction Error

There are three sources of error in our compression algorithm: rounding error in the CDF 9/7 wavelet transform, quantization error in the encoding process, and EZW pass thresholding. The first two sources are unavoidable, but they contribute only modestly to the total error. The third can be adjusted by the user of our tool to trade off space and accuracy. Obviously, we cannot afford to discard too much data, as severe error would hinder the characterization of effort. We characterize error in our compression algorithm in two ways. In this section, we provide a *quantitative* discussion of error in our algorithm. In the next section, we discuss the *qualitative* error of our method by comparing compressed data from our tool to exact data for several effort plots.

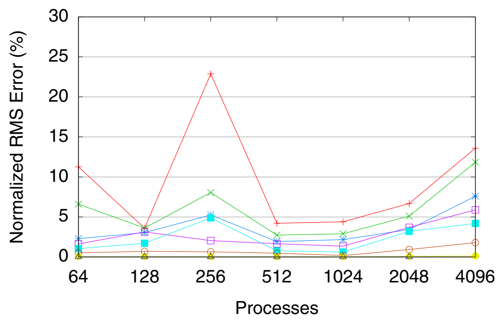
We use root mean-squared (RMS) error, normalized to the range of values observed, to evaluate reconstruction error quantitatively. For an m by n effort matrix E and its reconstruction R, the normalized RMS error is:

$$\text{nrmse}(E,R) = \frac{1}{\max(E) - \min(E)} \sqrt{\sum_{ij} \frac{(R_{ij} - E_{ij})^2}{mn}} \quad (2)$$

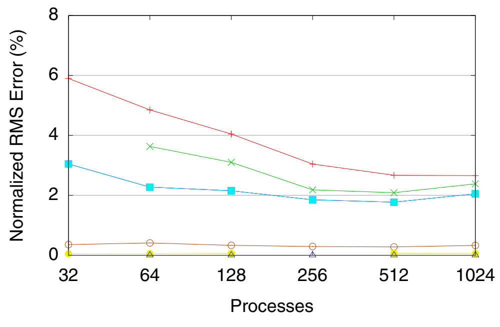
where $max(E)$ and $min(E)$ are the maximum and minimum values observed in the exact data. We normalize the error here so that we can compare the results across applications, job sizes, and input sets.



(a) Raptor, Coprocessor Mode



(b) Raptor, Virtual Node Mode

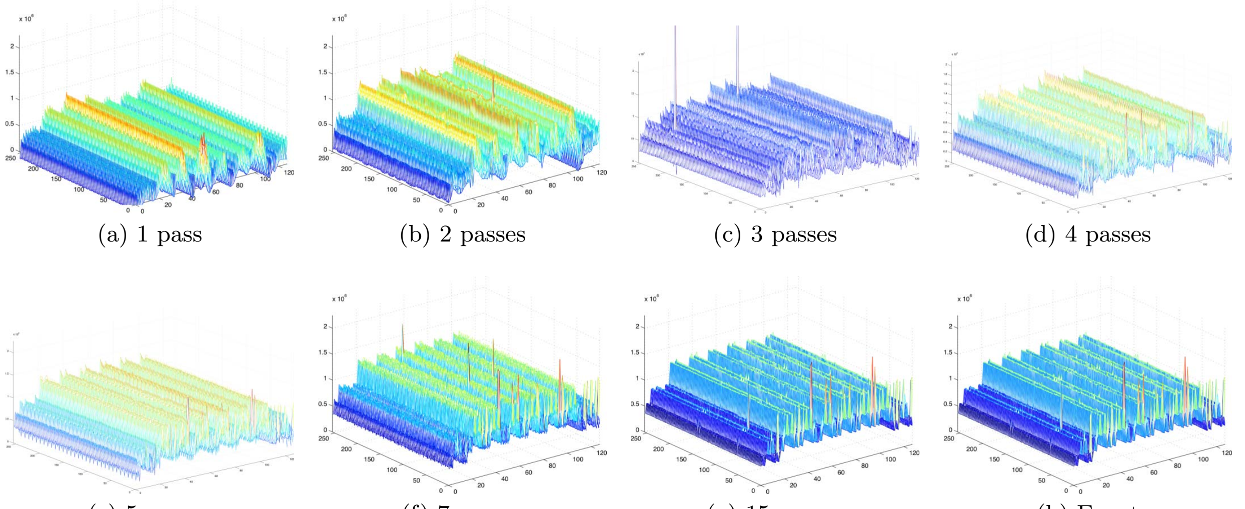


(c) ParaDiS, Coprocessor Mode

Figure 10: Median Normalized RMS error vs. system size for Raptor on BG/L.

We conducted 1024-process, 1024-progress step runs of Raptor and ParaDiS, varying the number of EZW passes output to the compressed files. Figure 9 shows the normalized RMS error for each of these runs. We use boxplots to give an idea of how error varies depending on the characteristics of different effort regions. For Raptor, there are 16 effort regions, and for ParaDiS, there are 120. Our box plots show rectangles from the top to bottom quartile of compression ratios, with whiskers extending out to the maximum and minimum values. The median value is denoted by a black tick inside the box.

For Raptor (Figure 9(a)) we see that the median error decreases from around 10% for a 1-pass run to near zero (8.8×10^{-6}%) for a full 64-pass run. For the first few passes, there is a wide range from 1% to 25%. After four passes, the



(a) 1 pass (b) 2 passes (c) 3 passes (d) 4 passes

(e) 5 passes (f) 7 passes (g) 15 passes (h) Exact

Figure 11: Progressively refined reconstructions of the remesh phase for 128-processor ParaDiS runs. All measurements are in nanoseconds (X axis: rank 0-127, Y axis: progress step 0-255).

median error is 4% and only the top quartile of error values exceeds 10%. By seven passes, median error is less than 1% and no error value exceeds 10%. Comparing these error values with the corresponding compression ratios shown in Figure 7(b), we see that median 4% measurement error can be achieved with compression ratios of over 500:1. The ParaDiS results in Figure 9(b) are similar to those from Raptor. Median error starts above 10% with a wide spread, but it drops quickly. Again, at seven passes error is less than 1% and by 30 passes there is hardly any loss of accuracy. Across the board, median error for ParaDiS is slightly lower than that for Raptor.

To assess whether reconstruction error remains stable as system size increases, we conducted scaling runs of Raptor and ParaDiS, varying system size and number of EZW passes. Again, we recorded exhaustive data along with compressed data at the end of each run, and we compared the two to obtain error values. Figure 10 shows the median normalized RMS error for these runs. For ParaDiS, error decreases as we scale the system size up, and it begins to level off as we approach 1024 processes. The decrease in error here is likely due to use of strong scaling in our ParaDiS runs. As the number of processes increases, the amount of work per process shrinks, and more processes are left idle. Compression improves as the number of similar idle processes grows. Our scaling runs of Raptor show more variable error, as we used a data set with heavier load. There are spikes in median error for the 256- and 4096- process runs in virtual node mode, as well as the 1024-process run of Raptor in coprocessor mode. In all cases, we see that the spikes are only significant for runs with one EZW pass. In these cases, the median error can jump above 20%. However, the median error is below 10% for all runs with three or more EZW passes, and we showed previously that a truncating to a modest number of EZW passes does not incur excessive costs in terms of data volume or compression time. Median error is lower than 5% with five passes for both applications, regardless of system size.

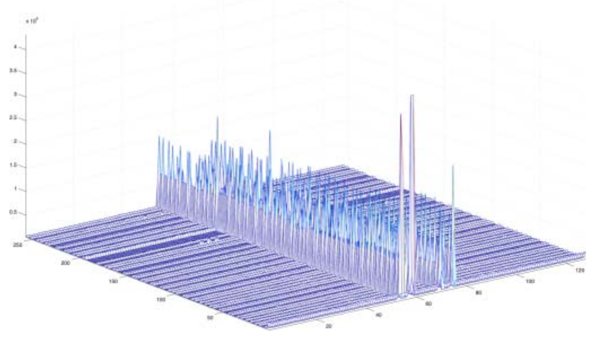
5.4 Qualitative Evaluation of Reconstruction

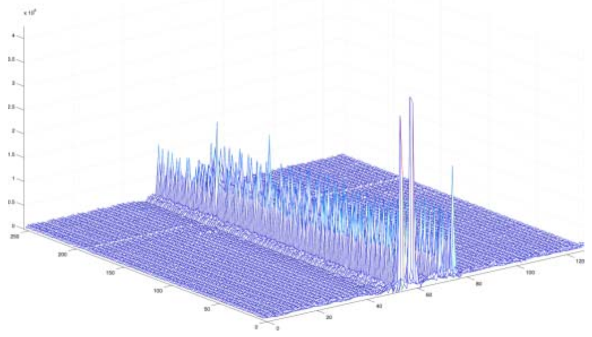
In §3 we gave a brief overview of the most useful properties of wavelet transforms for reconstructing load balance information. Specifically, we noted that the wavelet transform yields a multi-scale representation of its input data, and that it preserves local features. We ran 256 ParaDiS time steps with 128 processes on our Woodcrest cluster and plotted reconstructed effort for several phases of the application's execution. To illustrate the quality of reconstruction, we also recorded exact data for comparison. To illustrate load imbalance, we used a data set that was small enough that load could not be allocated evenly across all processes.

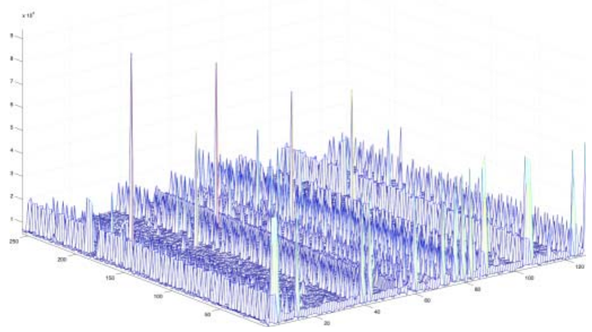
Figure 11 shows reconstructions of the effort for ParaDiS's remesh phase for varying EZW pass counts. As discussed in §5.3, lower numbers of passes correspond to higher levels of compression and larger error, which the figure reflects. With one EZW pass, the plot only crudely approximates the shape of the exact data and the entire effort plot is shifted down. With two passes, the plot, now at approximately the right position, captures the most significant peaks although finer details of the exact reconstruction are not present. After only four passes, the shape of the reconstruction is very close to that of the exact data, and small load spikes in the first few iterations have appeared. By 15 passes, the reconstruction essentially matches the exact data.

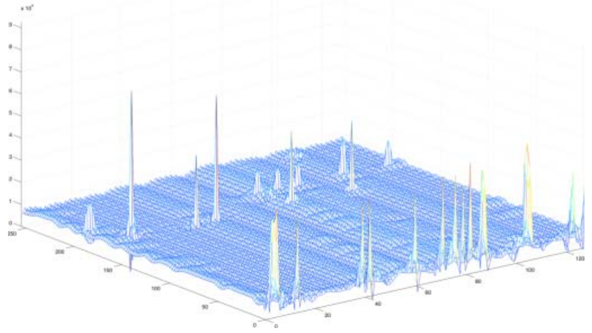
Figure 12 shows exact and reconstructed effort for four phases of ParaDiS. Figures 12(a) and 12(b) show the load distribution for ParaDiS's force computation. This phase, which is the most computationally intense region of ParaDiS, calculates forces on crystal dislocations. The reconstruction very closely matches the original data. Both clearly have two sets of processors where load is concentrated for the duration of the run. These sets correspond to the processes to which most of the initial data set was allocated. The reconstruction preserves the initial peaks, as well as finer details in the ridges that follow for both process sets.

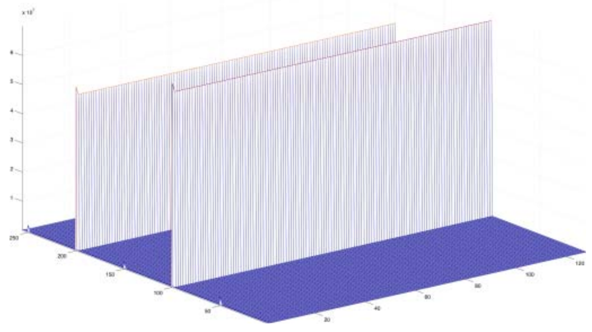
Figures 12(c) and 12(d) show the effort for collision computation in ParaDiS. We selected this phase because it il-

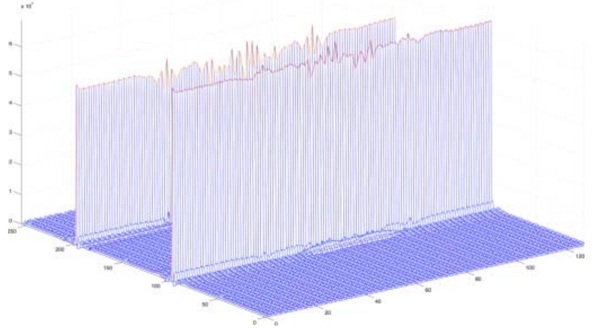

(a) Force Computation, Exact

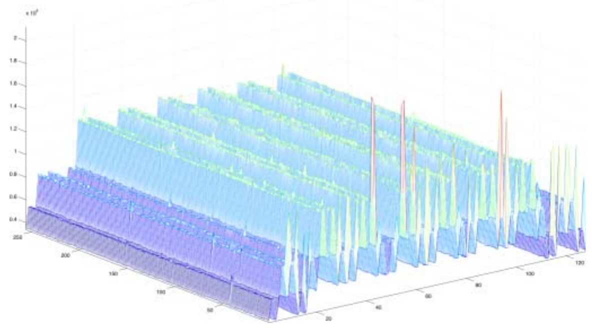

(b) Force Computation, Reconstructed


(c) Collision computation, Exact


(d) Collision computation, Reconstructed


(e) Checkpoint, Exact


(f) Checkpoint, Reconstructed


(g) Remesh, Exact

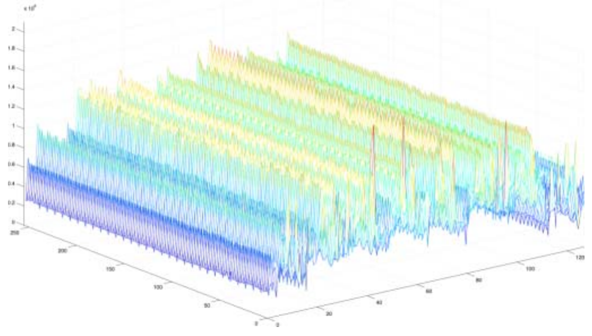

(h) Remesh, Reconstructed

Figure 12: Exact and reconstructed effort for a 128-process, 256-timestep run of ParaDiS on our Woodcrest cluster. All measurements are in nanoseconds (X axis: rank 0-127, Y axis: progress step 0-255). Reconstructions use four EZW passes.

lustrates the preservation of transient load. The collision computation is data-dependent in that it occurs only when simulated dislocations collide with one another. Our data has numerous spikes in the load and our compression framework preserves the larger ones. Noisy high frequency data at the base of the spikes is not preserved with only four EZW passes, but could be preserved using more passes.

Figures 12(e) and 12(f) show the load in the checkpoint phase of ParaDiS. For these runs, checkpoints were written to disk every 100 timesteps, and the load on all processes increases at this point. The two system-wide load spikes are clearly visible in the reconstruction at the same time steps at which they occurred in the original execution. Although the top of the spikes are slightly distorted, the reconstruction is almost identical to the original data.

The remesh phase of ParaDiS, shown in Figures 12(g) and 12(h), involves uneven load across processors, as well as variable-frequency data. With only four passes, our technique is unable to capture all detail, but major features are still present. The reconstruction preserves three spikes in the initial iteration as well as the six ridges that run through all time steps. Though not exact, this reconstruction is more than sufficient for characterizing system-wide load distribution and guiding optimization. And, as Figure 11 shows, we can increase the number of passes stored at a slight cost in compression if more detail is required.

6. RELATED WORK

Traditional performance analysis tools are insufficient for monitoring load-balance information on large systems. Profiling schemes [12, 23, 32] provide a whole-run view of application performance, but they discard timing information that is critical to understanding load evolution. Full event tracing tools [5, 12, 32] preserve timing information, but they fail to scale on large systems as their storage requirements are linear both in time and in system size.

MRNet [29] is a software overlay network that provides a generic interface to efficient multicast and reduction operations for scalable tools. MRNet uses a tree of processes between a tool front-end and a tool back-end to improve communication and to minimize perturbation. MRNet would enable us to collect effort model data at runtime, rather than post-mortem. We plan to incorporate it into a future version using the tool integration layer $P^N MPI$ [30].

SimPoint [28] detects application phases based on basic block vectors (BBVs). It provides highly accurate results for architectural simulations. However, the overhead of BBV collection is too high for monitoring direct measurements. Our effort region splitting method is similar in spirit, but its simplicity gives us low enough overhead to operate on active communication traces.

TAU [32] and ompP [14] support *phased profiling*, which captures the evolution of performance metrics over time without the unacceptable data volume of event traces. Our tool is similar, but it collects aggregate time for effort regions within progress steps instead of profile data within phases, and our data volumes are lower than those of phased profiles. The approaches are complementary; our techniques could be used to compress phased profile data at scale.

Previous efforts have used compression to efficiently store trace data. The Open Trace Format [21] supports compressed trace files using Zlib [13]. ScalaTrace [26] compresses traces on-line at the event level, and it performs additional inter-process compression at the end of execution. These techniques are lossless, and they are well-suited to full event traces. Our technique achieves very high rates of lossy compression for numerical data.

Existing work has addressed the use of the wavelet transform on parallel application trace data. The transform has been used to reduce post-mortem trace volume [9], and its output has been analyzed to extract program phase behavior [7, 18] and program structure [8]. Our tool differs from these in that it performs a parallel wavelet transform at runtime instead of in a post-mortem analysis step. We currently only use the wavelet transform for compression, but we are investigating its use for online, interprocess analysis.

Distributed sensor networks, especially when battery powered and using radio communication, are extremely limited in terms of both bandwidth and the total amount of information they can afford to collect. Given these severe constraints, the Compass project at Rice [34] has explored the use of wavelet techniques to efficiently manage sensed data, especially in the context of detecting anomalies in sensed physical fields.

7. CONCLUSIONS AND FUTURE WORK

Load imbalance is one of the most important factors limiting scalability in large scale parallel computation. Hence, understanding its impact and source is an essential step in improving application performance. For large, long-running jobs, however, the volume of data that must be collected and analyzed is prohibitive. Programs with evolving performance profiles are especially challenging since analyzing the problem requires detailed traces that record large numbers of events per process. This is infeasible with traditional approaches.

In this paper, we presented a novel approach to system-wide monitoring that achieves several orders of magnitude of data reduction and sublinear merge times, regardless of system size. We introduced a model for high-level load semantics in SPMD applications that can lend insight into performance problems. Using aggressive compression techniques from signal processing and image analysis, our approach can reduce and aggregate distributed load data to accommodate significant I/O bottlenecks. Additionally, our approach achieves very low error and high speed, even at the highest levels of compression.

We demonstrated our novel load-balance analysis framework using two actively used, full applications with dynamic behavior: Raptor and ParaDiS. Our framework is capable of efficiently handling both applications and captures information that has yielded insight into the evolution of load-balance problems, as demonstrated in our qualitative study of ParaDiS. Additionally, our evaluation showed that, even with timing and rank information, the size of the data files grows slowly with the number of processors and hence enables detailed measurement even at large scales. Further, we demonstrated that significant qualitative features of compressed data are preserved by our framework, even for very small compressed file sizes.

We showed with post-mortem data collection that our wavelet compression technique can achieve merge times suitable for online monitoring of production codes. We are currently hardening our tool for use as a component in an online monitoring framework, and we are adding case studies of additional applications at larger scale. Also, we plan to study how to detect progress loops in MPI traces without explicit instrumentation. We believe that MPI traces contain sufficient information to extract the main time step loop from most parallel applications. We plan to produce a fully automatic and transparent framework to efficiently analyze and optimize the load-balance behavior of any SPMD code.

8. REFERENCES

[1] M. D. Adams. The JPEG-2000 still image compression standard. Technical Report 2412, ISO/IEC JTC 1/SC 29/WG, December 2002.

[2] M. D. Adams and F. Kossentini. JasPer: a software-based JPEG-2000 codec implementation. In *Proceedings of the International Conference on Image Processing*, 2000.

[3] L.-M. Ang, H. N. Cheung, and K. Eshragian. EZW algorithm using depth-first representation of the wavelet zerotree. In *Fifth International Symposium on Signal Processing and its Applications (ISSPA)*, pages 75–78, Brisbane, Australia, August 1999.

[4] 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.

[5] H. Brunst, H.-C. Hoppe, W. E. Nagel, and M. Winkler. Performance optimization for large scale computing: The scalable VAMPIR approach. In *Computational Science - ICCS 2001*, pages 751–760, May 2001.

[6] 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.

[7] M. Casas, R. M. Badia, and J. Labarta. Automatic phase detection of MPI applications. *Parallel Computing: Architectures, Algorithms, and Applications*, 38:129–136, 2007.

[8] M. Casas, R. M. Badia, and J. Labarta. Automatic structure extraction from MPI applications. In *European Conference on Parallel Computing (Euro-Par)*, pages 3–12, 2007.

[9] M. Casas, R. M. Badia, and J. Labarta. Automatic analysis of speedup of MPI applications. In *International Conference on Supercomputing (ICS)*, pages 349–358, Kos, Greece, June 7-12 2008.

[10] 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.

[11] I. Daubechies. *Ten Lectures on Wavelets.* SIAM: Society for Industrial and Applied Mathematics, 1992.

[12] L. De Rose and D. A. Reed. SvPablo: A multi-language architecture-independent performance analysis system. In *International Conference on Parallel Processing*, 2000.

[13] P. Deutsch and J.-L. Gailly. ZLIB compressed data format specification version 3.3. RFC 1950, Internet Engineering Task Force, May 1996.

[14] K. Fürlinger and M. Gerndt. ompP: A profiling tool for OpenMP. In *Proceedings of the First International Workshop on OpenMP (IWOMP)*, 2005.

[15] 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.

[16] 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.

[17] D. A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the Institute of Radio Engineers*, 40(9):1098–1101, September 1952.

[18] T. Huffmire and T. Sherwood. Wavelet-based phase classification. In *PACT'06*, pages 95–104, September 16-20 2006.

[19] R. Kutil. Approaches to zerotree image and video coding on MIMD architectures. *Parallel Computing*, 28(7-8):1095–1109, August 2002.

[20] S. Louis and B. R. de Supinski. BlueGene/L: Early application scaling results. In *NNSA ASC Principal Investigator Meeting & BG/L Consortium System Software Workshop*, February 2005. Available from: http://www-unix.mcs.anl.gov/~beckman/bluegene/SSW-Utah-2005/BGL-SSW22-LLNL-Apps.pdf.

[21] A. D. Malony and W. E. Nagel. The open trace format (OTF) and open tracing for HPC. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 24, New York, NY, USA, 2006. ACM.

[22] X. Martorell, N. Smeds, R. Walkup, J. R. Brunheroto, G. Almási, J. A. Gunnels, L. De Rose, J. Labarta, F. Escalé, J. Gimenez, H. Servat, and J. E. Moreira. Blue Gene/L performance tools. *IBM Journal of Research and Development*, 49(2-3):407–424, 2005.

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

[24] MPI Forum. MPI: A message passing interface standard. *International Journal of Supercomputer Applications and High Performance Computing*, 8(3/4):159–416, 1994.

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

[26] 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.

[27] Paradyn Project, Madison, WI. *DynStackwalker Programmer's Guide*, July 13 2007. Version 0.6b. Available from: http://ftp.cs.wisc.edu/pub/paradyn/releases/current_release/doc/stackwalker.pdf.

[28] E. Perelman, M. Polito, J.-Y. Bouget, J. Sampson, B. Calder, and C. Dulong. Detecting phases in parallel applications on shared memory architectures. In *International Parallel and Distributed Processing Symposium (IPDPS)*, 2006.

[29] 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.

[30] M. Schulz and B. R. de Supinski. P^{N}MPI tools: A whole lot greater than the sum of their parts. In *Supercomputing 2007 (SC'07)*, 2007.

[31] J. M. Shapiro. Embedded image coding using zerotrees of wavelet coefficients. *IEEE Transactions on Signal Processing*, 41(12):3445–3462, December 1993.

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

[33] B. Sprunt. Pentium 4 performance-monitoring features. *IEEE Micro*, pages 64–71, 2002.

[34] R. S. Wagner, R. G. Baraniuk, S. Du, D. B. Johnson, and A. Cohen. An architecture for distributed wavelet analysis and processing in sensor networks. In *Information Processing in Sensor Networks (IPSN06)*, pages 243–250, New York, NY, USA, 2006. ACM Press.

[35] D. F. Walnut. *An Introduction to Wavelet Analysis.* Birkhäuser Boston, 2004.