ABSTRACT

RATN, PRASUN. Preserving Time in Large-Scale Communication Traces. (Under the direction of Professor Frank Mueller).

Analyzing the performance of large-scale scientific applications is becoming increasingly difficult due to the sheer size of performance data gathered. Recent work on scalable communication tracing applies online interprocess compression to address this problem. Yet, analysis of communication traces requires knowledge about time progression that cannot trivially be encoded in a scalable manner during compression.

We develop scalable time stamp encoding schemes for communication traces. At the same time, our work contributes novel insights into the scalable representation of time stamped data. We show that our representations capture sufficient information to enable what-if explorations of architectural variations and analysis for path-based timing irregularities while not requiring excessive disk space. We evaluate the ability of several time-stamped compressed MPI trace approaches to enable accurate timed replay of communication events. We evaluate our timing methods against various stages of compression to study the effects of compression on timing accuracy. Specifically, we measure accuracy by comparing the original application execution times with the compressed trace replay times. Our lossless traces are orders of magnitude smaller, if not near constant size, regardless of the number of nodes while preserving timing information suitable for application tuning or assessing requirements of future procurements. Our results prove time-preserving tracing without loss of communication information can scale in the number of nodes and time steps, which is a result without precedent.

Preserving Time in Large-Scale Communication Traces

by
Prasun Ratn

A thesis submitted to the Graduate Faculty of
North Carolina State University
in partial fullfillment of the
requirements for the Degree of
Master of Science

Computer Science

Raleigh, North Carolina

2008

APPROVED BY:

Dr. Xiaohui Gu	Dr. Xiaosong Ma

Dr. Frank Mueller
Chair of Advisory Committee

DEDICATION

Dedicated to my family.

Ma

Papa. Didi. Ila.

BIOGRAPHY

Prasun (Pra-soon) was born in a little town called Hazaribagh (Thousand Gardens) in India's Chotanagpur plateau. He grew up in a nearby town famous for it's mental asylums and went to college in another small town known for it's famous college. After graduating with a degree in Computer Science and Engineering, he moved to India's Silicon Valley where he found neither silicon nor any valley. There he spent valuable years of his life making camerphones for clients across the globe. In the fall of 2006, he started his master's degree in Computer Science at NC State.

ACKNOWLEDGMENTS

First of all I would like to thank Dr Frank Mueller for his support, guidance, ideas and faith in me. I am very grateful towards Bronis de Supinski for letting me work at the Lawrence Livermore National Laboratory last summer and both Bronis and Martin Schulz for their guidance and support. I would like to thank Dr Xiaosong Ma, Dr Xiaohui Gu and all other members of the Systems Research Group. I would also like to thank Todd, Vivek, Arun and my other labmates for their help and inputs. I would also like to thank my roommates for being supportive.

TABLE OF CONTENTS

LIST OF FIGURES

Chapter 1

Introduction

1.1 Background

There are many definitions as to what a supercomputer is, but supercomputing has been around for decades. The first usage of this term dates back to 1929[1]. Supercomputers came into prominence in the 1970s and 1980s when the market was dominated by large vector-based machines. The 1990s saw a decline in the market but, off late, the trend has picked up again. The IBM Roadrunner [2] supercomputer at Los Alamos National Laboratory has recently replaced the IBM Bluegene/L[3] at Lawrence Livermore National Laboratory as the fastest computer in the world and in the process achieved the 1 Petaflop (10^{15} floating point operations) per second computing landmark.

Such machines are widely used for simulations, model fitting or data analysis etc. in various areas such as nanoscience, biotechnology, climate research, astrophysics, chemistry, fusion research, drug research, homeland defense, nuclear technologies and many other fields. At centers like the Renaissance Computing Institute (RENCI) in North Carolina, supercomputers are employed for weather forecasting. For the weather domain, results are only useful if available prior to the time range forecasted. Not only do these machines speed up calculations by many orders of magnitude, they are sometimes useful in solving problems that may otherwise be unsolvable without this huge number crunching capability. For example, these days nuclear research wholly relies on simulations because actual experimentation is prohibitive due to various economic and political reasons. In the area of molecular dynamics, petascale computing enables longer timescale simulations involving larger numbers of atoms than in the past. This opens the door to unprecedented quantitative comparison of theoretical calculations for wide range of experiments. Thus,

[1] "SUPER COMPUTING MACHINES SHOWN" http://www.columbia.edu/acis/history/packard.html
[2] http://www.lanl.gov/roadrunner/
[3] https://asc.llnl.gov/computing_resources/bluegenel/

such machines bring about a quantitative as well as a qualitative shift in the computing landscape for scientific research.

Since traditional supercomputers are expensive, an alternate solution for achieving high-performance is to create "commodity clusters". These are clusters of ordinary desktop computers connected by a high-speed interconnect network. Off late, architectures designed specifically for commodity usage, such as graphics processors and gaming consoles, have also made inroads into the high-performance computing field. This is because these systems are geared towards highly parallelizable applications such as 3D physics computations, graphics rendering etc. In fact, a cheap supercomputing solution has been developed at NC State[4] using a cluster of Playstation 3 gaming consoles.

Meanwhile, considerations such as slow down of single processor performance and power consumption have led vendors to deploy low power processors connected using high-speed interconnects. For example, the Bluegene supercomputer from IBM uses a low power embedded RISC Power processor as a basic building block. The IBM Roadrunner supercomputer also uses commodity parts in its hybrid design.

The common theme here, between high-end supercomputers and cheap commodity clusters, is the growing importance of interconnect networks, such as Infiband, Gigabit Ethernet, Myrinet etc. Parallel performance was traditionally measured in terms of computational speedup, defined by rules such as Amdahls' Law, Gustafson's Law etc. Later, a new model called the LogP model [1] was introduced, that took into account communication properties, namely latency(L), overhead (o) and bandwidth (g), in addition to pure processor performance(p).

The most common programming model deployed on such systems is the message passing model. The basic idea here is to launch computational tasks on each "node" (*i.e.* a basic parallel computational unit consisting of one or more microprocessors connected to other nodes via an interconnect) operating on local data. These tasks communicate with each other using "messages" for synchronization or for data transfer. In practice, the programs running on each node are identical but operate on a different set of the input data. This model is called Single Program Multiple Data (SPMD). Among message passing models, the most common is the Message Passing Interface (MPI)[5].

There are various other hardware and software parallel solutions, but we will limit our discussion to the above context.

[4] http://moss.csc.ncsu.edu/ mueller/cluster/ps3/

[5] http://www.mpi-forum.org/

1.2 Motivation

Analyzing parallel applications is becoming increasingly difficult for large-scale systems. Most analysis tools either succinctly capture performance data or support pinpoint identification of inefficiencies, yet to combine both capabilities has been an elusive goal. These limitations arise from the techniques to obtain performance data. Aggregated data provides overall statistical information within a single data set of small size in a scalable manner for large numbers of nodes. Detailed traces, in contrast, on a per-node basis facilitate the detection of bottlenecks in time and location, yet inhibit scalability due to the large number and size of trace files.

Recent advances in online trace compression promise a solution for lossless trace recording [2]. This technique scalably achieves the "best of both worlds" by combining a succinct representation while retaining temporal ordering and location-specific information. Its scalable compression of communication traces captures detailed data in a single data set of small size and provides the ability to replay communication events. However, its representation lacks time stamp information for communication events, which is essential to determine bottlenecks and to facilitate what-if analyses that assess the impact of changing key system architecture aspects such as network bandwidth or latency.

1.3 Hypothesis

We contend that it is possible to capture and encode timing information using the above mentioned scalable tracing methods with reliable accuracy. We also argue that this can be achieved without sacrificing the lossless and size-scaling properties of the original method. We further contend that the original tracing methods are subject to optimizations that result in improved scalability results.

1.4 Contributions

This work contributes a set of time-preserving compression techniques for communication traces. We record "delta" times representing the respective communication and computation between and during communication events instead of traditional time stamps. A first technique utilizes simple statistical delta times on a per-event basis. A second approach employs histograms with a fixed number of bins for delta times. We combine this with a dynamic rebalancing scheme to equalize the number of items per bin while adjusting their value range constraints. A third method refines the previous one by distinguishing histograms not only per call stack but also by path sequence. Path-sensitivity allows the distinction of deltas whose values vary significantly depending on path origination with respect to loops.

1.5 Evaluation

We evaluate these techniques in the context of both per-node compression with distinct trace files per node and global compression, which consolidates data from different nodes over a reduction tree into a single trace file. Our approach is platform independent, but due to our resource and time limits, we performed full scale runs (upto 512 nodes) were possible on a 2048-node BlueGene/L (BG/L) machine at RENCI. We evaluated our framework using the NAS Parallel (MPI) Benchmark Suite[3] and the UMT2000 benchmark from the ASCI Purple Benchmark Suite[4] from LLNL. Experimental evaluations on the BlueGene/L (BG/L) platform demonstrate that the schemes capture sufficient timing information to detect communication inefficiencies and to locate their source while incurring scalable costs as the number of nodes or time steps increases. Timed replay of the traces is highly accurate with wall-clock time errors between -8% to 14% for per-node and -20% to 7% for global compression for a wide array of applications.

Overall, these results demonstrate that timed tracing without loss of communication events can scale in the number of nodes and time steps. The replay accuracy demonstrates that what-if emulators such as Dimemas [5] could use our delta-based traces with little loss in accuracy. We also could easily use them with timeline-based trace analysis tools commonly employed in today's development cycle of high-performance applications, like Vampir [6] or Jumpshot [7]. Importantly, our small trace files would allow the display and efficient scrolling of communication events without decompression from a single workstation. No longer will these tasks require complex visualization back-ends currently employed to handle long traces from many nodes.

1.6 Summary

In summary, we put forward a scheme to record timing information without sacrificing space scalability while guaranteeing accuracy. We also contribute optimizations to the trace recording framework that enhance trace scalability.

Chapter 2

Scalable Trace Compression

This work on time compression and replay builds on previous work on MPI tracing [2]. This scalable trace-compression framework is referred to as *ScalaTrace*. Previous work on ScalaTrace showed online trace compression can result in trace file sizes an order of magnitude smaller than previous approaches or, in some cases, even near constant size regardless of the number of nodes or application run length. While the original approach successfully compressed large-scale traces, one of the open questions was how scalability could be retained when timing information was encoded in such traces. We concisely describe ScalaTrace to provide the context for our solution to this challenge.

ScalaTrace uses the MPI profiling layer (PMPI) to intercept MPI calls during application execution. On each node, profiling wrappers trace all MPI functions, recording their call parameters, such as source and destination of communications, but without recording the actual message content. This intra-node information (task-level) is compressed on-the-fly. In addition, inter-node compression is performed upon application termination to obtain a single trace file that preserves structural information suitable for lossless replay.

Intra-Node Compression: ScalaTrace exploits application looping behavior to compress MPI call entries on-the-fly within each node. Regular section descriptors (RSDs) capture MPI events nested in a single loop in constant size [8] while power-RSDs (PRSDs) specify recursive RSDs nested in multiple loops [9]. 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 the former loop (RSD1) followed by a barrier. These constructs correspond to the code in Figure 2.1.

ScalaTrace uses several optimizations to compress events efficiently. These in-

```
compute_1();
for (i = 1; i < 1000; i++) {
 for (k = 1; k < 100; k++) {
  MPI_Send(...); /* send call 1 */
  MPI_Recv(...); /* recv call 1 */
  compute_2();
 }
 MPI_Barrier(...); /* barrier call 1 */
 compute_3(...);
}
```

Figure 2.1: Sample Code for PRSDs

clude efficient representations of stack walks, location-independent end-point encodings and aggregation of constructs, like MPI_Waitsome, for which the repetitions can vary from run to run. The algorithmic details of intra-node compression can be found elsewhere [2].

Inter-Node Compression: ScalaTrace combines local traces into a single global trace upon application completion within the PMPI wrapper for MPI_Finalize. This approach is in contrast to generating local trace files, which results in linearly increasing disk space requirements and does not scale as traces must be moved to permanent (global) file space. The I/O bandwidth, particularly in systems like BG/L with a limited number of I/O nodes, could suffer severely under such a load. ScalaTrace provides scalability through cross-node compression in a step-wise and bottom-up fashion over a binary tree. ScalaTrace merges events and structures (RSD/PRSDs) of nodes when events, parameters, structure and iteration counts match. Another set of generic and domain-specific optimizations, including the use of a radix tree in the merge step, ensures efficient inter-node compression. Further algorithmic details on the inter-node compression scheme are again available elsewhere [2].

We also introduce optimizations for both intra-node and inter-node compression schemes, such as special encodings, variable parameter matching and more. We talk about these in detail in Chapter 4.

The Challenge of Scalably Encoding Timing Information: Traditional tracing techniques annotate each event with a time stamp. ScalaTrace's compression schemes complicate encoding this information. Both mechanisms exploit repeated behavior to capture temporal event ordering in limited space. Time stamps necessarily vary with each

event. Further, time stamps introduce the problem of clock synchronization across the compute nodes. Even the interval between events will vary due to system interference and other effects that are not repeated behavior, which makes them counter to ScalaTrace's underlying philosophy. We discuss the time compression strategies in the following chapter.

Chapter 3

Preserving Time

As mentioned in the previous chapter, the focus of this work is the challenge posed by incorporating time information into the trace. To provide information on a timeline, we depart from traditional absolute time stamps in favor of relative "delta" time.

```
pre-wrapper {
 recordTime(); /* compute delta */
 ...
 resetTime ();
}
PMPI_Op (...);
post-wrapper {
 recordTime() /* communicate delta */
 ...
 resetTime ();
}
```

Figure 3.1: Delta Calculation

Delta times are recorded between adjacent communication events, both prior and after the actual communication routine is executed. All time spent outside MPI calls is treated as computation time and all time spent inside MPI calls (waiting time and actual communication time) is treated as communication time. Hence, the time difference between the pre- and post-wrapper of a communication routine denotes the communication overhead delta while the time differnce between the post-wrapper of one routine and the pre-wrapper of the next denotes computation time delta, as depicted in Figure 3.1.

Repetitive event sequences often have similar delta timings in regular SPMD

codes. However, a fine-grained clock skews delta times sufficiently to inhibit compression. Thus, we use statistical methods with increasing precision but also storage cost instead of recording the exact deltas. Our primary objective is to trade space for increasingly precise retention of timings. We also intend to preserve location information (such as node IDs) about outliers that may indicate communication inefficiencies, such as load imbalance.

We have augmented ScalaTrace and its associated replay engine (discussed below) with three timing encoding methods. First, we designed a low-cost statistical approach to capture aggregate delta times. Second, to capture computational imbalance, we developed a variation-preserving recording scheme using time delta histograms, still with a constant size trace representation, yet with a higher constant factor. Third, we extended the histogram approach with additional context information to distinguish different preceding MPI calls.

3.1 Statistical Delta Times

Our first technique records simple statistics of delta times on a per-event basis. We annotate each trace record with a single value for its associated compute and communicate deltas. Thus, we capture the program's entire compute and MPI timing characteristics through per-record annotations.

We associate compute and communicate deltas with each trace event corresponding to the wrapper in which it is captured. When combining trace records, we encode in the new RSD aggregate statistical time information including the maximum, minimum, average and variance of deltas in the records. Similarly, we encode these aggregate statistics when trace records are merged across nodes.

Compute deltas are collected for event pairs. For the example in Figure 2.1, we capture two compute deltas: (a) from the send to the receive and (b) from the receive to the send (between consecutive loop iterations). Further, we distinguish timing deltas from different entry paths and store the statistics for each path separately. This enables us to treat loop iterations differently from the actual loop bodies and hence leads to higher accuracy.

Statistical aggregation can detect bottlenecks that accompany events that occur with a repeated temporal order, unlike traditional profiling techniques. It also supports approximate communication replay such that relative timing is preserved at a coarse grain. It allows one to determine if significant imbalances exist by consulting minimum, maximum and variance. However, it does not capture finer-grained inefficiencies. This simple statistical approach also does not sufficiently characterize the distribution of time deltas in the min-max range. A single outlier can affect the average. Similarly, the aggregate statis-

tics misrepresent multi-modal distributions. Even with the standard deviation, we cannot detect several important types of anomalies or recreate the relative timing of individual events with high accuracy. Thus, we often need more advanced techniques to record delta times.

3.2 Dynamically Balanced Histograms

Our second approach uses histograms of delta times. By using a fixed number of histogram bins per trace entry, we retain ScalaTrace's desirable compression properties. The histograms capture outliers and other timing distribution properties missed by the aggregate statistics and, thus, can provide more insight into bottlenecks and computational imbalance.

We linearly divide the range of delta times associated with an event pair into k subrange bins, where k is a user-defined constant value. We increment the counter corresponding to subrange x when we observe a delta time of subrange $x \in \{1..k\}$. This scheme allows finer-grained analysis of the amount of imbalance between nodes, and it can be refined in repeated analysis runs by increasing the number of bins.

A key challenge for this scheme is anticipating the range and distribution of the delta times for an event pair, neither of which is known *a priori*. We address this challenge with a dynamic rebalancing scheme that equalizes the number of items per bin while adjusting their value range constraints. Our rebalancing scheme uses a weighted subrange partitioning scheme to achieve this goal. We dynamically expand existing value ranges when we observe a new extreme delta time. In essence, the length of a subrange monotonically increases with each new extreme value, and the existing histogram is adjusted on-the-fly.

Algorithm 1 shows our rebalancing scheme's initial stage. We arbitrarily assume the first value is the center of the delta distribution when we create the histogram. We set the range to twice that value and create a fixed number of equal-sized bins. We rely on our rebalancing scheme to automatically correct the bin sizes and ranges. This schemed is described below.

Algorithm 1 HistogramCreate(value v, numbins k)

range $\leftarrow 0 - 2 * v$
num-bins $\leftarrow k$
bin-size $\leftarrow 2v/k$

Algorithm 2 presents the dynamic rebalancing scheme for histograms. New values may be unevenly distributed across bins. Thus, bins are dynamically balanced by changing

Algorithm 2 HistogramAdd(value v)

find bin such that $bin_{min} \leq v$ and $v \leq bin_{max}$
$bin_{freq} \leftarrow bin_{freq} + 1$
$avg = bin_{avg}$
if $bin_{freq} = 1$ **then**
    $bin_{avg} = v$
else
    $bin_{avg} = bin_{avg+}(v - bin_{avg})/bin_{freq}$
end if
if $bin_{freq} = 1$ **then**
    $bin_{variance} = 0$
else
    $bin_{variance} = bin_{variance} + (v - avg) * (v - bin_{avg})$
end if
if bin_{freq} mod $smooth\_interval = 0$ **then**
    find B, such that B_{freq} is highest
    find $b1$, $b2$ such that $|b1_{freq} - b2_{freq}| < m_1$
      and $b1_{freq} + b2_{freq} \leq k * B_{freq}$
    merge $b1$, $b2$ into b such that
      $b_{min} = b1_{min}, b_{max} = b2_{max}$
    split B into $B1$, $B2$ such that
      $B1_{min} = B_{min}, B1_{max} = B_{avg}$
      $B2_{min} = B_{avg}, B1_{max} = B_{max}$
    update $b_{avg}, B1_{avg}, B2_{avg}$
    update $b_{variance}, B1_{variance}, B2_{variance}$
end if

the bin sizes appropriately. When a value is added to a bin, the minimum, maximum, average and variance for the bin are updated. By specifying a *smooth_interval* threshold (normally expressed as a percentage; when the bin sizes are very small, an absolute value is used as a threshold), one can control when rebalancing is triggered in response to bin sizes. Upon a rebalance action, the algorithm locates small, adjacent bins that can be combined and a large bin that can be split. While merging bins, the new bin takes the extreme values from the appropriate bins, the new average is the weighted average, and the new variance is updated accordingly assuming a uniform distribution [10]. Splitting a bin is more complex. We split the bins along the old average. Assuming a uniform

distribution, we set the new bin averages to be at a distance of sigma/2 on either side from the old average and the new bin sigma to be half the old sigma value.

Besides bin rebalancing during intra-node compression, histograms are merged during inter-node compression. Deltas from one histogram are moved to another while preserving node-imbalance information. More specifically, histograms retain annotations with a node for the minimum and one for the maximum delta times. The complexity of the algorithms is O(1) for bin creation, addition of elements and rebalancing (*i.e.*, merging and splitting bins) since the number of bins is constant. The inter-node merge operation on histograms is O(freq), *i.e.*, linear in the items per bin. We are exploring O(1) inter-node merging in order to reduce the ScalaTrace library overheads. Overall, this method effectively creates bins that span the entire value range and hold a similar number of samples.

3.3 Path-Sensitive Delta Times

Our third method refines the previous one by distinguishing histograms not only by call stack but also by path sequence. Hence, the entry path to and the exit path from a loop can be distinguished from the iteration path within the loop. This facilitates the distinction of compute deltas that vary significantly per the execution path.

For many scientific codes, the time spent in a loop is generally uniform across iterations, but the time spent in different loops or at different nesting depths can vary significantly. Consider Figure 2.1 where the MPI_Send call lies on multiple execution paths: on the first entry into the inner loop; on subsequent entries into it; and on repeated iterations of that loop. The associated compute delta depends on the preceding function call: compute_1; compute_2; or compute_3. Each call to compute_1 might consume a similar amount of time, but calls to compute_1 and compute_2 might vary substantially.

In our third method, we keep a list of delta histograms, each storing the timing information for a different path sequence. Each call site is annotated with a stack signature, which results from XORs of the PCs upon call stack traversal [2]. These stack signatures facilitate the distinction of call paths as signature difference is sufficient for call path inequality (while a matching signature requires per-call PC comparison along a call path). We annotate each trace record with a list of histograms, and each histogram has an associated stack signature, which must be the same as the stack signature of the previous operation when being combined by the compression algorithm. While merging records, a delta must be inserted into the appropriate histogram. This is determined by matching the previous operation's stack signature with the histograms in the list. If no matching histogram is found, a new one is created.

3.4 Time-Preserving Replay

An objective of collecting communication traces is their off-line analysis. Analysis tools, including timeline-based visualization tools, can directly operate on the trace. Communication traces also support generic replay of communication events without using the application code. This mechanism supports what-if analyses, as with Dimemas [5].

We have designed and implemented a replay engine that issues communication calls in the same order that they were originally issued by an application. Our replay engine does not decompress the trace. Instead, it interprets the compressed trace on-the-fly to issue communication calls. In effect, the replay engine implements the inverse functions of the compression algorithms. When it encounters an RSD or PRSD, it issues calls iteratively observing the structure, frequency and parameters of communication calls. Consider the tuple RSD1:<100, MPI_Send1, MPI_Recv1> again. Upon replay, 100 pairs of MPI_Send and MPI_Recv calls are issued with the respective parameters (omitted here for presentation purposes). Similarly, the tuple PRSD1:<1000, RSD1,MPI_Barrier1> is replayed by issuing 1000 pairs of RSD1 events and MPI_Barriers. The structure-preserving compression scheme is key to a scalable replay methodology, which does not require excess amounts of memory. In fact, the *compressed* trace size, which is often constant, loosely bounds its memory requirement.

Replay triggers all MPI calls over the same number of nodes with original payload sizes, yet with a *random* message content. We capture any data dependence arising from the communication in the traced message schedule so the replay reflects it. Thus, the replay incurs comparable bandwidth requirements on communication interconnects. However, the communication could exhibit different contention characteristics in the absence of timing information.

Our timing encoding methodologies provide a means to address this shortcoming. Our replay engine emulates computation by *sleeping* using nanosleep() to delay the next communication event by the proper amount of time (BG/L does not support nanosleep() so we use a busy-wait loop instead). It simply replays communication using the same end points.

Our timed replay implementation varies with the time encoding scheme.

With aggregate statistics, we simply replay the average times and ignore the extreme values. More sophisticated replay is possible, including the use of extreme values or choosing delays from a distribution. We select replay delays to reflect the distribution across bins accurately for histogram-based replay. Similar to our use of aggregate statistics, we replay average times within each bin. For inter-node compressed traces, we have the extreme node information, which we utilize to replicate the imbalance in compute

deltas. Path-sensitive replay adds another level of complexity. By considering the current communication event in conjunction with the previous one, we select the appropriate histogram to determine the correct delta value.

Overall, our replay mechanism is extremely portable, which can benefit rapid prototyping and tuning, albeit without any guarantee to resemble equivalent computational overhead when record and replay platforms are heterogeneous. Further, it serves as a guide for using our traces with emulators, which also require some method to scale compute deltas. Scaling compute deltas would require performance prediction, which is an area beyond the scope of this work.

3.5 Search in Time

Searching on the time axis is a common operation in trace visualization. Many visualizers perform a binary search on flat traces consisting of all events. Due to traditional trace file sizes, this search may be out-of-core (often on hard disk), which presents a significant performance impediment. Recent work has improved this approach by indexing trace files, which allows an in-core search on the indices. Our compressed trace (and not just a selected index set), in contrast, fits in core memory and has a much lower complexity.

To accelerate searches over a compressed trace, we construct a timing tree. Leaf nodes represent events (with delta times), interior nodes correspond to loops with weights (number of iterations of the loop), the root represents the main application level with one iteration, and edges are connecting loops to events or inner-more (nested) loops with weights (aggregate delta time range within current loop).

A search for an instance in time then has a time complexity of $O(h \times log(v))$, where h is the maximal horizontal dimension (sequence of timed events in one nesting level) and v is the maximal vertical dimension (number of nesting levels of loops containing traced operations). The former is loosely bounded by the number of calls in the source program, irrespective of their calling context (*i.e.*, without considering the call stack or any dynamic information) while the latter is bounded by the depth of loops (number of nesting levels) over the program's call graph. Since the nesting depth is generally small (typically no larger than ten), the complexity is independent of properties of the uncompressed, flat trace. In contrast, the complexity of a binary search is logarithmic to the size of the flat trace, which can be prohibitively large in terms of memory requirements as explained above so that the trace is forced out of core memory. Even if selected indices are utilized, the search remains logarithmic with a tunable constant factor (the fraction of selected indices), which does not change the complexity and also requires a second-level finer-grained search within the range between two selected indices.

To illustrate a search over a timing tree, consider a compressed trace consisting of pairs of (event,delta time) of (e1,1) and (e2,2) at the level of the program and events (e3,1), (e4,2) and (e5,2) within a loop of ten iterations or simply:

$PRSD1 = ((e1, 1), (e2, 1), PRSD2, iters = 1)$

$PRSD2 = ((e3, 1), (e4, 2), (e5, 2), iters = 10)$

We represent the trace by the timing tree depicted in Figure 3.2 to perform search in the time dimension. A search for time $t = 16$ then traverses the tree from the root to the third child ($2 \leq t \leq 52$), which represents the inner loop with ten iterations. Solving inequation $2 + i * 5 \leq t \leq 2 + (i + 1) * 5$ for $t = 16$ yields $i = 2$, *i.e.*, the target time is within the second iteration of the loop. More specifically, $t = 2 + i * 5 + k$ yields $k = 2$ where $1 \leq k \leq 3$ is matching subrange 1..3. Hence, the second child is traversed to yield event e4 in the second iteration of the loop as the search target.

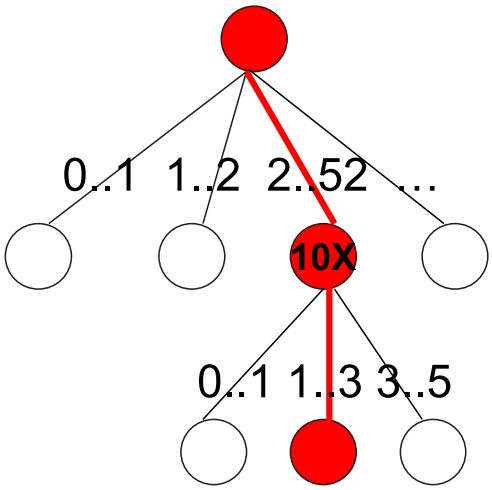


Figure 3.2: Search over Timing Tree for t=16

Chapter 4

Compression Optimizations

This work contributes a number of optimizations to the compression methods described in previous work [2]. In the initial results, the benchmarks fell into three categories with respect to trace size scalability. The three categories are, constant-size, sub-linear and non-scalable trace sizes. With our optimizations, many more benchmarks that earlier were in the non-scalable category now fall into the constant-size or sub-linear category and similarly some benchmarks which fell into the sub-linear category are now in constant-size category.

The following sections describe each optimization in detail.

4.1 Special Encodings

One of the central points, for both intra-node and inter-node compression to succeed is the way the parameters are encoded. For example, source or destination parameters are encoded as offsets and pointer parameters are encoded as lookup table indices. We contribute a number of methods to handle more such parameters in order to enhance the compression.

4.1.1 ANY_SRC

MPI_ANY_SOURCE is a special predefined MPI constant. When used in an MPI receive operation, it allows a message to be accepted from any sender, as opposed to a specified sender. Many MPI implementations, such as OpenMPI and MPICH, define this constant as a negative integer. In our default encoding, we store source parameters as offsets from the current node. If every node had an MPI receive call with MPI_ANY_SOURCE, they would record the source parameter as (MPI_ANY_SOURCE - *rank*). In this case, every node would record a different source parameter encoding.

We added a special check for MPI_ANY_SOURCE so that it is encoded as it is instead of encoding it as an offset. This ensures that all nodes have the same encoding for

this special parameter, thus enabling compression at the inter-node level.

We use a special integer value for our encoding so that the value is not interpreted as an offset when the trace is read by the replay engine.

4.1.2 ANY_TAG

MPI Send and Receive routines have a tag parameter, the semantics of which is left to the application developer. The MPI standard, however, does require that the tag parameter be matched when a message is received. Most of the time, it preserves the semantics to set the tag to MPI_ANY_TAG. However, developers some times use this parameter as a sequence count or as a direction indicator or even to specify message endpoints.

Having mismatching tags adversely affects intra-node merge when different iterations of a loop contain a send or receive message with different tags. We recognize that the tag parameter is does not have to be preserved for replay or other trace analysis. Hence, we discard tags during parameter recording in such cases and default to MPI_ANY_TAG for these messages.

An exception to this rule is where the application uses MPI_ANY_SOURCE in conjunction with the tag parameter to distinguish endpoints (*e.g.* in the LU benchmark of the NAS Parallel Benchmark Suite). In such cases, it is important to retain the tags. Right now, we compile separate versions of the library to discard or retain tags, but this can be automated in future work.

4.1.3 Keys and Colors

MPI_Comm_Split is an MPI operation used to create new communicators based on *keys* and *colors*. Usually, these values are either constant across nodes or have constant offsets from the node ranks. For example, in the BT benchmark (from the NAS Parallel Benchmark Suite), we observe the following usage:

```
call mpi_comm_split (MPI_COMM_WORLD, color, node, comm_setup, error)
```

where node is the rank and color is either 0 or 1 depending on whether the node is included in the communicator or not. Hence, we need to decide whether to encode the color and key parameters as absolute values or as relative offsets.

We devised the following strategy in the inter-node merge to tackle this problem. We decide to merge two MPI_Comm_Split events in one of the following conditions:

1. two nodes with either the same offset or the same absolute color/key; or
2. two groups with either the same offset or the same absolute color/key; or
3. a group and a node with the same offset or the same absolute color/key.

Here, a group implies an event with multiple participants whereas a node implies an event with a solitary participant. Note that the first condition dictates whether the events will be recorded with a relative or absolute parameter whereas the next two conditions decide whether or not to merge two events.

The events are labeled as relative or absolute so that the replay engine or any other trace parsing tool can read the correct values for each node.

4.1.4 MPI_Alltoallv

MPI_Alltoallv is an MPI collective in which each process sends data to all other processes; each process may send a different amount of data and provides displacement for the input and output data. The parameters involved in this call are the count and displacement arrays for send and receive. Since this operation is used to exchange variable amounts of data, the arrays do not match. We have designed a solution where we accumulate all the arrays into a single event during inter-node merge. While this solution is still non-scalable (because the arrays still grow exponentially) it allows Alltoallv events to compress during inter-node merge and further allows RSDs containing Alltoallv to compress during inter-node merge.

Another solution that we have devised (but not yet implemented) is to average the count and displacement numbers, thus effectively transforming the Alltoallv call into an Alltoall call. We studied the IS benchmark (in the NAS Parallel Benchmark suite), which has calls to MPI_Alltoallv, and noticed that the sum of the counts matched across the nodes and the variance of the actual numbers was small. For analysis purposes, sacrificing this information in order to obtain constant-size traces seems to be a good idea.

Another proposal we have is to find some middle ground between maintaining all information versus retaining aggregate information. Perhaps only retaining the counts would suffice or one could use a threshold to clip the counts (counts less than threshold are made zero) and then run length encode the counts, optionally combined with smoothing the counts over the threshold or by using histograms.

4.2 Merge Non-matching Parameters

In the previous section, we described a number of methods where specific parameter encodings benefit compression. However, we understand that we cannot always obtain proper parameter matches, not even after special encoding of parameters. Hence we put forward a technique that allows compression with approximate event matches (when not all parameters match exactly).

Let us discuss this in detail in the following sections.

4.2.1 Intra-node Merge

At the intra-node level, we already support a rudimentary variable parameter merge. This was achieved to some extent earlier by using parameter queues. It was implemented only for source (for receives), destination (for sends) and request (for waits) parameters. We now enhance this to include other parameters such as datatypes and counts. Unlike earlier, this implementation allows multiple mismatching parameters per event. This implementation has also been redesigned to allow compatibility with the inter-node compression scheme. The basic idea here is to construct an array of parameters in case of a parameter mismatch while most (other) events match. Though this still increases trace sizes linearly with the number of event occurrences, we manage to avoid duplicate event information due to compression. If we have repeated instances of an entire array, we increment the iteration count to restrict the array size. As an example, consider the following representation : *parameter iterations* $value_1$*,*$value_2$*,...,*$value_n$

4.2.2 Inter-node Merge

We take the previous method a step further by deploying it in the inter-node merge as well (although the inter-node was implemented earlier). We noticed that even when parameters do not match across all nodes, they tend to be same for groups of nodes. We take advantage of this property by creating <value, ranklist> tuples during inter-node merge. Thus, instead of having single parameters, we have an array of such tuples. We omit one ranklist whose representation takes the most number of bytes in order to limit space requirements. We can easily infer the participants for the value without ranklist because we know the participants for other values and also the participants for the entire event. Furthermore, the *value* parameter can either be a scalar value or an array of values. An array of values originates from two cases:

1. the parameter is itself an array, such as count or displacement arrays; or
2. the parameter is a result of the variable parameter intra-node merge described above.

During replay, the trace is parsed by each node. To correctly recognize a parameter, the parser iterates through the <value, ranklist> tuples and reads the value corresponding to the current rank. In case of intra-node arrays, the value in the array corresponding to the iteration step is used.

4.3 Speeding up Merge

Our analysis of the merge overheads shows that the intra-node merge overhead is quite nominal for most cases. In the case of non-scaling applications, however, the merge

overhead turns out to be significantly higher. We analyzed the inter-node merge algorithm and devised a new faster algorithm.

Algorithm 3 merge_queues(queue_t master, queue_t slave)

```
while master_iter do
    slave_iter = slave->head;
    while slave_iter do
        if match_sequence(master_iter, slave_iter) then
            /* build a list of dependencies by walking the dependence graph starting
            from slave_iter */
            yank_list = dfs (master, master_iter, slave, slave_iter);
            yank (yank_list);
            merge_nodes (master_iter, slave_iter);
            break;
        else
            go_next (slave_iter);
        end if
    end while
    go_next (master_iter);
end while
/* move any remaining ops from slave to master */
if !master_iter && slave->head then
    insert(master, master->tail, slave->head, slave->tail);
end if
```

4.3.1 A Faster Inter-node Merge Algorithm

During the inter-node merge, the events in a queue are considered independently when they do not share a participant. In other words, the relative order of two events in a queue does not matter if they do not share a participant. When we have a match, we need to move all dependent events from the queue being merged (slave queue) to the destination (master queue). This supports independent operation sequences left in the slave queue to be matched with an operation sequence from the master queue at a later time.

This dependency was earlier determined by intersecting the task participant list of all unmatched operation sequences with the task participant lists of the matched operation sequence. If the intersection is empty, there is no dependency. If there is no dependency, it is not necessary to move the operation sequence into the destination queue. Only those

operation sequences that do have a dependency must precede the master iterator while the rest remains in the slave queue.

The above mentioned step computes the task list intersection for the entire set of events preceding the slave iterator. This step was quite compute intensive and at times redundant because the dependencies had been computed at a child node.

We have developed a faster algorithm shown in Algorithm 3. We no longer need to scan the slave queue to determine the set of events in the unmatched sequence causally independent on the matched set. This is facilitated by the new causal ordering preservation strategy, which is explained in more detail below.

In our new algorithm, we maintain a dependence graph during the entire merge algorithm. At a leaf node of the reduction tree, the dependence graph simply constitutes a linked list (directed backwards in the temporal ordering of events) as the current node is a participant for each event. When a slave queue is received at non-leaf nodes from a child, the dependence graph is resurrected. (This step is linear in the size of the queue because pointer marshalling has to be done here). If a subsequence of matching events is encountered, any preceding non-matches are inspected for dependencies on the current slave event by performing a depth-first search (dfs) over the dependence subgraph originating from the current slave event (see Algorithm 3). (This is an improvement over the first-generation algorithm in that only dependencies reachable from the current event are considered.) Any dependent event is added to a yank list during the traversal, which eventually contains those events that casually depend on the participants of the current slave event (matched with a master event). Events in the yank list are inserted prior to the current event in the master queue through a yank routine, which ensures that the causal order is preserved.

The rest of the algorithm remains unchanged.

4.3.2 Evaluation

On evaluation of the above algorithm, we noticed that speeding up the merge did not have a significant effect on the non-scaling set of applications. Further enhancements reduce the searching overhead(by flushing the slave queue on collectives) did not significantly improve performance either.

We then profiled the code by compiling with gprof and evaluated some of the non-scaling benchmarks. From the gprof profile, it was apparent that the bulk of the time was spent on communication (in the BGLML_Messager_CMadvance routine on BG/L) The merge time also seems to be correlated to the trace size. These results indicated that it is difficult to improve on the merge overhead without improving compression because a significant chunk of time is consumed in sending and receiving large event queues across

the BG/L internetwork.

4.4 Handling Recursion

Another challenge to call sequence identification is posed by recursion. We have devised a method to scan backtraces to identify repeated subsequences of identical return addresses. During composition of the backtrace structure, trailing repetitions are immediately folded into their first occurrence. This guarantees that recorded events at different recursion depth receive identical stack signatures in our framework. Hence, these events will compress perfectly, just as if the algorithm was coded up iteratively rather than recursively. This approach covers direct and indirect recursion. Note that if a compiler determines that tail recursion elimination is legal and performs this optimization, then the stack nesting depth remains unchanged. In essence, our recursion folding scheme for signatures complements compiler optimizations for the sake of trace compression whenever tail recursion elimination is not legal.

4.5 Simple Trace Visualization

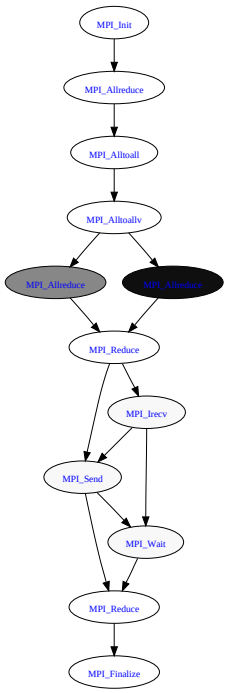


Figure 4.1: Sample trace visualization

We have built a simple trace visualization tool to interpret *ScalaTrace* traces. The aim, primarily, is to easily depict the level of compression and allow the developer to quickly focus on the "problem areas".

Each node in the trace graph in Figure 4.1 denotes an event. We draw an edge

from node u to node v if they share a participant and v occurs after u in the trace. The node color indicates the compression level: white nodes are fully compressed, black nodes are not compressed at all, grey nodes fall in between - the shade denotes the number of participating nodes.

The tool uses the trace parser to print out the node and dependence edge information in a format that can be easily visualized as a graph. This is accomplished by using a tool called *graphviz* [11].

Chapter 5

Experimental Framework

For all following experiments we use the MPI version of the NAS Parallel Benchmark (NPB) Suite [3] (version 3.2.1) with class C inputs as well as UMT2K benchmark from the ASCI Purple Benchmark Suite [4] from LLNL. The latter is an unstructured mesh transport code, which is a real-world test case known to exhibit load imbalance. It therefore both stresses our tracing approach and provides an interesting target to show the analysis techniques enabled by our approach. We use powers of two node counts for all codes, except for BT which requires a square number of processors. We do not report results for DT class C with 32 and 64 tasks due to input constraints.

We conducted our other experiments on a 2048-node BlueGene/L (BG/L) machine [12] with 1GB of memory per node. Correctness of replay was derived from showing that aggregate MPI statistics obtained via mpiP [13] from the application match that of the mpiP-instrumented replay.

Chapter 6

Experimental Results

We evaluate our methods to encode timing information with six experiments. First, we compare our timing techniques by looping over a message send and receive and varying the computation time per iteration, which evaluates the accuracy of using aggregate statistics versus histogram bin counts. Our next experiments study the benefits of using path-sensitive histograms, particularly with increasing MPI task counts. We compare replay times with and without path-sensitive histogram lists. We then compare replay accuracy when using a coarse-grained timer (`gettimeofday`) to using a fine-grained timing mechanism (`rts_gettimebase`). In the fourth and fifth experiments we vary the task count to assess both the accuracy and the effectiveness of compression in the presence of delta times. Our metrics are the accuracy of replay execution time, excluding the time to read the trace over the BG/L parallel file system, and trace file size. Finally, in the last experiment we analyze the runtime overhead of our tracing and compression mechanism.

6.1 Aggregate Statistics vs. Histograms

Figure 6.1 shows the replay accuracy of a send/receive microbenchmark with three different approaches for various run lengths (5 to 40 time steps). We generate three types of traces: aggregate statistics; 5 histogram bins; and 10 histogram bins. The first bar shows the uninstrumented application computation and MPI times, and the next three bars show replay times using our three trace types. We measure all times with mpiP [13]. The trace containing aggregate statistics deviates significantly from the original application time compared to the other two cases. We use 5 histogram bins to limit memory consumption in our remaining experiments since the replay results achieve reasonable accuracy.

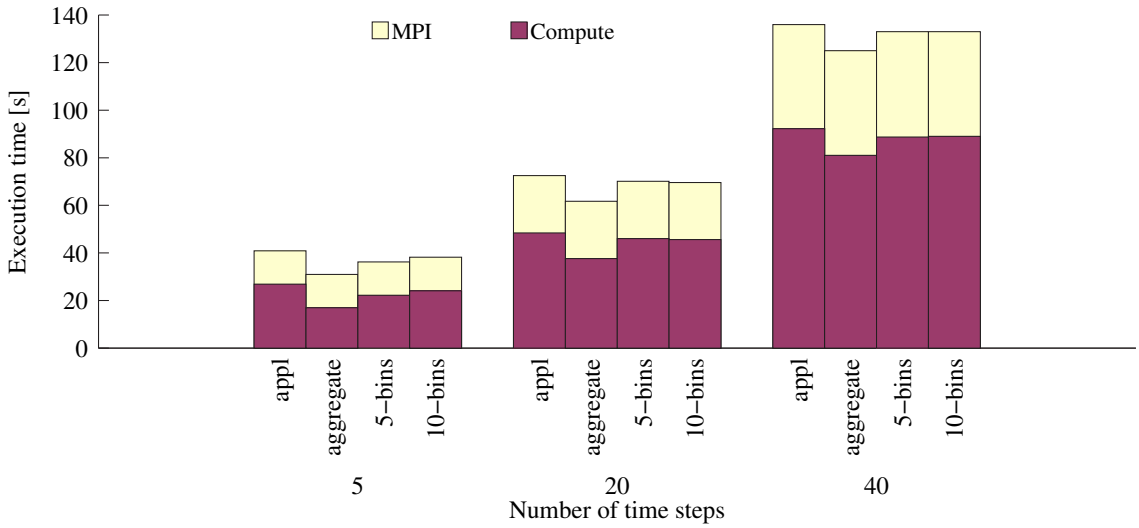


Figure 6.1: Using Histograms Increases Delta Accuracy

6.2 Contrasting Timing Techniques

Figure 6.2 shows that replay more accurately reflects application performance, particularly for larger number of nodes, for path-sensitive histograms vs. path-insensitive ones (one-path). Except for the uninstrumented case, computation (dark/maroon) and communication (light/yellow) times are distinguished in stacked bars. The bars depict aggregated execution times across all nodes for uninstrumented and mpiP-instrumented execution followed by replay with single-bin histograms and with single-bin path-sensitive histograms. We show the LU benchmark since it has a nested loop with widely varying compute times at different nesting levels, which emphasizes the impact more dramatically than other benchmarks. The loops contain send/receive pairs, which lead to substantial error in the MPI timings with inaccurate replay of the compute deltas when only a single bin is used.

6.3 Granularity of Timing

Figure 6.3 depicts results for the aggregated wall-clock times of LU using different timing sources. The different bars (in order) show uninstrumented execution, mpiP-enhanced application execution, replay with coarse-grained time (`gettimeofday`) and with fine-grained time (using the high-resolution BG/L timer) . The latter two utilize inter-node/globally compressed traces with additional mpiP instrumentation. The results illustrate that the *overhead of reading the time* from whatever source becomes more relevant as the number of nodes increases. We infer this from the significant change in computation time while the communication time is unaffected. This experiment demonstrates that the tolerance for timer overhead decreases as the task count increases. We discuss

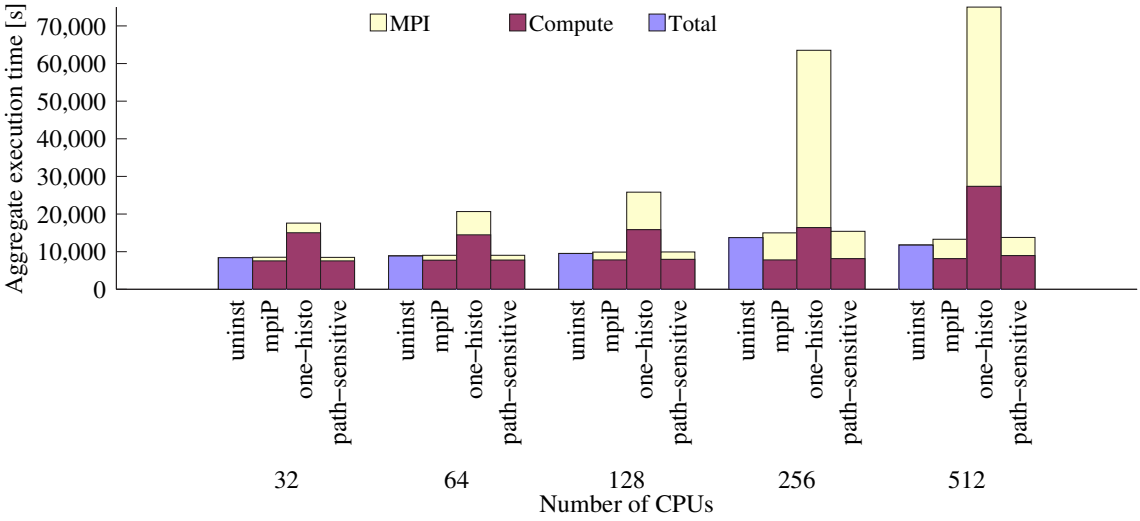


Figure 6.2: LU: Path-sensitive Histogram Benefits

the differences between uninstrumented, mpiP-instrumented and replay times for LU in the following.

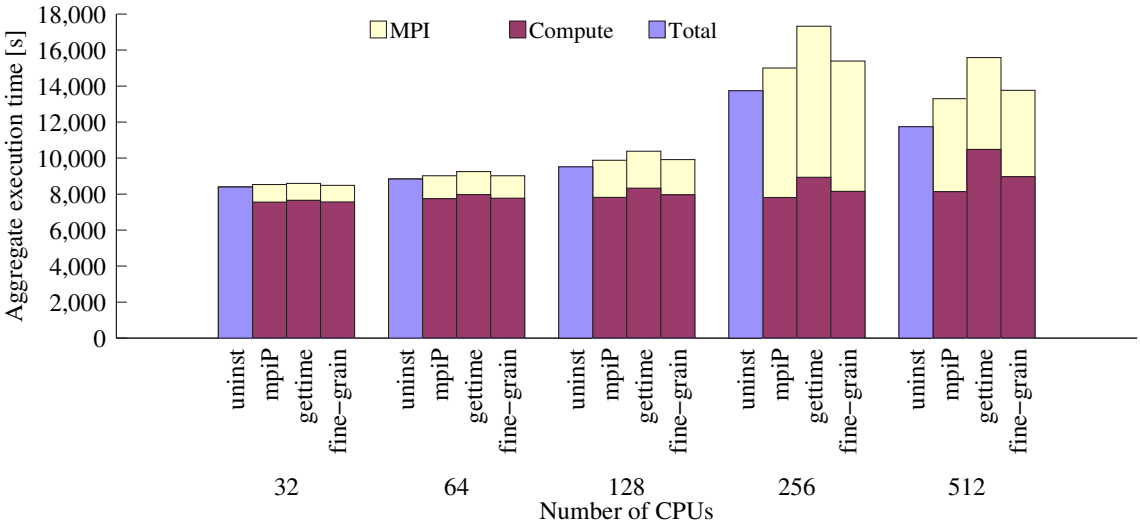


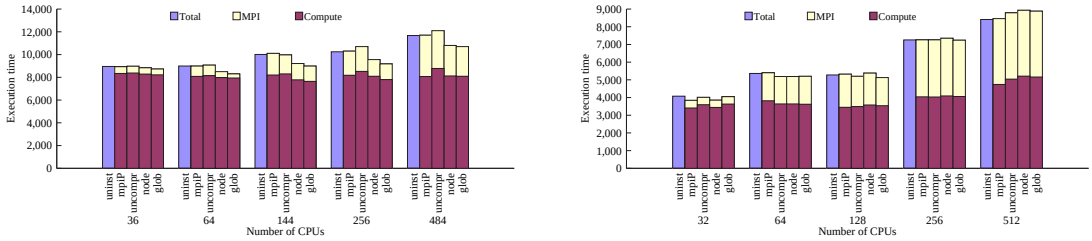
Figure 6.3: LU Times, Different Clocks

6.4 Timing Accuracy

Figures 6.4 and 6.5 depict the aggregated wall-clock times across all nodes for UMT2k and the NPB codes. For each set of results, five bars are shown:

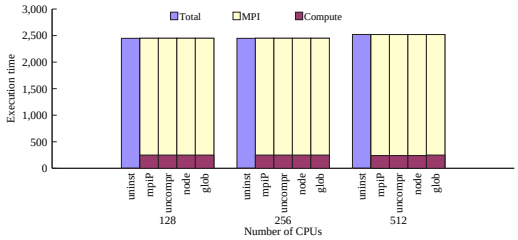
Bar 1: Original aggregated application execution time;

Bar 2: Aggregated execution time when linked with mpiP;

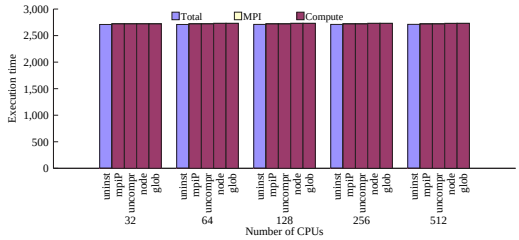


(a) BT

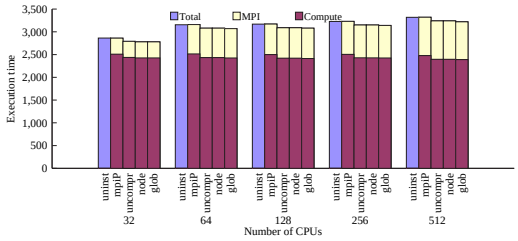
(b) CG



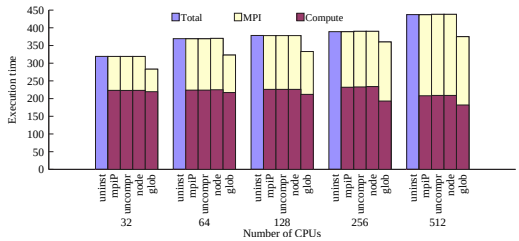
(c) DT



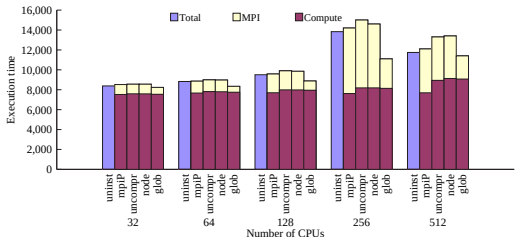
(d) EP



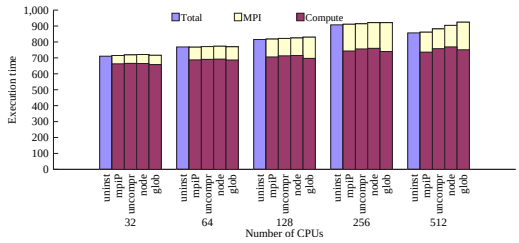
(e) FT



(f) IS



(g) LU



(h) MG

Figure 6.4: Aggregated NAS PB Wall-clock Times Across All Nodes.

Bar 3: Aggregated replay time with no compression (flat);

Bar 4: Intra-node compression aggregated replay time; and

Bar 5: Global compression aggregated replay time.

Global compression includes both intra and inter-node compression. We measure the times for bars 2 - 5 with mpiP and report the mpiP-measured communication and computation time. The data produced by mpiP also supports comparison of statistical data of call frequencies during replay with the mpiP run. These results validate that the replay engine correctly emulates the application's MPI usage. We varied the task count for each benchmark to assess the affect of strong scaling on the measured metrics.

The results in Figures 6.4 and 6.5 illustrate that aggregated wall-clock times are preserved well during replay of traces for most benchmarks. A comparison between uninstrumented and mpiP-instrumented reveals the overhead of "null" instrumentation. It reveals to what extent timing can be expected to be precise, *i.e.*, to what extent timing dilation is cause by the PMPI layer of instrumentation *vs.* the overhead of our trace compression. As the results indicate, the overhead stems from the former for most benchmarks.

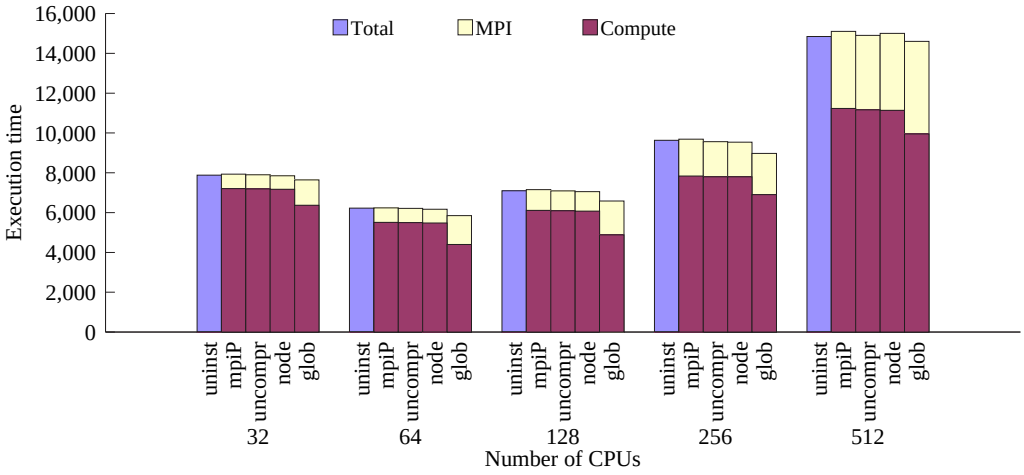
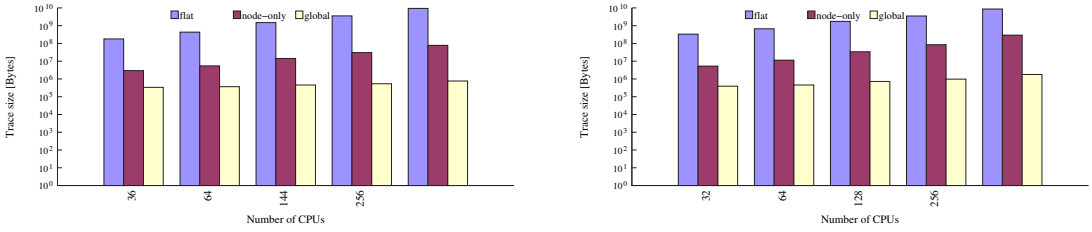


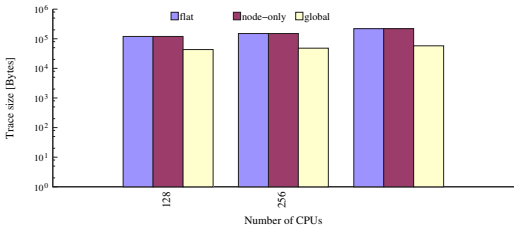
Figure 6.5: Aggregated UMT2k Wall-clock Times

Our replay results are generally very accurate. All benchmarks except for MG show nearly perfectly matching times regardless of the number of tasks. EP is a special case since it lacks any communication overhead so that replay is simply a sequence of sleeps, which is accurate, not surprisingly. IS, LU and BT show slightly lower global replay than mpiP times due to different communication overhead. In contrast, CG's global replay times can be slightly longer (512 nodes) or shorter (256 nodes), due to slightly larger computation and sometimes smaller communication overhead, apparently due to additional software layers and additional delays at collectives. Nonetheless, these replay times remain close to their mpiP counterparts. MG shows slightly increasing communication times with full compression for a larger number of tasks. The online replay mechanism may cause this behavior. Instead of decompressing traces, replay is realized by issuing sequences of

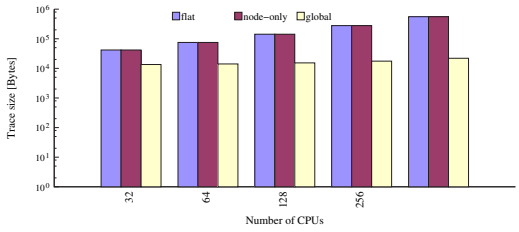


(a) BT

(b) CG

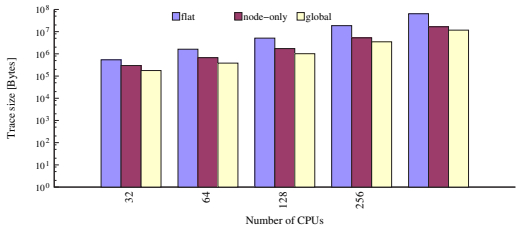


(c) DT

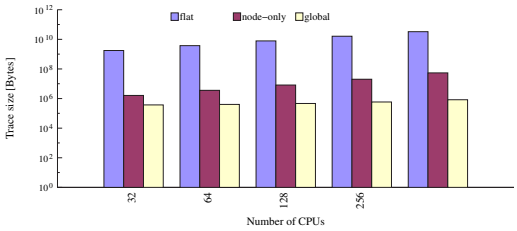


(d) EP

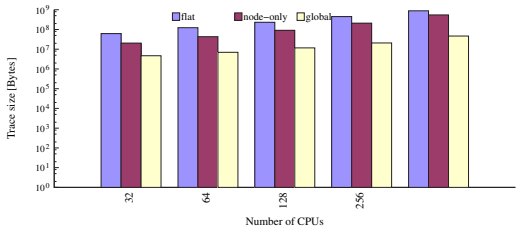
(e) FT



(f) IS



(g) LU



(h) MG

Figure 6.6: NPB Trace File Size per Node on BlueGene/L.

calls according to the PRSD trace records. Since PRSDs are nested by virtue of their structure, the replay is implemented as a recursive routine. The stack depth resulting from deep recursion may be a factor for timing in this case. Overall, our replay mechanism reproduces the original wall-clock times within -8% to 14% for intra-node and -20% to 7%

for full compression.

6.5 Trace Sizes

An important goal in adding delta times to ScalaTrace was to maintain its trace size properties [2]. In previous work, it was shown that applications fell into three categories: near-constant trace sizes regardless of number of tasks (DT, EP, and IS); sub-linear scaling with number of tasks (LU and MG); and those that do not scale (BT, CG, and FT). We have since made significant basic compression improvements with the exception of IS, as described below. Figures 6.6 and 6.7 show trace sizes after inclusion of time information and this trace compression optimization. The three sets are now (DT, EP, LU, BT, FT) for near-constant, (MG, CG) for sub-linear and (IS, UMT2k) for non-scalable sized traces.

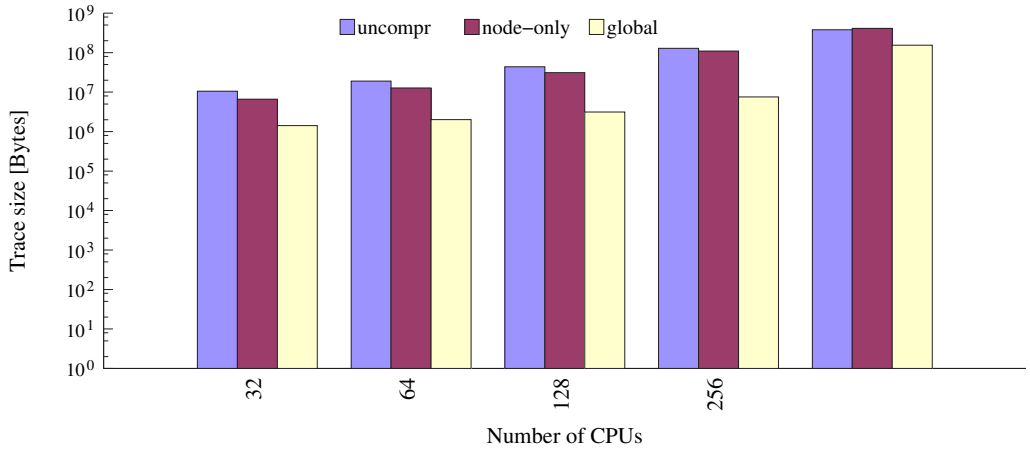
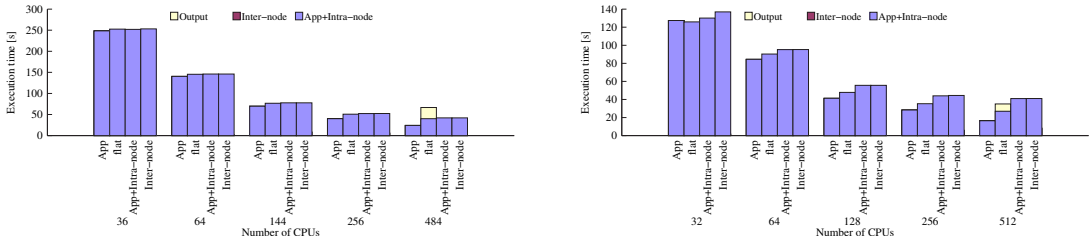


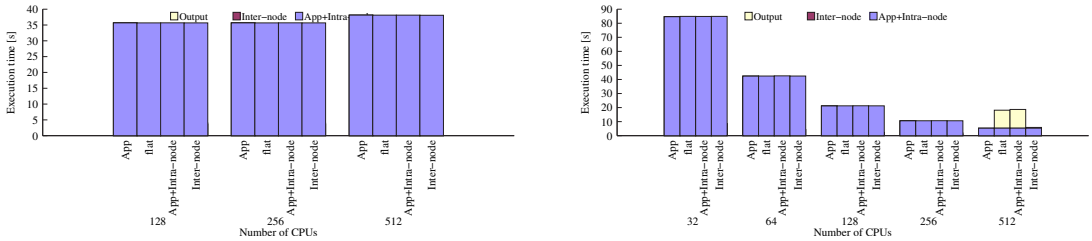
Figure 6.7: UMT2k Trace File Size per Node on BlueGene/L.

As mentioned before (in Chapter 4), we encode wildcard communication endpoints (ANY_SOURCE) directly instead of storing them as offsets. The LU benchmark profited significantly from this optimization. Another optimization that omits tags from point-to-point records significantly improved intra-node compression for BT. However, the LU benchmark uses tags to distinguish endpoints in which case we retain tags. The most significant improvements came from the inter-node compression optimization.

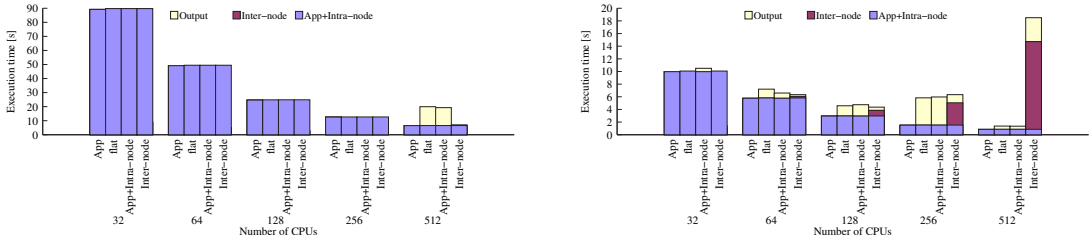
IS is non-scalable due to its dynamic rebalancing of work, which results in different sized payloads for an MPI_Alltoallv() collective upon each call (incorrectly encoded in constant size in a previous version of ScalaTrace). UMT2k falls into the non-scalable category in terms of trace sizes, as depicted in Figure 6.7. For these non-scalable cases, there is still room for improvement subject to ongoing investigation. But even for these cases, our compressed traces are three orders of magnitude smaller than currently used



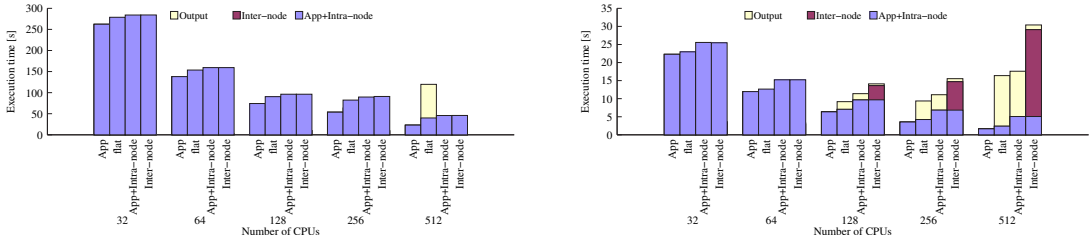
(a) BT (b) CG



(c) DT (d) EP



(e) FT (f) IS



(g) LU (h) MG

Figure 6.8: NAS PB wall-clock times at root node.

flat traces. Most significantly, delta times did not adversely affect trace sizes (even for the non-scalable traces), *i.e.*, non-compressible communication events were the same before and after time inclusion.

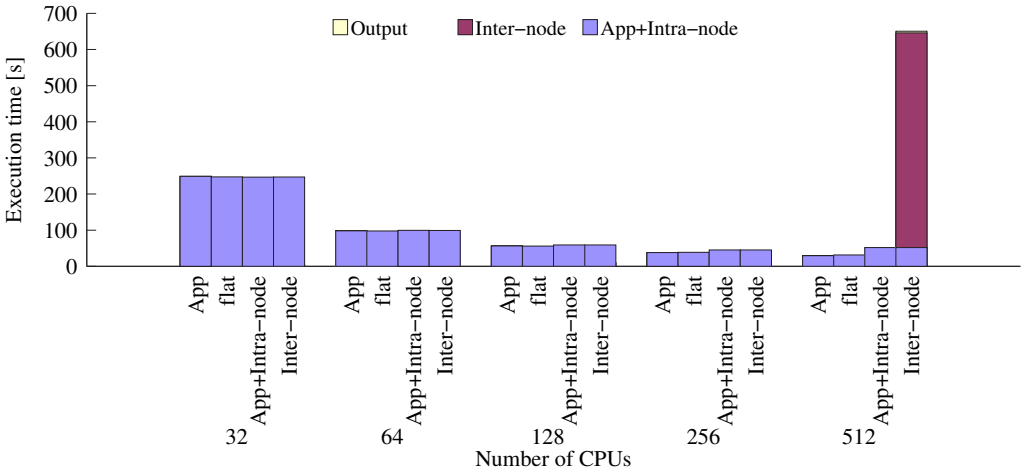


Figure 6.9: UMT2k wall-clock times at root node.

6.6 Tracing Overhead

Since task 0 performs the final stage of the inter-node compression and hence incurs the highest overhead, we measure its execution times during tracing runs to study our tracing overhead. Figures 6.8 and 6.9 show these runtimes for the uninstrumented baseline and the three trace options split into overall application runtime, which includes both tracing and intra-node compression overhead, I/O time, and inter-node compression within *MPI_Finalize*.

Our trace compression mechanism incurs negligible overhead for applications with small (DT and EP) or modest (FT and IS) numbers of communication events and hence trace sizes. For applications with larger traces, we observe an increase in runtime similar to flat tracing for applications with good intra-node compression (BT and LU) and only slightly higher with less successful compression (CG, MG, and UMT2k). Similarly, inter-node compression has little overhead when compression works well, while applications with less scalable traces (IS, MG, and UMT2k) show significantly larger overhead. The high overhead stems in both cases from searching trace records for potential matches, which are more prevalent when compression is poor, leading to higher search overhead. We have already identified a significant improvement for inter-node compression, although its overhead, which is particularly high for a few cases, occurs in *MPI_Finalize*. Hence, it does not perturb the actual application timings.

6.7 Detecting Communication Inefficiencies

An important application of encoding time deltas in compressed traces is to detect imbalances in communication. Traditional tracing tools generate flat traces per node. Hence, it would be necessary to look at hundreds or thousands of trace, each of

which could be a few gigabytes or longer, to locate communication anomalies.

Our trace representation simplifies this task considerably: inspection of only one small compressed trace is required. Within a trace, the histogram for each record contains load distribution information, such as extreme node information and the number of nodes within a particular range. Simply inspecting the trace suffices, manually or with minimal batch tool and/or visualization. Such a tool could be derived from our replay engine's trace parser.

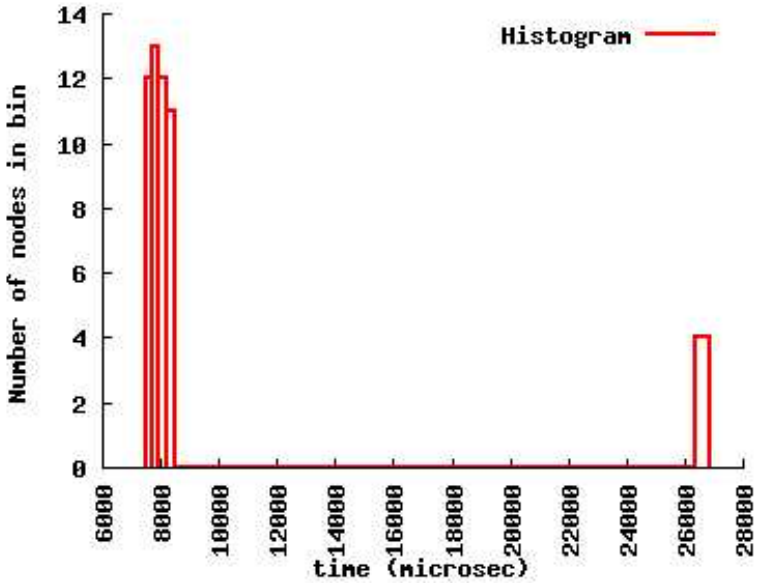


Figure 6.10: Delta histogram at one MPI_Barrier in UMT2k

Figure 6.10 shows an example from the UMT2k benchmark. As mentioned earlier, this benchmark is known to have load imbalance. We took a trace from a run of the benchmark and inspected the delta histograms of collectives. The figure clearly depicts the imbalance as we see the nodes falling into two distinct groups with extreme communication times, a sign of imbalance in computation immediately preceding the collective. The trace also indicates the nodes responsible for minimum and maximum times.

Chapter 7

Related Work

The mpiP tool, a lightweight profiling library for MPI applications, collects statistical information about MPI functions [13]. It reports aggregate metrics including average execution times. Our work distinguishes path-sensitive execution contexts instead of aggregate profiles. Our timing information is also of finer granularity in histograms that indicate nodes subject to certain timing ranges. This added information supports the detection of communication bottlenecks and their location, in contrast to mpiP.

Geimer et al. [14] obtain per-node traces stored locally at each node and later replay these communication traces on the same architecture with the same number of nodes to detect communication bottlenecks. Their performance results indicate replay times for their new analysis approach still diverge from the original application time by up to an order of magnitude, even when ignoring the additional overhead for file I/O with flat trace files. Later work generalized this approach to Grid environments using distributed time stamp synchronization [15]. In contrast, our experiments show closely matching execution times under replay, not only for compressed but also flat local traces. Their trace sizes are reported to reach 10GB, the same order of magnitude reported by others [16]. In fact, any flat trace representation, including commercial tool sets such as Vampir [17], is subject to extremely large trace files that are generally stored locally and increase linearly in size with both the number of MPI calls made and the number of tasks. This limits their applicability as scalability is compromised. In contrast, the technique [2] on which we build compresses traces to sizes that are three orders of magnitude smaller and do not significantly increase in size, if it all, during strong scaling. Our work shows that scalability need not be sacrificed even when timing information is included.

Paraver and Dimemas [5], an MPI tracing tool set, combine tracing functionality similar to Vampir (under the same limitations) with a discrete-event-based network performance simulator using traces. Dimemas replays traces in a simulation environment to

study architectural parameters. Tools like Dimemas and Vampir's trace visualizer could be used in conjunction with our replay time-preserving traces without having to decompress our trace representation, which could result in significant speedups for users. Casas et al. [18] recognize multi-level regularities in large, post-mortem trace files. By detecting patterns, they compress these flat trace files offline and can filter background (operating system) activity artifacts. The compression is reported to take up to an hour for benchmarks comparable to NAS. Our method, in contrast, compresses regularities on-the-fly and never generates any flat trace file.

The Open Trace Format (OTF) targets scalable tracing, yet without any advanced (domain-specific) compression scheme [19]. In contrast to our work, it uses regular zlib compression on blocks of data, which loses structure and limits analysis on the compressed format. It also does not support inter-node compression schemes. Hence, the complexity of aggregate trace size over n tasks is O(n). However, they can produce multiple streams and, hence, store and load them in parallel with user-defined granularity. An alternate trace format by the same group uses so-called cCCGs, a structural compression format that combines regular patterns into common sub-trees [20]. In practice, the observed storage requirement for regular event patterns is reported to be logarithmic due to combining nodes upon matching patterns and deltas. In contrast, our storage overhead is as low as constant when event patterns are regular *regardless* of the delta time properties as the number of histograms is bounded by a constant.

Arnold et al. [21] developed a scalable tool to identify task behavior equivalence classes with high similarity based on stack signatures. Their approach utilizes MRNet, a software overlay network that provides efficient multicast and reduction communications [22]. MRNet uses a tree of processes between the tool's front-end and back-ends to improve group communication. MRNet introduces additional complexity, which we decided to avoid in our current prototype. MRNet would support on-the-fly and asynchronous trace compression across tasks. By using MRNet, we would further reduce the memory pressure of our trace generator. MRNet could be used in a future version of our tool using $P^N MPI$ as the glue layer between the tools [23].

Freitag et al. [24] describe a window-based compression scheme and evaluate its applicability to OpenMP traces. Our PRSD compression is more powerful as it allows recursive compression online. Neyman et al. [25] designed a tool to detect races in PVM codes using a trace generation and replay tool while recent work by Mesnier et al. focuses on I/O trace generation and replay [26]. Neither of these scale as they do not perform any compression. Our approach could also handle MPI-IO calls similarly to regular MPI events.

Chapter 8

Conclusion

Storing communication traces with precise time information in a scalable manner is a significant challenge. Existing work either omits timing information for the sake of scalability, and thereby limits the usefulness of the data, or produces trace files with a total size linear to the number of processors and the overall runtime of the application, which is clearly infeasible for large scale environments.

In this work we presented three techniques to add timing information to a scalable trace format based on PRSDs. In all cases we rely on delta times rather than absolute time stamps to expose similarities in repeating call patterns. Our methods vary the level of timing detail encoded in the trace from aggregate statistics to path-specific histograms. We show that these techniques, particularly the latter, are sufficient to capture timing characteristics of the target application without significantly increasing the required storage space for the resulting traces. Our solution thus represents the first truly scalable tracing mechanism for MPI applications capable of capturing timing information along with the actual communication structure.

Our work towards optimizing compression methods have resulted in near-constant or sub-linear size traces for many more applications now.

Thus, we have achieved the goals that we had set and have proven our hypothesis stated initially. We expect that smaller, scalable traces with accurate timing information should eventually enable developers to extract more performance from their MPI applications.

Bibliography

[1] D. Culler, R. Karp, D. Patterson, A. Sahay, E. Santos, K. Schauser, R. Subramonian, and T. von Eicken. LogP: A practical model of parallel computation. *Communications of the ACM*, 39(11):78–85, November 1996.

[2] 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*, April 2007.

[3] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, D. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The NAS Parallel Benchmarks. *The International Journal of Supercomputer Applications*, 5(3):63–73, Fall 1991.

[4] The ASCI purple benchmarks. http://www.llnl.gov/asci/purple/benchmarks, 2002.

[5] V. Pillet, J. Labarta, T. Cortes, and S. Girona. PARAVER: A tool to visualise and analyze parallel code. In *Proceedings of WoTUG-18: Transputer and occam Developments*, volume 44 of *Transputer and Occam Engineering*, pages 17–31, April 1995.

[6] W. E. Nagel, A. Arnold, M. Weber, H. C. Hoppe, and K. Solchenbach. VAMPIR: Visualization and analysis of MPI resources. *Supercomputer*, 12(1):69–80, 1996.

[7] O. Zaki, E. Lusk, W. Gropp, and D. Swider. Toward scalable performance visualization with Jumpshot. *International Journal of High Performance Computing Applications*, 13(3):277–288, 1999.

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

[9] 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, March 2003.

[10] D. E. Knuth. *The Art of Computer Programming: Fundamental Algorithms*, volume 2. Addison-Wesley, 2 edition, 1973.

[11] Graphviz. http://graphviz.org/.

[12] N. Adiga and et al. An overview of the BlueGene/L supercomputer. In *Supercomputing*, November 2002.

[13] J. Vetter and M. McCracken. Statistical scalability analysis of communication operations in distributed applications. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2001.

[14] M. Geimer, F. Wolf, B. Wylie, and B. Mohr. Scalable parallel trace-based performance analysis. In *European PVM/MPI Users' Group Meeting*, 2007.

[15] Daniel Becker, Felix Wolf, Wolfgang Frings, Markus Geimer, Brian J.N. Wylie, and Bernd Mohr. Automatic trace-based performance analysis of metacomputing applications. In *International Parallel and Distributed Processing Symposium*, 2007.

[16] JaeWoong Chung, Chi Cao Minh, Austen McDonald, Travis Skare, Hassan Chafi, Brian D. Carlstrom, Christos Kozyrakis, and Kunle Olukotun. Tradeoffs in transactional memory virtualization. In *Architectural Support for Programming Languages and Operating Systems*, 2006.

[17] Holger Brunst, Hans-Christian Hoppe, Wolfgang E. Nagel, and Manuela Winkler. Performance optimization for large scale computing: The scalable VAMPIR approach. In *International Conference on Computational Science (2)*, pages 751–760, 2001.

[18] Marc Casas, Rosa Badia, and Jesus Labarta. Automatic structure extraction from mpi applications tracefiles. In *Euro-Par Conference*, August 2007.

[19] A. Knüpfer, R. Brendel, H. Brunst, H. Mix, and W. E. Nagel. Introducing the open trace format (OTF). In *International Conference on Computational Science*, pages 526–533, May 2006.

[20] Andreas Knupfer. Construction and compression of complete call graphs for post-mortem program trace analysis. In *International Conference on Parallel Processing*, pages 165–172, 2005.

[21] Dorian C. Arnold, Dong H. Ahn, Bronis R. de Supinski, Gregory L. Lee, Barton P. Miller, and Martin Schulz. Stack trace analysis for large scale debugging. In *International Parallel and Distributed Processing Symposium*, 2007.

[22] Philip C. Roth, Dorian C. Arnold, and Barton P. Miller. MRNet: A software-based multicast/reduction network for scalable tools. In *Supercomputing*, pages 21–36, Washington, DC, USA, 2003. IEEE Computer Society.

[23] Martin Schulz and Bronis R. de Supinski. $P^N MPI$ tools: A whole lot greater than the sum of their parts. In *Supercomputing*, 2007.

[24] F. Freitag, J. Caubet, and J. Labarta. On the scalability of tracing mechanisms. In *Euro-Par Conference*, pages 97–104, August 2002.

[25] Marcin Neyman, Michal Bukowski, and Piotr Kuzora. Efficient replay of PVM programs. In *European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 83–90, 1999.

[26] M. Mesnier, M. Wachs, R. Sambasivan, J. Lopez, J. Hendricks, and G. R. Ganger. //trace: Parallel trace replay with approximate causal events. In *USENIX Conference on File and Storage Technologies*, February 2007.