Probabilistic Communication and I/O Tracing with Deterministic Replay at Scale

Xing Wu[*], Karthik Vijayakumar[*], Frank Mueller[*], Xiaosong Ma[*†] and Philip C. Roth[†]
[*]*Department of Computer Science, North Carolina State University, Raleigh, NC 27695-7534*
[†]*Computer Science and Mathematics Division, Oak Ridge National Laboratory, Oak Ridge, TN 37831*

***Abstract*—With today's petascale supercomputers, applications often exhibit low efficiency, such as poor communication and I/O performance, that can be diagnosed by analysis tools. However, these tools either produce extremely large trace files that complicate performance analysis, or sacrifice accuracy to collect high-level statistical information using crude averaging.**

This work contributes Scala-H-Trace, which features more aggressive trace compression than any previous approach, particularly for applications that do not show strict regularity in SPMD behavior. Scala-H-Trace uses histograms expressing the probabilistic distribution of arbitrary communication and I/O parameters to capture variations. Yet, where other tools fail to scale, Scala-H-Trace guarantees trace files of near constant size, even for variable communication and I/O patterns, producing trace files orders of magnitudes smaller than using prior approaches. We demonstrate the ability to collect traces of applications running on thousands of processors with the potential to scale well beyond this level. We further present the first approach to deterministically replay such probabilistic traces (a) without deadlocks and (b) in a manner closely resembling the original applications.

Our results show either near constant sized traces or only sub-linear increases in trace file sizes irrespective of the number of nodes utilized. Even with the aggressively compressed histogram-based traces, our replay times are within 12% to 15% of the runtime of original codes. Such concise traces resembling the behavior of production-style codes closely and our approach of deterministic replay of probabilistic traces are without precedence.

I. Introduction

As supercomputers progress in scale and capability toward exascale levels, characterization of communication and I/O behavior is becoming increasingly difficult due to system size and complexity. The large numbers of processors/cores, increased aggregate memory capacity, complex interconnects, and increasingly larger gap between computation power and I/O performance create great challenges on effective and scalable ways for performance study of applications for efficient use of system resources. Challenges exist on the software side as well. Today, the complexity of extreme-scale scientific applications increases rapidly. Applications often integrate multiple software components and may exercise vastly different computation/communication models. Such codes are becoming more dynamic and diverging from strict, regular single program, multiple data (SPMD) behavior. Examples include multi-physics or coupled codes, where partitions of nodes implement different simulation models, work on separate datasets, or even conduct analytics tasks such as data reduction. Such applications exhibit multiple program, multiple data (MPMD) behavior as multiple nodes work on multiple sections of the program. For example, in climate simulations, some nodes simulate climate changes over land, while other nodes work on sea models. Hence, different modules, like land and sea, use different input data and algorithms resulting in different communication and I/O behavior within each module.

Many studies have investigated the communication and I/O characteristics of applications. They are facilitated with three main classes of tools: tracing tools, capable of capturing and recording all message events at the cost of high storage requirements; profiling tools that provide performance summaries trading off detailed level for low storage and runtime overhead; and communication and I/O kernels that eliminate computation and retain only communication and I/O behavior. Although application kernels are designed to capture the exact application behavior, it is difficult to keep these kernels up-to-date since the applications constantly evolve over time. Application traces, in contrast, can be readily generated by re-execution of instrumented application, to keep up with a changing code base. This makes application traces a preferred vehicle for performance analysis of parallel applications in practice.

The combination of job scale and application complexity, however, creates unique challenges for parallel tracing tools. On one end of the spectrum, traditional tracing tools (such as Vampir [1]) record all events sequentially for each parallel process. For large application runs on leadership-class supercomputers, this approach generates unmanageable trace file sizes, introducing prohibitive overheads, *e.g.*, for copying trace files from temporary to permanent storage, hitting the maximum storage limit, and even the need for a cluster plus another parallelized tool to perform trace analysis [2]. On the other end of the spectrum, tools that only report statistical information (such as mpiP [3]) may fail to deliver the level of detail needed in performance analysis or debugging.

On-the-fly trace compression [4], [5] provides lossless

This work was supported in part by NSF grants 0237570 (CAREER), 0546301 (CAREER), 0621470, 0937908, 0429653, 0937690, 0958311 and DOE DE-FG02-08ER25837, by the Office of Advanced Scientific Computing Research (U.S. Department of Energy), a joint faculty appointment between Oak Ridge National Laboratory (ORNL) and NC State University, and a senior visiting scholoarship at Tsinghua University. It used resources of the National Center for Computational Sciences at ORNL and was performed in part at ORNL, managed by UT-Battelle, LLC under Contract No. DE-AC05-00OR22725.

tracing *and* dramatically reduced trace file sizes, and it has recently been extended to conduct multi-level I/O tracing [6] in addition to capturing communication calls. However, effective compression builds on the homogeneous behavior across processes (inter-node compression) and repetitive behavior within a process (intra-node compression). With complex, irregular, or self-adjusting applications, such assumptions do not hold and compression fails due to mismatches between traced events.

In this work, we propose Scala-H-Trace, a novel approach to collect concise traces for applications exhibiting *non-SPMD* behavior by using (1) a *histogram-based* statistical application parameter compression, and (2) a Weighted Longest Common Subsequence (WLCS) based inter-node event matching algorithm. In other words, while past approaches proved effective for the easier problems of tracing SPMD codes, this work focuses on the much harder problems of tracing non-SPMD codes. Scala-H-Trace is motivated by the tradeoff between exact details and manageability of trace file size. Although having exact details helps in root cause analysis, lossless tracing becomes increasingly unaffordable on ultra-scale machines.

Scala-H-Trace follows a lossy philosophy where most of the advantages of the lossless approaches over profiling are preserved. Like the lossless approaches, Scala-H-Trace is able to trace a parallel application at the granularity of every single event. It preserves the temporal ordering of events as well as the timing information so that the "big picture" of an application's communication and I/O behavior is captured. Unlike traditional approaches, Scala-H-Trace utilizes statistical methods to record application parameters such as loop iteration count, message volume, and even the source/destination values of point-to-point communication events. A unique feature of Scala-H-Trace is that it enables the user to set a *merge precision level* during trace collection. This user-defined value drives the compression efficiency as well as the trace precision. If the trace precision falls below the specified threshold, exact event recording is used so that the trace fidelity is guaranteed. The size of such a trace file then becomes a function of the desired merge precision level, which can be tuned to obtain a manageable size while retaining trace artifacts suitable for performance analysis. Our histogram-based approach also reduces the tracing overhead as the time taken to compress smaller histogram-based traces is considerably less than the traditional lossless traces.

Histogram-based tracing creates new challenges for accurate replay of the traced events. To ensure the correctness of the captured trace and to reproduce the communication and I/O behavior, we designed a novel replay facility. The new replay tool replays the lossless traces [5] or well as lossy ones. For the latter, our tool employs a distributed, orchestrated and deterministic replay scheme. Our goal in the replay of histogram-based traces is not to capture exact original events but rather the existence of a sequence of events with comparable timings and communication endpoints. Resulting information can be useful in identifying bottlenecks and also the communication pattern of a particular application.

We evaluated our approach with the Parallel Ocean Program (POP) and two benchmarks from the NAS parallel benchmark suite. POP is both computational and I/O intensive and thus a representative application to evaluate our tool. Our results provide one to two orders of magnitude smaller trace files than the previous approach. We also evaluated our replay tool by replaying the histogram-based traces. The replay time only deviated 12% to 15% from the original application's time in most cases, even for most aggressively merged histogram-based traces.

In a nutshell, we made the following contributions:

- We proposed a WLCS-based inter-node event matching algorithm improving the inter-node trace compression for programs that exhibit non-SPMD behavior.
- We designed a histogram-based statistical tracing approach that features more aggressive trace compression based on a user-defined precision level that drives both compression efficiency and trace accuracy.
- We provided a distributed approach to replay statistical traces that does not require back-channel communication to preserve causal event ordering for correctness.

II. Background

Scala[IO]Trace [4], [5], [6] is a publicly available parallel application tracing library. It uses the MPI Profiling layer (PMPI) [7] to intercept MPI calls and to collect communication and, optionally, parallel I/O traces. It implements a set of sophisticated trace compression algorithms so that only a single, concise, and lossless trace file is generated for any large-scale parallel application. In this section, we briefly introduce the techniques used in Scala[IO]Trace to allow a later comparison with Scala-H-Trace.

A. Trace Compression

Scala[IO]Trace performs two types of compression: *intra-node* and *inter-node*. The former exploits the repetitive nature of timestep simulation in parallel scientific applications. The latter exploits the homogeneity in behavior among different processes of the SPMD codes.

At the intra-node level, loop compression is performed on-the-fly. Repetitive events in different iterations of loops are collected as Regular Section Descriptors (RSDs) [8]. Power-RSDs (PRSDs) are used to represent RSD events in nested loops [9]. Consider the following code snippet:

```
for( i = 0; i < 10; i++ ) {
    for( j = 0; j < 100; j++) {
        MPI_Irecv(LEFT, ...);
        MPI_Isend(RIGHT, ...);
        MPI_Waitall(...);
    }
    MPI_Allreduce(...);
}
```

Trace compression results in the following tuples: RSD1:{100, MPI_Irecv, MPI_Isend, MPI_Waitall} representing 100 iterations of MPI_Irecv, MPI_Isend and MPI_Waitall in the inner loop, PRSD1: {10, RSD1, MPI_Allreduce} denoting 10 iterations of RSD1 followed by MPI_Allreduce in the outermost loop. The algorithm uses the calling context of events to match repetitious behavior. This ensures that identical MPI functions originating from different call paths are not compressed together.

In typical parallel applications, parallel processes follow the same communication pattern but have different end-points as a result of communication with neighboring nodes. Scala[IO]Trace captures the similarities in communication patterns by utilizing a unique location-independent encoding technique to represent communication end-points for inter-node compression. The inter-node compression is performed along a radix tree among all nodes. The final compressed trace is generated at the root node. *Ranklist* is used to represent the task rank information of the participating nodes of MPI events that are merged across multiple nodes. A topology-aware encoding technique is designed to keep the ranklist representation concise and scalable.

B. Time Preservation

Another important feature of Scala[IO]Trace is the time preservation of captured traces. Instead of recording absolute timestamps, the tool records delta time of computation durations between adjacent communication calls. During RSD formation and inter-node compression, delta times are compressed with histograms to concisely represent the distribution of the recorded timing values. More details on collecting statistical timing information are provided elsewhere [5].

C. Timed Replay

ScalaReplay is a replay engine for Scala[IO]Trace traces. It is a parallel program that runs at the node size of the input trace and issues MPI and I/O calls according to the exact parameter values recorded in the trace, yet without the actual message payloads/file content. To capture the potential impact of computation on communication performance [10], ScalaReplay simulates the computational phases with timed delays between trace events based on recorded delta times.

III. Histogram-Based Trace Collection

Noeth *et al.* [4] provide trace compression techniques resulting in an almost constant sized trace file or sublinear increases in trace file size with strong scaling (increasing number of nodes). Yet, these results only hold for SPMD-style benchmarks. For production size scientific applications with non-SPMD patterns, such as the Parallel Ocean Program (POP) [11], the inter-node compression technique may fail to obtain a near-constant sized trace file with increasing number of nodes.

POP performs ocean simulation for multiple time steps. Each time step performs a set of computations and communications of an inner loop in multiple iterations. Due to different data-dependent convergence points in the computation across different timesteps, the number of inner loop iterations varies from timestep to timestep. Even though all MPI events originate from the same calling sequence (call stack), varying loop iteration counts in each timestep inhibit intra-node compression and thus negatively impact inter-node compression across all nodes. This behavior can also be observed in many Adaptive Mesh Refinement (AMR) applications in which the input set is dynamically rebalanced on a periodic basis.

To address these problems, we propose a novel method of tracing. We promote histogram-based trace information for a predefined user-tunable merge precision level to obtain higher compression rates of trace events — at the expense of accuracy. Consider the following 3 scenarios: (1) If the user sets the merge precision level to 100%, then only events with perfectly matching function parameters are merged. (2) If the user sets the merge precision level to 95%, then events with non-matching function parameters will be merged if and only if all pairs of parameters differ by no more than 5%. Should any pair of parameters exceed the 5% threshold, we fall back to lossless tracing. (3) If the user sets the merge precision level to 0%, then events with non-matching function parameters are also merged and the non-matching parameters are collected in histogram bins. Note that a merge precision level of 0% does not mean that the entire meaningful information is lost. Instead, the statistical function parameters collected in histogram bins still capture the overall behavior of the application. Depending on user needs, the smallest traces with high application resemblance collected using a 0% merge precision level may be much more useful than unmanageably large trace files. In this section, we explain what trace information is collected as histogram and discuss possible tradeoffs in collecting statistical information versus non-lossy information.

A. Histogram Construction

Our approach uses histograms to collect probabilistic information on varying application parameters at multiple levels in the trace. Histogram-based collection employs a technique to collect statistical information in dynamically balanced bins. The online balancing algorithm equalizes the number of items per bin while adjusting their value range constraints. We provide an option to set an interval after which bins are adjusted. Two bins with the lowest frequencies are combined and the bin with maximum frequency is split into two bins. We further store auxiliary information for each bin, such as minimum/maximum/average/variance/frequency, and maximum/minimum values over the entire value range (all bins) and the node ranks associated with those. This provides statistical distribution properties and outlier information, which can be used during replay or by performance analysis tools to enable root cause detection.

We have designed our system in a way to collect exact trace information as much as possible. Our compression algorithm attempts to match events originating from the same call stack. It compresses events only if all function parameters match. Histogram collection is triggered only if there is a mismatch in function parameters or in the loop information. In such cases, the difference between two values is checked against the user specified merge precision level. If the difference is within the target precision range, events are merged and the non-matching parameters are recorded in a histogram from there on. If the difference falls out of the target precision range, exact event recording takes place and failures in compression may happen.

B. *Iteration Count Histogram*

The loop iteration count denoted by PRSDs can be collected as a histogram. This enables better compression of repeating events in many scientific applications that otherwise would fail to compress due to data dependencies. Although the exact iteration count is lost in the final trace, the number of loop iterations directly depends on the computation, which, in turn, varies with different input sets. Hence, collecting statistical loop iteration counts only has a minor impact in capturing the communication behavior of the application. The main advantage of this approach is the ability to obtain a concise trace file by allowing a small percentage of lossy trace collection that otherwise would have resulted in a trace file of unmanageable size.

```
for( i = 0; i < 50; i++ )
    while( !converged() ) {
        do_calculation();
        MPI_Irecv(...);
        MPI_Send(...);
        MPI_Wait(...);
    }
```

Figure 1. Loop with Convergence Check

Consider the code snippet shown in Figure 1. If the iteration count matches across time-steps, *i.e.*, the inner *while* loop takes the same number of iterations to converge in each time-step, the resulting PRSD will be of the form PRSD1:{50, RSD1}, where RSD1 represents the inner *while* loop with MPI_Irecv, MPI_Send, and MPI_Wait as the loop body. However, due to mismatching convergence points across different time-steps, the strict match on iteration count required by lossless compression will lead to traces such as:

```
RSD1: <39, MPI_Irecv, MPI_Send, MPI_Wait>
RSD2: <40, MPI_Irecv, MPI_Send, MPI_Wait>
...
RSD49: <38, MPI_Irecv, MPI_Send, MPI_Wait>
RSD50: <42, MPI_Irecv, MPI_Send, MPI_Wait>,
```

Here, the expected PRSD is not formed due to mismatching RSDs across time steps. As a result, the per-node trace size is non-scalable with respect to the number of time-steps. The same problem leads to cascading compression failures across nodes as well. With inter-node event compression, compressed traces from different nodes are merged together. In applications with non-SPMD behavior, loops created during intra-node compression can have matching events across nodes, but fail to compress across nodes due to a mismatch in the loop iteration count. This prevents the entire loop from being merged, increasing the trace file size linearly with the number of nodes. For the same example in Figure 1, assuming different nodes take different number of iterations to converge, the final trace is stylized as follows:

```
Node1: PRSD1:{50, RSD1:<41,Irecv,Send,Wait>}
Node2: PRSD1:{50, RSD1:<39,Irecv,Send,Wait>}
Node3: PRSD1:{50, RSD1:<68,Irecv,Send,Wait>}
...
```

Since the per-node traces are not compressed but concatenated sequentially in the final trace — due to the mismatching iteration counts — the trace size is not scalable with respect to the total number of nodes.

Histogram-based trace collection ensures that events are always merged both within and across nodes, despite varying iteration counts. Hence, the resulting trace will have just one PRSD for the entire time-step calculation representing all the nodes. In addition, outliers that converge much slower, such as Node3, will be captured in a particular bin so that root cause analysis becomes possible.

C. *Function Parameter Histogram*

Apart from collecting loop iteration counts, MPI function parameters, such as Send/Recv volume, tag and source/destination ranks, can also be recorded in histograms. The Send/Recv data volume is important to capture the network load. Source/destination ranks of the point-to-point communication operations define the communication pattern and are thus critical for performance optimization and task mapping analysis. However, in applications with excessive data dependencies and non-SPMD behavior, the Send/Recv volumes and end-point patterns often vary across different time-steps and across nodes. To reduce compression failures caused by small deviations in communication parameters, the traditional (lossless) approach records timestep-inconsistent parameter values in a vector, which is further associated with a ranklist for inter-node compression. As an example, the trace snippet below shows the vector representation of the *COUNT* parameter of an MPI_Send called by 10 nodes:

```
COUNT: (90B, 92B, 87B)[ranks: 2, 7, 8]
COUNT: (89B, 93B, 90B)[ranks: 1, 5, 6, 9]
COUNT: (43B, 41B, 38B)[ranks: 0, 3, 4]
```

Although this compression results in a more concise representation than its uncompressed equivalent, it still suffers from increases in the trace size proportional to the number of timesteps and nodes, especially when no regularity for ranklists could be deduced.

Using histograms to collect data volume allows better compression of repeating events originating from the same call stack. For this example, let histograms have bins of values around 90 bytes and 40 bytes along with their frequencies. In addition to binning data volume, we also collect participating ranks in a bitmap and encode it in the trace file. This provides information on exact values that is missing from the histograms and aids post-mortem analysis tools. In the above example, an analysis tool may choose data volume of either around 90 bytes or 40 bytes according to the frequency and bitmap information, while volumes other than these are excluded from the pseudo-random selection.

IV. Deterministic Replay

While histogram-based trace collection is powerful in compressing irregular or dynamically changing events, the collected traces themselves create challenges for replaying and subsequent performance analysis. Since Scala-H-Trace collects statistical values for communication volume, tags, and end-points, the conventional ScalaTrace replay design for lossless traces, which takes an independent, uncoordinated approach among nodes, can lead to potential deadlocks due to statistical uncertainty, or may fail to re-create the original communication or I/O pattern with reasonable proximity. Hence, the core challenge of histogram-based replay is to ensure that events are issued in a deterministic manner across nodes and with coordinated parameter value selections for common communication end-points, data volume, *etc.*, of sends and receives.

Before we discuss the design of our new Scala-H-Trace replay tool, we first review the conventional design of replay for lossless traces in ScalaTrace [4]. For lossless traces, all participating nodes parse the trace file and *only act on events if the current node is a member of the participant list*. Then all nodes reissue MPI events one by one by identifying loops using the PRSD information and extracting individual MPI function parameters from the recorded trace. This replay tool also simulates the computation time by sleeping according to the recorded delta times. This ensures that the time-sensitive performance characteristics, such as network and I/O contention, and the original application runtime are preserved. In addition, the replay tool also helps to verify the correctness of the trace. By design, it ensures absence of deadlocks if the input trace is deadlock-free.

A. Scala-H-Trace Replay

With the histogram-based trace, the existing parallel replay functionality requires a complete overhaul to cope with statistical data instead of precise data. In our Scala-H-Trace approach, all participating nodes parse the entire trace file during replay. In contrast to ScalaTrace, *all nodes read and interpret* all MPI events. We ensure that the trace is interpreted in such a way that all nodes will agree on each specific random value selected from each histogram in the trace. This guarantees that, for example, the sender and the receiver will perform a communication operation with matching message volume. To achieve this, we designed a fully distributed algorithm. At initialization, all nodes select the same random seed. During replay, all nodes use the same sequence of random numbers to interpret each histogram-recorded parameter so that all nodes agree on the random value upon each selection of a replay parameter within the range of 0 and the total number of elements in the histogram. Nonetheless, a given node issues MPI calls only if it is a participant of the recorded events. In this way, no coordination via back-channel communication is required and the communication overhead that would otherwise be required to coordinate random parameter value selection is avoided.

B. Replay the Histogram-recorded Point-to-Point Messages

A unique challenge for our Scala-H-Trace replay algorithm is the replay of the point-to-point communication when the source/destination ranks are recorded with histograms. Conventional ScalaTrace replay design suffers from deadlock problems under such scenario. Consider a histogram-recorded point-to-point communication event between a sender $S1$ and a receiver $R1$:

```
RSD1:<MPI_Send,Range:{R1-x,..,R1+x}>[rank:S1]
RSD2:<MPI_Recv,Range:{S1-y,..,S1+y}>[rank:R1]
```

Without the coordinated random parameter value selection, the sending node $S1$ may randomly choose node $R2$ ($R1-x<R2<R1+x$) as the destination, while the receiving node $R1$ may randomly choose $S2$ ($S1-y<S2<S1+y$) as the source. As a result, this point-to-point communication will deadlock because the send and receive operations (or the corresponding wait operations) cannot proceed due to end-point mismatch.

This problem is addressed with our Scala-H-Trace replay design by always coordinating the random parameter value selection across nodes and generating receive operations on-the-fly. During the random selection of replay parameters, end-points of MPI_Send/MPI_Isend events are selected. Upon encountering a send event, once a node identifies itself as a receiver, the receiver node issues a receive call (MPI_Irecv) instead of a send. Hence, all receive communication events like MPI_Recv and MPI_Irecv are ignored. Since a particular receiver can also be a sender, only MPI_Irecv calls are internally issued followed by an internal MPI_Wait call. Such internal MPI_Wait calls are issued last, after all ranks have been parsed and all MPI_Send/MPI_Isend/MPI_Irecv calls have been issued. Any MPI_Wait/MPI_Waitall for receive operations in the original recorded trace are subsequently ignored (wait operations for non-blocking sends are preserved). Hence, for the same example above, RSD2 is ignored during replay. Assuming node $S1$ randomly chooses $R3$ ($R1-x<R3<R1+x$) as the destination, because of the coordinated parameter value selection, node $R3$ identifies itself as the receiver,

and a corresponding receive from $S1$ is therefore posted. Again, the histogram-based tracing and probabilistic replay inevitably involve the tradeoff between accuracy and efficiency. In this example, the Scala-H-Trace replay tool can only approximately reproduce the original communication pattern.

We further addressed replay challenges due to collectives via event reordering and proved that deterministic replay after reordering is deadlock free. Due to space constraints, this work is beyond the scope of this paper [12].

V. Inter-node Trace Compression

Beyond the histogram-based tracing technique, we have also designed a novel inter-node trace compression algorithm to fully exploit the SPMD nature of the scientific codes. While the histogram-based approach focuses on improving the compression of the application parameters, including the MPI parameters and the loop iteration count, the inter-node compression algorithm discussed in this section aims at an even higher level, namely the compression of two sequences of events, which is not addressed by the histogram-based approach. Note that, although they solve orthogonal problems, inter-node compression actually benefits from the histogram-based approach as parameter matching is accomplished by histogram merging (instead of requiring exact parameter matches for a merge).

The SPMD nature of the scientific codes causes participants of a parallel application to produce similar per node traces. E.g., if we treat a trace as a sequence of MPI events, traces from different nodes tend to have similar subsequences that contains most of MPI events. In addition, loop structures captures by PRSDs in ScalaTrace facilitate compression as traces from different nodes tend to have similar PRSD nests. ScalaTrace originally required not just similar but rather *identical* patterns, i.e., it failed to fully exploit similarities for inter-node trace compression. More specifically, identical loop structures, i.e., PRSDs with identical length, iteration count, and MPI event sequence were required. While this approach works well with the perfect SPMD-style codes, it is subject to scalability problems when traces slightly diverge between nodes. For the example below, let T_i be traces where each letter in a trace "string" represents an MPI event and the pair of parentheses represent the loop structures. The coarse-grained trace matching algorithm may merge the per-node traces T_1 and T_2 to T_3. Yet, an ideal compression would instead be trace T_4.

$$T_1 : a(b(bcb)db)a \quad T_2 : a(b(beb)fb)a$$
$$T_3 : a(b(bcb)db)(b(beb)fb)a \quad T_4 : a(b(bceb)dfb)a$$

Only if the inter-node trace matching algorithm does not miss the structural similarities can the probabilistic communication parameter compression (discussed below) be fully utilized. Hence, we have designed a novel, fine-grained event matching algorithm that recursively compares and merges the nested loop structures. Algorithm 1 outlines the recursive trace merging technique. This algorithm traverses traces of two nodes, T_1 and T_2, to identify the matching event pairs. Stand-alone events are compared by their MPI parameter values with the function PARAM_MATCH. If two events start structurally identical loop nests, i.e., loop nests with equal depth and equal iteration counts at each nest level, the function MATCH_LOOP is called. MATCH_LOOP then matches the loop bodies at each level starting from the innermost nest and recursively call itself if new matching loop heads are found. When a pair of matching events is identified, the preceding unmatched sequences are sequentially linked by DO_MERGE. Since we forward the cursors for both input sequences when a match is found, this algorithm, in practice, has a complexity of $O(n)$, where n is the length of the input traces given that two input traces are similar.

Algorithm 1 Recursive Trace Matching Algorithm

Precondition: T_1 and T_2: input per node traces
Postcondition: T_1 and T_2: recursively merged trace

```
 1: procedure MATCH_TRACE($T_1$, $T_1$)
 2:     for iter1 ← $T_1$.head, $T_1$.tail do
 3:         for iter2 ← $T_2$.head, $T_2$.tail do
 4:             if iter1 and iter2 start identical loop nests then
 5:                 MATCH_LOOP(iter1, iter2, depth_of_nest)
 6:             else
 7:                 if PARAM_MATCH(iter1, iter2) then
 8:                     DO_MERGE(iter1, iter2)
 9:                 end if
10:             end if
11:         end for
12:     end for
13: end procedure

14: procedure MATCH_LOOP(loop1, loop2, depth)
15:     for iter1 ← loop1.head, loop1.tail do
16:         for iter2 ← loop2.head, loop2.tail do
17:             if iter1 == loop1.head && iter2 == loop2.head && PARAM_MATCH(iter1, iter2) then
18:                 DO_MERGE(iter1, iter2)
19:             end if
20:             if iter1 and iter2 are single events && PARAM_MATCH(iter1, iter2) then
21:                 DO_MERGE(iter1, iter2)
22:             end if
23:             if iter1 and iter2 start identical loop nests then
24:                 MATCH_LOOP(iter1, iter2, depth_of_nest)
25:             end if
26:         end for
27:     end for
28:     if depth >1 then
29:         MATCH_LOOP(iter1, iter2, depth-1)
30:     end if
31: end procedure
```

Algorithm 1 may still fail to generate the best inter-node compression because traversing two sequences with the double-nested loop structure does not guarantee identifying the longest common subsequence. As an example, consider T_1 and T_2 below. Algorithm 1 will return the sequence T_3:

$$T_1 : abbbbb \quad T_2 : bbbbba \quad T_3 : bbbbbabbbbb$$

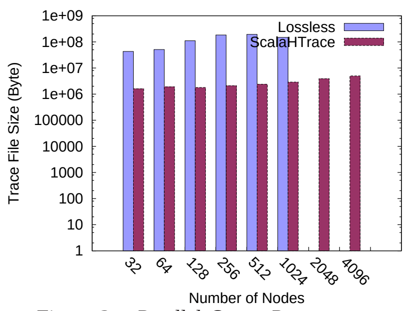

Figure 2. Parallel Ocean Program

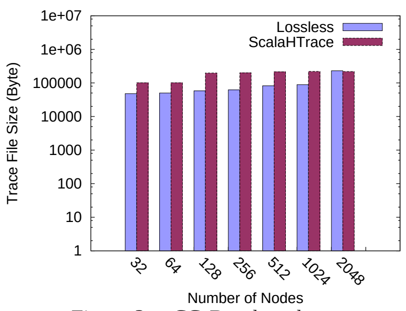

Figure 3. CG Benchmark

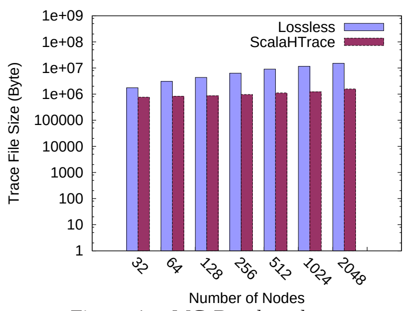

Figure 4. MG Benchmark

Matching event *a* is found before the longer subsequence *bbbbb*. To solve this problem, we integrated a *Weighted Longest Common Subsequence* (WLCS) algorithm into Algorithm 1. WLCS is adapted from the classic *Longest Common Subsequence* (LCS) algorithm. Since the loop structures in the trace should be treated as a whole, we enhanced LCS such that the matching loop structures are evaluated with a weight that equal to the length of their LCS. This results in compressing *bbbbb* first in the example above.

VI. Experimental Results

We evaluated Scala-H-Trace in three aspects: (1) its effectiveness of trace compression, (2) its statistical trace replay feature, and (3) its trace compression sensitivity to merge precision level settings. Experiments (1) and (2) utilize both the histogram compression approach and the WLCS-based recursive inter-node compression algorithm. Most of our experiments were conducted on Jaguar, the Cray XT4 system at ORNL. Each of compute node features a 2.1 GHz quad-core AMD Opteron 1354 processor and 8GB of DDR2 memory. The login nodes run a full-featured Linux version while the compute nodes run the Compute Node Linux microkernel. Due to unavailability of Jaguar in the final experimentation phase, the MG experiments were conducted on Jugene, an IBMBlue Gene/P system with 73,728 compute nodes and 294,912 cores, 2 GB memory per node, and the 3D torus and global tree interconnection networks.

We analyze the efficacy of Scala-H-Trace using a production-scale application, the Parallel Ocean Program (POP) [13], as the main challenge. The Parallel Ocean Program (POP) is an ocean circulation model developed at Los Alamos National Laboratory. Our experiments exercise a one degree grid resolution in which the problem size is 320x384 blocks and the individual block size is 5x6 resulting in a total of 4096 (64x64) blocks distributed to individual nodes. POP exhibits non-SPMD behavior, which leads to trace file size increases with the number of nodes for conventional trace tools, including ScalaTrace. POP is a large-scale application with challenging communication patterns. There five different dominant patterns equivalent to five micro-benchmarks, yet in combined complexity. Hence, this application provides an opportunity to show-case the effectiveness of histogram-based trace collection of Scala-H-Trace. We conducted experiments by varying the maximum number of blocks assigned to each node.

We further utilize the CG and MG benchmarks from the NAS Parallel Benchmark (NPB) suite of inputs sizes C to study the efficacy of Scala-H-Trace for different types of application behavior. Both CG and MG mostly exhibit SPMD behavior but differ significantly in the communication pattern impacting the compression effectiveness during trace collection. These two benchmarks are selected from the NAS benchmarks also because they were the challenging cases for ScalaTrace's lossless compression: Both were reported to result in sub-linear increases in the trace file size for ScalaTrace [4]. We have also tested Scala-H-Trace with the remaining NPB codes. They result in nearly constantly sized traces. In fact, Scala-H-Trace performs at least as well as the original ScalaTrace, which has been shown to handle NPB codes very well — with the exception CG and MG (the focus of this paper).

A. Trace Compression Effectiveness

We collected traces based on two different compression techniques. First, the original ScalaTrace is used, in which loop details and parameter values are captured losslessly and inter-node trace compression is performed with the coarse-grained matching scheme. Second, our novel histogram-based trace compression featuring Scala-H-Trace is used, in which trace information is collected in histograms for events and parameters that otherwise would not have compressed with the lossless trace compression, and inter-node compression is performed recursively. Trace file sizes are assessed under strong scaling, where we vary the number of nodes while keeping the overall problem size fixed.

Figure 2 depicts the trace file size for both lossless and histogram-based traces when varying the number of nodes. Note that the y axis is in log scale. Since POP exhibits non-SPMD behavior, we observe a linear increase in the trace file size in the case of lossless trace collection up to 256 nodes. The trace file size then stabilizes for 512 nodes and even declines for 1024 nodes. We identified that the timestep behavior becomes more regular at these levels, resulting in more effective inter-node compression. But we again observed an increasing trend in the case of 2048 nodes. For 2048 nodes and above, we could not even collect traces

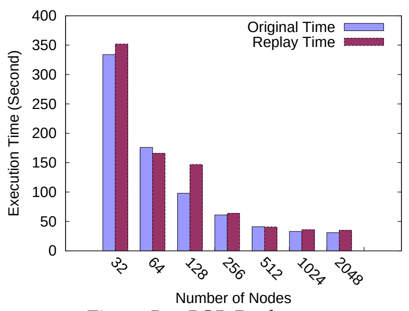

Figure 5. POP Replay

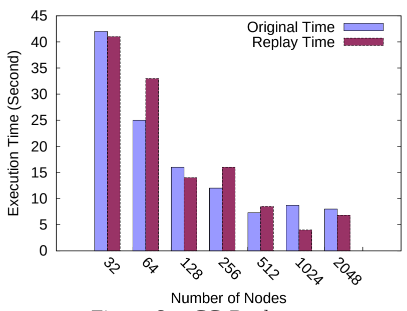

Figure 6. CG Replay

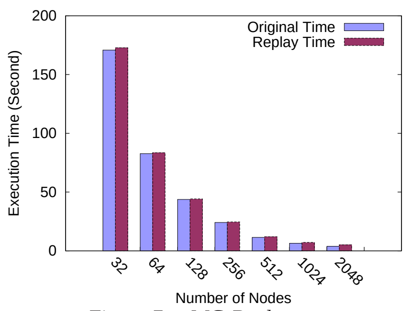

Figure 7. MG Replay

anymore as the trace file size was growing unmanageably fast and the time taken to merge hundreds of megabytes of per-node traces became prohibitive. With the histogram-based approach, there is a sub-linear increase in the trace file size. Moreover, histogram-based trace files are two orders of magnitude smaller than the lossless traces. This considerable reduction is obtained by aggressive compression of events and their function parameters in histograms. This clearly shows the efficacy of Scala-H-Trace to collect concise trace files even with applications exhibiting irregular behavior.

Figures 3 depicts trace file size for the CG benchmark. We observe an interesting trend in CG in which the trace file size for lossless traces is consistently 50% less than that of the histogram traces up to 1024 nodes, yet sizes match at 2048 node. Even though lossless traces are initially smaller than histogram traces, there is a consistent increase in the trace file size for the lossless case. In contrast, the size of histogram traces is almost constant with the increase in number of nodes. For lossless traces, non-matching function parameters for events with the same call stack are collected in vectors associated with a participant rank list. This representation is more concise than histograms for smaller number of nodes. With thousands of nodes, the vector-participant list pair for each event has increased in size to where it is at par with histogram traces. Unlike vector-participant lists, histogram representation is constant with the increase in number of nodes as the number of bins is fixed during the application run and even the outlier participant rank information is absorbed as constants in bins. It should also be noted that the trace file size for CG is in the order of hundreds of kilobytes. For larger applications with a similar communication behavior as CG yet with trace file sizes in hundreds of megabytes, such a linear (or even sub-linear) growth for lossless traces may simply not be scalable due to inter-node merge overheads, as discussed.

Figure 4 depicts the results for MG. MG exhibits a double nested 7-point stencil communication pattern in the 3D space. Due to the regular communication pattern and data-independent program behavior, compressing the MPI parameter values of MG works well for both lossless and histogram-based approaches. However, due to the slightly diverged per-node program behavior within a loop, the original inter-node compression algorithm of ScalaTrace failed to merge across communication groups. This caused trace sizes to increases linearly with the number of nodes. In contrast, with our novel fine-grained recursive approach, similar PRSDs are merged and the trace size grows sub-linearly, i.e., by a factor of two as the number of nodes is increased by a factor of 64.

B. Histogram-based Trace Replay

In the second set of experiments, we studied the replay effectiveness of histogram-based traces by comparing the original application runtime with the time taken to replay the recorded events. For these experiments, we always set the merge precision level to 0%, which is the most aggressive compression possible with Scala-H-Trace. More accurate replay may result from higher precision levels at the cost of larger traces, as will be discussed in Section VI-C.

Figure 5 depicts the replay time of histogram-based trace events compared to that of the application's original execution time. The compressed traces are fully forced histogram trace events where any non-matching function parameters or loop iterations are collected as histograms. Even with these traces, we see that the replay time for traces collected for 32-512 number of nodes are within 5% of the original execution time (with the exception of replay time for 128 nodes). Replay time accuracy drops to 12% for 1024 and 2048 nodes. Due to our experiment with strong scaling for POP, the original execution time for both 1024 and 2048 nodes (30 seconds) is much lower than that for fewer nodes (>100 seconds) so that even small deviations in absolute values during replay increase the error percentage. We conjecture that such deviations are unrealistic as POP for this particular input does not scale beyond 512 nodes so that such short times are unrealistic. Similarly, this problem would not occur under weak scaling as runtimes would not decrease with larger number of nodes. We observe a deviation of over 30% in the case of 128 nodes. We found that when the inter-node merge fails, the RSDs for events that happen roughly the same time will be recorded far apart in the trace. Since the new replay approach requires all nodes to interpret every event in the trace, these unmerged events introduce increased synchronization, which forces some nodes to wait for other nodes to join in a particular send event.

Figure 6 depicts the replay time for the CG benchmark.

In the majority of cases, the replay time is with 10% to 15% of the original application runtime. This inaccuracy is mainly caused by the random selection of application parameters such as the iteration count. This inaccuracy is further magnified by the fact that the input stops scaling at 512 nodes such that even a small absolute error increases the error percentage considerably. Again, this is a fundamental tradeoff between accuracy and trace size.

The replay time for MG under strong scaling is depicted in Figure 7. The averaged inaccuracy is 8.2%. We observe up to 34.2% inaccuracy for 2,048 nodes, which is due to an excessively short runtime of 3.8s with an absolute error of just 1.3s. For 1,024 nodes, this decreases to 12.5% and for 512 to 5.3% and so on indicating that the problem is only due to excessively short runtimes. After discarding this outlier due to strong scaling limitations, the replay timing accuracy for MG is high. As discussed before, with the recursive inter-node trace compression, we are able to achieve a nearly constant trace sizes for MG even without the histogram-based probabilistic approach. Due to the elimination of the imprecision, the timing behavior of the trace replay highly resembles that of the original MG benchmark.

C. Trace Sensitivity Study

Finally, we study the effect of varying merge precision levels on trace file sizes. This experiment serves as an illustration for the benefits of user-specified merge precision levels as a means to steer compression.

Figure 8 depicts the impact of varying merge precision levels on the final trace file size. We fixed the number of nodes to 512 for POP and measured trace file sizes for varying merge precision levels. We observe that even with a small decrease in the merge precision from 100% to 95%, the trace size reduced by more than a factor of three. This significant reduction is due to merging events with varying numbers of loop iterations for the timestep in POP.

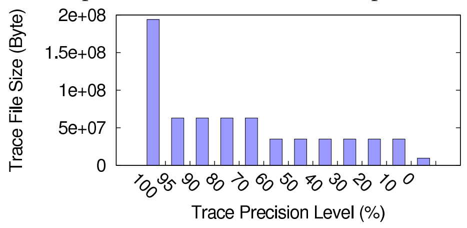

Figure 8. POP Trace Sensitivity for 512 nodes

The trace file size is constant up to a 70% merge precision level. At 60% precision, sizes drop again by almost 50%. This second reduction has been attributed to function parameters collected as histograms — in contrast to compression failures or non-scalable vector style compression of the lossless approach. Finally, another three-fold reduction in trace sizes is observed for forced histograms (0% merge precision level). At the 0% merge precision level, all non-matching values are represented as histograms, which results in the most concise trace possible with Scala-H-Trace. Overall, sensitivity experiments for merge precision levels show that small reductions in precision can significantly reduce the overall trace sizes. This particularly aids production-scale codes like POP, which otherwise cannot be feasibly traced without loss of information for thousands of nodes.

VII. Related Work

There are several tools, such as TAU [14], Vampir [1], Paraver [15] and SCALASCA [16], that capture communication and/or I/O trace events using library instrumentation similar to Scala-H-Trace. But only a few employ trace compression techniques to control the trace file size. Many of these tools depend on zlib for compression, which compresses blocks of data without preserving the structure of the trace, *i.e.*, post-processing/analysis only becomes feasible after decompression. This also increases the memory requirements, effectively rendering trace analysis infeasible on commodity desktops or laptops and sometimes even high-end workstations, depending on the uncompressed trace size. Unlike these techniques, ScalaTrace [4] compresses traces while preserving the trace structure in terms of order of events. As a result, post-processing/analysis can be performed without decompression. We utilize this concept of structure preserving compression in Scala-H-Trace. Yet while ScalaTrace and any of the aforementioned tracing tools record lossless traces with a subset or all event parameters, Scala-H-Trace establishes a different methodology. Parameters, event frequencies and participant lists of nodes are recorded as histograms when lossless compressing cannot be established within a user-specified merge precision level. Employing statistical methods results in more concise traces even for non-SPMD programs at the expense of loss of information. Our replay tool uses an algorithm to issue events on-the-fly using the compressed traces, much like ScalaTrace. Yet recorded parameters are replayed in a probabilistic manner, which creates novel challenges that are met by our distributed approach to coordinate event replay across nodes.

The mpiP tool, a lightweight profiling library for MPI applications, collects statistical information about MPI functions [3]. It collects aggregate metrics like number of MPI events issued by the application and average execution times. This is useful to provide high-level information on communication and I/O calls. In contrast, Scala-H-Trace captures all events in traces and employs more sophisticated histogram bins only when the need arises for applications exhibiting non-SPMD behavior. Beside the histogram information, we also record outlier information associated with each bin to detect communication bottlenecks and to provide a “big picture” of communication and I/O events in applications.

Kluge *et al.* [17] employ pattern matching techniques similar to ours to capture POSIX I/O calls in parallel programs. Unlike our approach, they perform post-mortem pattern matching only after collecting the application traces. They read the collected trace and create an I/O dependency graph thereby preserving the event order to do pattern matching. Even though post mortem pattern matching reduces the trace volume, this approach limits its usefulness in memory constrained systems like the IBM BlueGene family. Without online compression, either the memory footprint increases by holding the recorded trace or trace events are frequently written to disk, which affects the application execution behavior. They also do not employ pattern matching across nodes so that a trace file per node is required. This limits their approach in that they struggle with applications utilizing thousands of nodes due to parallel file system constraints. Our approach is immune to such limitations as a single trace file captures the behavior of all nodes with statistical information on a per-event and per-parameter basis.

Gao *et al.* [18] developed an event trace compression technique that performs static analysis on the application binary and collects loops and functions as structures. Along with these structure, a path grammar is constructed on-the-fly. Path grammars are then utilized to encode paths taken during execution. These structures are compressed individually and stored. Even the iteration count is stored along with the compressed structure traces. This loosely resembles the RSD and PRSD technique used in related work [8], [9], [19], [5]. But unlike Gao *et al.*'s work, our tool does not require the construction of grammars for individual applications separately. Our work employs a generalized trace compression approach based on call path stacks and records parameters exploiting statistical means. It is sufficient to link the tool library along with the application to collect traces. This generalization also enables comparative trace studies between two different applications.

VIII. Conclusion

We presented the design and implementation of Scala-H-Trace, which provides novel capabilities for more aggressive trace compression than any previous approach. Scala-H-Trace utilizes histograms based on a user-specified merge precision level. It features a distributed approach to deterministically replay statistical histogram traces where events are reissued without decompressing the original trace file. Experimental results demonstrate the ability to obtain a single, near constant sized trace file, even for production-scale scientific applications such as POP with non-SPMD behavior. Results also show that replay time for traced events are within 12%-15% of the original application runtime in majority of the cases, even for the fully forced histograms.

References

[1] W. E. Nagel, A. Arnold, M. Weber, H. C. Hoppe, and K. Solchenbach, "VAMPIR: Visualization and analysis of MPI resources," *Supercomputer*, vol. 12, no. 1, pp. 69–80, 1996.

[2] H. Brunst, D. Kranzlmüller, and W. Nagel, "Tools for Scalable Parallel Program Analysis - Vampir NG and DeWiz," *The International Series in Engineering and Computer Science, Distributed and Parallel Systems*, vol. 777, pp. 92–102, 2005.

[3] 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.

[4] M. Noeth, F. Mueller, M. Schulz, and B. de Supinski, "Scalatrace: Scalable compression and replay of communication traces in high performance computing," *Journal of Parallel Distributed Computing*, vol. 69, no. 8, pp. 969–710, Aug. 2009.

[5] P. Ratn, F. Mueller, B. R. de Supinski, and M. Schulz, "Preserving time in large-scale communication traces," in *International Conference on Supercomputing*, Jun. 2008, pp. 46–55.

[6] K. Vijayakumar, F. Mueller, X. Ma, and P. C. Roth, "Scalable i/o tracing and analysis," in *Workshop on Petascale Data Storage*, 2009, pp. 26–31.

[7] "MPI-2: Extensions to the message-passing interface," July 1997. [Online]. Available: http://www.mpi-forum.org/docs/docs.html

[8] P. Havlak and K. Kennedy, "An implementation of interprocedural bounded regular section analysis," *IEEE Transactions on Parallel and Distributed Systems*, vol. 2, no. 3, pp. 350–360, Jul. 1991.

[9] J. Marathe and F. Mueller, "Detecting memory performance bottlenecks via binary rewriting," in *Workshop on Binary Translation*, Sep. 2002.

[10] X. Wu, F. Mueller, and S. Pakin, "Automatic generation of executable communication specifications from parallel applications," in *International Conference on Supercomputing*, 2011, pp. 12–21.

[11] "The parallel ocean program (POP)," Los Alamos National Laboratory, 1996, http://climate.lanl.gov/Models/POP/.

[12] X. Wu, K.Vijayakumar, F. Mueller, X. Ma, and P. C. Roth, "Probabilistic communication and i/o tracing with deterministic replay at scale," Dept. of Computer Science, North Carolina State University, Tech. Rep. TR 2011-6, 2011.

[13] P. W. Jones, P. H. Worley, Y. Yoshida, J. B. White, III, and J. Levesque, "Practical performance portability in the parallel ocean program (pop): Research articles," *Concurr. Comput. : Pract. Exper.*, vol. 17, no. 10, pp. 1317–1327, 2005.

[14] S. S. Shende and A. D. Malony, "The tau parallel performance system," *Int. J. High Perform. Comput. Appl.*, vol. 20, no. 2, pp. 287–311, 2006.

[15] 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*, ser. Transputer and Occam Engineering, vol. 44, Apr. 1995, pp. 17–31.

[16] M. Geimer, F. Wolf, B. J. N. Wylie, E. Abraham, D. Becker, and B. Mohr, "The scalasca performance toolset architecture," in *Workshop on Scalable Tools for High-End Computing*, Jun. 2008.

[17] M. Kluge, A. Knüpfer, M. Müller, and W. E. Nagel, "Pattern matching and i/o replay for posix i/o in parallel programs," in *Euro-Par '09: Proceedings of the 15th International Euro-Par Conference on Parallel Processing*. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 45–56.

[18] X. Gao, A. Snavely, and L. Carter, "Path grammar guided trace compression and trace approximation," in *Symposium on High-Performance Distributed Computing*, 2006, pp. 57–68.

[19] M. Noeth, F. Mueller, M. Schulz, and B. R. de Supinski, "Scalable compression and replay of communication traces in massively parallel environments," in *International Parallel and Distributed Processing Symposium*, Apr. 2007.