ScalaTrace: Tracing, Analysis and Modeling of HPC Codes at Scale

Frank Mueller[1], Xing Wu[1], Martin Schulz[2], Bronis R. de Supinski[2], and Todd Gamblin[2]

[1] Dept. of Computer Science, North Carolina State University, Raleigh, NC 27695-7534, mueller@cs.ncsu.edu
[2] Lawrence Livermore National Laboratory, Center for Applied Scientific Computing, Livermore, CA 94551

Abstract. Characterizing the communication behavior of large-scale applications is a difficult and costly task due to code/system complexity and their long execution times. An alternative to running actual codes is to gather their communication traces and then replay them, which facilitates application tuning and future procurements. While past approaches lacked lossless scalable trace collection, we contribute an approach that provides orders of magnitude smaller, if not near constant-size, communication traces regardless of the number of nodes while preserving structural information. We introduce intra- and inter-node compression techniques of MPI events, we develop a scheme to preserve time and causality of communication events, and we present results of our implementation for BlueGene/L. Given this novel capability, we discuss its impact on communication tuning and on trace extrapolation. To the best of our knowledge, such a concise representation of MPI traces in a scalable manner combined with time-preserving deterministic MPI call replay are without any precedence.

Keywords: High-Performance Computing, Message Passing, Tracing

1 Introduction

Scalability is one of the main challenges of petascale computing. One central problem lies in a lack of scaling of communication. However, understanding the communication patterns of complex large-scale scientific applications is non-trivial. An array of analysis tools have been developed, both by academia and industry, to aid this process. For example, Vampir is a commercial tool set including a trace generator and GUI to visualize a time line of MPI events [2]. While the trace generation supports filtering, trace files, which are stored locally, grow with the number of MPI events in a non-scalable fashion. Another example is the mpiP tool that uses the profiling layer of MPI to gather user-configurable aggregate metrics for statistical analysis [10]. Locally stored profiling files are constrained in size by the number of unique call sites of MPI events, which is independent of the number of nodes. However, mpiP does not preserve the structure and temporal ordering of events, which limits its use to high-level analysis.

Other communication analysis tools have similar constraints: either their storage requirements do not scale or they are lossy with respect to program structure and temporal ordering.

In contrast to prior work, our work develops a scalable trace-driven approach to analyze MPI communication that can represent lossless, full traces in constant size. We demonstrate in our results that our objective has been achieved for a number of benchmarks. We have further developed tools (a) to replay communication, optionally with widely preserved timing information, (b) to detect inefficiencies in the utilization of the communication API, and (c) to extrapolate traces for strong scaling.

2 Lossless Tracing

Communication analysis tools are currently constrained in that either their storage requirements do not scale or they fall short in tracing all events by only providing aggregate statistics.

In contrast to prior work, ScalaTrace provides a scalable trace-driven approach to analyze MPI communication. While past approaches fail to gather full traces for hundreds of nodes in a scalable manner or only gather aggregate information, we have designed a framework that extracts full communication traces orders of magnitude smaller, if not near constant size, regardless of the number of nodes while preserving structural information and temporal event order.

Our trace-gathering framework utilizes the MPI profiling layer (PMPI) to intercept MPI calls during application execution. Profiling wrappers trace which MPI function was called along with call parameters within each node, such as source and destination of communications, yet without recording the actual message content. This intra-node information (task-level) is compressed on-the-fly. We also perform inter-node compression upon application termination to obtain a single trace file that preserves structural information suitable for lossless replay.

Intra-Node Compression: Within each node, we compress MPI call entries, generally repeated due to a code's loop structure, on-the-fly. To this extent, regular section descriptors (RSDs) are exploited to express MPI events nested in a single loop in constant size [3] while power-RSDs (PRSDs) are utilized to specify recursive RSDs nested in multiple loops [5]. 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 construct correspond to the code in Figure 1. The algorithmic details of MPI event compressions over PRSDs can be found elsewhere [7, 8].

To efficiently compress events, a set of generic and another set of domain-specific optimizations are performed. (1) Calling sequences are identified by generating a signature derived from a stack walk. Thus, call origins to common routines (*e.g.*, MPI_Send at call site 1) can be distinguished. (2) Communication end-points are encoded in a location-independent manner (relative to the rank of the current MPI task). This fosters identical encoding, *e.g.*, for stencil codes, across nodes. (3) Request handles are identified by a relative index into a handle buffer of constant size, which is dynamically updated. This abstracts

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

Fig. 1. Sample Code for PRSDs

from runtime-dependent data structures in a portable manner, *e.g.*, for handles returned by asynchronous communication calls such as MPI_Isend. (4) Iterative constructs with an indeterministic number of repetitions of a common event type are aggregated into a single event. For example, an application may wait for the completion of n asynchronous events using MPI_Waitsome in a loop, yet the MPI call may aggregate multiple completions as a result of each call. This will be abstracted as n calls.

Inter-Node Compression: Local traces are combined 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 BlueGene/L (BG/L) with a limited number of I/O nodes, could severely suffer under such a load. To guarantee scalability, we instead employ cross-node compression, step-wise and in a bottom-up fashion over a binary tree network overlay. To this extent, events and structures (RSD / PRSDs) of nodes are merged when events, parameters, structure and iteration counts match (see [7, 8] for algorithmic details).

We again employ a set of generic and domain-specific optimizations: (1) Sequences of nodes/task IDs are encoded using PRSDs, which allows a concise representation even for subsets of nodes as traces from different nodes are merged within the reduction tree. We utilize a radix tree whose encoding fosters efficient PRSD representations of sets of task IDs. (2) Events are temporally reordered when they originate from different nodes (and have no causal relation) to result in a more concise representation.

3 Deterministic Replay

One of the objectives of collecting communication traces is to analyze them off-line. One key virtue of our environment is that one can replay communication traces in a generic manner, even without the availability of application code. All that is needed is the trace itself. Our replay engine will be discussed in more detail in the preliminary results.

We have designed and implemented a replay engine that issues communication calls in the same order that they were originally issued by an application. The input to the replay engine consists of the compressed trace. The replay engine itself does not actually decompress this 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 in an interpretative manner. When it encounters an RSD or PRSD, it issues calls iteratively observing the structure, frequency and parameters of communication calls. Hence, our structure-preserving compression scheme is key to a scalable replay methodology, which does not require excess amounts of memory. In fact, its memory requirement is loosely bounded by the size of the *compressed* trace, which is often of constant size.

Using the replay engine, we conducted experiments to verify the correctness of our scalable compression approach. We replayed compressed traces to ensure MPI semantics are preserved, to verify that the aggregate number of MPI events per MPI call matches that of the original code and that the temporal ordering of MPI events within a node are observed. The results of communication replays confirmed the correctness of our approach. During replay, all MPI calls are triggered over the same number of nodes with original payload sizes, yet with a *random* message payload (content). Thus, the replay incurs comparable bandwidth requirements on communication interconnects, albeit with potentially different contention characteristics since event times are not preserved (addressed below). Communication replay also provides an abstraction from compute-bound application performance, which is neither captured nor replayed. This makes the replay mechanism extremely portable, even across platforms, which can benefit rapid prototyping and tuning. It also supports assessing communication needs of future platforms for large-scale procurements.

4 Preserving Time

The objective of trace analysis is generally to find inefficiencies in the code, *e.g.*, as indicated by load imbalance between nodes. Such analysis requires knowledge about the timing between events. Hence, conventional trace techniques simply attach a timestamp to all communication events. Such timestamps also facilitate a time-accurate replay. However, for a scalable compression-based tracing approach like ours, recording the precise timestamp is infeasible. Due to asynchronous event occurrence across nodes, the timing between nodes diverges over time so that absolute time differs. (This holds even if nodes were synchronized at a job start, which is generally not the case). Consequently, absolute timestamping would require recording the timing information for every single node without being able to compress, which leads to a linearly increasing trace size wrt. the total number of nodes.

Our trace compression scheme and the replay engine support two methods of capturing timing information of different tasks in computational sections (between any two communication calls) [9]. First, a low-cost statistical approach to capture delta times has been designed. Second, to resemble computational imbalance, a variation-preserving recording scheme was devised, still within a constant size trace representation, yet with a higher constant factor. Delta times denote the elapsed time between adjacent trace events. In contrast to absolute time, the relative notion of time makes delta times amenable to compression. Delta times are collected for event pairs. For example, RSD1 has two timing regions that will be captured: (a) from send to receive and (b) from receive to send (between consecutive loop iterations). Each delta time is associated with the terminal communication event, *i.e.*, at the beginning (prologue) of receive for (a) and the beginning of send for (b).

We dynamically create size-limited histograms of delta times suitable for our existing trace compression scheme. Based on a user-defined number of bins, delta times are recorded in an online balancing scheme to equalize bin volumes using a weighted subrange partitioning scheme (algorithm 2 in [9]).

5 Trace Extrapolation

Judging the effect of problem and/or task scaling on performance is a hard problem. While some advances have been made using application modeling or deriving similarities and characteristics from microbenchmarks [4, 6, 1], we follow a complementary direction. We extrapolate larger communication traces from smaller ones, which can then be used to replay these larger traces and empirically detect communication problems or project system requirements for future procurements of HPC systems. If communication is the impeding factor to scalability, our framework can aid in the analysis of codes and performance projections for existing and future systems.

The extrapolation of system overhead from small application runs to larger ones is a challenging problem that has not been solved. Yet, a number of subproblems become feasible with ScalaTrace. We consider the problem of strong (task) scaling. Given a *set* of communication traces of, say, 8, 16, 32 and 64 nodes, we can extrapolate the corresponding trace for n nodes of the same application. We devised a method to extrapolate larger communication traces from smaller ones for task (strong) scaling. The key insight is that communication parameters combined with PRSD iterators provide sufficient detail for this approach. *E.g.*, if a communication parameter depends on the number of columns of a matrix whose size is input dependent, then problem scaling will increase this parameter at a certain rate. Given a minimum of three data points, a fitting curve can be constructed to extrapolate the growth rate of this parameter. Thus, payloads can be adjusted according to such fitting curves. A larger problem size can also affect the number of timesteps of a convergence algorithm. By capturing the iterators of timesteps within histograms, such dependencies can be discovered and modeled again *via* curve fitting.

Similar to problem scaling, the size of a group communicator may depend on the total number of nodes, and this growth under task scaling can be extrapolated with curve fitting methods. For such node dependencies, we require $d + 1$ data points (traces) for a d-dimensional Cartesian layout and then apply Gaussian elimination to extrapolate parameters, such as communication end-points, for arbitrary sizes n. *E.g.*, after determining the dimensionality, we can infer the coefficients A, B, C, D for a given layout and solve for values V_i: $A \times n_i + B \times x_i \times y_i + C \times x_i + D = V_i$ for $i \in \{1, ..., d + 1\}$ where $n_i = f(z_i)$ to facilitate the calculation for row-major layouts. We have prototyped a trace extrapolation method for strong scaling that automatically transforms a set of traces of smaller size (number of nodes) to one of arbitrary size. The extrapolated traces have been replayed successfully, as reported in the next section.

6 Results

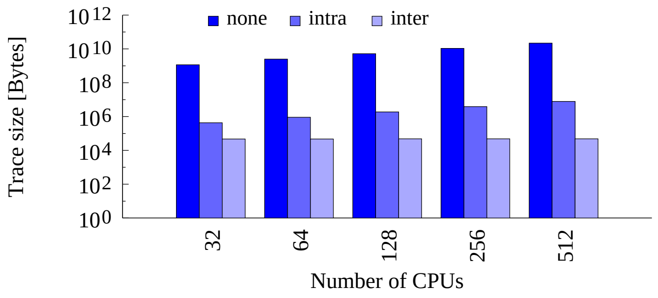

Fig. 2. LU Trace File, Varied # Nodes

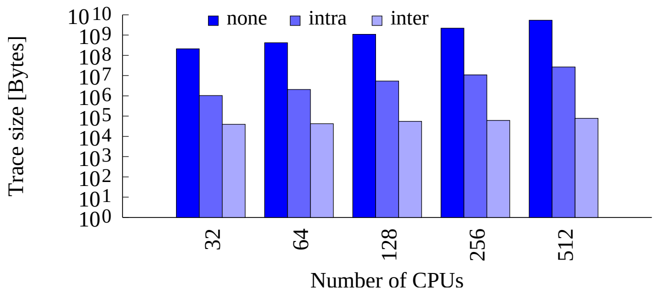

Fig. 3. CG Trace File, Varied # Nodes

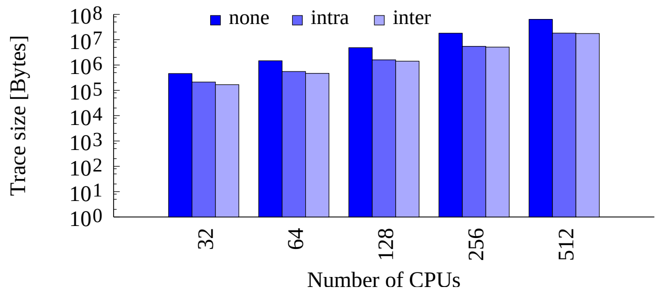

Fig. 4. IS Trace File, Varied # Nodes

We assessed the effectiveness of ScalaTrace through experiments with benchmarks and an application on BG/L. Our results confirm the scalability of our on-the-fly MPI trace compression by yielding orders of magnitude smaller or even near constant size traces for processor scaling and problem scaling.

We conducted experiments with the NAS Parallel Benchmark (NPB) codes for class C inputs [12] and UMT2k on BG/L. Figures 2-4 depict the trace file sizes on a logarithmic scale without compression (*none*), with local compression (*intra-node*) and with cross-node compression (*inter*). We identified three categories of codes wrt. inter-node compression efficiency: (1) those that result in near constant-size traces (DT, EP, LU, BT and FT), regardless of the number of nodes, (2) those with sub-linear scaling of trace size as the node count increases (MG and CG) and (3) those that do not scale yet (IS and UMT2k). The first class, represented by LU (Fig. 2), shows reductions for *intra* but only *inter* delivers constant size traces. The second class, represented by CG (Fig. 3), shows considerable compression at the node level (*intra*) and sub-linear (but not constant) sized traces for inter-node compression. The third class, represented by IS (Fig. 4), shows a similar trend but with a faster than linear growth rate for *inter*.

The next experiment assesses the effectiveness of delta times to resemble application behavior during replay. Figure 5 depicts the wall-clock time for the uninstrumented application, mpiP[10]-instrumented application and three replay options based on uncompressed, intra-node compressed and globally compressed traces. It shows highly accurate replay times irrespective of number of nodes and levels of compression, which is representative for all benchmarks.

We have also verified our extrapolation approach with a subset of the NAS Parallel Benchmark suite [13]. To verify the functional correctness, we replayed the extrapolated traces at the target node size. We instrumented both the replay engine and the original benchmarks with mpiP. We compared the MPI event statistics generated by mpiP. The results demonstrated the communication overheads at the target size are fully captured.

Besides the functional correctness, experimental results also indicate close resemblance in timing of extrapolated traces when replayed compared to application behavior at scaled size for up to 16k nodes (see Figure 6, dark/maroon

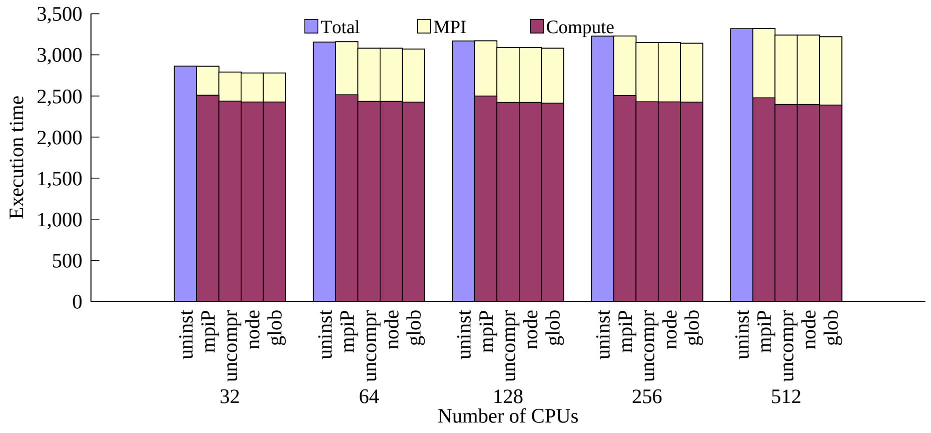

Fig. 5. FT Replay: Aggregate of All Nodes (BG/L)

bars are extrapolated, scaling is limited by input sizes). We are working on generalizing our approach to a large number of common communication patterns.

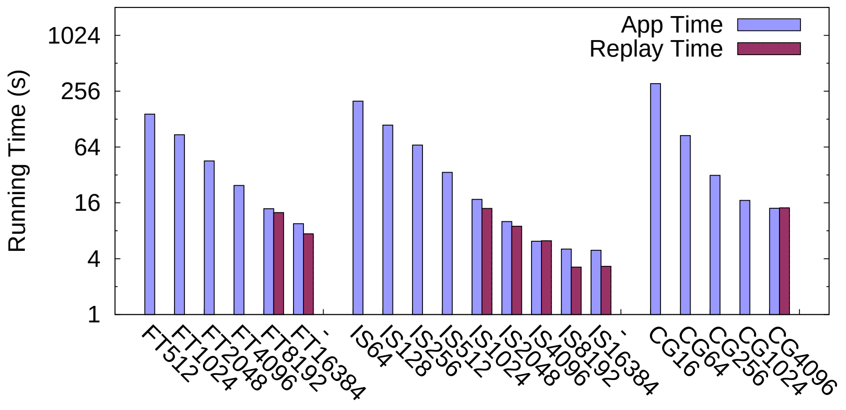

Fig. 6. Replay after Extrapolation on a BG/P

7 Conclusions

This paper gives an updated overview over ScalaTrace. ScalaTrace provides a scalable methodology for event tracing that has been demonstrated for MPI and I/O events. It annotates events with time-preserving information suitable for deterministic MPI call replay. Traces can further be extrapolated in the dimension of number of tasks (nodes) to assess communication scalability and assist procurement decisions for future HPC installations. Further information about ScalaTrace can be found elsewhere [7, 9, 8, 11].

8 Acknowledgements

This work was supported in part by NSF grants 0410203, 0429653, 0237570 (CAREER), 0937908, and 0958311. Part of this work was performed under the auspices of the U.S. Department of Energy by University of California Lawrence Livermore National Laboratory under contracts DE-AC05-00OR22725, W-7405-Eng-48, and DE-AC52-07NA27344 under LLNL-CONF-427005.

References

1. Bell, R., John, L.: Improved automatic testcase synthesis for performance model validation. In: International Conference on Supercomputing. pp. 111–120 (Jun 2005)
2. Brunst, H., Hoppe, H.C., Nagel, W.E., Winkler, M.: Performance optimization for large scale computing: The scalable VAMPIR approach. In: International Conference on Computational Science (2). pp. 751–760 (2001)
3. Havlak, P., Kennedy, K.: An implementation of interprocedural bounded regular section analysis. IEEE Transactions on Parallel and Distributed Systems 2(3), 350–360 (Jul 1991)
4. Kerbyson, D., Alme, H., Hoisie, A., Petrini, F., Wasserman, H., Gittings, M.: Predictive performance and scalability modeling of a large-scale application. In: Supercomputing (Nov 2001)
5. Marathe, J., Mueller, F., Mohan, T., de Supinski, B.R., McKee, S.A., Yoo, A.: METRIC: Tracking down inefficiencies in the memory hierarchy via binary rewriting. In: International Symposium on Code Generation and Optimization. pp. 289–300 (Mar 2003)
6. Marin, G., Mellor-Crummey, J.: Cross architecture performance predictions for scientific applications using parameterized models. In: SIGMETRICS Conference on Measurement and Modeling of Computer Systems (2004)
7. Noeth, M., Mueller, F., Schulz, M., de Supinski, B.R.: Scalable compression and replay of communication traces in massively parallel environments. In: International Parallel and Distributed Processing Symposium (Apr 2007)
8. Noeth, M., Mueller, F., Schulz, M., de Supinski, B.R.: Scalatrace: Scalable compression and replay of communication traces in high performance computing. Journal of Parallel Distributed Computing 69(8), 969–710 (Aug 2009)
9. Ratn, P., Mueller, F., de Supinski, B.R., Schulz, M.: Preserving time in large-scale communication traces. In: International Conference on Supercomputing. pp. 46–55 (Jun 2008)
10. Vetter, J., McCracken, M.: Statistical scalability analysis of communication operations in distributed applications. In: ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (2001)
11. Vijayakumar, K., Mueller, F., Ma, X., Roth, P.C.: Scalable multi-level i/o tracing and analysis. In: Petascale Data Storage Workshop (Nov 2009)

12. Wong, F., Martin, R., Arpaci-Dusseau, R., Culler, D.: Architectural requirements and scalability of the NAS parallel benchmarks. In: Supercomputing (1999)
13. Wu, X., Mueller, F.: Scalaextrap: trace-based communication extrapolation for spmd program. In: PPoPP (2011)