Scalable Compression and Replay of Communication Traces in Massively Parallel Environments *

Michael Noeth[1], Frank Mueller[1], Martin Schulz[2], Bronis R. de Supinski[2]

[1] North Carolina State University
Department of Computer Science
Raleigh, NC 27695-7534
mueller@cs.ncsu.edu

[2] Lawrence Livermore National Laboratory
Center for Applied Scientific Computing
Livermore, CA 94551
[schulzm,bronis]@llnl.gov

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 and present results of our implementation for BlueGene/L. Given this novel capability, we discuss its impact on communication tuning and beyond. To the best of our knowledge, such a concise representation of MPI traces in a scalable manner combined with deterministic MPI call replay are without any precedence.

1 Introduction and Overview

Scalability is one of the main challenges to 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. 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. 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, we propose 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 (Figure 1) 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. This intra-node information (task-level) is compressed on-the-fly. We perform inter-node compression upon application termination to obtain a single trace file that preserves structural information suitable for lossless replay.

We assess the effectiveness of our framework through experiments with benchmarks and an application on BlueGene/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.

*This work was supported in part by NSF grants CNS-0410203, CCF-0429653 and CAREER CCR-0237570. Part of this work was performed under the auspices of the U.S. Department of Energy by University of California Lawrence Livermore National Laboratory under contract No. W-7405-Eng-48, UCRL-CONF-227098.

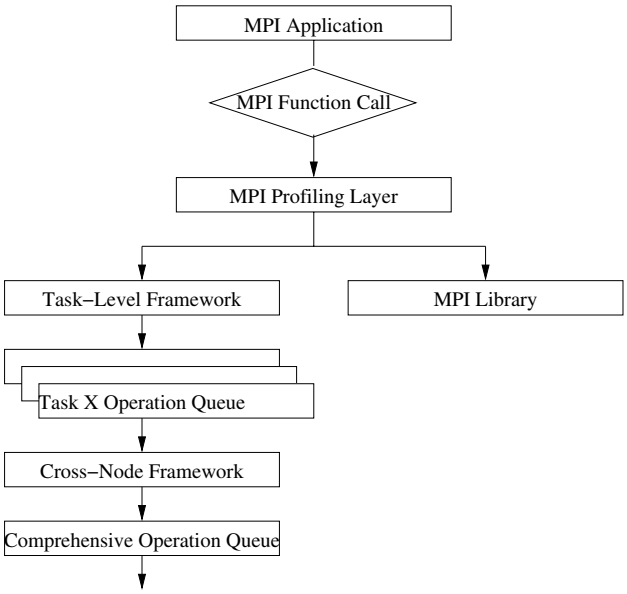


Figure 1. Interaction of Components

We have also designed a tool that replays our compressed trace independent of the original application and without decompressing the trace. Our replay mechanism, verifies our trace compression's correctness, can assist performance tuning MPI communication and facilitate projections of network requirements for future large-scale procurements.

To the best of our knowledge, such a concise, scalable representation of MPI traces combined with deterministic MPI call replay are without any precedence.

The paper is structured as follows. Section 2 and 3 detail intra- and inter-node trace compression. Sections 4 and 5 present the experimental framework and results. Section 6 contrasts this work with prior research. Section 7 summarizes our contributions.

2 Intra-Node/Task-Level Trace Compression

Lossless, yet space-efficient trace compression must preserve the structure and temporal order of events. Nonetheless, repetitive MPI events in loops with identical parameters should only require near constant size. We use the PMPI layer to provide wrappers of MPI calls that trace the source and destination of communication and other parameters of each MPI operation other than the actual message content. We compress these MPI call entries, generally repeated due to an application's loop structure, on-the-fly.

We extend regular section descriptors (RSDs) for single loops to express MPI events nested in a loop in constant size [5] while power-RSDs (PRSDs) are utilized to specify recursive RSDs nested in multiple loops [7]. 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.

The compression algorithm maintains a queue of MPI events and attempts to greedily compress the first matching sequence, an approach that is loosely based on the SIGMA scheme for memory analysis [3]. Our algorithm proceeds in four steps as depicted in Figure 2. First, head and tail of

```
Compress_Queue(Queue Op_Queue)
    Target_Tail = Op_Queue.tail
    Match_Tail = Search Op_Queue for Target_Tail match
    if (Match_Tail)
        Target_Head = Match_Tail.next
        Match_Head = Search Op_Queue for Target_Head match
        if (Match_Head)
            Sequence_Matches = TRUE
            Target_Iter = Target_Tail
            Match_Iter = Match_Tail
            while (Target_Iter && Target_Iter != Target_Head)
                if (Target_Iter does not match Match_Iter)
                    Sequence_Matches = FALSE
                    break
                Target_Iter = Target_Iter.prev
                Match_Iter = Match_Iter.prev
            if (Sequence_Matches)
                Increment iteration count on Match_Head
                Delete elements Target_Head to Target_Tail
```

Figure 2. Intra-Node Compression on MPI Events

a match are determined by iteratively inspecting queue elements from the "target tail" (end of the queue) backwards till a match is found (the "match tail") immediately succeeded by the "target head". Second, the "match head" is determined as the element following the "match tail" that matches the target head. Third, an element-wise comparison is conducted between head and tail of the "target" and the "match". Fourth, upon a complete match, the "match" is merged into the target by incrementing the RSD (or PRSD) counter — or by creating an RSD (or PRSD) upon initial match of two sequences.

For the first step, we impose a maximum window size for this search before entries are flushed (stored without compression). This ensures that long mismatches do not result in quadratic online search overhead.

We use several encoding techniques to represent MPI events. These encodings, which enable inter-node compression as we detail in the next section, are performed at the intra-node level.

Calling Sequence Identification: Identically named MPI calls, such as MPI_Send, may be scattered over various locations in a program. To distinguish the location of MPI events, our tracing framework further records the calling sequence by logging call sites of the calling stack. This call stack creates a unique signature of an MPI call chain. We require them to match when compression is attempted.

Location-independent Encodings: Communication end-points in SPMD programs often differ from one node to another. However, their position relative to the MPI task ID is often constant. Hence, our framework uses relative encodings of communication end-points, *i.e.*, an end-point is denoted as $\pm c$ for a constant c relative to the current MPI task ID. This fosters effective compression of location-specific parameters. Consider the communication pattern in Figure 3 depicting a 2D stencil where both nodes 9 and 10 communicate with relative neighbors -4, -1, +1 and +4.

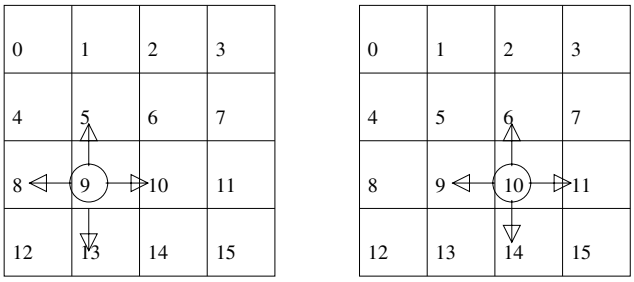


Figure 3. Communication Endpoint Encoding

Request Handles: We cannot simply record the invocation-dependent handles for asynchronous MPI calls. Instead, we record these handles in a buffer and find the matching invocation-dependent pointer in that handle buffer when a completion references the handle. The MPI event then records its handle offset relative to the last element of the buffer. Relative indexing again enables subsequent cross-node compression. We recreate this buffer on-the-fly during message replay and use the offset in the trace to obtain the correct handle pointer.

Certain MPI operations (*e.g.*, MPI_Waitall) allow an array of request handles to be specified. We observed that for some programs the size of these arrays depends on the number of nodes. Since handles are already represented as relative indices into the handle buffer we can effectively compress long arrays of handles using PRSDs. Here, the PRSDs specify (via indices) which handles in the buffer participate in the MPI operation. While originally motivated by handles, we apply this PRSD compression to arbitrary MPI parameters that must be retained in the trace (well beyond handles) and also in the cross-node compression framework. MPI parameters that increase linearly with the number of nodes are, of course, an impediment to application scalability. This is precisely where our tracing tool can provide a "red flag" to developers suggesting to replace point-to-point communication with collectives. Hence, our tool can be used to detect certain scalability problems in an algorithm's communication design.

Event Aggregation: Our approach must preserve event ordering and program structure information. However, non-deterministic repetitions of MPI calls, such as instances of MPI_Waitsome, present a challenge to cross-node compression. Depending on the number of completed asynchronous calls, a loop that terminates upon completion of n corresponding asynchronous calls may result in 1 to n MPI_Waitsome calls within its body. To address this problem early, we squash these MPI call sequences into a single event that records the number of completed asynchronous calls. This count preserves compression capabilities while exploiting MPI-specific semantics. Even during replay, successive MPI_Waitsome calls are aggregated until the recorded number of completions is reached.

3 Inter-/Cross-Node Trace 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 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.

Events and structures (RSD / PRSDs) of nodes are merged when events, parameters, structure and iteration counts match. First, the compressed trace of one child (slave queue) is merged into the local trace of the current node (master queue), then the trace of the other child (slaved) is similarly merged into this new master queue. We use the algorithm depicted in Figure 4 for each merge operation. We identify matching sequences of operations when merging the queues. This identification uses three iterators: the master and slave iterators and the slave head. The master iterator tracks the current operation sequence in the master queue. The slave head tracks the last matched operation sequence in the slave queue. Lastly, we use the slave iterator to identify matching sequences between the master queue and the slave queue.

The algorithm starts all iterators at the beginning of their queues. We increment the slave iterator until we find an operation sequence matching the current master iterator. If a

We actually use a recursive definition of iterators with a start point, depth and a sequence of n pairs of (stride, iterations) for this purpose, which is equivalent to nested PRSDs of the same depth.

```
merge algorithm(master_queue, slave_queue)
    master_iter = master_queue.head
    slave_head = slave_queue.head
    while (master_iter && slave_head)
        slave_iter = slave_head
        while (slave_iter)
            if (slave_iter == master_iter)
                insert operations between slave_head to
                    slave_iter before master_iter
                add slave_iter task participant list to
                    master_iter task participant list
                slave_head = slave_iter.next
                break
            slave_iter = slave_iter.next
        master_iter = master_iter.next
```

Figure 4. Merge Slave/Child into Master/Parent Trace

match is found, we first copy all unmatched operation sequences into the master queue preceding the master iterator. The unmatched sequences are those between the slave head (the last matching sequence in the slave queue) and the slave iterator (the current match in the slave queue). Thus, we maintain the order of operations of the slave queue. We then merge the slave iterator's task participant list with the match's (i.e., master iterator's) list.

Temporal Cross-Node Reordering: The merge algorithm compresses well at lower levels of the reduction tree but encounters problems at higher levels. The difficulties arise from merge disjoint sequences of MPI events in rank order. Consider entries (event;tasks) in master and slave queues $< (A;1),(B;2) >$ and $< (B;3),(A;4) >$. By matching A, the merged queue is $< (B;3),(A;1,4),(B;2) >$ indicating a potential to grow linearly during the merge. However, the temporal ordering between tasks is irrelevant in this example, and another legal queue would be $< (A;1,4),(B;2,3) >$, which provides a constant-size representation. When different tasks participate in the operation sequences, any ordering is legal. We test if the intersection of tasks in the unmatched sequence with those of the matched sequence is empty. If so, the merge algorithm then allows matches to occur one event at a time so that the resulting sequence may differ in the master compared to the original slave. The upper complexity bound of this operation is $O(n^2)$ for n events, but, due to the SPMD regularity of applications, the actual cost is generally constant.

Task ID Compression: In order to capture which subset of nodes participated in some set of events, we encode task IDs as PRSDs similarly to request handles during the merge process. Thus, we concisely represent cross-node similarities, even for stencil codes. Assuming non-wrap-around communication for the 2D stencil in Figure 3, interior nodes 5, 6, 9 and 10 have an identical communication pattern. Any pair of nodes between corners on the boundary as well as any corner nodes also have a unique pattern. Thus, we record nine different patterns for 2D stencils, regardless of the number of nodes. This approach makes cross-node compression feasible and results in a single concise trace file (in some instances of constant size) that is far more efficient than storing per-node trace files for later consolidation.

Reduction over a Radix Tree: We use a binary radix tree internally for the reduction (compression) step. The radix tree representation has several advantages over an arbitrary reduction tree. First, the tree is already balanced, which also balances computational compression cost during cross-node compression. Second, the compression of task IDs as RSDs is naturally facilitated by a radix tree. Any subtree of the radix tree has a constant, uniform distance between task IDs of the nodes in the subtree, which supports a single-RSD representation to describe matching events during task ID compression.

4 Experimental Framework

We gathered experimental results for 1D, 2D and 3D stencil benchmarks, codes from the NAS Parallel Benchmark suite and the Raptor application.

The 1D stencil has a one-dimensional logical space based on a task's MPI rank. Each task communicates with to its two left neighbors and two right neighbors (five-point stencil) during each time step. The communication step consists of sending and receiving from these neighbors. A task proceeds to its next time step only after it completes its sends and receives for the current time step.

The 2D stencil has a two-dimensional logical space in which each task's logical address (communication endpoint) is: $x = rank/dim; y = rank \bmod dim$ for dimension dim. Communication occurs with all eight neighbors (including diagonal neighbors) for a nine-point stencil. Other details are the same as with the 1D stencil.

The 3D stencil has a three-dimensional logical space in which each task's logical address is: $x = rank \bmod dim; y = rank/dim; z = rank/dim^2$. Communication occurs with all 26 neighbors (including diagonal neighbors) for a 27-point stencil. Other details are the same as before.

The NAS Parallel Benchmark (NPB) codes were selected from NPB version 3.2.1 for MPI [12]. We use class C inputs except for DT, where BG/L only had enough memory to allow a selection of class B. Raptor is a framework implementing a modern Godunov method for shock-flow simulations in a C++/Fortran hybrid with optional adaptive mesh refinement (AMR) support [4]. It supports MPI and pthreads parallelization and communicates on a 27-point

stencil *via* asynchronous communication. We use these capabilities in a hydro-dynamics simulation with a constant problem size per node while varying the number of nodes.

We conducted our experiments on a 1024-node BlueGene/L (BG/L) machine [1]. Each node has only 512MB of memory, which restricts application problem sizes. Hence, our traces must only consume small amounts of this memory. We report the task-0 (root node of the reduction tree), minimum, maximum and average memory consumption of the compression subsystem. We also report trace file sizes.

First, we varied the number of processors (nodes) to assess the effects of instrumentation (PMPI wrappers) on trace file sizes and memory usage. The number of processors was chosen as powers of two (for Raptor and NPB codes, except for BT due to input constraints) or n^d processors (for the stencil benchmark) for a d-dimensional stencil with a base of n, *e.g.*, $7^3 = 343$ nodes. For the stencil benchmarks we additionally vary the number of time steps to assess the effect of the number of iterations on trace file sizes.

5 Experimental Results

We conducted three sets of experiments. We assessed the effectiveness of our compression techniques by examining trace file sizes. We determined the overhead of inter-node compression in terms of memory consumption and for the overall time incurred for trace collection and file I/O. For the later, we assessed the cost of writing compressed traces, one per node, to I/O nodes over a Lustre parallel file system, which is the fastest global file system available on our experimental platform. Finally, we verified the correctness (lossless compression) of our approach during replay.

5.1 Trace Sizes & Memory Requirements

Fig. 5 depicts the size of trace files and the memory requirements on a per-node basis on BG/L for the tests described in the previous section.

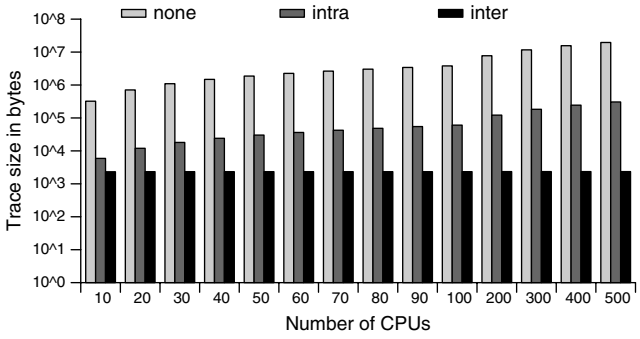
Figures 5(a), 5(c) and 5(e) depict the trace file sizes of the 1D, 2D and 3D stencil codes, respectively, for varying stencil sizes (number of nodes). We show trace sizes on a logarithmic scale for the nodes (a) without compression (none), (b) only with intra-node (task-level) compression and (c) with the additional step of inter-node compression. We observe a significant increase of two orders of magnitude in storage space without compression in the tested node range. Intra-node compression reduces this overhead by two orders of a magnitude, but trace sizes still increase by two orders of magnitude across the node range. Hence, neither approach is scalable with the number of nodes. The fully compressed trace sizes, in contrast, are constant in size independent of the number of nodes, which illustrates that our combined intra- and inter-node compression technique scales well. The resulting trace sizes, 2KB, 4KB and 12KB, for 1D, 2D and 3D stencils, concisely represent MPI events, in contrast to trace size ranges obtained without compression of 0.3-19MB, 0.3-29MB and 2MB-61MB. Increases between stencil sizes reflect the number of distinct patterns required to represent corner nodes, boundary nodes and interior nodes as RSDs.

As BG/L is a memory-constrained architecture with only 512 MB RAM per node, keeping the memory pressure low during on-the-fly compression is as important as the resulting trace file size. Figures 5(b), 5(d) and 5(f) depict the memory usage on a logarithmic scale reflecting the combined intra- and inter-node compression components for the 1D, 2D and 3D stencil benchmarks, respectively, over varying stencil sizes. We report minimum, average, maximum and node-0 (root node) memory usage over all nodes. Within each of these categories, memory usage is constant over different node sizes, which reinforces the claim of scalability of the approach. The average usage decreases as the number of nodes grows, which is a result of increasing height in the reduction tree where more nodes are at lower levels performing less inter-node compression work and, hence, requiring less memory. Besides the average, all other numbers remain constant when the number of nodes grows. The memory requirements at task-0, the root node, are generally close to the maximum memory usage, though, occasionally, a node at level 1 (child of the root) may require insignificantly more memory. We measured a minimum (maximum) memory usage of 1.6KB (6.4KB), 1.6KB (11.4KB) and 1.4KB (26KB) for the 1D, 2D and 3D stencil problems, respectively. This metric includes the merge queues for intra- and inter-node compression but excludes storage of the actual trace, which we reported as trace file sizes.

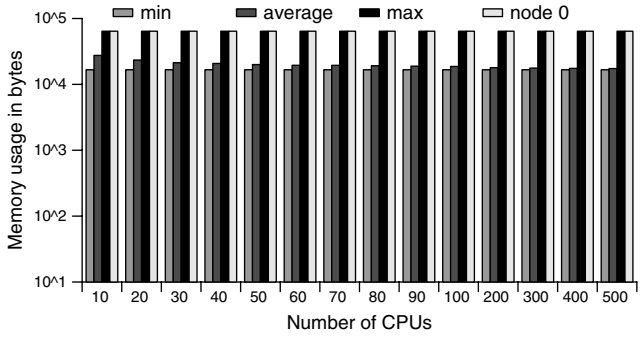
Figure 5(h) depicts the memory usage for Raptor confirming our prior observations with a complex application code. We also see a slight increase in the maximum memory usage of 38MB for 128 nodes to 55MB for 1024. Minor inefficiencies of inter-node compression, which we are currently addressing, cause this increase.

Figure 5(g) depicts the trace file size as we vary the number of time steps (i.e., the iteration bound of the outer-most convergence loop) and hold the number of nodes constant at 125 for the 3D stencil problem. While the uncompressed trace does not scale, both task-level (intra-node) and full compression provide constant-size, scalable results. This confirms that the number of loop iterations has no effect on compression after RSDs and PRSDs are formed, irrespective of inter-node compression. Results for the other benchmarks are equivalent and, therefore, omitted here.

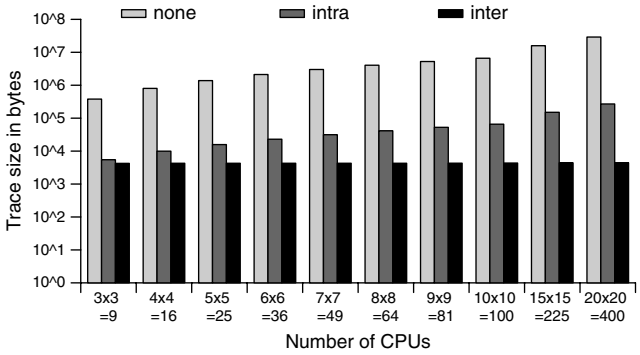
Figure 6 depicts the trace file sizes for the NPB suite on a log-scale. We can distinguish three categories of codes, those that result in near constant-size traces, regardless of



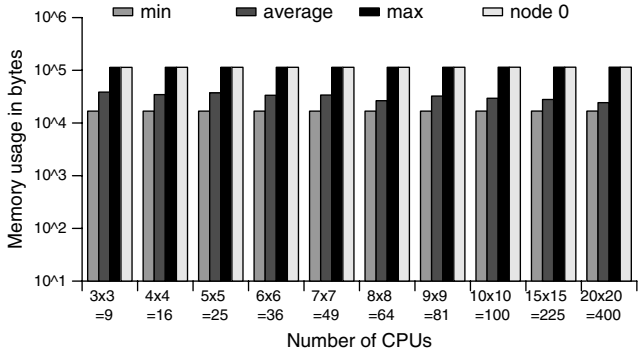
(a) 1D Stencil Trace File, Varied Number of Nodes



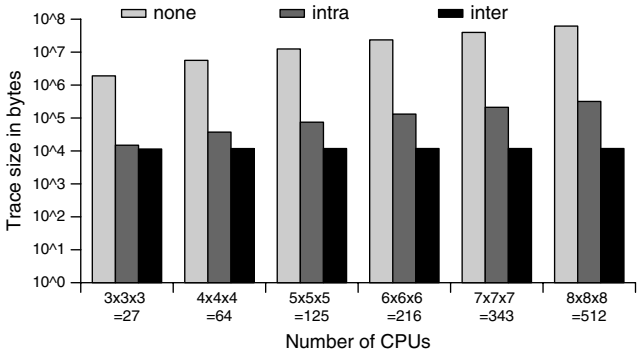
(b) 1D Stencil Memory Usage, Varied Number of Nodes



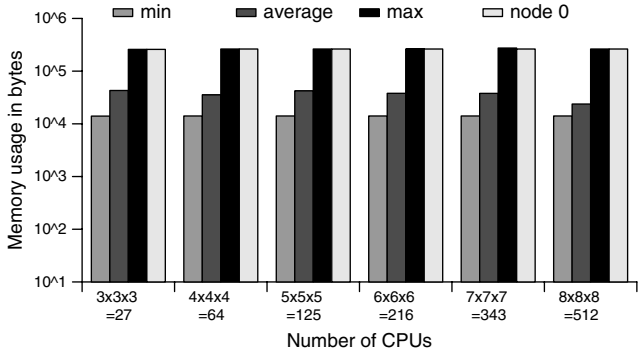
(c) 2D Stencil Trace File, Varied Number of Nodes



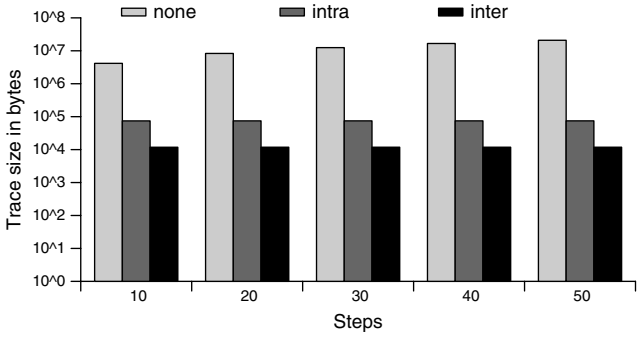
(d) 2D Stencil Memory Usage, Varied Number of Nodes



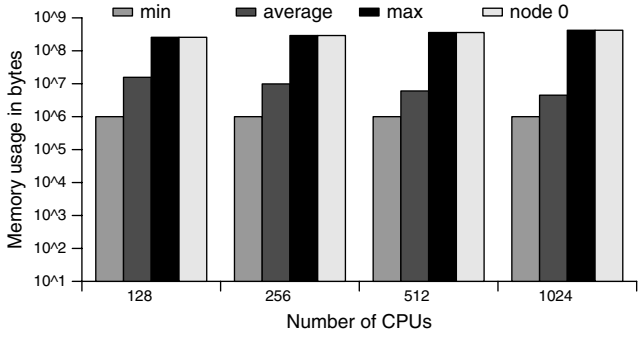
(e) 3D Stencil Trace File, Varied Number of Nodes



(f) 3D Stencil Memory Usage, Varied Number of Nodes



(g) 3D Stencil Trace File, Varyied Time Steps



(h) Raptor Application

Figure 5. Trace File Size and Memory Usage per Node on BlueGene/L

the number of nodes, those with sub-linear scaling of trace size as the node count increases and those that do not scale (yet).

DT, EP and IS (Figures 6(a), 6(b) and 6(c)) fall into the first category (DT only runs on $\geq$ 43 nodes, 32-node results are omitted). Their trace sizes increase exponentially with no compression or with only intra-node compression only. Inter-node compression results in constant trace sizes. These codes have few, very regular communication calls: a pipeline of sends and asynchronous receives along the chains of ranks plus some collective calls.

LU and MG (Figures 6(d) and 6(e)) fall into the second category. We still observe super-linear trace size increases without compression but sub-linear increases at orders of magnitude lower for inter-node compression. Intra-node compression works well for LU and is similar to our benchmark results for MG. We conclude that the main benefit of a size reduction by more than three order of magnitude stems from the intra-node scheme for LU.

Results for the remaining codes, BT, CG and FT, also shown in Figure 6, indicate reductions in trace size with inter-node compression of 2-4 orders relative to no compression and up to one order of magnitude compared to intra-node compression. More specifically, CG requires significantly larger trace sizes without compression due to a large number of point-to-point communications, some of which are asynchronous. FT, on the other hand, benefits more significantly from inter-node compression due to all-to-all collectives that are consolidated across nodes. Both codes show the smallest trace sizes for full compression. Nonetheless, all techniques show exponential increases (at different magnitudes), which indicates that there is room for improvement to obtain near-constant trace sizes.

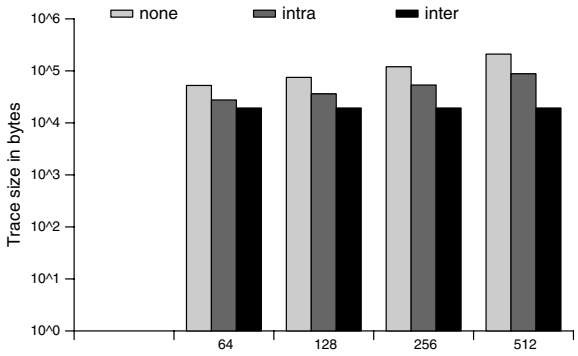
The non-scalability arises from communication patterns in some benchmarks that are not sufficiently abstracted with relative references to end-points. For example, we must modify the encoding for communications along the diagonal of a two-dimensional task layout. Similarly, absolute references, instead of relative indices, sometimes capture the communication pattern, as with a client-server code structure. Thus, we have identified minor shortcomings in our intra-node compression that can prevent inter-node compression. Current work is addressing these shortcomings.

Figures 7(a) and 7(b) depict the memory requirements for inter-node compression for EP and BT, respectively, on a logarithmic scale. For codes whose trace sizes scale, such as EP, the amount of memory used remains constant irrespective of the position of a node in the compression tree. Hence, our technique compresses well without additional memory cost for upper-level nodes in the tree. For non-scaling benchmarks like BT, memory usage is constant at leaf nodes (minimum metric) but increases for larger node counts towards the root (node 0). These two benchmarks are representative of all others, except for MG, which shows constant memory usage like EP, thereby indicating that our scheme is missing an opportunity for constant-size compression (since MG's trace sizes increased sub-linearly). This reconfirms the prior observation that inefficiencies in the intra-node scheme currently restrict inter-node merging. More significantly, we could further reduce costs by offloading the inter-node compression to an external reduction infrastructure. Figure 7(b) indicates a lower average cost, which shows the intermediate nodes in the compression (reduction) tree experience more merge overhead than the leaves (minimum value) but less than the nodes close to the root. This indicates that performing inter-node compression on nodes external to the application nodes and connected using a tree-based overlay network should result in further improvements. In particular on BG/L, dedicated I/O nodes that are automatically allocated together with any program partition can easily be used for this kind of work without requiring additional resources. In this case, the inter-node compression could also occur incrementally as traces are generated, which would allow them to be generated concurrently to the application's computation, thereby further reducing the overhead. MRNet [9] provides a framework for computation offloading to I/O nodes, an area that is the subject of future work.

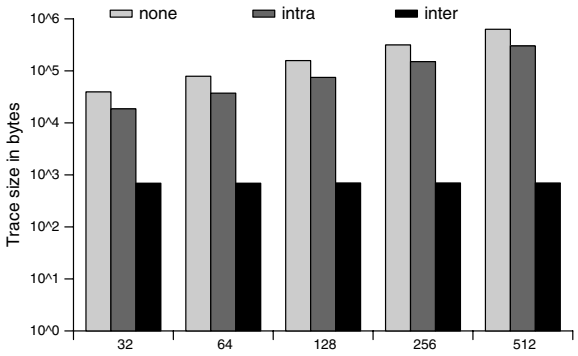
5.2 Inter-Node Merge Overhead

Figures 7(c), 7(d) and 7(e) depict the runtime overhead on a logarithmic scale for EP, LU and BT with no compression (none), with only intra-node compression and with inter-node compression. The first two include the overhead of writing a trace file per node to the parallel file system while inter-node includes the overhead of inter-node compression and that of writing the compressed trace at the root node. These times were measured as the difference between an instrumented run and an uninstrumented run of the respective benchmark. The three benchmarks are representative for the three classes of benchmarks.

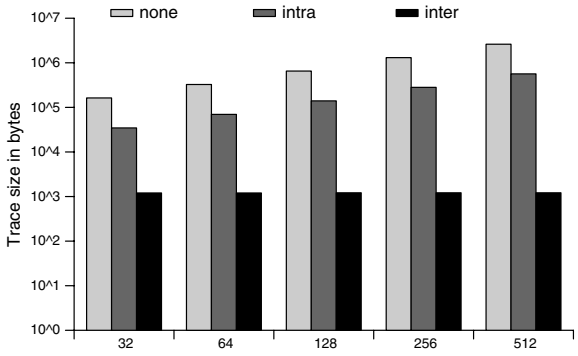
We observe that inter-node compression has the lowest overhead for EP, which represents the class of benchmarks with constant-space compression. This overhead increases slightly with the number nodes, yet a slower rate (and a much smaller absolute overhead) than the other schemes. LU's overhead is nearly the same, irrespective of the compression scheme. We suspect that there is room for improvement for intra- and inter-node compression due to missed opportunities, as discussed for benchmarks with sub-linear compression. BT shows the lowest overhead without compression. Not surprisingly, inter-node compression is most costly since BT belongs to the class of benchmarks with super-linear compression space requirements.



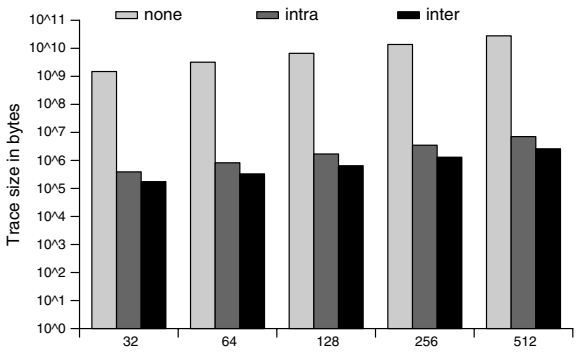
(a) DT Trace File, Varied Number of Nodes



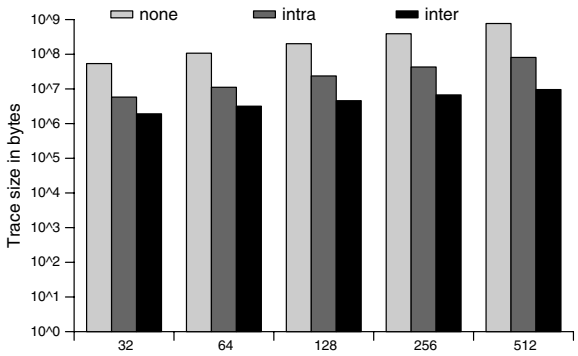
(b) EP Trace File, Varied Number of Nodes



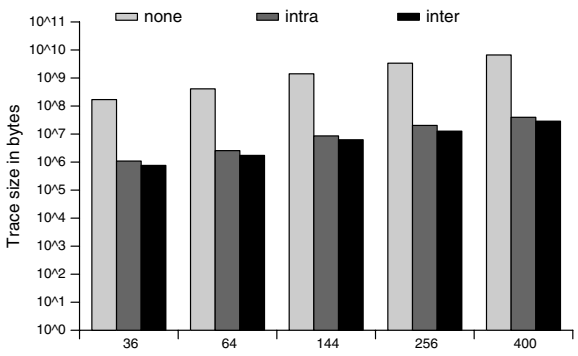
(c) IS Trace File, Varied Number of Nodes



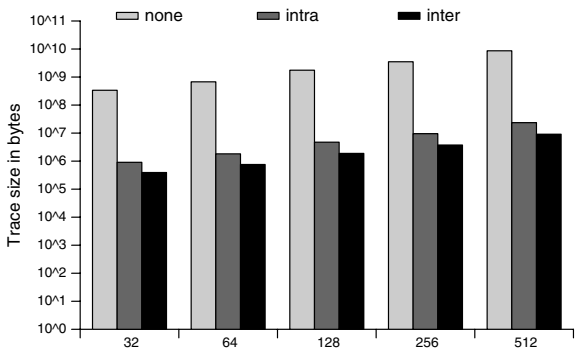
(d) LU Trace File, Varied Number of Nodes



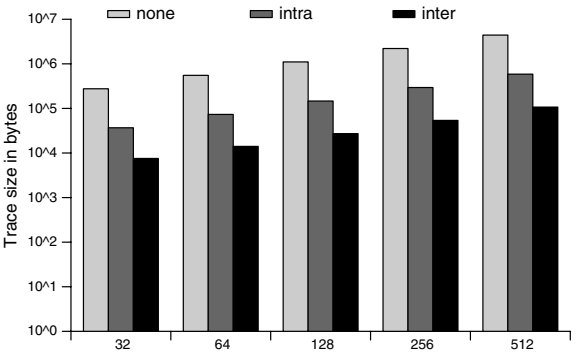
(e) MG Trace File, Varied Number of Nodes



(f) BT Trace File, Varied Number of Nodes

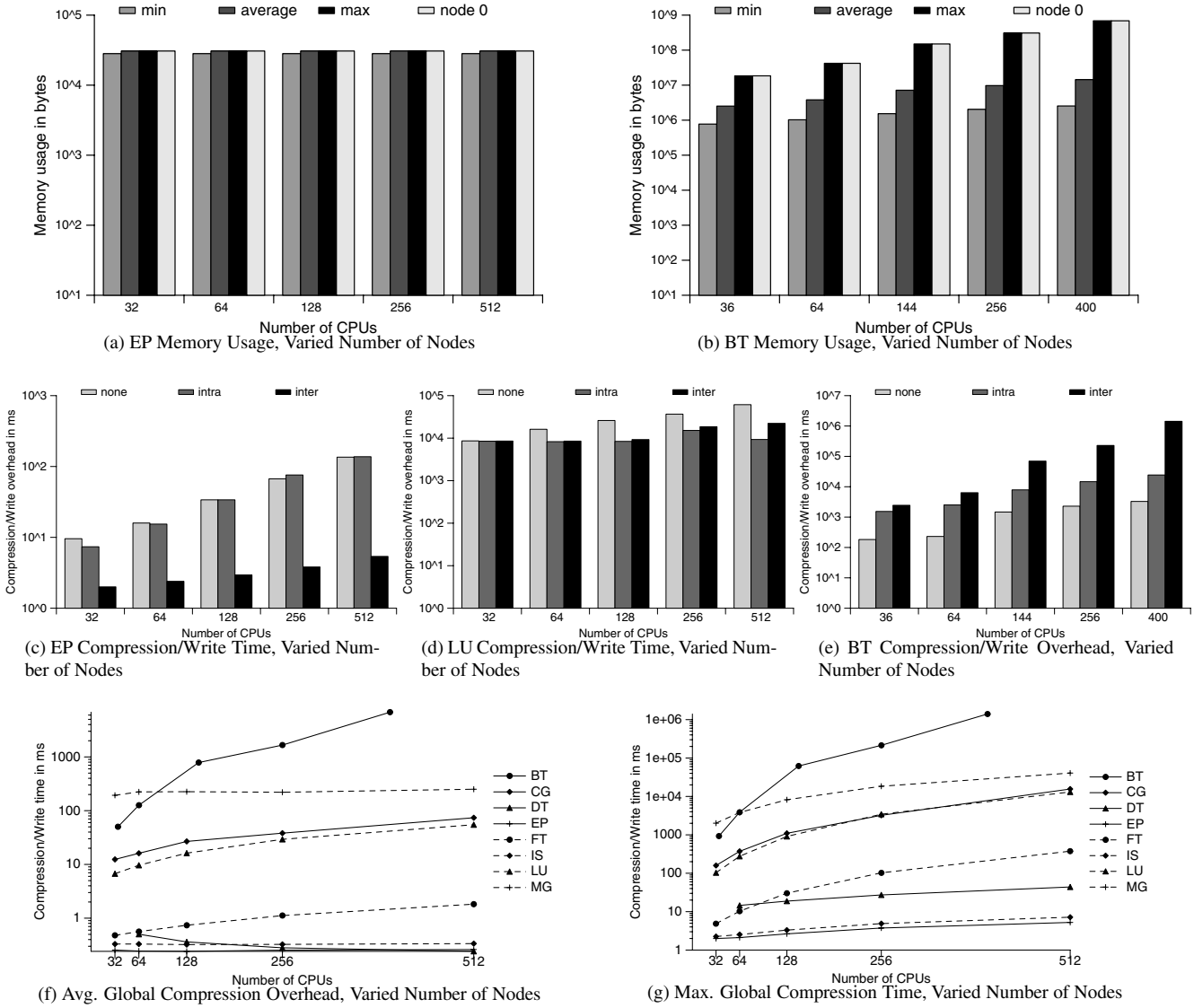


(g) CG Trace File, Varied Number of Nodes



(h) FT Trace File, Varied Number of Nodes

Figure 6. NPB Trace File Size per Node on BlueGene/L



(a) EP Memory Usage, Varied Number of Nodes

(b) BT Memory Usage, Varied Number of Nodes

(c) EP Compression/Write Time, Varied Number of Nodes

(d) LU Compression/Write Time, Varied Number of Nodes

(e) BT Compression/Write Overhead, Varied Number of Nodes

(f) Avg. Global Compression Overhead, Varied Number of Nodes

(g) Max. Global Compression Time, Varied Number of Nodes

Figure 7. NPB Memory Usage and Compression Time per Node on BlueGene/L

These benchmarks are representative for their respective classes, except for MG, which shows results comparable to those of BT.

Figures 7(f) and 7(g) show the average and maximum inter-node compression time measured inside of MPI_Finalize. These results indicate a wide spread of overhead, which does not necessarily correlate with the compression rate achieved. Consider Figure 7(g). Even though FT behaves super-linear while MG is sub-linear in compression, FT's compression time is smaller. This shows that the timing overhead is more closely related to the amount of MPI calls issued by the respective application than scalability of compression. We have identified possible optimizations in the inter-node merge algorithm to further reduce this overhead. Also, a discrepancy between average and maximum overheads (Figures 7(f) and 7(g)), indicates widely diverging loads between mid-level and top-level nodes during inter-node compression in the tree. Thus, other potential improvements involve off-loading inter-node compression to I/O nodes, as discussed previously.

5.3 Verification of Replay Correctness

We conducted additional experiments to verify the correctness of our 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. 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. We are currently pursuing these directions, among others to improve communication performance in a systematic, yet experimental manner on BG/L and to support procurement of large scale machines.

6 Related Work

RSDs have been used to describe data references in a loop [5]. PRSDs originally targeted on-the-fly memory trace compression [7]. While that work introduced the general concepts and an algorithm for compressing regular data references, our work uses an entirely different algorithm. Our task, compressing events composed of MPI call IDs and their parameters, is considerably more complex. We also use semantic-specific encodings, such as for MPI_Waitsome, which are unique to the trace domain. Further, our work is the first to utilize the structural information retained during compression, *i.e.*, our replay mechanism relies on this unique compression property. The approach is superior to run-length encoding and sliding window compression [13] in that it allows recursive compression while preserving loop structures in the compressed format.

The mpiP tool consists of a lightweight profiling library for MPI applications that collects statistical information about MPI functions[11]. It reports aggregate metrics. Hence, structural information and event ordering are not preserved. There are many other tools that report aggregate information, often based on the profiling layer of MPI, as is the case with mpiP. None of these tools are suitable for lossless tracing and later replay.

Vampir is a commercial tool set including a trace generator and a display engine to visualize MPI communication [2]. However, traces are generated in local files such that total trace file size increases linearly with both the number of MPI calls made and the number of tasks. This limits the applicability as scalability is compromised.

Paraver and Dimemas is an MPI tracing tool set from the University of Barcelona [8]. Paraver provides functionality similar to Vampir; its trace generator has similar limitations. Dimemas is a discrete-event-based network performance simulator that uses Paraver traces as input. It is the most similar existing tool to our replay mechanism. However, it does not support replaying traces on actual systems. Instead, it uses a processor ratio and network latency and bandwidth parameters to simulate the application's MPI usage on a theoretical alternative system. Our tool set provides scalable MPI tracing; the traces could be used in a discrete event simulator like Dimemas as well as with our replay mechanism.

MRNet is a software overlay network that provides efficient multicast and reduction communications for parallel and distributed tools and systems [9]. 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 initial 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. We plan to use MRNet in a future version of our tool set.

The Open Trace Format (OTF) is targeted at scalable tracing, yet without any advanced (domain-specific) compression scheme [6]. In contrast to our work, it uses regular zlib compression on blocks of data, which loses structure and limits analysis on the compressed format. They also do not support cross-node compression schemes. Hence, the complexity of aggregate trace size over n processors is O(n). However, they have the ability to produce multiple streams and, hence, store and load them in parallel with user-defined granularity.

A characterization of MPI communication patterns for the NAS parallel benchmarks has determined that communication end-points are, if not static, almost exclusively persistent and hardly ever dynamic [10]. Here, persistent denotes a set of end-points that, once determined dynamically, does not change anymore. This is consistent with our findings and explains why our compression techniques are scalable within the domain of SPMD programs.

7 Conclusion

One of the central problems in petascale computing is posed by the requirement for communication to scale to hundreds, if not thousands of nodes. However, communication patterns of large-scale scientific applications are often too complex to analyze at the source-code level. While tools exist to analyze aggregate metrics statistically in a scalable manner, temporal ordering and structural information are generally lost in such an approach. Other tools employ traces, which grow significantly in size as the problem size (number of iterations to convergence) increases and become harder to commit to global file systems as the number of nodes increases.

In contrast to prior work, we present a trace-driven approach to analyze MPI communication that scales by extracting full communication traces orders of magnitude smaller or even of near-constant size regardless of the number of nodes while preserving structural and temporal-order information of events. We employ representations of regular section descriptors, power-sets of them and a multitude of *relative* encoding techniques to enable compact representations of MPI event sequences. A first intra-node compression is followed by inter-node compression over a reduction tree to result in a single trace file that fits into a fraction of the core memory of a node. Experimental results on BlueGene/L confirm our claim of concise, if not near constant size, representation for benchmarks and a full-sized application. We assessed the correctness of our approach by verifying the temporal orderings and aggregate counts of MPI events using our unique replay mechanism. This replay mechanism may aid performance tuning of MPI communication and facilitate projections of network requirements for future large-scale procurements.

To the best of our knowledge, our contributions of orders of magnitude smaller and sometimes constant-size representation of MPI traces in a scalable manner combined with deterministic replay are without precedence.

References

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

[2] H. Brunst, H.-C. Hoppe, W. E. Nagel, and M. Winkler. Performance optimization for large scale computing: The scalable vampir approach. In *International Conference on Computational Science (2)*, pages 751–760, 2001.

[3] L. DeRose, K. Ekanadham, J. K. Hollingsworth, and S. Sbaraglia. SIGMA: A simulator infrastructure to guide memory analysis. In *Supercomputing*, Nov. 2002.

[4] J. Greenough, A. Kuhl, L. Howell, A. Shestakov, U. Creach, A. Miller, E. Tarwater, A. Cook, and B. Cabot. Raptor – software and applications on bluegene/l. BG/L workshop paper 22, Lawrence Livermore National Lab, Oct. 2003. http://www.llnl.gov/asci/platforms/bluegene/papers/22greenough.pdf.

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

[6] 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.

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

[8] 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, Apr. 1995.

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

[10] S. Shao, A. Jones, and R. Melhem. A compiler-based communication analysis approach for multiprocessor systems. In *International Parallel and Distributed Processing Symposium*, 2006.

[11] 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.

[12] F. Wong, R. Martin, R. Arpaci-Dusseau, and D. Culler. Architectural requirements and scalability of the NAS parallel benchmarks. In *Supercomputing*, 1999.

[13] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.