# ScalaTrace: Scalable Compression and Replay of Communication Traces for High Performance Computing ⋆

Michael Noeth [a] Prasun Ratn [a] Frank Mueller [a,*] Martin Schulz [b]
Bronis R. de Supinski [b]

[a] *North Carolina State University Department of Computer Science Raleigh, NC 27695-7534*

[b] *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 long execution times. While many tools to study this behavior have been developed, these approaches either aggregate information in a lossy way through high-level statistics or produce huge trace files that are hard to handle.

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 that are capable of extracting an application's communication structure. We further present a replay mechanism for the traces generated by our approach and discuss 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 precedent.

*Key words:* High-Performance Computing, Scalability, Communication Tracing
*PACS:* 07.05.Bx

⋆ An earlier version of this paper appeared at IPDPS'07 [20]. This journal version extends the earlier paper by novel domain-specific intra- and inter-node compression techniques, a completely redesigned inter-node merge algorithm, novel results with a larger class of codes resulting in near-constant trace sizes, a study to identify the timestep loop and extended related work. 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 Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344. (LLNL-JRNL-403992)

* Corresponding author.
*Email addresses:* prasun.r@ncsu.edu (Prasun Ratn), mueller@cs.ncsu.edu (Frank Mueller), schulzm@llnl.gov (Martin Schulz), bronis@llnl.gov (Bronis R. de Supinski).

## 1 Introduction and Overview

Petascale systems will have hundreds of thousands of processors, and their effective use will require efficient interprocess communication through complex network topologies. To optimize application behavior in such environments, we require tools that can collect and analyze the communication behavior of complex applications and large-scale runs. This, however, is a non-trivial problem, and a wide array of analysis tools has been developed, both by academia and industry, to aid this process. They can generally be divided into two main classes: *tracing* tools, capable of capturing and recording all messaging events, albeit at the cost of high storage requirements; and *profiling* tools, designed to provide low-overhead performance summaries trading off storage space for accuracy.

One of the best known examples of a tracing tool for MPI communication is Vampir [6], a tool set consisting of 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. In contrast to this, tools like mpiP [27], a well-known MPI profiler, gather user-configurable aggregate metrics for statistical analysis. The output results of such profilers are typically constrained in size by the number of unique call sites of MPI events, which is independent of the number of nodes. However, profilers do not preserve the structure and temporal ordering of events, which limits their use to high-level analysis.

In contrast to this prior work, we propose a novel approach that not only bridges the worlds of tracing and profiling by combining the advantages from both, but also provides a highly scalable solution. We have designed ScalaTrace, a tracing framework that extracts full communication traces that are orders of magnitude smaller than traditional traces, potentially even constant size, regardless of the number of nodes while preserving structural information and temporal event order.

Figure 1 shows a high-level overview of our ScalaTrace framework. Like most MPI tools, we rely on the the MPI profiling layer (PMPI) to intercept MPI calls during application execution and use it to install wrappers for each MPI call. Within each node, these wrappers trace which MPI functions were called along with all function parameters (except for the message payload) and compress this task-level information on-the-fly into local operation queues within each task. At the termination of the application, we then use a cross-node framework to perform inter-node compression and to obtain a single trace file in the form of a comprehensive operation queue that preserves the complete message trace of the application without loss of information. Further, the final compressed data implicitly contains the structure of the application's communication behavior enabling efficient application replay for further analysis or even a direct inspection of the application's communication structure.

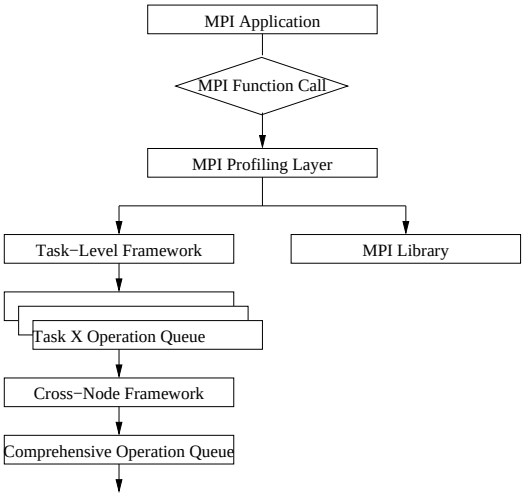



Fig. 1. Interaction of Components

We assess the effectiveness of our framework through experiments using the MPI NAS benchmarks as well as large, scientific applications on BlueGene/L. Our results confirm the scalability of our on-the-fly MPI trace compression by yielding orders of magnitude smaller or in several cases even near constant size traces for processor and problem scaling. 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 in the performance tuning of MPI communication and facilitates 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 precedent.

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

ScalaTrace uses a bi-level runtime compression approach: after creating trace records for each observed MPI event, we first perform node local (intra-node) compression on the fly, followed by a global (inter-node) compression step. Both steps must provide lossless, yet space-efficient compression requiring us to extract and to store the underlying structure of the communication. Together, this enables us to represent repetitive MPI events in loops with identical parameters in constant size.

We achieve this intra-node compression by extending the idea of describing single loops using regular section descriptors (RSDs) [14] to express MPI events nested

```
Compress_Queue(Queue Op_Queue)
    Target_Tail = Op_Queue.tail
    do
        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
    while (No_Sequence_Match && distance(Target_Tail, Match_Tail) < window)
```

Fig. 2. Intra-Node Compression on MPI Events

inside a loop in constant size. We represent an RSD as a tuple <*length, event*$_1$, ..., *event*$_n$> where the length indicates the loop trip count and events are MPI trace events (MPI calls and their parameters). Similarly, we rely on power-RSDs (PRSDs) [17] to specify recursive RSDs nested in multiple loops. PRSDs generalize RSD representations by allowing events to be RSDs (or PRSDs) themselves, effectively nested lists of events for nested loops and sequences of lists for sequences of loops. 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 [9]. Our algorithm uses two sequences, the "target" and the "match" sequence, each with its own head and tail pointer. The former describes the already detected sequence sets that have been converted into PRSDs, while the latter is formed by newly acquired trace records. The matching process proceeds in four steps, as depicted in Figure 2.

First, we determine the head and tail of the match sequence by traversing the local event queue backwards from the end of the queue, which forms the "target tail",

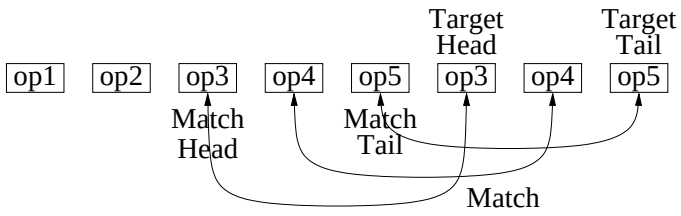



Fig. 3. Intra-Node Compression Scenario

until we find a match (the “match tail”) immediately followed by the “target head”. Second, we identify the element following the “match tail” that matches the “target head” and note it as the “match head”. Third, we conduct an element-wise comparison between head and tail of the “target” and the “match”. Fourth, upon a complete match, we merge the “match” into the “target” by incrementing the RSD (or PRSD) counter — or by creating an RSD (or PRSD) upon initial match of two sequences. For practical reasons, we impose a maximum window size for this search in the first step before entries are flushed (stored without compression). This ensures that long mismatches do not result in quadratic online search overhead. Our experiments show that these restrictions do not impact our ability to compress traces in most cases. We used a windows size of 500 in our experiments.

Figure 3 depicts an example for online trace compression. A sequence of MPI operations are successively pulled into the window that the algorithm operates on. Let the target tail be the last operation, op5, in the window. The match tail then is the corresponding first occurrence of op5, followed by the target head and preceded by the merge head (two operations back). Upon traversing each element between tail and head to ensure a pairwise match, we either extend an existing RSD in its length by one, or we create a new RSD for the matching subsequence of size two. In the latter case, we obtain an $RSD1 :< 2, op3, op4, op5 >$ in this example.

Matches have to be adjacent at a loop / PRSD level for compression to occur. Non-adjacent matches with regular patterns interspersed at a different (but constant) rate are compressed through multi-level PRSD formation. Irregular interspersed patterns, in contrast, will hinder effective compression.

Our compression scheme requires exact matches between sequences of operations both within a node and across nodes. To enable such matches in as many cases as possible, we use series of encoding techniques, which are described individually in the following paragraphs. They are all applied at the intra-node level, but prepare traces for the inter-node compression.

**Calling Sequence Identification:** Identically named MPI calls, such as MPI_Send, may be scattered over various locations in a program. To distinguish events from different locations, it is therefore not sufficient to just record the MPI event type itself. Instead, we must capture its calling context. We achieve this by recording the calling sequence that leads to the MPI event and gather this information from the stack trace during the MPI event. We represent each location as a unique signature

of the stack trace, which must match when compression is attempted.

A stack signature may consist of a number of backtrace addresses of the program counters (return addresses), one for each stack frame. For deeply nested calling constructs, a comparison of two backtraces can become costly. To speed up this process, we also store a hash of all backtraces computed as the exclusive or (XOR) of all backtrace addresses. A match of the hash values for two backtraces is a necessary condition for a matching backtrace. Hence, we first perform an XOR comparison before a pairwise match for each frame is attempted, eliminating a large fraction of unnecessary and costly stack trace comparisons.

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

**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 4 depicting a 2D stencil where the two interior nodes 9 and 10 show the same exact same pattern by communicating with their relative neighbors -4, -1, +1 and +4. Similarly, all border and corner nodes communicate with the same relative neighbors. Therefore, each group of nodes can be compressed to a constant size trace.

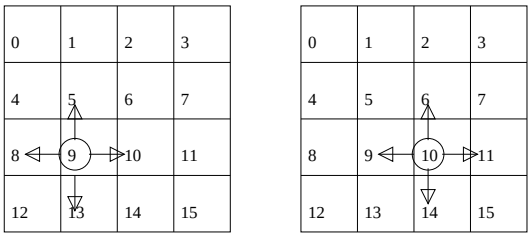



Fig. 4. Communication End-point Encoding

Receives using wildcard end-points (e.g., MPI_ANY_SOURCE) are handled as a special case. Such end-points are stored explicitly rather than as relative offsets. Our evaluation shows that significant compression benefits can be obtained by this domain-specific optimization. Another improvement omits tags from point-to-point

records as they were often redundant and adversely affected compression. In effect, such cases are handled equivalently to that of a wildcard value (MPI_ANY_TAG). This occasionally resulted in signifi cantly improved intra-node compression. Yet, the scheme is invalid if tags are utilized to distinguish end-points. We have designed a method for automatic detection of the relevance of tags to record them only when required. To determine if tags are semantically relevant, both relative and absolute addressing are attempted. If one of the methods results in a match between end-points of multiple nodes, then it is chosen over the other.

Our relative end-point encoding fails to capture rare cases of absolute addressing of end-points, *e.g.*, to communicate information back to the root node (task 0) or to a coordination node within a subgroup communicator. Solutions outlined above (for MPI_ANY_SOURCE and MPI_ANY_TAG encodings) address both of these cases. We are currently exploring implementation issues for these solutions as we must integrate them into the inter-node merge algorithm (see next section).

**Request Handles:** In MPI, handles representing request objects for outstanding asynchronous communication are managed by the MPI library. Often, they are represented by opaque pointers to internal data structures and, hence, do not exhibit any obvious repetitive patterns. We therefore cannot simply record the invocation-dependent handles for asynchronous MPI calls. Instead, we record these handles in a buffer and fi nd 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 specifi ed. 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. [1] 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.

To illustrate our method for retaining handle information, consider the example in Figure 5. In a sequence of three asynchronous (non-blocking) communication calls,

[1] We use a recursive definition of iterators with a start point, depth and a sequence of $n$ pairs of (stride, iterations), which is equivalent to nested PRSDs of the same depth.

three handles are used and consequently recorded in the handle buffer. The handle pointer is set to H3 at this point. Next, the first handle is used in another MPI call to inquire about the completion of the earlier call. This is abstracted as a reference to the handle recorded in the buffer two entries prior to the current handle pointer. Hence, instead of non-portable pointers to dynamically allocated handle objects, we record portable handle indices relative to the current handle pointer.


Fig. 5. Sample Handle Buffer with Relative Indexing

**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.

**Dealing with Inherent Application Load Imbalance:** Application codes that perform intrinsic load balancing actually pose a challenge to ScalaTrace. Due to the balancing act, the data volume exchanged between nodes dynamically increases or decreases between successive timesteps depending on the amount of data cells handled locally. Hence, communication calls show different data volumes, which breaks the regularity of communication calls and results in potentially poor intra-node compression.

In one application, this behavior was observed for MPI_Alltoallv(). Yet, while individual message payloads varied, the collective payload over all nodes remained constant, which is not uncommon. This opens up an opportunity for improved trace compression. Considering that computation time is either ignored or statistically aggregated [22], we could record the average per-node payload, which is constant again. This ensures perfect compression. Traces further retain the integrity to be suitable for later replay (see Section 5.4). If we record extreme values (minimum/maximum) and associated node information as well, then outliers can still be detected. This ensures that such traces retain sufficient information for communication analysis and tuning.

## 3 Inter-/Cross-Node Trace Compression

After the local compression step, we combine the trace records stored in each node's local memory to a global trace. We perform this operation upon application completion within the PMPI wrapper for MPI_Finalize. To guarantee scalability, we employ cross-node compression step-wise and in a bottom-up fashion over a binary tree. In contrast to most traditional approaches this allows us to avoid the creation of local trace files, which would result in linearly increasing disk space requirements and 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.

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 (slave) is similarly merged into this new master queue. A first-generation algorithm described in prior work [20] traversed both master and slave queues. If a subsequence of events between the queues matched, the slave events were combined with the master events by merging participant lists (node IDs of master and slave). Intermediate non-matching slave events (which had causal dependence on the matching events) were simply inserted in place in the master to maintain causal ordering (see below). The process of determining causal dependence involved the calculation of the intersection of the task participant set in the unmatched sequence with that in the matched sequence. This process required a linear scan of the slave queue starting from the head.

Based on the experience with this algorithm and to avoid such a worst case complexity, we developed a second-generation algorithm, which is depicted in Figure 6 for each merge operation. As one of the key differences, we no longer need to scan the slave queue to determine the set of events in the unmatched sequence without causal dependence on the matched set. This is facilitated by the new causal ordering preservation strategy, which is explained in more detail below.

The algorithm identifies matching sequences of operations when merging the queues. This identification uses master and slave iterators, which indicate the beginning of matching subsequences 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 event subsequence matching the current master iterator. If a match is found, we merge the slave iterator's task participants with those of the master iterator. In case of a perfect match, this constitutes the *merge_nodes* operation. If the subsequence is preceded with non-matches, only causally dependent events are promoted to the master queue at this time (as explained below). Upon termination of both loops, all unmatched (and causally independent) operations are appended to the master queue. Thus, we maintain the order of operations of the

```
merge algorithm(master_q, slave_q)
    master_iter = master_q.head
    slave_head = slave_q.head
    while (master_iter) // new: loop only over master_q
        slave_iter = slave_head
        while (slave_iter) // loop over slave_q
            if (match_subsequence(slave_iter, master_iter)) // new: partial match
                // new: build dependence chain, walk dependence graph from slave_iter
                yank_list = dfs(master, master_iter, slave, slave_iter)
                yank(yank_list)
                merge_nodes(master_iter, slave_iter)
                break
            slave_iter = slave_iter.next
        master_iter = master_iter.next
    // move any remaining (independent) events from slave to master
    if (¬ master_iter && slave_head)
        insert(master_q.head, master_q.tail, slave_q.head, slave_q.tail)
```

Fig. 6. 2nd Generation Merge of Slave/Child into Master/Parent Trace

slave queue with regard to their causal dependencies.

Our first-generation algorithm required exact matches of all parameters during the merge procedure. In the second-generation, we relaxed these constraints to allow mismatches for selected parameters (*e.g.*, source/destination) complemented by a separate ordered list of (value, ranklist) pairs to explicitly record these (generally rare) mismatches. Since these ranklists are stored as PRSDs in compressed format, this results in a constant-size representation for regular patterns of end-points, otherwise it grows sub-linearly in size. We observed the most significant improvements from this optimization compared to other enhancements over our first-generation approach.

**Causal Cross-Node Reordering:** The merge algorithm compresses well at lower levels of the reduction tree but encounters problems at higher levels. The difficulties arise when disjoint sequences of MPI events are encountered 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 ordering of the operations in each sequence is irrelevant in this example, since they occurred on different nodes: when disjoint tasks participate in event sequences, any ordering is legal. Hence, another queue with the same semantic information is $< (A;1,4), (B;2,3) >$, which now provides a constant-size representation.

In addition, in our first-generation algorithm, we considered all possible dependencies. A dependence is said to exist if two events have at least one common participant. To this extent, we intersected the current event's participant list with

the participant list of all unmatched events in the slave queue prior to the current event, one at a time. If the intersection of tasks in the unmatched sequence with those of the matched sequence is empty, then no causal order exists.

Our second-generation algorithm maintains a dependence graph during the entire merge algorithm. At a leaf node of the reduction tree, the dependence graph is simply a linked list (directed backwards in the temporal ordering of events) as the current node is a participant for each event. When a slave queue is received at non-leaf nodes from a child, the dependence graph is reconstructed. If a subsequence of matching events is encountered, any preceding non-matches are inspected for dependencies on the current slave event by performing a depth-first search (dfs) over the dependence subgraph originating from the current slave event (see Figure 6). This is an improvement over the first-generation algorithm in that only dependencies reachable from the current event are considered. Any dependent event is added to a *yank list* during the traversal, which eventually contains those events that casually depend on the participants of the current slave event (matched with a master event). Events in the yank list are inserted prior to the current event in the master queue through a yank routine, which ensures that the causal order is preserved.

Matching events between master and slave queues are then merged. During this merge process, participant lists of corresponding events are combined, and existing dependence chains are updated to reflect their promotion to the master queue. These dependence updates can morph the dependence graph into a forest structure, *i.e.*, multiple root nodes (with different participant lists) may point to joint or disjoint dependence chains of different participant subsets.

The upper complexity bound of the overall merge operation is $O(n^2)$ for $n$ events in each queue. Yet, the actual cost is generally constant due to typical regularity of SPMD applications.

Figure 7 depicts four steps of our improved merge algorithm for a master queue (top) and a slave queue (bottom). We denote the sequence of events by forward edges and dependence chains by backward edges. Merging events A for nodes 1 and 2 in step (a) results in a new compound node (A,1,2). The merge procedure further results in combining the two dependence chains in a common sink represented by this merged node. The next match occurs for event B in step (b), which results in a merged node (B,1,2) in (c) prepended by the dependence chains of the slave originating from B. The dependence chain is captured in the yank list and then yanked into the master prior to node B. After a match for event C, a compound node of (C,1,2) is formed in (d). The dependence chains from B and C now unify nodes 1 and 2 although only node 2 participates in communication in D and E.

**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

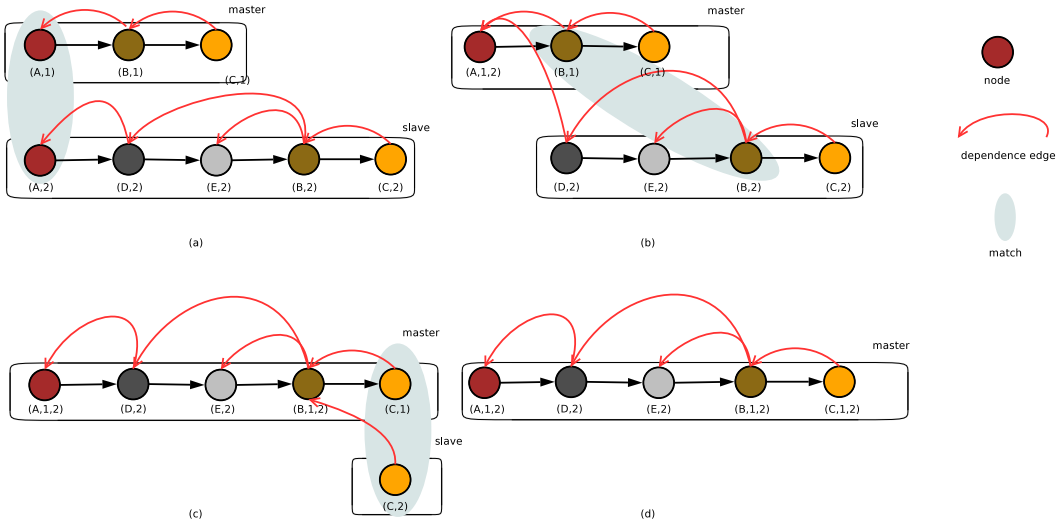



Fig. 7. Inter-Node Merge Scenario

for stencil codes. Assuming non-wrap-around communication for the 2D stencil in Figure 4, interior nodes 5, 6, 9 and 10 have an identical communication pattern. Any boundary as well as corner node also has a unique pattern. Thus, we record nine different patterns for this five-point stencil, 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 (merge) step. The radix tree representation has several advantages over an arbitrary reduction tree. First, the tree is already balanced, which also balances computational merge 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.

Figure 8 depicts a radix tree for 15 nodes. Merging nodes across different levels naturally results in regular patterns suitable for concise representation as RSDs. For instance, nodes 7 and 11 form an RSD of length two with stride 4 starting at node 7 ($< 2, 4, 7 >$). Combining this RSD with node 3 extends it to $< 3, 4, 3 >$. At node 1, the RSDs of the children, namely $< 3, 4, 3 >$ and $< 3, 4, 4 >$ are combined, and by merging them with node 1, we yield $< 7, 1, 2 >$. This illustrates the conciseness of end-points resulting naturally from a radix-tree representation.

**Options for Out-of-Band Compression:** Our second-generation merge algorithm is an improvement but it still impacts overall trace compression by deferring the inter-node merge to program termination (MPI_Finalize). The lower bound on algorithmic complexity for this operation over a reduction tree remains quadratic in

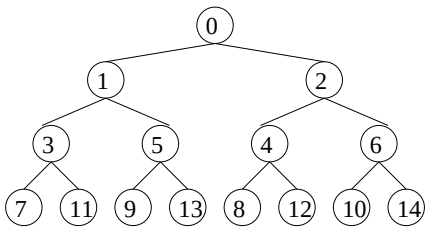



Fig. 8. Sample Radix Tree and Task ID Regularities

the number of events. Thus, the post-computational merging of events, which performs reasonably in our experiments, has limited scalability in the extreme. Alternatively, we could perform inter-node merging in the background on a separate set of nodes to improve scalability. While this is costly on typical cluster architectures, many modern scalable systems provide such resources in the form of I/O nodes. For example, BG/L systems dedicate an I/O node to a set of compute nodes. These I/O nodes can be utilized for computational background work [23]. This alternative would require merge operations that work asynchronously from the creation of the tracing information inside the ScalaTrace MPI wrapper routines. Thus, we must redesign both intra-node compression and inter-node merge algorithms to work incrementally and on-the-fly as new MPI events are encountered in order to support it. Such a radical change is beyond the scope of this paper.

## 4 Experimental Framework

We gathered experimental results for 1D, 2D and 3D stencil codes, a recursion benchmark, the codes from the NAS Parallel Benchmark suite, as well as two scientifi c applications, Raptor and UMT2k.

The 1D stencil has a one-dimensional logical space based on a task's MPI rank. Each task communicates with its two left neighbors and two right neighbors (fi vepoint 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 of size $dim * dim$ in which each task's logical address (communication endpoint) is: $x = rank \bmod dim; y = rank/dim$ 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 of size $dim * dim * dim$ in which each task's logical address is: $x = rank \bmod dim; y = (rank/dim) \bmod dim; z = rank/dim^2$. Communication occurs with all 26 neighbors (including diagonal neighbors) for a 27-point stencil. Other details are the same as before.

The recursion benchmark is a modified version of the 3D stencil benchmark. Here, the timestep loop is defined as a recursive function instead of an iterative loop, as the original 3D stencil benchmark.

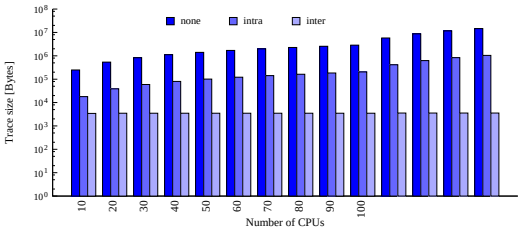
The NAS Parallel Benchmark (NPB) codes were selected from NPB version 3.2.1 for MPI [29] class C. Raptor is a framework implementing a modern Godunov method for shock-flow simulations in a C++/Fortran hybrid with optional adaptive mesh refinement (AMR) support [13]. It supports MPI and pthreads parallelization 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. UMT2k is an unstructured mesh transport code that solves the first-order form of the steady-state Boltzmann transport equation [2].

We conducted our experiments on a 2048-node BlueGene/L (BG/L) machine [3]. Each node has only 1GB 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.

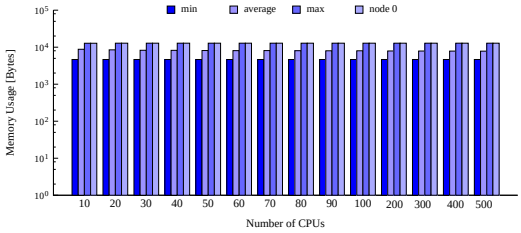
We varied the number of processors (nodes) to assess the effects of instrumentation (PMPI wrappers) on trace file sizes and memory usage. The number of processors was chosen as powers of two (for Raptor, UMT2k and NPB codes, except for BT and with the omission of 32 and 64 nodes for DT, both due to input 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

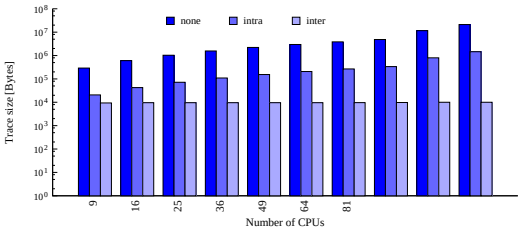
We conducted three sets of experiments. We assessed the effectiveness of our compression techniques by examining trace file sizes. We determined the overhead of inter-node compression in terms of memory consumption and for the overall time incurred for trace collection and file I/O. For the latter, we assessed the cost of writing compressed traces, one per node, to I/O nodes over GPFS. On our BG/L system, 16 compute nodes share one I/O node for a total of 128 I/O nodes. Finally, we verified the correctness (lossless compression) of our approach during replay.



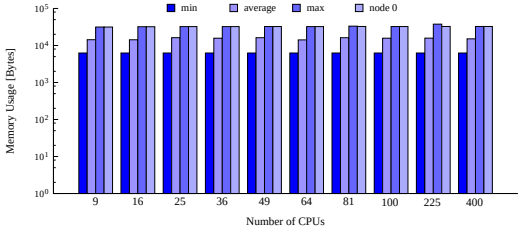

(a) 1D Stencil Trace File, Varied # Nodes



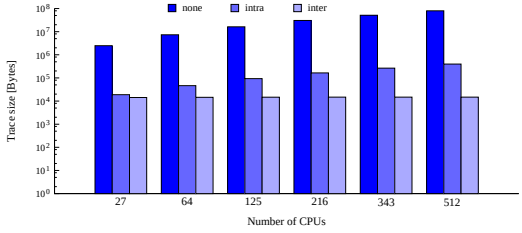

(b) 1D Stencil Mem. Usage, Varied # Nodes



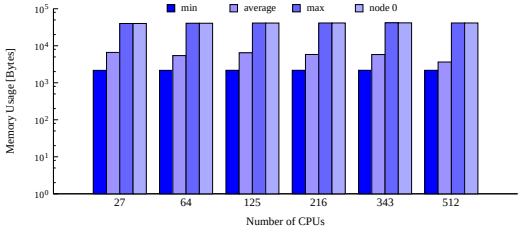

(c) 2D Stencil Trace File, Varied # Nodes



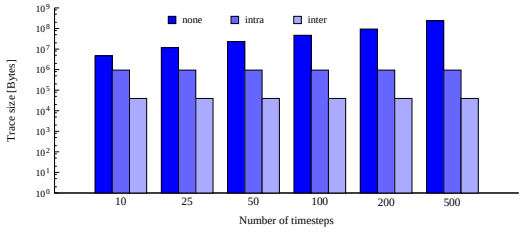

(d) 2D Stencil Mem. Usage, Varied # Nodes



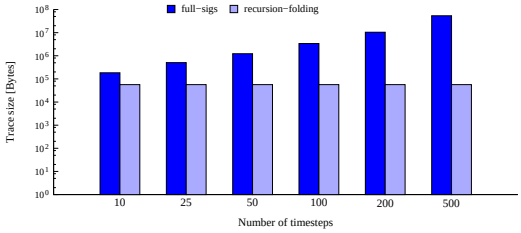

(e) 3D Stencil Trace File, Varied # Nodes




(f) 3D Stencil Mem. Usage, Varied # Nodes




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




(h) Recursion Trace File, Varied Recursion Depth

Fig. 9. Trace File Size (left) and Memory Usage of the Compression Algorithm (right) for Microbenchmarks per Node on BlueGene/L

### *5.1 Trace Sizes & Memory Requirements*

Fig. 9 depicts the size of trace files and the memory requirements of the actual compression algorithm on a per-node basis on BG/L for the tests described in the previous section. Figures 9(a), 9(c) and 9(e) show the trace file sizes of the 1D, 2D and 3D stencil codes, respectively, for varying number of nodes. We show trace

sizes on a logarithmic scale for different node numbers (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 signifi cant 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 magnitude, but trace sizes still increase by two orders of magnitude across the node range. Hence, neither approach is scalable with respect to the number of nodes. The fully compressed trace sizes, in contrast, are constant in size irrespective of the number of nodes, which illustrates that our combined intra- and inter-node compression technique scales well. The resulting trace sizes, 3.5KB, 10KB and 14KB, for 1D, 2D and 3D stencils, concisely represent MPI events, in contrast to trace size ranges obtained without compression of 0.25-14MB, 0.3-21MB and .25-80MB. Increases between stencil sizes reflect the number of distinct patterns required to represent corner nodes, boundary nodes and interior nodes as distinct RSDs.

As BG/L is a memory-constrained architecture, keeping the memory pressure low during on-the-fly compression is as important as the resulting trace fi le size. Figures 9(b), 9(d) and 9(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. This metric includes the merge queues for intra- and inter-node compression but excludes storage of the actual trace. 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 our 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 insignifi cantly more memory. We measured a minimum (maximum) memory usage of 5KB (13KB), 7KB (33KB) and 3KB (42KB) for the 1D, 2D and 3D stencil problems (per node), respectively.

Figure 9(g) depicts the trace fi le 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 confi rms that the number of loop iterations has no effect on compression after RSDs and PRSDs are formed, irrespective of inter-node merge compression.

Results from the recursion experiment depicted in Figure 9(h) show trace sizes for different numbers of timesteps, where each timestep is coded as one recursive call. The results indicate that trace sizes with inter-node compression are orders of mag-

nitude larger when full backtrace signatures are recorded as opposed to recursion-folding signatures (see Section 2). The full signature overhead grows proportionally to the recursion depth, i.e., the savings due to recursion folding are even higher as recursion depth increases.

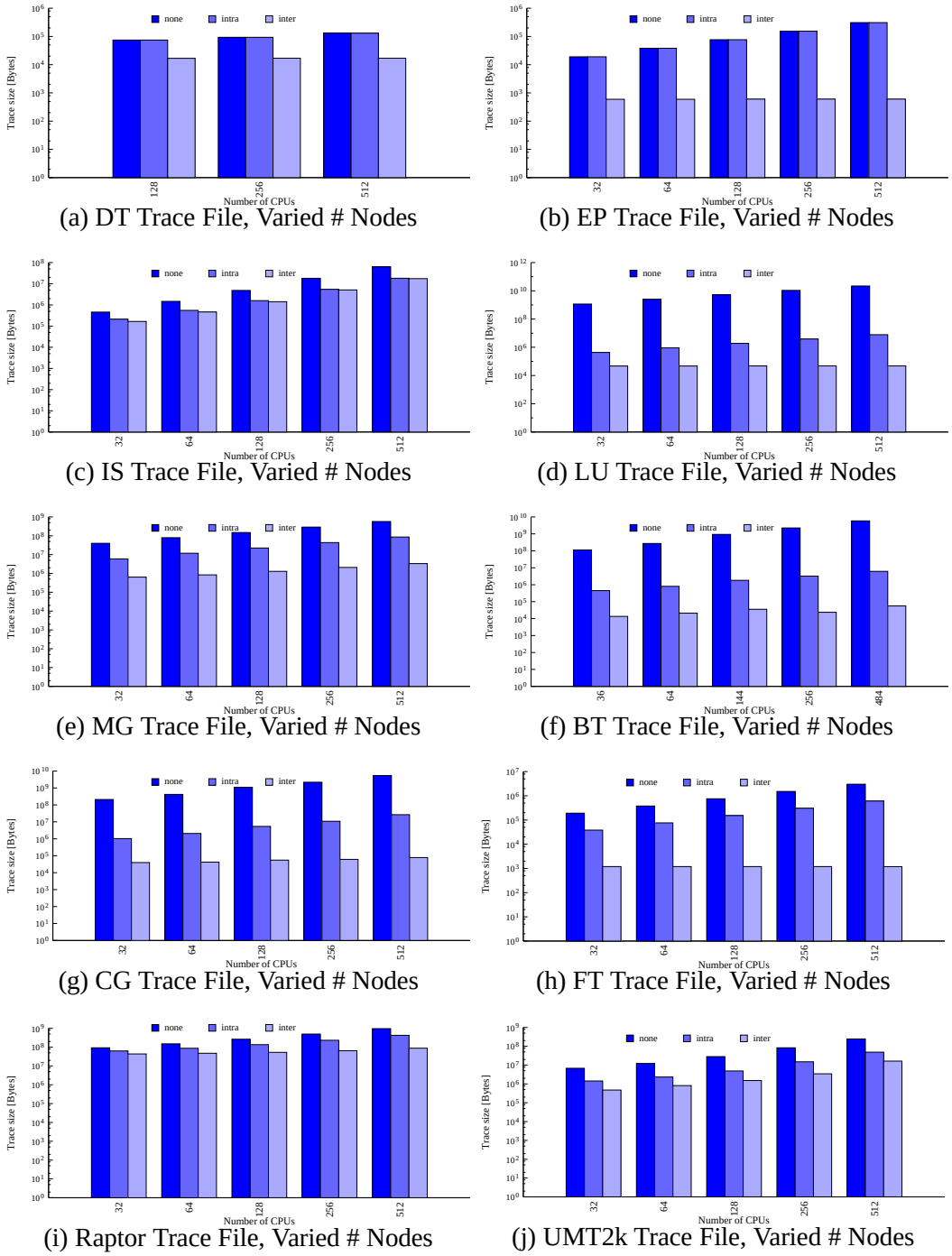
Figure 10 shows the trace file sizes for the NPB suite as well as the applications Raptor and UMT2k on a log-scale. We can distinguish three categories of codes, those that result in near constant-size traces, regardless of the number of nodes, those with sub-linear scaling of trace size as the node count increases and those that do not scale with our current techniques.

In our previous work, we observed that for our inter-node compression techniques the applications mapped into these three categories as follows [20]: DT, EP, and IS exhibited near-constant trace sizes irrespective of number of tasks; LU and MG showed sub-linear scaling with number of tasks; and BT, CG, and FT resulted in faster growing non-scalable traces sizes. Using our second generation algorithm describe in Sections 2 and 3) we significantly improved these results further, with the exception of IS, as described below.

With these novel domain-specific compression enhancements, more applications fall into the first category for inter-node compression. DT, EP, LU, and FT show near-constant trace sizes. MG, BT, CG and Raptor exhibit trace sizes with sub-linear growth as the number of nodes increases. IS and UMT2k result in non-scalable traces sizes.

For the first category (DT, EP, LU and FT), trace sizes increase exponentially without compression or with 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 profited significantly from encoding wildcard communication end-points (MPI_ANY_SOURCE) directly instead of storing them as offsets. FT benefited from relaxed communication parameter matching, i.e., mismatches in selected parameters, such as source/destination, are tolerated and complemented by an ordered list of (value, ranklist) pairs.

MG, BT, CG and Raptor 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. Relative to our earlier results, CG benefited from relaxed communication parameter matching, similar to FT. BT's improvement is due to the omission of tags for point-to-point communication where they were deemed semantically irrelevant (see Section 2), which significantly lowered intra-node compression sizes. Intra-node compression works well for MG, BT and CG, but end-point mismatches in inter-node compression prevent better compression. More specifically, a reduction step coded as a sequence of sends / non-blocking receives over an application-specific overlay tree in BT prevents better




(a) DT Trace File, Varied # Nodes

(b) EP Trace File, Varied # Nodes

(c) IS Trace File, Varied # Nodes

(d) LU Trace File, Varied # Nodes

(e) MG Trace File, Varied # Nodes

(f) BT Trace File, Varied # Nodes

(g) CG Trace File, Varied # Nodes

(h) FT Trace File, Varied # Nodes

(i) Raptor Trace File, Varied # Nodes

(j) UMT2k Trace File, Varied # Nodes

Fig. 10. NPB, Raptor and UMT2k Trace File Size per Node on BlueGene/L

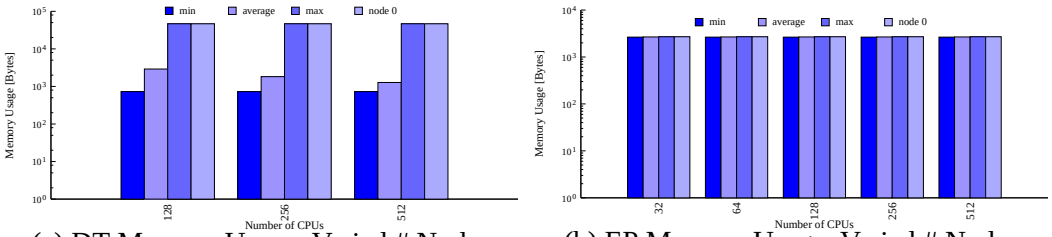
compression, which, if coded as a native MPI reduction, would have compressed perfectly. MG utilizes 3D overlay to select communication endpoints whose mapping is a mismatch for relative encoding. Only Raptor shows much lower compression rates for intra-node (or inter-node) methods due to its unstructured mesh transport communication. We conclude that the main benefi t of a size reduction by multiple orders of magnitude stems from the intra-node scheme for regular

communications while unstructured communications may results in lower savings. Nonetheless, inter-node merge can result in up to two order of magnitude of compression, as seen for BT in this category, for a total of five orders of magnitude savings over uncompressed traces at 484 nodes.

Results for the remaining codes, IS and UMT2k, indicate reductions in trace size with inter-node compression of about 2 orders relative to no compression and up to one order of magnitude compared to intra-node compression. IS is non-scalable due to its dynamic rebalancing of work, which results in different sized payloads for an MPI_Alltoallv() collective upon each call (incorrectly encoded in constant size in a previous version of ScalaTrace). Constant-size traces could be obtained here, but only with a domain-specific parameter optimization that aggregates values and looses information (see end of Section 2). UMT2k falls into the non-scalable category. It still has room for improvement subject to ongoing investigation (end of Section 2). But even for these cases, our compressed traces are already at least two orders of magnitude smaller than traces without compression.

Figure 11 depicts the memory requirements for inter-node compression for the same set of codes on a logarithmic scale. For codes whose trace sizes scale (DT, EP, LU and FT), 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 (MG, BT, CG, Raptor and, even more so, IS, UMT2k), memory usage is constant at leaf nodes (minimum metric) but increases for larger node counts towards the root (node 0). These two fundamental trends are representative of all codes.

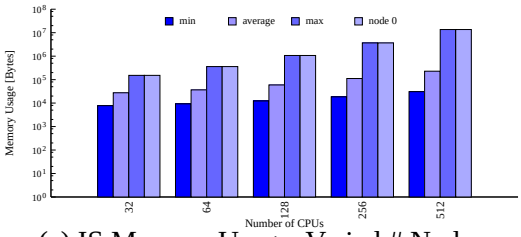
This creates additional memory pressure on the nodes responsible for higher layers of the reduction tree and reduces the memory available to applications. We could reduce these memory costs by off-loading the inter-node compression to an external reduction infrastructure only leaving the leave nodes of the reduction tree, which only have low memory requirements, co-located with application tasks. 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 [23] provides a framework for computation offloading to I/O nodes (see end of Section 3).



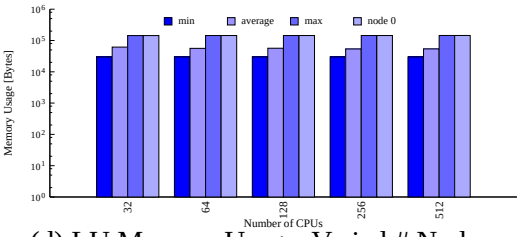

(a) DT Memory Usage, Varied # Nodes



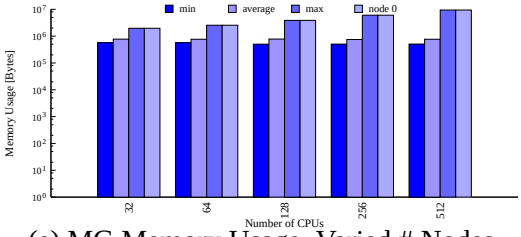

(b) EP Memory Usage, Varied # Nodes



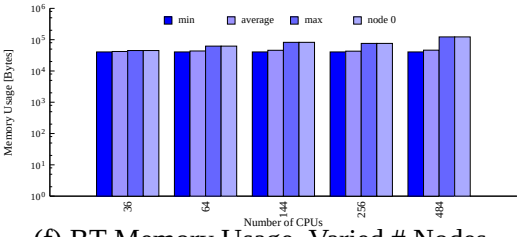

(c) IS Memory Usage, Varied # Nodes



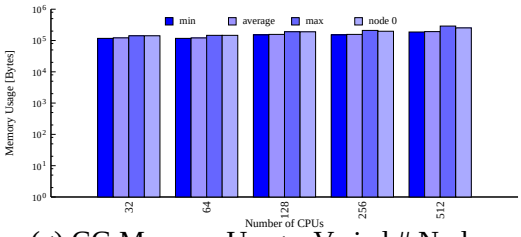

(d) LU Memory Usage, Varied # Nodes



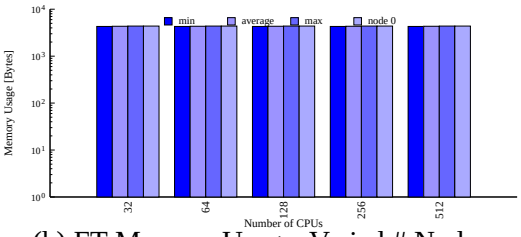

(e) MG Memory Usage, Varied # Nodes



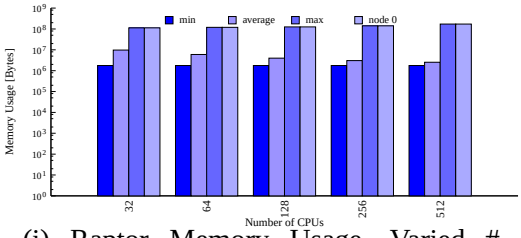

(f) BT Memory Usage, Varied # Nodes



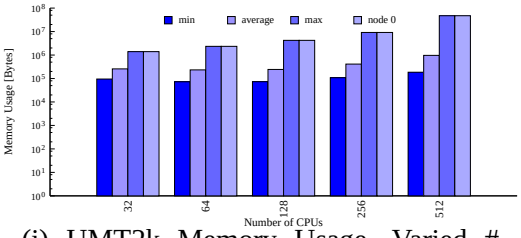

(g) CG Memory Usage, Varied # Nodes




(h) FT Memory Usage, Varied # Nodes




(i) Raptor Memory Usage, Varied # Nodes

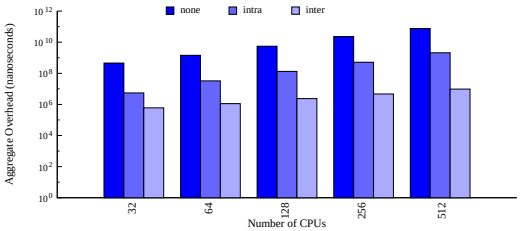



(j) UMT2k Memory Usage, Varied # Nodes

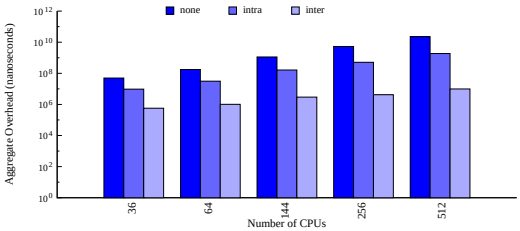
Fig. 11. NPB, Raptor and UMT2k Memory Usage per Node on BlueGene/L

## *5.2 Inter-Node Merge Overhead*

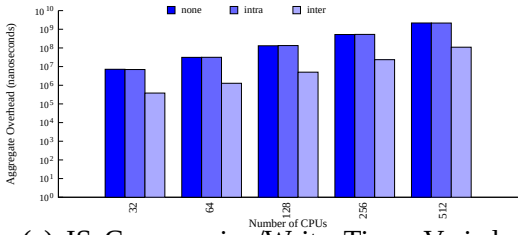
Figures 12(a), 12(b) and 12(c) depict the runtime overhead on a logarithmic scale for LU, BT and IS with no compression (none), with only intra-node compression



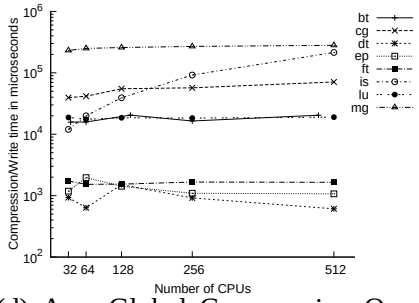

(a) LU Compression/Write Time, Varied Number of Nodes



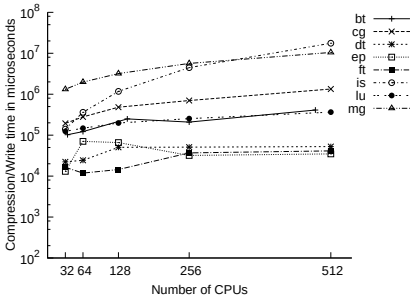

(b) BT Compression/Write Time, Varied Number of Nodes




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




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




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

Fig. 12. NPB Compression / Write Time per Node on BlueGene/L

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 of the three classes of benchmarks.

We observe that inter-node compression has the lowest overhead for LU, 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. BT'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. IS shows the lowest overhead without compression. Not surprisingly, inter-node compression is most costly since IS belongs to the class of benchmarks with super-linear compression space requirements.

These benchmarks are representative for their respective classes.

Figures 12(d) and 12(e) show the average and maximum inter-node compression time measured inside of MPI_Finalize. These results indicate a wide spread of overhead, which generally correlates with the compression rate achieved. Consider Figure 12(e). IS has the highest asymptotic overhead exceeding that of other NPB codes, followed by MT, BT and CG and then the remaining code with near-constant size trace sizes. However, relative differences exist, particularly for smaller number of nodes since the overhead with fewer nodes is more closely related to the amount of MPI calls issued by the respective application than scalability of compression. Also, average and maximum overheads (Figures 12(d) and 12(e)), widely match in their trends between mid-level and top-level nodes during inter-node compression in the tree. Still, other enhancements, such as off-loading inter-node compression to I/O nodes, should further lower the overhead.

### *5.3 Timestep Loop Identification*

The trace format utilized by ScalaTrace preserves the program structure, even in its compressed form. This provides novel opportunities for program analysis in a scalable and efficient manner. To illustrate this capability, we conducted a study with the NPB codes to determine their timestep loop, *i.e.*, the outermost loop of the code that contained repeated MPI calls. This timestep loop is of particular interest for performance modeling as convergence algorithms are often based on either fixed-iteration bounds for the number of timesteps or epsilon-based error constraints resulting in input-specific number of timesteps.

| NPB Code | Actual # Timesteps | Derived # Timesteps |
|---|---|---|
| BT | 200 | 200 |
| CG | 75 | $1 + 37 \times 2$ |
| DT | N/A | N/A |
| EP | N/A | N/A |
| IS | 10 | $2 \times 5, 2 \times 2 + 2 \times 3$ |
| LU | 250 | 250 |
| MG | 20 | $20, 2 \times 10$ |

Table 1
Actual and Derived (from Trace) Number of Timesteps

Table 1 depicts for each NPB code the actual number of timesteps for class C inputs and the equivalent number derived from the compressed trace representation. Benchmarks DT and EP do not contain a timestep loop at all. The traces identified the exact number of timesteps for BT and LU within their trace representation. For IS, mismatches between MPI call parameters result in an outer loop with a longer pattern of fewer iterations, *e.g.*, a sequence of three MPI calls is flattened into a sequence of six calls, which is then repeated five times. The total number of MPI calls (30) is the same, and so is the number of unique MPI calls (three) if parameters

were ignored, which are repeated ten times. Yet, two different compressed patterns are observed in intra-node traces ($2 \times 5$ and $2 \times 2 + 2 \times 3$ repetitions). CG and MG resulted in traces that, at first sight, did not reveal the number of timesteps. Closer inspection revealed mismatches between parameters that are similar to IS but result in repetitions with a reordered event sequence. For example, instead of repetitions of a <*, op1, op2, op3> sequence, we would observe op1, op2, <*, op3, op1, op2>, op3 in the trace with one less iteration, which is actually equivalent, and we report them in Table 1 as expressions to emphasize that these patterns are less obvious. We are currently refining our trace compression scheme to handle parameter lists similar to ranklists, either at inter-node compression or at inter-node merge. This would result in a more obvious timestep pattern. Locating the timestep loop in the trace further gives ScalaTrace the ability to indicate the location of the loop in the source code. By traversing the stack trace, we can indicate the location of the actual MPI calls of a timestep loop (or any other loop in the trace, for that matter), and we can provide information about calls from higher-level routines along the backtrace. The loop itself can thus typically be located in the source code as being contained within the highest stack frame with a common call across multiple MPI calls within a PRSD. Occasionally, the loop may be coded at a higher level if a subroutine contains just the sequence of calls without iteration, but following the call sequence to find the actual loop is trivial.

### *5.4 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 as well as 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 with our ScalaReplay tool, 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. In our recent work, we showed that delta time recording of computational overhead still results in near constant-size traces for many codes and enables time-preserving replay of communication traces without running the actual application [22]. This supports the assessment of communication needs for current and particularly 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 [14]. PRSDs originally targeted on-the-fly memory trace compression [17]. 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 [30] 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 [27]. It reports aggregate metrics. Hence, structural information and event ordering are not preserved. There are many other tools [1,26,28,12] that report aggregate information, often based on the profiling layer of MPI, as is the case with mpiP or produce terabytes of traces. None of these tools are suitable for lossless tracing and later replay in a scalable manner with a single trace file in the megabyte range.

Vampir is a commercial tool set including a trace generator and a display engine to visualize MPI communication [6]. 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. In contrast, our technique compresses traces to sizes that are three orders of magnitude smaller and do not significantly increase in size, if at all, during strong scaling. Our current work [22] further shows that scalability need not be sacrificed even when timing information is included.

Paraver/Dimemas is an MPI tracing tool set from the Technical University of Catalonia (UPC) in Barcelona [21]. Paraver provides functionality similar to Vampir and its trace generator has similar limitations. Dimemas is a discrete-event-based network performance simulator that uses Paraver traces as input. While it bears some similarities to our replay mechanism, 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.

Casas et al. [7] recognize multi-level regularities in large, post-mortem trace files. By detecting patterns, they compress these flat trace files offline and can filter background (operating system) activity artifacts. The compression is reported to take up to an hour for benchmarks comparable to NAS. Our method, in contrast, compresses regularities on-the-fly and never generates any flat trace file.

Geimer et al. [11] obtain per-node traces stored locally at each node and later replay these communication traces on the same architecture with the same number of nodes to detect communication bottlenecks. Later work generalized this approach to Grid environments using distributed time stamp synchronization [5]. In contrast, the focus of our work is primarily on concise tracing. The representation of time is beyond the scope and discussed in one of our more recent papers [22]. While our traces remain small and often constant size, their trace sizes are reported to reach 10GB, the same order of magnitude reported by others [8].

Arnold et al. [4] developed a scalable tool to identify task behavior equivalence classes with high similarity based on stack signatures. Their approach utilizes MRNet, a software overlay network that provides efficient multicast and reduction communications [23]. MRNet uses a tree of processes between the tool's front-end and back-ends to improve group communication. MRNet introduces additional complexity, which we decided to avoid in our current prototype. MRNet would support on-the-fly and asynchronous trace compression across tasks. By using MRNet, we would further reduce the memory pressure of our trace generator. MRNet could be used in a future version of our tool using $P^N MPI$ as the glue layer between the tools [24].

The Open Trace Format (OTF) is targeted at scalable tracing, yet without any advanced (domain-specific) compression scheme [15]. 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, OTF provides the ability to produce multiple streams and, hence, store and load a trace in parallel with user-defined granularity.

An alternate trace format by the same group uses so-called cCCGs, a structural compression format that combines regular patterns into common sub-trees [16]. This ultimately results in a rooted directed acyclic graph where ancestors can have common children, which denotes a regularity (or commonality) between these ancestors in terms of their calling pattern of traced events. In their data structure, measurements (*e.g.*, delta times) are stored one-by-one per call up to a maximum branching factor $b$. Beyond that threshold, an interior node is split into two nodes (with an aggregate storage capacity of $2b$) so that further delta values can be stored. By combining deltas that only differ by a certain percentage of their value, common nodes in the graph can be combined resulting in a set of parents for the merge candidate. The theoretical upper bound on their storage complexity is linear to the

number of traced events, a case that would only occur if all deltas differ significantly. In practice, the observed storage requirement for regular event patterns is reported to be logarithmic due to combining nodes upon matching patterns and deltas. In contrast, our storage overhead is as low as constant when event patterns are regular.

Freitag et al. [10] describe a window-based compression scheme and evaluate its applicability to OpenMP traces. Our PRSD compression is more powerful as it allows recursive compression online. Neyman *et al.* [19] designed a tool to detect races in PVM codes using a trace generation and a replay tool. Also, recent work by Mesnier *et al.* focuses on I/O trace generation and replay [18]. Neither of these techniques scale as they do not perform any compression. Our approach is also designed to handle MPI I/O calls much the same as regular MPI events.

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 [25]. 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 of thousands of nodes. However, communication patterns of large-scale scientific applications are often too complex to analyze at the source-code level. Existing tools can be classified into two different categories: profilers gather aggregate metrics statistically in a scalable manner, but temporal ordering and structural information are generally lost in such an approach. Tracing tools, on the other hand, provide complete, lossless traces, which, however, grow significantly in size as the problem size or the number of processors increases. This makes it harder, if not infeasible, to commit such traces to a global file system.

In this work we present a novel tracing framework that combines the advantages of both approaches: we extract full communication traces, which are orders of magnitude smaller or even of near-constant size regardless of the number of nodes while preserving all structural and temporal-order information. 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 full-sized applications.

We assessed the correctness of our approach by comparing the temporal orderings and aggregate counts of MPI events during the original run with the replay. This replay mechanism may further 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 precedent.

## References


[1] OpenSpeedshop for linux.

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

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

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

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

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

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

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

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

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

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

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

[13] Jeff Greenough, Allen Kuhl, Louis Howell, Alek Shestakov, Ulrike Creach, Al Miller, Ellen Tarwater, Andrew Cook, and Bill Cabot. Raptor – software and applications on bluegene/l. BG/L workshop paper 22, Lawrence Livermore National Lab, October 2003. http://www.llnl.gov/asci/platforms/bluegene/papers/22greenough.pdf.

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

[15] 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.

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

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

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

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

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

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

[22] P. Ratn, F. Mueller, Bronis R. de Supinski, and M. Schulz. Preserving time in large-scale communication traces. In *International Conference on Supercomputing*, pages 46–55, June 2008.

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

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

[25] Shuyi Shao, Alex Jones, and Rami Melhem. A compiler-based communication analysis approach for multiprocessor systems. In *International Parallel and Distributed Processing Symposium*, 2006.

[26] S. Shende and A. Malony. Portable profiling and tracing for parallel, scientific applications using c++. In *SIGMETRICS Symposium on Parallel and Distributed Tools (SPDT)*, pages 134–145, 1998.

[27] 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.

[28] F. Wolf and B. Mohr. Kojak - a tool set for automatic performance analysis of parallel applications. In *Proc. of the European Conference on Parallel Computing (Euro-Par)*, volume 2790 of *Lecture Notes in Computer Science*, pages 1301–1304, Klagenfurt, Austria, August 2003. Springer. Demonstrations of Parallel and Distributed Computing.

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

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