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] Department of Computer Science, North Carolina State University, Raleigh, NC
[2] Lawrence Livermore National Laboratory, CASC, Livermore, CA
mueller@cs.ncsu.edu, phone: +1.919.515.7889, fax: +1.919.515.7896

Abstract

Characterizing the communication behavior of large-scale applications is a difficult and costly task due to code and system complexity as well as 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 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, near constant-size 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 fall into the former or later category, *i.e.*, their storage requirements either do not scale or they are lossy with respect to program structure and temporal ordering.

In contrast to prior work, we promote a trace-driven approach to analyze MPI communication that scales. 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 of near constant size regardless of the number of nodes while preserving structural and temporal-order information of events.

Our trace-gathering framework, depicted in 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. Such intra-node information (task-level) is compressed on-the-fly. Upon application termination, inter-node compression is triggered over all nodes resulting in a single trace file that preserves structural information suitable for lossless replay.

We assess the effectiveness of our framework by conducting experiments with benchmarks and an application on BlueGene/L. The results obtained confirm the scalability of on-the-fly MPI trace compression by yielding near constant size traces for processor scaling and problem scaling.

In addition to trace generation, we have designed a tool to replay compressed traces on-the-fly independent of the original application and without decompressing the trace. We demonstrate this ability to verify the correctness of our trace compression. We also discuss the use of the replay mechanism in performance tuning MPI

* 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.

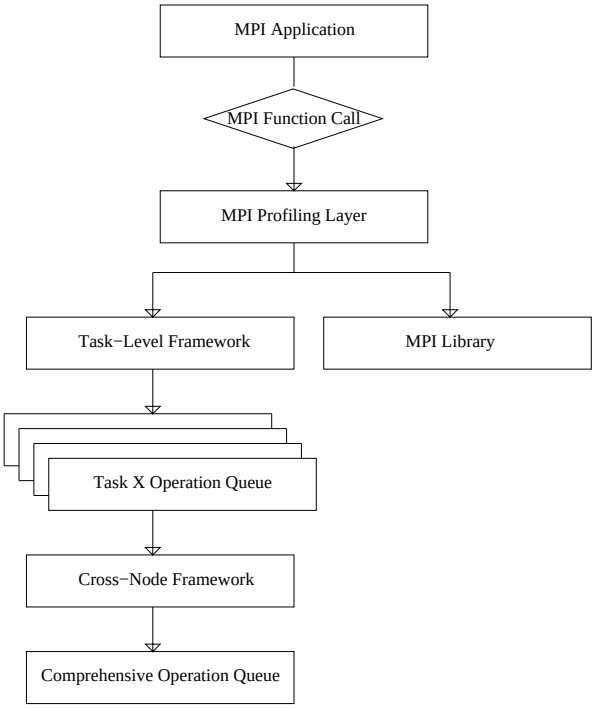


Fig. 1. Interaction of Components

communication and facilitating projections of network requirements for future large-scale procurements.

To the best of our knowledge, near constant-size representation of MPI traces in a scalable manner 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. Using the PMPI layer, wrappers of MPI calls are specified to generate an entry per call, including the type of MPI call, source and destination of communication and other parameters, yet excluding actual message content. These MPI call entries, generally repeated due to an application's loop structure, are subject to on-the-fly compression.

We extend regular section descriptors (RSDs) for single loops to express MPI events nested in a loop in constant size [4] while power-RSDs (PRSDs) are utilized to recursively specify RSDs nested in multiple loops [5]. MPI events may occur in inner-most or outer loops alike 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 (a) the former loop (RSD1) followed by (b) 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 [2]. Our algorithm, depicted in Figure 2, proceeds in four steps. First, head and tail of 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.

```
Compress_Queue(Queue Op_Queue
    Target_Tail = Op_Queue.tail
    Match_Tail = Search for match of Target_Tail
    if (Match_Tail)
        Target_Head = Match_Tail.next
        Match_Head = Search for match of Target_Head
        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
```

Fig. 2. Task-level Compression on MPI Event Queue

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.

A number of encoding techniques are used to enable compression across nodes. While inter-node compression is detailed in the next section, the following encoding schemes 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 the call sites of the calling stack in stack walk. Call sites, including the MPI call itself, create a unique signature of an MPI call-chain and are subsequently treated as part of the event, *i.e.*, they are required 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, relative encodings of communication end-points are utilized by our framework, *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 as an example 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.

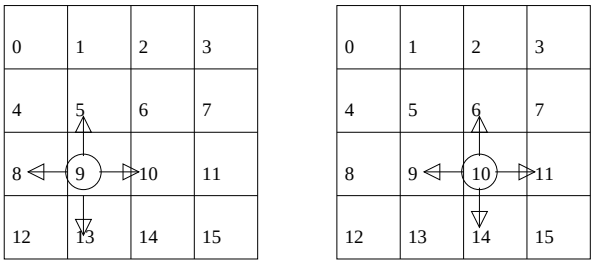


Fig. 3. Communication Endpoint Encoding

Request Handles: Handles for asynchronous MPI calls are runtime-specific, *i.e.*, they cannot simply be recorded as their abstractions (generally pointers) are invocation-dependent. We record these handles in a buffer. When an asynchronous call references a handle, the invocation-dependent pointer is determined by finding a match in the handle buffer. The MPI event then records its handle offset relative to the last element of the buffer. Relative indexing again enables subsequent cross-node compression. Upon message replay, this buffer is created on-the-fly and indexed by 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, as explained before, we can now 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 have to be retained in the trace (well beyond handles) and, as explained later, 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 application developers suggesting to them to replace point-to-point communication with collectives. Hence, our tool can be utilized to detect certain scalability problems within the communication design of algorithms.

Event Aggregation: Our approach generally has to preserve event ordering and program structure information. However, non-deterministic repetitions of MPI calls, such as triggered by, *e.g.*, 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, such sequences of MPI calls are are squashed into a single event that records the number of completed asynchronous calls. Notice that this preserves compression capabilities while exploiting MPI-specific semantics. Even during replay, successive MPI_Waitsome calls are aggregated till the recorded number of completions is reached.

3. Inter-Node/Cross-Node Trace Compression

Local traces are combined into a single global trace upon application completion, *i.e.*, 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 have to 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, stepwise 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

[1] 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.

child (slave queue) is merged into the local trace of the current node (master queue), then the trace of the other child (slaved) is merged similarly into master queue resulting from the former merger. Each merge operation is performed by the algorithm depicted in Figure 4. The

```
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
```

Fig. 4. Merge Slave (Child) into Master (Parent) Trace

slave queue is merged into the master queue by identifying matching sequences of operations using three iterators, namely, master and slave iterators as well as a slave head. The master iterator is used as a place holder for the current operation sequence in the master queue. The slave head is used as a place holder for the last matched operation sequence in the slave queue. Lastly, the slave iterator is used to identify matching sequences between the master queue and the slave queue.

The algorithm starts all iterators at the beginning of their respective queues. The slave iterator works its way down the slave queue attempting to find an operation sequence matching the current master iterator. If a match is found, all mismatching operation sequences are first copied into the master queue preceding the master iterator. The mismatching operation sequences are those between the slave head (the last matching operation sequence in the slave queue) and the slave iterator (the current matching operation sequence in the slave queue). This ensures that the order of operations from the slave queue is maintained. The slave iterator's task participant list is then appended to its twin's task participant list (demarcated by the master iterator).

Temporal Cross-Node Reordering: The merge algorithm compresses well at lower levels of the reduction tree but runs into problems at higher levels. The problem lies in its inability to merge differing sequences from multiple nodes in a deterministic 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. This is determined by testing if the intersection of tasks in the unmatched sequence with those of the matched sequence is empty. 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: During the merge process, task IDs are encoded as PRSDs themselves to specify which subset of nodes participated in some set of events.[2] This allows us to 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 resulting in nine different patterns to record for 2D stencils, regardless of the number of nodes. This approach makes cross-node compression feasible and results in a single near constant-size trace file, which is by far more efficient than storing per-node trace files and later consolidating them.

Reduction over a Radix Tree: We internally use a binary radix tree 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 across these nodes during task ID compression.

4. Experimental Framework

The communication trace components can be integrated transparently into arbitrary MPI applications, either by using dynamic linking (as in most environments) or by explicitly linking with our components (*e.g.*, in BG/L environments where static linking is required). Exper-

[2] 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.

imental results have been gathered for 1D, 2D and 3D stencil benchmarks, codes from the NAS Parallel Benchmark and the Raptor application.

The 1D stencil has a one-dimensional logical space based on a task's MPI rank. For each time step in the 1D stencil, a task will communicate to its two left neighbors and two right neighbors (three-point stencil). The communication step consists of sending and receiving from these neighbors. A task will only proceed to its next step after both the sends and receives for the current step are complete.

The 2D stencil has a two-dimensional logical space where a logical address is calculated as $x = rank/dimension, y = rank \bmod dimension$. Communication occurs with all eight neighbors (including diagonal neighbors) for a nine-point stencil. See 1D stencil for other details.

The 3D stencil has a three-dimensional logical space where a logical address is calculated as $x = rank \bmod dimension, y = rank/dimension$, $z = rank/dimension^2$. Communication occurs with all 26 neighbors (including diagonal neighbors) for 27-point stencil. See 1D stencil for other details.

The NAS Parallel Benchmark (NPB) codes were selected from NPB version 3.2.1 for MPI [9]. The CG, EP, FT and IS codes worked without problems on BG/L while the remaining benchmarks (BT, DT, LU, MG, SP) experienced compilation or execution problem, some of which may be unique to the BG/L system environment on the compute nodes with their proprietary non-POSIX compliant kernel and libraries. (We expect to eventually have results for all of the NPB.)

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 [3]. It supports MPI and Pthreads parallelization and communicates on a 27-point stencil using asynchronous communication. We utilize the MPI capabilities in a hydro-dynamics simulation using the same input while varying the number of processors.

Experiments are conducted on a 1024-node BlueGene/L (BG/L) machine. This architecture has severe limitations on memory [1]. Hence, any traces generated may only consume small amounts of the available core memory. The memory consumption of the compression subsystem is measured and reported as task-0 (root node of the reduction tree), minimum, maximum, average value. Furthermore, the size of trace files is reported. Two sets of experiments were conducted. First, the number of processors (nodes) were varied 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 NPB codes and raptor) or n^d processors (for the stencil benchmark) for a d-dimensional stencil with a base of n, *e.g.*, $7^3 = 343$ nodes. Second, the number of time steps is varied to assess the effect of the number of iterations on trace file sizes.

Additional experiments were conducted to assess the cost of writing compressed traces, one per node, to a parallel file system. While cross-node merging primarily inflicts time overhead, writing per-node traces to the file system may result in large storage and I/O requirements. We also measured the time for writing to I/O nodes over a Lustre parallel file system, which is the fastest global file system available on this BG/L machine.

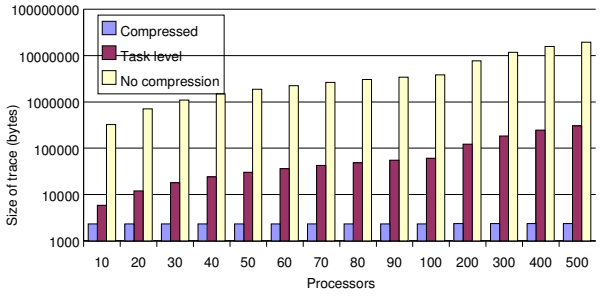
5. Experimental Results

We conducted three sets of experiments to assess the effectiveness of our compression techniques, to determine the overhead of cross-node compression and to verify the correctness (lossless compression) of our approach during replay.

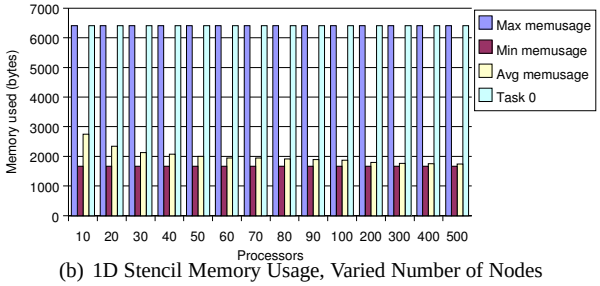
5.1 Trace Sizes and Memory Requirements

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

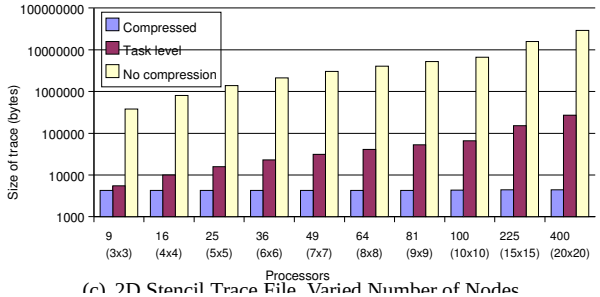
Figures 5(a), 5(c) and 5(e) depict the trace file sizes of the 1D, 2D and 3D stencil codes, respectively, for varying stencil sizes (number of nodes). Trace sizes are reported on a logarithmic scale for the nodes (a) without compression, (b) only with task-level (intra-node) compression and (c) with 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. Task-level compression reduces this overhead by two orders of a magnitude, but the increasing trend in size over the number of nodes is retained (increase of two orders of magnitude again). 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. This is in contrast to 0.3-19MB, 0.3-29MB and 2MB-61MB for the respective stencil sizes in the absence of compression (ranges for the smallest and largest stencil sizes). Increases between stencil sizes reflect the number of distinct patterns required to represent corner nodes, boundary nodes and interior nodes as RSDs.



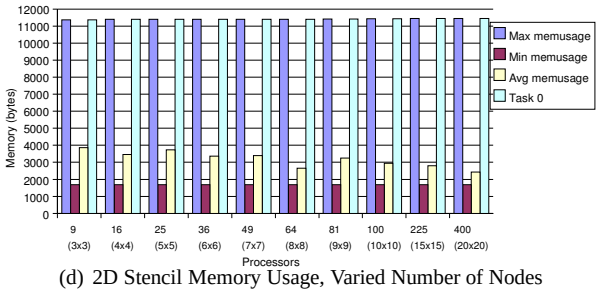
(a) 1D Stencil Trace File, Varied Number of Nodes



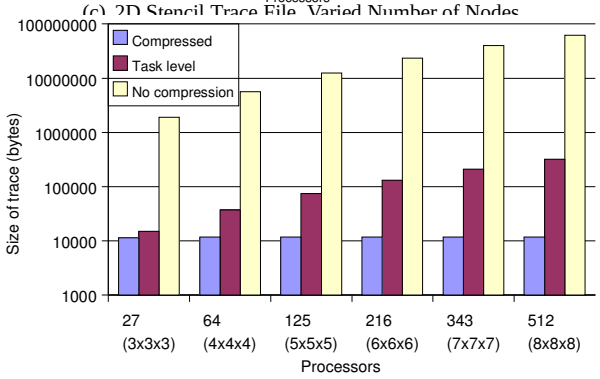
(b) 1D Stencil Memory Usage, Varied Number of Nodes



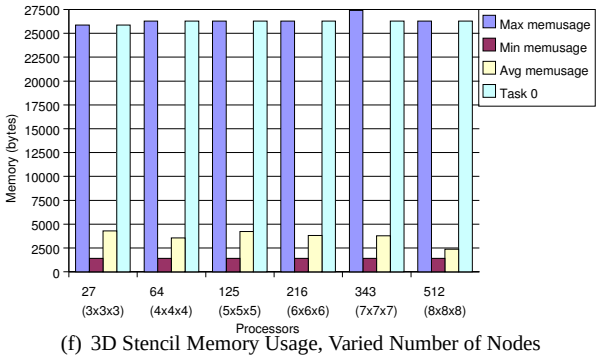
(c) 2D Stencil Trace File, Varied Number of Nodes



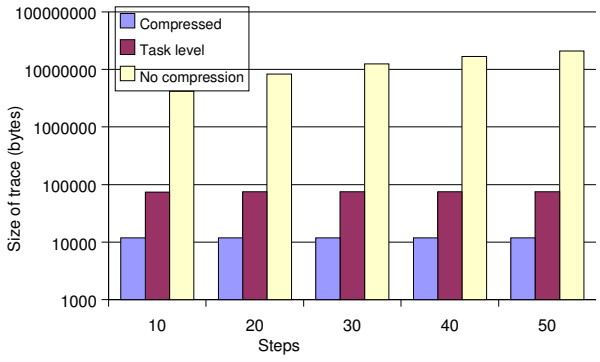
(d) 2D Stencil Memory Usage, Varied Number of Nodes



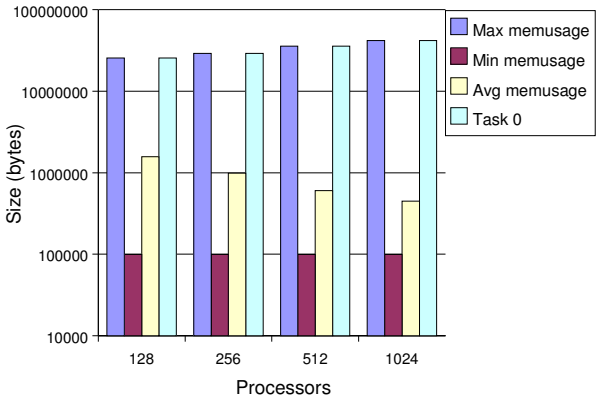
(e) 3D Stencil Trace File, Varied Number of Nodes



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



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



(h) Raptor Application

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

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 reflecting the intra- and inter-node compression components for the 1D, 2D and 3D stencil benchmarks, respectively, over varying stencil sizes. Minimum, average, maximum and task-0 (root node) memory usage is reported 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 high watermark of 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. Notice that this metric includes the merge queues for intra- and inter-node compression but excludes storage of the actual trace, which is reported as trace file sizes, as discussed before.

Figure 5(h) depicts the memory usage for Raptor confirming most of the prior benchmark observations for a complex application code. In addition to the previous observations, a slight increase in the maximum memory usage of 38MB to 55MB for 128 and 1024 nodes, respectively, can be seen. This increase is due to minor inefficiencies of cross-node compression currently being addressed, as is the total amount of memory required due to the severely memory-constrained nature of BG/L nodes.

Figures 6(c) and 6(g) depict the trace file size for the NPB codes EP and IS, respectively. Without or with intra-node compression only, trace sizes increase exponentially. Yet, Full compressing with cross-node merging results in constant trace sizes. There are only few and very regular communication calls in these codes, *i.e.*, only collectives, except for a pipeline of sends and asynchronous receives along the chains of ranks. Figures 6(d) and 6(h) depict the memory requirements for full compression. The amount of memory used remains unchanged 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.

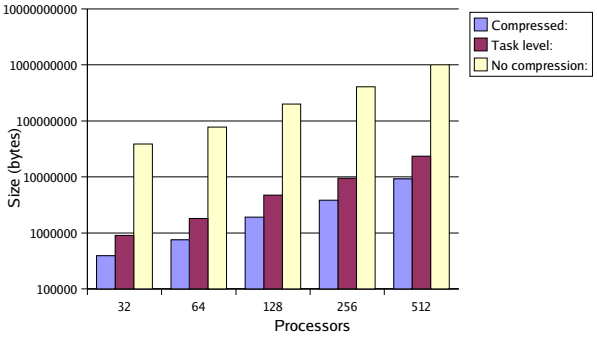
Figures 6(a) and 6(e) depict the trace file size for the NPB codes CG and FT, respectively. CG requires significantly larger trace sizes without compression due to a large number of point-to-point communications, some of which are asynchronous. FT, on the other hand, benefits more significantly from cross-node compression, which is due to all-to-all collectives that are consolidated across nodes. Both codes show the smallest trace sizes for full compressing. Nonetheless, all techniques show exponential increases (at different magnitudes), which indicates that there is room for improvement to obtain near-constant trace sizes. We have found shortcomings in our intra-node compression that later prevents cross-node merging, *e.g.*, when absolute references instead of relative indices are used. Figures 6(b) and 6(f) depict the corresponding memory requirements for full compression, which vary significantly (one to two orders of magnitude) depending on the node location in the reduction tree. This reconfirms the prior observation that inefficiencies in the intra-node scheme adversely affect cross-node merging at this time.

Figure 5(g) depicts the trace file size as the number of time steps is varied for the 3D stencil problem, *i.e.*, as the iteration bound of the outer-most convergence loop is varied while the number of nodes remains constant at 125 processors. 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.

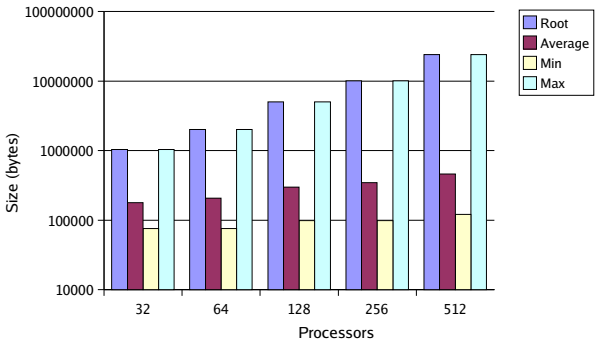
5.2 Verification of Correctness during Replay

Additional experiments were conducted to verify the correctness of our approach. We replayed compressed traces to assess if MPI semantics are preserved, to verify that the aggregate number of MPI events per MPI call matches that of the original code and that temporal ordering of MPI events within a node are observed. The results of communication replays confirmed the correctness of our approach to this respect.

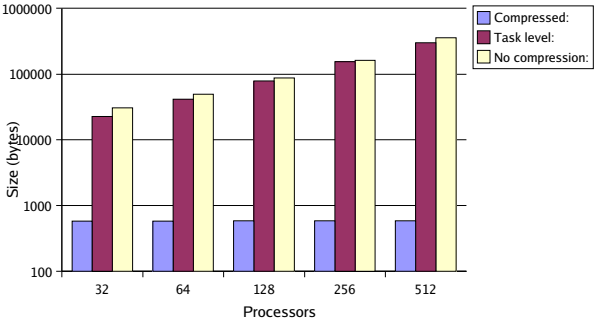
During replay, all MPI calls are triggered over the same number of nodes with original payload sizes, yet with a “random” message payload (content). This inflicts comparable bandwidth requirements on communication interconnects, albeit with potentially different contention characteristics. Communication replay also provides an abstraction from compute-bound application performance, which is neither captured nor replayed. This makes the replay mechanism extremely portable,



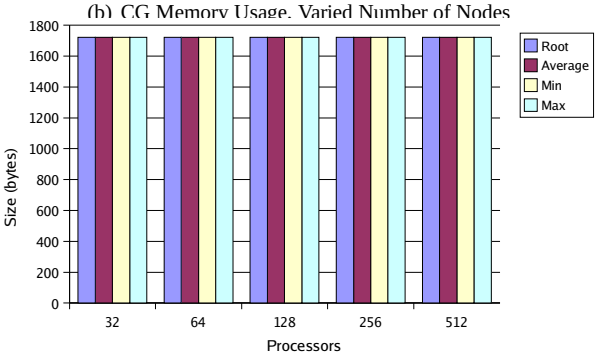
(a) CG Trace File, Varied Number of Nodes



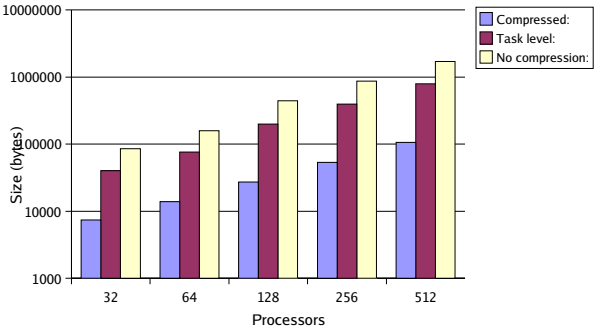
(b) CG Memory Usage, Varied Number of Nodes



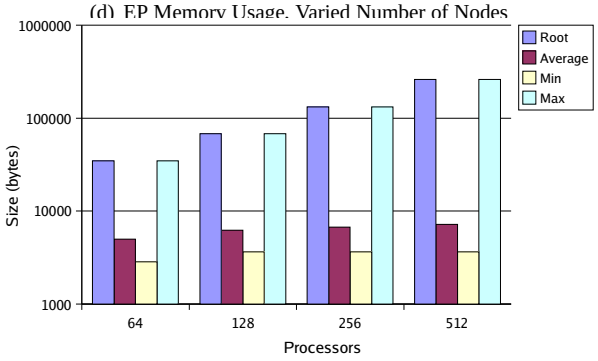
(c) EP Trace File, Varied Number of Nodes



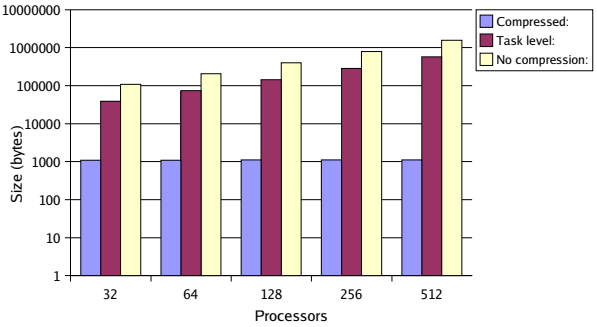
(d) EP Memory Usage, Varied Number of Nodes



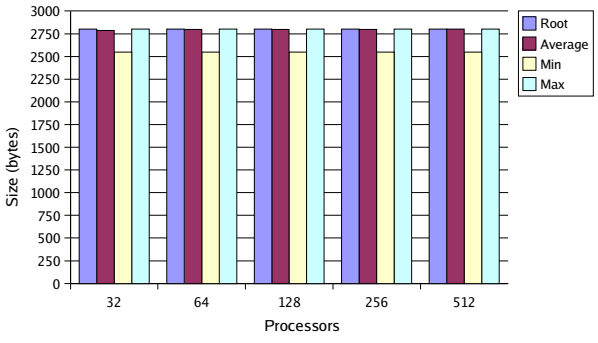
(e) FT Trace File, Varied Number of Nodes



(f) FT Memory Usage, Varied Number of Nodes



(g) IS Trace File, Varied Number of Nodes



(h) IS Memory Usage, Varied Number of Nodes

Fig. 6. NPB Trace File Size and Memory Usage per Node on BlueGene/L

even across platforms, which can benefit rapid prototyping and tuning.

The replay mechanism opens up tremendous opportunities beyond the verification of correctness. As mentioned before, it may be utilized for rapid prototyping of communication tuning as well as for 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, possibly in the context of current NSF large-scale computing infrastructure calls.

6. Related Work

RSDs have been used to describe data references in a loop [4]. The idea of PRSDs originates from on-the-fly memory trace compression [5]. 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. Furthermore, our work is the first one to utilize the structural information retained during compression, *i.e.*, our replay mechanism relies on this unique compression property.

The mpiP tool consists of a lightweight profiling library for MPI applications that collects statistical information about MPI functions, *i.e.*, aggregate metrics are reported [8]. Hence, structural information and event ordering are not preserved. There are many other tools that report aggregate information, often based of 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. 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. 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. It is the most similar existing tool to our replay mechanism. However, it does not provide the ability to replay 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 [6]. 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 the final version of our tool set.

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 even dynamic [7]. 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 promote a trace-driven approach to analyze MPI communication that scales by extracting full communication traces 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 near constant size

compression for microbenchmarks and a full-sized application. We assessed the correctness of our approach by verifying 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 near constant-size representation of MPI traces in a scalable manner combined with deterministic MPI call replay are without any precedence.

References

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

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

[3] 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.

[4] 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.

[5] 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.

[6] 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.

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

[8] 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.

[9] F. C. Wong, R. P. Martin, R. H. Arpaci-Dusseau, and D. E. Culler. Architectural requirements and scalability of the NAS parallel benchmarks. In ACM, editor, *SC'99: Oregon Convention Center 777 NE Martin Luther King Jr. Boulevard, Portland, Oregon, November 11–18, 1999*, pages ??–??, New York, NY 10036, USA and 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1999. ACM Press and IEEE Computer Society Press.