ABSTRACT

NOETH, MICHAEL J. Scalable Compression and Replay of Communication Traces in Massively Parallel Environments. (Under the direction of Associate Professor Dr. Frank Mueller).

Characterizing the communication behavior of large-scale applications is a difficult and costly task due to code and system complexity as well as the time to execute such codes. 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. Given this novel capability, we discuss its impact on communication tuning and beyond.

Scalable Compression and Replay of Communication Traces in Massively Parallel Environments

by

Michael J. Noeth

A thesis submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Master of Science in

Computer Science

Raleigh

2006

Approved By:

Dr. Xiaosong Ma | Dr. Tao Xie

Dr. Frank Mueller
Chair of Advisory Committee

To my parents: Thank you for everything.

Biography

Michael Noeth was born May 26, 1982 in Washington D.C. He received his Bachelor of Science in Computer Science from the University of Virginia in May of 2004. With the defense of this thesis, he will receive his Master's of Science in Computer Science from North Carolina State University in December of 2006.

Acknowledgements

I would like to acknowledge the following people for their support in completing my thesis: My adviser, Dr. Frank Mueller, the folks at Lawrence Livermore National Labs, Bronis R. de Supinski, Martin W. J. Schulz, and Dong Ahn, my thesis committee, Dr. Tao Xie, and Dr. Xiaosong Ma and all the graduate students in the systems lab. Their guidance and support made this thesis possible.

Contents

List of Figures **vii**

List of Tables **ix**

1 Introduction **1**
1.1 Message Passing Interface - An Overview 2
1.1.1 MPI Implementations 2
1.1.2 Profiling MPI 2
1.1.3 Stencil Codes in MPI 2
1.2 Motivation 3
1.3 Design Overview 4
1.3.1 Recording Traces 4
1.3.2 Replaying Traces 5
1.4 Paper Layout 5

2 Task-Level Compression Framework **6**
2.1 Task-Level Overview 6
2.2 Umpire's Role 7
2.2.1 Initialization Routines 8
2.2.2 Recording Routines 8
2.2.3 Cleanup Routines 9
2.3 Compression Over MPI Calls 9
2.3.1 Compression Algorithm Complexity 12
2.4 Compression Algorithm Examples 12
2.4.1 Single Terminal Example 12
2.4.2 Multiple Terminal Example 13
2.5 Cross-Node Framework Interoperability 17
2.5.1 Stack Trace Signatures 17
2.5.2 Source and Destination Offsets 19
2.5.3 Request Offsets 21
2.5.4 MPI_Waitsome Special Case 23

3 Cross-Node Compression Framework **26**
3.1 Overview 26
3.2 Tree Overlay 27
3.3 Merging Algorithm 29
3.3.1 Merge Algorithm Description 29
3.3.2 Merge Algorithm Examples 30
3.3.3 Operation Sequence Dependencies 32
3.3.4 Merge Algorithm Complexity 33
3.4 Recursive Task Participant Lists 33

4 Replay Mechanism **36**
4.1 Scanning for Task-Level Operation Queues 36
4.2 Replay Algorithm 38
4.3 Time Deltas - Simulating Computation 39

5 Experimental Results **40**
5.1 Experimental Environment 40
5.1.1 Blue Gene/L Architecture 40
5.1.2 Stencil Micro-Benchmarks 42
5.1.3 Raptor Production-Scale Codes 42
5.2 Design Decision Experiments 43
5.3 Validation Procedures 47
5.4 Performance Results 47
5.5 Replay Results 50

6 Related Work **53**

7 Conclusion **55**

Bibliography **57**

List of Figures

1.1 Interaction of recording components 5

2.1 Compression algorithm 10
2.2 Single terminal compression example 14
2.3 Multiple terminal compression example 15
2.4 Multiple terminal compression example continued 16
2.5 Stack signature example distinguishing call sites 18
2.6 Destination / source parameter compression algorithm 20
2.7 2D stencil 21
2.8 Request offset example 22
2.9 MPI_Waitsome definition [14] 24
2.10 General usage of MPI_Waitsome 25

3.1 Cross-node example 27
3.2 Tree overlay algorithm controlling merger process 28
3.3 Bit operations used on processor rank within the binary tree 28
3.4 Binary tree overlay for 20 tasks 29
3.5 Merge algorithm used to merge slave trace into master trace 30
3.6 Merge algorithm example 31
3.7 Merge algorithm dependency issue 32
3.8 Recursive task participant list 34
3.9 Logical stencil - interior nodes circled 34
3.10 Algorithm to unwind recursive task participant list 34

4.1 Comprehensive trace format 37
4.2 Algorithm to replay a compressed operation queue 38

5.1 Hierarchical layout of BG/L 41
5.2 2D stencil illustrating nine distinct communication groups 44
5.3 Task participant list - Range implementation (logarithmic scale) 45
5.4 Balanced binary tree 45
5.5 Task participant list - RSD implementation (logarithmic scale) 46
5.6 Trace file size per node on BlueGene/L 48
5.7 Memory usage per node on BlueGene/L 51

5.8 3D stencil trace file, varied time steps 51

List of Tables

2.1 Replay operation data structure 9
2.2 Uncompressed call trace 18
2.3 Naive (non structural) RSD 19
2.4 Structural RSD 19
2.5 Event sequence for request buffer example 23

Chapter 1

Introduction

An important aspect of performance analysis in any parallel environment is communication efficiency. Evaluation of communication efficiency can be approached in many ways, yet we focus on run-time analysis. Within the run-time analysis domain, one can use statistical sampling to analyze communication with low computational overhead to the application being profiled [20, 22]. On the other end of the spectrum, another approach to run-time communication analysis is to collect a complete trace of all communication. Most analysis techniques use variations on these approaches to achieve their goals trading off between paying low instrumentation costs (statistical approach) or gathering more communication information (full trace approach). This thesis explores recording a complete trace of all communication for post-run analysis as well as a distilled communication replay.

In massively parallel environments, such as IBM's Blue Gene/L (BG/L), new tools are required to digest and store the enormous amounts of data created by a full trace on a system with thousands of nodes [1] . We propose a two-tier framework to collecting traces. The first tier accumulates a queue of communication operations on a per task basis (*i.e.*at the task-level). In order to minimize memory usage as well as communication bandwidth for the second tier of the framework, the task-level framework compresses the operation queue on-the-fly. The second tier takes advantage of the "single program, multiple data" (SPMD) paradigm [7] by merging all the task-level traces into a single comprehensive cross-node trace. Our framework is able to collect traces of MPI applications on BG/L for real world applications in a scalable manner. The framework also allows for replay of the traces collected to distill communication from the rest of the application.

1.1 Message Passing Interface - An Overview

The Message Passing Interface (MPI) is a specification for a library of functions that perform communication primitives based on the message passing paradigm [14]. MPI was designed with high-performance computing (HPC), massively-parallel machines and workstation clusters as the primary target. It provides the means for multiple threads of execution working simultaneously and cooperatively using communication to solve a problem. Generally, the multiple threads of execution run on separate processors. Occasionally, more than one thread may run per processor(s) [5]. Each thread of execution is referred to as a task.

1.1.1 MPI Implementations

Since MPI is only a specification, there are multiple implementations of MPI available. Some of the most popular ones today are MPICH [11], LAM MPI [5], and vendor-specific implementations (independently developed for specific architectures or based on existing implementations to provide additional features). Slight variations between implementations can mean tools for one particular implementation may not work on another. An additional goal of this project was to create a tracing and replaying tool that would work on any MPI implementation.

1.1.2 Profiling MPI

In order to record MPI communication, a mechanism is necessary to determine what function calls are being made to the MPI library. Most MPI implementations provide a profiling layer that wraps each MPI call and allows a user to instrument additional code directly before or after the MPI call. This simple wrapping technique exposes the necessary information for lossless replay. Specifically, it exposes which function was called as well as all of its parameters.

1.1.3 Stencil Codes in MPI

MPI's basis on the message-passing paradigm makes it possible to implement stencil codes fairly easily. Our recording framework is specifically tuned to capture stencil traces.

A stencil code usually begins in a certain state with each task representing a logical

portion. Each logical portion communicates its state with neighboring portions in a step-wise fashion. After each step, logical portions update their state. Usually, this activity occurs until the system has reached a steady state (*i.e.*the logical portions are no longer changing beyond a certain threshold).

As an example, consider the Jacobi Relaxation problem. In this problem, there is a floor divided up into a grid of tiles. Each tile represents a logical portion of the problem. Each interior tile will have eight tiles surrounding it (also called nine-point stencil). Each tile begins with a different initial state (*i.e.*temperature). At each step during the problem (where a step represents the passage of time), the tiles communicate their current state (temperature) to all neighboring tiles. Using this information, the tiles update their current state (temperature) based on a convergence (heat diffusion) algorithm. Once the tiles are no longer changing state (temperature) or diverge less than some epsilon, a steady state has been reached and the Jacobi Relaxation algorithm terminates.

1.2 Motivation

Scalability is one of the main challenges to Peta-scale 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. Instead of source-code analysis, we promote a trace-driven approach to analyze MPI communication. While past approaches fail to gather full traces for hundreds of nodes in a scalable manner or only gather aggregate information, we have designed a framework that extracts full communication traces of near constant size regardless of the number of nodes while preserving structural information of the program. In addition, compressed traces can be replayed on-the-fly independent of the original application, which aids performance tuning of MPI communication and facilitates projections on network requirements for future large-scale procurements.

A complete trace of all MPI operations is advantageous to have. With a complete trace, post-processing analysis of communication is possible. A rich set of automated tools can be developed to take advantage of all the information kept in each trace. Although we have not developed any automated post-processing tools, it is clear that with a comprehensive list of all communication activity, one should be able to determine the communication patterns of the profiled application. Since our work focuses on stencil codes, we found that

some of the compression algorithms we use can be helpful in identifying specific information about the stencil. It is possible to extract the dimension of the stencil, the layout, and even the number of steps executed.

Another advantage of a complete trace is the ability to replay the communication distilled from the rest of the application. Used in combination with other profiling tools, we can determine where communication bottlenecks may be occurring. In addition, the ability to replay an application was of interest on BG/L to investigate task layout configurations in Miranda [6]. Based on the task layout, performance results varied widely and the replay tool would prove useful in identifying a model for improved task layout.

1.3 Design Overview

Our trace-driven approach to analyzing MPI communication is based on two tools that record and replay traces. The recording tool is relinked into an application to replace the MPI library. All of the MPI calls are still performed, but additional code has been inserted to create a communication trace. The replay tool uses traces generated by the recording tool to replay MPI communication.

1.3.1 Recording Traces

Recording traces is accomplished through a wrapper generator that allows us to instrument an arbitrary MPI implementation with additional code. In the wrappers, we trace which MPI function was called along with call parameters for each task. As each call is traced, we perform on-the-fly compression. Upon application termination, inter-node compression is triggered over all tasks resulting in a single trace that preserves structural information suitable for the replay tool.

Figure 1.1 depicts the recording framework. Once an MPI application is re-linked with our library, all MPI calls are intercepted by the MPI function wrapper. At this point, the MPI function and associated parameters are recorded by the task-level framework and compressed on-the-fly. The MPI function wrapper also makes a call to the actual MPI library so the communication function is invoked. At the termination of the application, each task has its own operation queue. These queues are compressed into a single trace file by merging them together in the cross-node framework. The result is a single comprehensive operation queue.

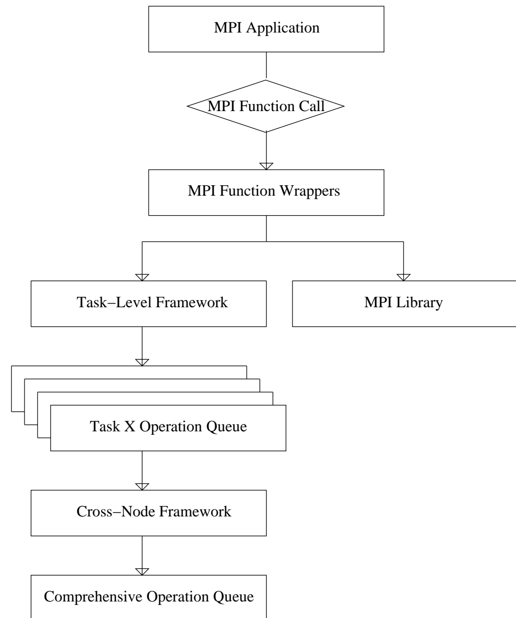


Figure 1.1: Interaction of recording components

1.3.2 Replaying Traces

The comprehensive operation queue generated by the recording tool is used to replay and analyze an application's distilled MPI communication. The entire trace file is read into memory so disk reads do not affect performance analysis. The trace is replayed by unwinding the compressed traces on-the-fly using a decompression algorithm.

1.4 Paper Layout

This thesis is structured as follows. Chapter 2 presents the task-level framework design. This includes discussion of design, compression algorithms, and other special considerations necessary for interoperability between the task-level framework and the cross-node framework. Chapter 3 discusses the design details of the cross-node framework. Chapter 4 discusses how the replay mechanism works with traces created by the recording tool. Chapter 5 discusses validation procedures, experiments to decrease storage and memory requirements, and experiments with real applications. Chapter 6 cites related work and differentiates our system from contemporary projects. Finally, Chapter 7 summarizes the work.

Chapter 2

Task-Level Compression Framework

The objective of this work is to capture an application's communication patterns for performance analysis and replay purposes. We perform two phases in order to capture a compressed trace of an application's MPI communication. The first phase is at the task-level where MPI calls are intercepted and recorded. The second phase of compression merges each task-level trace into a single comprehensive trace. This chapter describes the process of tracing MPI calls at the task-level.

2.1 Task-Level Overview

Recording the MPI communication of an application at the task-level is trivial if allotted unlimited storage and memory resources. It would simply consist of noting what communication calls were made and the associated parameters for each task. For c communication calls, each task would have c records. The number of communication calls is closely associated with an application's complexity, but it is suffice to say that this methodology would result in potentially enormous trace files.

In order to mitigate the size of task-level traces, we perform compression over the MPI communication operations performed by each task. Since there are usually very regular patterns (mostly loops) in which MPI communication is performed, this method drastically reduces the size of the individual trace. Better compression at the task-level is possible using bzip compression [17] but the algorithm used does not maintain any structural information

about the trace. Without additional information about the trace, further compression is not possible. Thus, the second phase of capturing MPI communication across nodes becomes impossible using this type of algorithm. Further discussion of why the second phase is necessary is discussed in the following chapter.

Each processor participating in the execution of an MPI parallel program performs a series of communication functions to accomplish its goal. The task-level compression framework captures these communication function calls using Umpire's [21] MPI wrapper generator and a specification that instructs Umpire how to instrument each call. The specification is expressly written to accumulate which call was made and what the parameters were at the time of the call. Each task collects a trace in the task-level compression framework. Initial traces were taken on a few MPI applications written for BG/L and the sizes of the files generated ranged from fifty to one hundred gigabytes per task. The current approach would be un-scalable on BG/L when attempting to use its full 64 thousand processors (hundreds of terabytes of storage would be required). We use compression over MPI calls to decrease the size of the trace files at the task-level and mitigate the communication needs when merging the traces in the second phase of capturing a single comprehensive trace.

2.2 Umpire's Role

Umpire was originally designed to provide run time debugging information for MPI applications in the form of deadlock detection, resource exhaustion, etc. In order to perform these debugging features, Umpire generates wrapper functions for all MPI calls within an MPI implementation (Umpire is portable and can be used on any MPI 1.2 implementations). The wrappers are used to collect information about which calls are being made, increment counters, and allows for many of the run-time debugging features to perform their task of detecting specific programming errors.

In the task-level compression framework, Umpire is used to provide hooks into an arbitrary MPI implementation's function calls. This is a very useful feature which allows the task-level compression framework to record all MPI communication. The task-level compression framework doesn't need any of the error detection capabilities that Umpire provides but rather the portable wrapper generation feature. By using Umpire to generate the function wrapper calls we are able to perform traces on any MPI implementation. All

the extraneous error detection code was stripped out of Umpire to simplify the task-level compression framework as much as possible.

For extensibility purposes, Umpire uses a specification to instruct the wrapper generator on what code to insert before and after an MPI call is made. All debugging insertions were removed and replaced with simple trace recording code. There are three classes of recording code: (1) initialization routines, (2) recording communication routines, and (3) cleanup routines.

2.2.1 Initialization Routines

Since every MPI application calls MPI_Init before any other MPI call, this call was instrumented with additional initialization routines. During initialization, the following tasks are performed by parsing an input file:

- Determine the output format (we currently support uncompressed ASCII, uncompressed binary, and a compressed ASCII format).
- Setup and allocate memory for all data structures necessary for the output format selected.
- Open a file for the trace to be written to.
- Record important global information such as the task MPI rank and the actual value of MPI_COMM_WORLD (this is useful in the event that someone wishes to replay traces on a different machine or MPI implementation than the recording was performed). Other control information is also initialized.

2.2.2 Recording Routines

The routines that record call and parameter information of MPI calls perform their tasks based on the output format determined by the initialization routine. In all output formats, a single basic objective is accomplished: record what MPI call was made and the parameters used for the call. In order to minimize the number of calls profiled, the task-level compression framework only records MPI calls that perform non-informational work. Thus, calls such as MPI_Comm_rank, MPI_Comm_size, and MPI_Type_size are not recorded because they will not affect the replay if left out. These calls are informational, and no communication or state changes to the MPI state machine are performed.

Table 2.1: Replay operation data structure

Call / meta info		**Parameter info**	
data type	*name*	*data type*	*name*
int	sync_bytes	int	comm
int	seq_num	int	count
short int	op	int	dest
int	arrays	int	source
		N/A	ETC

In the uncompressed output formats, the call and parameter information is written directly to file. In the compressed ASCII output format, the information is stored in the data structure depicted in Table 2.2.2 to be later compressed into a single trace file. A queue of these data structures is kept in memory, which the compression algorithm is run on to compress any possibly repeated calls.

2.2.3 Cleanup Routines

The final class of routines are for cleanup purposes. The cleanup routine is inserted into MPI_Finalize since it should be the final MPI call made in an MPI application. This routine serves to clean up by releasing memory allocated for data structures. It also serves as the starting point for merging individual task-level traces into a single comprehensive trace.

2.3 Compression Over MPI Calls

Compression takes advantage of patterns to represent a list of terminals using regular section descriptors (RSDs) and power regular section descriptors (PRSDs) [13]. An RSD can be used to describe a pattern recognized in a sequence of terminals. To further our compression capabilities, PRSDs can represent patterns of RSDs and terminals in a similar fashion. For the purposes of our task-level compression framework, we are most interested in compressing operation terminals (a specific MPI operation and its associated parameter list) into a series of terminals, RSDs and PRSDs.

The algorithm compresses the queue of operations recorded as each operation is appended onto the tail of the queue. After an operation is appended onto the tail of the queue, the algorithm in Figure 2.1 is run. It is a greedy algorithm that attempts to match

```
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
```

Figure 2.1: Compression algorithm

the first sequence it finds based on a memory analysis compression algorithm called SIGMA [15]. There are four distinct steps the algorithm goes through to attempt to compress a newly appended operation terminal:

Step 1: Compression Target Identification

This step's purpose is to identify a sequence of operation terminals, RSDs, or PRSDs by designating the head (where the sequence begins in the queue) and the tail (where the sequence ends in the queue). Identification of the tail is trivial; the end of the queue is always the compression target's tail. The head of the compression target is found by searching backwards through the queue starting from the tail. The algorithm is trying to identify a matching element for the compression target's tail. Once a match has been found, we set the head of the compression target to be the next element in the queue. If no match is found, compression is not possible and the algorithm discontinues. Conversely, if a match is found, the queue is then searched for a sequence that matches the compression target.

In the event that a long sequence of events is in the queue, this step can become quite costly. An example of this situation occurs when the algorithm continuously fails to compress any of the operations. The queue continues to grow and becomes more and more costly to search. To mitigate the effects of long uncompressed sequences, a window can be specified in the input file. The window specifies how far back in the queue this step can go before the algorithm decides that it cannot find a matching element. A warning is issued by the task-level framework to indicate that additional compression may be possible whenever the window is reached and the search is halted. Only a warning is issued since the resultant trace is semantically the same for replay purposes although it may have the potential for further compression.

Step 2: Merge Target Identification

This step's purpose is to identify a sequence of operation terminals, RSDs, or PRSDs that match the compression target. Identification of the tail is trivial; from the last step, the algorithm already knows a potential matching tail (the element right before the compression target's head). A search is commenced from the merge target's tail to identify a head that matches the compression target's head. On a match, the algorithm continues on to the next step. If no match is found, compression of the compression target identified by the previous step is not possible. The algorithm goes back to the previous step to identify another potential sequence of operations as the compression target.

Step 3: Match Verification

This step's purpose is to ensure that all elements of the compression target and the merge target match. Two iterators begin at the tails of each sequence that ensure each element between the head and the tail match. In addition to matching operations and parameters, loop counters (referred to as looping information) are also embedded into already existing sequences and must also be matched. The head of each sequence is a special case in which looping information does not necessarily have to match. If the algorithm attempts to merge a sequence into an existing RSD or PRSD, the heads of the two sequences do not have to have perfectly matching loop information for a merge to take place. If any elements of the sequences do not match (beside the heads), compression of the current

compression target is not possible and the algorithm goes back to the first step to try and identify another compression target.

Step 4: Compression

This step's purpose is to perform the actual compression now that a compression target and merge target have been identified as valid. The head of the merge target has the necessary loop information encoded (this information includes an iteration count and a pointer to the end of the sequence). If the merge target's head did not have loop information encoded before, an identifier for the merge target's tail is recorded and the number of loop iterations is set to two. If the merge target's head already had loop information encoded, the algorithm finds the proper loop level and adds another iteration. The final step is to delete all the elements from in the compression target's sequence from the end of the operation queue. Since they are encoded in the merge target's head as an additional iteration, these elements are no longer necessary.

2.3.1 Compression Algorithm Complexity

The algorithm's complexity is based on the number MPI operations traced, T. For T operations, the algorithm must traverse through the current operation queue, which could be as long as T. To avoid a O(T^2) algorithmic complexity, a window of size W, can be specified, which limits how far from the tail of the queue the algorithm will search back. When the window is in use, the algorithmic complexity becomes O(TW).

2.4 Compression Algorithm Examples

To illustrate how compression works in the task-level framework, two simple examples will be presented. For simplicity's sake, 'op#' will represent a terminal. If the '#' matches, then the terminals match. The first scenario is merging a single terminal into an RSD. The second scenario is merging multiple terminals into an RSD.

2.4.1 Single Terminal Example

For the first example, to illustrate the single terminal case, we examine the operation stream, op1 op2 op3 op3 op3. It is apparent that op3 is the single terminal we are

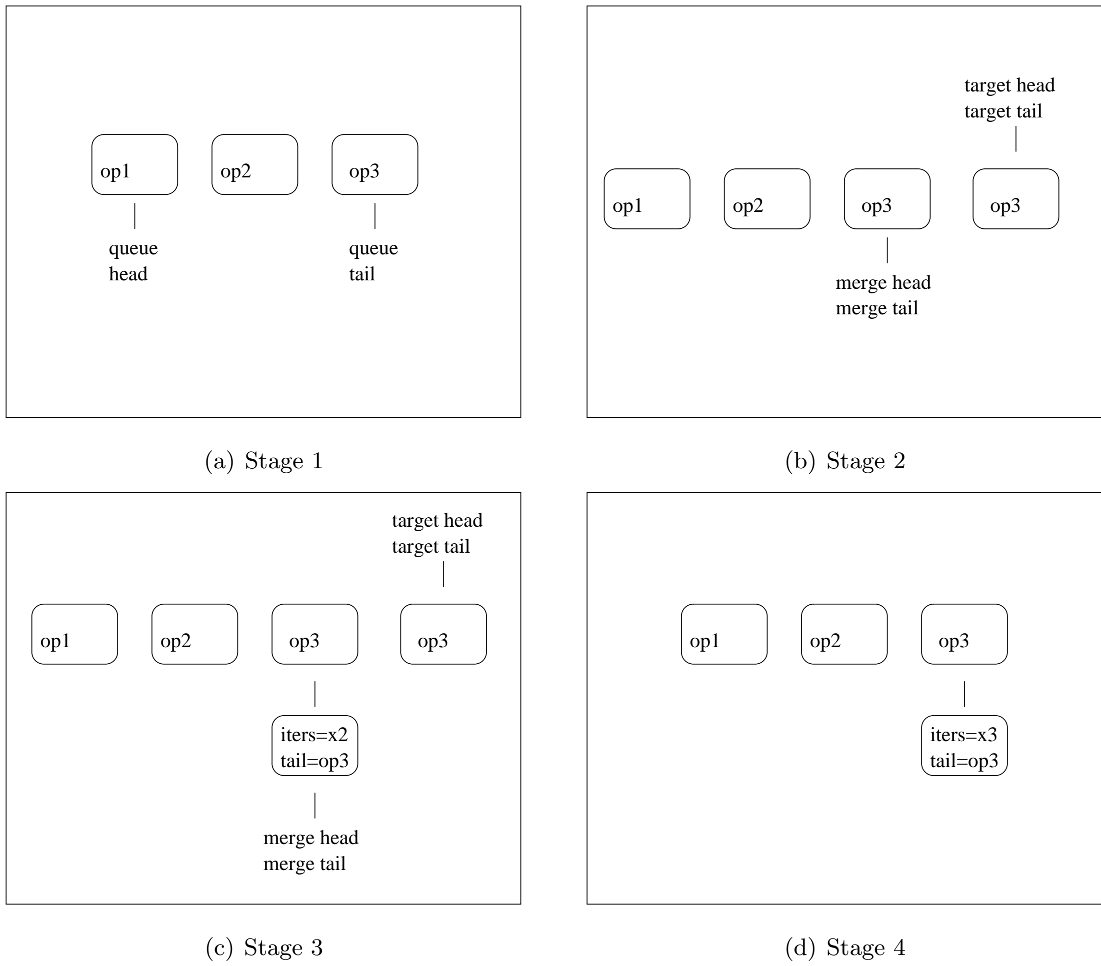
attempting to merge. The following description explores how the algorithm compresses the single terminal stream step by step:

- Stage 1 - When op1, op2, and op3 are appended onto the end of the queue, there are no matches, and no compression is performed (step 1 fails). Figure 2.2(a) depicts the state of the queue after these operations have been processed.
- Stage 2 - When op3 is appended onto the end of the queue, a match is found and step 1 of the algorithm sets the head and tail of the target. A matching merge target is identified in step 2 of the algorithm. Figure 2.2(b) depicts the state of the queue after step 2 of the algorithm. Step 3 compares the merge target and compression target to verify the single terminal being compressed matches.
- Stage 3 - The 1st and 2nd op3 have been compressed. Additional information regarding the compression is attached to the former merge target's head as depicted in Figure 2.2(c). A 3rd op3 is added and a match is found. The sequence of events in Stage 2 is repeated as depicted in Figure 2.2(c).
- Stage 4 - The final state at the end of the compression of the operation stream is depicted in Figure 2.2(d). Note that when the 3rd op3 was compressed, the loop information is simply adjusted rather than creating new loop information as in Stage 3.

2.4.2 Multiple Terminal Example

For the second example, to illustrate the multiple terminal case, we examine the operation stream op1 op2 op3 op4 op5 op3 op4 op5. The following description explores how the algorithm compresses the multiple terminal case stream step by step:

- Stage 1 - When op1, op2, op3, op4 and op5 are appended onto the tail of the queue, there are no matches and no compression is performed (step 1 fails). See Figure 2.3(a) for the state of the operation queue.
- Stage 2 - When op3 is appended to the tail of the queue, we find a match and the target head is set to op4. The merge tail is trivially set but we cannot find a merge head (step 2 fails). Figure 2.3(b) depicts where the three pointers (target tail, target head, and merge tail) are set.



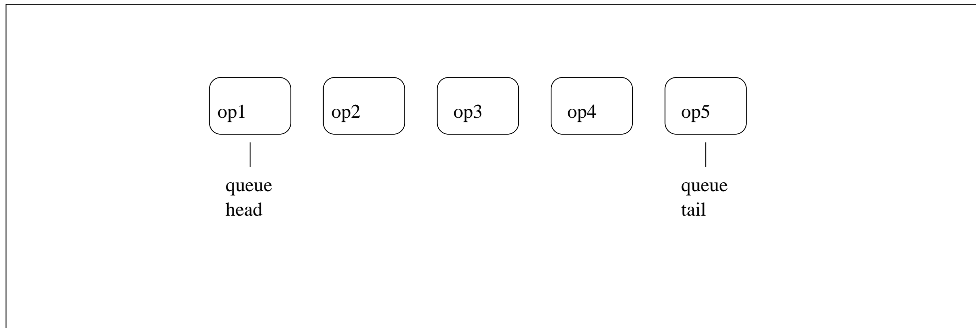
(a) Stage 1

(b) Stage 2

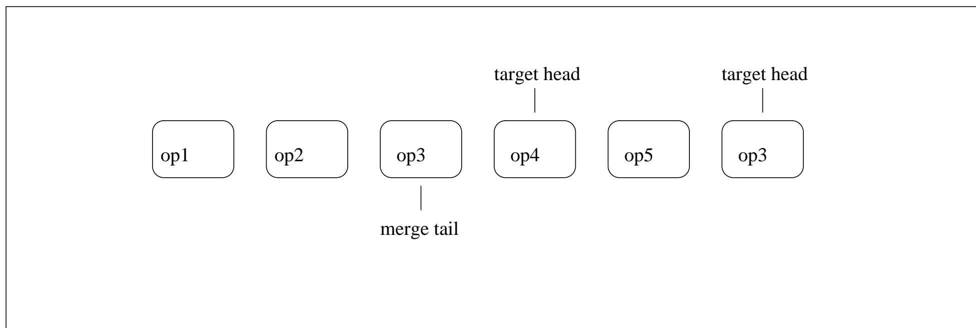
(c) Stage 3

(d) Stage 4

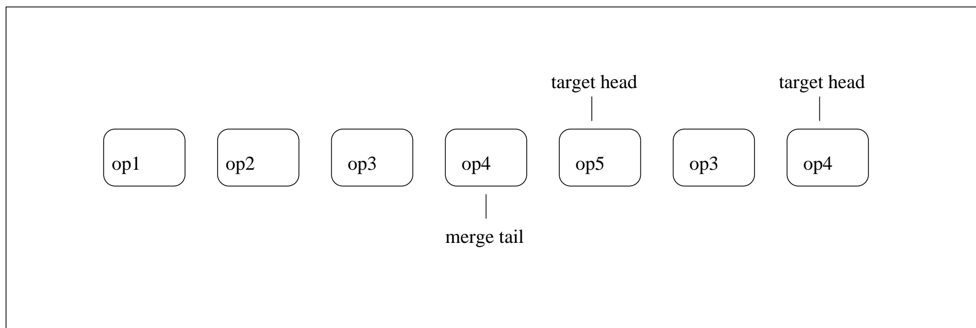
Figure 2.2: Single terminal compression example



(a) Stage 1

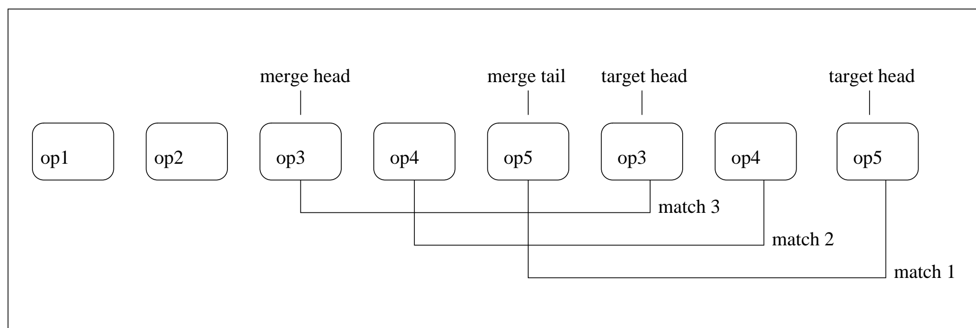


(b) Stage 2

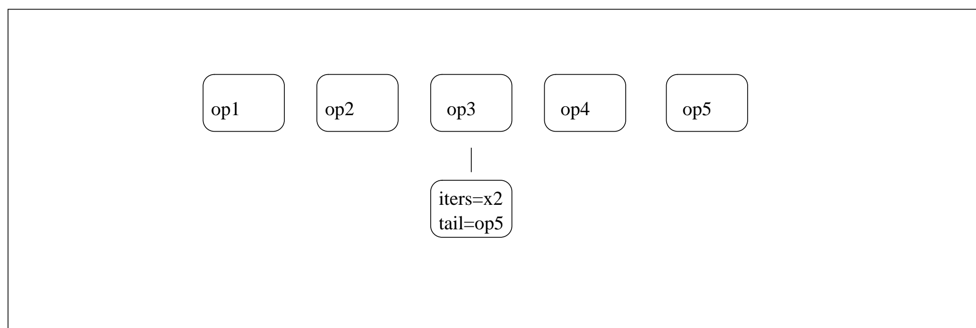


(c) Stage 3

Figure 2.3: Multiple terminal compression example



(a) Stage 4



(b) Stage 5

Figure 2.4: Multiple terminal compression example continued

- Stage 3 - When op4 is appended to the queue, we find a match and a similar failure to Stage 2 occurs (step 2 fails). See Figure 2.3(c) for the state of the operation queue.
- Stage 4 - When op5 is appended, we find a match and the target head is set. We are also able to find a matching merge head. A scan over each element shows that all match. Figure 2.4(a) depicts how the algorithm works from right to left scanning for matches.
- Stage 5 - After compression occurs, we arrive at the final state of the operation queue. Note that the repeated op3, op4, and op5 in 2.4(a) were deleted from the end of the queue and are replaced with the RSD definition below op 3 in Figure 2.4(b).

2.5 Cross-Node Framework Interoperability

Additional features and components were necessary for the task-level framework to provide traces amiable to the cross-node framework (which merges the traces into a single trace). Structural information is necessary to take advantage of SPMD nature of MPI programs [7] during the merging phase. To maintain structural information, a stack trace signature was added to each operation to distinguish it from similar MPI operations at a different location in the code. Source and destination offsets and request offsets were also added to enable the cross-node framework to merge similar operation sequences with differing parameters. Due to the nondeterministic nature of the function, MPI_Waitsome, a special method of tracing this call was also added.

2.5.1 Stack Trace Signatures

In the task-level framework, a naive approach to compression may simply attempt to match each part of an operation terminal (*i.e.* what MPI call it was and the parameters). On a match, the algorithm would attempt to compress the new operation terminal into an RSD. In order to achieve a better chance of compression when merging across nodes, we perform a more selective approach that reduces the amount we can compress at the task-level but allows us to keep structural information necessary for the cross-node framework.

We use a stack trace to generate a unique signature for each MPI function called. By adding this additional matching metric to the compression algorithm, we are able to keep structural information in our compressed traces.

```
void bar()
    call foo() // call site 1

void foo()
    call MPI_Call_1(param list 1) // call site 2

void main()
    ...
    call MPI_Call_1(param list 1) // call site 3
    call foo() // call site 4
    call bar() // call site 5
    for(i = 0 to 5)
        call MPI_Call_1(param list 1) // call site 6
    ...
```

Figure 2.5: Stack signature example distinguishing call sites

Table 2.2: Uncompressed call trace

Sequence Num	**Operation**	**Parameters**	**Call site**
0	MPI_Call_1	param list 1	3
1	MPI_Call_1	param list 1	4,2
2	MPI_Call_1	param list 1	5,1,2
3	MPI_Call_1	param list 1	6
4	MPI_Call_1	param list 1	6
5	MPI_Call_1	param list 1	6
6	MPI_Call_1	param list 1	6
7	MPI_Call_1	param list 1	6

Consider the example in Figure 2.5. In this example code, every MPI call made is the same call with the same parameter list (*i.e.* they match naively). Table 2.2 illustrates an uncompressed trace of the calls made when the code in Figure 2.5 is run. If we use naive matching, we can compress all eight calls into a single RSD represented in Table 2.3 but we lose structural information necessary for the cross-node compression framework. We find it more useful for merging traces to maintain structure and distinguish between calls based on their stack trace (equivalent to the 'call site #' in Figure 2.5). The structural information allows us to attempt to find regular expressions to represent the parameter lists better. Table 2.4 shows the compressed trace with maintained structural information.

In order to implement the stack signature used to distinguish between similar MPI calls, a stack walk is performed in the Umpire wrapper of each MPI call. The stack walk

Table 2.3: Naive (non structural) RSD

RSD 1
Op = MPI_Call_1
Params = list 1
Iterations = 8

Table 2.4: Structural RSD

RSD 1	RSD 2	RSD 3	RSD 4
Op = MPI_Call_1	Op = MPI_Call_1	Op = MPI_Call_1	Op = MPI_Call_1
Params = list 1	Params = list 1	Params = list 1	Params = list 1
Stack trace = 3	Stack trace = 4,2	Stack trace = 5,1,2	Stack trace = 6
Iterations = 1	Iterations = 1	Iterations = 1	Iterations = 5

records the call location (rather than the return address) and concatenates each location in a string to generate a unique identifier. The memory addresses recorded in the string are separated by an underscore character. Using this methodology we are able to distinguish sequence 1 and sequence 2 as separate calls in Table 2.2. Alternatively, a simple line number and file name was considered as a unique signature but was rejected since it would not be able to perform the aforementioned distinction: the final call site is 2 and, thus, they would match.

Since a string compare is necessary to determine if two separate calls match, we also keep track of an XOR signature. The XOR signature is an integer which XORs every memory address encountered on the stack walk. Thus, a quick check to see if the XOR signatures match can be made before calling the more costly string compare function. An XOR signature match alone is not enough to accept a match, but it is enough to reject one. This is because a single XOR signature can represent multiple full string signatures. Due to proximity in memory addresses, this occurs with a far higher probability than one might intuitively expect.

2.5.2 Source and Destination Offsets

As an additional measure to achieve a better chance of compression when merging across nodes, we have implemented a method of tracking differences in parameters within the same MPI call. Currently, this additional measure of compression is only performed across MPI_Send and MPI_Recv calls on the destination and source parameters, respectively. If only the destination or source parameter differ during the verify step of compression, a list

```
if only [destination, source] parameter do not match
    CASE 1: merge_iter & target_iter do not have offset lists
        offset1 = rank - merge_iter->parameter
        offset2 = rank - target_iter->parameter
        create an offset list encoded on merge_iter containing offset1 & offset2

    CASE 2: merge_iter has an offset list and target_iter does not
        offset1 = rank - target_iter->parameter
        add offset1 to merge_iter offset list

    CASE 3: merge_iter & target_iter have offset lists
        if both lists match: normal compression occurs
        else discontinue compression
```

Figure 2.6: Destination / source parameter compression algorithm

of parameters is created and maintained. In order to achieve cross-node compression, we track the offset of the destination or source parameter from the task's MPI_COMM_WORLD rank. The algorithm for parameter compression is shown in Figure 2.6. During the verify step of compression (see step 3 of the compression description), when the parameters are checked to see if they match, if only the destination or source do not match, than the parameter offset algorithm is run.

Consider communication with neighbors following a stencil paradigm. Stencil communication is usually performed many times over loops. Since the same stencil communication is used, 'CASE 3' in Figure 2.6 will discontinue parameter compression if it detects that the stencil has begun to repeat itself. This allows us to keep the offset lists relatively short and justifies the simplistic approach to parameter compression.

Parameter compression targeting MPI_Send and MPI_Recv calls seeks to take advantage of stencil codes. Generally, when point to point communication is used in an MPI application, a pattern is present. This pattern usually has all tasks participating in the SPMD paradigm communicating with neighboring processors. We have observed that the communication with neighbors can be captured by the rank offset of the parameter. Thus, multiple tasks can share a common sequence by comparing the parameter offsets.

Consider the two-dimensional stencil depicted in Figure 2.7 as an example of how multiple tasks can share a common sequence of events. The left grid displays task 9's view of the logical 2D space while the right grid displays task 10's view of the logical 2D space.

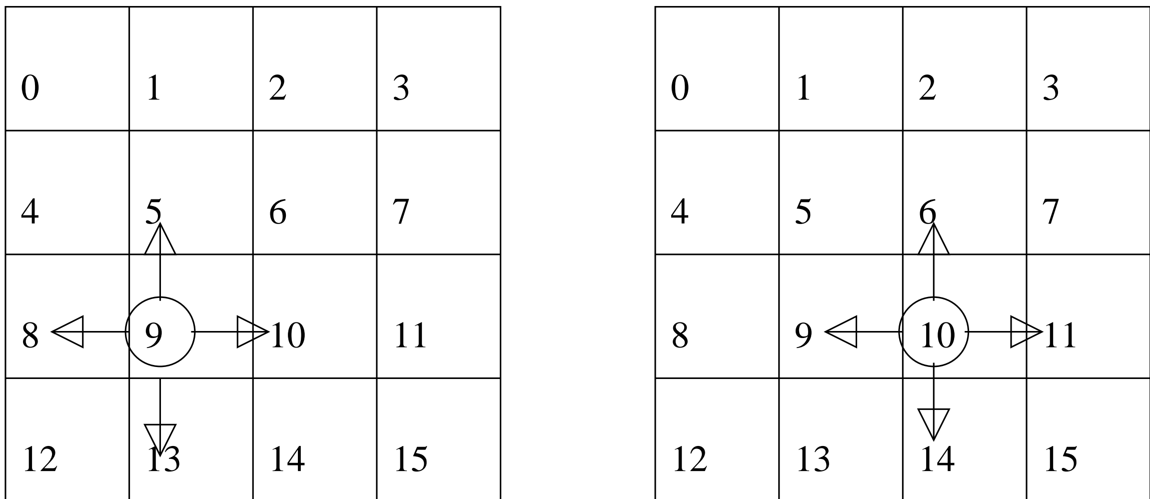


Figure 2.7: 2D stencil

Both tasks are performing a five-point stencil beginning at their left (task 9 communicates with task 8 first and task 10 communicates with task 9 first). Communication is depicted by an arrow and it iterates in a clockwise fashion. Based on this premise, task 9 communicates with tasks 8, 5, 10, and 13 while task 10 communicates with tasks 9, 6, 11, and 14 (in that order). If we replace the targets of communication with offsets, both tasks 9 and 10 communicate with tasks -1, -4, +1, and +4 relative to their rank. Thus, when we move into the cross-node compression, both tasks 9 and 10 can use the same compressed sequence of events which increases compression.

2.5.3 Request Offsets

Request parameters can have similar effects to the destination and source parameters during the cross-node framework's merging process. Request handles are used during asynchronous communication to indicate when certain events have completed or to wait on those events. Thus, a request handle is generally used twice: once to associate it with an asynchronous event, and once to determine if that event has occurred (although the second usage is not necessary). When a request is associated with an asynchronous event, it is assigned an integer handle that the MPI implementation uses to identify the event.

The problem is that request handles are non-deterministically assigned integers. For example, if task 0 and task 1 both execute the code listed in Figure 2.8, the cross-

```
MPI_Request sreq_list[NUM_REQS]
MPI_Request rreq_list[NUM_REQS]
for i = 0 to NUM_REQS
    MPI_Irecv(..., rreq_list[i])
    MPI_Isend(..., sreq_list[i])
    // perform computation
    MPI_Wait(rreq_list[i])
    MPI_Wait(sreq_list[i])
```

Figure 2.8: Request offset example

node framework should be able to merge the sequence of events as the same. This is not always the case since the request parameter for the MPI_Isend and MPI_Irecv may generate different integers as handles. Thus, the cross-node framework would not be able to merge the sequences.

This problem is handled with a circular request buffer. Each task allocates an array of request handles (the size is specified in an input file and is loaded during the initialization stage). A size variable (representing the size of the request buffer) and a current variable (representing the current position in the request buffer) are tracked throughout the execution of the task-level framework. When a request is associated with an asynchronous event, such as MPI_Isend, the request handle is stored in the request buffer at the current position and the current variable is incremented. It is not necessary to track the request handle further (*i.e.* with the operation's parameter data) since a similar mechanism will be used in the replay framework to reconstruct the communication (described in detail later).

When a request handle is used for a second time (*i.e.* MPI_Wait, MPI_Test, etc), rather than recording the actual handle with the operation, we record its offset from the current variable in the request buffer. This allows for the MPI implementation to assign arbitrary request handles across tasks but still allows the cross-node frame work the ability to merge similar event sequences.

As an example, assume that we are observing the behavior of task 0 and task 1 executing the code in Figure 2.8. Table 2.5 depicts the resultant sequence of events for task 0 and task 1. If we assume that m does not equal n, then if we were to simply record the request handles, the cross-node frame work would be unable to merge these two sequences. When using the request buffer, on the first iteration, the send and receive wrappers, which capture parameters, ignore the request handles (they are recorded in the request buffer

Table 2.5: Event sequence for request buffer example

t0 Operation	t0 Request Handle	t1 Operation	t1 Request Handle
MPI_Isend	m	MPI_Isend	n
MPI_Irecv	m + 1	MPI_Irecv	n + 1
MPI_Wait	m	MPI_Wait	n
MPI_Wait	m + 1	MPI_Wait	n + 1
MPI_Isend	m + 2	MPI_Isend	n + 2
MPI_Irecv	m + 3	MPI_Irecv	n + 3
MPI_Wait	m + 2	MPI_Wait	n + 2
MPI_Wait	m + 3	MPI_Wait	n + 3
...	...	...	...
MPI_Isend	m + (NUM_REQS - 1)	MPI_Isend	n + (NUM_REQS - 1)
MPI_Irecv	m + NUM_REQS	MPI_Irecv	n + NUM_REQS
MPI_Wait	m + (NUM_REQS - 1)	MPI_Wait	n + (NUM_REQS - 1)
MPI_Wait	m + NUM_REQS	MPI_Wait	n + NUM_REQS

instead), and the wait wrapper record offsets -2 and -1 for both tasks. Each subsequent iteration would also result in offsets -2 and -1 for both tasks. This allows the cross-node framework the ability to merge the similar sequences.

The reason the buffer is circular is that a wrap around occurs when the end of the buffer is reached. The idea behind this is that the buffer need only be as large as the number of active requests. Once a request has been looked up, it is no longer going to be referred to by that handle as per the MPI specification. Thus, we can overwrite the value without a problem. If the number of active requests does exceed the buffer space allocated, when the second usage of a request handle occurs and the task-level framework attempts to look up its offset, it will fail. An error message is shown to the user indicating they must allocate additional request buffer space to rectify the problem.

2.5.4 MPI_Waitsome Special Case

The function call MPI_Waitsome fails to compress well in the cross-node framework akin to the point-to-point and the asynchronous calls previously described. The failure to achieve good compression stems from the common problem of mismatched parameters across tasks. An output parameter for the MPI_Waitsome call, which depends on how many asynchronous operations completed, is the mismatched parameter in this case. Due to the nondeterministic nature of asynchronous operations, a call to MPI_Waitsome by one task can result in a different output parameter than another task attempting to perform the

same functionality. As a result, a special method of tracing MPI_Waitsome was necessary for improved cross-node merging.

MPI_WAITSOME(
incount, **array_of_requests**, **outcount**,
array_of_indices, **array_of_statuses**
)
[IN incount] length of array_of_requests (integer)
[INOUT array_of_requests] array of requests (array of handles)
[OUT outcount] number of completed requests (integer)
[OUT array_of_indices] array of indices of operations that completed (array of integers)
[OUT array_of_statuses] array of status objects for operations that completed (array of Status)

Waits until at least one of the operations associated with active handles in the list have completed. Returns in outcount the number of requests from the list array_of_requests that have completed. Returns in the first outcount locations of the array array_of_indices the indices of these operations (index within the array array_of_requests; the array is indexed from zero in C and from one in Fortran). Returns in the first outcount locations of the array array_of_status the status for these completed operations. If a request that completed was allocated by a nonblocking communication call, then it is deallocated, and the associated handle is set to MPI_REQUEST_NULL.

Figure 2.9: MPI_Waitsome definition [14]

To understand the problem, we must consider the definition of MPI_Waitsome in Figure 2.9. MPI_Waitsome takes an array ”array_of_requests” of length ”incount” and blocks until any of the asynchronous requests in ”array_of_requests” are completed. By definition, asynchronous communication is nondeterministic. Since the output parameter ”outcount” is dependent on nondeterministic communication, it is also nondeterministic. Thus, the parameter can vary across tasks depending on how many of the asynchronous requests have been satisfied.

```
int total_count = 0
math(while(total_count < N))
    int outcount
    MPI_Waitsome(... outcount ...)
    total_count += outcount
    Process data on completed requests...
```

Figure 2.10: General usage of MPI_Waitsome

To counter the nondeterministic nature of MPI_Waitsome, we take advantage of the function's semantics. MPI_Waitsome is a synchronization primitive that allows overlapping communication and computation for better performance. When multiple asynchronous point-to-point calls have been issued, it is possible to begin processing data when any of the requests have been satisfied. Figure 2.10 depicts MPI_Waitsome's general usage. Because most applications adhering to the SPMD programming paradigm will have a common loop bound "N", it is possible to exploit this fact for better compression.

Based on the general usage of MPI_Waitsome, a special handler in the compression algorithm at the task-level is used. Upon encountering a call to MPI_Waitsome, the compression algorithm delays the on-the-fly compression of the operation queue. Any subsequent MPI_Waitsome calls are coalesced by summing the "outcount" parameter. While additional MPI_Waitsome calls are combined, the compression algorithm continues to delay the on-the-fly compression. The coalescing process terminates when an MPI call other than MPI_Waitsome occurs. Before the new call is added onto the tail of the operation queue, the compression algorithm is run to compress the MPI_Waitsome call. After this completes, the compression algorithm continues as originally described by attempting to compress the MPI call into the operation queue.

Chapter 3

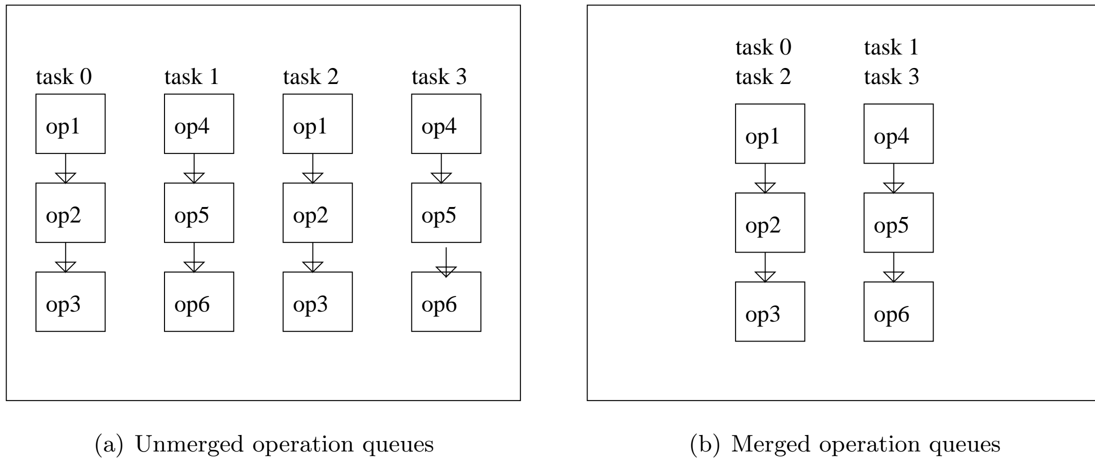
Cross-Node Compression Framework

As noted in the previous chapter, our approach encompasses two phases in order to capture a compressed trace of an application's MPI communication. The first phase captures the communication at the task-level, while the second phase merges each task-level trace into a single comprehensive trace. This is important when scaling to larger jobs with many tasks (*i.e.* BG/L can support up to 64 thousand tasks). Due to the SPMD nature of MPI programs, many of the tasks perform the same operations. The cross-node compression framework takes advantage of this by compressing all the individual traces obtained by the task-level compression into a single, condensed trace.

3.1 Overview

After an application being profiled has reached its last MPI call (MPI_Finalize), a queue containing all MPI operations executed exists in each tasks' memory. If traces were written to file at this point, we would be left with an un-scalable solution. As job sizes were increased, the amount of space necessary for the resultant trace files would grow linearly with the number of nodes (*i.e.* a 1 MB trace file with 64 thousand tasks would result in 64 GBs of data).

In order to create a task-scalable trace, we merge individual traces generated by the task-level framework. Since many of the operation sequences overlap across tasks, there is no need to replicate them in the final trace. Rather than write out the duplicated sequences,



(a) Unmerged operation queues

(b) Merged operation queues

Figure 3.1: Cross-node example

task participant lists are appended to one sequence (while the duplicates are discarded). For example, consider four tasks with operation queues depicted in Figure 3.1(a). We see that tasks 0 and 2 perform the same sequence of operations and tasks 1 and 3 also have the same sequence. It is possible to represent the operation sequences in Figure 3.1(a) in half the space using task participant lists. After merging the operation queues, we would have only two sequences, as depicted in Figure 3.1(b). This is the premise of the cross-node compression framework.

3.2 Tree Overlay

The merging algorithm that the cross-node framework uses combines two operation queues into a single queue. Generally, MPI applications have more than two tasks, so a control mechanism must guide the merging process. A balanced binary tree [12] overlay acts as that controller. Upon termination of an MPI application, each task has its own local operation queue which is passed up the binary tree towards the root.

The algorithm in Figure 3.2 is used to control the merging process. The algorithm creates three distinct levels within the binary tree: (1) leaf nodes, (2) interior nodes, and (3) the root node. Leaf nodes do not perform any merging since they have no children to receive operation queues from; they immediately send their local queue to their parents (the checks for a left or right child fail in Figure 3.2). Interior nodes (all nodes that are not leaf nodes nor the root node) begin receiving queues from their children and merge those with

```
if left child exists
    recv left child's queue
    merge left child's queue into local queue
if right child exists
    recv right child's queue
    merge right child's queue into local queue
if rank == root
    write local queue to file
else
    send local queue to parent
```

Figure 3.2: Tree overlay algorithm controlling merger process

their own local queue. Once an interior node has merged both its children's queues into its local queue, the resultant queue is sent to its parent. This process is repeated until all queues are merged at the root node. After the merge at the root node is complete, there are no more queues that have to be merged and the complete trace is written to file.

A balanced binary tree works best for this process because it provides load balancing implicitly. The merging and sending are the most costly part of cross-node compression. Since the tree is balanced, each level in the tree performs approximately the same amount of work in parallel. If the tree were unbalanced, the side with more nodes would become a bottleneck.

```
Left Child
    given RANK
    set MSB + 1 = 1
    set MSB = 0
Right Child
    given RANK
    set MSB + 1 = 1
    set MSB = 1
Parent
    given RANK
    set MSB - 1 = 1
    set MSB = 0
```

Figure 3.3: Bit operations used on processor rank within the binary tree

The tree overlay itself is generated based on processor rank as shown in Figure 3.3. MSB stands for most significant bit. To determine the rank of a task's child, the MSB

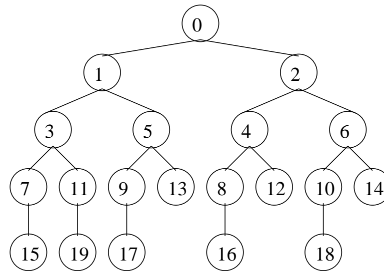


Figure 3.4: Binary tree overlay for 20 tasks

of its local MPI rank is incremented. From there, the left child is determined by cleaning the MSB while the right child is determined by setting the MSB. To determine the parent, the inverse operation is performed. The MSB is unset, while the MSB - 1 is set. The tree overlay is always the same except that it continues to grow depending on how many tasks are participating within a job. A tree overlay for 20 tasks is depicted in Figure 3.4.

3.3 Merging Algorithm

The tree overlay controls the merger of operation queues. The actual merger of two queues is described in this section. The process specifies one of the queues as a master and the other as slave. The slave queue is merged into the master queue after which the slave queue is discarded.

3.3.1 Merge Algorithm Description

The algorithm used to accomplish merging is shown in Figure 3.5. Two operation queues are provided to the algorithm as parameters: a master queue and a slave queue. The slave queue is merged into the master queue by identifying matching sequences of operations. To identify a match, three iterators are used: (1) a master iterator, (2) a slave head, and (3) a slave iterator. 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 se-

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

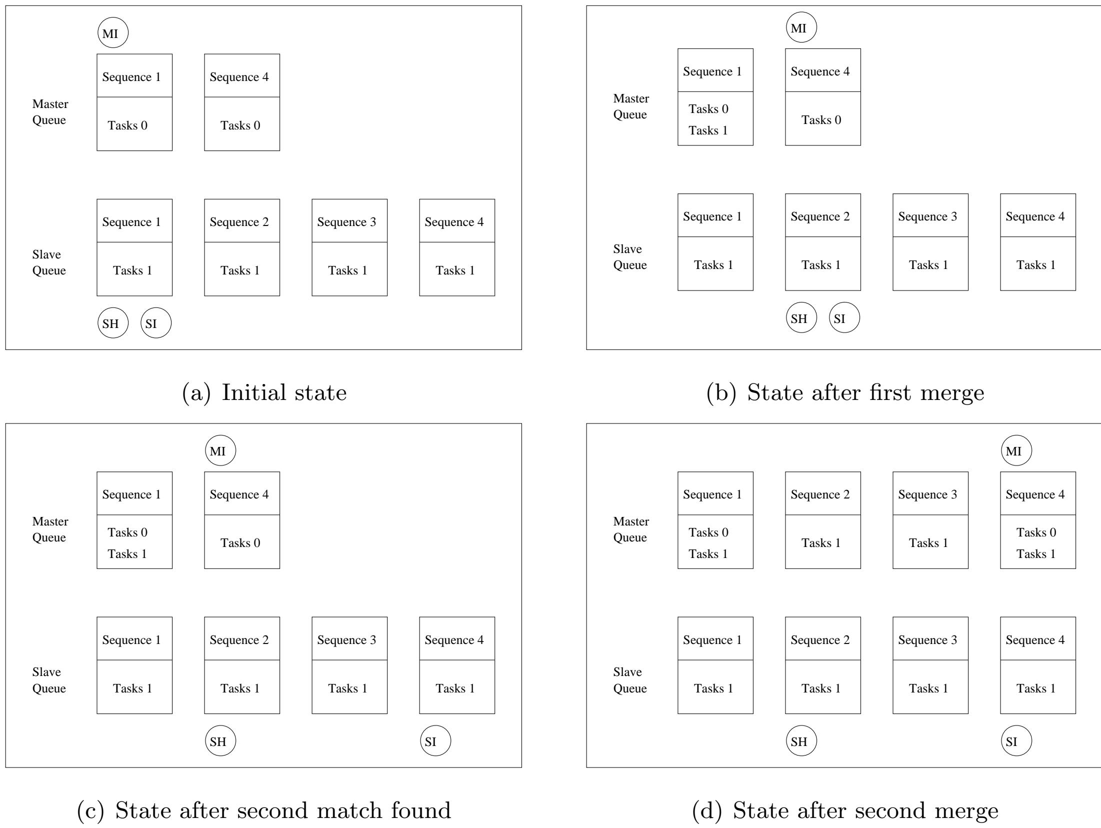
Figure 3.5: Merge algorithm used to merge slave trace into master trace

quence matching the current master iterator. If a match is found, all unmatched operation sequences are first copied into the master queue preceding the master iterator. The unmatched 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).

3.3.2 Merge Algorithm Examples

To clarify how the merge algorithm works, Figure 3.6(a) shows tasks 0 and 1 merging their operation queues. Task 1's operation queue is the slave that will be merged into task 0's operation queue, the master. The circles represent iterators with MI = master iterator, SH = slave head, and SI = slave iterator.

The algorithm begins with all iterators initially pointed to the beginning of their respective queues, as shown in Figure 3.6(a). Immediately, the algorithm detects that the slave iterator and the master iterator match. Figure 3.6(b) shows the state after the first merge has taken place. Note that because the slave iterator was also the slave head, there were no unmatched operations that had to be inserted before the master iterator. Figure 3.6(c) depicts the next match found. This time, there are unmatched operation sequences that must be inserted into the master queue. These operations are copied into the master



(a) Initial state

(b) State after first merge

(c) State after second match found

(d) State after second merge

Figure 3.6: Merge algorithm example

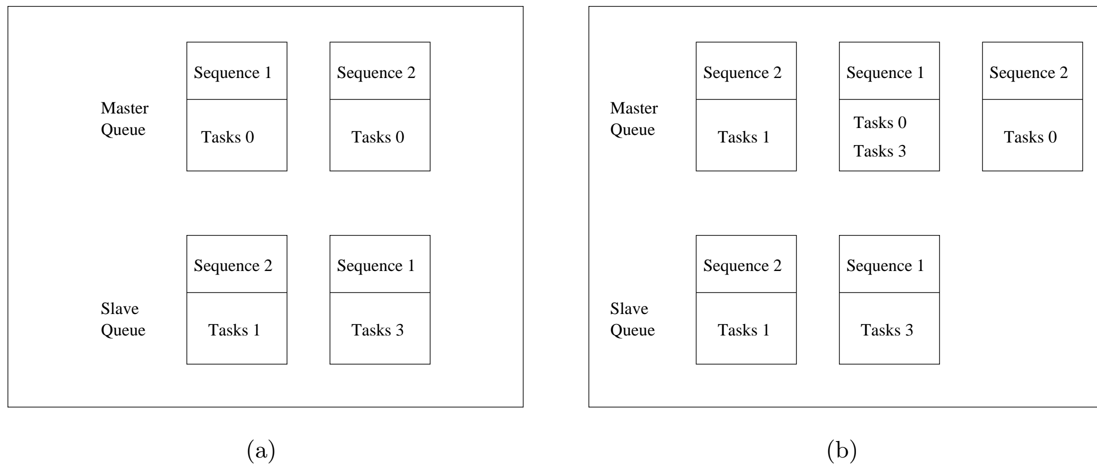


Figure 3.7: Merge algorithm dependency issue

queue and the task participant list in the master queue is updated. The final state of the master queue is depicted in Figure 3.6(d). The master queue now represents task 0 and 1's operation queues, and the slave queue can be discarded.

3.3.3 Operation Sequence Dependencies

The merging algorithm merges queues well between the leaf node level and interior node level of the tree, but there is a problem when merging between interior node levels and above. Figure 3.7 shows a situation in which four tasks participate in a job. Assume that task 0 performs sequence 1 and sequence 2, task 1 performs only sequence 2, task 2 performs only sequence 1, and task 3 performs only sequence 2. After tasks 1 and 3 have merged their queues, the resultant operation queue is depicted as the slave queue in Figure 3.7(a). On the first pass when attempting to find a match for sequence 1, the algorithm identifies the second operation sequence in the slave queue as a match. As a result, an additional copy of sequence two is inserted before sequence 1 in the master queue seen in Figure 3.7(b).

It is actually possible to avoid this situation since the ordering of operation sequences does not matter in the slave queue depicted in Figure 3.7(a). The reason that order does not matter is because different tasks participate in the operation sequences. This dependency can be determined by intersecting the task participant list of all unmatched operation sequences with the task participant lists of the matched operation sequence. If the

intersection is empty, there is no dependency. If there is no dependency, it is not necessary to copy the operation sequence into the master queue. Only those operation sequences that do have a dependency must precede the master iterator while the rest can stay in the slave queue. This allows for independent operation sequences remaining in the slave queue to be matched with an operation sequence from the master queue at a later time.

3.3.4 Merge Algorithm Complexity

The algorithm traverses the slave queue starting at the last matching operation sequence for each operation sequence in the master queue. This results in an algorithmic complexity of $O(n^2)$. In practice, due to the SPMD nature of MPI applications, many of the operation sequences match and the worst case scenario is seldom encountered.

3.4 Recursive Task Participant Lists

After the merging process is complete, each operation has a task participant list associated with it. Each list contains at least one task's rank and as many as the total number of tasks participating in the job. Since many MPI operations are performed across the MPI_COMM_WORLD communicator, it is important to represent the task participant lists concisely.

A recursive definition is most suitable for representing the task participant list. Originally, a simple list was used but with many MPI operations being performed over the MPI_COMM_WORLD communicator, another method had to be found. More information about the selection process is provided in the Results chapter where the recursive definition is compared to a regular list and task ID ranges).

The recursive definition is similar in nature to a list of RSDs. An array of integers is used to represent the recursive task participant list. The reasoning behind this is that originally a list (an array of integers) was used to represent which tasks participated in an operation. In order to fit into the cross-node framework smoothly, it was easier to utilize an integer array than a new data-structure. The array structure is shown in Figure 3.8. The first element specifies how many recursive lists there are. All subsequent elements are the recursive lists themselves. Within a recursive list, a starting element is specified and a depth. The starting element is the lowest task's rank. The depth indicates how many

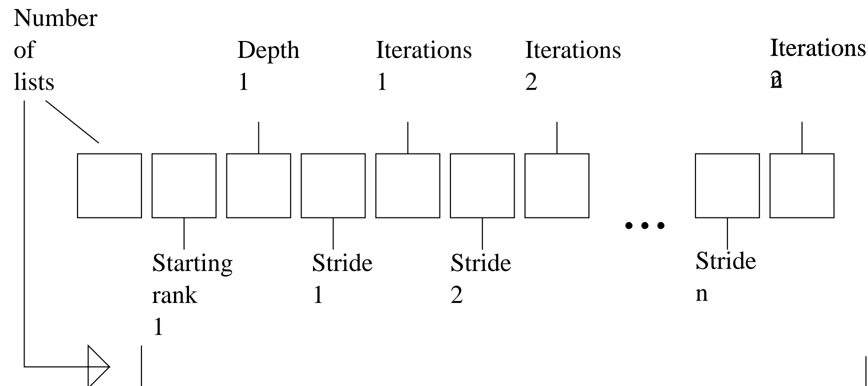


Figure 3.8: Recursive task participant list

0	1	2	3	4
5	(6)	(7)	(8)	9
10	(11)	(12)	(13)	14
15	(16)	(17)	(18)	19
20	21	22	23	24

Figure 3.9: Logical stencil - interior nodes circled

recursions (iterations) must occur to fully unwind the list (in Figure 3.8, the depth is n). At each depth, a stride and number of iterations is specified.

```
Unwind(int *list, int start, int depth)
    stride = list[0]
    iters = list[1]
    for(rank = start; rank ¡= iters; rank += stride)
        if(depth == 0) print rank
        else Unwind(list + 2, rank, depth - 1)
```

Figure 3.10: Algorithm to unwind recursive task participant list

Although the recursive task participant lists seem to work well in most cases, they were specifically designed with stencil problems in mind. To depict how they work, we will use a 25 node example logically laid out in a 2D space as shown in Figure 3.9. In this example, let us assume that all nodes are communicating on a nine-point stencil (*i.e.* all

their immediate neighbors). The interior nodes (highlighted with circles in Figure 3.9) are able to communicate with all their neighbors while all other nodes cannot (*i.e.* node 2 does not have a "northern" neighbor, node 23 does not have a "southern" neighbor, etc.). Thus, when merging is complete, the highlighted nodes will have overlapping operation sequences and must be represented together in a task participant list. The task participant list for the interior nodes in Figure 3.9 is: 1 6 2 5 3 1 3. The first element 1 specifies that there is only a single list. The next two elements specify that the starting rank is 6 and this list has a depth of 2.

To find which tasks participate in an operation, we unwind the list using the algorithm in Figure 3.10. Although unwinding a recursive task participant list is generally done in the replay tool (to determine if a task should replay an operation or not), it provides insight into the recursive definition.

Chapter 4

Replay Mechanism

Compressed traces produced by the recording tool described in the previous two chapters can be replayed on-the-fly independent of the original application. This opens tremendous opportunities for rapid prototyping of communication tuning, as well as for facilitating projections of network requirements and assessing communication needs for future large-scale procurements.

Replay is essentially accomplished by performing the inverse of the algorithms describing trace compression. First, the comprehensive trace is converted into a task-level operation queues for each task involved in the job. This step is the inverse operation of the cross-node framework merge and results in each task having a compressed operation queue (comprised of operation terminals, RSDs, and PRSDs). The compressed operation queues are replayed on-the-fly using an algorithm that is analogous to the inverse of the task-level framework compression. In addition to replaying MPI communication, computation simulation can also be turned on. This feature inserts time deltas between MPI operations based on timing averages embedded in each operation terminal.

The output of the replay tool is an overall execution time. This time reflects the average time it took each task to complete all of its communication.

4.1 Scanning for Task-Level Operation Queues

The first step to un-compressing the comprehensive trace is to determine which tasks perform which operations. It is unnecessary for each task in a job to be aware of the MPI communication performed by other job participants. Thus, in order to save memory

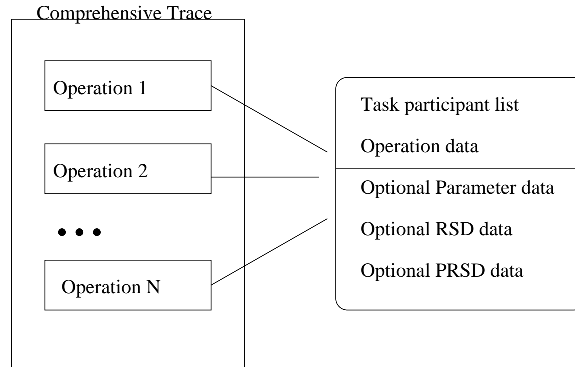


Figure 4.1: Comprehensive trace format

resources, each task scans the comprehensive trace for operations that it must perform. The replay tool does not begin timing communication until each task has its own individual operation queue.

To understand the scanning process, it is necessary to know how the comprehensive trace is laid out. The structure of the comprehensive trace is depicted in Figure 4.1. Operation entries are listed in the order they were executed. Each entry consists of a mandatory and optional section. The mandatory section is composed of a task participant list (described in section 3.4) and operational data (including which MPI function was called and its parameters). Every operation must contain this information for replay to be possible. The optional section of an entry is comprised of any combination of parameter data, RSD data, or PRSD data. Parameter data contains any special parameter compression as described in section 2.5.2. The RSD and PRSD data provide looping information such as iteration counts and how many operations the loop spans.

Each task generates an operation queue by scanning the comprehensive trace. The task participant list of each operation is initially parsed to determine if a task must perform that operation. If the task's MPI rank is among those in the task participant list, then it parses the rest of the mandatory section of the operation and any optional sections as well. The parsed data is packaged into an operation structure similar to those in the task-level framework and appended to the tail of the local operation queue. If the task's MPI rank is not among those in the task participant list, all subsequent data is ignored until another task participant list is encountered. The scanning process has an algorithmic complexity linear to the number of operation entries in the comprehensive trace.

4.2 Replay Algorithm

Once each task has its individual operation queue in memory, replay becomes possible. At this point, the replay tool starts a timer that will reflect how long communication takes. Replay of the operational queue is performed on-the-fly to conserve memory since the queue remains in compressed format. Only the cross-node compression has been undone up to this point.

```
replay_queue(node head, PRSD info)
    if(info == NULL)
        iters = 0
    else
        iters = into.iters
    Set tail to end of sequence
    for i = 0 to iters
        op = head
        while(op != tail)
            if(op has PRSD info)
                replay_queue(op, info.next)
            else
                replay_op(op)
            op = op.next
```

Figure 4.2: Algorithm to replay a compressed operation queue

The replay tool uses the recursive algorithm in Figure 4.2 to replay its operation queue. The initial call to replay_queue is made with the head of the operation queue and NULL as arguments. Since no PRSD data was passed, the iteration count is set to zero, and the tail variable is set to the tail of the operation queue. An iterator, denoted as op in Figure 4.2, is used to traverse the operation queue. If op does not have any PRSD (looping) data, it is immediately replayed. The call to replay_op uses a switch statement to lookup and perform the requested MPI communication operation.

In the event that the operation iterator has PRSD data, a recursive call to replay_queue is made. PRSD data represents looping and is a linked list of two-tupple data structures. The two-tupples consist of an iteration count and information to determine which of the proceeding elements in the queue belong in the loop. Each element in the linked list of two-tupples represents a deeper loop depth. Upon re-entering replay_queue, the iteration count is noted and the tail is set to the MPI operation at the end of the loop

based on the current PRSD data. The for loop ensures that the loop is replayed the correct number of times. The while loop ensures that all loop members are replayed. If another operation with PRSD information is encountered, the algorithm will make another recursive call. Control is relinquished once the sequence of operations in the queue between the head and the tail have been replayed as many times as the PRSD iteration count specifies.

4.3 Time Deltas - Simulating Computation

In order to model an application more accurately, the notion of a time delta can be enabled in the replay tool. Time deltas are the amount of time spent between MPI operations. They reflect run-time (excluding MPI operations), which we refer to as computational time.

Computational time is implemented in the function replay_op from Figure 4.2. Before the operation is replayed, the time delta embedded in the operation's data-structure is looked up. We use a busy loop that polls a timer until the amount of time specified by the time delta has elapsed before continuing.

Chapter 5

Experimental Results

The implementation of the record and replay tools required extensive experimentation to design, validate, and test the performance of the system as a whole. All experiments took place on a BG/L architecture at Lawrence Livermore National Laboratory using a core set of benchmarks and production-scale code. Design decisions driven by experimentation helped determine the best implementation methodology and are presented in this chapter. The techniques we used to validate the system are also presented. Lastly, the performance of the entire system is evaluated.

5.1 Experimental Environment

This section details the experimental environment. It begins with a brief overview of the BG/L architecture (the system used for all testing). A description of each benchmark and production-scale code used in the experiments follows.

5.1.1 Blue Gene/L Architecture

Blue Gene/L (BG/L) is a collaborated effort between International Business Machines (IBM) and Lawrence Livermore National Laboratory (LLNL) to create a massively parallel environment [4, 8]. The BG/L at LLNL is comprised of 65,536 compute nodes (although we were restricted to a maximum of 1,024 nodes) with each node based on system-on-a-chip integration [2]. Each compute node contains two embedded PowerPC 440 cores each with an individual L1 cache, an individual L2 cache, a shared L3 cache and

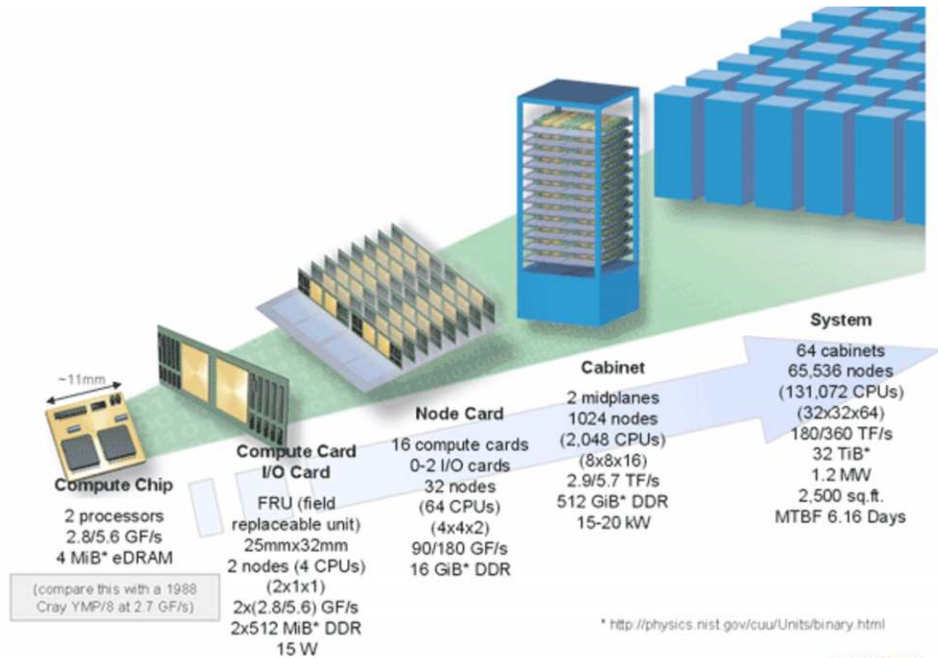


Figure 5.1: Hierarchical layout of BG/L

512 Megabytes of RAM. The nodes are organized in a hierarchical fashion as depicted in Figure 5.1.

Inter-node communication is accomplished over five distinct networks: torus, tree, Ethernet, JTAG, and global interrupts. The torus network performs point-to-point communication over a 3D torus (each node has 6 bi-directional links connected to its 6 nearest neighbors) while the tree network performs scalable global communication. These networks are where a majority of MPI communication takes place.

Each compute node runs atop a light, UNIX-like proprietary kernel called the compute node kernel (CNK). The compute nodes are controlled by a set of I/O nodes that run a standard distribution of Linux. The I/O nodes are in turn controlled by a set of login nodes also running Linux. A slightly modified version of MPICH is the implementation of MPI used on BG/L (with a few plug-in modules for additional functionality) [3].

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 (*i.e.*in BG/L environments where static linking is required).

5.1.2 Stencil Micro-Benchmarks

Our recording tool specifically targets stencil-like communication. A majority of our initial design decisions derive from three stencil micro-benchmarks in a 1D, 2D, and 3D logical space. We wrote each micro-benchmark so that we could manipulate multiple aspects of the stencil to investigate scalability from a number of perspectives. The number of tasks in a job is configurable, as is the workload (by controlling how many steps the stencil takes before it converges).

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. 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). See 1D stencil for other details.

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

5.1.3 Raptor Production-Scale Codes

Raptor is a framework implementing a modern Godunov method for shock-flow simulations in a C++/Fortran hybrid with optional adaptive mesh refinement (ARM) support [9]. It supports MPI and Pthreads parallelization and communicates on a twenty-seven-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.

5.2 Design Decision Experiments

Experimentation helped determine the implementation methodology. Early in the implementation phase, we began running our stencil micro-benchmarks instrumented with the recording tool and noticed a task scalability issue. As job-size increased, the resultant trace file also began to increase in size linearly with the number of tasks. Analysis of the trace files identified the task participant lists (introduced in chapter 3) as linearly dependent on the job-size.

Each operation in a trace must have a task participant list so the replay tool can distinguish which MPI operations are performed by which tasks. Due to the SPMD nature of our stencil micro-benchmarks (and many other scientific codes), most of the operations in the trace were performed by large sets of tasks. In our original design of the recording tool, task participant lists were implemented as an array of integers where each member of the array represented a separate task. Since many of the operations performed were equivalent across nodes, task participant lists grew linearly with the number of tasks in a job.

To address the scalability issue, we proposed two new methods of storing task participant lists. The first method was an array of integers in which consecutive pairs represented a range of tasks. Consider the array of integers: 1, 5, 7, 10, 11, 11. The array contains three pairs of integers representing three separate ranges. Thus, tasks 1 to 5, 7 to 10, and 11 would be members of that task participant list. The second method was an RSD representation (described in section 3.4).

Separate prototypes for the range and RSD's task participant list were implemented. The prototypes simulated merging an MPI operation guided by a balanced binary tree overlay (described in section 3.2). For the prototypes, an MPI operation consisted of a task participant list and operation ID. An input file was used to associate distinct sets of tasks to an operation ID. As children passed their operation queues up the binary tree, parents would consult the input file to find if merging were possible (a merge would take place if the operation IDs matched; otherwise, two separate operations would persist and be passed towards the root).

The results gathered for the two task participant list implementations included statistics for the list size at task 0, the average list size over all tasks, and the maximum list size during the binary tree traversal. Since trace file size was the original indicator of a scalability issue, the list size at task 0 (where the operation queue is completely merged and

0	1	2	3	4
5	6	7	8	9
10	11	12	13	14
15	16	17	18	19
20	21	22	23	24

Figure 5.2: 2D stencil illustrating nine distinct communication groups

the trace written to file) is an important metric. It gives insight into how a task participant list will affect the trace file size as the job size increases. In addition to the size of the trace file, we also examined the effects of list size on memory usage at each task. Memory usage at each task is affected by task participant lists in the same manner as the final trace size (the same linear dependence exists). By examining the list size at each task, we can determine the maximum and average list size for all tasks. These metrics give insight into task scalability with respect to memory usage as job size is varied.

Results are reported by the number of list elements in a node's task participant list. Because our original analysis indicated the scalability issue occurred in the stencil micro-benchmarks, we generated prototype inputs based on a 2D stencil. Figure 5.2 illustrates how a nine-point stencil in a 2D grid has nine different operation sets delineated by different shapes (the four corners each constitute a unique set). Each operation set has a different communication pattern based on which of its surrounding neighbors have valid MPI ranks. For example, the interior nodes marked with circles can communicate with all eight surrounding neighbors whereas the nodes marked with squares do not communicate in the "southern" direction and, thus, only communicate with five neighbors.

Figure 5.3 shows the performance of the range implementation of a task participant list. All three metrics continue to exhibit a linear dependence on job-size. The dependence on job-size exists because of the stencil's layout. The diamond, circle, and triangle groups from Figure 5.2 are the stem of the problem. The range implementation of a task participant list cannot represent these groups concisely because the tasks do not have contiguous MPI

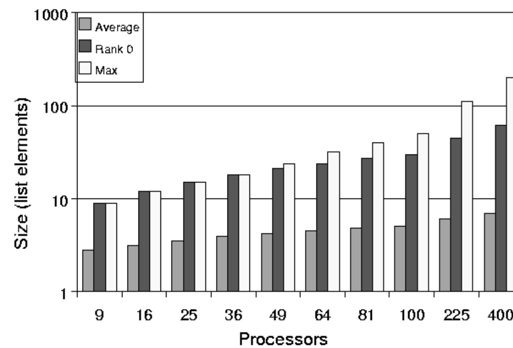


Figure 5.3: Task participant list - Range implementation (logarithmic scale)

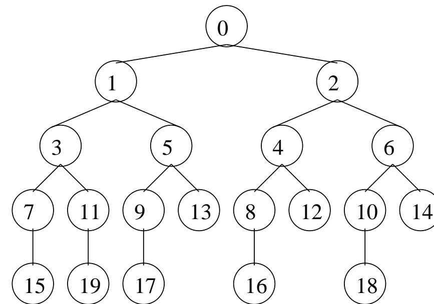


Figure 5.4: Balanced binary tree

ranks. As job-size increases (thus increasing the 2D grid), each of the problem groups increases in size. Consider an increase from the 5x5 grid in Figure 5.2 to a 6x6 grid. The interior node group marked by circles gains an additional row resulting in an additional range that must be represented in the task participant list. Thus, trace file size and memory usage remain linearly dependent on job-size with the range implementation of a task participant list.

Figure 5.3 also shows a growing disparity between task 0's list size and the maximum list size. In memory-constrained environments, this becomes a major problem. After examining the range implementation more closely, we determined the problem was an artifact of the balanced binary tree we were using to control the merging process. By following the merging process from leaf node to root, we saw that operations performed by all tasks could not be represented concisely near the root of the tree. Consider the binary tree in

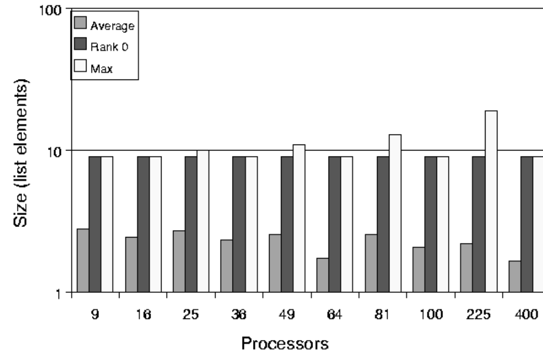


Figure 5.5: Task participant list - RSD implementation (logarithmic scale)

Figure 5.4 at nodes 1 and 2. Node 1 accumulates all of the odd numbered operations during its merging phase because all of its descendants (children, grandchildren, etc.) have odd MPI ranks. The same is true for node 2 and the even numbered MPI ranks. The range implementation of task participation lists cannot represent a list of even or odd tasks concisely because it relies on contiguous integers. As a result, memory usage balloons out near the root of the tree to as much as N / 2 (for operations in which all tasks participate) list elements where N is the number of processors.

Figure 5.5 shows the performance of the RSD implementation of a task participant list. All metrics exhibit nearly a constant size list for varying job-sizes. The RSD implementation uses a stride to concisely represent the groups which the range implementation could not. Thus, the diamond, circle, and triangle groups from Figure 5.2 can be represented in a single list-element. Additionally, the regularity of the binary tree's layout is taken advantage of by the RSD implementation. Regardless of location in the tree, all descendants exhibit a very regular pattern that the RSD implementation concisely represents. As a result, the RSD implementation of a task participant list does not suffer from memory usage ballooning at any point in the binary tree (in contrast to the range implementation). With evidence that the RSD implementation performed better for both file size and memory usage scalability, we implemented task participation lists as RSDs rather than ranges in the final system.

5.3 Validation Procedures

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.

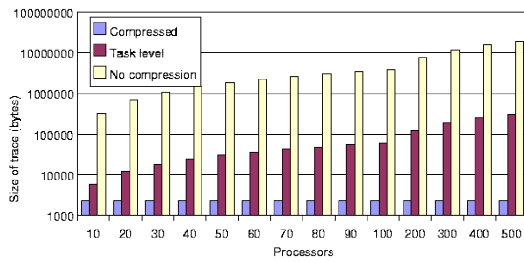
We also modified the record and replay tools to generate a list of uncompressed calls. In the recording tool, this amounted to instrumenting the MPI function wrappers with additional code to write out all operations to file for each task. The replay tool was similarly modified by instrumenting the function, replay_op (see section 4.2 for replay algorithm details), with code to write all operations to file in the same manner. In both tools, instrumentation occurred right before the actual MPI call was performed. Each micro-benchmark and production scale code presented in the Results chapter was recorded and replayed using the modified tools. Running the tools resulted in two sets of uncompressed operation lists for each task. A text comparison of the uncompressed traces for all applications yielded no differences and satisfied us that we were performing a lossless replay.

The final validation procedure we employed was instrumenting the record and replay tools with mpiP. This tool collects statistical information about MPI functions (time spent in functions, number of function calls, etc.) and generates a report at the application's termination. Each micro-benchmark and production scale code presented in the Results chapter was recorded and replayed using the tools instrumented with mpiP. Data pertaining to the which function calls were made in the reports was compared, and we found no differences.

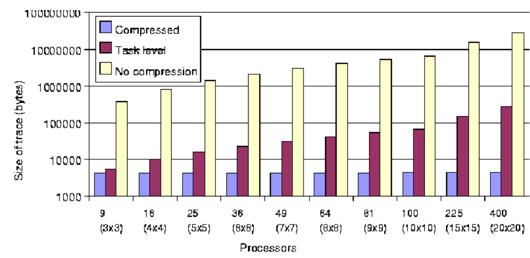
5.4 Performance Results

Results are obtained for a stencil benchmark and a production-scale code, Raptor. The metrics explored are the size of trace files and memory requirements on a per-node basis on BG/L.

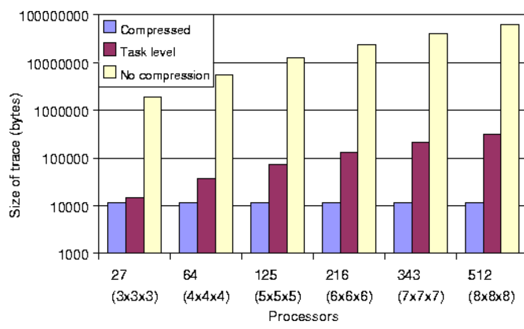
Figure 5.6 depicts the trace file sizes of the 1D 2D and 3D stencil code for varying stencil sizes (number of nodes) as well as Raptor for varying job sizes. Trace sizes are reported on a logarithmic scale for the nodes (1) without compression, (2) only with task-



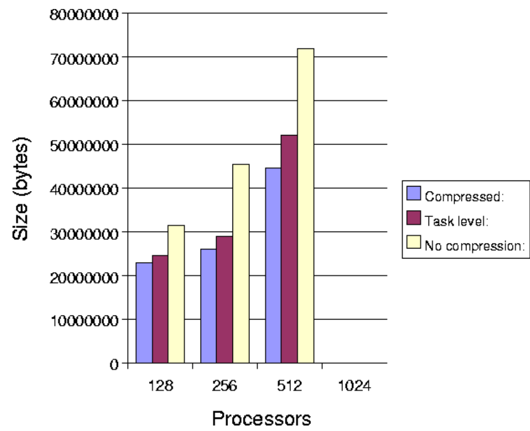
(a) 1D stencil trace file, varied number of nodes



(b) 2D stencil trace file, varied number of nodes



(c) 3D stencil trace file, varied number of nodes



(d) Raptor trace file, varied time steps

Figure 5.6: Trace file size per node on BlueGene/L

level (intra-node) compression and (3) 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 for the stencil micro-benchmarks (Figures 5.6(a) to 5.6(c)). 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.

Figure 5.6(d) depicts the trace file size for Raptor confirming most of the prior benchmark observations for a complex application code. In addition to the previous observations, an increase in the file size from 23MB to 45MB for 128 and 512 nodes, respectively, can be seen. This increase is due to minor inefficiencies of cross-node compression currently being addressed. We believe the problem is due to array parameters of MPI calls. In Raptor, a time-step consists of stencil communication followed by processing synchronized by MPI_Waitsome. The arrays passed to MPI_Waitsome are equal in length to the number of tasks participating in the job.

As BG/L is a memory-constrained architecture with only 512 MB RAM per node, not only the resulting trace sizes matter. Keeping the memory pressure low during on-the-fly compression is equally important. Figure 5.7 depict the memory usage reflecting the intra- and inter-node compression components for the 1D, 2D and 3D stencil benchmarks, and Raptor 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 (Figures 5.7(a) to 5.7(c)). 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.

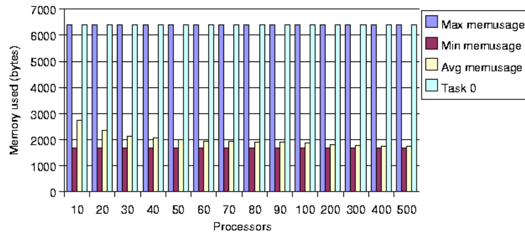
Figure 5.7(d) depicts the memory usage for Raptor confirming most of the prior benchmark observations for a complex application code. In addition to the previous observations, a slight increase in the maximum memory usage of 38MB to 55MB for 128 and 1024 nodes, respectively, can be seen. This increase 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.

Figure 5.8 depicts the trace file size as the number of time steps is varied, *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.

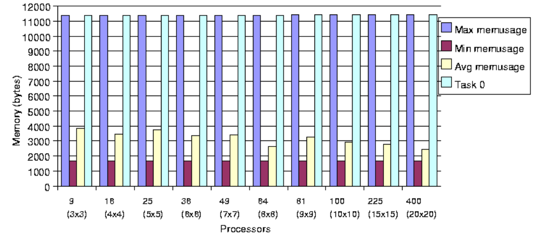
5.5 Replay Results

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

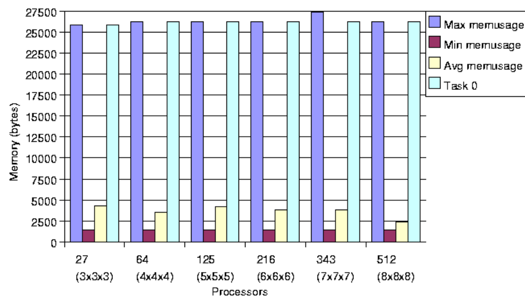
The replay mechanism opens up tremendous opportunities beyond the verification of correctness. As mentioned before, it may be utilized for rapid prototyping of communication tuning as well as for assessing communication needs of future platforms for large-scale procurements. We are currently pursuing these directions, among others to improve commu-



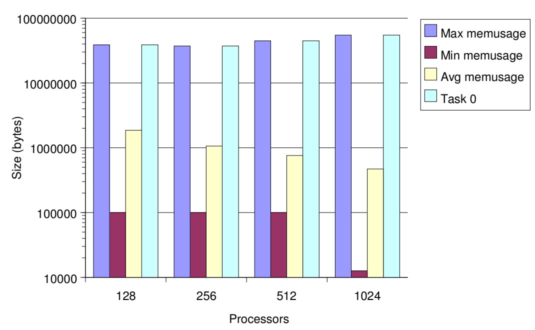
(a) 1D stencil memory usage, varied number of nodes



(b) 2D stencil memory usage, varied number of nodes



(c) 3D stencil memory usage, varied number of nodes



(d) Raptor memory usage, varied number of nodes

Figure 5.7: Memory usage per node on BlueGene/L

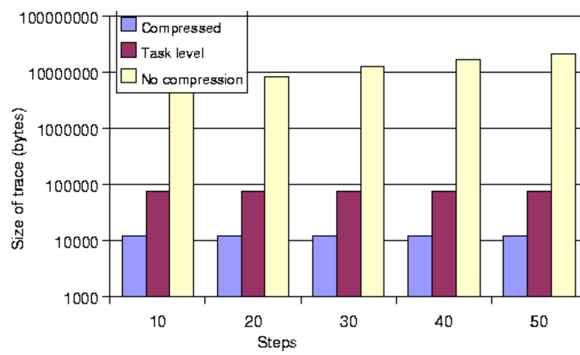


Figure 5.8: 3D stencil trace file, varied time steps

nication 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.

Chapter 6

Related Work

RSDs have been used to describe data references in a loop [10]. The idea of PRSDs originates from memory trace compression performed on-the-fly [13]. While their work introduced the general concepts and an algorithm for compressing regular data references, our work uses an entirely different algorithm. This is due to the considerably more complex task of compressing events composed of MPI call signatures and their parameters. We also utilize semantically-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 [19]. Hence, structural information and event ordering are not preserved. There are many other tools that report aggregate information, often based of the profiling layer 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 trace file sizes increase linearly with the number of MPI calls made. This limits the applicability as scalability is compromised.

MRNet is a software overlay network that provides efficient multicast and reduction communications for parallel and distributed tools and systems [16]. MRNet uses a tree of processes between the tool's front-end and back-ends to improve group communication performance. We experimented with MRNet to support inter-node compression. Yet, due

to the advantages of the radix tree representation for compression and the simplistic nature of the reduction tree, we decided to maintain our own reduction infrastructure.

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 [18]. 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.

Chapter 7

Conclusion

One of the central problems in Peta-scale 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 statistically analyze aggregate metrics 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 descriptor, 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.

Bibliography

[1] Dong H. Ahn and Jeffrey S. Vetter. Scalable analysis techniques for microprocessor performance counter metrics. pages 1–16.

[2] George Almási, George S. Almasi, Daniel K. Beece, Ralph Bellofatto, Gyan Bhanot, Randy Bickford, Matthias A. Blumrich, Arthur A. Bright, Jose Brunheroto, Calin Cascaval, José G. Castaños, Luis Ceze, Paul Coteus, Siddhartha Chatterjee, Dong Chen, G. Chiu, T. M. Cipolla, Paul Crumley, Alina Deutsch, M. B. Dombrowa, Wilm E. Donath, Maria Eleftheriou, Blake G. Fitch, Joseph Gagliano, Alan Gara, Robert S. Germain, Mark Giampapa, Manish Gupta, Fred G. Gustavson, Shawn Hall, Ruud A. Haring, Dave Heidel, Philip Heidelberger, Lorraine Herger, Dirk Hoenicke, T. Jamal-Eddine, Gerard V. Kopcsay, A. P. Lanzetta, Derek Lieber, M. Lu, Mark P. Mendell, L. Mok, José E. Moreira, Ben J. Nathanson, Matthew Newton, Martin Ohmacht, Rick A. Rand, Richard D. Regan, Ramendra K. Sahoo, Alda Sanomiya, Eugen Schenfeld, Sarabjeet Singh, Peilin Song, Burkhard D. Steinmacher-Burow, Karin Strauss, Richard A. Swetz, Todd Takken, R. Brett Tremaine, Mickey Tsao, Pavlos Vranas, T. J. Christopher Ward, Michael E. Wazlowski, J. Brown, T. Liebsch, A. Schram, and G. Ulsh. Blue gene/L, a system-on-A-chip. In *CLUSTER*, page 349. IEEE Computer Society, 2002.

[3] George Almási, Charles Archer, José G. Castaños, Manish Gupta, Xavier Martorell, José E. Moreira, William Gropp, Silvius Rus, and Brian R. Toonen. MPI on bluegene/L: Designing an efficient general purpose messaging solution for a large cellular system. In Jack Dongarra, Domenico Laforenza, and Salvatore Orlando, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface,10th European PVM/MPI Users' Group Meeting, Venice, Italy, September 29 - October 2, 2003, Pro-*

ceedings, volume 2840 of *Lecture Notes in Computer Science*, pages 352–361. Springer, 2003.

[4] George Almási, Ralph Bellofatto, Jose Brunheroto, Calin Cascaval, José G. Castaños, Luis Ceze, Paul Crumley, C. Christopher Erway, Joseph Gagliano, Derek Lieber, Xavier Martorell, José E. Moreira, Alda Sanomiya, and Karin Strauss. An overview of the blue gene/L system software organization. In Harald Kosch, László Böszörményi, and Hermann Hellwagner, editors, *Euro-Par 2003. Parallel Processing, 9th International Euro-Par Conference, Klagenfurt, Austria, August 26-29, 2003. Proceedings*, volume 2790 of *Lecture Notes in Computer Science*, pages 543–555. Springer, 2003.

[5] Greg Burns, Raja Daoud, and James Vaigl. LAM: An open cluster environment for MPI, 1994.

[6] Andrew W. Cook, William H. Cabot, Peter L. Williams, Brian J. Miller, Bronis R. de Supinski, Robert K. Yates, and Michael L. Welcome. Tera-scalable algorithms for variable-density elliptic hydrodynamics with spectral accuracy. In *SC*, page 60. IEEE Computer Society, 2005.

[7] Frederica Darema-Rogers, V. A. Norton, and G. F. Pfister. Using a single-program-multiple-data computational model for parallel execution of scientific applications. Research Report RC 11552, IBM T.J. Watson Research Center, Yorktown Heights, New York, November 1985.

[8] A. Gara, M. A. Blumrich, D. Chen, G. L.-T. Chiu, P. Coteus, M. E. Giampapa, R. A. Haring, P. Heidelberger, D. Hoenicke, G. V. Kopcsay, T. A. Liebsch, M. Ohmacht, B. D. Steinmacher-Burow, T. Takken, and P. Vranas. Overview of the Blue Gene/L system architecture. *IBM Journal of Research and Development*, 49(2/3):195–212, ???? 2005.

[9] 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.

[10] 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.

[11] Ewing Lusk. Installation guide to mpich, a portable implementation of MPI, February 09 1996.

[12] Makinen. A survey on binary tree codings. *COMPJ: The Computer Journal*, 34, 1991.

[13] Jaydeep Marathe, Frank Mueller, Tushar Mohan, Bronis R. de Supinski, Sally A. McKee, and Andy Yoo. METRIC: Tracking down inefficiencies in the memory hierarchy via binary rewriting. In *CGO*, pages 289–300. IEEE Computer Society, 2003.

[14] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*. University of Tennessee, Knoxville, TN, June 1995.

[15] Luiz De Rose, Kattamuri Ekanadham, Jeffrey K. Hollingsworth, and Simone Sbaraglia. SIGMA: a simulator infrastructure to guide memory analysis. pages 1–13.

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

[17] Julian Seward. bzip2 and libbzip2 - a program and library for data compression. http://www.bzip.org/1.0.3/bzip2-manual-1.0.3.pdf, 1996.

[18] Shuyi Shao, Alex Jones, and Rami Melhem. A compiler-based communication analysis approach for multiprocessor systems. In *IPDPS*, 2006.

[19] J. Vetter and M. McCracken. Statistical scalability analysis of communication operations in distributed applications. In *PPoPP*, 2001.

[20] Jeffrey S. Vetter. Dynamic statistical profiling of communication activity in distributed applications. In *SIGMETRICS*, pages 240–250. ACM, 2002.

[21] Jeffrey S. Vetter and Bronis R. de Supinski. Dynamic software testing of MPI applications with umpire. In *SC*, 2000.

[22] Jeffrey S. Vetter and Michael O. McCracken. Statistical scalability analysis of communication operations in distributed applications. In *PPOPP*, pages 123–132, 2001.