

2010

Scalable event tracking on high-end parallel systems

Kathryn Marie Mohror
Portland State University

Let us know how access to this document benefits you.

Follow this and additional works at: http://pdxscholar.library.pdx.edu/open_access_etds

 Part of the [Systems Architecture Commons](#)

Recommended Citation

Mohror, Kathryn Marie, "Scalable event tracking on high-end parallel systems" (2010). *Dissertations and Theses*. Paper 2811.

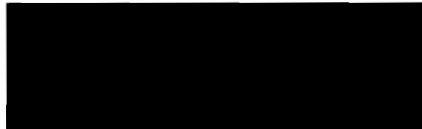
[10.15760/etd.2805](https://pdxscholar.library.pdx.edu/etd/2805)


This Dissertation is brought to you for free and open access. It has been accepted for inclusion in Dissertations and Theses by an authorized administrator of PDXScholar. For more information, please contact pdxscholar@pdx.edu.


DISSERTATION APPROVAL

The abstract and dissertation of Kathryn Marie Mohror for the Doctor of Philosophy in Computer Science were presented December 11, 2009, and accepted by the dissertation committee and the doctoral program.

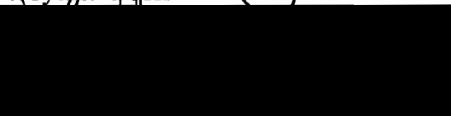
COMMITTEE APPROVALS:


Karen L. Karavanic, Chair



Jingke Li


Suresh Singh


Bryant York


Christopher M. Monsere
Representative of the Office of Graduate Studies

DOCTORAL PROGRAM APPROVAL:


Wu-chi Feng, Director
Computer Science Ph.D. Program

ABSTRACT

An abstract of the dissertation of Kathryn Marie Mohror for the Doctor of Philosophy in Computer Science presented December 11, 2009.

Title: Scalable Event Tracing on High-End Parallel Systems

Accurate performance analysis of high end systems requires event-based traces to correctly identify the root cause of a number of the complex performance problems that arise on these highly parallel systems. These high-end architectures contain tens to hundreds of thousands of processors, pushing application scalability challenges to new heights. Unfortunately, the collection of event-based data presents scalability challenges itself: the large volume of collected data increases tool overhead, and results in data files that are difficult to store and analyze. Our solution to these problems is a new measurement technique called trace profiling that collects the information needed to diagnose performance problems that traditionally require traces, but at a greatly reduced data volume. The trace profiling technique reduces the amount of data measured and stored by capitalizing on the repeated behavior of programs, and on the similarity of the behavior and performance of parallel processes in an

application run. Trace profiling is a hybrid between profiling and tracing, collecting summary information about the event patterns in an application run. Because the data has already been classified into behavior categories, we can present reduced, partially analyzed performance data to the user, highlighting the performance behaviors that comprised most of the execution time.

SCALABLE EVENT TRACING ON HIGH-END PARALLEL SYSTEMS

by

KATHRYN MARIE MOHROR

A dissertation submitted in partial fulfillment of the
requirements for the degree of

DOCTOR OF PHILOSOPHY
in
COMPUTER SCIENCE

Portland State University
©2010

Acknowledgments

I thank my dissertation advisor, Karen L. Karavanic, for her extraordinary guidance, understanding, and support over these years. Because of her efforts, I am solidly prepared for my future research career. I know that at this moment I sincerely appreciate all she has done for me, as well as I know that my appreciation will only grow over time as I begin to fully realize the depth of her commitment to my success. Thank you, Karen.

I thank my dissertation committee (Jingke Li, Suresh Singh, Bryant York, and Christopher Monsere) for taking the time to provide thoughtful feedback and advice on my research. Your expertise made my dissertation stronger.

Thank you to my fellow students in the High Performance Computing Lab at Portland State University, and most especially to Rashawn Knapp, for your camaraderie, your willingness to be a sounding board for new ideas, and ability to sit through countless practice talks.

I thank John May and Lawrence Livermore National Laboratory for giving me opportunities for collaboration and for access to LLNL computing resources.

Last, but certainly not least, I thank my husband and family for their unwavering support during my time in school. I know I was sometimes a very distracted wife, mother, daughter, sister, but you all supported my goals regardless. Thank you for cheering me on during the good times and cheering me up during the bad. If it weren't for you, the road would have been much more difficult.

Table of Contents

Acknowledgments	i
List of Figures.....	iv
List of Tables.....	vi
1 Introduction	1
1.1 Motivation	1
1.1.1 Uses of Event Tracing	2
1.1.2 Summary.....	10
1.2 Dissertation Contributions.....	10
1.3 Dissertation Organization.....	12
2 Related Work.....	13
2.1 Perturbation	13
2.2 Trace File Size Reduction	15
2.2.1 Trace File Compression.....	15
2.2.2 Measuring or Writing Less Data	16
2.3 Analysis Tool and Visualization Scalability	21
3 Study of Tracing Overheads.....	23
3.1 Experiment Design	23
3.2 Results	28
3.2.1 Event Counts and Trace File Sizes	28
3.2.2 Execution Time	29
3.2.3 Execution Time vs Event Counts	32
3.3 Conclusions	33
4 Trace Profiling.....	35
4.1 Background.....	36
4.2 Trace Profiling Technique.....	36
4.2.1 Trace Segmentation.....	40
4.2.2 Intra-process Segment Comparison.....	41
4.2.3 Inter-process Segment Comparison.....	42
4.3 Trace Profile Segment Comparison Methods.....	45
4.3.1 Distance Methods	45
4.3.2 Iteration-based Methods	51
4.4 Traditional Trace and Trace Profile Size Models.....	52
4.4.1 Traditional Trace	52
4.4.2 Trace Profile	54
4.5 Traditional Trace and Trace Profile Size Comparison Using Models	57
4.5.1 Traditional Trace	58
4.5.2 Trace Profile	60
4.5.3 Comparison of Traditional Trace and Trace Profile.....	60
4.6 Trace Profiling and Visualization.....	62
4.7 Summary.....	63
5 Trace Comparison Methods	65

5.1	Evaluation Methodology	66
5.1.1	Benchmarks	66
5.1.2	Application	70
5.1.3	Instrumentation.....	70
5.1.4	Evaluation Criteria.....	71
5.2	Intra-process Reduction Evaluation Studies.....	74
5.2.1	Threshold Study.....	74
5.2.2	Comparative Study	77
5.3	Inter-process Reduction Evaluation Studies.....	85
5.3.1	Threshold Study.....	85
5.3.2	Comparative Study	87
5.4	Combined Inter-process and Intra-process Reduction Evaluation	93
5.4.1	Size and Degree of Matching	93
5.4.2	Approximation Distance.....	94
5.4.3	Retention of Trends	96
5.4.4	Discussion.....	98
5.5	Discussion.....	98
5.5.1	Trace Similarity Methods	98
5.5.2	Intra- and Inter-process Matching	99
5.6	Summary.....	100
6	Prototype Runtime Trace Profiler	102
6.1	Current Prototype Implementation	102
6.1.1	Trace Profiler Front End.....	102
6.1.2	Trace Profiler Instrumentation Library	104
6.1.3	Runtime Operations.....	106
6.2	Experimental Setup	108
6.2.1	Application	108
6.2.2	Machine	108
6.2.3	Tool Configurations.....	108
6.3	Results	110
6.3.1	Execution Time	110
6.3.2	Total File Size.....	112
6.3.3	Flushes.....	114
6.4	Discussion.....	114
7	Conclusions	65
7.1	Future Work.....	117
7.1.1	Performance Tool Memory Bounds	118
7.1.2	Trace Profiler Measurement Overheads.....	118
8	References	122
	Appendix: Additional Trace Similarity Study Results.....	128

List of Figures

Figure 1 Performance of Uninstrumented Executions	25
Figure 2 Experiment Environment.	27
Figure 3 Performance of Instrumented Executions.	30
Figure 4 Tracing Overhead with Maximum Event Count in a Single Rank.	32
Figure 5 Data from Traditional Trace	37
Figure 6 Process Group	38
Figure 7 Segment Context Marking	40
Figure 8 Algorithm for Intra-process Segment Matching	41
Figure 9 Intra-process Segment Matching	42
Figure 10 Inter-process Segment Matching	43
Figure 11 Algorithm for Inter-process Matching	44
Figure 12 Wavelet Transform Example	50
Figure 13 Trace Profile Format	55
Figure 14 Input Data to Traditional Trace Size Model	57
Figure 15 Inputs to Trace Profile Size Model	58
Figure 16 Traditional Trace and Trace Profiling Sizes for Random-Barrier	61
Figure 17 Trace Profiler Visualization	62
Figure 18 KOJAK and Derivation of Our Performance Diagnosis Representation.	73
Figure 19 Intra-process Reduction: Percentage File Sizes and Degree of Matching.	78
Figure 20 Intra-process Reduction: Approximation Distance Results for All Methods at Default Thresholds.....	79
Figure 21 Intra-process Reduction: KOJAK Performance Trends for dyn_load_balance For Each Method at Default Thresholds.....	82
Figure 22 Intra-process Reduction: KOJAK Performance Trends for 1to1r_1024 for Each Method at Default Thresholds.	83
Figure 23 Inter-process Reduction: Percentage File Sizes for Methods at Default Thresholds.....	88
Figure 24 Inter-process Reduction: Degree of Matching for Methods at Default Thresholds.....	89
Figure 25 Inter-process Reduction: Approximation Distance for the Methods at Default Thresholds.....	89
Figure 26 Inter-process Reduction: KOJAK Performance Trends for early_gather for Each Method at Default Thresholds.....	91

Figure 27 Inter-process Reduction: KOJAK Performance Trends for NtoN_1024 for Each Method at Default Thresholds	91
Figure 28 Combined Reduction: Percentage File Sizes for Methods at Default Thresholds.....	95
Figure 29 Combined Reduction: Degree of Matching for Methods at Default Thresholds.....	95
Figure 30 Combined Reduction: Approximation Distance for Methods at Default Thresholds.....	96
Figure 31 Combined Reduction: KOJAK Performance Trends for NtoN_32 for Each Method at Default Thresholds	97
Figure 32 Combined Reduction: KOJAK Performance Trends for sweep3d_32p for Each Method at Default Thresholds	97
Figure 33 Example Segment Context Marking and Names	105
Figure 34 Example Instrumentation for Message Passing Function	106
Figure 35 Execution Time of Sweep3d Measured with TAU and TP	111
Figure 36 Write Overhead for Sweep3d with TAU and TP	112
Figure 37 Total Size of Files Generated for Sweep3d with TAU and TP.....	113
Figure 38 Average File Size Per Rank for TAU and TP.....	113
Figure 39 Total Buffer Flush Count for Sweep3d with TAU and TP.....	115
Figure 40 Average Flush Count Per Rank with TAU and TP.....	115

List of Tables

Table 1	Correlation of Total Wall Time with Maximum Event Count in a Rank...	31
Table 2	Symbols for Full Trace and Trace Profile Models	53
Table 3	Sizes of Fields in Trace Profiling Data Structures	56
Table 4	Sizes of Trace Profile and Full Trace	59
Table 5	Trace Profile Instrumentation Library Interface	107

1 Introduction

The major contribution of this dissertation is a novel, low-overhead technique for collecting event traces on high-end computing systems. We collect the information needed to correctly diagnose certain complex performance problems at a greatly reduced data volume over traditional event trace collection methods. Other contributions of this dissertation include: an in-depth measurement study of the overheads of traditional event trace collection; an evaluation of methods for determining event trace equivalence; and post-mortem and runtime prototypes of the new event trace collection technique to demonstrate its viability.

1.1 Motivation

Today's high-end architectures contain tens to hundreds of thousands of processors, pushing application scalability challenges to new heights. Performance analysis is a necessary step to adapt codes to utilize a target high end machine. Correct diagnosis of certain complex performance problems that arise on high end systems requires detailed event traces. An "event" is a runtime occurrence of a program activity, such as a machine instruction or basic block execution, memory reference, function call, or a message send or receive. Generating event traces involves writing a time stamped record for each event, into a buffer or file for later analysis. Unfortunately, the collection of event traces presents scalability challenges: the act of measurement perturbs the target application; and the large volume of collected data results in data files that are difficult, or even impossible, to store and analyze [45].

There are several documented cases of performance problems that appear only when the application is run at a large scale [32, 51], driving the need to be able to collect event traces for large runs. We have a conundrum: we need traces to correctly diagnose important performance problems, but the sheer volume of data collected makes collecting full traces at the very least prohibitive, and in the worst case impossible. For this reason, solving the scaling challenges of event tracing is an important problem for high end computing.

1.1.1 Uses of Event Tracing

Requirements for the accuracy and types of information in a trace vary based on the intended use: correctness testing and debugging, simulation, or performance analysis.

Correctness testing and debugging generally only require that the trace retain the relative ordering of events. For example, inspecting a trace of a parallel program could indicate the reason for a deadlock situation by showing the ordering of synchronization operations; a parallel program might hang because a process is waiting for a message that was never sent.

Simulation requires traces that retain the order of events and possibly some timing information. Traces for simulation can be used to predict application performance on new or theoretical hardware. The events in the trace can be replayed using either averaged or predicted timing information for the new hardware. Generally, a single time value is used for all event occurrences instead of individual timing measurements for each event occurrence. For example, the average time to execute a send operation

could be used as the time for all send operations in the trace. This tradeoff allows acceptable accuracy with faster time to simulated results and smaller trace files.

Performance analysis requires not only the relative ordering of events, but the timing information for individual events. Performance problems do not necessarily occur with a high degree of regularity, e.g. in every iteration of a loop, so individual event timings are needed to show the root causes of problems. For example, trace data can show a time-varying load imbalance in a parallel job, which causes some ranks to be late to a synchronization operation at varying times during the program execution. The individual event timings can show what events are taking more time in the slower ranks and in what iterations the slowness occurs. In this dissertation, we focus solely on collecting event traces for the purpose of performance analysis.

1.1.1.1 The Necessity of Event Tracing

Event tracing is used for understanding the causality of events, understanding the interactions between program elements, and identifying behaviors from event patterns. Although other performance measurement techniques, such as profiling, exhibit better scaling properties at the high end, the detail collected in an event trace is needed to correctly diagnose certain performance problems.

An event trace can show the causality of events, which is helpful when a specific set of events lead to a performance issue. An example of this is found in a case study showing the benefits of Stardust, a tool for collecting and retrieving end-to-end performance traces in a distributed system [62]. The researchers in this study investigated a user's reported problem with I/O performance. From an event trace of

the program, they were able to see the sequence of events that caused the poor performance, a series of small requests.

Event traces are useful for showing the interactions between program elements, because interactions can sometimes be difficult or impossible to understand from static analysis. For example, understanding the interactions between program elements is useful in the realm of parallel program debugging. Kranzlmüller et al. use event graphs, generated from trace files, to discover bugs in parallel programs [36]. They use the relationships between processes revealed by the program trace to determine where race conditions due to non-deterministic execution could occur.

Event-patterns in traces can be analyzed to reveal properties of programs, such as performance problems and locations of possible optimization. An example of performance problem that event patterns can help diagnose is the “Late Sender” problem. This is the situation where the receiving process waits at a blocking receive call waiting because the sending process hasn’t yet reached the matching send call. The relative timing of events in the trace would show that the send operations started late and caused the receive operations to block. Event patterns can also be analyzed to suggest performance optimizations. Kranzlmüller et al. present a method for recognizing point-to-point communication patterns in program traces that correspond to collective communication operations [38]. Since collective communication operations are often tuned for high performance on each platform, they suggest to the user to replace the recognized point-to-point sequence of operations with a collective communication operation.

Profiling and tracing represent two ends of the spectrum in the trade-off between the level of detail and the amount of data collected in performance measurement. Profiling provides summary information and therefore is more scalable than tracing. For example, a profile can show which functions used the most amount of time in an execution. This tells a performance analyst a crucial piece of information: where the program is spending most of its time, identifying candidates for performance improvements. Profiling has advantages over tracing, because it causes less perturbation to the target program and produces smaller performance data files. Tracing a program results in a sequence of time-stamped events, possibly with accompanying performance information, e.g. the start and end times of a particular routine, or details about message-passing events, such as the sending and receiving processes and the communicator used. Tracing provides more detail about the performance of the program, at the cost of greater perturbation to the target program and larger resulting data files. Although the costs of collecting event traces are higher, there are situations where the level of detail provided by tracing is required; the types of information provided by profiling are, in many cases, too limited for correct diagnosis of certain performance problems [7, 37]. An example of such a performance problem is the previously described “Late Sender” problem in a message-passing program. While a profile could indeed show that excessive time was being spent in receive operations, the data is not sufficient to distinguish between a late sender or some other root cause, such as network contention that caused the message to be received late. In contrast, an event trace captures the relative timing of events, and

would show that the send operations started late and caused the receive operations to block.

Several case studies indicate the need for tracing tools that can scale to large numbers of concurrent processes, because there are instances when a performance problem only arises after scaling the execution beyond a certain point. Kale et al. studied the performance of the NAMD application and found that several performance issues only appeared when the application was scaled above 1000 processors [32], and a new performance issue appeared after scaling above 2000 processors. A developer working with the ViSUS code encountered a hang in the program only after it was scaled to at least 8192 tasks [6]. A scientist working on the CCSM code reported intermittent hangs with 472 processes [6]. Several researchers examined the performance of the SAGE benchmark on ASCI Q [51] and noted a striking divergence from the performance predicted by their model when they scaled the application above 512 processors.

1.1.1.2 The Scalability Problems of Event Tracing

Although the information obtained from tracing is needed for correctly diagnosing certain types of performance problems, three key issues prevent it from being a scalable performance measurement technique: perturbation of the application, the large volume of data collected, and difficulties in analyzing the highly-detailed data.

Perturbation of the measured program is caused by the execution of added trace instrumentation instructions, the memory used by the trace buffer, and the flushing the trace buffer to disk. These perturbations increase the execution time of the program

and have the potential to alter the program's behavior [40]. For example, in one of our experiments, a traced run of Sphot [1] took roughly 50 times longer to execute than the untraced run.

Event tracing has the potential to create prohibitively large data files, especially for highly-parallel, long-running programs. Several researchers have noted this problem [65, 79]. As an example, in one study, we encountered event counts on the order of 10^{10} , for 32-process runs of Sphot that only ran for a few minutes [45]. The file size of the merged trace was 424 GB. We were fortunate, because the system we used provided a ~250 TB file system with no individual quotas for temporary storage. We wrote the traces to this file system and transferred them to tape for long-term storage. If we had not had these resources, we would not have been able to conduct many of our experiments.

Large trace files pose a challenge to analysis tools. They require significant amounts of memory and computation for merging, opening, and displaying the traces. Commonly, during a traced execution, each individual process writes data to its own trace file. At some point, either at the end of the execution, or as a post-mortem step, the individual trace files are merged into a single file, ordering the events from different processes by their time stamps. This merging can be computationally intensive; in our experiments, we found that the merging could take orders of magnitude more time than the execution time of the application. The act of simply opening and displaying a trace file is problematic as well. Generally, a trace visualization tool needs to scan the entire trace into memory as a preprocessing step.

In one of our studies, opening a 600 MB trace from a four-process run proved to be impossible for one trace-analysis tool. After 30 minutes of waiting, the tool reported that it had run out of memory and could not open the file. After a time-consuming conversion to a different trace file format, a different tool successfully opened the file, but required a parallel back-end to do so.

The level of detail produced by tracing makes human analysis of the data a significant task. Locating performance problems by looking at a display of a trace of hundreds or thousands of processes could fairly be described as finding a needle in a haystack. Current trace visualization tools commonly present Gantt charts, showing a bar plot of event occurrences over time, left to right, with one bar per process or task. Generally, the visualization initially shows the entire timeline, and the user has the option to zoom in on portions of the timeline, and possibly on specific ranks, to see more detail. At the high end, full-scale trace visualizations become extremely difficult to read, as the tool user must scroll through thousands of processes and lengthy time lines. It becomes a matter of either being able to see the whole picture, but not being able to see enough detail to draw conclusions about patterns in the trace; or being able to see the needed details, but losing the perspective of the whole picture.

1.1.1.3 Case studies illustrating the problems of tracing

Other researchers using tracing tools for performance analysis have described the scalability problems of tracing, which, in some cases, prevented them from performing their experiments.

One researcher we corresponded with recounted a particularly “painful” experience trying to trace a communication pattern that occurred several hours after the start of the execution of his application. He performed only a few experiments for comparison, because he ran out of quota, in spite of the fact that he had taken measures to reduce the amount of trace data collected, by not starting the trace until several hours into the execution of the program, and stopping it immediately after the target iteration. Each reduced run generated several GB of data. The trace analysis tool took a couple of hours to open and display a single trace file. (John May, personal communication)

Winstead et al. used a tracing tool to study the I/O performance of an application [70]. Although their small test runs had no problems, when they scaled up to 512 processors, several problems appeared. First, their runs generated huge amounts of data, which resulted in unwieldy trace files and significant I/O overhead in the target program. The I/O overhead had the potential to seriously perturb the loosely synchronized application. In addition, the overwhelming amount data exercised a file system bug and caused a system crash.

Chung et al. evaluated several state-of-the-art tracing tools for scalability on Blue Gene/L [13]. They found that the execution-time overhead of tracing grew faster than linearly with the number of MPI processes, and that the volume of trace data rapidly reached the order of 100 GB, which they argued was too large for efficient analysis or visualization. In their studies, they only executed up to 2048 processes, only a small fraction of the 131-thousand processor capacity of Blue Gene/L.

1.1.2 Summary

Event traces of parallel programs are an essential tool for correctly identifying the root cause of an important class of performance problems; however, the large volume of data collected creates challenges for measurement, storage, and analysis, and, in some cases, prevents measurement experiments from being conducted *at all*. The measured data is perturbed by the execution of measurement instructions, as well as by the movement of the collected data to store it on disk or to transfer it across the network. This perturbation increases the running time of the execution, and has the potential to alter the measurements by an unacceptable level. The sizes of trace files can easily reach gigabytes for even short-running executions with a small degree of parallelism. This can limit what experiments are performed, given a particular user's available file system resources. Analysis of huge amounts of data is challenging for both tools and humans. Although ad-hoc methods exist for reducing the amount of data collected in the trace, these methods require the user to partially analyze the problem and take extra steps before the measurement run. In addition, reducing the amount of data collected in these ways has the potential to miss the information needed for diagnosing the problem. Case studies show the need for tracing parallel programs at large scales, because performance problems do not always exhibit themselves during small scale runs.

1.2 Dissertation Contributions

Given the need for gathering event-based trace data for larger application runs and the scalability challenges of gathering trace data using traditional methods, our goal

was to develop a low overhead performance measurement technique for collecting event traces.

Our first task was to perform a detailed study to investigate the scalability problems of gathering traces. We used the results from the study to frame our proposed approach to a scalable method for gathering trace data on high-end systems. Our study showed that the overhead of writing the trace data to disk during the execution increased with increasing numbers of writing processes, while the overhead of trace measurement excluding the writing scaled with the amount of data being measured. The results of our study suggest the need for a measurement method that collects event-based performance information while reducing the amount of performance data, and severely limiting or eliminating the need to write any data to disk during the execution.

Our solution is a new performance measurement technique that is a hybrid between profiling and tracing, *trace profiling*. The technique produces a summary of the event details collected during a program run, and saves enough information to adequately describe the dominant performance behaviors of the execution. Because event trace data is compressed locally, the trace profiling method reduces the major source of perturbation from event trace collection on today's high end supercomputing systems: periodic flushing of trace data to disk during execution. To address the problem of large data volumes, the technique identifies event patterns that are similar enough that only one copy need be retained, thereby significantly reducing the amount of data that needs to be stored. In addition, the reduced data volume decreases the

memory and computation burden on analysis tools and the amount of data that needs to be rendered by a visualization tool.

We implemented a post-mortem prototype of the trace profiling method to illustrate the viability of the technique and to evaluate methods for deciding trace similarity. A critical piece of an implementation of the trace profiling technique is the choice of a method for deciding when traces are similar enough to be considered equivalent. Using our post-mortem implementation, we evaluated several methods for deciding trace similarity for compression, amount of error introduced into the measurements, and whether the compressed data still contained the information needed to make a correct performance diagnosis.

We implemented a prototype runtime trace profiler. We present a study of trace profiling overheads, including a comparison to traditional event trace collection.

1.3 Dissertation Organization

In Chapter 3, we present related work. We present the study of the overheads of traditional event trace collection in Chapter 4. The design of the trace profiling technique is described in Chapter 5. In Chapter 6, we demonstrate the post-mortem implementation of our technique and the evaluation of methods for deciding trace similarity. Chapter 7 describes our runtime implementation and its evaluation. Finally, we conclude in Chapter 8.

2 Related Work

Other researchers have investigated reducing or eliminating the scalability problems associated with tracing: perturbation of the application program, unmanageable file sizes, and visualization and analysis challenges.

2.1 Perturbation

Because perturbation is intrinsic to measurement [17], research focuses on techniques to lower or limit the overheads, remove the overheads in the resulting data, and to measure and model the overheads.

Researchers have investigated methods to lower the overheads of tracing [37, 50, 55, 58, 75]. The Event Monitoring Utility (EMU) was designed to allow the user to adjust how much data was collected in each trace record, thereby altering the amount of measurement overhead [37]. The authors found the writing overhead to be the largest monitoring overhead. Falcon has several features to reduce the amount of perturbation in the target program [23]. The buffer sizes used for tracing can be adjusted at run time, and it uses double-buffering to reduce the overhead of transmitting the event data to the tool monitor threads. It also allows the type of performance measurement to be altered at run time, switching between high- and low-overhead measurement techniques on the fly. The EventSpace tool has several features designed to lower the overheads of gathering traces [8]. The trace buffer is only accessed as needed by the monitoring threads. The trace buffers have a fixed size; the oldest entries are discarded to make room for new entries. This means that trace data is

not stored permanently unless it is read by the monitor threads. Also, the tool employs distributed data analysis to reduce the overhead of sending the trace data on the network.

Several researchers have developed techniques to attempt to remove overheads from the reported data [14, 20, 69, 72, 76]. Yan and Listgarten [76] specifically addressed the overhead of writing the trace buffer to disk in AIMS by generating an event marker for these write operations and removing the overhead in a post-processing step.

Several researchers have reported on the overheads of tracing. Yan and Schmidt argued that the most intrusive activities were the allocation of memory buffers to save the trace buffer and the periodic flushing of the trace buffer to disk [79]. Gu et al. reported that the most expensive operations were event buffering and transmission [23]. Chung et al [13] evaluate several profiling and tracing tools on BG/L in terms of total overhead and write bandwidth, and note that the overheads of tracing are high and that the resulting trace files are unmanageably large. They suggest that the execution time overhead is substantially affected by generation of trace file output, but provide no measurements for their claim.

Two research efforts have developed models of the overheads in measurement systems. Malony et al. developed a model to describe the overheads of trace data and describe the possible results of measurement perturbation [40], then extended it to cover the overheads of SPMD programs [57]. They assumed that in the case of programs that do not communicate, the perturbation effect for each processor is only

due to the events that occur on that processor. However, they noted, as we do, that the execution time of traced programs was influenced by other factors than just the events in each processor independently. They did not explore this further. Waheed et al. [67] explored the overheads of trace buffer flushing and modeled two different flushing policies [66]. They found that the differences between the policies decreased with increased buffer sizes. Their model did not account for the interaction between writing processes when modeling the buffer flushing policies – instead, they assumed a constant latency for all writes of the trace buffer.

A primary difference between our results and prior work investigating tracing overheads is that we identify a previously unexplored scalability problem with tracing. To the best of our knowledge, while others have noted that the largest overhead of tracing is writing the data, none have shown how this overhead changes while increasing the scale of application runs.

2.2 Trace File Size Reduction

Several research efforts focus on reducing the size of the trace file. The efforts fall into two categories: trace file compression, and measuring or writing less trace data.

2.2.1 Trace File Compression

Several researchers have reported on efforts to reduce file sizes by compression. Researchers working with the AIMS performance tool noted compression of 40-50% in trace file sizes when using a binary representation of the trace, as opposed to an ASCII encoding of the trace [79]. They also found that introducing new trace records

to represent event pairs that commonly occur together, such as function entry and exit, and message events and associated message data, resulted in trace compression of 38% in an ASCII encoding of the trace. The Pablo SDDF trace file format has both ASCII and binary representations [54]. The Pablo developers reported that, in their experience, the binary representation of traces ranged from 42-75% smaller than the ASCII representation of the files [7]. The Open Trace Format (OTF) [33] uses Lib compression [16] to compress the ASCII traces either on-the-fly or as a post-processing step. The OTF developers found that OTF compressed trace file sizes were about half the size of STF trace files for the applications they examined. While these compression methods do reduce the size of trace files, the size of the traces still scales with the number of events measured, determined by the number of concurrent processes and the length of the program run. Gamblin et al. use the CDF 9/7 wavelet transform to compress traces collected for the purposes of detecting load imbalance [18]. Knüpfer developed a method called Compressed Complete Call Graphs (CCGs) that takes a trace file and compresses it based on the event stream and event measurements to ease the burden on trace analysis tools [34]. These two methods require that all data be collected before compression begins, which means that the problems of collecting and storing a large amount of data still exist.

2.2.2 Measuring or Writing Less Data

This section details methods for reducing the amount of data collected by measuring or writing less data. These techniques fall into three categories: simple methods that omit data, methods that alter the type of measurement employed based

on some rule, and methods that decide when sections of traces are similar and measure or store a reduced number of pattern executions.

2.2.2.1 Simple Omission Methods

Generally, tracing tools provide API calls that give the user the option of starting and stopping tracing of the application at any point during the execution [5, 2, 71, 46, 52, 58, 60, 63, 74, 80]. This makes it possible for users to reduce the amount of data that is collected and to potentially reduce the size of the trace data files to reasonable levels. Unfortunately, there is a risk that this method might cause the trace to omit critical information needed for diagnosing the performance problem, and it increases the burden on the tool user to identify the approximate location of the problem, and to make code changes to control which events are captured.

TAU [58] reduces the amount of data collected by allowing users to disable instrumentation in routines that are called very frequently and have short duration. TAU also includes a tool called `tau_reduce` that uses profile data to discover which functions should not be instrumented in a user program, and feeds this information to the automatic source instrumentor. Here, the size of the trace file still scales with the number of concurrent processes and the length of the run.

2.2.2.2 Altering Measurement Type Methods

Three tools alter the type of performance data collected to reduce file size. Pablo [7] gives users the option of specifying an event-rate threshold. If an event occurs at a greater rate than the threshold, a less invasive method of measurement, such as event

counting, is employed. Vetter presents a method for statistically sampling MPI events [65]. Each time an MPI event is encountered, it is either sampled or not. For each sampled event, the tool can record statistics, log the event to a trace file, or even ignore the data. Falcon provides a choice of measurement sensors: sampling, tracing, and extended, that can be interchanged at runtime to flexibly alter the amount of data collected [23]. An example is sampling performance until a problem is detected and then turning on tracing to get more detailed information. Although each of these methods reduces the amount of data collected, they do so at the risk that some important performance behaviors will be missed.

2.2.2.3 Trace Similarity Methods

Several researchers reduce trace file size by deciding when sections of traces are similar enough that a reduced number of copies of the section need to be retained. Methods in this category include deletion of similar trace sections; trace sampling; statistical clustering; and signal processing.

Some researchers use a combination of event names and measurements to decide when traces are similar. Knüpfer and Spooner define two sections of traces as similar if the call graph context and measurements of the events are equal. Knüpfer defines equality using both relative and absolute differences [34]; Spooner et al. use the relative difference in instruction counts [60].

Another approach defines similarity by event names. By ignoring event measurements, this approach has the potential to miss important performance behaviors if there is performance variability in different iterations of the same event

stream. Chung et al. use a filter that detects repeated communication patterns [13]; they keep performance data for only one instance of each pattern. Freitag et al. use a periodicity detector to notice repeating sequences of events and keep a reduced number of iterations of each sequence [15]. Similarly, Yan and Schmidt detect repeating sequences of events and store the average measurements of those events [79]. Noeth and Mueller also detect repeated sequences of message-passing events and store one copy of each sequence; they optionally store summary information about the events, such as average measurements [49]. In later work, they include the ability to store more detailed timing information: statistical “delta” times, histograms, or histograms by call sequence [53].

Other efforts use trace sampling to reduce trace size. Carrington et al. use trace sampling to reduce the amount of time it takes to gather memory reference traces for the purpose of performance modeling [10]. They collect data for a reduced number of executions of the basic blocks in a program. Vetter presents a method for statistically sampling MPI events [65]. Each time an MPI event is encountered, it is either sampled or not. Gamblin et al. use statistical sampling with a user-specified confidence interval and metric. [19]. Although sampling methods do reduce the amount of trace data collected, they have the potential to miss critical performance behaviors that occur during unmeasured portions of the program.

Aguilera et al. [4], Nickolayev et al.[48], and Lee et al. [39] apply statistical clustering to traces and select a representative trace for each cluster of processes. Nickolayev and Lee use the Euclidean distance for clustering, while Aguilera uses a

metric based on the amount of communication between two processes. These clustering methods reduce trace data across processes, but do not reduce trace data within a process (temporal reduction). As a result, file sizes will still scale with the running time of the application.

Several groups apply methods from signal processing to traces. Casas et al. and Huffmire et al. use the Haar wavelet transform to automatically determine the phases of a program [11, 30]. Hauswirth et al. use dynamic time warping to decide when two traces are similar for aligning multiple traces [26].

Researchers have evaluated several methods for deciding the goodness of a particular trace similarity metric. Ratn et al. use aggregate statistical measures, such as total time spent in a function, to evaluate their method [53]. Gamblin et al. compute a trace confidence measure to evaluate their trace sampling results, which tells the percentage of time the mean trace of sampled processes is within an specified error bound of the mean trace of the full trace [19]. In their wavelet transform method, Gamblin et al. use a root mean square measure to estimate the error in reduced traces [18]. They also present qualitative results, showing a visualization based on a reduced trace compared with one from a complete trace. Yan et al. compare the measurements in their reduced trace against the real trace time stamp by time stamp and produce both a relative and absolute measure of the overall differences [77]. In addition, they also present whole program statistical measurements and visualizations for qualitative comparison.

2.3 Analysis Tool and Visualization Scalability

Three trace file formats address the scalability problems faced by trace analysis tools. The Scalable Logfile Format (SLOG) was developed to address the scalability problems encountered by visualization tools [73]. In SLOG, events are partitioned into intervals called Bounding Boxes, which are organized into a binary tree. This organization of the data allows the visualization tool to display a low-resolution representation of the trace data without reading in the entire trace file. The Structured Trace Format (STF) was designed to write the data to multiple files to allow the files to be read and written in parallel [2]. Their goals were to make the format be as compact as possible and allow for fast random access to the data and easy extraction of the data. The Open Trace Format (OTF) was designed to address the challenges that come with the ever-increasing scales of HPC platforms [33]. It was designed so that the trace could be processed by a parallel backend, which reduces the time to open and visualize very large trace files. OTF uses an ASCII encoding which enables a tool to do a binary search on files for time intervals.

Several researchers have worked to reduce the amount of data presented to the user in order to facilitate understanding of the performance of the program. Vetter presents a method for identifying communication inefficiencies by applying machine learning techniques to trace files of MPI communication events [64]. The end result is a breakdown of the communication events that were considered “normal” and “abnormal” (e.g. late sends or late receives), and the location in the source code from which they were called. The AIMS performance tool suite computes performance

indices from parallel program traces [78]. Performance indices are designed to quantify program characteristics to locate bottlenecks. An example of a performance index is the communication overhead index, which gives an indication of how much of the program's execution time was spent in communication activities. Although both of these methods greatly reduce the amount of data presented to the user and help to identify performance problems, neither method shows causal information for the problems.

The scalability of trace visualizations is not a new topic [24, 28, 27, 41, 47]; however, the continuing upward scaling of high end systems drives a continuing need for more scalable solutions. Knüpfer et al. show how CCGs can be used to facilitate visual understanding of trace data [35]. Color blocks that can be interactively decomposed represent behavior patterns in the execution. The visualization scales with the number of parallel entities in the execution and with the running time of the execution. Spooner and Kerbyson present a tool that takes multiple traces as input and outputs visualizations that highlight the differences between the traces [60]. Their primary goal was to generate visualizations that indicate performance differences in multiple executions over time. They note that their tool could be used to compare iterations within a single execution; however, this is achieved by either creating a separate trace file for each iteration or extracting the iteration data from the trace as a post-mortem step. Neither of these methods addresses the problem of collecting and storing the possibly enormous amount of trace data.

3 Study of Tracing Overheads

We conducted a measurement study to discover the scalability challenges in event tracing. We used the results of this study as a guide when designing our low-overhead approach to gathering event traces.

3.1 Experiment Design

Our experiments are designed to focus on separating runtime tracing overhead into two distinct components: the overhead of just the trace instrumentation, and the overhead of flushing the trace buffer contents to files. We performed runs with and without trace instrumentation (*instr* and *noInstr*); and with and without buffer flush to file enabled (*write* or *noWrite*), then calculated the overheads using the following metrics:

- *Wall clock time*: `MPI_Wtime`, measured after `MPI_Init` and before `MPI_Finalize`. The following are not included in this measurement: instrumentation overhead for tool setup, finalization, and function calls before/after the timer is started/stopped; and writing overhead for trace file creations, final trace buffer flushes before file closure, trace file closure, and, in the case of MPE, trace file merging.
- *Write overhead*: Average total wall clock time of the *write* runs minus average total wall clock time of the *noWrite* runs

- *Instrumentation overhead*: Average total wall clock time of the runs that did not write the trace buffer minus average total wall clock time of the *noBuff_noInstr_noWrite* runs

Given our goal of pushing to the current scaling limits of tracing, we wanted to measure an application with a very high rate of communication, so that trace records for a high number of MPI communication events would be generated. We picked SMG2000 (SMG) [9] from the ASC Purple Benchmark suite. SMG is characterized by an extremely high rate of messages: in our four process runs, SMG executed 434,272 send and receive calls in executions that took approximately 15 seconds. For comparison, we also included another ASC Purple Benchmark, SPhot (SP) [1]. SP is an embarrassingly parallel application; in a four-process, single-threaded execution of 512 runs with a total execution time of 350 seconds, the worker processes pass 642 messages, and the master process passes 1926 messages. We configured both applications with one thread per MPI process.

To vary the number of processes, we used weak scaling for the SMG runs. As we increased the number of processors, we altered the processor topology to $P * 1 * 1$, where P is the number of processors in the run, and kept the problem size per processor, $n_x * n_y * n_z$, the same, thereby increasing the total problem size. We used both weak and strong scaling for the SP runs, referred to as SPW and SPS respectively. We configured these approaches by changing the Nruns parameter in the input file input.dat, which controls the total amount of work done in a single

execution. For strong scaling, we kept Nruns constant at 512 for all processor counts; for weak scaling, we set Nruns equal to the number of MPI ranks.

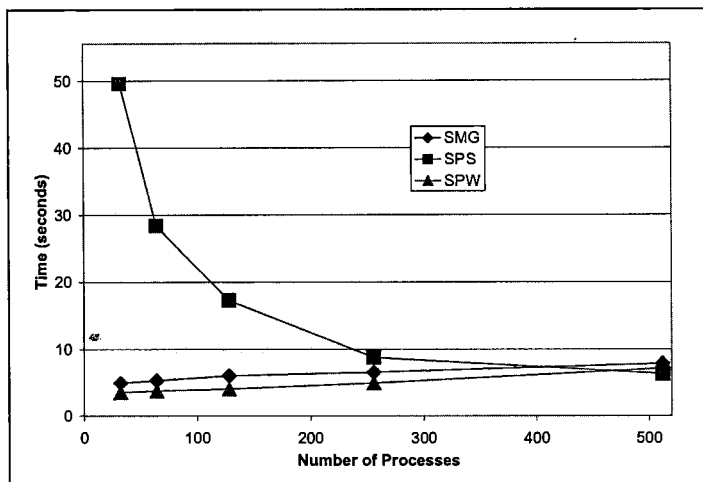


Figure 1 Performance of Uninstrumented Executions

We used SMG’s built-in metrics to measure Wall Clock Time, summing the values reported for the three phases of the execution: Struct Interface, SMG Setup, and SMG Solve. We used the native SPhot wall clock time values for Wall Clock Time. Figure 1 shows the scaling behavior of the uninstrumented applications. As expected, the execution time of SPS decreases with increasing numbers of processors, since we are keeping the total problem size constant.

In some sense the choice of a particular tracing tool was irrelevant to our goals: we wanted to investigate a “typical” tracing tool. However, we wanted to avoid results that were in some way an artifact of one tool’s particular optimizations. Therefore, we used two different robust and commonly used tracing tools for our experiments: TAU and MPE.

We built several versions of TAU version 2.15.1 [58]. For the *noWrite* versions we commented out the one line in the trace buffer flush routine of the TAU source that actually calls the `write` system call. We altered the number of records stored in the trace buffer between flushes, by changing the `#define` for `TAU_MAX_RECORDS` in the TAU source for each size and rebuilding, to test two different buffer sizes: 0.75 MB (32,768 TAU events); 1.5 MB (default size for TAU; 65,536 TAU events); 3.0 MB (131,02 TAU events); and 8.0 MB (349,526 TAU events). We used the default level of instrumentation for TAU, which instruments all function entries and exits.

MPE (the MultiProcessing Environment (MPE2) version 1.0.3p1 [80]) uses the MPI profiling interface to capture the entry and exit time of MPI functions as well as details about the messages that are passed between processes, such as the communicator used. To produce an MPE library that did not write the trace buffer to disk, we commented out three calls to `write` in the MPE logging source code. We also had to comment out one call to `CLOG_Converge_sort` because it caused a segmentation fault when there was no data in the trace files. This function is called in the MPE wrapper for `MPI_Finalize`, so it did not contribute to the timings reported in the SMG metrics. We altered the buffer sizes by changing the value of the environment variable `CLOG_BUFFERED_BLOCKS`. We also set the environment variable `MPE_LOG_OVERHEAD` to “no” so that MPE did not log events corresponding to the writing of the trace buffer. In MPE, each MPI process writes its own temporary trace file. During `MPI_Finalize`, these temporary trace files are merged into one trace file, and the temporary trace files are deleted. The temporary

and merged trace files were written in CLOG2 format. We used two different buffer sizes: 1.5 MB (24 CLOG buffered blocks), and 8.0 MB (default size for MPE; 128

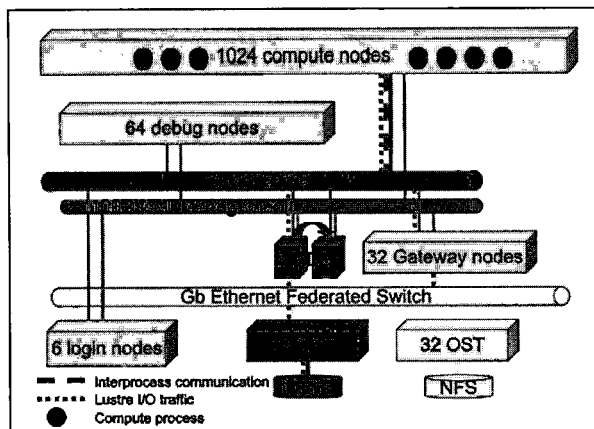


Figure 2 Experiment Environment.

The MPI processes in our experiments, represented by purple circles in the diagram, ran on a subset of the 1024 compute nodes of MCR. MPI communication between the processes traveled over the Quadrics QsNet Elan3 interconnect, shown by the purple dashed line. The I/O traffic for the Lustre file system, represented by the blue dotted line, also traveled over the Quadrics interconnect. Metadata requests went to one of two metadata servers (MDS), a fail-over pair. File data requests first went through the gateway nodes to an object storage target (OST), which handled completing the request on the actual parallel file system hardware.

CLOG buffered blocks). For SPW only, we altered the SPhot source to call MPE logging library routines to log events for all function calls, to correspond to the default TAU behavior more directly. We refer to this as “MPc” for MPE with customized logging. For the SPW MPc experiments, we disabled the trace file merge step in `MPI_Finalize`, because it became quite time consuming with larger trace files.

We collected all of our results on MCR, a 1152-node Linux cluster at LLNL running the CHAOS operating system [21] (See Figure 2). Each node comprises two 2.4 GHz Pentium Xeon processors and 4 GB of memory. All executions ran on the

batch partition of MCR. The trace files, including any temporary files, were stored using the Lustre file system [61]. This platform is representative of many high end Linux clusters in current use.

Each of our experiment sets consisted of thirty identical executions.

3.2 Results

In this section, we present results from a study of tracing overheads as we scale up the number of application processes. These results are part of a larger investigation; full details are available as a technical report [44]. In this study, we examined how the overheads of tracing change as the application scales. We ran sets of experiments with 32, 64, 128, 256, and 512 processes, traced with TAU and MPE, using buffer sizes of 1.5 and 8.0 MB.

3.2.1 Event Counts and Trace File Sizes

Here we describe the event counts generated while tracing the applications. Complete details can be found in the technical report [44]. For SMG, the counts for TAU and MPE exhibit similar trends, but are different by roughly an order of magnitude. As the numbers of processors double, the per-process event counts and trace data written by each process increase slightly (in part due to increased communication), while the total number of events and resulting trace file sizes double. For SPS, there are markedly different results between TAU and MPE; the event counts differ by six orders of magnitude. This is because with TAU we are measuring all function entries and exits, whereas with MPE we measure only MPI activity. For both

TAU and MPE, doubling the number of processors results in the per-process event counts decreasing by half.

For TAU only, the total event count and resulting trace file sizes remain constant, whereas for MPE, the maximum per-process event count, the total event count, and resulting trace file sizes increase slightly. For SPW, the counts for TAU and MPC are nearly identical, while the counts for MPE differ. Again, this is because of differences in what was measured by the tools. The total event count and trace file sizes for MPE are roughly six orders of magnitude less than those of TAU and MPC.

We use this information to derive an expectation for tracing overheads for the different applications and tools. For the weakly-scaled SMG and SPW, we expect that the overheads of tracing would remain relatively constant with increasing numbers of processors because the amount of data being collected and written per-process remains relatively constant. However, for SPW with MPE, we expect to see very little overheads due to the small amount of data collected. For SPS and TAU, we expect the overheads of tracing to decrease with increasing numbers of processors, because the amount of data being collected and written per-process decreases with increasing processes. For SPS with MPE, we expect to see very little overhead because of the small amount of data collected.

3.2.2 Execution Time

Figure 3 shows the average wall clock times for our experiments broken down into time spent in application code, trace instrumentation, and writing the trace buffer. The graph on the left shows the measurements for SMG with TAU and MPE, and SPW

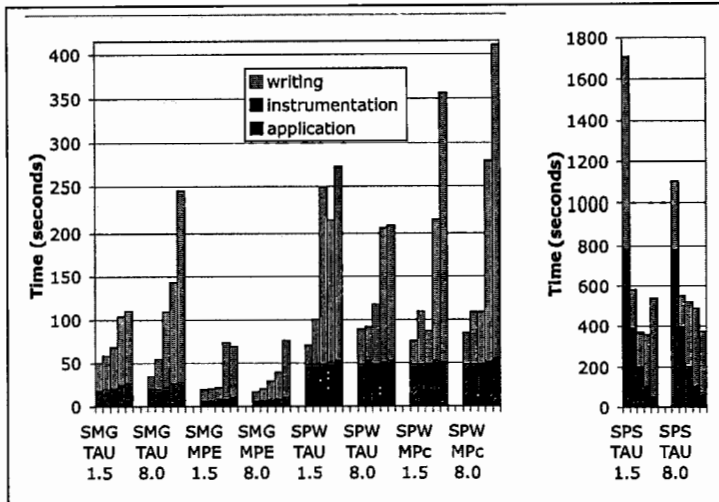


Figure 3 Performance of Instrumented Executions.

Here we show the total execution time for SMG measured with TAU and MPE, and SPW measured with TAU. The colors in the bars indicate the time spent in application code, time in trace instrumentation, and time writing the trace buffer. Each bar in a set represents the average behavior of executions with 32, 64, 128, 256, and 512 processes, respectively. The set labels include (top to bottom): the benchmark name, the measurement tool, and the buffer size.

with TAU and MPC. In each run set, we see the same trend; as the number of processes increases, the total execution time increases, largely due to the time spent writing the trace buffer. The time spent in the application code and in trace instrumentation remains relatively constant. The graph on the right shows the execution times of SPS with TAU. Here, as the numbers of processes increase, the total execution time decreases. However, even though the time spent in writing the trace buffer decreases with increasing processors, it does not decrease as rapidly as the time spent in instrumentation or application code. For SPS and SPW with MPE, the differences between the *write* and *noWrite* executions were indistinguishable due to the very small amounts of data collected and written.

Table 1 Correlation of Total Wall Time with Maximum Event Count in a Rank

		SMG		SPS		SPW		
Buffer Sz	Write?	TAU	MPE	TAU	MPE	TAU	MPE-C	MPE
1.5	yes	0.96	0.85	0.91	-0.78	0.69	0.80	0.98
8.0	yes	0.97	0.90	0.95	-0.81	0.61	0.76	0.98
1.5	no	0.98	0.98	0.99	-0.70	0.81	0.55	0.96
8.0	no	0.98	0.98	0.99	-0.79	0.74	0.77	0.95

We computed the percentage contribution to variation using three-factor ANOVA, with the buffer size, the number of processes, and whether or not the trace buffer was written to disk as the factors [44]. In general, there was quite a bit of variation in the running times of the executions that wrote the trace buffer, which explains the high contribution of the residuals. Sources of variability in writing times for the different executions include: contention for file system resources, either by competing processes in the same execution, or by other users of Lustre; contention for network resources, either by other I/O operations to Lustre, or by MPI communication; and operating system or daemon interference during the write. Any user of this system gathering trace data would be subject to these sources of variation in their measurements. For SMG measured with TAU and MPE, the largest contributing factor was whether or not the buffer was written, at 33% and 26%, respectively. The largest contributing factor for SPS with TAU was the number of processes in the run (19%), followed closely by whether or not the trace buffer was written (14%). SPS with MPE had the number of processes as the dominating factor at 51%. SPW with TAU and MPE both had writing the trace buffer as the largest contributor, at 34% and 24%, while SPW with MPE had the number of processes as the largest, at 81%. The

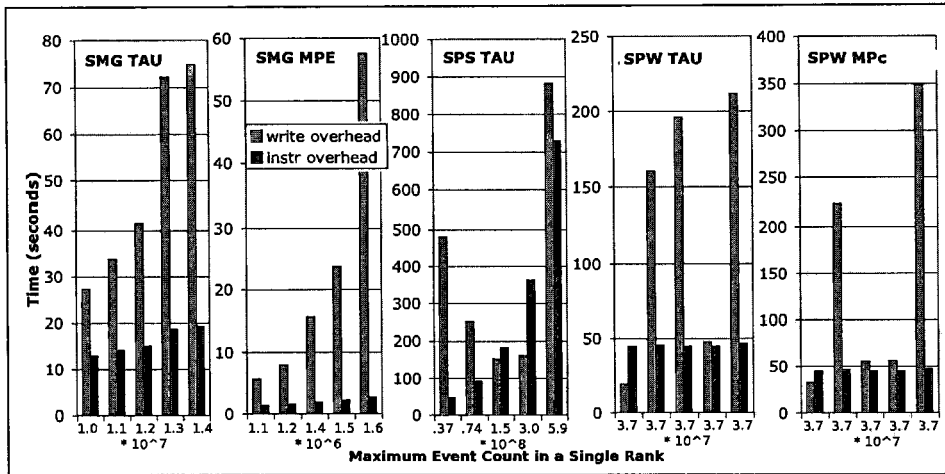


Figure 4 Tracing Overhead with Maximum Event Count in a Single Rank.

The groups of bars from left to right in the charts represent different processor counts: for SMG they represent 32, 64, 128, 256, and 512 processes; for SPS they represent 512, 256, 128, 64, and 32 processes; for SPW, they represent 32, 256, 128, 64, and 512 processes.

differences in the dominating factors for the SP runs with MPE are attributed to the comparatively very small amount of data collected.

3.2.3 Execution Time vs Event Counts

Table 1 shows the correlation of the average total wall clock time with the maximum event count over all ranks. SPS with MPE had a relatively weak negative correlation with the maximum event count, because as the process count increases, the number of messages that the master process receives increases, and the execution time decreases, giving a negative correlation. In general, executions that did not write the trace buffer to disk had a higher correlation with the event count than did the executions that did write the trace buffer to disk.

Figure 4 shows the overheads of writing and instrumentation as the maximum number of events in a single rank increases. For SMG with TAU and MPE, we see a

clear pattern. The instrumentation overhead appears to vary linearly with the number of events, while the overhead of writing the trace increases much more rapidly, and does not appear to have a linear relationship with the event count. The behavior of SPS is different, because in this application, as the number of events increases, the number of processes decreases; however, the instrumentation overhead still appears to have a linear relationship with the event count. The write overhead is high at higher event counts, but also at the low event counts, when the number of writing processes is higher. For SPW, the instrumentation overhead is relatively constant, as expected since the number of events does not change much between the run sets. However, the writing overhead fluctuates widely. The reason for this is that the maximum event count in a rank does not monotonically increase or decrease with increasing processors as it does for SMG or SPS.

3.3 Conclusions

In our scaling experiments, the execution times of the *noWrite* runs tended to scale with the maximum number of events. However, the execution times of the *write* runs did not scale as strongly with the number of events, and tended to scale with increasing numbers of processors, possibly due to contention caused by sharing the file system resource. Our results suggest that the trace writes will dominate the overheads more and more with increasing numbers of processes. They indicate that the trace overheads are sensitive to the underlying file system.

Realization of a scalable approach to tracing will require an overall reduction in the total amount of data. Data reduction is needed not only to reduce runtime

overhead, but also to address the difficulties of storing and analyzing the resulting files. We incorporated the results of our measurement studies into the design of our approach to low-overhead event tracing.

4 Trace Profiling

We have developed a novel approach to performance measurement designed to address the scalability problems of gathering event-based data. Our approach is a hybrid between profiling and tracing that we call trace profiling. The goal of trace profiling is to gather enough information to adequately describe the dominant performance behaviors of the execution, at a greatly reduced data volume than gathered by a traditional tracing tool. The trace profiling technique detects event patterns, or segments, in the execution trace that have similar behavior. Segments with similar behavior are merged, so that only one copy of the segment is retained. Thus, a trace profile contains a summary of the event patterns that occurred during program execution.

In Section 5.1, we start by describing traditional event trace collection in order to provide background for explaining and evaluating trace profiling. Next, in Section 5.2, we describe the trace profiling technique. We present an overview of the technique followed by our methodology for marking segments in traces and for segment merging. In Section 5.3 we detail the methods we use for detecting segments with similar behavior. In Section 5.4, we present models for predicting the sizes of traditional traces and trace profiles; in Section 5.5, we use the models to predict the size reduction achievable by trace profiling and to compare traditional tracing and trace profiling. Finally, in Section 5.6, we illustrate the potential benefits of trace profiling for visualization and analysis tools.

4.1 Background

Traditional tracing results in an in-order listing of the events that occurred during an application run. Generally speaking, a traditional tracing tool creates a record for each event encountered during the execution and stores it in a buffer in memory. When the buffer becomes full, the contents of the buffer are flushed to disk, and the buffer is reused. With most tracing tools, each process creates its own event trace; the individual traces can optionally be merged at the end of the execution.

A traditional event trace of a parallel program contains two types of information: a mapping of event identifiers to the event names, e.g. the function `main` might have identifier 1; and a series of records that contain data about program events. In this document, we will call the mapping of event identifiers to event names an event map. The event map can reside in the same file as the event records or a separate file. The event records contain data about function entries or exits, message passing data, other performance measurement data, or bookkeeping information. Examples of bookkeeping records include records that indicate the start and stopping times of flushing the trace buffer. For function events, there is a separate record each for event entry and event exit. We show a diagram of example trace files for a parallel application run with two-process in Figure 5.

4.2 Trace Profiling Technique

Trace profiling is different from traditional tracing because it doesn't maintain a complete, in-order list of event entry and exit records for each process. A trace profiler

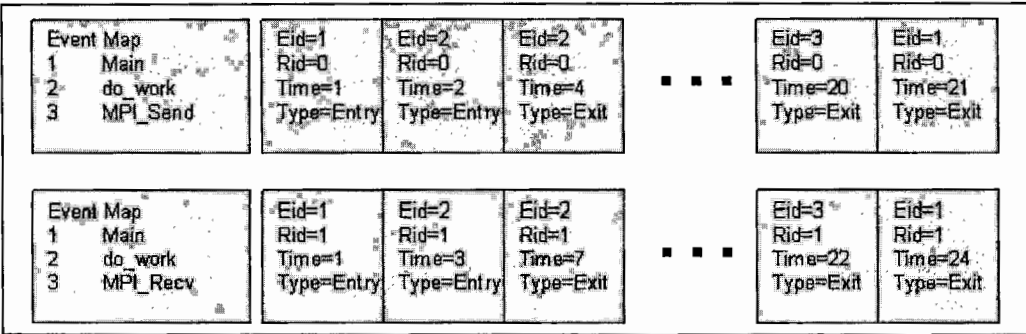


Figure 5 Data from Traditional Trace

This figure shows the data obtained from a traditional trace. Each process outputs its own event map and event records file(s). The event records file is simply an in-order series of event records gathered from start of the program to the end of the program. In the above diagram, Eid refers to the event identifier given in the event map and Rid is the rank identifier. Each record has a timestamp and indicates the event type, e.g. entry or exit.

partitions the processes of the parallel program into process groups. A process group contains the performance information for one or more processes that had the “same” behavior. Each process group contains a list of segments. A segment is simply an in-order series of events and their associated information for a portion (or segment) of the execution of the program. Each process group maintains a segment execution list, which is a listing of the order of segment executions and the timestamp at which each segment execution began. Each segment maintains its time duration and a listing of the events that executed in the segment. For each event, we maintain the relative starting time of the event with respect to the start of the segment execution, the event’s duration, and any other associated information, such as message passing data.

We show a representation of the data for a sample process group in Figure 6. The process group contains the data for ranks 0, 2, and 4. There were two segments in the execution that executed two times each. Segment 0 executed first at time=1 and again ,

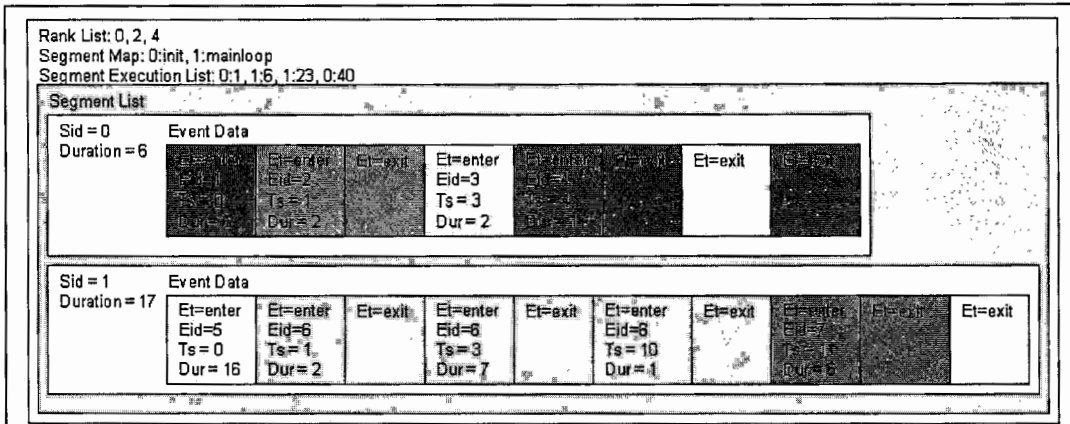


Figure 6 Process Group

This figure shows an example process group and the information it contains about segments. It maintains the ranks of the processes that it represents in the rank list. The segment map gives a mapping between segment identifiers (Sid) and the name of the segment. The segment execution list keeps track of the order of segment executions and their start times (Sid:start_time). The segments for the process group are in the segment list. Each segment has a duration and event data for the events that executed during that segment. Each event has an entry record that contains an identifier, Eid, a start time, Ts, and a duration, Dur; and an exit record.

at time=40; Segment 1 executed at time=6 and again at time=23. We keep separate entry and exit records for each event: the entry event records store the event identifier a start time relative to the start of the segment, and a duration; the end records simply mark the ending of the events so that the function call stack can be maintained.

A trace profile contains the following items:

- *Event Map*: One event map stores the mapping of event names to event identifiers for all process groups.
- *Process Groups*: Each process group contains the following:
 - *Rank List*: A list of ranks, which tells which processes' data the process group contains.

- *Segment Map*: A mapping between segment identifiers and segment names.
- *Segment Execution List*: A listing of segment identifiers and timestamps, telling the start time of each segment execution in a process group.
- *Segment List*: A list of segments for a single process group. Each segment has a header which gives each segment's identifier and duration, and is followed by a list of event data.
 - *Event Start Record*: An identifier, a timestamp which gives its start time relative to the start of the segment; a duration; and possibly one or more message data records.
 - *Message Data Record*: A type (send or receive), a rank identifier (source or destination), a tag, and a communicator.
 - *Event End Record*: A marker that indicates an event end.

We merge segments both within and across processes. Segments are merged if they are equal, as determined by a given difference method. A trace profiler can have multiple difference methods for deciding segment equality; we describe and compare several methods in Section 5.3. We describe the criteria and algorithms for intra- and inter-process merging in Sections 5.2.2 and 5.2.3.

```

int main(){
    start_segment("main_0");
    MPI_Init();
    end_segment("main_0");
    for(i=0; i < 100; ++i){
        start_segment("main_loop_1_1");
        do_work();
        MPI_Allgather();
        end_segment("main_loop_1_1");
    }
    for (j=0; j < 10; ++j){
        start_segment("main_loop_2_1");
        do_other_work();
        end_segment("main_loop_2_1");
        while(k < otherRanks){
            start_segment("main_loop_2_1_1");
            MPI_Sendrecv();
            end_segment("main_loop_2_1_1");
        }
        start_segment("main_loop_2_2");
        stop_segment("main_loop_2_2");
    }
    start_segment("main_1");
    MPI_Finalize();
    end_segment("main_1");
}

```

Figure 7 Segment Context Marking

We show a single function, `main()` with the instructions added to mark the segment contexts. We mark an initial segment at the start of `main`, all loops that contain at least one function event, and code regions surrounding marked loops. The segment context names are hierarchical: the second loop is marked "main_loop_2_1" and its subloop is marked "main_loop_2_1_1". Segment marking is automated using a dynamic instrumentation library.

4.2.1 Trace Segmentation

We insert segment markers into the source code or program binary. We define *segments* as follows: the initial segment starts at entry to `main`; for each program loop containing at least one measured event, we stop the current segment before the loop starts, start a new segment at the top of each loop iteration, stop the segment at the bottom of the loop iteration, and start a new segment after the last iteration of the loop completes; and end the final segment at program termination. The *segment context* is the section of code, for example, the `main_loop_1_1` loop in Figure 7.

4.2.2 Intra-process Segment Comparison

For intra-process trace reduction, we compare the segments for each context pair wise to determine if they are similar. If they are, we say that the segments *match* and retain a single representative segment. Each segment s_i contains an ordered list of events $E_i = \{e_0, e_1, \dots, e_m\}$. We maintain a list *storedSegments*, which contains the segments that represent the performance behaviors in the execution, and a list *segmentExecs* that holds the starting times and identifier of each representative segment so that we can later recreate a full trace. Given an equivalence operator \approx for

```

For  $i = 0$  to  $\text{len}(E_{new})$ :
     $E_{new}[i].start = E_{new}[i].start - s_{new}.start$ 
     $E_{new}[i].end = E_{new}[i].end - s_{new}.start$ 
     $s_{new}.end = s_{new}.end - s_{new}.start$ 
     $match = \text{False}$ 
    For  $i = 0$  to  $\text{len}(\text{storedSegments})$ :
         $s_{stored} = \text{storedSegments}[i]$ 
         $match = \text{compareSegments}(s_{new}, s_{stored})$ 
        If  $match = \text{True}$ :
             $\text{segmentExecs} = \text{segmentExecs} \cup (s_{stored}.id, s_{new}.start)$ 
            break
    If not  $match$ :
         $s_{new}.id = \text{getNewId}()$ 
         $\text{segmentExecs} = \text{segmentExecs} \cup (s_{new}.id, s_{new}.start)$ 
         $s_{new}.start = 0$ 
         $\text{storedSegments} = \text{storedSegments} \cup s_{new}$ 

Boolean  $\text{compareSegments}(s_{new}, s_{stored})$ :
    If  $s_{new}.context \neq s_{stored}.context$ : return False
    If  $\text{len}(E_{new}) \neq \text{len}(E_{stored})$ : return False
    For  $i = 0$  to  $\text{len}(E_{new})$ :
        If  $E_{new}[i].id \neq E_{stored}[i].id$ : return False
    If  $s_{new} \approx s_{stored}$ : return True
    Else: return False

```

Figure 8 Algorithm for Intra-process Segment Matching

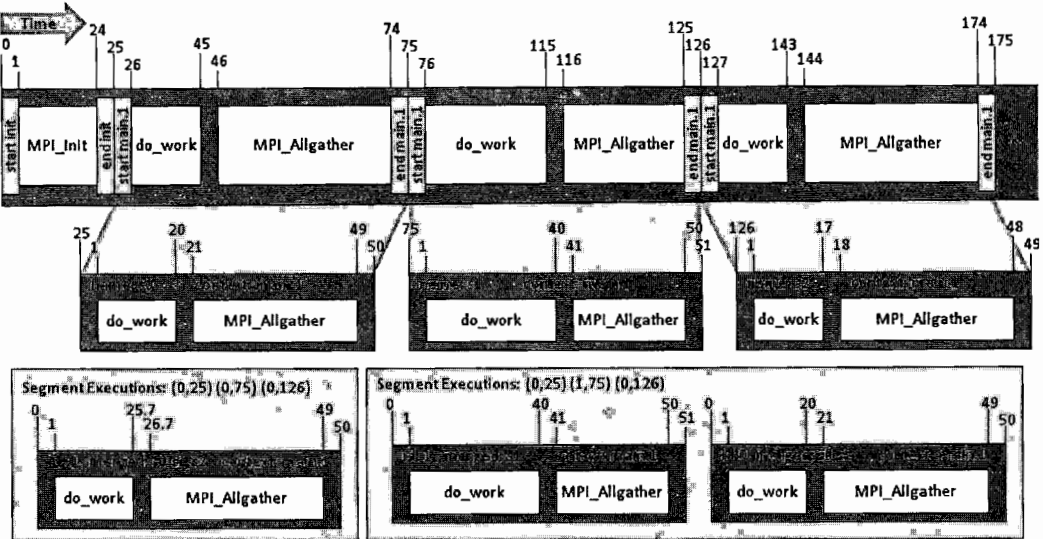


Figure 9 Intra-process Segment Matching

Here we show a portion of an example trace and three segments to illustrate segment matching. The top bar represents a portion of a trace for the program in Figure 7. Time increases from left to right, and time values are indicated above the bar. Segments markers are shown as light gray rectangles with vertical text that indicates the context of the segment. Events are shown in white boxes. Below the trace, we show the result of segmentation. In each of the three segments, the time stamps for the events and ending time of segments are adjusted relative to the start time of the segment. We name the segments s_0 , s_1 , and s_2 . In the bottom row, we show two examples of segment matching (See Section 5.3.).

some similarity metric, and a segment s_{new} that has events E_{new} the algorithm comparing segments is shown in Figure 8. Note that a segments match requires that segments have the same context and the same number of events occurring in the same order.

4.2.3 Inter-process Segment Comparison

For inter-process trace reduction, we compare the stored segments lists that were collected for each process. Initially, each trace profile contains data for a single process group, each of which only contains data for a single rank. Given two trace

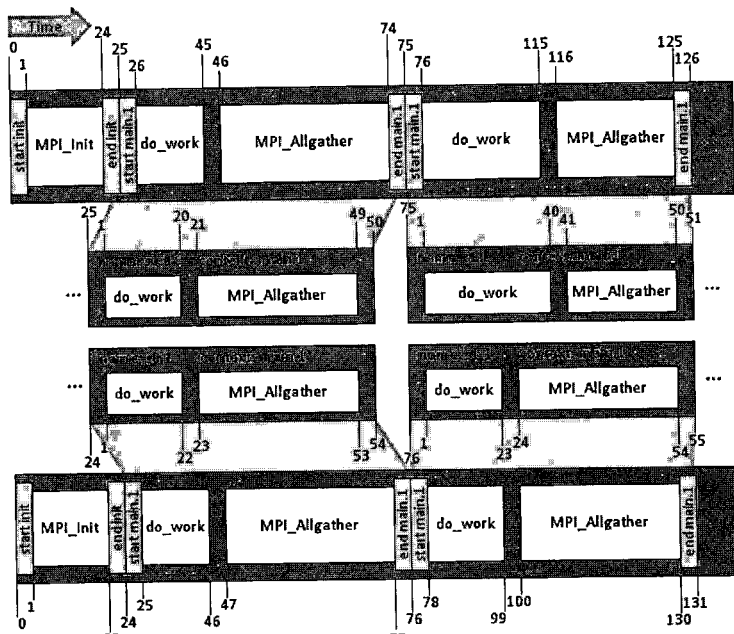


Figure 10 Inter-process Segment Matching

The top and bottom bars represent traces for different ranks of the program in Figure 7. Time values on the bars increase from left to right. Segments markers are gray rectangles with text that tells the segment context. Events are white boxes. Between the traces, we show the result of segmentation. We name the segments $s0.x$ and $s1.x$; x indicates the rank that wrote the trace. In the segments, the time stamps for the events and segment end times are adjusted relative to the segment start time. To decide matching, we examine the segments pairwise in order, comparing segment start times and all event timings.

profiles with equal numbers of segments, we compare each pair of segments in order and determine if they are similar. If all segments in both traces are deemed similar, we say that the trace profiles *match*, add the new process rank identifier into the process group, and retain a single *representative trace profile* for the process group. After comparing all trace profiles, we end up with a set of representative trace profiles, one for each process group. We give an example of trace matching in Figure 10. In

```

Boolean compareProcessGroups( $P_0, P_1$ )
   $SE_0 = P_0.segmentExecs$ 
   $SE_1 = P_1.segmentExecs$ 
  If  $len(SE_0) \neq len(SE_1)$ : return False
  For  $i = 0; i < len(SE_0); ++i$ :
     $id_0:time_0 = SE_0[i]$ 
     $id_1:time_1 = SE_1[i]$ 
    If  $time_0 \neq time_1$ : return False
     $S_0 = P_0.segments$ 
     $S_1 = P_1.segments$ 
     $s_0 = S_0[id_0]$ 
     $s_1 = S_1[id_1]$ 
     $match = compareSegments(s_0, s_1)$ 
    If not  $match$ : return False
  return True

Boolean compareSegments( $s_0, s_1$ ):
  If  $s_0.context \neq s_1.context$ : return False
  If  $len(E_0) \neq len(E_1)$ : return False
  For  $i = 0$  to  $len(E_0)$ :
    If  $E_0[i].id \neq E_1[i].id$ : return False
  If  $s_0 \approx s_1$ : return True
  Else: return False

```

Figure 11 Algorithm for Inter-process Matching

addition to comparing event measurements, we also check message passing parameters: source/target rank, bytes transferred, message tags, and communicators. All parameters save the source/target rank must be identical; the source/target rank can be either the same offset, e.g. rank+1 in a nearest neighbor communication pattern, or the same rank, e.g. all ranks send to rank 0.

To compare two process groups P_0 and P_1 , with respective segment execution lists, SE_0 and SE_1 , where $SE_i = \{id_0:time_0, id_1:time_1, \dots, id_k:time_k\}$, and stored segment lists, S_0 and S_1 , where $S_j = \{s_0, s_1, \dots, s_m\}$, we follow the algorithm in Figure 11.

4.3 Trace Profile Segment Comparison Methods

We used several methods to decide the similarity of segments. Each of these is described below. Our choices were inspired by methods used by other researchers to reduce traces (See Chapter 3). They fall into two categories: distance methods and iteration-based methods.

4.3.1 Distance Methods

The distance methods produce a difference measure, which is then compared against a user-supplied threshold to determine the presence or absence of a match. Several of the distance methods are standard methods for computing distances between values and sets of values. We use the relative difference (*relDiff*), absolute difference (*absDiff*), and three variations on the Minkowski distance (*Manhattan*, *Euclidean*, *Chebyshev*), and wavelet transforms (*avgWave*, *haarWave*).

4.3.1.1 Relative Difference

We compare the relative differences between each event measurement against a user-defined threshold; if greater, the events are not equal:

$$relDiff(x_1, x_2) = \frac{|x_1 - x_2|}{\max(x_1, x_2)} \quad \text{Eq. 1}$$

To see how *relDiff* matches segments, we consider our example in Figure 9. We compute the relative differences between each of the paired measurements in the segments. If any are above our chosen threshold, say 0.5, then the match fails. Comparing *s2* with *s1*, we first compare the start times of the `do_work` event: $x_1=1$ and $x_2=1$, with relative difference 0. Since the relative difference is less than 0.5, we

continue on computing relative differences. Next we check the end times for the `do_work` event. Here we compute a relative difference: $x_1=17$ and $x_2=40$, giving a relative difference of 0.58. This is above our threshold, so the segments do not match. When we compare s_2 with s_0 , we find that no differences are greater than 0.15 ($x_1=17$, $x_2=20$), so the segments match. The new segment is discarded since its behavior is reflected in the measurements in s_0 .

The relative difference function compares each measurement with its paired counterpart in isolation. The computed difference is proportional to the magnitude of the paired measurements, meaning that larger differences between larger measurements don't overshadow differences in smaller measurements. Because the difference between each measurement pair will be judged in isolation, the relative difference should be one of the strictest difference criteria in our set. The choice of threshold used will have a large bearing on the degree of matching, and hence on the reduction in file size.

One problem with *relDiff* appears when comparing time stamps in a series. For example, assume the threshold for comparing time stamps is 0.25. When we compare events that start at times 1 and 2, the relative difference is $\frac{2-1}{2} = 0.5$. This would result in a failure to match the events even though there is a difference of only one time unit between the events. In contrast, if we compare events that start at 100 and 125, the relative difference is 0.2, which is a match even though there is a difference of 25 time units. We expect *relDiff* to produce reduced traces with a low amount of error, but with less file size reduction.

4.3.1.2 Absolute Difference

As with the *relDiff*, each measurement is compared with its counterpart. A fixed size difference, determined by a threshold, is allowed for each measurement pair. Using our example segments in Figure 9, and a threshold of 20, we see that *s2* will not match *s1*, because the end times of `do_work` are 23 time units apart. However, there are no differences larger than 3 between *s2* and *s0*, so those two segments match. The threshold choice has an impact on file size and accuracy. We expect this method to produce fairly accurate results, especially with respect to the timing of events across processes, because unlike *relDiff* it will not have an unfair bias towards events that occur later in the trace.

4.3.1.3 Minkowski Distance

We compute the Minkowski distance between segments using the formula in Eq. 2. If the distance is greater than a user-specified threshold multiplied by the maximum value in the event measurements, then the events are not equal. The Manhattan, Euclidean, and Chebyshev distances are special cases of the Minkowski distance, with *m* equal to 1, 2, and $\lim_{m \rightarrow \infty}$ respectively [25]. The Chebyshev distance is defined to be the largest difference between two measurements.

$$L_m = \left\{ \sum_{i=1}^n |x_i - y_i|^m \right\}^{1/m} \quad \text{Eq. 2}$$

Using our example in Figure 9, to compare *s2* and *s1*, we create a vector of the measurements for *s2*, (49, 1, 17, 18, 48), and one for *s1*, (51, 1, 40, 41, 50). The Manhattan, Euclidean, and Chebyshev distances between these vectors are 50, 32.6,

and 23, respectively. The largest measurement in the pair of vectors is 51. If we choose a threshold of 0.2, then the highest the computed distance can be for a match is 10.2, so s_2 and s_1 will not match using any of the Minkowski distances. When we compare s_0 , (50, 1, 20, 21, 49), with s_2 , we get distances of 8, 4.5, and 3. The maximum value in the two vectors is 50, so the highest the distances can be for a match is 10. This means that s_2 would match s_0 for each of these distance metrics.

There are several issues to consider for the Minkowski distances:

- As m increases in the Minkowski distance (See Eq. 2.), the influence of the larger differences increases, and the influence of the smaller differences decreases. In the extreme case of the Chebyshev distance, only the maximum difference has any bearing on the distance value.
- As the number of measurements being compared increases, the values of the Manhattan and Euclidean distances increase. Given vectors of constant differences greater than 1, the Manhattan distance increases quite rapidly linearly, and the Euclidean distance increases in the manner of \sqrt{x} . If the differences are all between 0 and 1, the computed distances increase more slowly.
- When time stamp values are being compared, e.g. start time and end time for events, the values are always increasing within a segment. This means that longer segments are judged less critically than shorter segments, because the maximum values that are compared with the distance measurement are larger.

Based on these trends, we expect that the Manhattan distance would give the most accurate results, because it gives larger weight to the smaller differences. The Euclidean distance would give slightly less accurate results, given the bias towards larger differences. The Chebyshev distance would be least accurate, because it only accounts for the largest difference measure.

4.3.1.4 Wavelet Transform

The discrete wavelet transform iteratively decomposes a signal of size L into two subsignals of size $L/2$. The first $L/2$ values give the trends in the original signal, and the second $L/2$ values give the fluctuations. Intuitively, it computes the averages and differences between pairs of numbers [31]. We give examples of transformations in Figure 12.

We use two wavelet transforms in our experiments: the average transform described in Figure 12 (*avgWave*), and the Haar transform (*haarWave*). The Haar transform is very similar to the average transform, with the only difference being that the averages and differences are multiplied by $\sqrt{2}$ [68]. For example, the trends computed in step 3 in Figure 12 would be $(9\sqrt{2}, 24.25\sqrt{2})$. For our implementation, we construct a vector for each of the segments to be compared. The first element of each vector is the relative start time of the segment, which is 0 in all cases. This is followed by the event entry and exit time stamps for all events in the segment. The last

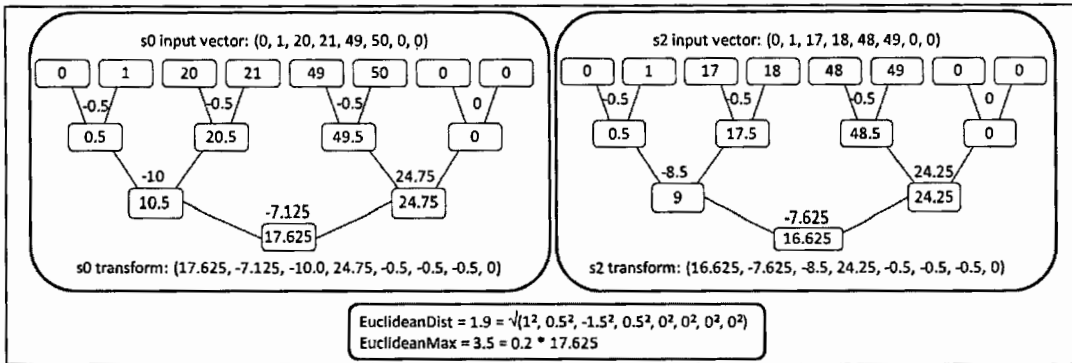


Figure 12 Wavelet Transform Example

Here we show two example average wavelet transforms. We iteratively compute averages (shown in boxes) and differences (shown between edges) for pairs of numbers, starting with the original vector. To compare the two transforms of s_0 and s_2 , we compute the Euclidean distance between them and compare it against a threshold (0.2) multiplied by the largest element in the vectors (17.625).

element is the exit time of the segment. Both transforms require an input vector with a length that is a power of two. We allocate space for the vector so that its length is the next power of two after the number of time stamps in the vector. We zero-pad the vector after the last time stamp element to the end. To compare transformed vectors, we compute the Euclidean distance between them [12] and compare it against a threshold multiplied by the largest value in the pair of transformed vectors. In Figure 12, we show an example comparison of the segments s_0 and s_2 from Figure 9. Because the computed Euclidean distance, 1.9, is less than the maximum allowed, 3.5, s_0 and s_2 match.

For both transforms, the values in the transformed vectors will be smaller than the values in the original vectors. The Haar transform has several properties that the average transform does not, including preservation of the Euclidean distance [12]. However, its values will be larger than those of the average transform since all values

are multiplied by $\sqrt{2}$. For the Haar transform, we expect more accurate results than from the Euclidean distance because the maximum value in the transformed vector will be smaller than the maximum value in the original vector, so the threshold test will be stricter. The values in the vector from the average transform will be smaller still; however, the Euclidean distance is not preserved, so the potential exists for a less strict test than the Euclidean distance.

4.3.2 Iteration-based Methods

We include two iteration-based methods: *iter_k* and *iter_avg*.

4.3.2.1 Keep K Iterations

For *iter_k*, we only keep a fixed number of each traced segment of code. We expect this method to produce small data files. For our example in Figure 9, if $k=3$, we would keep all three copies of the main.1 segment in the list of stored segments. However, if $k=2$, then we would keep *s0* and *s1* and discard *s2*.

4.3.2.2 Keep Average Iterations

The *iter_avg* method keeps the average measurements for each traced section of code. We expect this method to produce the smallest data sizes, since segments with the same context and same events will always match. To illustrate this method, we use the segments in Figure 9 and the stored segments scenario on the left. For this method, we never have more than one copy of the main.1 segment, and end up with a single copy of the main.1 segment that contains averages of the values of *s0*, *s1*, and *s2*.

We expect that these methods will produce fairly accurate data for applications that have little behavior variability, but poorly for applications that do have performance variabilities.

4.4 Traditional Trace and Trace Profile Size Models

In this section, we present models that predict the amount of data collected for trace profiling as well as traditional tracing. We illustrate the models with a small example and extrapolate the results to higher scales.

4.4.1 Traditional Trace

A traditional trace contains an event map and list of event and message records for each process. We model the size of a traditional trace with the following equation:

$$fullTraceSize = \sum_{i=1}^P (EventMap + EventData + MessageData) , \quad Eq. 3$$

where P is the number of processes in the run. The sizes of the *EventMap*, *EventData*, and *MessageData* are modeled by the following equations:

$$EventMap = E_n (I_s + N_s) \quad Eq. 4$$

$$EventData = E_c E_s \quad Eq. 5$$

$$MessageData = M_c M_s \quad Eq. 6$$

The meanings of the symbols in these equations are given in Table 2. The size of the event map is the product of the number of unique events (E_n) by the sum of the size of the event identifiers (I_s) and the size of the event names (N_s). For simplicity, we use a single number for the length of the event names, the median length. The amount of event data for each process is the product of the number of events in the file (E_c) and the size of each event record (E_s). The amount of message data is the number of

messages in the file (M_c) multiplied by the size of the message record (M_s). We see that the size of a traditional trace file will be determined by the number of processes and the event and message counts for the processes.

Table 2 Symbols for Full Trace and Trace Profile Models

Symbol	Meaning
N_s	The size of strings representing the names of events or segments
I_s	The size of the event or segment identifier
E_n	The number of unique events in the execution
P	The number of processes or process groups in the file
R_c	The number of ranks in a process group
R_s	Size of rank representation
S_n	The number of unique segments per process group
S_e	The count of segment executions per process group
T_s	The size of a timestamp
E_c	The number of events
E_s	Amount of data stored per event
M_c	The number of messages
M_s	Amount of data stored per message
H_s	Size of headers
H_{sd}	Size of segment data header

We use values for the sizes of event identifiers, event data, and message data based on those used by the TAU tool, configured for function entry and exit tracing and for gathering message passing data. TAU generates 24-byte records for each entry and exit event, so $E_s = 48$. For message passing events, 96 bytes of data are generated, $M_s = 96$. When we evaluate our model, we use slightly different event counts than an actual TAU trace would contain. We exclude several record types to ensure a more fair comparison between traditional tracing and trace profiling. The event records we exclude are:

- Records for any functions that are not included in trace profiling segments. For example, the function `main` is not included in any segment, so it is not included in the model for traditional tracing.
- Records for segment markers that we inserted into the code. Each segment marker generates entry and exit events.
- Any minor administrative records, e.g. TAU's `EVINIT` or `FLUSH_CLOSE` events.

4.4.2 Trace Profile

A trace profile contains a single event map, followed by data for one or more process groups. Each process group has a rank list, a segment map, a segment execution list, and data for segments. For parsing purposes, we added section headers to the file that indicate the type and number of records that follow the header. We show the format of a trace profile in Figure 13.

We use the following function to predict the size of a trace profile:

$$traceProfileSize = H_s + EventMap + \sum_{i=1}^P (RankList + SegmentMap + SegmentExecList + Segments) \quad Eq. 7$$

The definitions for the symbols used in the equations are in Table 2. The equation to compute the size of the event map in a trace profile is the same as for the traditional trace file (See Eq. 4). The equations for computing the sizes of the rank list, the segment map, segment execution list, and segments are given below:

$$RankList = H_s + R_s R_c \quad Eq. 8$$

$$SegmentMap = H_s + (N_s + I_s) S_n \quad Eq. 9$$

$$SegmentExecList = H_s + (T_s + I_s) S_e \quad Eq. 10$$

$$Segments = H_s + \sum_{j=1}^{S_c} (H_s + H_{sd} + EventData + MessageData) \quad \text{Eq. 11}$$

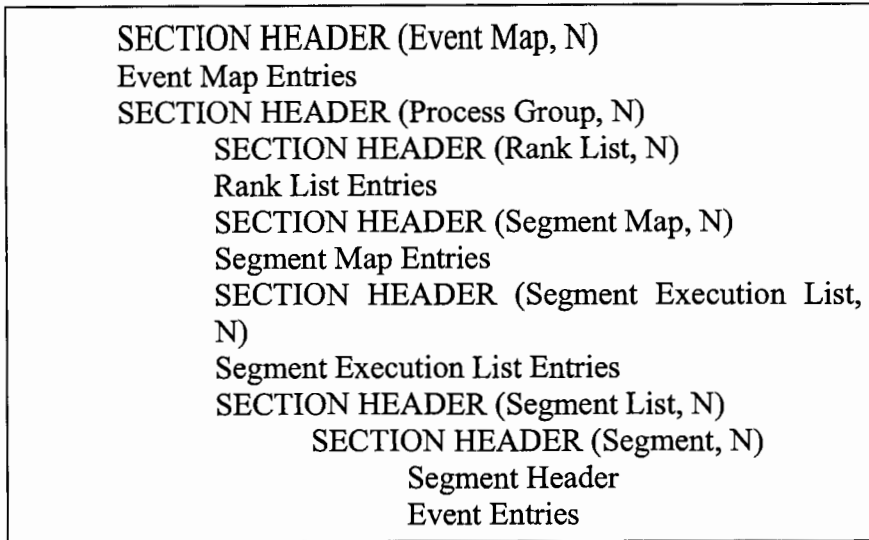


Figure 13 Trace Profile Format

This figure shows the format of a trace profile. Each section header tells the type of data that will follow it and how many entries of that type to expect (N). There is one event map per trace profile, followed by data for one or more process groups. Each process group has a rank list, a segment map, a segment execution list, and a list of the segments. The section header for each segment tells how many event entries to expect. The segment header gives the segment identifier and its duration.

The equations for computing the size of the event data and message data are the same as for the traditional trace and are given in Eq. 5 and Eq. 6. The size of a trace profile will largely depend on the degree of merging for process groups and segments, and on the amount of event and message data collected for each segment. If no process groups or segments merge, the size of the trace profile will scale with the number of processes in the run and the number of events and messages for each process, like a traditional trace. If segments merge, but process groups do not, the trace profile size

Table 3 Sizes of Fields in Trace Profiling Data Structures

Data type	Field	Type	Size (bits)
Section Header	id	char	8
	count	int	32
Event Map	id	int	32
	size	short int	16
	name	char array	N_s
Rank List	id	int	32
Segment Map	id	int	32
	size	char	8
	name	char array	N_s
Segment Execution List	id	int	32
	time stamp	double	64
Segment Data Header	id	int	32
	duration	double	64
Event Data	enter and exit markers	char	8
	id	int	32
	relative start	double	64
	duration	double	64
	message data	char	8
Message Data	type (send/recv) and src/dest rank	int	32
	bytes	int	32
	tag	short int	16
	comm	short int	16

will scale with the number of processes. If process groups merge, but segments do not, then the size of the trace profile will depend on the amount of event and message data collected in the segments. If there is segment and process group merging, then the size of the trace profile will depend on the number of performance behaviors in the

execution. The sizes of the fields in a trace profile are shown in Table 3. We use values from this table for evaluating our model.

4.5 Traditional Trace and Trace Profile Size Comparison Using Models

Our example program for size comparison is random-barrier, a simple MPI benchmark from the PPerfMark suite [43]. The random-barrier program has a single main loop. In each iteration of the main loop, a rank is chosen at random to be the bottleneck and cause the other ranks to block in `MPI_Barrier`. We manually

$$\begin{aligned} E_c &= [550, 250, 250, 250] \\ E_n &= [21, 21, 21, 21] \\ N_s &= 39 \\ P &= 4 \\ M_c &= [200, 50, 50, 50] \\ fullTraceSize &= 99.4 \text{ KB} \end{aligned}$$

Figure 14 Input Data to Traditional Trace Size Model

partitioned the program into three segments: init, mainloop, and finalize, using TAU's phase begin and end events. We ran the benchmark with four MPI processes for 50 iterations, resulting in 1 execution per process of init, 50 executions per process of mainloop, and 1 execution per process of finalize. Init has 44 function calls and 0 messages; mainloop has 10 function calls and 4 messages in rank 0, and 4 function calls and 1 message in ranks 1-3; finalize has 6 function calls and 0 messages. We generated a full trace of the execution using TAU.

4.5.1 Traditional Trace

We computed the size of a traditional trace for this four-process run of random-barrier using our model. The inputs to the model are showing in Figure 14, resulting in a predicted size of 99.4 KB, shown in Table 4. The actual size of the full amount of

$N_s = 39$ $E_n = 23$ $P = 4$ $R_c = [1 \ 1 \ 1 \ 1]$ $S_n = [5 \ 5 \ 5 \ 5]$ $S_e = [52 \ 52 \ 52 \ 52]$ <i>traceProfileSize</i> = 10.9 KB	$E_c = \begin{bmatrix} 44 & 10 & 10 & 10 & 6 \\ 44 & 4 & 4 & 4 & 6 \\ 44 & 4 & 4 & 4 & 6 \\ 44 & 4 & 4 & 4 & 6 \end{bmatrix}$ $M_c = \begin{bmatrix} 0 & 4 & 4 & 4 & 0 \\ 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 \end{bmatrix}$
---	---

Figure 15 Inputs to Trace Profile Size Model

data generated was 109.7 KB. The differences arise from two sources: we are estimating the size of the event files by using the median size of the strings describing the events; and we are excluding administrative events, e.g. FLUSH_CLOSE, and segment marker events from both the event file size and the trace file size. The difference in the sum of the event file sizes is 2493 bytes, and the difference in the trace file sizes is 13053 bytes, with 9984 bytes accounted for by segment boundary markers. The size of the full trace is broken down by rank and portion of code in Table 4.

Table 4 Sizes of Trace Profile and Full Trace

PROGRAM SECTION	DATA TYPE	TRACE PROFILE SIZE (BYTES)	FULL TRACE SIZE (BYTES)
<i>Whole Execution</i>	Section Headers (Event Map and Process Group)	10	0
	Event Map	1012	5769
<i>Rank 0</i>	Rank List	9	0
	Segment Map	225	0
	Segment Execution List	629	0
	Section Header (Segment List)	5	0
<i>init</i>	Segment Header	17	0
	Event Data	1012	2112
	Message Data	0	0
<i>mainloop</i>	Segment Header	51	0
	Event Data	690	24000
	Message Data	144	19200
<i>finalize</i>	Segment Header	17	0
	Event Data	138	288
	Message Data	0	0
<i>Ranks 1-3 combined</i>	Rank List	27	0
	Segment Map	675	0
	Segment Execution List	1887	0
	Section Header (Segment List)	15	0
<i>init</i>	Segment Header	51	0
	Event Data	3036	6336
	Message Data	0	0
<i>mainloop</i>	Segment Header	153	0
	Event Data	828	28800
	Message Data	108	14400
<i>finalize</i>	Segment Header	51	0
	Event Data	414	864
	Message Data	0	0
<i>Trace Profile Total (KB)</i>		10.9	99.4

4.5.2 Trace Profile

Using a post-mortem prototype trace profiler, we generated a trace profile from the full TAU trace of the execution. The difference operator we used was the Euclidean distance with an event difference threshold of 0.25. The end result was a 10.9 KB file containing data for four process groups, meaning that the behavior of each of the processes was different enough that they were not combined. Each process group contained the same segment count: 1 init, 3 mainloop, and 1 finalize, indicating that of the 50 iterations of mainloop, 3 were found to be representative of the behavior of the process for that segment. In Figure 15 we show the inputs that we fed into our trace profile size model. The computed size was 11204 bytes, or 10.9 KB, shown in Table 4. The real size of the trace profile was 10318 bytes, or 10.1 KB. The differences in the actual and computed sizes of the trace profile are due to the estimation of the sizes of strings by the parameter N_s . The differences between the actual and computed event and segment map were 250 and 636 bytes, respectively.

4.5.3 Comparison of Traditional Trace and Trace Profile

Now we use our model to extrapolate the sizes of the trace profile and traditional trace to 64K processes for the random-barrier example, assuming the conditions of the four-process run. We show how the sizes grow with increasing processes in Figure 16. We extrapolate three scenarios for the trace profile. In the first scenario, we assume that no segments or process groups merge (“No Merge”). In the second, there are always two process groups, one containing rank 0, and the other containing the rest of the ranks; and each process group has three segments, one init, one mainloop, and one

finalize (“Total Merge”). For the last, we assume the conditions that using the Euclidean distance gave us for the four process run: each rank is in its own process group and each process group contains five segments: 1 init, 3 mainloop, and 1 finalize (“Euclidean Distance”). The size of the traditional trace file reaches 1 GB at 64K processes, while the size of the “No Merge” trace profile is approximately 620 MB. This means that even if no processes or event patterns were found to be similar in the execution, the size of the resulting file will still be smaller than a full traditional trace. The size difference is largely due the fact that a trace profile does not write

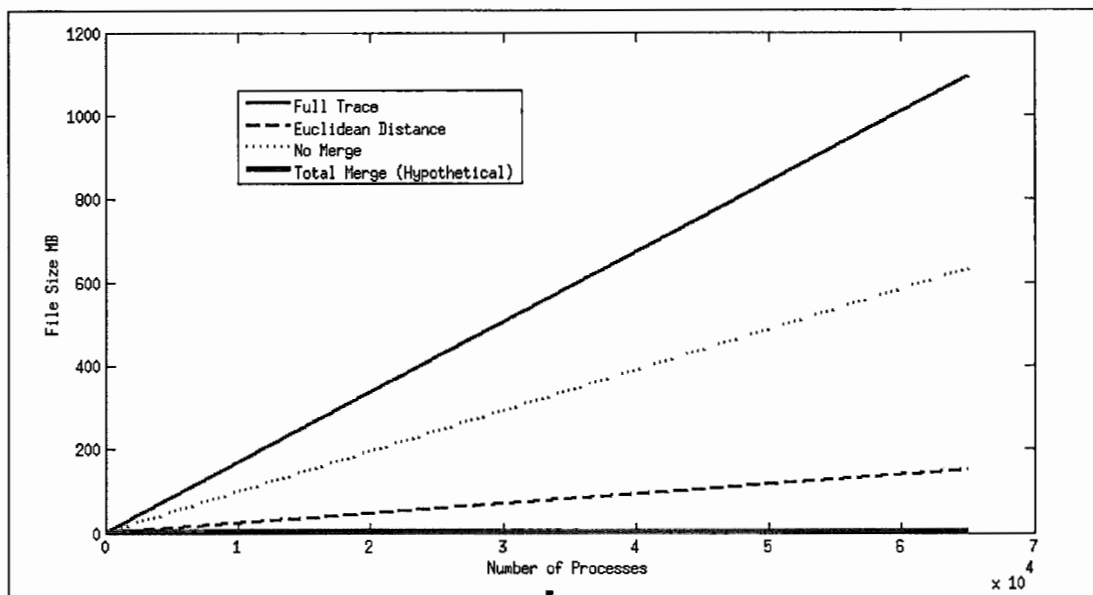


Figure 16 Traditional Trace and Trace Profiling Sizes for Random-Barrier

Here we show the predicted sizes of full traces and trace profiles of the random-barrier program for executions with up to 64,000 processes. The solid blue line (“Full Trace”) shows the predicted size of the full trace in MB. The dotted red line (“No Merge”) shows the predicted size of a trace profile if no processes and no segments were found to have behaved similarly. The dashed green line (“Euclidean Distance”) shows the size of the trace profile if no process groups merged, and each process group had five segments (1 init, 3 mainloop, and 1 finalize). The solid aqua line (“Total Merge”) shows the size of the trace profile if there were always two process groups and total merging of segments.

complete separate event records for function entries and exits, but instead uses the entry record and maintain a start time, duration, and simply marks the event exit with a record that only uses a single byte of storage. The “Total Merge” and “Euclidean Distance” trace profiles reach 0.25 MB and 147 MB, respectively, at 64K processes.

4.6 Trace Profiling and Visualization

The trace profiling technique can also address the scalability challenges in trace visualization analysis. First, the total amount of data has been reduced significantly, easing the memory and computation requirements of analysis tools. Second, because

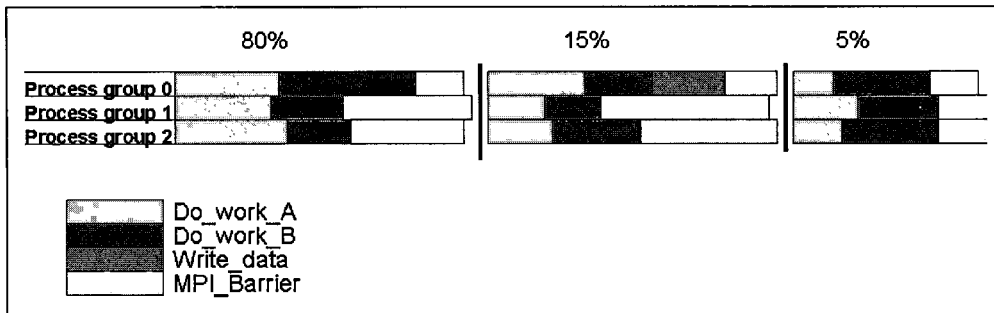


Figure 17 Trace Profiler Visualization

An example trace profile visualization showing the percentage of time processes spent in temporally aligned behavior patterns.

the behavior patterns in the execution have already been extracted, the tool can easily present partially analyzed information to the user, reducing the time taken for identifying performance problems. A visualization of a trace profile could show the percentage of time processes in the execution spent in temporally aligned segments. Such a presentation of this information could significantly reduce the time needed for diagnosing performance problems when compared to visually inspecting a full program trace for potentially very many processes. An example mock-up trace profile

visualization is shown in Figure 17. This figure shows a representation of a multi-process execution. There were three groups of processes that behaved similarly: process group 0, process group 1, and process group 2. In the first segment (on the left), we see that 80% of the time, process group 0 spent more time in `Do_work_B` and was late to `MPI_Barrier`, causing the other ranks to block. In the middle segment, we see that 15% of the time, process group 0 was again late to `MPI_Barrier`, but was late because it executed `Write_data`, and spent somewhat more time in `Do_Work_A` than the other processes. In the segment on the right, we see that 5% of the time, the processes behaved roughly the same, and reached `MPI_Barrier` at approximately the same time. Immediately, the user would be able to see that 95% of the time, process group 0 is causing the other processes to block in `MPI_Barrier`, and will see that 80% of the time it is due to a load imbalance in the `Do_work_B` function.

4.7 Summary

Trace profiling is a novel performance measurement technique for gathering event-based performance data. In this chapter, we first described the trace profiling technique and segment difference methods, followed by models that predicted the size of trace profiles and traditional traces. We used the models and a simple benchmark to illustrate the possible data reduction achievable from using trace profiling over traditional traces. Our example showed that even if no merging occurs, the trace profile is still smaller than a full traditional trace, and that with the degree of merging we obtained when using our prototype implementation, our model predicted a size savings of a factor of 10 between trace profiling and a traditional full trace for 64K

processes. Finally, we described how the reduced amount of data produced in a trace profile could ease the memory and computation requirements for analysis and visualization tools. We showed an example visualization of a trace profile, illustrating how it could potentially facilitate a user's understanding of the performance of programs more easily than by visualizing an entire program trace.

In the next chapter, we present a study of methods for comparing traces and demonstrate that trace profiling can produce reduced traces that still retain the necessary information for correct performance. Following this, we present our runtime implementation of a trace profiler and evaluate its overheads compared to traditional traces.

5 Trace Comparison Methods

In this chapter, we demonstrate that trace profiling can produce reduced traces that retain the information needed for correct performance diagnosis of programs. To do so, we perform a comparative study of similarity methods in current or proposed use for trace reduction. Using a post-mortem implementation of a trace profiler, we apply the similarity methods to the task of deciding segment matching and evaluate the methods for file size reduction, trace error, and retention of performance trends. Our goal is to determine a similarity method that yields adequate trace reduction and also retains the information needed for correct performance analysis. Achieving our goal required that we answer several key questions:

What metrics can we use to evaluate and compare trace similarity methods? In addition to file size reduction, we developed and used metrics for error, greatest possible file size reduction (i.e. potential for repeated patterns), and consistency of performance diagnosis.

How much error should be allowed? Values that will likely never be exactly equal need to be compared. We had to decide how much each measurement can vary, and weigh the consequences of the amount of error. If we are matching traces for the purpose of trace compression, then a larger allowed error between traces would mean larger number of matches, and thus a smaller trace file. However, the larger error might prevent the correct performance diagnosis from being made.

How can we measure the “goodness” of each approach? Most trace compression studies report the reduction of file size achieved; but no matter how much compression

is achieved, if the reduced trace no longer contains the data needed for accurate performance diagnosis, the method is not useful for our purpose. We evaluate each approach not just on amount of compression, but also on amount of error and consistency of diagnosis, and discuss the tradeoffs in weighting the different metrics.

5.1 Evaluation Methodology

In this section we detail our framework for the evaluation of similarity methods. We investigate traces collected for a set of benchmarks with known behaviors, and for a full application, running on a Linux cluster. We apply our post-mortem trace profiler to full execution traces, varying the similarity method used to determine repeating patterns within the trace. We evaluate the methods for intra-process segment matching only, inter-process segment matching only, and combined intra- and inter-process segment matching. Our evaluation focuses on three metrics: file size reduction, amount of error in the trace, and retention of performance trends. For file size reduction we simply compare the sizes of the reduced traces to the full-sized traces from which they were derived. We calculate the trace error by recreating an approximated full-sized trace from the reduced version, then comparing it to the actual full trace. We evaluate retention of performance trends by feeding the actual and approximated full traces into a performance analysis tool and examining any differences in the results.

5.1.1 Benchmarks

We crafted our benchmarks to represent classes of performance behaviors that occur in parallel programs on high end systems. These performance behaviors can

appear with a high degree of regularity, sporadically, or progressively change over the iterations in the execution. To reflect this, we created a set of regularly behaving benchmarks, a set of irregularly behaving benchmarks, and a benchmark that simulates dynamic load balancing. Because we know the behavior patterns in each benchmark, we can evaluate how well each of the methods retains the performance behaviors.

We used the APART Test Suite (ATS) to create our benchmarks. The ATS a collection of utilities designed to create programs with known behavior for testing parallel performance tools [22]. We chose behavior patterns from the ATS that represent performance problems that require trace data for correct diagnosis. For parallel programs, these performance behaviors fall into four categories based on the communication pattern being used. We describe these communication patterns here using MPI functions as examples.

$N \rightarrow 1$. *N processes send data to 1 process.* If any of the sending processes are late, then the receiving process blocks, waiting for them to execute the send operation. Example MPI functions for this pattern are `MPI_Reduce` and `MPI_Gather`, with corresponding performance behavior problems *early_reduce* and *early_gather*.

$1 \rightarrow N$. *1 process sends data to N processes.* If the sending process is late, then all N receiving processes will block until the send is executed. Example functions are `MPI_Bcast` and `MPI_Scatter`. The corresponding performance problems are *late_broadcast* and *late_scatter*.

$1 \rightarrow 1$. *1 process sends to 1 process.* There are two cases. In the case of a non-blocking send and a blocking receive, if the sending process is late, the receiving

process will block. In the case of a synchronous send, the sending process will block if the receiving process is late. Example communication routines are `MPI_Ssend` and `MPI_Recv`, with corresponding performance problems *late_receiver* and *late_sender*.

$N \rightarrow N$. *N processes send to N processes*. Here, all N processes depend on all other processes involved in the communication to proceed. If any of the N are late, then the rest of the processes block until all have reached the communication routine. An example is `MPI_Barrier` with corresponding performance problem *imbalance_at_barrier*.

Benchmarks with Regular Behavior. We chose five example benchmarks provided with ATS with regular behavior: *early_gather*, *imbalance_at_mpi_barrier*, *late_receiver*, *late_sender*, and *late_broadcast*. Each of the benchmarks simulates a program with the given behavior problem with the same severity in each iteration. In other words, all iterations of each program will exhibit the performance problem and all iterations should be very similar. All runs had 8 processes.

We expect the similarity methods to do relatively well on this set of benchmarks since the iterations have regular behavior. They should be able to find a large number of segments matches and still retain the correct performance behaviors.

Benchmarks with Irregular Behavior. For this category, we used ATS to create new benchmarks with irregular behavior. The benchmarks simulate the system interference identified by Petrini et al. when they ran an application on ASCI Q [51]. The system interference prevented the application from scaling as predicted. The benchmarks contain iterations with work periods that last approximately 1 *ms*

followed by a communication step, using the communication patterns described previously. The load for each process is constant in each iteration and across processes: the only performance problem comes from the interference. We simulated the system noise using timers to interrupt the processes as described by Petrini et al. We used two simulation scenarios. The first was a 32-process run, with each of the 32 processes simulating the interrupts specific to the 32 nodes in an ASCI Q cluster. The second was also a 32-process run, but with the simulated amount of system interruptions that would occur if there were 1024 processes in the run. When we refer to the benchmarks in the first category, we use the communication pattern and either a `_32` or a `_1024`, to indicate whether 32 or 1024 processes were simulated, respectively.

For these benchmarks, we expect the methods to find a high number of matches, since most iterations are very similar. However, it will be important that they don't falsely match undisturbed and disturbed iterations, as this has the potential to mask or amplify the periodic behavior changes due to the simulated interruptions.

Dynamic Load Balancing. Here, we used ATS to create a program that simulates an application that does dynamic load balancing. For this benchmark, the performance of the iterations starts at about 1 *ms* and gets progressively worse, with one-half of the processes doing more work each iteration and the other half doing less work in each iteration, until the "load balancer" is triggered. The "load balancer" readjusts the amount of work on each processor to be equal. The performance problem exhibited by this program is *imbalance at mpi all to all*, which falls in the N-to-N communication

category. This benchmark is referred to as *dyn_load_balance* and was run with 8 processes.

For this benchmark, we expect less overall matching since behavior changes with each iteration and very close performance behaviors reoccur only after each simulated load balance. Here it will be important that the similarity methods do not match segments with larger differences because the load imbalance may no longer be apparent in the reduced trace.

5.1.2 Application

We chose Sweep3D 2.2b, a structured mesh application that computes a 1-group time-independent discrete ordinates three-dimensional Cartesian geometry neutron transport problem [3]. Structured mesh applications have a regular partitioning of the data, where all interior data blocks have equal numbers of neighbors. It is likely that the performance will be very regular over the course of the program, which means that the reduction methods should be able to find a large number of segment matches without introducing a large amount of error. We collected traces for two runs of this application: an 8-process run with input file *input.50*, *sweep3d_8p*; and a 32-process run with input *input.150*, *sweep3d_32p*.

5.1.3 Instrumentation

We used the dynamic instrumentation library Dyninst [29] to instrument the full application for both function entry and exit tracing as well as inserting segment begin and end markers. The simple benchmarks were marked manually. See Section 5.2.1 for details on program segmentation.

5.1.4 Evaluation Criteria

We chose four criteria to evaluate the metrics: percentage of full trace file size, degree of matching, approximation distance, and retention of correct performance trends.

5.1.4.1 Percentage of Full Trace File Size

We present the savings in file size as a percentage of the full, non-reduced trace file, as a relative measure of size reduction. We expect *iter_avg* to perform the best in this category since it matches all segments with the same context, regardless of the measurement values in the segments.

5.1.4.2 Degree of Matching

The degree of matching metric is a measure of how many segment matches occurred. We define it to be the ratio of the number of matches to the number of possible matches. The number of possible matches is limited by the structure of the program. For example, some portions of the code may only execute one time, e.g. an initialization step, and will not match any other event sequence in the trace.

5.1.4.3 Approximation Distance

We estimate the error in the trace by recreating a full trace from the reduced trace and comparing each time stamp with its counterpart in the original full trace. The approximation distance metric tells the 90th percentile of absolute differences between

paired measurements in the original and reduced traces.¹ For this metric, high values for *iter_k* and *iter_avg* mean that there is irregularity in the execution that is not being captured in the iterations that are retained. High values for *absDiff* give a rough indication of the absolute difference of time stamps from the true values in the full trace. High values for the Minkowski and wavelet methods mean that there are high maximum values in the set of values being compared, relative to the distance between those values.

5.1.4.4 Retains Correct Performance Trends

Arguably, the most important criterion for evaluating a trace matching metric for the purposes of performance analysis is deciding whether or not the reduced trace still indicates the same performance problems as the full trace. For example, if an analyst inspecting a full trace detects a late sender performance problem, the same problem should be detected in the reduced trace with approximately the same severity. The KOJAK tool set was developed to aid parallel performance analysts in the challenging task of performance diagnosis [71]. KOJAK's EXPERT tool reads in a trace file and produces a data file containing performance diagnoses. Each diagnosis consists of a

¹ When recreating full traces for the *iter_k* method, we used the last segment that executed of each pattern to fill in the segment executions that were not collected. Alternatives include using the average measurements from the *k* collected segments, or using the centroid of those *k* segments as determined by a clustering algorithm.

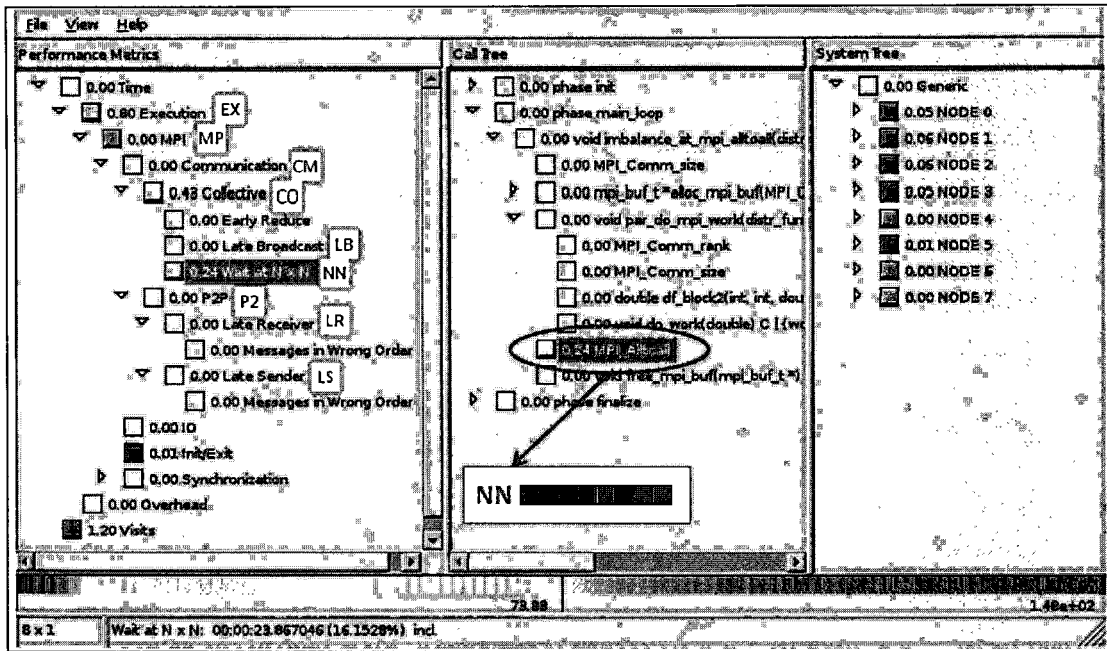


Figure 18 KOJAK and Derivation of Our Performance Diagnosis Representation.

Here we show a screenshot of KOJAK's EXPERT tool displaying the performance diagnosis for `dyn_load_balance`. The color bar on the bottom shows the severity levels, with blue being low and red high, and gray indicating 0 or close to 0. The left panel shows the performance metrics; the middle panel shows the code locations; and the right panel shows the processes. The color blocks next to each metric, code location, and process show the severity for the selected combination. Above, we have selected the function `MPI_Alltoall` and the "Wait at NxN" metric. This combination has green or "medium-low" severity and the severity is close to 0 for ranks 4, 6, and 7 and fairly low for ranks 0-3 and 5. We represent this diagnosis by abbreviating the metric name, e.g. NN for "Wait at N x N," coloring the metric abbreviation according to the severity indicated in the code location pane, and coloring squares for each process according to their severity levels. White squares indicate negative severities. We show the abbreviations we use for selected KOJAK metrics in white rectangles next to the metric names.

metric, a code location, and a severity for each thread in the run [59]. KOJAK's CUBE tool reads in the analysis data and presents a visualization to the user, indicating the most important performance trends in the trace in a hierarchical manner.

We use the CUBE visualization tool to compare the performance diagnoses for the recreated traces against the diagnoses for the full trace (See Figure 18). We determine whether a performance analyst would come to the same conclusions about the reduced trace as the full trace. If not, then the reduced trace is not adequate for performance analysis. We admit that this is a subjective test; however, we followed a set of guidelines when deciding if the diagnoses were sufficiently similar, so all the methods were subjected to the same criteria.

5.2 Intra-process Reduction Evaluation Studies

In this section, we present the results of two studies evaluating the similarity methods for intra-process segment matching using the criteria and programs described in Sections 6.1.1 and 6.1.2. We first present a threshold study for the similarity methods from the distance metric category. From this study, we choose a threshold for each of these methods that represents the best tradeoff in terms of file size reduction, measurement error, and retention of performance trends. In the second study, we present the results of a comparative study of the similarity methods, using the thresholds found to be best for each method in the threshold study.

5.2.1 Threshold Study

We investigated the behavior of the methods in reducing the traces of the benchmarks while varying the thresholds that determine whether two given segments should match or not match. The thresholds for *relDiff*, Minkowski distances, and the wavelet transforms were 0.1, 0.2, 0.4, 0.6, 0.8, and 1.0. The thresholds for *iter_k* were 1, 10, 50, 100, 500, and 1000, and for *absDiff* were powers of 10 from 10^1 to 10^6 .

Since no thresholds are used with the *iter_avg* method, it was not included in this study. The criteria we used to evaluate the methods were file size, approximation distance, and retention of performance trends (For full results, refer to the Appendix.). For each method, we chose a representative threshold to be used when comparing the methods against each other.

relDiff. The file size for each benchmark and the sweep3d runs decreased relatively steadily with increasing threshold. The approximation distance remained small until the 0.8 threshold, after which there was a large jump for many of the benchmarks and sweep3d_32p. Performance trends were correctly retained for most programs up to a threshold of 0.8. Based on the jump in approximation distance and loss of performance trends after threshold 0.8, we chose 0.8 as the best threshold for *relDiff*.

absDiff. Here the file sizes for the benchmarks and sweep3d dropped off fairly quickly at a threshold of 100 and continued to decrease slightly with increasing threshold. The approximation distance stayed relatively low up to a threshold of 10^4 , after which there was a sharp increase for several of the benchmarks and sweep3d_32p. Performance trends were retained for most programs at a threshold of less than 10^3 . Because the file sizes were relatively low and performance trends were retained at 10^3 , we chose 10^3 as the representative threshold for *absDiff*.

Manhattan, Euclidean, and Chebyshev. When observing file sizes changes, the *Manhattan* and *Euclidean* methods behaved quite similarly; the *Chebyshev* method showed some differences. For the *Manhattan* and *Euclidean* methods with the regular

benchmarks, the 1-to-1 irregular benchmarks, and *sweep3d*, file sizes decreased relatively steadily with increasing threshold; with the other irregular benchmarks, the file size decreased only slightly with increasing threshold, because a matching that was close to optimal was reached early, at a threshold of 0.1. For *Chebyshev* with the 1-to-1 irregular benchmarks and *sweep3d*, file size decreased with increasing threshold; with the regular benchmarks and remaining irregular benchmarks, file size was relatively constant with increasing threshold. For all three methods, we observed the following behavior in approximation distance: with the regular benchmarks, approximation distance was relatively constant with increasing threshold; with the 1-to-1 irregular benchmarks, approximation distance increased with increasing threshold; with the remaining benchmarks, the approximation distance remained low until after the threshold of 0.8, after which there was a large jump. For *sweep3d* and *Manhattan* and *Euclidean*, approximation distance increased with increasing threshold; for *Chebyshev*, the approximation distance was small and relatively constant until after the 0.8 threshold. For retention of performance trends, the Manhattan distance did well up to a threshold of 0.4, and the Euclidean and Chebyshev distances did well up to 0.2. We based our selection of best thresholds for these methods on the retention of performance trends metric, because we consider this metric to be the most important. We chose 0.4 as the best threshold for the Manhattan distance and 0.2 for the Euclidean and Chebyshev distances.

Wavelet Transforms. For all evaluation criteria, *avgWave* and *haarWave* performed similarly. For all programs, file sizes decreased with increasing threshold,

up to the point of perfect matching, after which no further decrease in size is possible. The best threshold in this category appears to be 0.4 for both methods, because file size decrease levels off after this threshold. The approximation distance for both methods remained steady with increasing threshold for the regular benchmarks and the irregular N-to-1, N-to-N, and 1-to-N benchmarks. The approximation distance increased with increasing thresholds for the irregular 1-to-1 benchmarks and sweep3d. The threshold 0.2 is best for approximation distance, because of the relatively higher values for the `dyn_load_balance` benchmark and sweep3d after this threshold. For the majority of programs, performance trends were retained for both methods at thresholds below 0.2. For these reasons, we chose 0.2 as the best threshold for the wavelet transform methods.

iter_k. Generally speaking, there was an increase in file size and decrease in approximation distance with increasing k . Performance trends were retained for most programs up to threshold 10. The choice for the best value of k wasn't clear, but we chose $k=10$ as the best because the performance trends were retained for most programs at this threshold.

5.2.2 Comparative Study

In this section, we present comparative results for the different methods using size and degree of matching; approximation distance; and retention of performance trends as the evaluation criteria. Based on the results of the threshold study in Section 6.2.1,

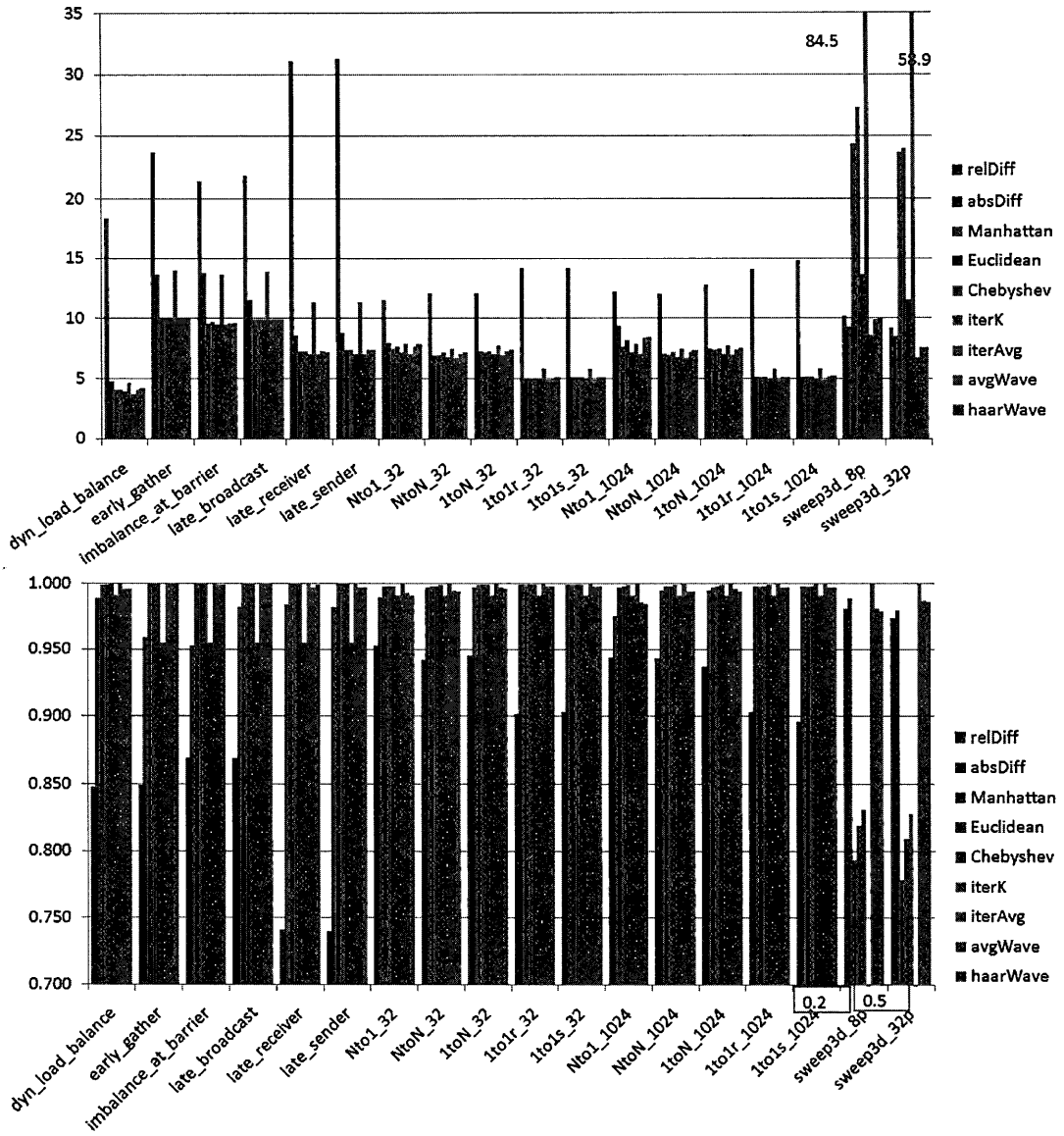


Figure 19 Intra-process Reduction: Percentage File Sizes and Degree of Matching.

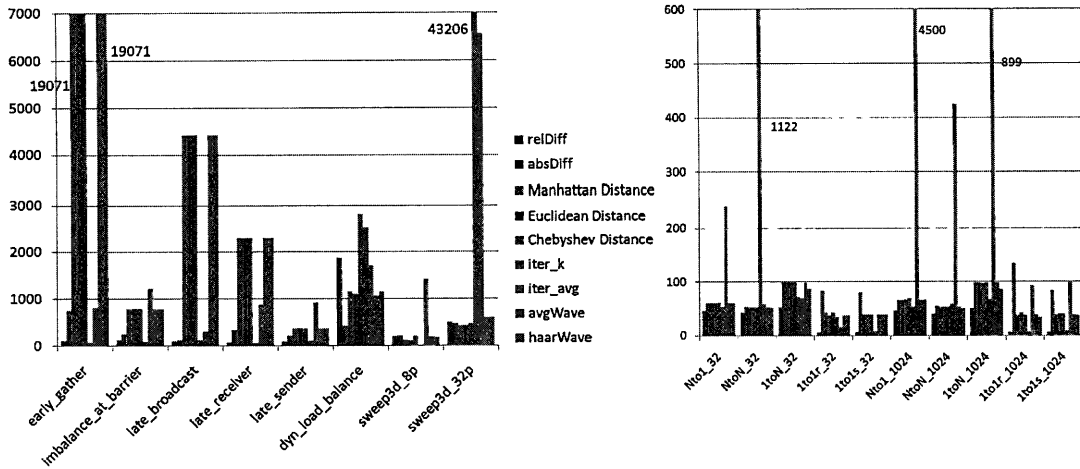


Figure 20 Intra-process Reduction: Approximation Distance Results for All Methods at Default Thresholds.

we present results for the best performing threshold for each method: 0.8 for *relDiff*, 1000 for *absDiff*, 0.4 for *Manhattan*, 0.2 for *Euclidean* and *Chebyshev*, 10 iterations for *iter_k*, and 0.2 for *avgWave* and *haarWave*.

5.2.2.1 Size and Degree of Matching

We present the data for reduction of traces for each method in Figure 19. The *iter_avg* method gives the best case values for this category, since exactly one segment is retained for each unique segment context.

The benchmark data shows that for the most part, the degree of matching for each of the methods is greater than 0.9, meaning that greater than 90% of the segments were matched. Exceptions occur with *relDiff*, which had degree of matching scores as low as 0.74. *RelDiff* had the highest file sizes and lowest degree of matching scores. The next largest file sizes are generated with the *iter_k* method; however, they are not much higher than those for the other methods. The Minkowski distances, *avgWave*, and *haarWave* all have nearly identical results, with *Chebyshev* having a very slight

advantage over the others. *AbsDiff* had only slightly larger file sizes than the Minkowski distances.

For *sweep3d*, the results are somewhat different. Because this application has very regular behavior, we expected the results to be similar to those of the benchmarks. However, because of the program structure, there are more segments, as well as differences within the segments, e.g. message passing parameters, that cause segments not to match. We see that *iter_k* performed the worst, with the highest file sizes and lowest degree of matching scores. This is because *iter_k* needed to keep 10 copies of each individual segment, regardless of how similar in performance they actually were, whereas the high degree of matching often results in fewer than 10 copies. The next worst performing were the Minkowski distances, again with *Chebyshev* having the smallest file sizes. The wavelet methods performed best, followed by *absDiff* and *relDiff*, each with very close to perfect matching and lowest possible file sizes.

The obvious best method in this category is *iter_avg*, since all segments match by definition. A comparison of the average file sizes for each of the other methods yields the following ranking: *avgWave*, *haarWave*, *Chebyshev*, *absDiff*, *Manhattan*, *Euclidean*, *iter_k*, *relDiff*.

5.2.2.2 Approximation Distance

Figure 20 shows the approximation distance results for each of the methods. The methods show similar trends across the benchmarks with regular behavior. The *relDiff*, *absDiff*, *iter_k*, and *iter_avg* methods have consistently low values. The Minkowski distances, *avgWave*, and *haarWave* transform behave similarly, and have

the highest values overall. The results for the `dyn_load_balance` benchmark show a different set of behavior, with *absDiff* having the lowest value, followed by *avgWave*, *Euclidean*, *Manhattan*, and *haarWave*. The irregular benchmarks had lower overall approximation distance values than the other benchmarks, with similar results across the benchmarks. The worst performing methods in this case were *iter_avg* and *iter_k*. However, the approximation distance values are low in comparison to those for the other set of benchmarks.

The results for `sweep3d` show *iter_avg* performing the worst for the 8-process run, and *iter_k* and *iter_avg* the worst for the 32-process run, indicating that there are performance behaviors not being captured by those two methods.

The methods that performed the best in this category are *relDiff*, followed by *absDiff*, and then *iter_avg*. The rest of the methods allowed significant error into at least one of the reduced traces.

5.2.2.3 Retention of Performance Trends

- We present summaries of the performance diagnoses given by KOJAK for selected benchmarks in Figure 21 and Figure 22. We show how we derive the performance diagnoses charts and abbreviations for metric names in Figure 18.

- For the benchmarks with regular behavior, nearly all the methods performed quite well. For `late_receiver`, all methods except *iter_avg* performed equally well, with all performance trends retained. The results for *iter_avg* with `late_receiver` showed differences significant enough that they may lead to an inaccurate performance

	MPI_Alltoall				do_work	
no loss					NN	EX
relDiff					NN	EX
absDiff					NN	EX
Manhattan					NN	EX
Euclidean					NN	EX
Chebyshev					NN	EX
iter_k					NN	EX
iter_avg					NN	EX
avgWave					NN	EX
haarWave	EX	MP	CM	CO	NN	EX

Figure 21 Intra-process Reduction: KOJAK Performance Trends for `dyn_load_balance` For Each Method at Default Thresholds.

Here we show the results for each reduction method in the `MPI_Alltoall` and `do_work` functions. The first row shows the diagnoses for the full trace. Each box in a row shows a performance diagnosis for a single combination of metric and code location.

assessment. For `early_gather`, all but the Minkowski distances, `avgWave`, and `haarWave` retained the correct performance trends. The results for `imbalance_at_barrier` showed that the Minkowski distances, `absDiff`, `iter_avg`, `avgWave`, and `haarWave` retained the performance trends, while `relDiff` and `iter_k` both showed a negative value for the major performance diagnosis. The amount of error introduced into the reduced traces caused time stamps to be skewed enough that the performance diagnoses resulted in negative values. We show the major performance trends for `dyn_load_balance` in `MPI_Alltoall` and `do_work` as reported by the KOJAK tools for the full trace and all methods in Figure 21. The results for the no loss trace clearly indicate that the lower ranks are spending more time in `MPI_Alltoall`, because the upper ranks are spending more time in `do_work`. None of the methods gave perfect results for the `dyn_load_balance` benchmark; however, `absDiff`, `Manhattan`, `Euclidean`, `avgWave`, and `haarWave` gave

	MPI Ssend					MPI Recv					do work
no loss	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
relDiff	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
absDiff	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
Manhattan	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
Euclidean	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
Chebyshev	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
iter_k	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
iter_avg	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
avgWave	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
haarWave	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX

Figure 22 Intra-process Reduction: KOJAK Performance Trends for 1to1r_1024 for Each Method at Default Thresholds.

the closest performance diagnoses because for the most part they maintained the performance differences due to load imbalance between the upper and lower ranks. Although *Manhattan*, *Euclidean*, *avgWave*, and *haarWave* lost the disparity in `do_work`, the diagnosis “Wait at NxN” is non-negative and maintains the disparity in behavior. *AbsDiff* maintained the disparity in performance in `do_work`, but reported that “Wait at NxN” was negative. All other methods lose the expected disparity in `do_work`.

For the irregular benchmarks, all methods did pretty well on the N-to-1 and 1-to-N benchmarks, with the exception of *iter_avg*, which failed on three benchmarks, and *Chebyshev*, which failed on `Nto1_1024`. *AbsDiff* did less well on the 1-to-1 and N-to-N benchmarks. We show the data for `1to1r_1024` in Figure 22. *AbsDiff* picked up on the variations in the iterations due interference, which caused some performance diagnoses to be skewed in a positive or negative direction. The best performers for these benchmarks were *Manhattan*, *Euclidean*, and *avgWave*, followed by *relDiff*, and *haarWave*. *AbsDiff* and *iter_avg* both only showed correct diagnoses for one benchmark, `1to1r_32` and `1to1s_32`, respectively.

For `sweep3d_8p` and `sweep3d_32p`, all methods but `iter_avg` and `iter_k` produced correct data. `Iter_k` showed a non-existent disparity in rank performance in `pmpi_recv` in `sweep3d_8p` and a greatly inflated severity in `pmpi_recv` in `sweep3d_32p`. `Iter_avg` showed a much lower severity in `sweep_` than did the no-loss trace for both `sweep3d_8p` and `sweep3d_32p`.

The best methods in this category were *Manhattan*, *Euclidean*, and *avgWave* which correctly diagnosed 17 out of the 18 execution traces. *HarrWave* did second best, correctly diagnosing 16.¹ The rest of the methods in order were: *relDiff* (14); *absDiff* and *Chebyshev* (13); *iter_k* (12); and *iter_avg* (6). The relatively poor performance of *iter_k* in this category could be due to our choices in implementing this method¹. It is possible that the first iterations are more subject to variabilities in execution, before the processes synchronize into their regular behavior patterns, and that the last segment is not the best choice as a fill in for missing segments.

5.2.2.4 Discussion

To determine best method for comparing traces, we take the highest ranking methods from each category and weigh the importance of each of the categories. The best methods from the size category were *iter_avg*, followed by *avgWave*, *haarWave*, and *Chebyshev*. Those from the approximation distance category were *relDiff* and *absDiff*, followed by *iter_avg*. Finally, the methods that best retained performance trends were *avgWave*, *Manhattan*, *Euclidean*, and *haarWave*. One could argue that the absolute most important criteria for judging these methods is whether or not they retain the correct performance trends, because that is the point of collecting the traces

in the first place. However, almost equally important is the ability to collect, store, and analyze the trace data at all. Given that *avgWave* performed well in both the size and retention of performance trends categories, we choose *avgWave* as the best method of the ones studied for intra-process segment comparison.

5.3 Inter-process Reduction Evaluation Studies

We evaluated the similarity methods for their ability to find inter-process matches. We first present a threshold study for the similarity methods. From this study, we choose a threshold for each of these methods that represents the best tradeoff in terms of file size reduction, measurement error, and retention of performance trends. Next, we present the results of a comparative study of the similarity methods, using the thresholds found to be best for each method in the threshold study. We did not evaluate *iter_avg* or *iter_k* in this section, because utilizing them for the purpose of inter-process matching is nonsensical.

5.3.1 Threshold Study

We investigated the behavior of the similarity methods while varying the thresholds that determine whether two given segments should match or not match. The thresholds for *relDiff*, Minkowski distances, and the wavelet transforms were 0.1, 0.2, 0.4, 0.6, 0.8, and 1.0. The thresholds for *absDiff* were powers of 10 from 10^1 to 10^6 . The criteria we used to evaluate the methods were file size, approximation distance, and retention of performance trends (For full results, refer to the Appendix.). For each method, we chose a representative threshold to be used when comparing the methods against each other.

relDiff. The relative difference method performed poorly for inter-process matching. For all benchmarks and sweep3d, *relDiff* only found matches when the threshold was 1.0, which means any amount of error was allowed when comparing the segments. None of the reductions produced by *relDiff* retained correct performance trends. For the purpose of our comparison study of inter-process matching, we chose 0.8 as the best threshold for *relDiff*. Because no matches were achieved, correct performance trends were retained.

absDiff. For the benchmarks, file sizes tended to start to decline and approximation distances began to increase at a threshold of 10^4 . For the most part, performance trends were retained for the benchmarks at and below 10^4 . *AbsDiff* was unable to find any matches for sweep3d_8p; *absDiff* did find matches for sweep3d_32p at and above 10^5 , but correct performance trends were not retained. Correct performance trends were retained for the majority of the codes at thresholds at or less than 10^4 . Based on these results, we chose 10^4 as the best threshold for *absDiff*.

Minkowski distances. The three methods performed similarly for the benchmarks. File sizes decreased relatively steadily with increasing threshold and approximation distances increased most sharply above thresholds of 0.4. Performance trends were retained for the majority of the benchmarks for thresholds at or above 0.4. All three methods performed the same for sweep3d, finding no matches for either sweep3d_8p or sweep3d_32p at any threshold. We chose 0.4 as the best threshold for all three methods.

Wavelet transforms. For the benchmarks, both methods performed similarly. File sizes decreased steadily with increasing threshold. The approximation distances for the most part remained low until reaching the 0.6 threshold. For the majority of the benchmarks, correct performance trends were retained for thresholds of 0.4 and higher. Neither method found matches for *sweep3d_8p*. Both found matches for *sweep3d_32* at threshold 0.8, however performance trends were not retained. Based on the tradeoffs of size reduction and retention of trends, we chose 0.4 as the best threshold for both *avgWave* and *haarWave*.

5.3.2 Comparative Study

Here, we present a comparative study of inter-process reductions achieved by the similarity methods at the thresholds chosen in Section 6.3.1. We evaluate the methods for file size reduction, amount of matching, and retention of correct performance trends. Although *relDiff* was unable to find any acceptable inter-process reductions at thresholds below 1.0, we include its results at the 0.8 threshold as a measure of the worst-case scenario for file size reduction and merging. For *absDiff*, we used the threshold 10^4 ; for *Manhattan*, *Euclidean*, *Chebyshev*, *avgWave*, and *haarWave*, we used 0.4.

5.3.2.1 Size and Degree of Matching

We show the percentage file size and degree of matching for all methods and benchmarks in Figure 23 and Figure 24. *RelDiff* performed the most poorly since it was unable to find any matches for any of the benchmarks or *sweep3d*. *AbsDiff* performed better for the irregular benchmarks, with an average 59.7% percent file

size, than it did for the regular benchmarks, with an average of 87.5%. The Manhattan and Euclidean distances and wavelet transforms performed similarly, with average percent file sizes close to 60% and average degrees of matching at 0.4. *Chebyshev* achieved the greatest amount of reductions, with average percent file size at 44.8% and average degree of matching at 0.6. Outliers in the set of benchmarks were 1to1r_1024 and 1to1s_1024. None of the methods but *Chebyshev* were able to find

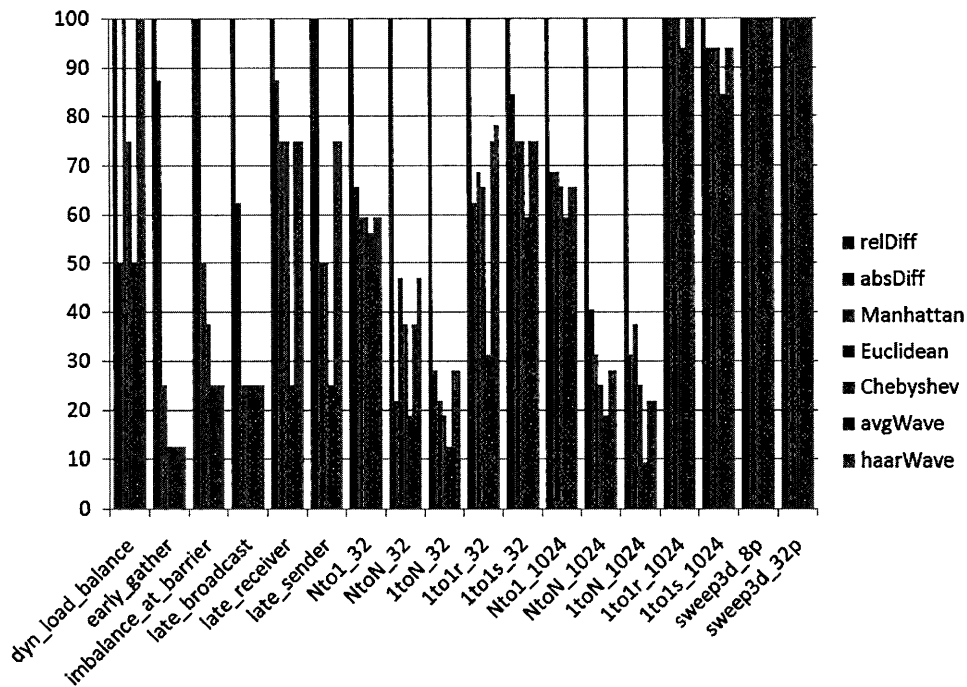


Figure 23 Inter-process Reduction: Percentage File Sizes for Methods at Default Thresholds

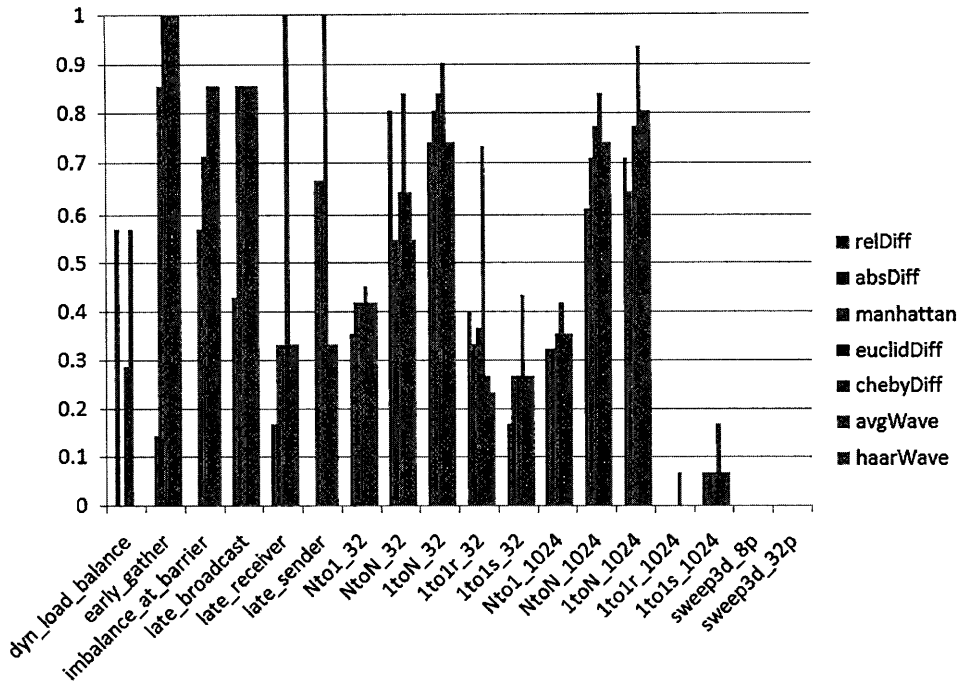


Figure 24 Inter-process Reduction: Degree of Matching for Methods at Default Thresholds

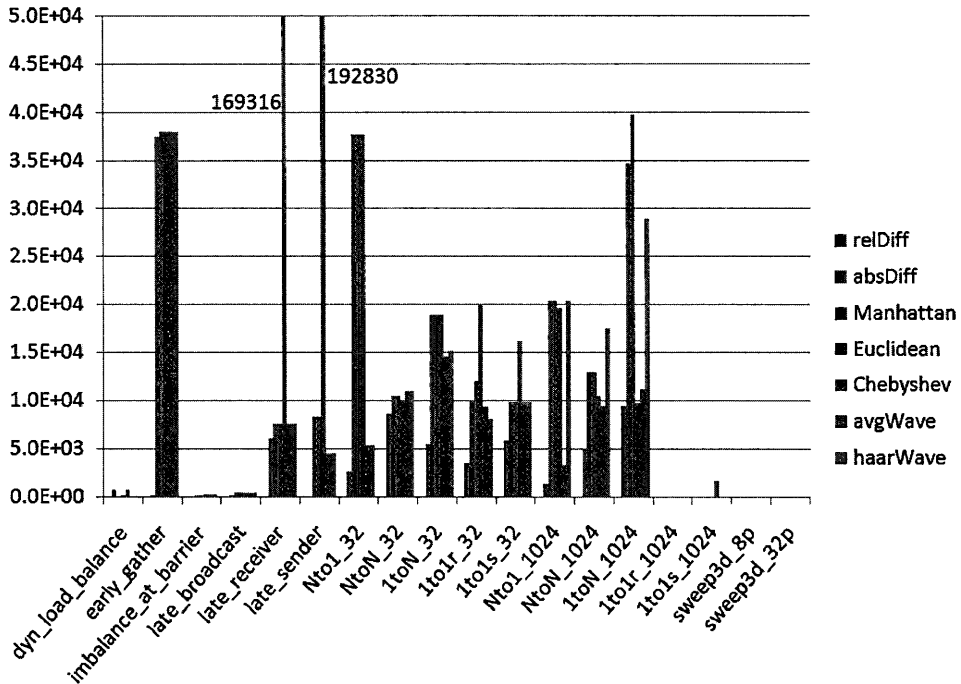


Figure 25 Inter-process Reduction: Approximation Distance for the Methods at Default Thresholds

acceptable matches for 1to1r_1024, which only achieved 2 matches of the 30 possible. For 1to1s_1024, all methods but *Chebyshev* found 2 matches of 30, while *Chebyshev* found 5. Additionally, none of the methods were able to find acceptable matches for sweep3d, so the percentage file sizes are all at 100% and degrees of matching are 0 for all methods for sweep3d_8p and sweep3d_32p. The in order rankings of the methods for this category were *Chebyshev* as the best performer, followed by *Euclidean*, *avgWave*, *haarWave*, *Manhattan*, *absDiff*, and *relDiff*.

5.3.2.2 Approximation Distance

Generally speaking, the approximation distances for all codes were relatively low (See Figure 25). The approximation distances for the regular benchmarks were, on average, an order of magnitude lower than those of the irregular benchmarks. The exception was the early_gather benchmark, which had relatively high approximation distance values for all methods, excluding *absDiff*. The approximation distances for 1to1r_1024, 1to1s_1024, sweep3d_8p, and sweep3d_32p are 0 or very low, since no or very little matches were found for those traces. Excluding *relDiff*, since it achieved no acceptable matchings, the in order rankings of the methods in this category were *absDiff*, *avgWave*, *haarWave*, *Manhattan*, *Euclidean*, and *Chebyshev*.

5.3.2.3 Retention of Performance Trends

For this category, *relDiff* retained trends for all programs. However, since *relDiff* did not find any inter-process matches, we exclude it as a contender for top-performing method in this category. *AbsDiff* and the wavelet transforms performed similarly on average for the regular and irregular benchmarks, correctly diagnosing

about 80% (*absDiff*) and 60% (*avgWave* and *haarWave*) of the programs from each category. *Manhattan* and *Euclidean* correctly diagnosed about 40% of the regular benchmarks, and 64% (*Manhattan*) and 72% (*Euclidean*) of the irregular ones. *Chebyshev* retained trends for 60% of the regular benchmarks, and only 36% of the irregular benchmarks. Since none of the methods found matches for *sweep3d*, trends were retained by default.

We show examples of the KOJAK diagnoses produced for *early_gather* and *NtoN_1024* in Figure 26 and Figure 27. For *early_gather*, we see that only *absDiff* was able to retain the correct performance trends. This is likely due to the low number of matches achieved by *absDiff* for this benchmark. None of the methods found acceptable reductions for *NtoN_1024*. All showed reduced severity for the “Barrier

	MPI Gather						do work
no loss	EX ██████████	MP ██████████	CM ██████████	CO ██████████	ER ██████████	EX ██████████	
relDiff	EX ██████████	MP ██████████	CM ██████████	CO ██████████	ER ██████████	EX ██████████	
absDiff	EX ██████████	MP ██████████	CM ██████████	CO ██████████	ER ██████████	EX ██████████	
Manhattan	EX ██████████	MP ██████████	CM ██████████	CO ██████████	ER ██████████	EX ██████████	
Euclidean	EX ██████████	MP ██████████	CM ██████████	CO ██████████	ER ██████████	EX ██████████	
Chebyshev	EX ██████████	MP ██████████	CM ██████████	CO ██████████	ER ██████████	EX ██████████	
avgWave	EX ██████████	MP ██████████	CM ██████████	CO ██████████	ER ██████████	EX ██████████	
haarWave	EX ██████████	MP ██████████	CM ██████████	CO ██████████	ER ██████████	EX ██████████	

Figure 26 Inter-process Reduction: KOJAK Performance Trends for *early_gather* for Each Method at Default Thresholds

	MPI Barrier						do work
no loss	EX █████	MP █████	SN █████	BA █████	WB █████	BC █████	EX █████
relDiff	EX █████	MP █████	SN █████	BA █████	WB █████	BC █████	EX █████
absDiff	EX █████	MP █████	SN █████	BA █████	WB █████	BC █████	EX █████
Manhattan	EX █████	MP █████	SN █████	BA █████	WB █████	BC █████	EX █████
Euclidean	EX █████	MP █████	SN █████	BA █████	WB █████	BC █████	EX █████
Chebyshev	EX █████	MP █████	SN █████	BA █████	WB █████	BC █████	EX █████
avgWave	EX █████	MP █████	SN █████	BA █████	WB █████	BC █████	EX █████
haarWave	EX █████	MP █████	SN █████	BA █████	WB █████	BC █████	EX █████

Figure 27 Inter-process Reduction: KOJAK Performance Trends for *NtoN_1024* for Each Method at Default Thresholds

Completion” diagnosis, and introduced variation in the severities at the rank level that don’t appear in the no-loss diagnosis. Additionally, all but *absDiff* reduced the severity of the “Wait at Barrier” diagnosis.

In this category, the *absDiff* method performed best, correctly diagnosing 14 out of the 18 programs. Next, all three of *Euclidean*, *avgWave*, and *haarWave* correctly diagnosed 12, followed by *Manhattan* (11) and *Chebyshev* (9).

5.3.2.4 Discussion

Generally speaking, we found that there were far fewer inter-process matches achieved with good results than we initially expected. We expected there to be a larger number of matches in the regularly behaving benchmarks, but discovered that, overall, a higher number of matches was found for the irregular benchmarks with a greater level of retention of trends. Upon inspection, we found that there were no possible inter-process matches for *sweep3d_8p*, given its message passing behavior and our matching criteria (We require that all message passing parameters, e.g. message tags and bytes match for the traces to match.). However, we were surprised that no acceptable matches of the 16 possible were found for *sweep3d_32p*.

In the file size reduction and degree of matching category, the top performers were *Chebyshev*, *Euclidean*, *avgWave*, *haarWave*. For the approximation distance category, the best methods were *absDiff*, *avgWave*, *haarWave*, and *Manhattan*. In the category of retaining performance trends, the best methods were *absDiff*, followed by three methods in a tie: *Euclidean*, *avgWave*, and *haarWave*. To choose the best overall method in this category, we consider both file size reduction and retention of trends.

Although *Chebyshev* produced the smallest data files, it produced reduced traces with the greatest amount of error and least retention of performance trends. Because *Euclidean*, *avgWave*, and *haarWave* performed very similarly and relatively well in file size reduction and retention of trends, we choose all three methods as the top methods for inter-process reduction.

5.4 Combined Inter-process and Intra-process Reduction Evaluation

In this section, we compare the abilities of the similarity methods to produce reduced traces using both intra- and inter-process reduction. Excluding *iter_k* and *iter_avg*, we use two different thresholds for each method, one for intra-process matching and the other for inter-process matching. For *iter_k* and *iter_avg*, we perform intra-process matching only. The thresholds we use in this study are those that we found to be the best for each method in Sections 6.2 and 6.3. For intra- and inter-process reduction respectively, the thresholds were: *relDiff* (0.8, 0.8), *absDiff* (10^3 , 10^4), *Manhattan* (0.4, 0.4), *Euclidean*, *Chebyshev*, *avgWave*, *haarWave* (0.2, 0.4). For *iter_k*, we used $k=10$, but no threshold for inter-process reduction since this method does not perform inter-process reduction. We evaluate the methods as we did for the intra- and inter-process only studies, for file size reduction, introduction of error into the reduced trace, and retention of correct performance trends. (For full results, refer to the Appendix.)

5.4.1 Size and Degree of Matching

For the degree of matching, we compute the sum of the intra- and inter-process matches that were found as a fraction of the total number of intra- and inter-process

matches that could possibly be found (See Figure 29). For intra-process matching, *iter_avg* decides all segments with the same context match. Because the possible number of intra-process matches is much higher than the possible number of inter-process matches, *iter_avg* has the highest degree of matching overall. The ranking of the methods in order from highest average degree of matching to lowest is: *iter_avg*, *avgWave*, *haarWave*, *absDiff*, *Euclidean*, *Chebyshev*, *Manhattan*, *iter_k*, and *relDiff*.

Unlike when considering intra- and inter-process matching in isolation, the expected file size does not directly follow the degree of matching (See Figure 28). A method such as *iter_avg* that achieves a high degree of intra-process matching but doesn't perform inter-process matching can generate larger reduced trace files, because a single inter-process match has the potential for more file size savings than multiple intra-process matches. The methods in order of smallest average file size to largest are: *Chebyshev*, *avgWave*, *Euclidean*, *haarWave*, *Manhattan*, *absDiff*, *iter_avg*, *iter_k*, and *relDiff*.

5.4.2 Approximation Distance

We show the results for the approximation distance in Figure 30. Overall, the Chebyshev distance introduced the most error and *relDiff* introduced the least error into the reduced traces. On average, more absolute error was introduced into the reduced traces of the regular benchmarks than the irregular benchmarks. The methods

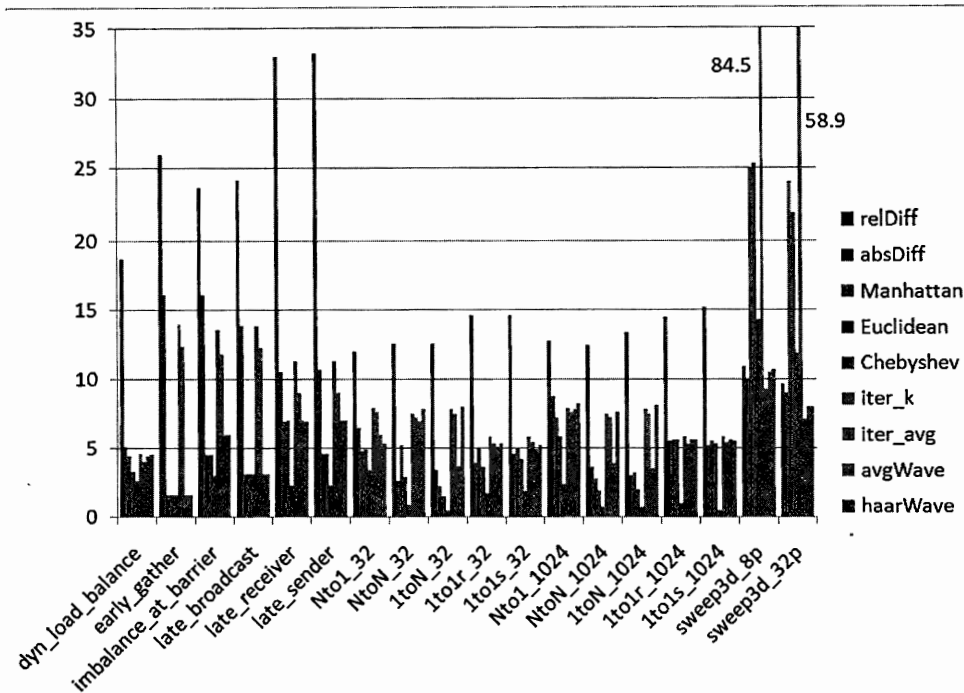


Figure 28 Combined Reduction: Percentage File Sizes for Methods at Default Thresholds

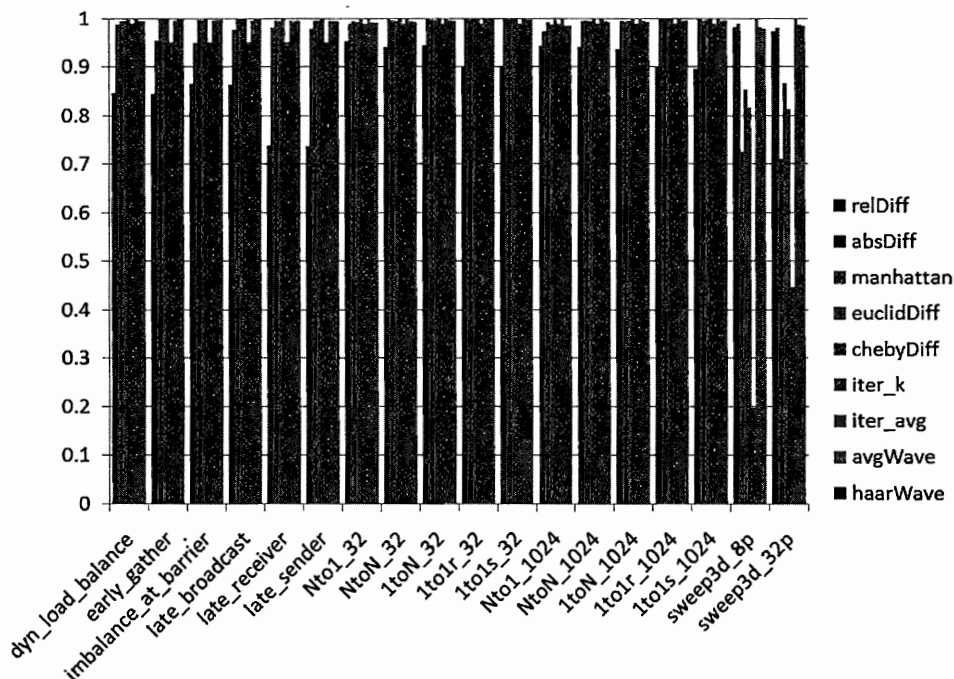


Figure 29 Combined Reduction: Degree of Matching for Methods at Default Thresholds

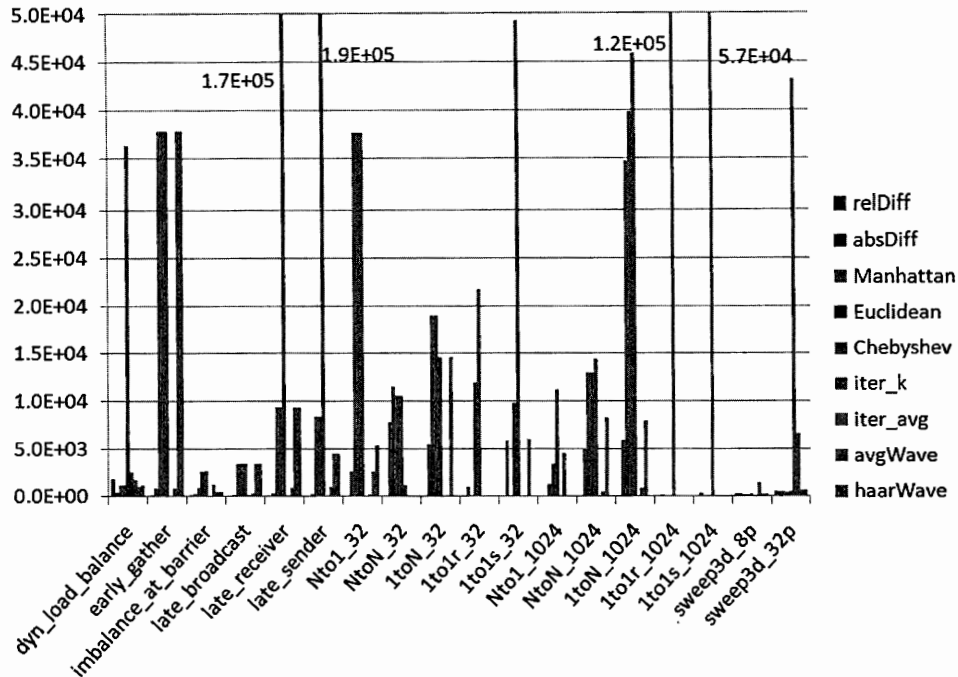


Figure 30 Combined Reduction: Approximation Distance for Methods at Default Thresholds

in order of smallest to largest average approximation distance are: *relDiff*, *iter_avg*, *absDiff*, *iter_k*, *haarWave*, *avgWave*, *Manhattan*, *Euclidean*, and *Chebyshev*.

5.4.3 Retention of Trends

Overall, the methods were able to produce the most acceptable reduced traces for the regular benchmarks. An exception was the *late_receiver* benchmark, for which none of the methods retained performance trends. All reported reduced severity for the “Late Receiver” diagnosis and lost the correct rank-level severities for that diagnosis

The methods did less well for the irregular benchmarks. Notably, none of the methods produced acceptable reduced traces for *NtoN_32* or *NtoN_1024*. We show the results for *NtoN_32* in Figure 31. For all of the methods, the severity of “Wait at Barrier” is under-reported and the rank-level severities do not match those of the no-

	MPI Barrier						do work
no loss	EX	MP	SN	BA	WB	BC	EX
relDiff	EX	MP	SN	BA	WB	BC	EX
absDiff	EX	MP	SN	BA	WB	BC	EX
Manhattan	EX	MP	SN	BA	WB	BC	EX
Euclidean	EX	MP	SN	BA	WB	BC	EX
Chebyshev	EX	MP	SN	BA	WB	BC	EX
iter_k	EX	MP	SN	BA	WB	BC	EX
iter_avg	EX	MP	SN	BA	WB	BC	EX
avgWave	EX	MP	SN	BA	WB	BC	EX
haarWave	EX	MP	SN	BA	WB	BC	EX

Figure 31 Combined Reduction: KOJAK Performance Trends for NtoN_32 for Each Method at Default Thresholds

loss trace. *Iter_k* and *iter_avg* did the worst for this benchmark, with severities misreported for all of the diagnoses.

None of the methods produced reduced traces that retained trends for sweep3d. We show the KOJAK diagnoses for sweep3d_32p in Figure 32. All methods reported reduced severity for “Execution Time” in the *sweep* function. All but *iter_k* showed reduced severity in “Late Sender”, while *iter_k* showed different rank-level severities than the results from the no-loss trace. The *iter_k* method did the worst overall, showing increased severities for all but the “Late Sender” diagnosis in *pmpi_recv*.

	pmpi_recv						sweep
no loss	EX	MP	CM	P2	LS	MO	EX
relDiff	EX	MP	CM	P2	LS	MO	EX
absDiff	EX	MP	CM	P2	LS	MO	EX
Manhattan	EX	MP	CM	P2	LS	MO	EX
Chebyshev	EX	MP	CM	P2	LS	MO	EX
iter_k	EX	MP	CM	P2	LS	MO	EX
iter_avg	EX	MP	CM	P2	LS	MO	EX
avgWave	EX	MP	CM	P2	LS	MO	EX
haarWave	EX	MP	CM	P2	LS	MO	EX

Figure 32 Combined Reduction: KOJAK Performance Trends for sweep3d_32p for Each Method at Default Thresholds

The *relDiff* method produced the largest number of acceptable reduced traces, 10 out of 18. *Manhattan*, *Euclidean*, *avgWave*, and *haarWave* retained correct performance trends in 9 reduced traces. The other methods in order are *absDiff* (8), *Chebyshev* (7), *iter_k* (6), *iter_avg* (4).

5.4.4 Discussion

The best performing methods in the file size reduction category were *Chebyshev*, *avgWave*, and *Euclidean*. From the approximation distance category, the best methods were *relDiff*, *iter_avg*, and *absDiff*. The methods that best retained performance trends were *relDiff*, followed by a tie between *Manhattan*, *Euclidean*, *avgWave*, and *haarWave*. Given its performance in the file size reduction and retention of performance trends categories, we choose *avgWave* as the best method for reducing traces using both intra- and inter-process reductions.

5.5 Discussion

Here we discuss our expectations for the similarity methods and matching scenario: intra-process only, inter-process only, and combined intra- and inter-process matching.

5.5.1 Trace Similarity Methods

For *relDiff*, we expected low error and relatively large files, which is exactly what we found to be true. For *absDiff*, we expected low error. We did find that *absDiff* had lower error when compared to most methods. We expected the Minkowski distances would favor long segments and error would be lowest for *Manhattan*, followed by

Euclidean, and highest for *Chebyshev*. While we did definitely see more error in the traces produced by the *Chebyshev* method, the differences in the results for the *Manhattan* and *Euclidean* methods were largely undistinguishable. We expected *iter_k* and *iter_avg* to produce low error traces for programs with regular behavior and for *iter_avg* to have the lowest overall file sizes. We indeed found that *iter_k* did well for regularly behaving programs and less well for programs with varying behavior patterns. *Iter_avg* produced better results for the regular benchmarks than the irregular ones; the averaging of measurements tended to cause loss of information needed for diagnosis. For *avgWave* and *haarWave*, we expected stricter comparisons than *Euclidean*. Indeed, the wavelet transforms produced slightly larger files for the benchmark traces; however, the reduced traces of *sweep3d* were smaller than those produced by *Euclidean*.

5.5.2 Intra- and Inter-process Matching

We expected the trace similarity methods to identify high degrees of both intra- and inter-process matches, and that the number of intra-process matches would be much higher because of the higher number of possible intra-process matches. We expected that inter-process matches would yield the greatest gains in terms of file size and a similar level of retention of performance trends across intra- and inter-process matching.

We found that the results for intra-process only matches followed our expectations, but that the results for inter-process only matches did not. While inter-process matching did achieve the highest gains in terms of file size reduction, there

were a much lower number of inter-process matches that retained correct performance behaviors than we expected. This is due to the larger number of measurements that must match according to the similarity method used in order for an inter-process match to be successful, from differing message passing parameters across ranks, and slight variations in events in initialization segments

5.6 Summary

We developed a post-mortem trace profiler and used it to demonstrate the viability of trace profiling for trace size reduction and for producing reduced traces that retain the behaviors needed for correct performance analysis. Additionally, we developed a new methodology for evaluating definitions for similarity between event traces for the purpose of performance analysis. We identified criteria for comparing the similarity methods: file size reduction, degree of matching, approximation distance, and retention of correct performance trends. We applied these criteria, using benchmarks with known performance behaviors, as well as with the application *sweep3d*. We evaluated the similarity methods for how well they reduced traces using intra-process reduction only, inter-process reduction only, and combined intra- and inter-process reductions.

For intra-process reduction, the *avgWave* method had the best retention of performance behaviors and good trace file size reduction. The greatest trace file reductions were achieved with the *iter_avg* method; however, the error in those traces led to loss of important performance trends in the data. Because of this we found that

using the *avgWave* method was the best trade-off in terms of error in the reduced trace and file size reduction.

In our inter-process reduction study, we discovered that less matching occurred than what we expected. *Euclidean*, *avgWave*, and *haarWave* performed very similarly and relatively well in file size reduction and retention of trends, so we choose all three methods as the top methods for inter-process reduction. We found that *Chebyshev* produced the smallest data files, and that it produced reduced traces with the greatest amount of error and least retention of performance trends, so it was not chosen as the best method in this study.

For combined intra- and inter-process reduction, again *Chebyshev* produced small files with large amounts of error and lost trends. Based on the ability of *avgWave* to produce reduced traces that are relatively small with low error and high rate of retention of performance trends, we chose it as the best method for combined intra- and inter-process reduction.

6 Prototype Runtime Trace Profiler

We demonstrated the viability of the trace profiling technique in terms of correctness and file size reduction in our post-mortem studies in Chapter 6. Here, our goal is to demonstrate that the overhead of writing the collected trace data is lower with a runtime trace profiler than with a traditional tracing tool. In this chapter, we first describe our current implementation of a prototype runtime trace profiler.² Then, we detail our experimental setup for evaluating the overheads and resulting files of the prototype against a state-of-the-art traditional tracing tool on a typical high-end Linux cluster. Finally, we present the results of our experiments.

6.1 Current Prototype Implementation

The current runtime trace profiler consists of a front end and a back end trace profiler instrumentation library. The front end launches the program to be measured and inserts calls in the program to the trace profiler instrumentation library. The trace profiler instrumentation library contains routines that implement the trace profiling technique. This initial implementation supports single-threaded MPI applications.

6.1.1 Trace Profiler Front End

The front end of the trace profiling tool starts and controls the execution of the target process, locates instrumentation points, and inserts calls to the trace profile instrumentation library at those points. We use the Dyninst dynamic instrumentation library for process control and instrumentation insertion [29]. We start a separate

² In this version of the prototype, we reduce the amount of collected data with intra-process merging only. Inter-process merging is left as a post-mortem activity.

front end process for each rank in the parallel run by starting the front end as a parallel job and giving it arguments that indicate the program to start and measure, as well as other arguments that control measurement details, e.g. the distance metric to use to compare segments and comparison thresholds. For example, if we are running on a machine that uses the `srun` command to start parallel jobs, the command `srun -n 8 ./traceProfiler -d haar_wave targetProgram` would start 8 front end `traceProfiler` processes according to the policies of the resource manager on the machine and compare them using the `haarWave` distance metric. Each front end would be responsible for a single instance of `targetProgram`.

Before starting its target program, the front end sets the environment variable `LD_PRELOAD` to load the back end library into the measured process when it is started. By doing this, the measurement routines in the trace profile instrumentation library are available and can be called from the measured process. Next, the front end creates the process, but does not execute it until after inserting instrumentation. The front end locates the functions and loops in the program. It assigns identifiers to all functions and context names and identifiers to all segments, and passes the names and identifiers to the trace profile instrumentation library. It instruments the entry and exit of all functions with calls into the trace profile instrumentation library. Segment markers are inserted with calls to the trace profile instrumentation library. An initial segment is started at the entry to `main` or `MAIN_`. Then, for each loop that contains a user-specified number of function calls, the current segment is stopped and a new segment is started at the top of the loop and stopped at the bottom of the loop. At the

end of the loop execution a new segment is started. At program termination, the final segment is stopped³. Segment contexts for non-loop portions of code are named as a concatenation of the enclosing function name and an integer that makes the name unique. Segment contexts for loops are assigned as a concatenation of the enclosing function name, the hierarchical loop name as assigned by Dyninst, and an integer that makes the segment name unique. See Figure 33 for an example code snippet with segment marker instrumentation. Note that the consequences of marking segments in this manner mean that some segments will contain no events, e.g `main_2`, all segments are disjoint, and that it is possible for a function's entry and exit to cross segment boundaries if the function or its callees contain a loop that is marked as a segment.

After inserting all instrumentation, the front end starts the execution of the measured process. At termination of the measured process, the front end process exits.

6.1.2 Trace Profiler Instrumentation Library

In this section, we describe the interface to the trace profile instrumentation library and its runtime operations.

6.1.2.1 Instrumentation Interface

We show the interface to the trace profile instrumentation library in Table 5. The entry and exit of functions are recorded by calls to the `enterRoutine` and

³ If the target program is a Fortran application, a call to `exitRoutine` for `MAIN_` is executed before the final segment is stopped. In some Fortran implementations, the `MAIN_` function is part of the Fortran library and not part of the user code. It is responsible for executing the main program unit of the user's Fortran application. As a result, the `MAIN_` function may not exit.

```

int main(){
    enterSegment("main_0");
    MPI_Init();
    exitSegment();
    for(i=0; i < 100; ++i){
        enterSegment("main_loop_1_1");
        do_work();
        MPI_Allgather();
        exitSegment();
    }
    enterSegment("main_1");
    exitSegment();
    for (j=0; j < 10; ++j){
        enterSegment("main_loop_2_1");
        do_other_work();
        exitSegment();
        while(k < otherRanks){
            enterSegment("main_loop_2.1_1");
            MPI_Sendrecv();
            exitSegment();
        }
        enterSegment("main_2");
        exitSegment();
    }
    enterSegment("main_3");
    MPI_Finalize();
    exitSegment();
}

```

Figure 33 Example Segment Context Marking and Names

exitRoutine functions, respectively. Segment boundaries are marked with calls to enterSegment and exitSegment. We use the PMPI interface to selectively collect details about MPI message passing activities. The function MPI_Init contains instrumentation to call the trace profile instrumentation library function, openTrace, which performs initialization activities. The MPI function definitions for sending and receiving operations contain calls to sendMessage and recvMessage which record details about sends and receives: source or target rank, bytes transferred, message tag, and communicator. In Figure 34, we show the instrumented definition of MPI_Send. When the user code calls MPI_Send, the instrumented function in the trace profile instrumentation library is executed, which in turn calls PMPI_Send, the actual call into the message passing library.

```

int MPI_Send( void *buf, int count, MPI_Datatype datatype, int dest, int
tag, MPI_Comm comm){
    int returnVal;
    int typesize;

    if (dest != MPI_PROC_NULL) {
        PMPI_Type_size( datatype, &typesize );
        sendMessage(translateRankToWorld(comm,dest), typesize*count, tag,
comm);
        returnVal = PMPI_Send(buf, count, datatype, dest, tag, comm);
    }
    return returnVal;
}

```

Figure 34 Example Instrumentation for Message Passing Function

6.1.3 Runtime Operations

The current version of the trace profiler front end takes arguments that specify the distance metric to be used for comparing segment data and comparison thresholds. The methods implemented into the runtime prototype are those that are described in Section 5.3. Intra-process segment matching is performed at runtime and inter-process segment matching is a post-mortem activity.

When the measured process is started, it makes an initial call to `enterSegment` to create the first segment and a call to `enterRoutine` for the entry to `main` or `MAIN_`. Presumably, the first function call in the measured program is to `MPI_Init`. In the instrumentation for `MPI_Init`, the trace profiler performs initialization activities, such as setting up the data structures for storing the collected data and getting the rank identifier for the process. Then, all function event data is collected in the `enterRoutine` and `exitRoutine` functions until the next call to `exitSegment`. When `exitSegment` is called the first time, there are no other segments stored that could be a potential match, so the first segment is inserted into

Table 5 Trace Profile Instrumentation Library Interface

INSTRUMENTATION LIBRARY FUNCTION	ACTIONS
defineEvents (eventList)	Called before target program execution. The argument eventList is a listing of function and segment identifiers and names.
openTrace (rank)	Called in MPI_Init. Performs initialization activities.
endProgram ()	Called at program termination. Calls exitSegment ³ .
enterSegment (id)	Called at a segment entry marker. Sets current segment.
exitSegment ()	Called at segment exit marker. Exits current segment.
enterRoutine (id)	Called at function entry to record function entry time and identifier.
exitRoutine (id)	Called at function exit time to record exit time and identifier.
sendMessage (dest, bytes, tag, comm)	Called from instrumented MPI calls to record details about send operations.
recvMessage (source, bytes, tag, comm)	Called from instrumented MPI calls to record details about recv operations.
MPI_*	Intercept calls to the actual MPI library and call appropriate trace profile instrumentation library function to record message passing details. They also execute the call into the MPI library to execute the operation, e.g. MPI_Send calls sendMessage and PMPI_Send.

the list of stored representative segments. For all subsequent calls to enterSegment, a new segment is created. At the matching call to exitSegment, the segment is terminated and compared using the selected distance metric against the segments with the same context that have already been stored as representatives.

6.2 Experimental Setup

In this section, we report on the experiments we performed to evaluate the overheads of our current runtime prototype trace profiler (TP). We evaluate our prototype against a state-of-the-art traditional tracing tool, TAU. We evaluate both tools for instrumentation and writing overhead as defined in Chapter 4, and for resulting file size.

6.2.1 Application

We evaluated the prototype using Sweep3d [3] described in Section 6.1.2. We ran the application with a problem size of $5 \times 5 \times 6400$ with $MK=30$ and $MMI=2$. We measured the application with 32, 64, 128, 256, 512, and 1024 processors.

6.2.2 Machine

We ran our experiments on Hera at LLNL. Hera is an 864-node Linux cluster, where each node contains 4 AMD quad-core processors, for a total of 16 CPUs per node. The nodes are connected by an Infiniband switch and are connected to a Lustre file system. For each experiment, we ran jobs that utilized all CPUs on each node, e.g. a 32-process run spanned two nodes, and wrote all trace data to the Lustre file system. The configuration of Hera is very similar to the machine shown in Figure 2.

6.2.3 Tool Configurations

We used TAU version 2.17.1 [58] for our experiments. We configured TAU to collect entry and exit events for all functions. Note that we did not insert segment

markers into the program when measuring with TAU, so the number of instrumentation points for TAU was lower than it was for our trace profiler prototype.

Our experiments here are similar to those described in Section 4.1. We performed runs without trace instrumentation (*noInstr*), and with and without buffer flush to file enabled (*write* or *noWrite*). Additionally, we experimented with two different buffer sizes in our experiments: 1.5 MB (*def*, default size for TAU) and 8.0 MB (*8MB*, default size for the widely-used MPE [80]). We altered the buffer size and *write/noWrite* configuration of TAU as described in Section 4.1. The choices of buffer size and *write/noWrite* configuration for our prototype trace profiler are runtime options. To measure execution time, we used the wall clock time reported by the application. We evaluate only overhead that occurs after `MPI_Init` and before `MPI_Finalize`; this means that initial file creation and file closing are not included in the overheads for both tools. Additionally, we do not evaluate any post-mortem activities, such as trace file merging for TAU or inter-process matching for our prototype.

When the buffer of either tool is full, the trace data is flushed to disk and the buffer is emptied to collect more data. The flushing policy of our prototype is somewhat different from that of TAU's. TAU simply creates a fixed-size buffer and inserts a series of fixed-size trace records into it. Our implementation creates data structures to hold process groups and the segments contained in each process group. We flush TAU's buffer when the amount of data collected is exactly the maximum buffer size. We flush the data structures from our prototype at the end of a segment when the

amount of data collected meets or exceeds the buffer size. The consequences of this are that our tool may flush less frequently than TAU.

We implemented a simple buffer flushing policy for our prototype. We simply flush the data for the entire process group and then reset the process group data to empty. We append the process group data at the end of the file, and additionally update the section header containing the number of process groups in the file.

We ran 30 identical runs for each combination of number of processes, tracing tool, buffer size, *write* or *noWrite*, and for our prototype, flushing policy. For our prototype runs, we used the *avgWave* distance metric for segment matching, with threshold 0.2 and inserted segment markers in loops that contained at least 10 function calls. We report the average timing information for each configuration. For TAU, the trace file sizes and number of flushes are deterministic, so we report exact values. For our prototype, the sizes and number of flushes vary depending on how many segment matches occur at runtime. We report the averages of these measurements.

6.3 Results

In this section, we present the results of our experiments. We evaluate the tracing tools for instrumentation and writing overhead, size of generated files, and flush count.

6.3.1 Execution Time

We show the execution time of *sweep3d* measured under the various configurations with increasing processor count in Figure 35. The execution times for the application with no instrumentation are labeled *noInstr*. The first four bars after *noInstr* in each processor-count grouping show the TAU runs; the last four bars show

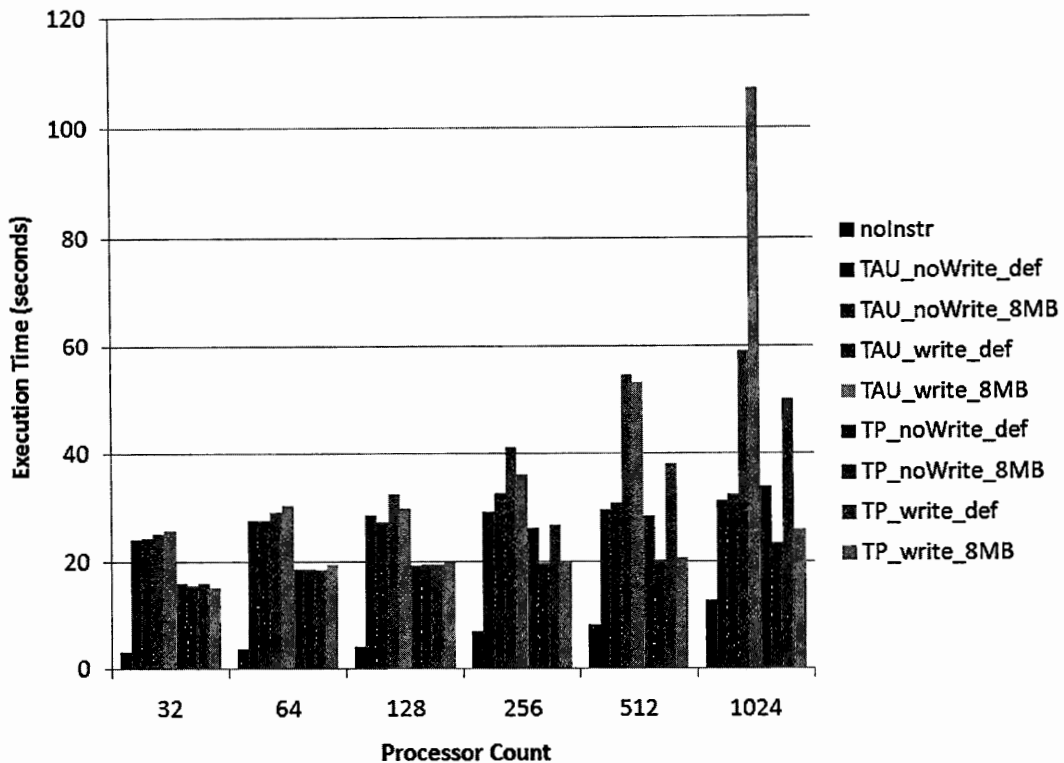


Figure 35 Execution Time of Sweep3d Measured with TAU and TP

the results for our prototype (TP). The difference between the execution times for *noInstr* and the *noWrite* runs shows the amount of instrumentation overhead caused by each tool and buffer size. The difference between the *noWrite* and *write* runs shows the writing overhead of the tool configuration.

We examine the writing overhead in more detail in Figure 36. At the smaller processor counts (32, 64, and 128), we see that the writing overhead is not very detectable. However, at the larger processor counts the writing overhead for TAU and our prototype increases with increasing processor count. At the default buffer size, the writing overheads of TAU and of our prototype become more noticeable and increase with increasing processor count. However, our prototype introduces less overhead

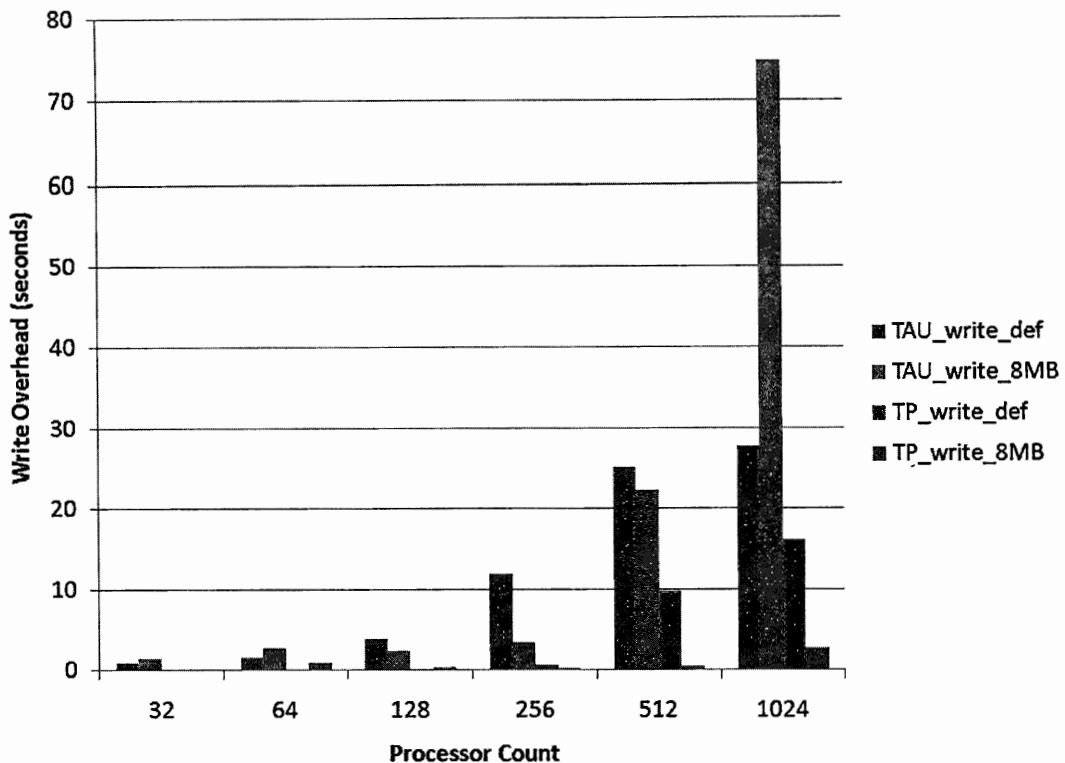


Figure 36 Write Overhead for Sweep3d with TAU and TP

than TAU. With the 8 MB buffer, the writing overhead of TAU increases dramatically with increasing processor count, but the writing overhead for our prototype using the 8 MB buffer increases very slowly with increasing processor count.

6.3.2 Total File Size

We show the sum of file sizes generated during the *write* runs in Figure 37 and the average file size per rank in Figure 38. In all cases, the amount of data written increases with increasing processor count. However, the file sizes generated using our prototype did not increase as rapidly as did those of TAU. When using the default buffer size with our prototype, the total amount of data was higher than when using the 8 MB buffer.

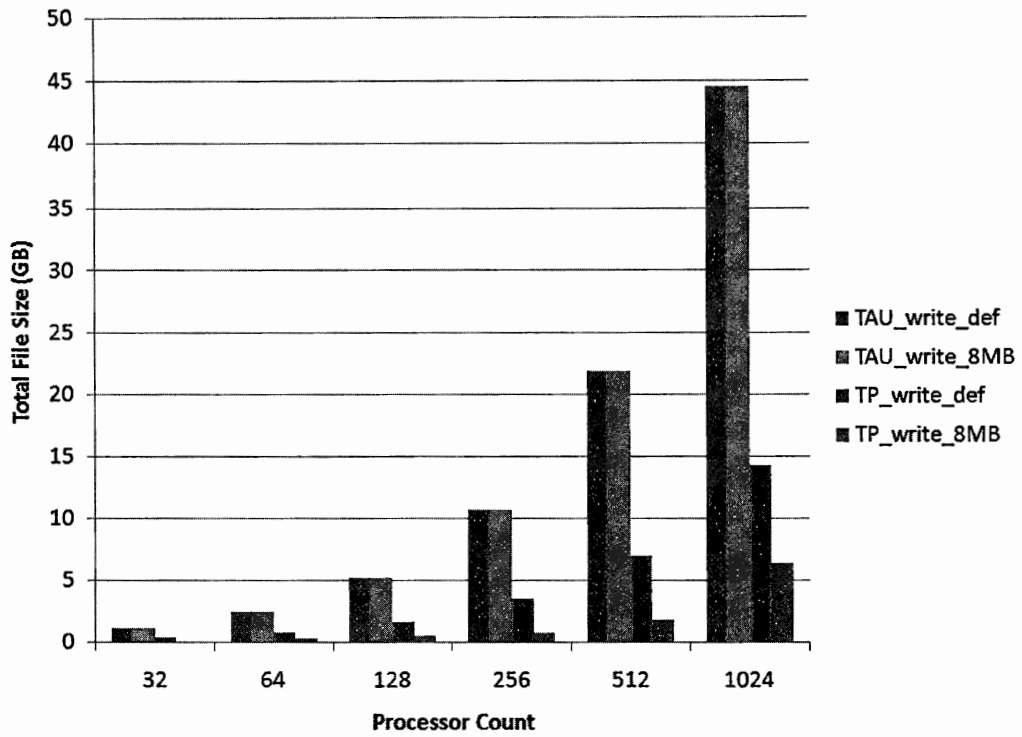


Figure 37 Total Size of Files Generated for Sweep3d with TAU and TP

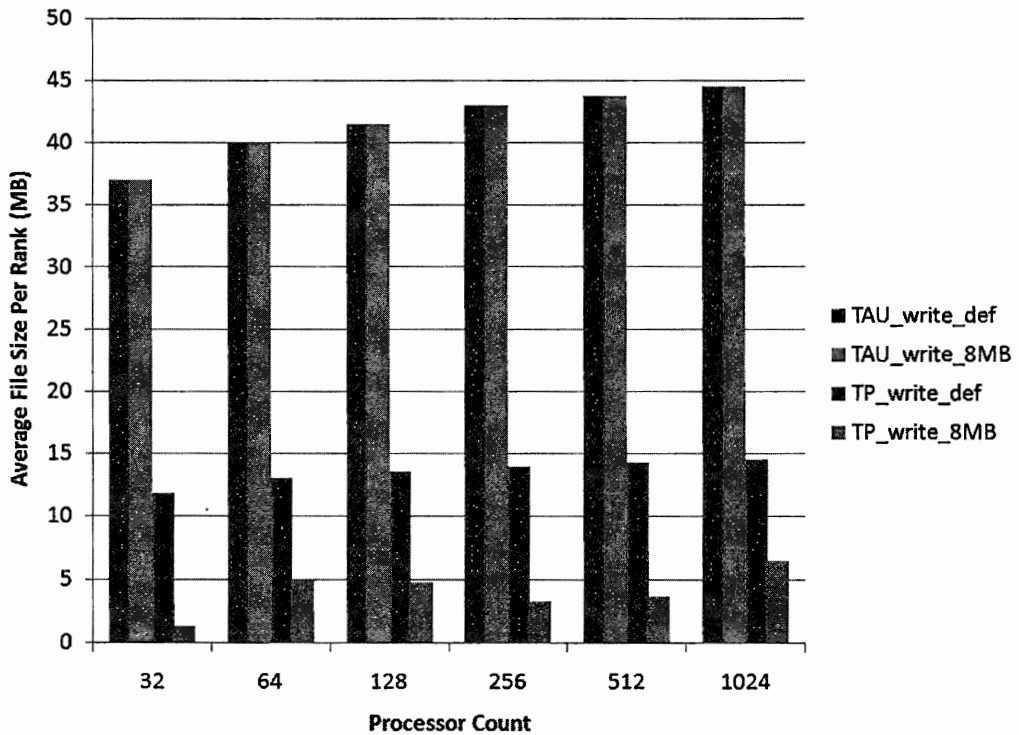


Figure 38 Average File Size Per Rank for TAU and TP

6.3.3 Flushes

The total buffer flushes over all ranks executed by each tool configuration are shown in Figure 39 and the average per rank flushes are shown in Figure 40. TAU with the default size buffer generated the most buffer flushes overall, followed by our prototype using the default size buffer. The least amount of flushes was executed by our prototype using the 8MB buffer. In all cases, the number of flushes increased with increasing processor count.

6.4 Discussion

Our goal for trace profiling was to reduce the overheads of tracing by reducing the amount of trace data being written to disk during runtime. The writing overhead and resulting data files from our trace profiler prototype were much smaller than those of TAU. As expected, the writing overheads still scaled with the number of ranks in the run, because of contention for the shared file system resources. However, the overheads did not increase nearly as quickly as with TAU. In general, we found that the instrumentation overhead of our prototype was on the same order of the instrumentation overhead of TAU.

The choice of buffer size for TAU had much less of an impact on overheads and file size than it did with our prototype. Of course, the choice of buffer size would not impact the file sizes generated by TAU; the same number of events will be written regardless. The buffer size choice greatly impacted the performance of our prototype. The larger buffer size allowed the tool to find a larger number of segment matches, resulting in less flushes, less writing overhead, and smaller data files.

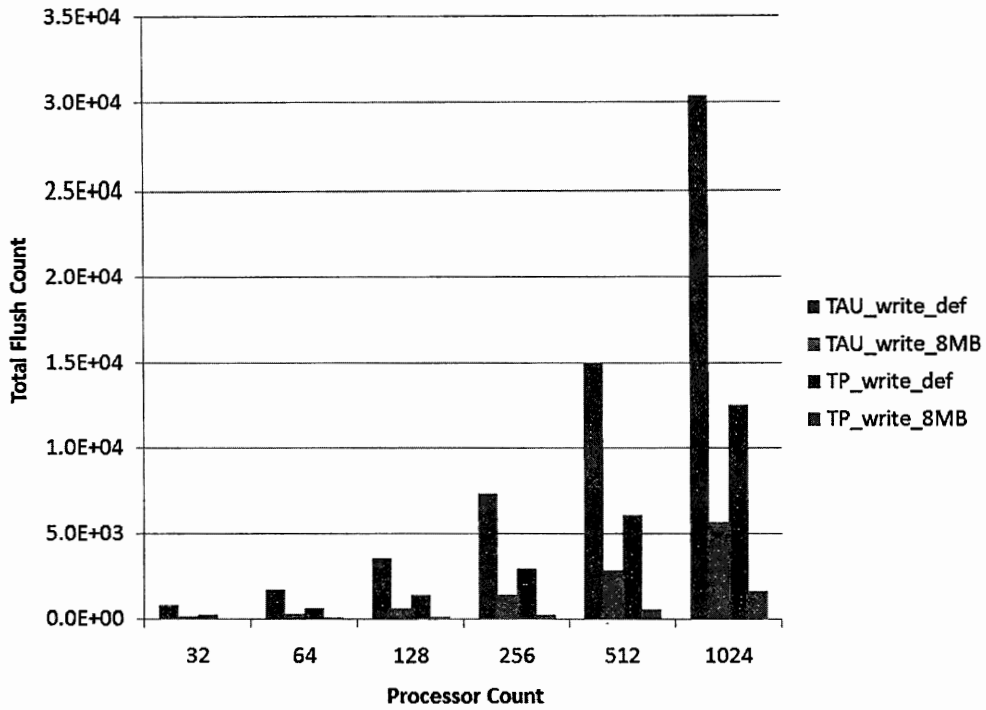


Figure 39 Total Buffer Flush Count for Sweep3d with TAU and TP

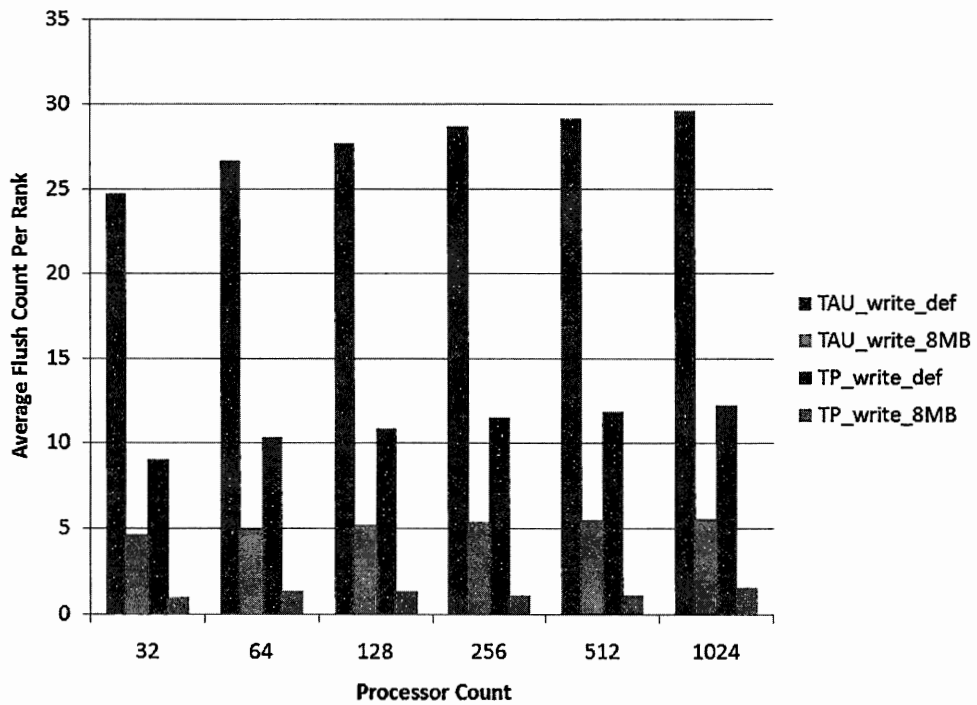


Figure 40 Average Flush Count Per Rank with TAU and TP

7 Conclusions

In this dissertation, we present a novel performance measurement technique for collecting event-based performance data and demonstrate its viability for low-overhead event trace collection for the purpose of parallel performance analysis.

Our first contribution is a study of the overheads of traditional event trace collection. We demonstrate that event tracing using traditional methods on high-end parallel systems is not scalable. The act of collecting such highly detailed performance information and periodically flushing the collected data to disk unduly perturbs the measured program. Additionally, the trace files created scale with the running time and number of concurrent entities in the parallel run. Our study indicates that the major scalability problems of traditional tracing are the overhead due to periodically flushing the event data to disk and the large resulting trace files.

The second contribution of this dissertation is trace profiling, a new low-overhead measurement technique for gathering event-based performance data. Trace profiling is a hybrid of tracing and profiling and collects summary information about event patterns that occur during program execution. Trace profiling addresses the major scalability problems of traditional tracing: periodic flushing of trace data to disk; and the unmanageably large trace files that are generated. The technique detects repeated event patterns both within and across processes in a parallel run. Because intra-process event pattern matching is done at runtime, a reduced data volume is flushed to disk during execution, which results in lower tool overhead due to writing. Additionally,

the sizes of the files generated by trace profiling are greatly reduced compared to traditional tracing.

The third research contribution is a study of similarity metrics for identifying patterns in event traces. We evaluate several metrics for file size reduction, introduction of error, and retention of correct performance behaviors in the reduced trace. In our study, retention of performance trends was the most important criteria for evaluating similarity metrics for trace reduction. Our study indicates that the average wavelet transform method performs the best in terms of retention of performance trends and file size reduction. This study demonstrates the viability of trace profiling in terms of its ability to collect useful traces that retain the important behavior patterns at a reduced data volume.

Our fourth contribution shows the low overheads of runtime trace profiling. We implemented a prototype runtime trace profiler and evaluated it against a state-of-the-art traditional tracing tool on a typical high-end Linux cluster. We demonstrate that the overheads of collecting event-based measurement data using trace profiling are lower than that of traditional tracing and that the resulting data files are smaller.

We conclude that trace profiling is a viable method for low-overhead collection of event-based performance data on high end systems.

7.1 Future Work

Potential directions for future work include: investigation of memory bounds for performance tools; and evaluation of the implementation choices for a trace profiler and their consequences in terms of measurement overheads.

7.1.1 Performance Tool Memory Bounds

Any performance measurement tool that is designed to measure large, long-running parallel programs must bound the amount of memory used in order to be scalable; a tool cannot simply store all collected data in memory without potentially incurring serious consequences to the application performance. In our runtime study, we discovered that the choice of memory bound for storage of event data impacted the performance of our prototype. The bound affected the amount of event pattern matching that could occur at runtime, and thus the amount of data that was flushed periodically during the run. Use of a smaller memory bound resulted in more flushes, less event pattern matching, and larger resulting data files than use of a larger memory bound. In our study, we experimented with two memory bounds, chosen because they are the default buffer sizes for two commonly-used traditional tracing tools. In our experience, performance tools either have hard-coded default memory bounds or allow the user to choose the bounds to be used. In either case, there is no guidance given to the user as to what bound would be a good choice for a particular performance tool measuring a particular application class on a particular architecture. A direction for future work would be to develop a model for describing the interactions between tool memory bounds, application characteristics, and architecture. The model could be used to provide guidelines for choosing the best memory bounds for a given situation.

7.1.2 Trace Profiler Measurement Overheads

Although all performance measurement techniques introduce perturbation of varying degrees into the program being measured, performance measurement with a trace profiler implementation has the potential to introduce irregular perturbation. In

the best case, when measuring regularly behaving programs with a trace profiler, there will be a high degree of matching of event patterns, which will greatly reduce the number of comparisons that need to be performed during runtime. However, in the worst case, when measuring programs that have irregular behavior over time, the number of intra-process event pattern matches will likely decrease, meaning there will be a larger number of comparisons that need to be performed over time during runtime and possibly more data that needs to be flushed to honor the memory bounds of the tool. Additionally, in a parallel run, if more matches are found in some ranks than others, then the amount of time spent in comparison operations across ranks will vary. This could introduce perturbation that could affect the behavior of other ranks, if, for example, some ranks are waiting for communication from ranks that have a larger number of event pattern comparisons to make. Future research could examine the potential consequences of the perturbation introduced by trace profiling for different classes of programs.

The choice of where to introduce segment markers into a program has the potential to impact both the amount of perturbation introduced into the program and the number of event pattern matches that can be identified. If segment markers are placed in all loops, then the instrumentation overhead increases because more instrumentation instructions are executed. However, the number of segment matches is likely to increase greatly, because the amount of event data in each segment is smaller. If segment markers are placed in loops more selectively, then instrumentation overhead decreases, but segment matching might decrease because of the larger amount of event

data in each segment. A direction of future research would be to investigate the tradeoffs of segment marking policies.

Our evaluation of the runtime prototype included a single simple policy for honoring the memory bounds for storing event data: flush all event data when the memory bounds are reached. Although this policy was very simple, it had the disadvantage of flushing event patterns that could potentially match future event patterns, resulting in less matching and larger file sizes. One possible option for honoring the memory bounds is to never write any data to disk during the run, but instead compress or fold the data in some manner. For example, when the amount of data collected by the Paradyn performance tool reaches the memory bound, adjacent data bins are averaged and the memory size is reduced by half [42]. A trace profiler might increase the given threshold and reevaluate the stored segments for further matches to reduce memory usage. However, care would need to be taken to ensure that performance trends are not lost by allowing more error into the reduced trace. Future studies of trace profiling could investigate variations on policies for honoring memory bounds and their impact on the scalability of the technique in terms of flush counts and resulting data file sizes.

In our evaluation of the prototype runtime trace profiler, we evaluated the prototype for overheads that occur at runtime, which excludes inter-process matching overheads. Although post-execution inter-process merging would not perturb the measured program, the computation time for the merging should still be scalable. An option for scalable inter-process merging is to explore runtime merging. Segments that

are flushed during runtime could be checked for inter-process matches using a tree-based data reduction infrastructure such as MRNet [56], which would reduce the overall amount of data being written to disk, and in turn, reduce the writing overhead. An avenue of future work is to investigate and evaluate scalable methods for inter-process merging in a trace profiler implementation.

8 References

- [1] Sphot benchmark. <http://www.llnl.gov/asci/purple/benchmarks/limited/sphot/>. downloaded Dec. 8, 2006.
- [2] Intel trace collector 7.0 user's guide. <ftp://download.intel.com/support/performance-tools/cluster/analyzer/sb/itreferenceguide.pdf>, January 2007.
- [3] The ASCI sweep3D readme file. http://www.c3.llnl.gov/pal/software/sweep3d/sweep3d_readme.html, January 2009.
- [4] Maria Gabriel Aguilera, Patricia J. Teller, Michela Taufer, and F. Wolf. A systematic multi-step methodology for performance analysis of communication traces of distributed applications based on hierarchical clustering. In *IPDPS*, 2006.
- [5] Allinea user's guide version 1.2. Available by request from support@allinea.com, February 2007.
- [6] Dorian Arnold, Dong H. Ahn, Bronis R. de Supinski, Gregory Lee, Barton P. Miller, and Martin Schulz. Stack trace analysis for large scale applications. In *International Parallel and Distributed Processing Symposium (IPDPS 2007)*, Long Beach, CA, USA, March 26-30 2007.
- [7] R. Ayd. *The Pablo Self-Defining Data Format*. <ftp://ftp.renci.org/pub/archive/Pablo.Release.5/SDDF/Documentation/SDDF.ps.gz>, 1992. Downloaded on March 7, 2007.
- [8] L. Bongo, O. Anshus, and J. Bjørndalen. Low overhead high performance runtime monitoring of collective communication. In *Proceedings of the 2005 International Conference on Parallel Processing (ICPP)*, Oslo, Norway, pages 455–464, June 14-17 2005.
- [9] Peter N. Brown, Robert D. Falgout, and Jim E. Jones. Semicoarsening multigrid on distributed memory machines. *SIAM J. Sci. Comput.*, 21(5):1823–1834, 2000.
- [10] Laura Carrington, Allan Snaveley, Xiaofeng Gao, and Nicole Wolter. A performance prediction framework for scientific applications. In *Workshop on Performance Modeling - ICCS*, 2003.
- [11] Marc Casas, Rosa M. Badia, and Jesús Labarta. Automatic phase detection of MPI applications. In Christian H. Bischof, H. Martin Bücker, Paul Gibbon, Gerhard R. Joubert, Thomas Lippert, Bernd Mohr, and Frans J. Peters, editors, *PARCO*, volume 15 of *Advances in Parallel Computing*, pages 129–136. IOS Press, 2007.
- [12] Kin-Pong Chan and Ada Wai-Chee Fu. Efficient time series matching by wavelets. In *Data Engineering, 1999. Proceedings., 15th International Conference on*, pages 126–133, Mar 1999.
- [13] I-Hsin Chung, Robert E. Walkup, Hui-Fang Wen, and Hao Yu. MPI performance analysis tools on Blue Gene/L. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing (SC'06)*, page 123, New York, NY, USA, 2006. ACM.

- [14] A. Fagot and J. de Kergommeaux. Systematic assessment of the overhead of tracing parallel programs. In *Proceedings of 4th Euromicro Workshop on Parallel and Distributed Processing*, pages 179–185, 1996.
- [15] Felix Freitag, Julita Corbalan, and Jesus Labarta. A dynamic periodicity detector: Application to speedup computation. In *Proceedings of the 15th International Parallel and Distributed Processing Symposium (IPDPS'01), San Francisco, CA, USA*, April 23-17 2001.
- [16] J. Gailly and M. Alder. *zlib 1.1.4 Manual*. <http://www.zlib.net/manual.html>, March 2002. downloaded on on January 30, 2007.
- [17] Jason Gait. A probe effect in concurrent programs. *Softw. Pract. Exper.*, 16(3):225–233, 1986.
- [18] Todd Gamblin, Bronis de Supinski, Martin Schulz, Rob Fowler, and Daniel Reed. Scalable load-balance measurement for SPMD codes. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, 2008.
- [19] Todd Gamblin, Rob Fowler, and Daniel A. Reed. Scalable methods for monitoring and detecting behavioral equivalence classes in scientific codes. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS'08), Miami, FL*, April 14-28 2008.
- [20] J. Gannon, K. Williams, M. Andersland, J. Lumpp, Jr., and T. Casavant. Using perturbation tracking to compensate for intrusion propagation in message passing systems. In *Proceedings of the 14th International Conference on Distributed Computing Systems, Poznan, Poland*, pages 414–421, June 21-24 1994.
- [21] J. Garlick and C. Dunlap. Building chaos: an operating environment for livermore linux clusters. Technical Report UCRL-ID-151968, Lawrence Livermore National Laboratory, Feb. 2002.
- [22] Michael Gerndt, Bernd Mohr, and Jesper Larsson Träff. A test suite for parallel performance analysis tools. *Concurrency and Computation: Practice and Experience*, 19(11):1465–1480, August 2007.
- [23] Weiming Gu, Greg Eisenhauer, Karsten Schwan, and Jeffrey Vetter. Falcon: on-line monitoring and steering of large-scale parallel programs. *Concurrency: Practice and Experience*, 10(9):699–736, Dec 1998.
- [24] S.T. Hackstadt, A.D. Malony, and B. Mohr. Scalable performance visualization for data-parallel programs. pages 342–349, May 1994.
- [25] Jiawei Han and Micheline Kamber. *Data Mining: Concepts and Techniques*. Morgan Kaufmann, 2005.
- [26] Matthias Hauswirth, Amer Diwan, Peter F. Sweeny, and Michael C. Mozer. Automating vertical profiling. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 281 – 296, October 16-20 2005.
- [27] Michael T. Heath and Jennifer A. Etheridge. Visualizing the performance of parallel programs. volume 8, pages 29–39, Los Alamitos, CA, USA, 1991. IEEE Computer Society Press.
- [28] M.T. Heath, A.D. Malony, and D.T. Rover. The visual display of parallel performance data. *Computer*, 28(11):21–28, Nov 1995.

- [29] Jeff Hollingsworth, Barton Miller, and Jon Cargille. Dynamic program instrumentation for scalable performance tools. In *Proceedings of Scalable High Performance Computing Conference, Knoxville, TN, USA*, pages 841–850, May 23-25 1994.
- [30] Ted Huffmire and Tim Sherwood. Wavelet-based phase classification. In *PACT '06: Proceedings of the 15th international conference on Parallel architectures and compilation techniques*, pages 95–104, New York, NY, USA, 2006. ACM.
- [31] A. Jensen and A. la Cour-Harbo. *Ripples in Mathematics: The Discrete Wavelet Transform*. Springer-Verlag, 2001.
- [32] Laxmikant V. Kalé, Sameer Kumar, Gengbin Zheng, and Chee Wai Lee. Scaling molecular dynamics to 3000 processors with projections: A performance analysis case study. In *International Conference on Computational Science (ICCS 2003), Melbourne, Australia and St. Petersburg, Russia*, pages 23–32, June 2-4 2003.
- [33] A. Knüpfer, R. Brendel, H. Brunst, H. Mix, and W. Nagel. Introducing the open trace format (OTF). In *Proceedings of International Conference on Computational Science (ICCS), Reading, UK*, pages 526–533, May 28-31 2006.
- [34] Andreas Knüpfer. A new data compression technique for event based program traces. In *International Conference on Computational Science*, pages 956–965, 2003.
- [35] Andreas Knüpfer, Bernhard Voigt, Wolfgang E. Nagel, and Hartmut Mix. Visualization of repetitive patterns in event traces. In *PARA*, pages 430–439, 2006.
- [36] Dieter Kranzlmüller, Siegfried Grabner, and Jens Volkert. Event graph visualization for debugging large applications. In *SPDT '96: Proceedings of the SIGMETRICS symposium on Parallel and distributed tools, Philadelphia, Pennsylvania, USA*, pages 108–117, 1996.
- [37] Dieter Kranzlmüller, Siegfried Grabner, and Jens Volkert. Monitoring strategies for hypercube systems. In *PDP '96: Proceedings of the 4th Euromicro Workshop on Parallel and Distributed Processing (PDP '96)*, page 486, Washington, DC, USA, 1996. IEEE Computer Society.
- [38] Dieter Kranzlmüller, Andreas Knüpfer, and Wolfgang E. Nagel. Pattern matching of collective MPI operations. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA '04), Las Vegas, Nevada, USA*, June 21-24 2004.
- [39] Chee Wai Lee, Celso Mendes, and Laxmikant V. Kalé. Towards scalable performance analysis and visualization through data reduction. In *13th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS 2008) held in conjunction with IPDPS 2008*, 2008.
- [40] Allen D. Malony, Dan Reed, and Harry Wijshoff. Performance measurement intrusion and perturbation analysis. *Transactions on Parallel and Distributed Systems*, 3(4):433–450, July 1992.
- [41] Barton P. Miller. What to draw? when to draw?: an essay on parallel program visualization. *J. Parallel Distrib. Comput.*, 18(2):265–269, 1993.
- [42] Barton P. Miller, Mark D. Callaghan, Jonathan M. Cargille, Jeffrey K. Hollingsworth, R. Bruce Irvin, Karen L. Karavanic, Krishna Kunchithapadam, and Tia

- Newhall. The Paradyn parallel performance measurement tool. *Computer*, 28(11):37–46, 1995.
- [43] Kathryn Mohror and Karen L. Karavanic. Performance tool support for MPI-2 on linux. In *SC '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, page 28, Washington, DC, USA, 2004. IEEE Computer Society.
- [44] Kathryn Mohror and Karen L. Karavanic. A study of tracing overhead on a high-performance linux cluster. Technical Report TR-06-06, Portland State University Computer Science Department, December 2006.
- [45] Kathryn Mohror and Karen L. Karavanic. Towards scalable event tracing for high-end systems. In *High Performance Computing and Communications, Third International Conference (HPCC 2007), Houston, Texas, USA*, pages 695–706, September 26-28 2007.
- [46] W. E. Nagel, A. Arnold, M. Weber, H. C. Hoppe, and K. Solchenbach. VAMPIR: Visualization and analysis of MPI resources. *Supercomputer*, 12(1):69–80, 1996.
- [47] Oscar Naím and Anthony J. G. Hey. Visualization of do-loop performance. In *HPCN Europe*, pages 878–887, 1997.
- [48] O. Nickolayev, P. Roth, and D. Reed. Real-time statistical clustering for event trace reduction. *International Journal of High Performance Computing Applications*, 11(2):69–80, 1997.
- [49] Michael Noeth, Frank Mueller, Martin Schulz, and Bronis R. de Supinski. Scalable compression and replay of communication traces in massively parallel environments. In *21th International Parallel and Distributed Processing Symposium (IPDPS'07)*, March 2007.
- [50] D. Ogle, K. Schwan, and R. Snodgrass. Application-dependent dynamic monitoring of distributed and parallel systems. *IEEE Transactions on Parallel and Distributed Systems*, 4(7):762–778, July 1993.
- [51] Fabrizio Petrini, Darren J. Kerbyson, and Scott Pakin. The case of the missing supercomputer performance: Achieving optimal performance on the 8,192 processors of ASCI Q. In *Proceedings of the 2003 ACM/IEEE conference on Supercomputing (SC'03), Phoenix, Arizona, USA*, page 55, November 15-21 2003.
- [52] V. Pillet, J. Labarta, T. Cortes, and S. Girona. PARAVÉR: A tool to visualise and analyze parallel code. In *Proceedings of WoTUG-18: Transputer and occam Developments*, volume 44, pages 17–31, Amsterdam, 1995. IOS Press.
- [53] Prasun Ratn, Frank Mueller, Bronis R. de Supinski, and Martin Schulz. Preserving time in large-scale communication traces. In *ICS '08: Proceedings of the 22nd annual international conference on Supercomputing*, pages 46–55, New York, NY, USA, 2008. ACM.
- [54] D. Reed, R. Olson, R. Aydt, T. Madhyastha, T. Birkett, D. Jensen, B. Nazief, and B. Totty. Scalable performance environments for parallel systems. In *Proceedings of the 6th Distributed Memory Computing Conference*, pages 562–569, May 28 - April 1 1991.
- [55] D. Reed, P. Roth, R. Aydt, K. Shields, L. Tavera, R. Noe, and B. Schwartz. Scalable performance analysis: the pablo performance analysis environment. In

Proceedings of the Scalable Parallel Libraries Conference, Mississippi State, MS, USA, pages 104–113, October 6-8 1993.

[56] Philip C. Roth, Dorian C. Arnold, and Barton P. Miller. MRNet: A software-based multicast/reduction network for scalable tools. In *SC '03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, page 21, Washington, DC, USA, 2003. IEEE Computer Society.

[57] S. Sarukkai and A. Malony. Perturbation analysis of high level instrumentation for spmd programs. In *Proceedings of the 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, San Diego, CA*, pages 44–53, 1993.

[58] Sameer Shende and Allen D. Malony. The tau parallel performance system. *International Journal of High Performance Computing Applications*, 20(2):287–311, 2006.

[59] F. Song, F. Wolf, N. Bhatia, J. Dongarra, and S. Moore. An algebra for cross-experiment performance analysis. In *Proc. of the International Conference on Parallel Processing (ICPP)*, pages 63–72, Montreal, Canada, August 2004. IEEE Society.

[60] Daniel P. Spooner and Darren J. Kerbyson. Performance feature identification by comparative trace analysis. *Future Generation Comp. Syst.*, 22(3):369–380, 2006.

[61] Cluster File Systems. Lustre: A scalable, high-performance file system. <http://www.lustre.org/docs/whitepaper.pdf>, November 2002. downloaded June 2006.

[62] Eno Thereska, Brandon Salmon, John D. Strunk, Matthew Wachs, Michael Abd-El-Malek, Julio Lopez, and Gregory R. Ganger. Stardust: tracking activity in a distributed storage system. In *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems, SIGMETRICS 2006, Saint Malo, France*, pages 3–14, June 26-30 2006.

[63] Using cray performance analysis tools. <http://docs.cray.com/books/S-2376-31/S-2376-31.pdf>, October 2006. Downloaded on Feb. 23, 2007.

[64] Jeffrey Vetter. Performance analysis of distributed applications using automatic classification of communication inefficiencies. In *ICS '00: Proceedings of the 14th international conference on Supercomputing*, pages 245–254, New York, NY, USA, 2000. ACM.

[65] Jeffrey Vetter. Dynamic statistical profiling of communication activity in distributed applications. In *Proceedings of ACM SIGMETRICS 2002 International Conference on Measurement and Modeling of Computer Systems, Marina Del Rey, CA, USA*, pages 240–250, June 15-19 2002.

[66] A. Waheed, V. F. Melfi, and D. T. Rover. A model for instrumentation system management in concurrent computer systems. In *HICSS '95: Proceedings of the 28th Hawaii International Conference on System Sciences*, page 432, Washington, DC, USA, 1995. IEEE Computer Society.

[67] A. Waheed, D. Rover, and J. Hollingsworth. Modeling and evaluating design alternatives for an on-line instrumentation system: A case study. *IEEE Transactions on Software Engineering*, 24(6):451–470, June 1998.

[68] James S. Walker. *A Primer on Wavelets and Their Scientific Applications*. Chapman & Hall/CRC, 2008.

- [69] K. Williams, M. Andersland, J. Gannon, J. Lumpp, Jr., and T. Casavant. Perturbation tracking. In *Proceedings of the 32nd IEEE Conference on Decision and Control, San Antonio, TX*, pages 299–316, 1996.
- [70] C. Winstead, H. Pritchard, and V. McKoy. *Tuning I/O Performance on the Paragon: Fun with Pablo and Norma*. IEEE Computer Society Press, 1996.
- [71] F. Wolf, B. Mohr, J. Dongarra, and S. Moore. Automatic analysis of inefficiency patterns in parallel applications. *Concurrency and Computation: Practice and Experience*, 19:1481–1496, 2007.
- [72] Felix Wolf, Allan Malony, Sameer Shende, and Alan Morris. Trace-based parallel performance overhead compensation. In *Proceedings of the International Conference on High Performance Computing and Communications (HPCC), Sorrento, Italy*, September 2005.
- [73] C. Eric Wu, Anthony Bolmarcich, Marc Snir, David Wootton, Farid Parpia, Anthony Chan, Ewing Lusk, and William Gropp. From trace generation to visualization: a performance framework for distributed parallel systems. In *Supercomputing '00: Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM)*, page 50, Washington, DC, USA, 2000. IEEE Computer Society.
- [74] C. Eric Wu, Hubertus Franke, and Yew-Huey Liu. A unified trace environment for ibm sp systems. *IEEE Parallel Distrib. Technol.*, 4(2):89–93, 1996.
- [75] K. Yaghmour and D. Dagenais. Measuring and characterizing system behavior using kernel-level event logging. In *Proceedings of the USENIX Annual 2000 Technical Conference, San Diego, CA, USA*, pages 13–26, June 2000.
- [76] J. Yan and S. Listgarten. Intrusion compensation for performance evaluation of parallel programs on a multicomputer. In *Proceedings of the 6th International Conference on Parallel and Distributed Systems, Louisville, KY*, October 14-16 1993.
- [77] Jerry C. Yan, Haoqiang H. Jin, and Melisa A. Schmidt. Performance data gathering and representation from fixed-size statistical data. Technical Report NAS-98-003, NASA Ames Research Center, 1998.
- [78] Jerry C. Yan and Sekhar R. Sarukkai. Analyzing parallel program performance using normalized performance indices and trace transformation techniques. *Parallel Computing*, 22(9):1215–1237, 1996.
- [79] Jerry C. Yan and Melisa Schmidt. Constructing space-time views from fixed size trace files – getting the best of both worlds. In *Parallel Computing: Fundamentals, Applications and New Directions, Proceedings of the Conference (ParCo'97), Bonn, Germany*, pages 633–640, September 19-22 1997.
- [80] Omer Zaki, Ewing Lusk, William Gropp, and Deborah Swider. Toward scalable performance visualization with Jumpshot. *The International Journal of High Performance Computing Applications*, 13(3):277–288, Fall 1999.

Appendix: Additional Trace Similarity Study Results

List of Figures

Fig. 1 Intra-process Reduction: File Size and Approximation Distance for Varying Duration Thresholds and Relative Distance	133
Fig. 2 Intra-process Reduction: File Size and Approximation Distance for Varying Threshold and Absolute Distance.....	134
Fig. 3 Intra-process Reduction: File Size and Approximation Distance for Varying Threshold and Manhattan Distance	135
Fig. 4 Intra-process Reduction: File Size and Approximation Distance for Varying Threshold and Euclidean Distance	136
Fig. 5 Intra-process Reduction: File Size and Approximation Distance for Varying Threshold and Chebyshev Distance.....	137
Fig. 6 Intra-process Reduction: File Size and Approximation Distance for Varying Threshold and Keep k Iterations.....	138
Fig. 7 Intra-process Reduction: File Size and Approximation Distance for Varying Threshold and Average Wavelet Transform.....	139
Fig. 8 Intra-process Reduction: File Size and Approximation Distance for Varying Threshold and Haar Wavelet Transform	140
Fig. 9 Intra-process Reduction: File Size and Approximation Distance for Varying Thresholds for Sweep3d and relDiff, absDiff, Manhattan	141
Fig. 10 Intra-process Reduction: File Size and Approximation Distance for Varying Thresholds for Sweep3d and Euclidean, Chebyshev, iter_k	142
Fig. 11 Intra-process Reduction: File Size and Approximation Distance for Varying Thresholds for Sweep3d and Wavelet Transforms.....	143
Fig. 12 Intra-process Reduction: Retention of Performance Trends with Varying Thresholds for dyn_load_balance.....	144
Fig. 13 Intra-process Reduction: Retention of Performance Trends with Varying Thresholds for early_gather.....	145
Fig. 14 Intra-process Reduction: Retention of Performance Trends with Varying Threshold for imbalance_at_mpi_barrier	146
Fig. 15 Intra-process Reduction: Retention of Performance Trends with Varying Threshold for late_broadcast	147
Fig. 16 Intra-process Reduction: Retention and Performance Trends with Varying Thresholds for late_receiver	148
Fig. 17 Intra-process Reduction: Retention of Performance Trends with Varying Thresholds for late_sender.....	149
Fig. 18 Intra-process Reduction: Retention of Performance Trends with Varying Thresholds for Nto1_32.....	150

Fig. 19 Intra-process Reduction: Retention of Performance Trends with Varying Thresholds for NtoN_32	151
Fig. 20 Intra-process Reduction: Retention of Performance Trends with Varying Thresholds for 1toN_32	152
Fig. 21 Intra-process Reduction: Retention of Performance Trends with Varying Thresholds for 1to1r_32	153
Fig. 22 Intra-process Reduction: Retention of Performance Trends with Varying Thresholds for 1to1s_32	154
Fig. 23 Intra-process Reduction: Retention of Performance Trends with Varying Thresholds for Nto1_1024	155
Fig. 24 Intra-process Reduction: Retention of Performance Trends with Varying Thresholds for NtoN_1024	156
Fig. 25 Intra-process Reduction: Retention of Performance Trends with Varying Thresholds for 1toN_1024	157
Fig. 26 Intra-process Reduction: Retention of Performance Trends with Varying Thresholds for 1to1r_1024	158
Fig. 27 Intra-process Reduction: Retention of Performance Trends with Varying Thresholds for 1to1s_1024	159
Fig. 28 Intra-process Reduction: Retention of Performance Trends with Varying Thresholds for sweep3d_8p	160
Fig. 29 Intra-process Reduction: Retention of Performance Trends with Varying Thresholds for sweep3d_32p	161
Fig. 30 Inter-process Reduction: File Size and Approximation Distance for Varying Duration Thresholds and Relative Distance	162
Fig. 31 Inter-process Reduction: File Size and Approximation Distance for Varying Duration Thresholds and Absolute Distance	163
Fig. 32 Inter-process Reduction: File Size and Approximation Distance for Varying Duration Thresholds and Manhattan Distance	164
Fig. 33 Inter-process Reduction: File Size and Approximation Distance for Varying Duration Thresholds and Euclidean Distance	165
Fig. 34 Inter-process Reduction: File Size and Approximation Distance for Varying Duration Thresholds and Chebyshev Distance	166
Fig. 35 Inter-process Reduction: File Size and Approximation Distance for Varying Duration Thresholds and Average Wavelet	167
Fig. 36 Inter-process Reduction: File Size and Approximation Distance for Varying Duration Thresholds and Haar Wavelet	168
Fig. 37 Inter-process Reduction: File Size and Approximation Distance for Varying Thresholds for Sweep3d and relDiff, absDiff, and Manhattan	169

Fig. 38 Inter-process Reduction: File Size and Approximation Distance for Varying Thresholds for Sweep3d and Euclidean and Chebyshev.....	170
Fig. 39 Inter-process Reduction: File Size and Approximation Distance for Varying Thresholds for Sweep3d and avgWave and haarWave	171
Fig. 40 Inter-process Reduction: Retention of Performance Trends with Varying Thresholds for dyn_load_balance.....	172
Fig. 41 Inter-process Reduction: Retention of Performance Trends with Varying Thresholds for early_gather.....	173
Fig. 42 Inter-process Reduction: Retention of Performance Trends with Varying Thresholds for imbalance_at_barrier.....	174
Fig. 43 Inter-process Reduction: Retention of Performance Trends with Varying Thresholds for late_broadcast.....	175
Fig. 44 Inter-process Reduction: Retention of Performance Trends with Varying Thresholds for late_receiver	176
Fig. 45 Inter-process Reduction: Retention of Performance Trends with Varying Thresholds for late_sender.....	177
Fig. 46 Inter-process Reduction: Retention of Performance Trends with Varying Thresholds for Nto1_32.....	178
Fig. 47 Inter-process Reduction: Retention of Performance Trends with Varying Thresholds for NtoN_32.....	179
Fig. 48 Inter-process Reduction: Retention of Performance Trends with Varying Thresholds for 1toN_32.....	180
Fig. 49 Inter-process Reduction: Retention of Performance Trends with Varying Thresholds for 1to1r_32.....	181
Fig. 50 Inter-process Reduction: Retention of Performance Trends with Varying Thresholds for 1to1s_32	182
Fig. 51 Inter-process Reduction: Retention of Performance Trends with Varying Thresholds for Nto1_1024.....	183
Fig. 52 Inter-process Reduction: Retention of Performance Trends with Varying Thresholds for NtoN_1024.....	184
Fig. 53 Inter-process Reduction: Retention of Performance Trends with Varying Thresholds for 1toN_1024.....	185
Fig. 54 Inter-process Reduction: Retention of Performance Trends with Varying Thresholds for 1to1r_1024	186
Fig. 55 Inter-process Reduction: Retention of Performance Trends with Varying Thresholds for 1to1s_1024.....	187
Fig. 56 Inter-process Reduction: Retention of Performance Trends with Varying Thresholds for sweep3d_8p.....	188

Fig. 57 Inter-process Reduction: Retention of Performance Trends with Varying Thresholds for sweep3d_32p.....	189
Fig. 58 Combined Reduction: Retention of Performance Trends with Default Thresholds for dyn_load_balance.....	190
Fig. 59 Combined Reduction: Retention of Performance Trends with Default Thresholds for early_gather.....	190
Fig. 60 Combined Reduction: Retention of Performance Trends with Default Thresholds for imbalance_at_barrier.....	190
Fig. 61 Combined Reduction: Retention of Performance Trends with Default Thresholds for late_broadcast.....	191
Fig. 62 Combined Reduction: Retention of Performance Trends with Default Thresholds for late_receiver.....	191
Fig. 63 Combined Reduction: Retention of Performance Trends with Default Thresholds for late_sender.....	191
Fig. 64 Combined Reduction: Retention of Performance Trends with Default Thresholds for Nto1_32.....	192
Fig. 65 Combined Reduction: Retention of Performance Trends with Default Thresholds for NtoN_32.....	192
Fig. 66 Combined Reduction: Retention of Performance Trends with Default Thresholds for 1toN_32.....	192
Fig. 67 Combined Reduction: Retention of Performance Trends with Default Thresholds for 1to1r_32.....	193
Fig. 68 Combined Reduction: Retention of Performance Trends with Default Thresholds for 1to1s_32.....	193
Fig. 69 Combined Reduction: Retention of Performance Trends with Default Thresholds for Nto1_1024.....	193
Fig. 70 Combined Reduction: Retention of Performance Trends with Default Thresholds for NtoN_1024.....	194
Fig. 71 Combined Reduction: Retention of Performance Trends with Default Thresholds for 1toN_1024.....	194
Fig. 72 Combined Reduction: Retention of Performance Trends with Default Thresholds for 1to1r_1024.....	194
Fig. 73 Combined Reduction: Retention of Performance Trends with Default Thresholds for 1to1s_1024.....	195
Fig. 74 Combined Reduction: Retention of Performance Trends with Default Thresholds for sweep3d_8p.....	195
Fig. 75 Combined Reduction: Retention of Performance Trends with Default Thresholds for sweep3d_32p.....	195

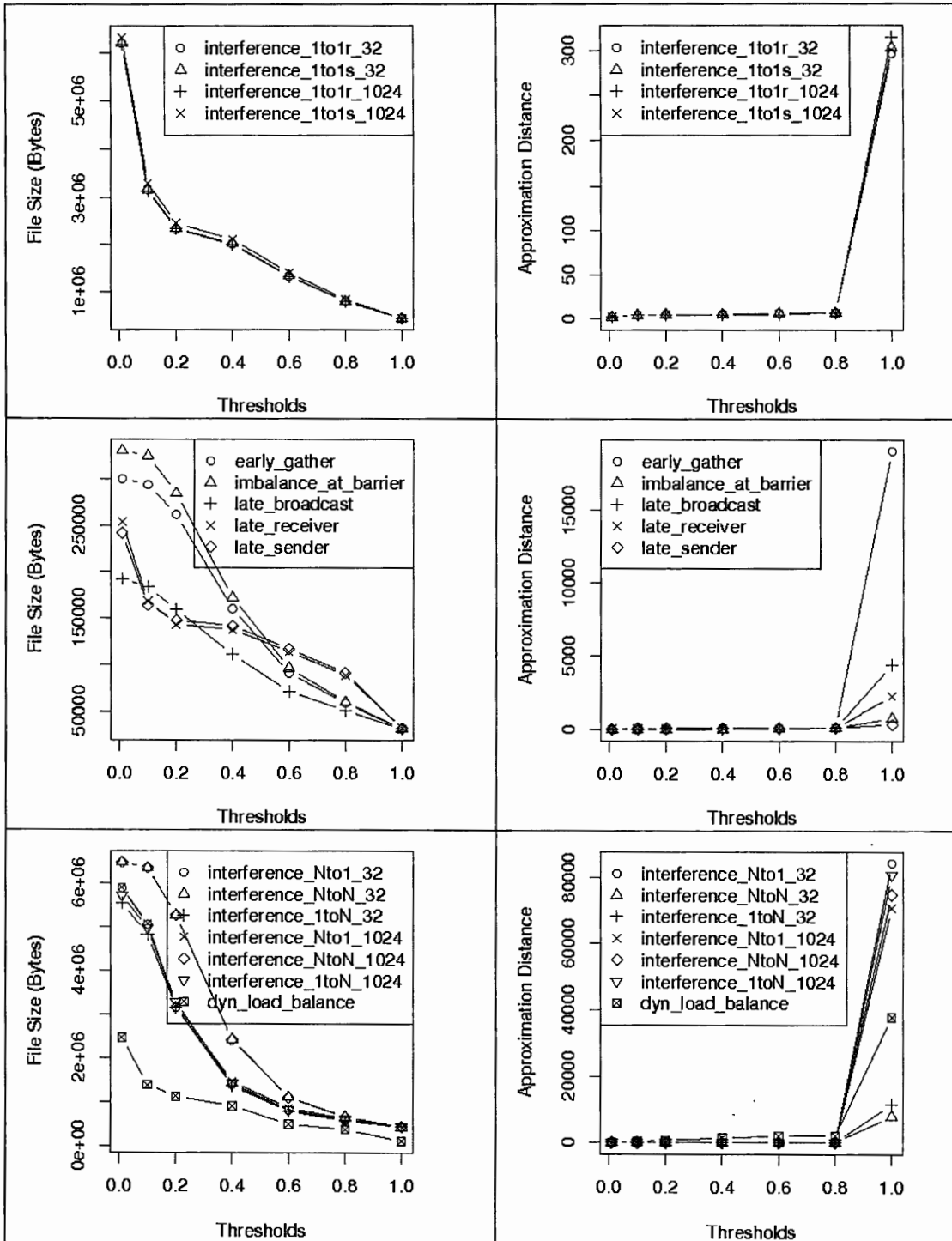


Fig. 1 Intra-process Reduction: File Size and Approximation Distance for Varying Duration Thresholds and Relative Distance

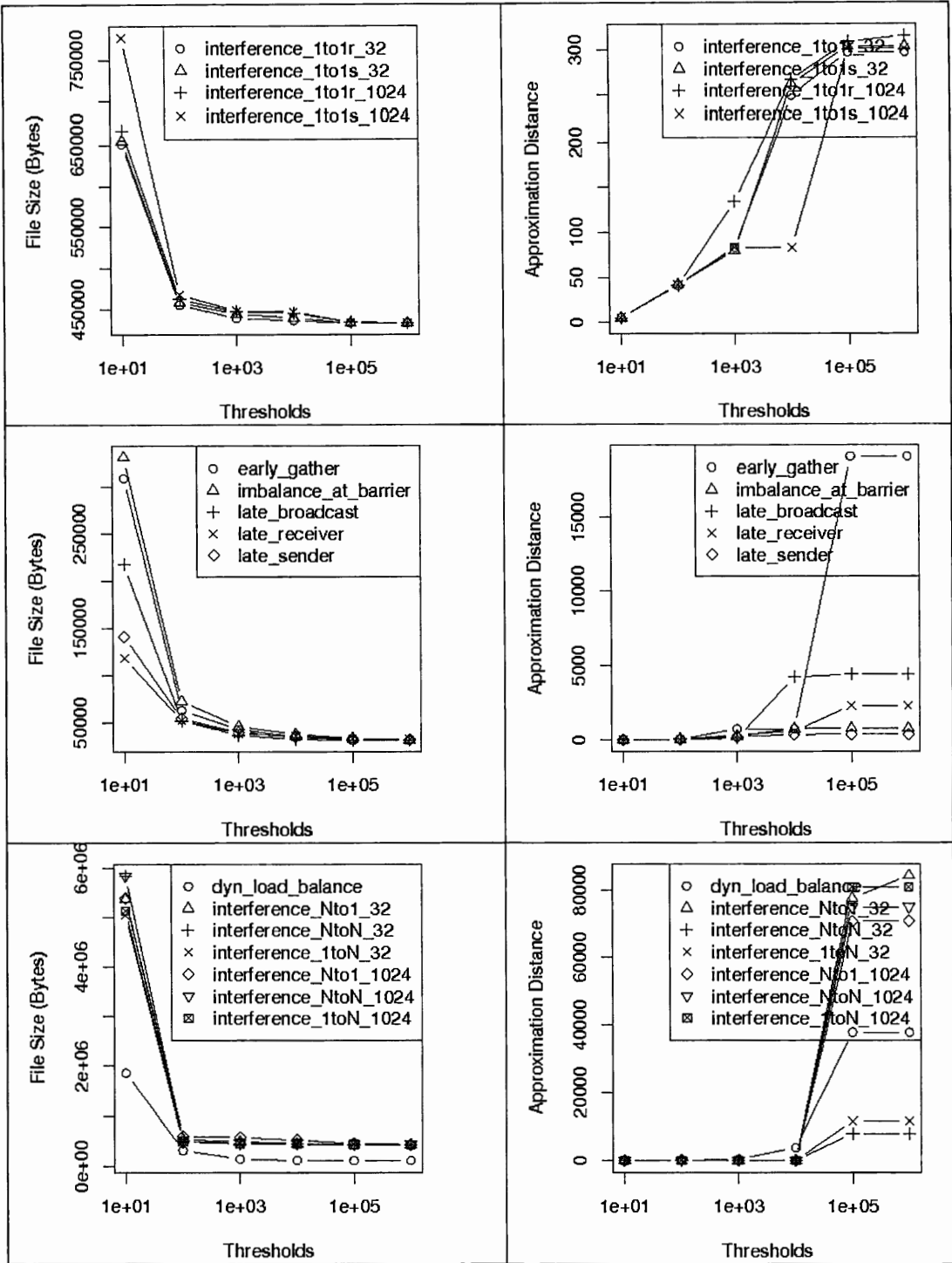


Fig. 2 Intra-process Reduction: File Size and Approximation Distance for Varying Threshold and Absolute Distance

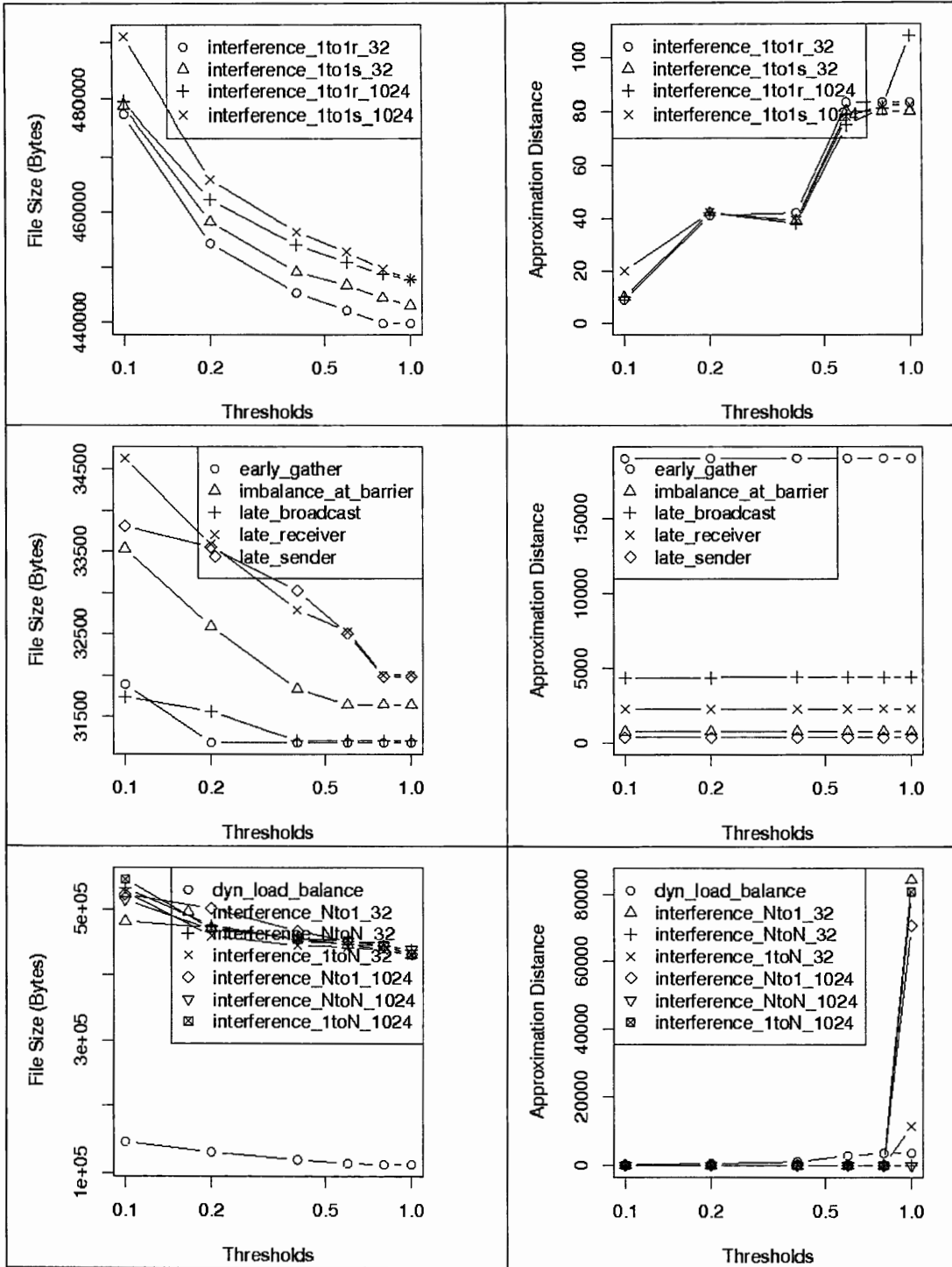


Fig. 3 Intra-process Reduction: File Size and Approximation Distance for Varying Threshold and Manhattan Distance

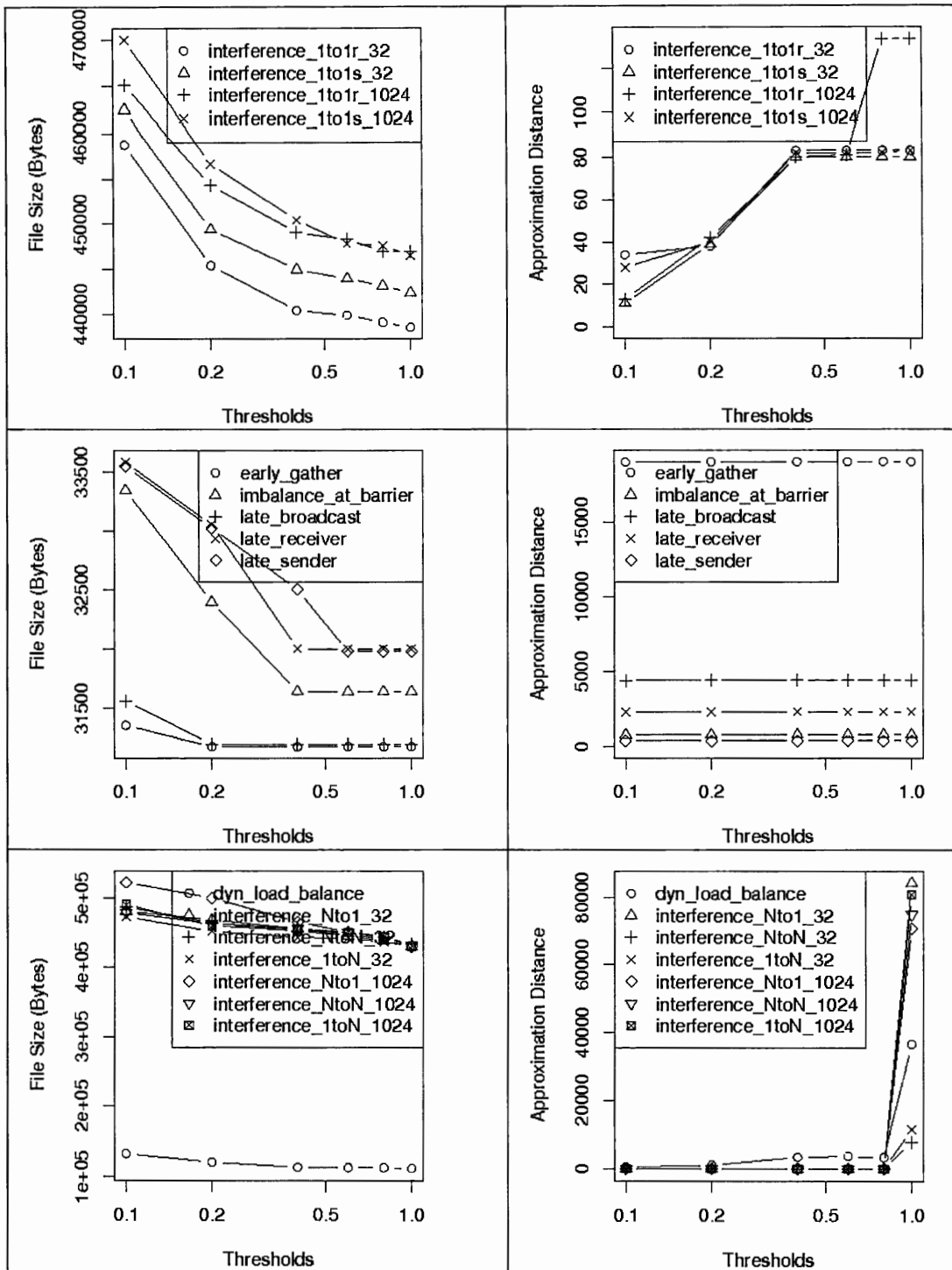


Fig. 4 Intra-process Reduction: File Size and Approximation Distance for Varying Threshold and Euclidean Distance

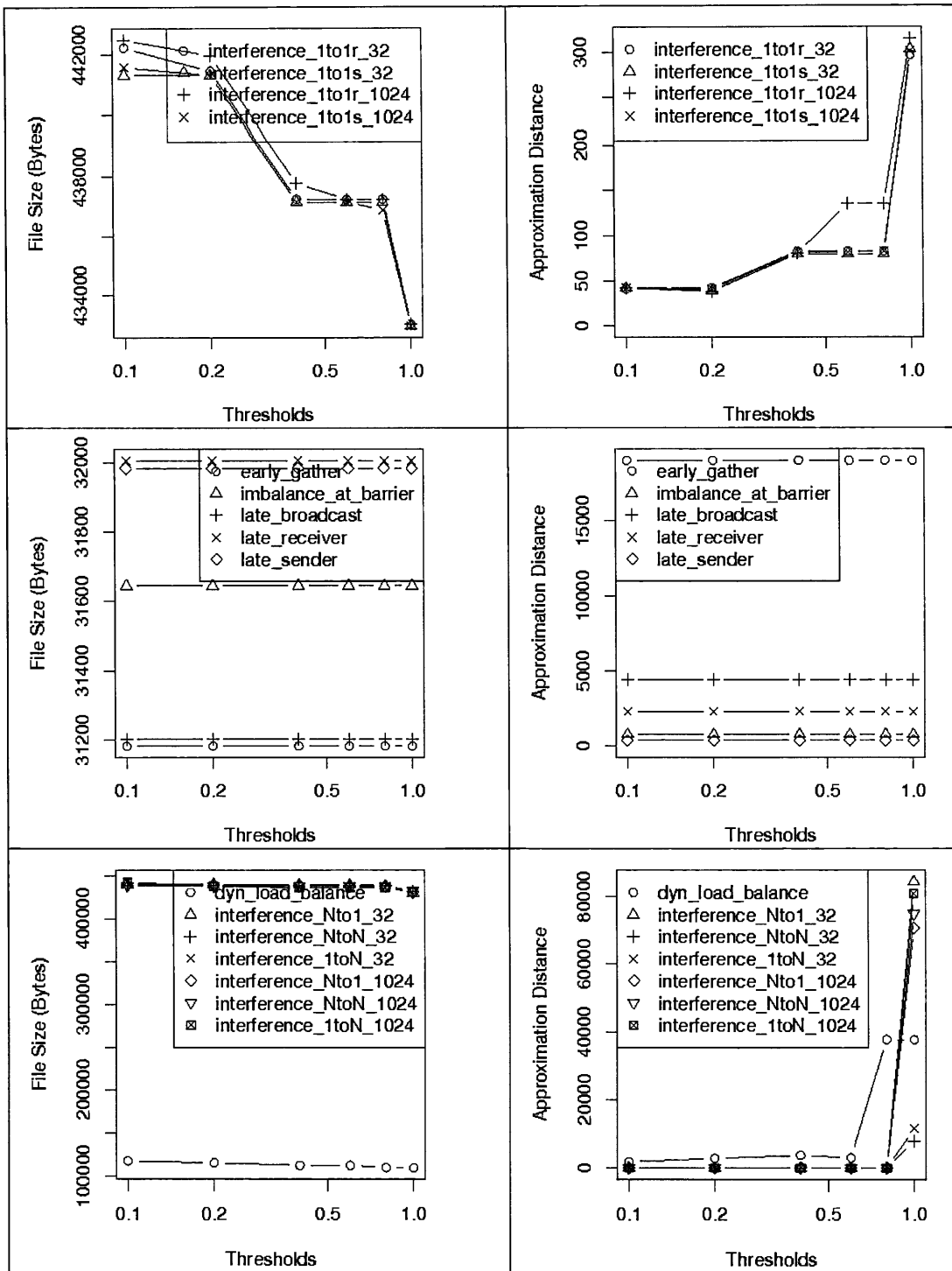


Fig. 5 Intra-process Reduction: File Size and Approximation Distance for Varying Threshold and Chebyshev Distance

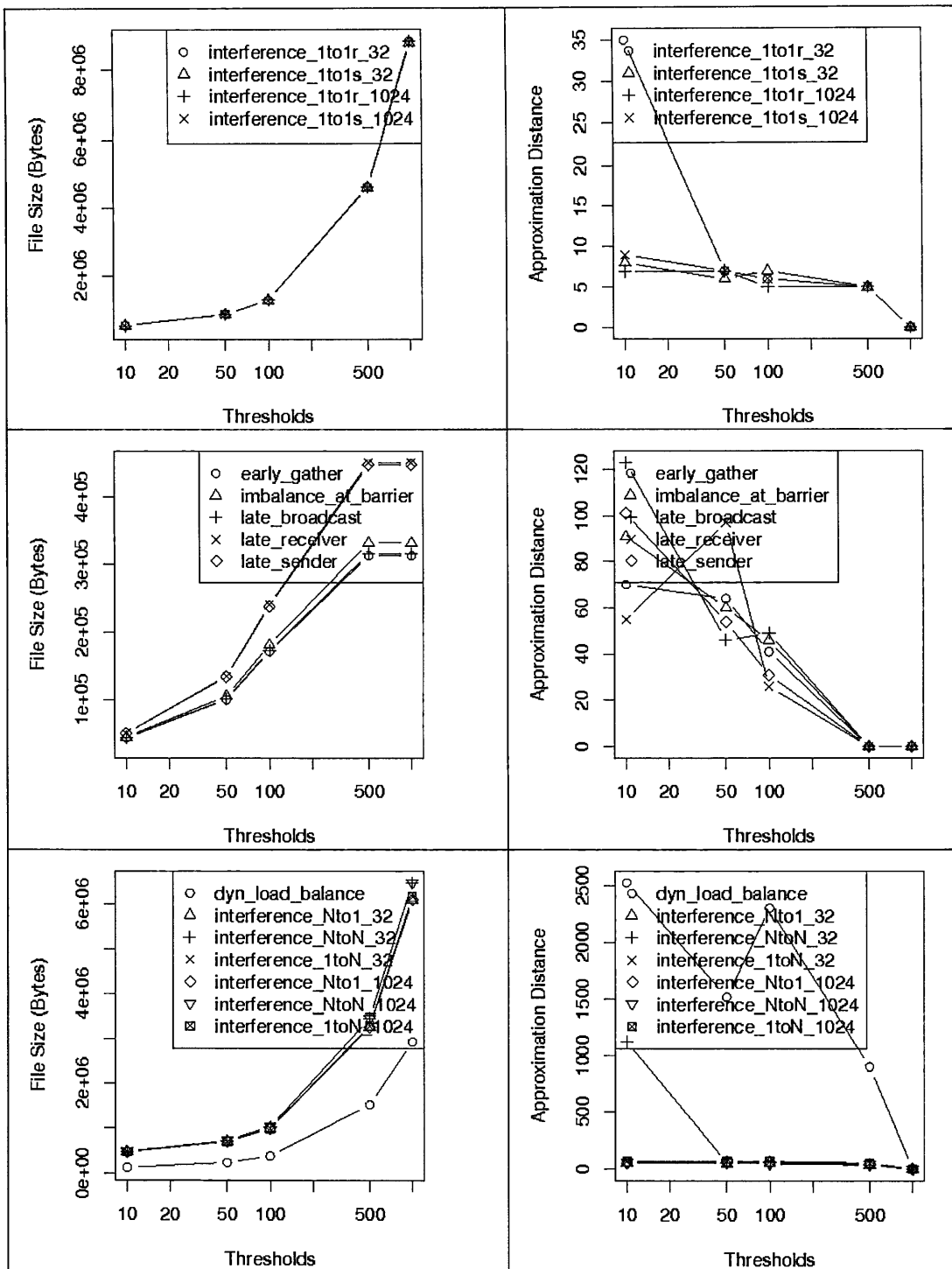


Fig. 6 Intra-process Reduction: File Size and Approximation Distance for Varying Threshold and Keep k Iterations

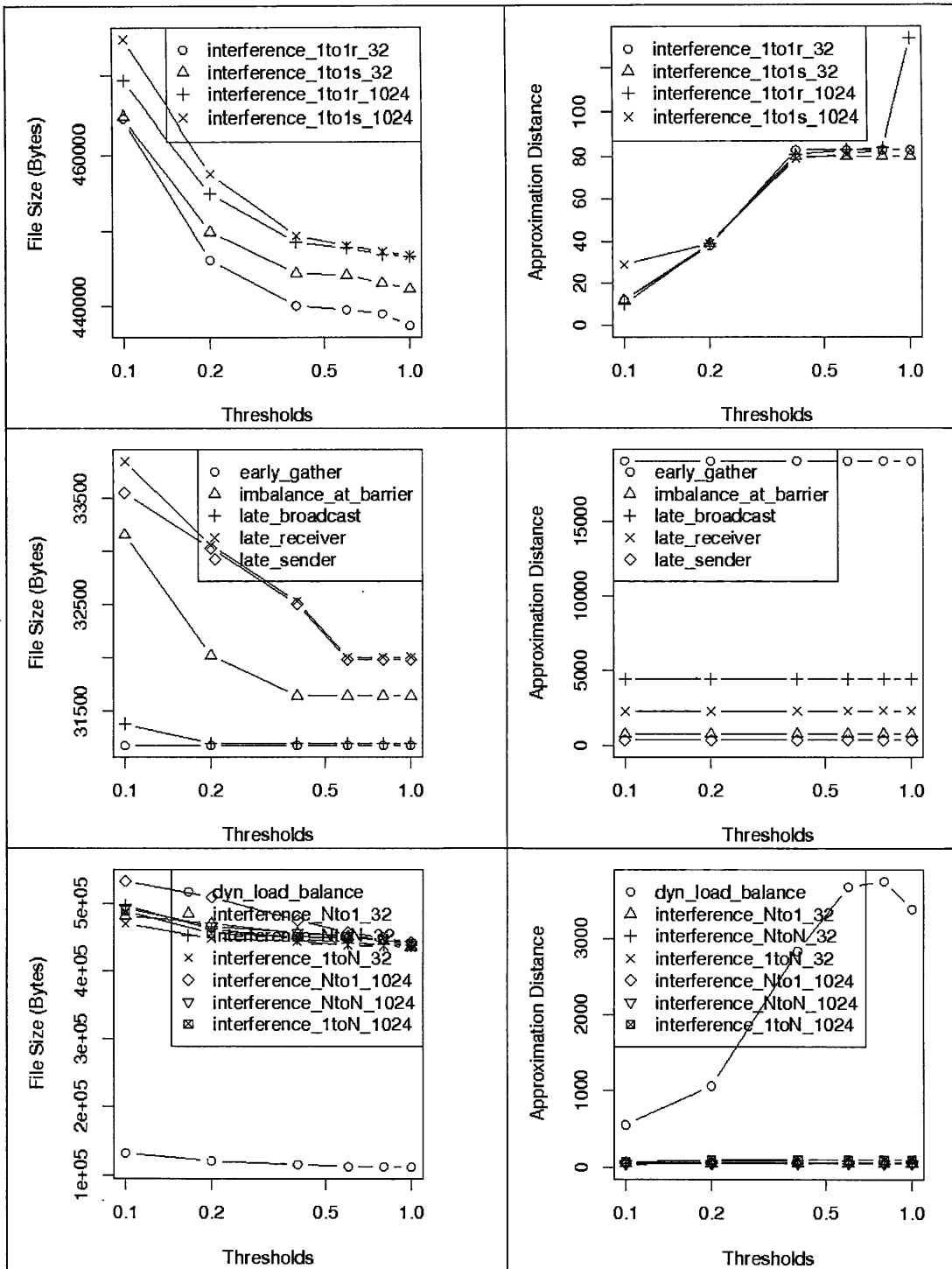


Fig. 7 Intra-process Reduction: File Size and Approximation Distance for Varying Threshold and Average Wavelet Transform

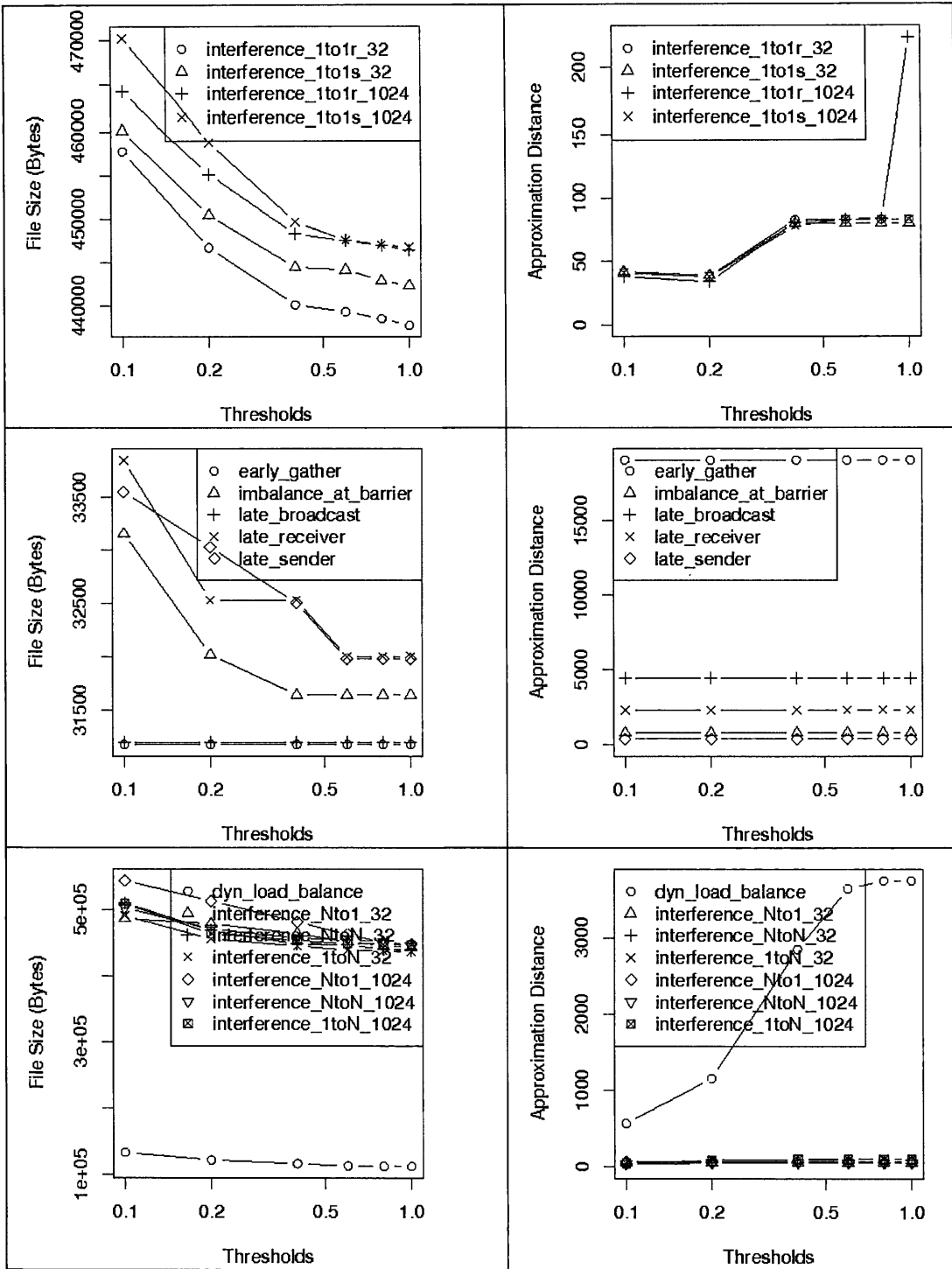


Fig. 8 Intra-process Reduction: File Size and Approximation Distance for Varying Threshold and Haar Wavelet Transform

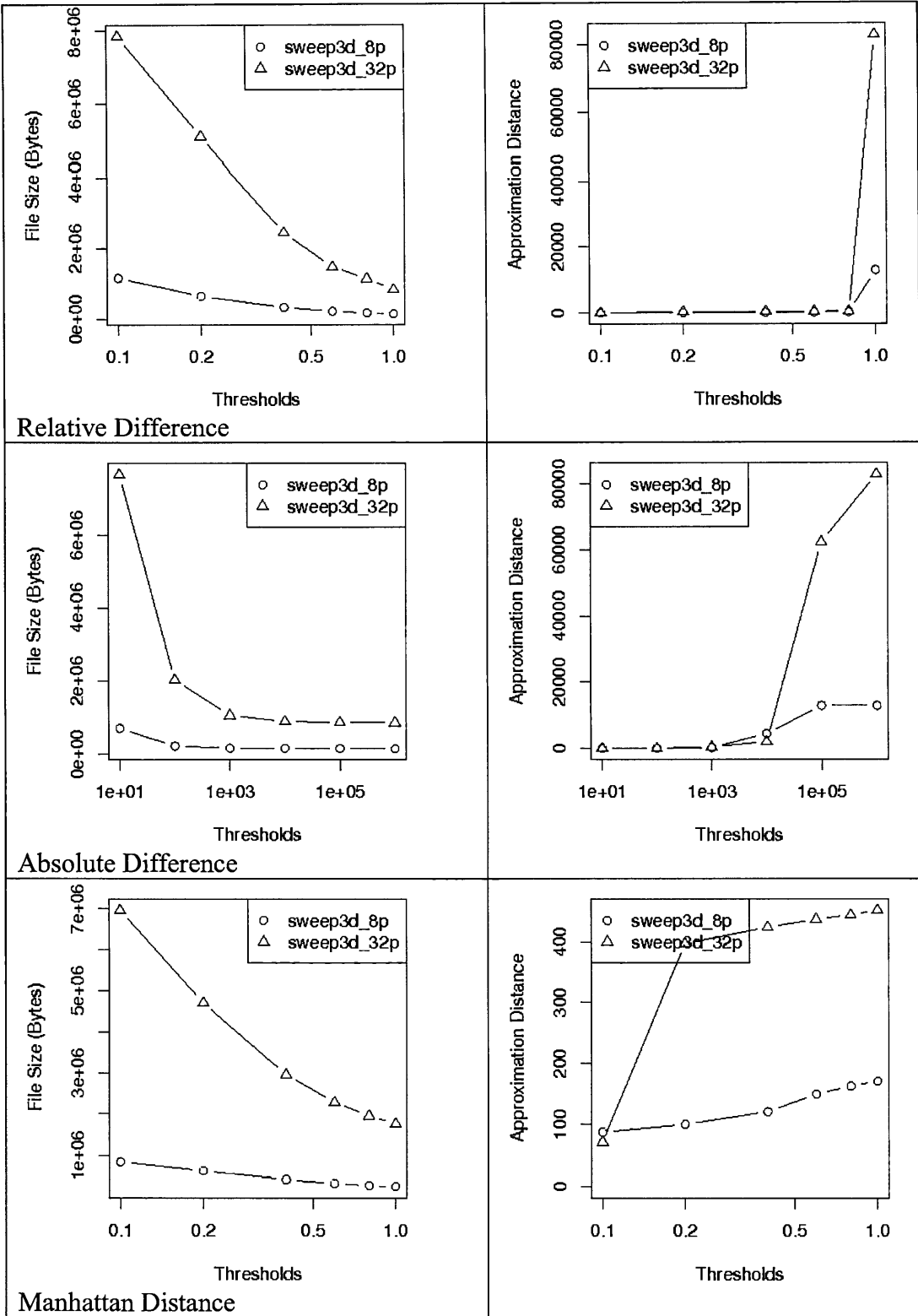


Fig. 9 Intra-process Reduction: File Size and Approximation Distance for Varying Thresholds for Sweep3d and relDiff, absDiff, Manhattan

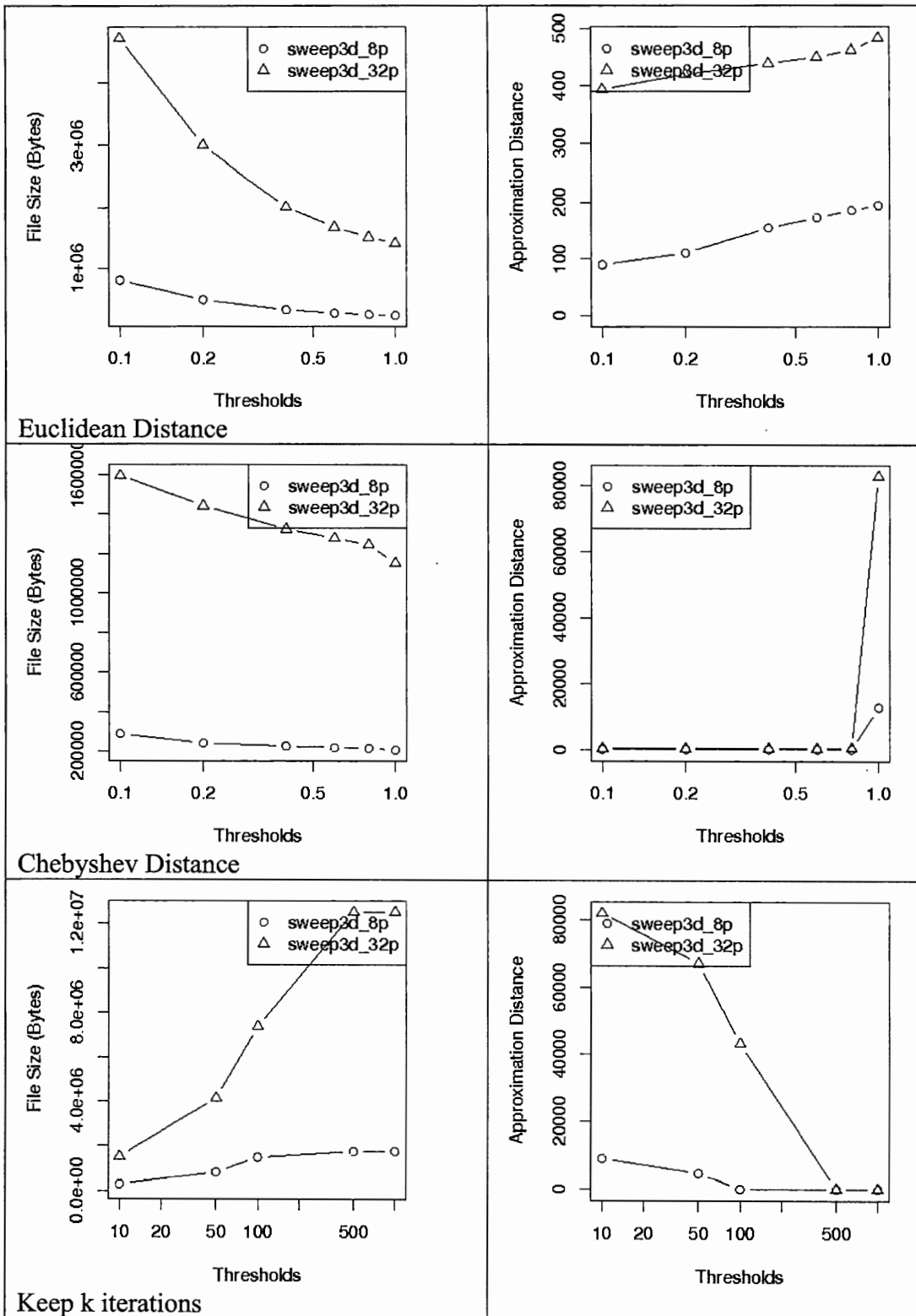


Fig. 10 Intra-process Reduction: File Size and Approximation Distance for Varying Thresholds for Sweep3d and Euclidean, Chebyshev, iter_k

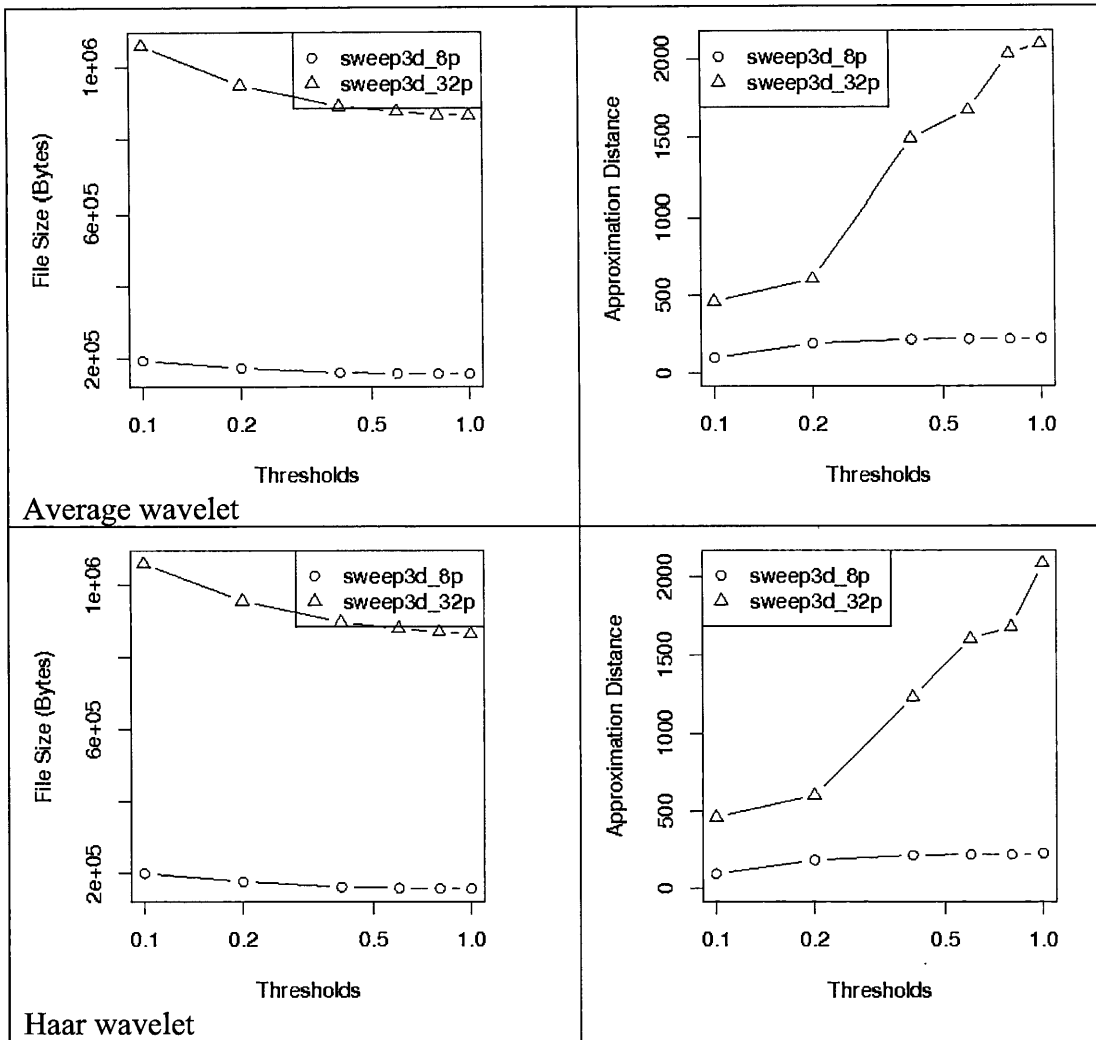


Fig. 11 Intra-process Reduction: File Size and Approximation Distance for Varying Thresholds for Sweep3d and Wavelet Transforms

		MPI Alltoall					do work
relative difference	no loss	EX	MP	CM	CO	NN	EX
	0.1	EX	MP	CM	CO	NN	EX
	0.2	EX	MP	CM	CO	NN	EX
	0.4	EX	MP	CM	CO	NN	EX
	0.6	EX	MP	CM	CO	NN	EX
	0.8	EX	MP	CM	CO	NN	EX
absolute difference	10	EX	MP	CM	CO	NN	EX
	100	EX	MP	CM	CO	NN	EX
	1000	EX	MP	CM	CO	NN	EX
	10000	EX	MP	CM	CO	NN	EX
	100000	EX	MP	CM	CO	NN	EX
	1000000	EX	MP	CM	CO	NN	EX
Manhattan distance	0.1	EX	MP	CM	CO	NN	EX
	0.2	EX	MP	CM	CO	NN	EX
	0.4	EX	MP	CM	CO	NN	EX
	0.6	EX	MP	CM	CO	NN	EX
	0.8	EX	MP	CM	CO	NN	EX
	1.0	EX	MP	CM	CO	NN	EX
Euclidean distance	0.1	EX	MP	CM	CO	NN	EX
	0.2	EX	MP	CM	CO	NN	EX
	0.4	EX	MP	CM	CO	NN	EX
	0.6	EX	MP	CM	CO	NN	EX
	0.8	EX	MP	CM	CO	NN	EX
	1.0	EX	MP	CM	CO	NN	EX
Chebyshev distance	0.1	EX	MP	CM	CO	NN	EX
	0.2	EX	MP	CM	CO	NN	EX
	0.4	EX	MP	CM	CO	NN	EX
	0.6	EX	MP	CM	CO	NN	EX
	0.8	EX	MP	CM	CO	NN	EX
	1.0	EX	MP	CM	CO	NN	EX
Average Wavelet	0.1	EX	MP	CM	CO	NN	EX
	0.2	EX	MP	CM	CO	NN	EX
	0.4	EX	MP	CM	CO	NN	EX
	0.6	EX	MP	CM	CO	NN	EX
	0.8	EX	MP	CM	CO	NN	EX
	1.0	EX	MP	CM	CO	NN	EX
Haar Wavelet	0.1	EX	MP	CM	CO	NN	EX
	0.2	EX	MP	CM	CO	NN	EX
	0.4	EX	MP	CM	CO	NN	EX
	0.6	EX	MP	CM	CO	NN	EX
	0.8	EX	MP	CM	CO	NN	EX
	1.0	EX	MP	CM	CO	NN	EX
Keep k iterations	500	EX	MP	CM	CO	NN	EX
	100	EX	MP	CM	CO	NN	EX
	50	EX	MP	CM	CO	NN	EX
	10	EX	MP	CM	CO	NN	EX
	1	EX	MP	CM	CO	NN	EX
	average	EX	MP	CM	CO	NN	EX

Fig. 12 Intra-process Reduction: Retention of Performance Trends with Varying Thresholds for dyn_load_balance

		MPI Gather						do work
	no loss	EX	MP	CM	CO	ER	EX	
relative difference	0.1	EX	MP	CM	CO	ER	EX	
	0.2	EX	MP	CM	CO	ER	EX	
	0.4	EX	MP	CM	CO	ER	EX	
	0.6	EX	MP	CM	CO	ER	EX	
	0.8	EX	MP	CM	CO	ER	EX	
	1.0	EX	MP	CM	CO	ER	EX	
absolute difference	10	EX	MP	CM	CO	ER	EX	
	100	EX	MP	CM	CO	ER	EX	
	1000	EX	MP	CM	CO	ER	EX	
	10000	EX	MP	CM	CO	ER	EX	
	100000	EX	MP	CM	CO	ER	EX	
	1000000	EX	MP	CM	CO	ER	EX	
Manhattan distance	0.1	EX	MP	CM	CO	ER	EX	
	0.2	EX	MP	CM	CO	ER	EX	
	0.4	EX	MP	CM	CO	ER	EX	
	0.6	EX	MP	CM	CO	ER	EX	
	0.8	EX	MP	CM	CO	ER	EX	
	1.0	EX	MP	CM	CO	ER	EX	
Euclidean distance	0.1	EX	MP	CM	CO	ER	EX	
	0.2	EX	MP	CM	CO	ER	EX	
	0.4	EX	MP	CM	CO	ER	EX	
	0.6	EX	MP	CM	CO	ER	EX	
	0.8	EX	MP	CM	CO	ER	EX	
	1.0	EX	MP	CM	CO	ER	EX	
Chebyshev distance	0.1	EX	MP	CM	CO	ER	EX	
	0.2	EX	MP	CM	CO	ER	EX	
	0.4	EX	MP	CM	CO	ER	EX	
	0.6	EX	MP	CM	CO	ER	EX	
	0.8	EX	MP	CM	CO	ER	EX	
	1.0	EX	MP	CM	CO	ER	EX	
Average Wavelet	0.1	EX	MP	CM	CO	ER	EX	
	0.2	EX	MP	CM	CO	ER	EX	
	0.4	EX	MP	CM	CO	ER	EX	
	0.6	EX	MP	CM	CO	ER	EX	
	0.8	EX	MP	CM	CO	ER	EX	
	1.0	EX	MP	CM	CO	ER	EX	
Haar Wavelet	0.1	EX	MP	CM	CO	ER	EX	
	0.2	EX	MP	CM	CO	ER	EX	
	0.4	EX	MP	CM	CO	ER	EX	
	0.6	EX	MP	CM	CO	ER	EX	
	0.8	EX	MP	CM	CO	ER	EX	
	1.0	EX	MP	CM	CO	ER	EX	
Keep k iterations	500	EX	MP	CM	CO	ER	EX	
	100	EX	MP	CM	CO	ER	EX	
	50	EX	MP	CM	CO	ER	EX	
	10	EX	MP	CM	CO	ER	EX	
	1	EX	MP	CM	CO	ER	EX	
	average	EX	MP	CM	CO	ER	EX	

Fig. 13 Intra-process Reduction: Retention of Performance Trends with Varying Thresholds for early_gather

		MPI Barrier					do work
relative difference	no loss	EX	MP	SN	BA	WB	EX
	0.1	EX	MP	SN	BA	WB	EX
	0.2	EX	MP	SN	BA	WB	EX
	0.4	EX	MP	SN	BA	WB	EX
	0.6	EX	MP	SN	BA	WB	EX
	0.8	EX	MP	SN	BA	WB	EX
absolute difference	1.0	EX	MP	SN	BA	WB	EX
	10	EX	MP	SN	BA	WB	EX
	100	EX	MP	SN	BA	WB	EX
	1000	EX	MP	SN	BA	WB	EX
	10000	EX	MP	SN	BA	WB	EX
	100000	EX	MP	SN	BA	WB	EX
Manhattan distance	100000	EX	MP	SN	BA	WB	EX
	0.1	EX	MP	SN	BA	WB	EX
	0.2	EX	MP	SN	BA	WB	EX
	0.4	EX	MP	SN	BA	WB	EX
	0.6	EX	MP	SN	BA	WB	EX
	0.8	EX	MP	SN	BA	WB	EX
Euclidean distance	1.0	EX	MP	SN	BA	WB	EX
	0.1	EX	MP	SN	BA	WB	EX
	0.2	EX	MP	SN	BA	WB	EX
	0.4	EX	MP	SN	BA	WB	EX
	0.6	EX	MP	SN	BA	WB	EX
	0.8	EX	MP	SN	BA	WB	EX
Chebyshev distance	1.0	EX	MP	SN	BA	WB	EX
	0.1	EX	MP	SN	BA	WB	EX
	0.2	EX	MP	SN	BA	WB	EX
	0.4	EX	MP	SN	BA	WB	EX
	0.6	EX	MP	SN	BA	WB	EX
	0.8	EX	MP	SN	BA	WB	EX
Average Wavelet	1.0	EX	MP	SN	BA	WB	EX
	0.1	EX	MP	SN	BA	WB	EX
	0.2	EX	MP	SN	BA	WB	EX
	0.4	EX	MP	SN	BA	WB	EX
	0.6	EX	MP	SN	BA	WB	EX
	0.8	EX	MP	SN	BA	WB	EX
Haar Wavelet	1.0	EX	MP	SN	BA	WB	EX
	0.1	EX	MP	SN	BA	WB	EX
	0.2	EX	MP	SN	BA	WB	EX
	0.4	EX	MP	SN	BA	WB	EX
	0.6	EX	MP	SN	BA	WB	EX
	0.8	EX	MP	SN	BA	WB	EX
Keep k iterations	1.0	EX	MP	SN	BA	WB	EX
	500	EX	MP	SN	BA	WB	EX
	100	EX	MP	SN	BA	WB	EX
	10	EX	MP	SN	BA	WB	EX
	1	EX	MP	SN	BA	WB	EX
	average	EX	MP	SN	BA	WB	EX

Fig. 14 Intra-process Reduction: Retention of Performance Trends with Varying Threshold for imbalance_at_mpi_barrier

		MPI Bcast					do work
	no loss	EX	MP	CM	CO	LB	EX
relative difference	0.1	EX	MP	CM	CO	LB	EX
	0.2	EX	MP	CM	CO	LB	EX
	0.4	EX	MP	CM	CO	LB	EX
	0.6	EX	MP	CM	CO	LB	EX
	0.8	EX	MP	CM	CO	LB	EX
	1.0	EX	MP	CM	CO	LB	EX
absolute difference	10	EX	MP	CM	CO	LB	EX
	100	EX	MP	CM	CO	LB	EX
	1000	EX	MP	CM	CO	LB	EX
	10000	EX	MP	CM	CO	LB	EX
	100000	EX	MP	CM	CO	LB	EX
	1000000	EX	MP	CM	CO	LB	EX
Manhattan distance	0.1	EX	MP	CM	CO	LB	EX
	0.2	EX	MP	CM	CO	LB	EX
	0.4	EX	MP	CM	CO	LB	EX
	0.6	EX	MP	CM	CO	LB	EX
	0.8	EX	MP	CM	CO	LB	EX
	1.0	EX	MP	CM	CO	LB	EX
Euclidean distance	0.1	EX	MP	CM	CO	LB	EX
	0.2	EX	MP	CM	CO	LB	EX
	0.4	EX	MP	CM	CO	LB	EX
	0.6	EX	MP	CM	CO	LB	EX
	0.8	EX	MP	CM	CO	LB	EX
	1.0	EX	MP	CM	CO	LB	EX
Chebyshev distance	0.1	EX	MP	CM	CO	LB	EX
	0.2	EX	MP	CM	CO	LB	EX
	0.4	EX	MP	CM	CO	LB	EX
	0.6	EX	MP	CM	CO	LB	EX
	0.8	EX	MP	CM	CO	LB	EX
	1.0	EX	MP	CM	CO	LB	EX
Average Wavelet	0.1	EX	MP	CM	CO	LB	EX
	0.2	EX	MP	CM	CO	LB	EX
	0.4	EX	MP	CM	CO	LB	EX
	0.6	EX	MP	CM	CO	LB	EX
	0.8	EX	MP	CM	CO	LB	EX
	1.0	EX	MP	CM	CO	LB	EX
Haar Wavelet	0.1	EX	MP	CM	CO	LB	EX
	0.2	EX	MP	CM	CO	LB	EX
	0.4	EX	MP	CM	CO	LB	EX
	0.6	EX	MP	CM	CO	LB	EX
	0.8	EX	MP	CM	CO	LB	EX
	1.0	EX	MP	CM	CO	LB	EX
Keep k iterations	500	EX	MP	CM	CO	LB	EX
	100	EX	MP	CM	CO	LB	EX
	50	EX	MP	CM	CO	LB	EX
	10	EX	MP	CM	CO	LB	EX
	1	EX	MP	CM	CO	LB	EX
		average	EX	MP	CM	CO	LB

Fig. 15 Intra-process Reduction: Retention of Performance Trends with Varying Threshold for late_broadcast

		MPI Ssend					do work
	no loss	EX	MP	CM	P2	LR	EX
relative difference	0.1	EX	MP	CM	P2	LR	EX
	0.2	EX	MP	CM	P2	LR	EX
	0.4	EX	MP	CM	P2	LR	EX
	0.6	EX	MP	CM	P2	LR	EX
	0.8	EX	MP	CM	P2	LR	EX
	1.0	EX	MP	CM	P2	LR	EX
absolute difference	10	EX	MP	CM	P2	LR	EX
	100	EX	MP	CM	P2	LR	EX
	1000	EX	MP	CM	P2	LR	EX
	10000	EX	MP	CM	P2	LR	EX
	100000	EX	MP	CM	P2	LR	EX
	1000000	EX	MP	CM	P2	LR	EX
Manhattan distance	0.1	EX	MP	CM	P2	LR	EX
	0.2	EX	MP	CM	P2	LR	EX
	0.4	EX	MP	CM	P2	LR	EX
	0.6	EX	MP	CM	P2	LR	EX
	0.8	EX	MP	CM	P2	LR	EX
	1.0	EX	MP	CM	P2	LR	EX
Euclidean distance	0.1	EX	MP	CM	P2	LR	EX
	0.2	EX	MP	CM	P2	LR	EX
	0.4	EX	MP	CM	P2	LR	EX
	0.6	EX	MP	CM	P2	LR	EX
	0.8	EX	MP	CM	P2	LR	EX
	1.0	EX	MP	CM	P2	LR	EX
Chebyshev distance	0.1	EX	MP	CM	P2	LR	EX
	0.2	EX	MP	CM	P2	LR	EX
	0.4	EX	MP	CM	P2	LR	EX
	0.6	EX	MP	CM	P2	LR	EX
	0.8	EX	MP	CM	P2	LR	EX
	1.0	EX	MP	CM	P2	LR	EX
Average Wavelet	0.1	EX	MP	CM	P2	LR	EX
	0.2	EX	MP	CM	P2	LR	EX
	0.4	EX	MP	CM	P2	LR	EX
	0.6	EX	MP	CM	P2	LR	EX
	0.8	EX	MP	CM	P2	LR	EX
	1.0	EX	MP	CM	P2	LR	EX
Haar Wavelet	0.1	EX	MP	CM	P2	LR	EX
	0.2	EX	MP	CM	P2	LR	EX
	0.4	EX	MP	CM	P2	LR	EX
	0.6	EX	MP	CM	P2	LR	EX
	0.8	EX	MP	CM	P2	LR	EX
	1.0	EX	MP	CM	P2	LR	EX
Keep k Iterations	500	EX	MP	CM	P2	LR	EX
	100	EX	MP	CM	P2	LR	EX
	50	EX	MP	CM	P2	LR	EX
	10	EX	MP	CM	P2	LR	EX
	1	EX	MP	CM	P2	LR	EX
		average	EX	MP	CM	P2	LR

Fig. 16 Intra-process Reduction: Retention and Performance Trends with Varying Thresholds for late_receiver

		MPI Recv					do work
	no loss	EX	MP	CM	P2	LS	EX
relative difference	0.1	EX	MP	CM	P2	LS	EX
	0.2	EX	MP	CM	P2	LS	EX
	0.4	EX	MP	CM	P2	LS	EX
	0.6	EX	MP	CM	P2	LS	EX
	0.8	EX	MP	CM	P2	LS	EX
	1.0	EX	MP	CM	P2	LS	EX
absolute difference	10	EX	MP	CM	P2	LS	EX
	100	EX	MP	CM	P2	LS	EX
	1000	EX	MP	CM	P2	LS	EX
	10000	EX	MP	CM	P2	LS	EX
	100000	EX	MP	CM	P2	LS	EX
	1000000	EX	MP	CM	P2	LS	EX
Manhattan distance	0.1	EX	MP	CM	P2	LS	EX
	0.2	EX	MP	CM	P2	LS	EX
	0.4	EX	MP	CM	P2	LS	EX
	0.6	EX	MP	CM	P2	LS	EX
	0.8	EX	MP	CM	P2	LS	EX
	1.0	EX	MP	CM	P2	LS	EX
Euclidean distance	0.1	EX	MP	CM	P2	LS	EX
	0.2	EX	MP	CM	P2	LS	EX
	0.4	EX	MP	CM	P2	LS	EX
	0.6	EX	MP	CM	P2	LS	EX
	0.8	EX	MP	CM	P2	LS	EX
	1.0	EX	MP	CM	P2	LS	EX
Chebyshev distance	0.1	EX	MP	CM	P2	LS	EX
	0.2	EX	MP	CM	P2	LS	EX
	0.4	EX	MP	CM	P2	LS	EX
	0.6	EX	MP	CM	P2	LS	EX
	0.8	EX	MP	CM	P2	LS	EX
	1.0	EX	MP	CM	P2	LS	EX
Average Wavelet	0.1	EX	MP	CM	P2	LS	EX
	0.2	EX	MP	CM	P2	LS	EX
	0.4	EX	MP	CM	P2	LS	EX
	0.6	EX	MP	CM	P2	LS	EX
	0.8	EX	MP	CM	P2	LS	EX
	1.0	EX	MP	CM	P2	LS	EX
Haar Wavelet	0.1	EX	MP	CM	P2	LS	EX
	0.2	EX	MP	CM	P2	LS	EX
	0.4	EX	MP	CM	P2	LS	EX
	0.6	EX	MP	CM	P2	LS	EX
	0.8	EX	MP	CM	P2	LS	EX
	1.0	EX	MP	CM	P2	LS	EX
Keep k iterations	500	EX	MP	CM	P2	LS	EX
	100	EX	MP	CM	P2	LS	EX
	50	EX	MP	CM	P2	LS	EX
	10	EX	MP	CM	P2	LS	EX
	1	EX	MP	CM	P2	LS	EX
	average	EX	MP	CM	P2	LS	EX

Fig. 17 Intra-process Reduction: Retention of Performance Trends with Varying Thresholds for late_sender

		MPI Gather					do work
	no loss	EX	MP	CM	CO	ER	EX
relative difference	0.1	EX	MP	CM	CO	ER	EX
	0.2	EX	MP	CM	CO	ER	EX
	0.4	EX	MP	CM	CO	ER	EX
	0.6	EX	MP	CM	CO	ER	EX
	0.8	EX	MP	CM	CO	ER	EX
	1.0	EX	MP	CM	CO	ER	EX
absolute difference	10	EX	MP	CM	CO	ER	EX
	100	EX	MP	CM	CO	ER	EX
	1000	EX	MP	CM	CO	ER	EX
	10000	EX	MP	CM	CO	ER	EX
	100000	EX	MP	CM	CO	ER	EX
	1000000	EX	MP	CM	CO	ER	EX
Manhattan distance	0.1	EX	MP	CM	CO	ER	EX
	0.2	EX	MP	CM	CO	ER	EX
	0.4	EX	MP	CM	CO	ER	EX
	0.6	EX	MP	CM	CO	ER	EX
	0.8	EX	MP	CM	CO	ER	EX
	1.0	EX	MP	CM	CO	ER	EX
Euclidean distance	0.1	EX	MP	CM	CO	ER	EX
	0.2	EX	MP	CM	CO	ER	EX
	0.4	EX	MP	CM	CO	ER	EX
	0.6	EX	MP	CM	CO	ER	EX
	0.8	EX	MP	CM	CO	ER	EX
	1.0	EX	MP	CM	CO	ER	EX
Chebyshev distance	0.1	EX	MP	CM	CO	ER	EX
	0.2	EX	MP	CM	CO	ER	EX
	0.4	EX	MP	CM	CO	ER	EX
	0.6	EX	MP	CM	CO	ER	EX
	0.8	EX	MP	CM	CO	ER	EX
	1.0	EX	MP	CM	CO	ER	EX
Average Wavelet	0.1	EX	MP	CM	CO	ER	EX
	0.2	EX	MP	CM	CO	ER	EX
	0.4	EX	MP	CM	CO	ER	EX
	0.6	EX	MP	CM	CO	ER	EX
	0.8	EX	MP	CM	CO	ER	EX
	1.0	EX	MP	CM	CO	ER	EX
Haar Wavelet	0.1	EX	MP	CM	CO	ER	EX
	0.2	EX	MP	CM	CO	ER	EX
	0.4	EX	MP	CM	CO	ER	EX
	0.6	EX	MP	CM	CO	ER	EX
	0.8	EX	MP	CM	CO	ER	EX
	1.0	EX	MP	CM	CO	ER	EX
Keep k iterations	500	EX	MP	CM	CO	ER	EX
	100	EX	MP	CM	CO	ER	EX
	50	EX	MP	CM	CO	ER	EX
	10	EX	MP	CM	CO	ER	EX
	1	EX	MP	CM	CO	ER	EX
		average	EX	MP	CM	CO	ER

Fig. 18 Intra-process Reduction: Retention of Performance Trends with Varying Thresholds for Nto1_32

		MPI Barrier						do work
	no loss	EX	MP	SN	BA	WB	BC	EX
relative difference	0.1	EX	MP	SN	BA	WB	BC	EX
	0.2	EX	MP	SN	BA	WB	BC	EX
	0.4	EX	MP	SN	BA	WB	BC	EX
	0.6	EX	MP	SN	BA	WB	BC	EX
	0.8	EX	MP	SN	BA	WB	BC	EX
absolute difference	1.0	EX	MP	SN	BA	WB	BC	EX
	10	EX	MP	SN	BA	WB	BC	EX
	100	EX	MP	SN	BA	WB	BC	EX
	1000	EX	MP	SN	BA	WB	BC	EX
	10000	EX	MP	SN	BA	WB	BC	EX
	100000	EX	MP	SN	BA	WB	BC	EX
Manhattan distance	1000000	EX	MP	SN	BA	WB	BC	EX
	0.1	EX	MP	SN	BA	WB	BC	EX
	0.2	EX	MP	SN	BA	WB	BC	EX
	0.4	EX	MP	SN	BA	WB	BC	EX
	0.6	EX	MP	SN	BA	WB	BC	EX
	0.8	EX	MP	SN	BA	WB	BC	EX
Euclidean distance	1.0	EX	MP	SN	BA	WB	BC	EX
	0.1	EX	MP	SN	BA	WB	BC	EX
	0.2	EX	MP	SN	BA	WB	BC	EX
	0.4	EX	MP	SN	BA	WB	BC	EX
	0.6	EX	MP	SN	BA	WB	BC	EX
	0.8	EX	MP	SN	BA	WB	BC	EX
Chebyshev distance	1.0	EX	MP	SN	BA	WB	BC	EX
	0.1	EX	MP	SN	BA	WB	BC	EX
	0.2	EX	MP	SN	BA	WB	BC	EX
	0.4	EX	MP	SN	BA	WB	BC	EX
	0.6	EX	MP	SN	BA	WB	BC	EX
	0.8	EX	MP	SN	BA	WB	BC	EX
Average Wavelet	1.0	EX	MP	SN	BA	WB	BC	EX
	0.1	EX	MP	SN	BA	WB	BC	EX
	0.2	EX	MP	SN	BA	WB	BC	EX
	0.4	EX	MP	SN	BA	WB	BC	EX
	0.6	EX	MP	SN	BA	WB	BC	EX
	0.8	EX	MP	SN	BA	WB	BC	EX
Haar Wavelet	1.0	EX	MP	SN	BA	WB	BC	EX
	0.1	EX	MP	SN	BA	WB	BC	EX
	0.2	EX	MP	SN	BA	WB	BC	EX
	0.4	EX	MP	SN	BA	WB	BC	EX
	0.6	EX	MP	SN	BA	WB	BC	EX
	0.8	EX	MP	SN	BA	WB	BC	EX
Keep k iterations	1.0	EX	MP	SN	BA	WB	BC	EX
	500	EX	MP	SN	BA	WB	BC	EX
	100	EX	MP	SN	BA	WB	BC	EX
	50	EX	MP	SN	BA	WB	BC	EX
	10	EX	MP	SN	BA	WB	BC	EX
	1	EX	MP	SN	BA	WB	BC	EX
	average	EX	MP	SN	BA	WB	BC	EX

Fig. 19 Intra-process Reduction: Retention of Performance Trends with Varying Thresholds for NtoN_32

		MPI Bcast					do work
	no loss	EX	MP	CM	CO	LB	EX
relative difference	0.1	EX	MP	CM	CO	LB	EX
	0.2	EX	MP	CM	CO	LB	EX
	0.4	EX	MP	CM	CO	LB	EX
	0.6	EX	MP	CM	CO	LB	EX
	0.8	EX	MP	CM	CO	LB	EX
	1.0	EX	MP	CM	CO	LB	EX
absolute difference	10	EX	MP	CM	CO	LB	EX
	100	EX	MP	CM	CO	LB	EX
	1000	EX	MP	CM	CO	LB	EX
	10000	EX	MP	CM	CO	LB	EX
	100000	EX	MP	CM	CO	LB	EX
	1000000	EX	MP	CM	CO	LB	EX
Manhattan distance	0.1	EX	MP	CM	CO	LB	EX
	0.2	EX	MP	CM	CO	LB	EX
	0.4	EX	MP	CM	CO	LB	EX
	0.6	EX	MP	CM	CO	LB	EX
	0.8	EX	MP	CM	CO	LB	EX
	1.0	EX	MP	CM	CO	LB	EX
Euclidean distance	0.1	EX	MP	CM	CO	LB	EX
	0.2	EX	MP	CM	CO	LB	EX
	0.4	EX	MP	CM	CO	LB	EX
	0.6	EX	MP	CM	CO	LB	EX
	0.8	EX	MP	CM	CO	LB	EX
	1.0	EX	MP	CM	CO	LB	EX
Chebyshev distance	0.1	EX	MP	CM	CO	LB	EX
	0.2	EX	MP	CM	CO	LB	EX
	0.4	EX	MP	CM	CO	LB	EX
	0.6	EX	MP	CM	CO	LB	EX
	0.8	EX	MP	CM	CO	LB	EX
	1.0	EX	MP	CM	CO	LB	EX
Average Wavelet	0.1	EX	MP	CM	CO	LB	EX
	0.2	EX	MP	CM	CO	LB	EX
	0.4	EX	MP	CM	CO	LB	EX
	0.6	EX	MP	CM	CO	LB	EX
	0.8	EX	MP	CM	CO	LB	EX
	1.0	EX	MP	CM	CO	LB	EX
Haar Wavelet	0.1	EX	MP	CM	CO	LB	EX
	0.2	EX	MP	CM	CO	LB	EX
	0.4	EX	MP	CM	CO	LB	EX
	0.6	EX	MP	CM	CO	LB	EX
	0.8	EX	MP	CM	CO	LB	EX
	1.0	EX	MP	CM	CO	LB	EX
Keep k iterations	500	EX	MP	CM	CO	LB	EX
	100	EX	MP	CM	CO	LB	EX
	50	EX	MP	CM	CO	LB	EX
	10	EX	MP	CM	CO	LB	EX
	1	EX	MP	CM	CO	LB	EX
	average	EX	MP	CM	CO	LB	EX

Fig. 20 Intra-process Reduction: Retention of Performance Trends with Varying Thresholds for 1toN_32

		MPI Ssend					MPI Recv					do work
	no loss	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
Relative Difference	0.1	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.2	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.4	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.6	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.8	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	1.0	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
Absolute Difference	10	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	100	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	1000	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	10000	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	100000	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	1000000	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
Manhattan distance	0.1	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.2	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.4	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.6	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.8	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	1.0	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
Euclidean distance	0.1	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.2	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.4	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.6	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.8	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	1.0	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
Chebyshev distance	0.1	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.2	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.4	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.6	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.8	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	1.0	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
Average Wavelet	0.1	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.2	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.4	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.6	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.8	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	1.0	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
Haar Wavelet	0.1	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.2	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.4	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.6	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.8	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	1.0	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
Keep k iterations	500	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	100	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	50	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	10	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	1	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	average	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX

Fig. 21 Intra-process Reduction: Retention of Performance Trends with Varying Thresholds for 1to1r_32

		MPI Ssend					MPI Recv					do work
	no loss	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
Relative Difference	0.1	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.2	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.4	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.6	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.8	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	1.0	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
Absolute Difference	10	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	100	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	1000	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	10000	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	100000	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	1000000	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
Manhattan distance	0.1	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.2	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.4	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.6	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.8	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	1.0	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
Euclidean distance	0.1	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.2	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.4	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.6	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.8	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	1.0	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
Chebyshev distance	0.1	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.2	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.4	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.6	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.8	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	1.0	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
Average Wavelet	0.1	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.2	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.4	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.6	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.8	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	1.0	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
Haar Wavelet	0.1	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.2	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.4	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.6	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.8	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	1.0	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
Keep k iterations	500	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	100	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	50	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	10	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	1	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	average	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX

Fig. 22 Intra-process Reduction: Retention of Performance Trends with Varying Thresholds for 1to1s_32

		MPI Gather					do work
	no loss	EX	MP	CM	CO	ER	FX
relative difference	0.1	EX	MP	CM	CO	ER	FX
	0.2	EX	MP	CM	CO	ER	FX
	0.4	EX	MP	CM	CO	ER	FX
	0.6	EX	MP	CM	CO	ER	FX
	0.8	EX	MP	CM	CO	ER	FX
	1.0	EX	MP	CM	CO	ER	FX
absolute difference	10	EX	MP	CM	CO	ER	FX
	100	EX	MP	CM	CO	ER	FX
	1000	EX	MP	CM	CO	ER	FX
	10000	EX	MP	CM	CO	ER	FX
	100000	EX	MP	CM	CO	ER	FX
	1000000	EX	MP	CM	CO	ER	FX
Manhattan distance	0.1	EX	MP	CM	CO	ER	FX
	0.2	EX	MP	CM	CO	ER	FX
	0.4	EX	MP	CM	CO	ER	FX
	0.6	EX	MP	CM	CO	ER	FX
	0.8	EX	MP	CM	CO	ER	FX
	1.0	EX	MP	CM	CO	ER	FX
Euclidean distance	0.1	EX	MP	CM	CO	ER	FX
	0.2	EX	MP	CM	CO	ER	FX
	0.4	EX	MP	CM	CO	ER	FX
	0.6	EX	MP	CM	CO	ER	FX
	0.8	EX	MP	CM	CO	ER	FX
	1.0	EX	MP	CM	CO	ER	FX
Chebyshev distance	0.1	EX	MP	CM	CO	ER	FX
	0.2	EX	MP	CM	CO	ER	FX
	0.4	EX	MP	CM	CO	ER	FX
	0.6	EX	MP	CM	CO	ER	FX
	0.8	EX	MP	CM	CO	ER	FX
	1.0	EX	MP	CM	CO	ER	FX
Average Wavelet	0.1	EX	MP	CM	CO	ER	FX
	0.2	EX	MP	CM	CO	ER	FX
	0.4	EX	MP	CM	CO	ER	FX
	0.6	EX	MP	CM	CO	ER	FX
	0.8	EX	MP	CM	CO	ER	FX
	1.0	EX	MP	CM	CO	ER	FX
Haar Wavelet	0.1	EX	MP	CM	CO	ER	FX
	0.2	EX	MP	CM	CO	ER	FX
	0.4	EX	MP	CM	CO	ER	FX
	0.6	EX	MP	CM	CO	ER	FX
	0.8	EX	MP	CM	CO	ER	FX
	1.0	EX	MP	CM	CO	ER	FX
Keep k iterations	500	EX	MP	CM	CO	ER	FX
	100	EX	MP	CM	CO	ER	FX
	50	EX	MP	CM	CO	ER	FX
	10	EX	MP	CM	CO	ER	FX
	1	EX	MP	CM	CO	ER	FX
		average	EX	MP	CM	CO	ER

Fig. 23 Intra-process Reduction: Retention of Performance Trends with Varying Thresholds for Nto1_1024

		MPI Barrier						do work
	no loss	EX	MP	SN	BA	WB	BC	EX
relative difference	0.1	EX	MP	SN	BA	WB	BC	EX
	0.2	EX	MP	SN	BA	WB	BC	EX
	0.4	EX	MP	SN	BA	WB	BC	EX
	0.6	EX	MP	SN	BA	WB	BC	EX
	0.8	EX	MP	SN	BA	WB	BC	EX
absolute difference	10	EX	MP	SN	BA	WB	BC	EX
	100	EX	MP	SN	BA	WB	BC	EX
	1000	EX	MP	SN	BA	WB	BC	EX
	10000	EX	MP	SN	BA	WB	BC	EX
	100000	EX	MP	SN	BA	WB	BC	EX
Manhattan distance	0.1	EX	MP	SN	BA	WB	BC	EX
	0.2	EX	MP	SN	BA	WB	BC	EX
	0.4	EX	MP	SN	BA	WB	BC	EX
	0.6	EX	MP	SN	BA	WB	BC	EX
	0.8	EX	MP	SN	BA	WB	BC	EX
Euclidean distance	0.1	EX	MP	SN	BA	WB	BC	EX
	0.2	EX	MP	SN	BA	WB	BC	EX
	0.4	EX	MP	SN	BA	WB	BC	EX
	0.6	EX	MP	SN	BA	WB	BC	EX
	0.8	EX	MP	SN	BA	WB	BC	EX
Chebyshev distance	0.1	EX	MP	SN	BA	WB	BC	EX
	0.2	EX	MP	SN	BA	WB	BC	EX
	0.4	EX	MP	SN	BA	WB	BC	EX
	0.6	EX	MP	SN	BA	WB	BC	EX
	0.8	EX	MP	SN	BA	WB	BC	EX
Average Wavelet	0.1	EX	MP	SN	BA	WB	BC	EX
	0.2	EX	MP	SN	BA	WB	BC	EX
	0.4	EX	MP	SN	BA	WB	BC	EX
	0.6	EX	MP	SN	BA	WB	BC	EX
	0.8	EX	MP	SN	BA	WB	BC	EX
Haar Wavelet	0.1	EX	MP	SN	BA	WB	BC	EX
	0.2	EX	MP	SN	BA	WB	BC	EX
	0.4	EX	MP	SN	BA	WB	BC	EX
	0.6	EX	MP	SN	BA	WB	BC	EX
	0.8	EX	MP	SN	BA	WB	BC	EX
Keep k iterations	500	EX	MP	SN	BA	WB	BC	EX
	100	EX	MP	SN	BA	WB	BC	EX
	50	EX	MP	SN	BA	WB	BC	EX
	10	EX	MP	SN	BA	WB	BC	EX
	1	EX	MP	SN	BA	WB	BC	EX
	average	EX	MP	SN	BA	WB	BC	EX

Fig. 24 Intra-process Reduction: Retention of Performance Trends with Varying Thresholds for NtoN_1024

		MPI Bcast					do work
	no loss	EX	MP	CM	CO	LB	
relative difference	0.1	EX	MP	CM	CO	LB	
	0.2	EX	MP	CM	CO	LB	
	0.4	EX	MP	CM	CO	LB	
	0.6	EX	MP	CM	CO	LB	
	0.8	EX	MP	CM	CO	LB	
	1.0	EX	MP	CM	CO	LB	EX
absolute difference	10	EX	MP	CM	CO	LB	
	100	EX	MP	CM	CO	LB	
	1000	EX	MP	CM	CO	LB	
	10000	EX	MP	CM	CO	LB	
	100000	EX	MP	CM	CO	LB	EX
	1000000	EX	MP	CM	CO	LB	EX
Manhattan distance	0.1	EX	MP	CM	CO	LB	
	0.2	EX	MP	CM	CO	LB	
	0.4	EX	MP	CM	CO	LB	
	0.6	EX	MP	CM	CO	LB	
	0.8	EX	MP	CM	CO	LB	EX
	1.0	EX	MP	CM	CO	LB	EX
Euclidean distance	0.1	EX	MP	CM	CO	LB	
	0.2	EX	MP	CM	CO	LB	
	0.4	EX	MP	CM	CO	LB	
	0.6	EX	MP	CM	CO	LB	
	0.8	EX	MP	CM	CO	LB	EX
	1.0	EX	MP	CM	CO	LB	EX
Chebyshev distance	0.1					LB	EX
	0.2					LB	EX
	0.4					LB	EX
	0.6					LB	EX
	0.8					LB	EX
	1.0	EX	MP	CM	CO	LB	EX
Average Wavelet	0.1	EX	MP	CM	CO	LB	
	0.2	EX	MP	CM	CO	LB	
	0.4	EX	MP	CM	CO	LB	
	0.6	EX	MP	CM	CO	LB	
	0.8	EX	MP	CM	CO	LB	
	1.0	EX	MP	CM	CO	LB	
Haar Wavelet	0.1	EX	MP	CM	CO	LB	
	0.2	EX	MP	CM	CO	LB	
	0.4	EX	MP	CM	CO	LB	
	0.6	EX	MP	CM	CO	LB	
	0.8	EX	MP	CM	CO	LB	
	1.0	EX	MP	CM	CO	LB	
Keep k iterations	500	EX	MP	CM	CO	LB	EX
	100	EX	MP	CM	CO	LB	EX
	50	EX	MP	CM	CO	LB	EX
	10	EX	MP	CM	CO	LB	EX
	1	EX	MP	CM	CO	LB	EX
	average	EX	MP	CM	CO	LB	EX

Fig. 25 Intra-process Reduction: Retention of Performance Trends with Varying Thresholds for 1toN_1024

		MPI Ssend					MPI Recv					do work
	no loss	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
Relative Difference	0.1	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.2	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.4	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.6	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.8	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	1.0	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
Absolute Difference	10	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	100	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	1000	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	10000	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	100000	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	1000000	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
Manhattan distance	0.1	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.2	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.4	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.6	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.8	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	1.0	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
Euclidean distance	0.1	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.2	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.4	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.6	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.8	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	1.0	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
Chebyshev distance	0.1	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.2	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.4	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.6	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.8	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	1.0	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
Average Wavelet	0.1	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.2	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.4	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.6	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.8	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	1.0	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
Haar Wavelet	0.1	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.2	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.4	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.6	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.8	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	1.0	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
Keep k iterations	500	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	100	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	50	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	10	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	1	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	average	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX

Fig. 26 Intra-process Reduction: Retention of Performance Trends with Varying Thresholds for 1to1r_1024

		MPI Ssend					MPI Recv					do work
	no loss	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
Relative Difference	0.1	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.2	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.4	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.6	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.8	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	1.0	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
Absolute Difference	10	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	100	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	1000	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	10000	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	100000	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	1000000	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
Manhattan distance	0.1	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.2	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.4	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.6	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.8	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	1.0	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
Euclidean distance	0.1	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.2	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.4	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.6	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.8	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	1.0	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
Chebyshev distance	0.1	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.2	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.4	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.6	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.8	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	1.0	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
Average Wavelet	0.1	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.2	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.4	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.6	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.8	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	1.0	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
Haar Wavelet	0.1	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.2	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.4	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.6	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.8	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	1.0	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
Keep k Iterations	500	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	100	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	50	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	10	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	1	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	average	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX

Fig. 27 Intra-process Reduction: Retention of Performance Trends with Varying Thresholds for 1to1s_1024

		mpi_recv						sweep
	no loss	EX	MP	CM	P2	LS	MO	EX
relative difference	0.1	EX	MP	CM	P2	LS	MO	EX
	0.2	EX	MP	CM	P2	LS	MO	EX
	0.4	EX	MP	CM	P2	LS	MO	EX
	0.6	EX	MP	CM	P2	LS	MO	EX
	0.8	EX	MP	CM	P2	LS	MO	EX
	1.0	EX	MP	CM	P2	LS	MO	EX
absolute difference	10	EX	MP	CM	P2	LS	MO	EX
	100	EX	MP	CM	P2	LS	MO	EX
	1000	EX	MP	CM	P2	LS	MO	EX
	10000	EX	MP	CM	P2	LS	MO	EX
	100000	EX	MP	CM	P2	LS	MO	EX
	1000000	EX	MP	CM	P2	LS	MO	EX
Manhattan distance	0.1	EX	MP	CM	P2	LS	MO	EX
	0.2	EX	MP	CM	P2	LS	MO	EX
	0.4	EX	MP	CM	P2	LS	MO	EX
	0.6	EX	MP	CM	P2	LS	MO	EX
	0.8	EX	MP	CM	P2	LS	MO	EX
	1.0	EX	MP	CM	P2	LS	MO	EX
Euclidean distance	0.1	EX	MP	CM	P2	LS	MO	EX
	0.2	EX	MP	CM	P2	LS	MO	EX
	0.4	EX	MP	CM	P2	LS	MO	EX
	0.6	EX	MP	CM	P2	LS	MO	EX
	0.8	EX	MP	CM	P2	LS	MO	EX
	1.0	EX	MP	CM	P2	LS	MO	EX
Chebyshev distance	0.1	EX	MP	CM	P2	LS	MO	EX
	0.2	EX	MP	CM	P2	LS	MO	EX
	0.4	EX	MP	CM	P2	LS	MO	EX
	0.6	EX	MP	CM	P2	LS	MO	EX
	0.8	EX	MP	CM	P2	LS	MO	EX
	1.0	EX	MP	CM	P2	LS	MO	EX
Average Wavelet	0.1	EX	MP	CM	P2	LS	MO	EX
	0.2	EX	MP	CM	P2	LS	MO	EX
	0.4	EX	MP	CM	P2	LS	MO	EX
	0.6	EX	MP	CM	P2	LS	MO	EX
	0.8	EX	MP	CM	P2	LS	MO	EX
	1.0	EX	MP	CM	P2	LS	MO	EX
Haar Wavelet	0.1	EX	MP	CM	P2	LS	MO	EX
	0.2	EX	MP	CM	P2	LS	MO	EX
	0.4	EX	MP	CM	P2	LS	MO	EX
	0.6	EX	MP	CM	P2	LS	MO	EX
	0.8	EX	MP	CM	P2	LS	MO	EX
	1.0	EX	MP	CM	P2	LS	MO	EX
Keep k iterations	500	EX	MP	CM	P2	LS	MO	EX
	100	EX	MP	CM	P2	LS	MO	EX
	50	EX	MP	CM	P2	LS	MO	EX
	10	EX	MP	CM	P2	LS	MO	EX
	1	EX	MP	CM	P2	LS	MO	EX
		average	EX	MP	CM	P2	LS	MO

Fig. 29 Intra-process Reduction: Retention of Performance Trends with Varying Thresholds for sweep3d_32p

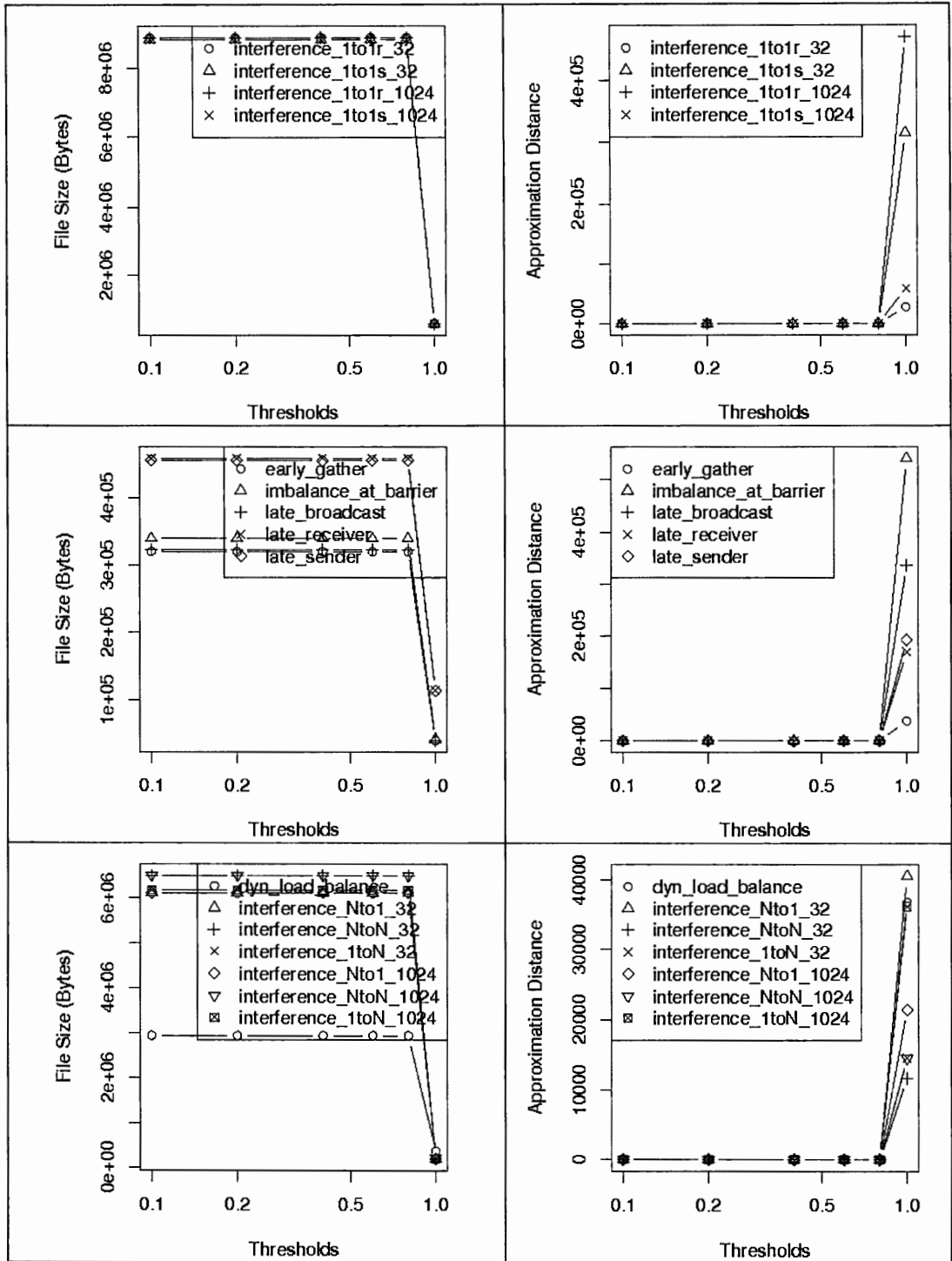


Fig. 30 Inter-process Reduction: File Size and Approximation Distance for Varying Duration Thresholds and Relative Distance

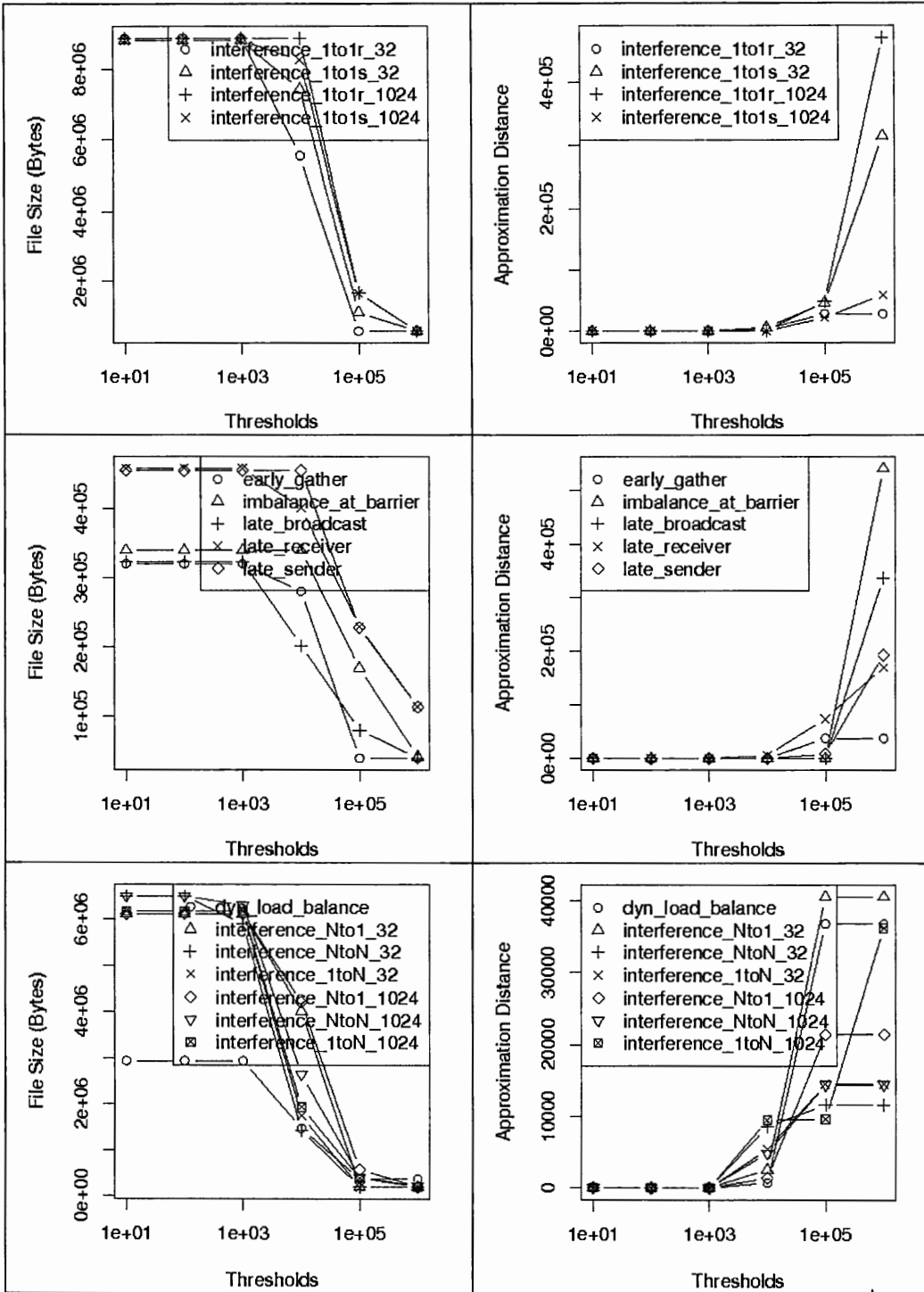


Fig. 31 Inter-process Reduction: File Size and Approximation Distance for Varying Duration Thresholds and Absolute Distance

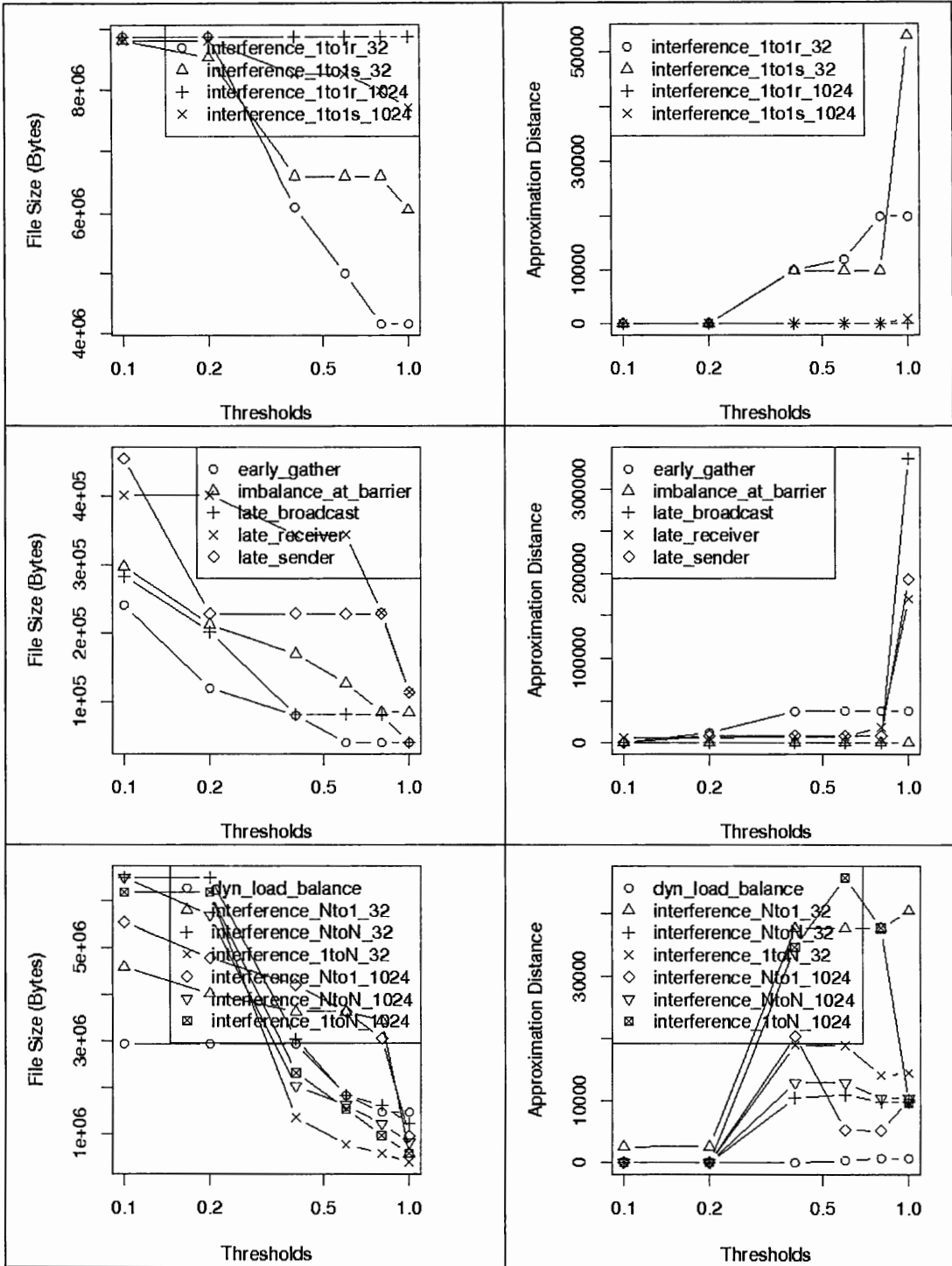


Fig. 32 Inter-process Reduction: File Size and Approximation Distance for Varying Duration Thresholds and Manhattan Distance

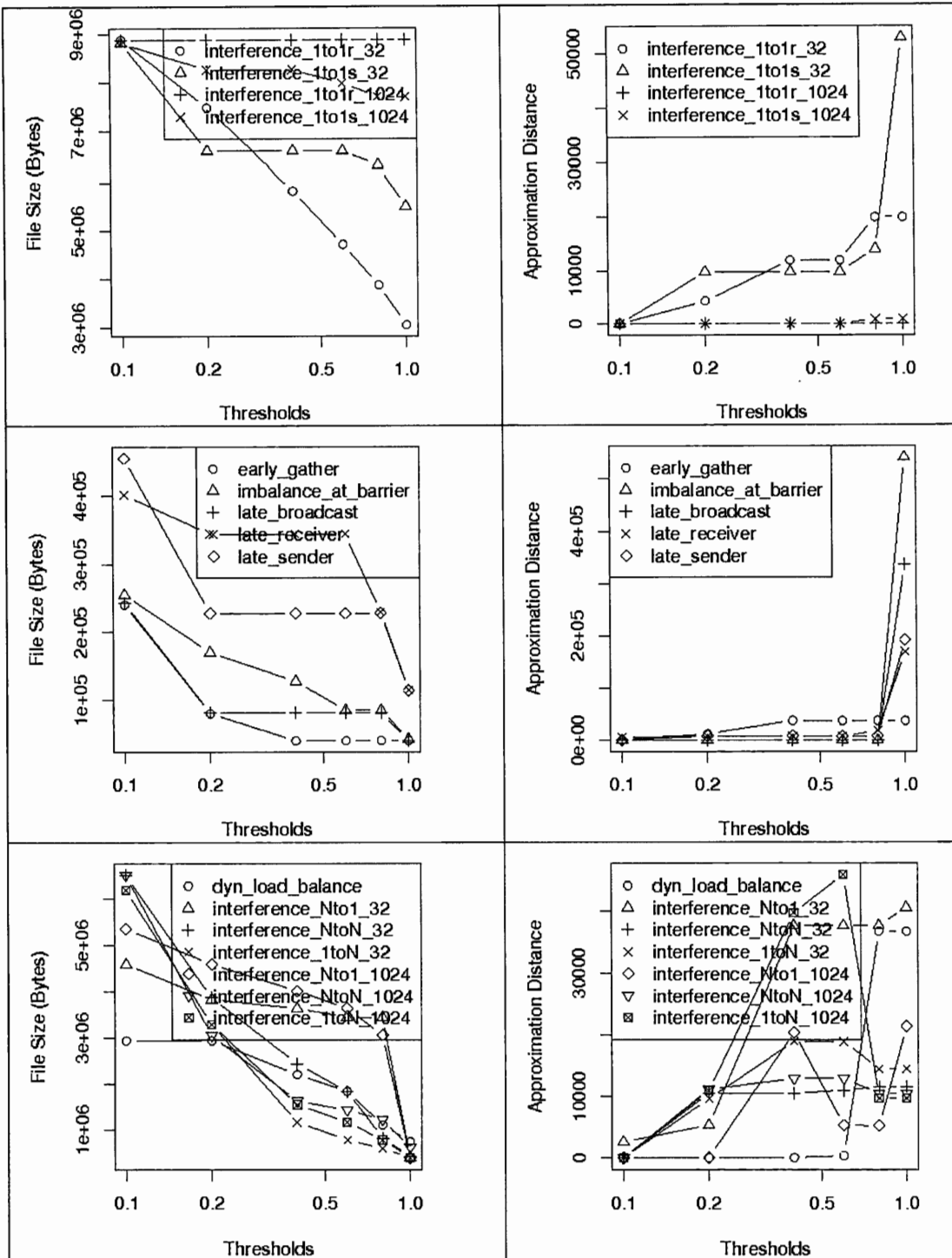


Fig. 33 Inter-process Reduction: File Size and Approximation Distance for Varying Duration Thresholds and Euclidean Distance

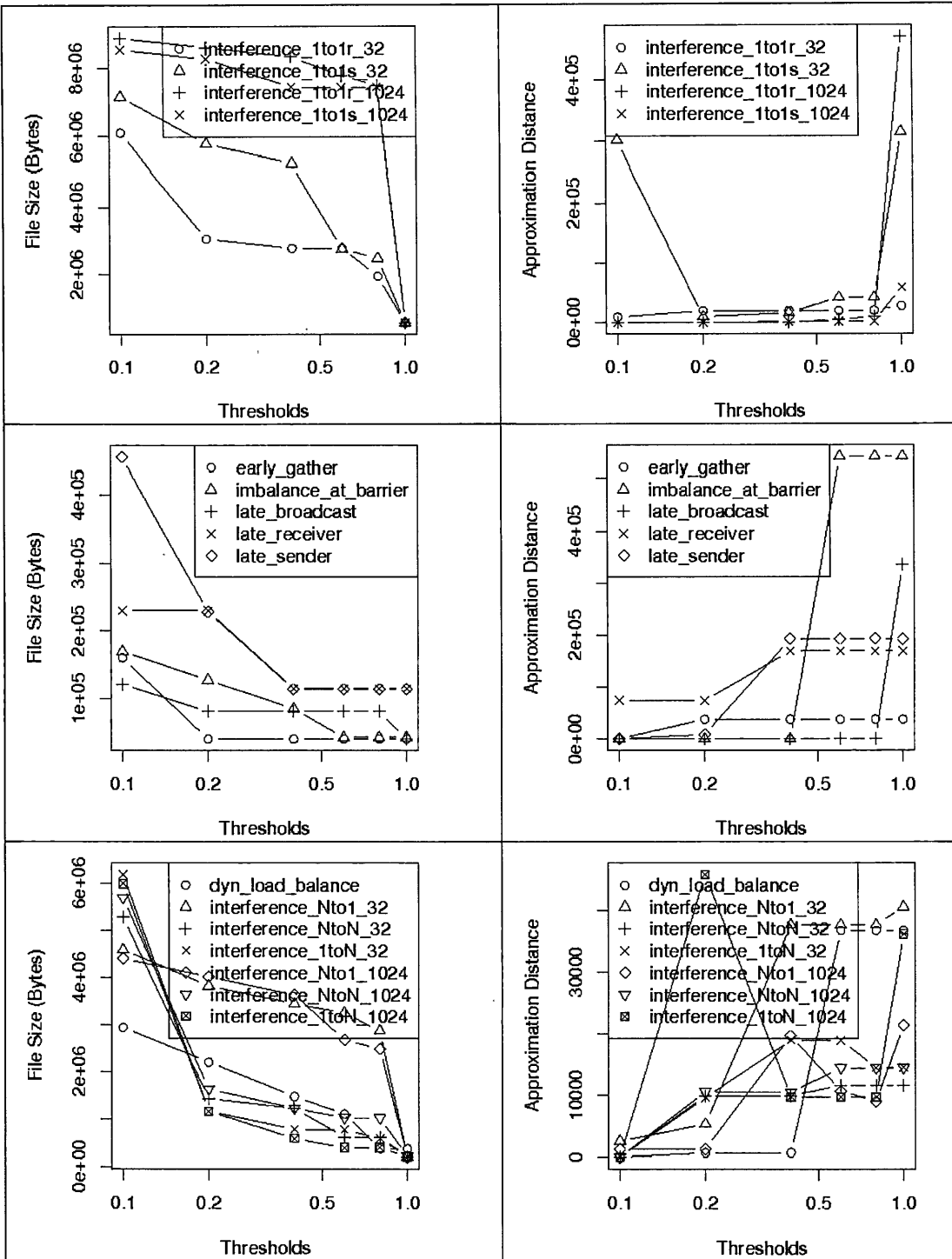


Fig. 34 Inter-process Reduction: File Size and Approximation Distance for Varying Duration Thresholds and Chebyshev Distance

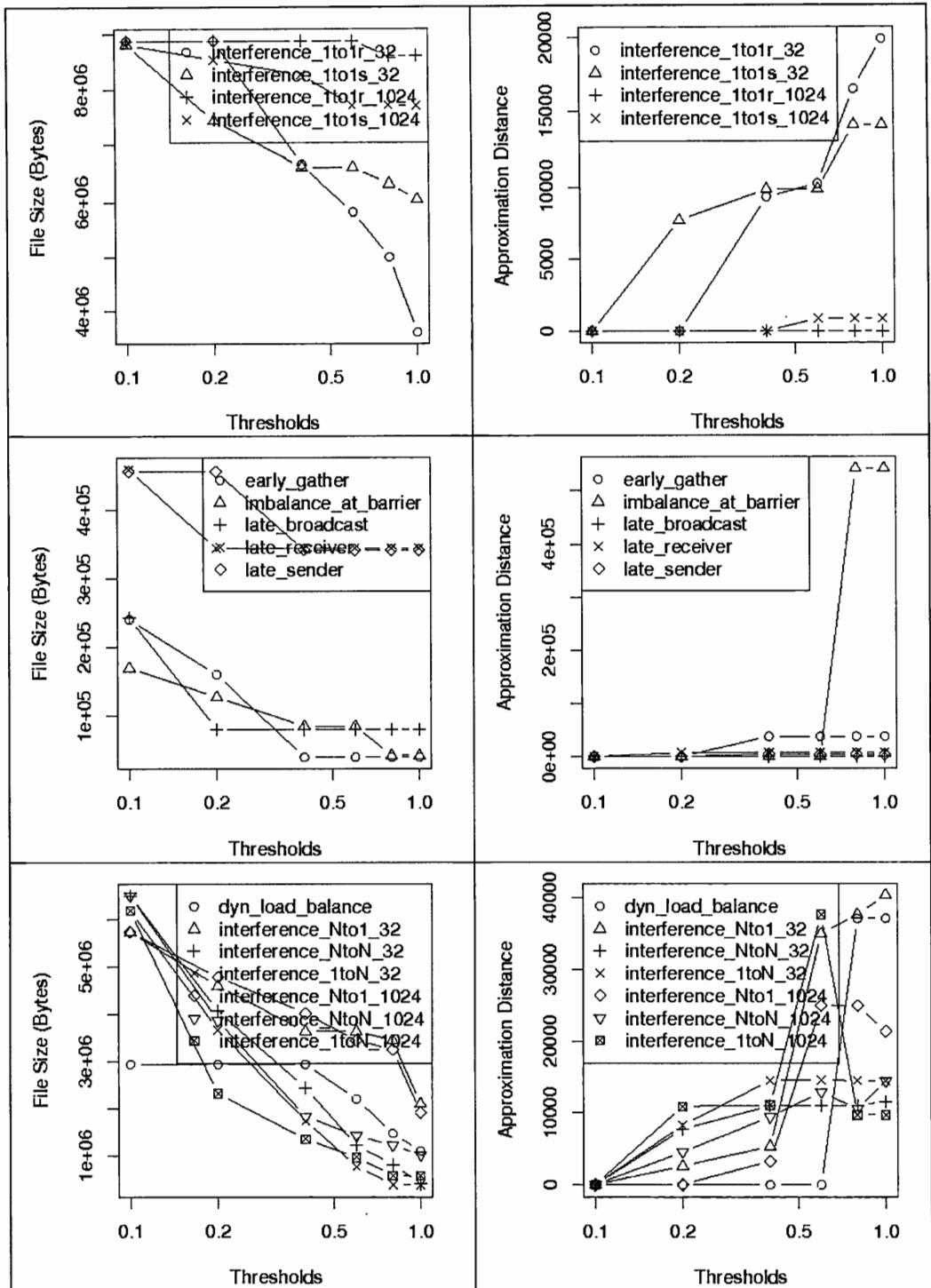


Fig. 35 Inter-process Reduction: File Size and Approximation Distance for Varying Duration Thresholds and Average Wavelet

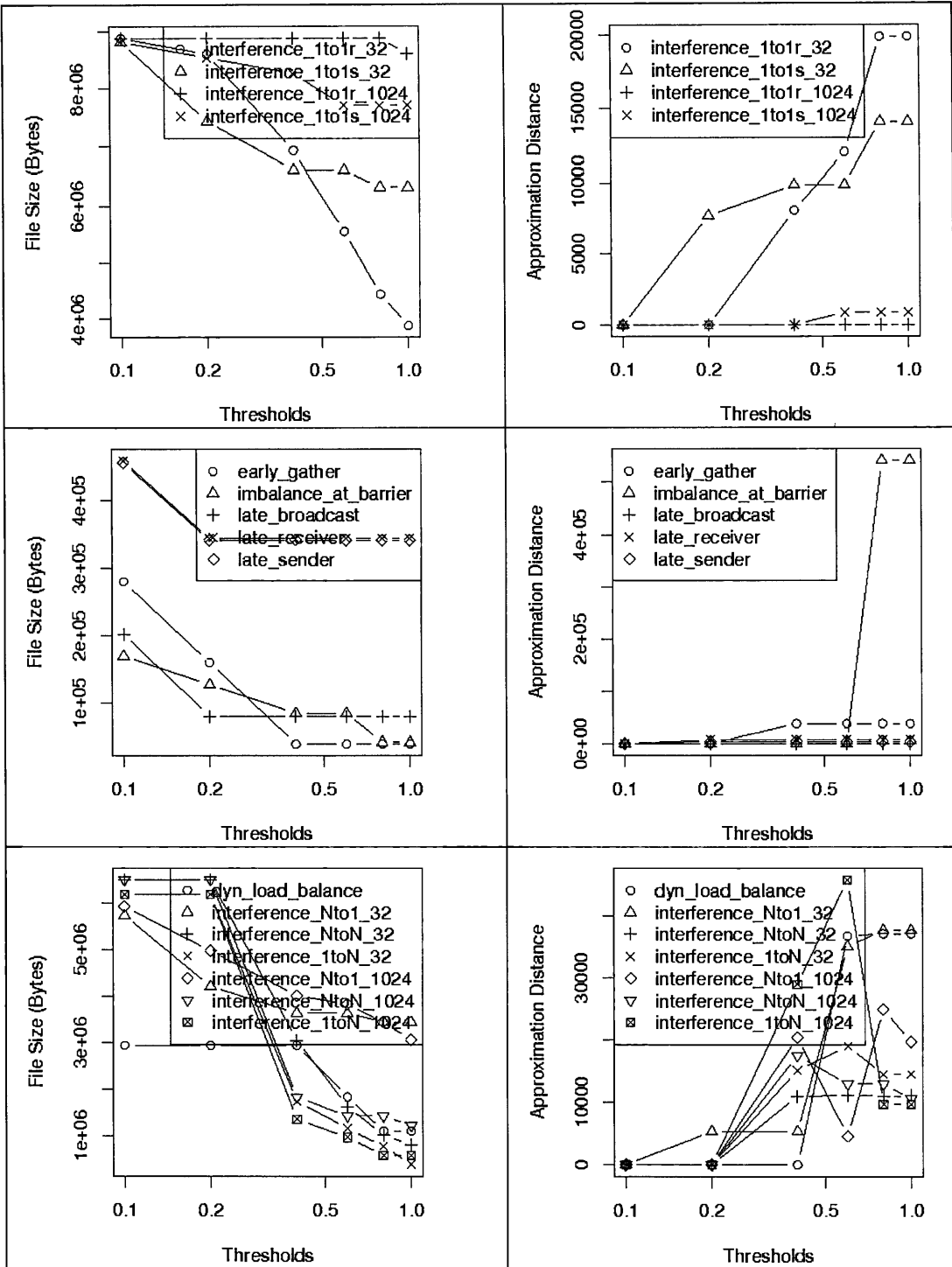


Fig. 36 Inter-process Reduction: File Size and Approximation Distance for Varying Duration Thresholds and Haar Wavelet

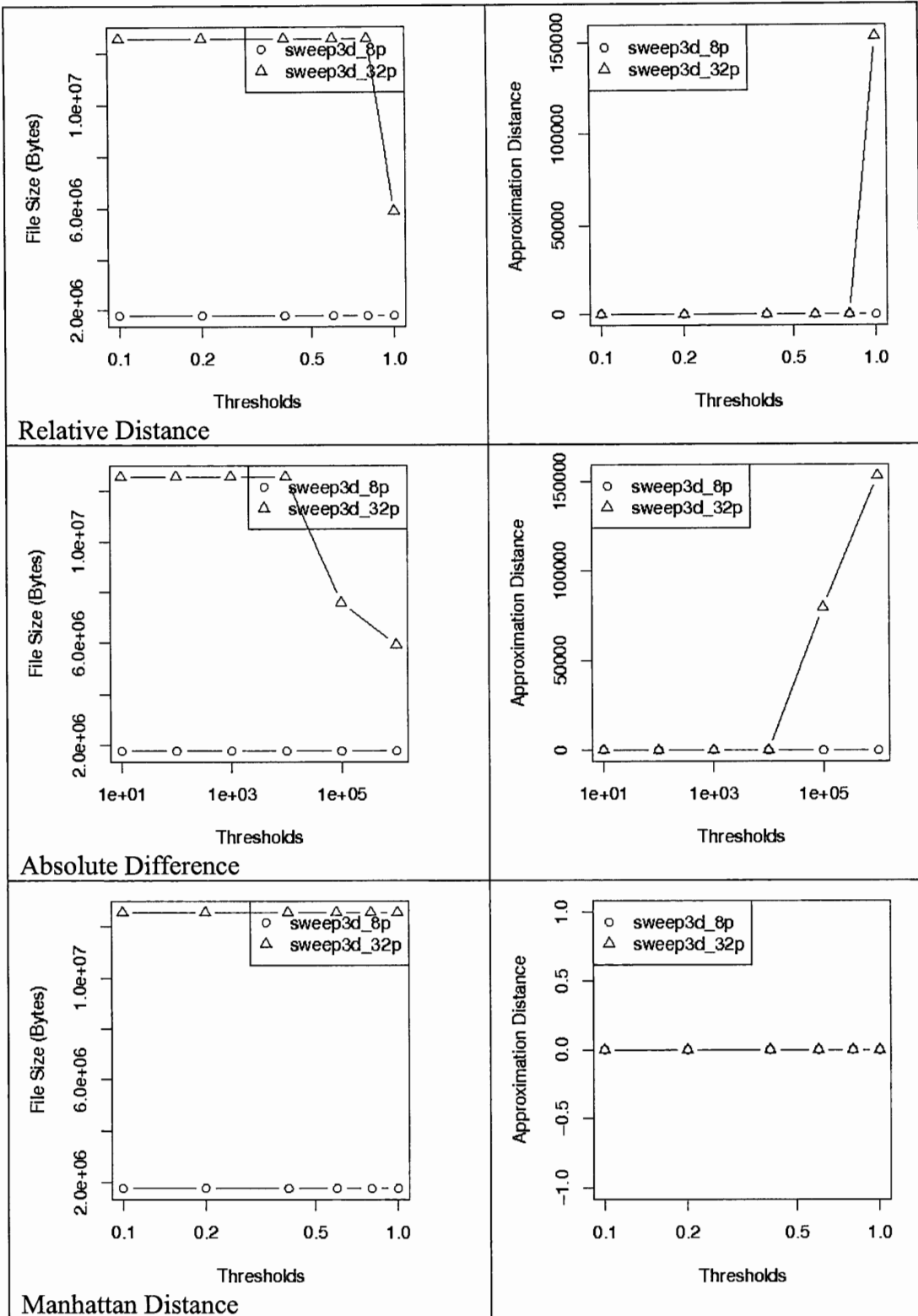


Fig. 37 Inter-process Reduction: File Size and Approximation Distance for Varying Thresholds for Sweep3d and reIDiff, absDiff, and Manhattan

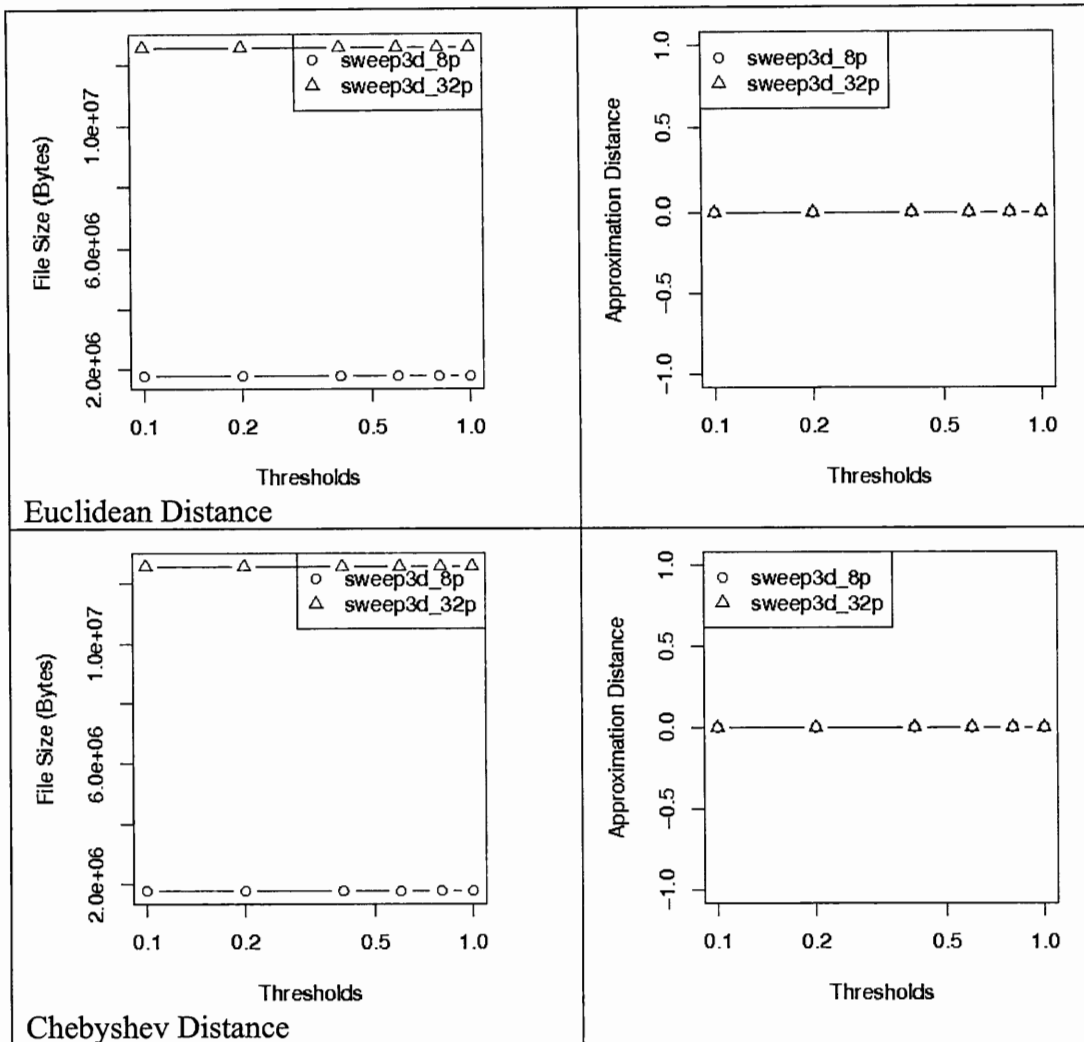


Fig. 38 Inter-process Reduction: File Size and Approximation Distance for Varying Thresholds for Sweep3d and Euclidean and Chebyshev

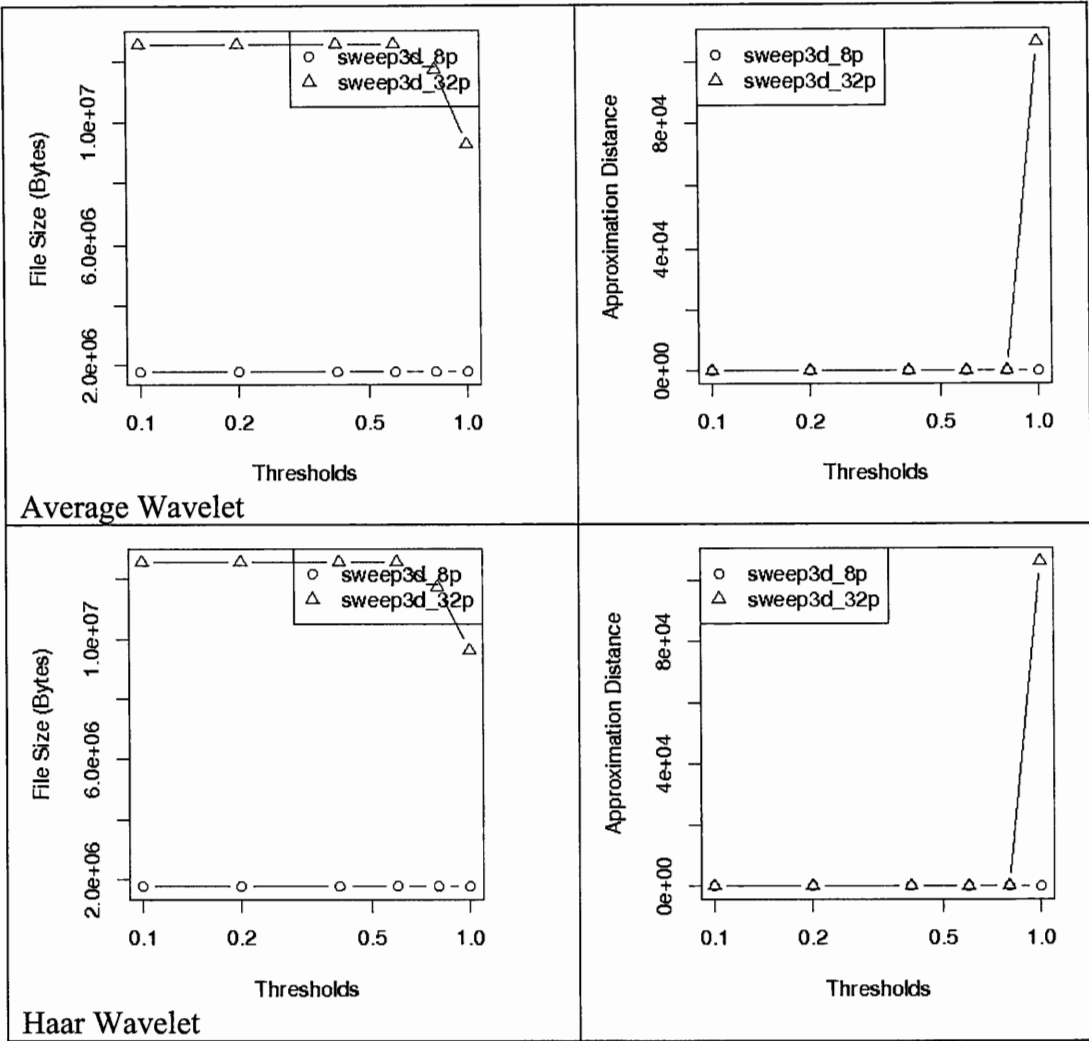


Fig. 39 Inter-process Reduction: File Size and Approximation Distance for Varying Thresholds for Sweep3d and avgWave and haarWave

		MPI Alltoall						do work
relative difference	no loss	EX	EX	EX	EX	NN	EX	
	0.1	EX	EX	EX	EX	NN	EX	
	0.2	EX	EX	EX	EX	NN	EX	
	0.4	EX	EX	EX	EX	NN	EX	
	0.6	EX	EX	EX	EX	NN	EX	
	0.8	EX	EX	EX	EX	NN	EX	
absolute difference	10	EX	MP	CM	CO	NN	EX	
	100	EX	EX	EX	EX	NN	EX	
	1000	EX	EX	EX	EX	NN	EX	
	10000	EX	EX	EX	EX	NN	EX	
	100000	EX	MP	CM	CO	NN	EX	
	1000000	EX	MP	CM	CO	NN	EX	
Manhattan distance	0.1	EX	EX	EX	EX	NN	EX	
	0.2	EX	EX	EX	EX	NN	EX	
	0.4	EX	EX	EX	EX	NN	EX	
	0.6	EX	EX	EX	EX	NN	EX	
	0.8	EX	EX	EX	EX	NN	EX	
	1.0	EX	EX	EX	EX	NN	EX	
Euclidean distance	0.1	EX	EX	EX	EX	NN	EX	
	0.2	EX	EX	EX	EX	NN	EX	
	0.4	EX	EX	EX	EX	NN	EX	
	0.6	EX	EX	EX	EX	NN	EX	
	0.8	EX	MP	CM	CO	NN	EX	
	1.0	EX	MP	CM	CO	NN	EX	
Chebyshev distance	0.1	EX	EX	EX	EX	NN	EX	
	0.2	EX	EX	EX	EX	NN	EX	
	0.4	EX	EX	EX	EX	NN	EX	
	0.6	EX	MP	CM	CO	NN	EX	
	0.8	EX	MP	CM	CO	NN	EX	
	1.0	EX	MP	CM	CO	NN	EX	
Average Wavelet	0.1	EX	EX	EX	EX	NN	EX	
	0.2	EX	EX	EX	EX	NN	EX	
	0.4	EX	EX	EX	EX	NN	EX	
	0.6	EX	EX	EX	EX	NN	EX	
	0.8	EX	MP	CM	CO	NN	EX	
	1.0	EX	MP	CM	CO	NN	EX	
Haar Wavelet	0.1	EX	EX	EX	EX	NN	EX	
	0.2	EX	EX	EX	EX	NN	EX	
	0.4	EX	EX	EX	EX	NN	EX	
	0.6	EX	MP	CM	CO	NN	EX	
	0.8	EX	MP	CM	CO	NN	EX	
	1.0	EX	MP	CM	CO	NN	EX	

Fig. 40 Inter-process Reduction: Retention of Performance Trends with Varying Thresholds for dyn_load_balance

		MPI Gather					do work
relative difference	no loss	EX	MP	CM	CO	ER	EX
	0.1	EX	MP	CM	CO	ER	EX
	0.2	EX	MP	CM	CO	ER	EX
	0.4	EX	MP	CM	CO	ER	EX
	0.6	EX	MP	CM	CO	ER	EX
	0.8	EX	MP	CM	CO	ER	EX
	1.0	EX	MP	CM	CO	ER	EX
absolute difference	10	EX	MP	CM	CO	ER	EX
	100	EX	MP	CM	CO	ER	EX
	1000	EX	MP	CM	CO	ER	EX
	10000	EX	MP	CM	CO	ER	EX
	100000	EX	MP	CM	CO	ER	EX
	1000000	EX	MP	CM	CO	ER	EX
	10000000	EX	MP	CM	CO	ER	EX
Manhattan distance	0.1	EX	MP	CM	CO	ER	EX
	0.2	EX	MP	CM	CO	ER	EX
	0.4	EX	MP	CM	CO	ER	EX
	0.6	EX	MP	CM	CO	ER	EX
	0.8	EX	MP	CM	CO	ER	EX
	1.0	EX	MP	CM	CO	ER	EX
	Euclidean distance	0.1	EX	MP	CM	CO	ER
0.2		EX	MP	CM	CO	ER	EX
0.4		EX	MP	CM	CO	ER	EX
0.6		EX	MP	CM	CO	ER	EX
0.8		EX	MP	CM	CO	ER	EX
1.0		EX	MP	CM	CO	ER	EX
Chebyshev distance		0.1	EX	MP	CM	CO	ER
	0.2	EX	MP	CM	CO	ER	EX
	0.4	EX	MP	CM	CO	ER	EX
	0.6	EX	MP	CM	CO	ER	EX
	0.8	EX	MP	CM	CO	ER	EX
	1.0	EX	MP	CM	CO	ER	EX
	Average Wavelet	0.1	EX	MP	CM	CO	ER
0.2		EX	MP	CM	CO	ER	EX
0.4		EX	MP	CM	CO	ER	EX
0.6		EX	MP	CM	CO	ER	EX
0.8		EX	MP	CM	CO	ER	EX
1.0		EX	MP	CM	CO	ER	EX
Haar Wavelet		0.1	EX	MP	CM	CO	ER
	0.2	EX	MP	CM	CO	ER	EX
	0.4	EX	MP	CM	CO	ER	EX
	0.6	EX	MP	CM	CO	ER	EX
	0.8	EX	MP	CM	CO	ER	EX
	1.0	EX	MP	CM	CO	ER	EX

Fig. 41 Inter-process Reduction: Retention of Performance Trends with Varying Thresholds for early_gather

		MPI Barrier					do work
relative difference	no loss	EX	MP	SN	EA	WB	EX
	0.1	EX	MP	SN	EA	WB	EX
	0.2	EX	MP	SN	EA	WB	EX
	0.4	EX	MP	SN	EA	WB	EX
	0.6	EX	MP	SN	EA	WB	EX
	0.8	EX	MP	SN	EA	WB	EX
	1.0	EX	MP	SN	EA	WB	EX
absolute difference	10	EX	MP	SN	EA	WB	EX
	100	EX	MP	SN	EA	WB	EX
	1000	EX	MP	SN	EA	WB	EX
	10000	EX	MP	SN	EA	WB	EX
	100000	EX	MP	SN	EA	WB	EX
	1000000	EX	MP	SN	EA	WB	EX
Manhattan distance	0.1	EX	MP	SN	EA	WB	EX
	0.2	EX	MP	SN	EA	WB	EX
	0.4	EX	MP	SN	EA	WB	EX
	0.6	EX	MP	SN	EA	WB	EX
	0.8	EX	MP	SN	EA	WB	EX
	1.0	EX	MP	SN	EA	WB	EX
Euclidean distance	0.1	EX	MP	SN	EA	WB	EX
	0.2	EX	MP	SN	EA	WB	EX
	0.4	EX	MP	SN	EA	WB	EX
	0.6	EX	MP	SN	EA	WB	EX
	0.8	EX	MP	SN	EA	WB	EX
	1.0	EX	MP	SN	EA	WB	EX
Chebyshev distance	0.1	EX	MP	SN	EA	WB	EX
	0.2	EX	MP	SN	EA	WB	EX
	0.4	EX	MP	SN	EA	WB	EX
	0.6	EX	MP	SN	EA	WB	EX
	0.8	EX	MP	SN	EA	WB	EX
	1.0	EX	MP	SN	EA	WB	EX
Average Wavelet	0.1	EX	MP	SN	EA	WB	EX
	0.2	EX	MP	SN	EA	WB	EX
	0.4	EX	MP	SN	EA	WB	EX
	0.6	EX	MP	SN	EA	WB	EX
	0.8	EX	MP	SN	EA	WB	EX
	1.0	EX	MP	SN	EA	WB	EX
Haar Wavelet	0.1	EX	MP	SN	EA	WB	EX
	0.2	EX	MP	SN	EA	WB	EX
	0.4	EX	MP	SN	EA	WB	EX
	0.6	EX	MP	SN	EA	WB	EX
	0.8	EX	MP	SN	EA	WB	EX
	1.0	EX	MP	SN	EA	WB	EX

Fig. 42 Inter-process Reduction: Retention of Performance Trends with Varying Thresholds for imbalance_at_barrier

		MPI Recv					do work
relative difference	no loss	EX	MP	CM	P2	LS	EX
	0.1	EX	MP	CM	P2	LS	EX
	0.2	EX	MP	CM	P2	LS	EX
	0.4	EX	MP	CM	P2	LS	EX
	0.6	EX	MP	CM	P2	LS	EX
	0.8	EX	MP	CM	P2	LS	EX
	1.0	EX	MP	CM	P2	LS	EX
absolute difference	10	EX	MP	CM	P2	LS	EX
	100	EX	MP	CM	P2	LS	EX
	1000	EX	MP	CM	P2	LS	EX
	10000	EX	MP	CM	P2	LS	EX
	100000	EX	MP	CM	P2	LS	EX
	1000000	EX	MP	CM	P2	LS	EX
Manhattan distance	0.1	EX	MP	CM	P2	LS	EX
	0.2	EX	MP	CM	P2	LS	EX
	0.4	EX	MP	CM	P2	LS	EX
	0.6	EX	MP	CM	P2	LS	EX
	0.8	EX	MP	CM	P2	LS	EX
	1.0	EX	MP	CM	P2	LS	EX
Euclidean distance	0.1	EX	MP	CM	P2	LS	EX
	0.2	EX	MP	CM	P2	LS	EX
	0.4	EX	MP	CM	P2	LS	EX
	0.6	EX	MP	CM	P2	LS	EX
	0.8	EX	MP	CM	P2	LS	EX
	1.0	EX	MP	CM	P2	LS	EX
Chebyshev distance	0.1	EX	MP	CM	P2	LS	EX
	0.2	EX	MP	CM	P2	LS	EX
	0.4	EX	MP	CM	P2	LS	EX
	0.6	EX	MP	CM	P2	LS	EX
	0.8	EX	MP	CM	P2	LS	EX
	1.0	EX	MP	CM	P2	LS	EX
Average Wavelet	0.1	EX	MP	CM	P2	LS	EX
	0.2	EX	MP	CM	P2	LS	EX
	0.4	EX	MP	CM	P2	LS	EX
	0.6	EX	MP	CM	P2	LS	EX
	0.8	EX	MP	CM	P2	LS	EX
	1.0	EX	MP	CM	P2	LS	EX
Haar Wavelet	0.1	EX	MP	CM	P2	LS	EX
	0.2	EX	MP	CM	P2	LS	EX
	0.4	EX	MP	CM	P2	LS	EX
	0.6	EX	MP	CM	P2	LS	EX
	0.8	EX	MP	CM	P2	LS	EX
	1.0	EX	MP	CM	P2	LS	EX

Fig. 45 Inter-process Reduction: Retention of Performance Trends with Varying Thresholds for late_sender

		MPI Gather					do work
relative difference	no loss	EX	MP	CM	CO	ER	EX
	0.1	EX	MP	CM	CO	ER	EX
	0.2	EX	MP	CM	CO	ER	EX
	0.4	EX	MP	CM	CO	ER	EX
	0.6	EX	MP	CM	CO	ER	EX
	0.8	EX	MP	CM	CO	ER	EX
absolute difference	1.0	EX	MP	CM	CO	ER	EX
	10	EX	MP	CM	CO	ER	EX
	100	EX	MP	CM	CO	ER	EX
	1000	EX	MP	CM	CO	ER	EX
	10000	EX	MP	CM	CO	ER	EX
Manhattan distance	100000	EX	MP	CM	CO	ER	EX
	1000000	EX	MP	CM	CO	ER	EX
	0.1	EX	MP	CM	CO	ER	EX
	0.2	EX	MP	CM	CO	ER	EX
	0.4	EX	MP	CM	CO	ER	EX
	0.6	EX	MP	CM	CO	ER	EX
Euclidean distance	0.8	EX	MP	CM	CO	ER	EX
	1.0	EX	MP	CM	CO	ER	EX
	0.1	EX	MP	CM	CO	ER	EX
	0.2	EX	MP	CM	CO	ER	EX
	0.4	EX	MP	CM	CO	ER	EX
Chebyshev distance	0.6	EX	MP	CM	CO	ER	EX
	0.8	EX	MP	CM	CO	ER	EX
	1.0	EX	MP	CM	CO	ER	EX
	0.1	EX	MP	CM	CO	ER	EX
	0.2	EX	MP	CM	CO	ER	EX
Average Wavelet	0.4	EX	MP	CM	CO	ER	EX
	0.6	EX	MP	CM	CO	ER	EX
	0.8	EX	MP	CM	CO	ER	EX
	1.0	EX	MP	CM	CO	ER	EX
	Haar Wavelet	0.1	EX	MP	CM	CO	ER
0.2		EX	MP	CM	CO	ER	EX
0.4		EX	MP	CM	CO	ER	EX
0.6		EX	MP	CM	CO	ER	EX
0.8		EX	MP	CM	CO	ER	EX

Fig. 46 Inter-process Reduction: Retention of Performance Trends with Varying Thresholds for Nto1_32

		MPI Barrier						do_work
	no loss	EX	MP	SN	BA	WR	BC	EX
relative difference	0.1	EX	MP	SN	BA	WR	BC	EX
	0.2	EX	MP	SN	BA	WR	BC	EX
	0.4	EX	MP	SN	BA	WR	BC	EX
	0.6	EX	MP	SN	BA	WR	BC	EX
	0.8	EX	MP	SN	BA	WR	BC	EX
	1.0	EX	MP	SN	BA	WR	BC	EX
absolute difference	10	EX	MP	SN	BA	WR	BC	EX
	100	EX	MP	SN	BA	WR	BC	EX
	1000	EX	MP	SN	BA	WR	BC	EX
	10000	EX	MP	SN	BA	WR	BC	EX
	100000	EX	MP	SN	BA	WR	BC	EX
	1000000	EX	MP	SN	BA	WR	BC	EX
Manhattan distance	0.1	EX	MP	SN	BA	WR	BC	EX
	0.2	EX	MP	SN	BA	WR	BC	EX
	0.4	EX	MP	SN	BA	WR	BC	EX
	0.6	EX	MP	SN	BA	WR	BC	EX
	0.8	EX	MP	SN	BA	WR	BC	EX
	1.0	EX	MP	SN	BA	WR	BC	EX
Euclidean distance	0.1	EX	MP	SN	BA	WR	BC	EX
	0.2	EX	MP	SN	BA	WR	BC	EX
	0.4	EX	MP	SN	BA	WR	BC	EX
	0.6	EX	MP	SN	BA	WR	BC	EX
	0.8	EX	MP	SN	BA	WR	BC	EX
	1.0	EX	MP	SN	BA	WR	BC	EX
Chebyshev distance	0.1	EX	MP	SN	BA	WR	BC	EX
	0.2	EX	MP	SN	BA	WR	BC	EX
	0.4	EX	MP	SN	BA	WR	BC	EX
	0.6	EX	MP	SN	BA	WR	BC	EX
	0.8	EX	MP	SN	BA	WR	BC	EX
	1.0	EX	MP	SN	BA	WR	BC	EX
Average Wavelet	0.1	EX	MP	SN	BA	WR	BC	EX
	0.2	EX	MP	SN	BA	WR	BC	EX
	0.4	EX	MP	SN	BA	WR	BC	EX
	0.6	EX	MP	SN	BA	WR	BC	EX
	0.8	EX	MP	SN	BA	WR	BC	EX
	1.0	EX	MP	SN	BA	WR	BC	EX
Haar Wavelet	0.1	EX	MP	SN	BA	WR	BC	EX
	0.2	EX	MP	SN	BA	WR	BC	EX
	0.4	EX	MP	SN	BA	WR	BC	EX
	0.6	EX	MP	SN	BA	WR	BC	EX
	0.8	EX	MP	SN	BA	WR	BC	EX
	1.0	EX	MP	SN	BA	WR	BC	EX

Fig. 47 Inter-process Reduction: Retention of Performance Trends with Varying Thresholds for NtoN_32

		MPI Bcast					do_work
relative difference	no loss	EX	MP	CM	CO	LB	EX
	0.1	EX	MP	CM	CO	LB	EX
	0.2	EX	MP	CM	CO	LB	EX
	0.4	EX	MP	CM	CO	LB	EX
	0.6	EX	MP	CM	CO	LB	EX
	0.8	EX	MP	CM	CO	LB	EX
absolute difference	1.0	EX	MP	CM	CO	LB	EX
	10	EX	MP	CM	CO	LB	EX
	100	EX	MP	CM	CO	LB	EX
	1000	EX	MP	CM	CO	LB	EX
	10000	EX	MP	CM	CO	LB	EX
	100000	EX	MP	CM	CO	LB	EX
Manhattan distance	1000000	EX	MP	CM	CO	LB	EX
	0.1	EX	MP	CM	CO	LB	EX
	0.2	EX	MP	CM	CO	LB	EX
	0.4	EX	MP	CM	CO	LB	EX
	0.6	EX	MP	CM	CO	LB	EX
	0.8	EX	MP	CM	CO	LB	EX
Euclidean distance	1.0	EX	MP	CM	CO	LB	EX
	0.1	EX	MP	CM	CO	LB	EX
	0.2	EX	MP	CM	CO	LB	EX
	0.4	EX	MP	CM	CO	LB	EX
	0.6	EX	MP	CM	CO	LB	EX
	0.8	EX	MP	CM	CO	LB	EX
Chebyshev distance	1.0	EX	MP	CM	CO	LB	EX
	0.1	EX	MP	CM	CO	LB	EX
	0.2	EX	MP	CM	CO	LB	EX
	0.4	EX	MP	CM	CO	LB	EX
	0.6	EX	MP	CM	CO	LB	EX
	0.8	EX	MP	CM	CO	LB	EX
Average Wavelet	1.0	EX	MP	CM	CO	LB	EX
	0.1	EX	MP	CM	CO	LB	EX
	0.2	EX	MP	CM	CO	LB	EX
	0.4	EX	MP	CM	CO	LB	EX
	0.6	EX	MP	CM	CO	LB	EX
	0.8	EX	MP	CM	CO	LB	EX
Haar Wavelet	1.0	EX	MP	CM	CO	LB	EX
	0.1	EX	MP	CM	CO	LB	EX
	0.2	EX	MP	CM	CO	LB	EX
	0.4	EX	MP	CM	CO	LB	EX
	0.6	EX	MP	CM	CO	LB	EX
	0.8	EX	MP	CM	CO	LB	EX

Fig. 48 Inter-process Reduction: Retention of Performance Trends with Varying Thresholds for 1toN_32

		MPI Ssend					MPI Recv					do work
	no loss	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
Relative Difference	0.1	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.2	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.4	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.6	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.8	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	1.0	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
Absolute Difference	10	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	100	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	1000	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	10000	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	100000	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	1000000	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
Manhattan distance	0.1	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.2	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.4	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.6	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.8	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	1.0	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
Euclidean distance	0.1	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.2	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.4	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.6	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.8	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	1.0	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
Chebyshev distance	0.1	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.2	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.4	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.6	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.8	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	1.0	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
Average Wavelet	0.1	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.2	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.4	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.6	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.8	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	1.0	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
Haar Wavelet	0.1	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.2	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.4	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.6	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.8	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	1.0	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX

Fig. 49 Inter-process Reduction: Retention of Performance Trends with Varying Thresholds for 1to1r_32

		MPI Ssend					MPI Recv					do work
	no loss	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
Relative Difference	0.1	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.2	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.4	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.6	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.8	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	1.0	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
Absolute Difference	10	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	100	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	1000	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	10000	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	100000	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	1000000	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
Manhattan distance	0.1	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.2	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.4	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.6	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.8	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	1.0	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
Euclidean distance	0.1	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.2	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.4	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.6	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.8	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	1.0	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
Chebyshev distance	0.1	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.2	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.4	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.6	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.8	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	1.0	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
Average Wavelet	0.1	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.2	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.4	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.6	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.8	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	1.0	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
Haar Wavelet	0.1	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.2	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.4	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.6	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.8	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	1.0	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX

Fig. 50 Inter-process Reduction: Retention of Performance Trends with Varying Thresholds for 1to1s_32

		MPI Gather					do work
relative difference	no loss	EX	MP	CM	CO	ER	FX
	0.1	EX	MP	CM	CO	ER	FX
	0.2	EX	MP	CM	CO	ER	FX
	0.4	EX	MP	CM	CO	ER	FX
	0.6	EX	MP	CM	CO	ER	FX
	0.8	EX	MP	CM	CO	ER	FX
	1.0	EX	MP	CM	CO	ER	FX
absolute difference	10	EX	MP	CM	CO	ER	FX
	100	EX	MP	CM	CO	ER	FX
	1000	EX	MP	CM	CO	ER	FX
	10000	EX	MP	CM	CO	ER	FX
	100000	EX	MP	CM	CO	ER	FX
	1000000	EX	MP	CM	CO	ER	FX
Manhattan distance	0.1	EX	MP	CM	CO	ER	FX
	0.2	EX	MP	CM	CO	ER	FX
	0.4	EX	MP	CM	CO	ER	FX
	0.6	EX	MP	CM	CO	ER	FX
	0.8	EX	MP	CM	CO	ER	FX
	1.0	EX	MP	CM	CO	ER	FX
Euclidean distance	0.1	EX	MP	CM	CO	ER	FX
	0.2	EX	MP	CM	CO	ER	FX
	0.4	EX	MP	CM	CO	ER	FX
	0.6	EX	MP	CM	CO	ER	FX
	0.8	EX	MP	CM	CO	ER	FX
	1.0	EX	MP	CM	CO	ER	FX
Chebyshev distance	0.1	EX	MP	CM	CO	ER	FX
	0.2	EX	MP	CM	CO	ER	FX
	0.4	EX	MP	CM	CO	ER	FX
	0.6	EX	MP	CM	CO	ER	FX
	0.8	EX	MP	CM	CO	ER	FX
	1.0	EX	MP	CM	CO	ER	FX
Average Wavelet	0.1	EX	MP	CM	CO	ER	FX
	0.2	EX	MP	CM	CO	ER	FX
	0.4	EX	MP	CM	CO	ER	FX
	0.6	EX	MP	CM	CO	ER	FX
	0.8	EX	MP	CM	CO	ER	FX
	1.0	EX	MP	CM	CO	ER	FX
Haar Wavelet	0.1	EX	MP	CM	CO	ER	FX
	0.2	EX	MP	CM	CO	ER	FX
	0.4	EX	MP	CM	CO	ER	FX
	0.6	EX	MP	CM	CO	ER	FX
	0.8	EX	MP	CM	CO	ER	FX
	1.0	EX	MP	CM	CO	ER	FX

Fig. 51 Inter-process Reduction: Retention of Performance Trends with Varying Thresholds for Nto1_1024

		MPI Barrier						do work
relative difference	no loss	EX	MP	SN	RA	WB	BC	EX
	0.1	EX	MP	SN	RA	WB	BC	EX
	0.2	EX	MP	SN	RA	WB	BC	EX
	0.4	EX	MP	SN	RA	WB	BC	EX
	0.6	EX	MP	SN	RA	WB	BC	EX
	0.8	EX	MP	SN	RA	WB	BC	EX
absolute difference	1.0	EX	MP	SN	RA	WB	BC	EX
	10	EX	MP	SN	RA	WB	BC	EX
	100	EX	MP	SN	RA	WB	BC	EX
	1000	EX	MP	SN	RA	WB	BC	EX
	10000	EX	MP	SN	RA	WB	BC	EX
	100000	EX	MP	SN	RA	WB	BC	EX
Manhattan distance	1000000	EX	MP	SN	RA	WB	BC	EX
	0.1	EX	MP	SN	RA	WB	BC	EX
	0.2	EX	MP	SN	RA	WB	BC	EX
	0.4	EX	MP	SN	RA	WB	BC	EX
	0.6	EX	MP	SN	RA	WB	BC	EX
	0.8	EX	MP	SN	RA	WB	BC	EX
Euclidean distance	1.0	EX	MP	SN	RA	WB	BC	EX
	0.1	EX	MP	SN	RA	WB	BC	EX
	0.2	EX	MP	SN	RA	WB	BC	EX
	0.4	EX	MP	SN	RA	WB	BC	EX
	0.6	EX	MP	SN	RA	WB	BC	EX
	0.8	EX	MP	SN	RA	WB	BC	EX
Chebyshev distance	1.0	EX	MP	SN	RA	WB	BC	EX
	0.1	EX	MP	SN	RA	WB	BC	EX
	0.2	EX	MP	SN	RA	WB	BC	EX
	0.4	EX	MP	SN	RA	WB	BC	EX
	0.6	EX	MP	SN	RA	WB	BC	EX
	0.8	EX	MP	SN	RA	WB	BC	EX
Average Wavelet	1.0	EX	MP	SN	RA	WB	BC	EX
	0.1	EX	MP	SN	RA	WB	BC	EX
	0.2	EX	MP	SN	RA	WB	BC	EX
	0.4	EX	MP	SN	RA	WB	BC	EX
	0.6	EX	MP	SN	RA	WB	BC	EX
	0.8	EX	MP	SN	RA	WB	BC	EX
Haar Wavelet	1.0	EX	MP	SN	RA	WB	BC	EX
	0.1	EX	MP	SN	RA	WB	BC	EX
	0.2	EX	MP	SN	RA	WB	BC	EX
	0.4	EX	MP	SN	RA	WB	BC	EX
	0.6	EX	MP	SN	RA	WB	BC	EX
	0.8	EX	MP	SN	RA	WB	BC	EX

Fig. 52 Inter-process Reduction: Retention of Performance Trends with Varying Thresholds for NtoN_1024

		MPI Bcast					do work
relative difference	no loss	EX	MP	CM	CO	LB	
	0.1	EX	MP	CM	CO	LB	
	0.2	EX	MP	CM	CO	LB	
	0.4	EX	MP	CM	CO	LB	
	0.6	EX	MP	CM	CO	LB	
	0.8	EX	MP	CM	CO	LB	
	1.0	EX	MP	CM	CO	LB	
absolute difference	10	EX	MP	CM	CO	LB	
	100	EX	MP	CM	CO	LB	
	1000	EX	MP	CM	CO	LB	
	10000	EX	MP	CM	CO	LB	
	100000	EX	MP	CM	CO	LB	
	1000000	EX	MP	CM	CO	LB	
Manhattan distance	0.1	EX	MP	CM	CO	LB	
	0.2	EX	MP	CM	CO	LB	
	0.4	EX	MP	CM	CO	LB	
	0.6	EX	MP	CM	CO	LB	
	0.8	EX	MP	CM	CO	LB	
	1.0	EX	MP	CM	CO	LB	
Euclidean distance	0.1	EX	MP	CM	CO	LB	
	0.2	EX	MP	CM	CO	LB	
	0.4	EX	MP	CM	CO	LB	
	0.6	EX	MP	CM	CO	LB	
	0.8	EX	MP	CM	CO	LB	
	1.0	EX	MP	CM	CO	LB	
Chebyshev distance	0.1	EX	MP	CM	CO	LB	
	0.2	EX	MP	CM	CO	LB	
	0.4	EX	MP	CM	CO	LB	
	0.6	EX	MP	CM	CO	LB	
	0.8	EX	MP	CM	CO	LB	
	1.0	EX	MP	CM	CO	LB	
Average Wavelet	0.1	EX	MP	CM	CO	LB	
	0.2	EX	MP	CM	CO	LB	
	0.4	EX	MP	CM	CO	LB	
	0.6	EX	MP	CM	CO	LB	
	0.8	EX	MP	CM	CO	LB	
	1.0	EX	MP	CM	CO	LB	
Haar Wavelet	0.1	EX	MP	CM	CO	LB	
	0.2	EX	MP	CM	CO	LB	
	0.4	EX	MP	CM	CO	LB	
	0.6	EX	MP	CM	CO	LB	
	0.8	EX	MP	CM	CO	LB	
	1.0	EX	MP	CM	CO	LB	

Fig. 53 Inter-process Reduction: Retention of Performance Trends with Varying Thresholds for 1toN_1024

		MPI Ssend					MPI Recv					do work
	no loss	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
Relative Difference	0.1	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.2	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.4	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.6	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.8	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	1.0	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
Absolute Difference	10	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	100	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	1000	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	10000	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	100000	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	1000000	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
Manhattan distance	0.1	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.2	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.4	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.6	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.8	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	1.0	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
Euclidean distance	0.1	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.2	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.4	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.6	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.8	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	1.0	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
Chebyshev distance	0.1	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.2	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.4	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.6	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.8	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	1.0	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
Average Wavelet	0.1	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.2	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.4	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.6	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.8	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	1.0	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
Haar Wavelet	0.1	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.2	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.4	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.6	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.8	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	1.0	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX

Fig. 54 Inter-process Reduction: Retention of Performance Trends with Varying Thresholds for 1to1r_1024

		MPI Ssend					MPI Recv					do work
	no loss	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
Relative Difference	0.1	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.2	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.4	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.6	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.8	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	1.0	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
Absolute Difference	10	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	100	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	1000	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	10000	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	100000	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	1000000	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
Manhattan distance	0.1	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.2	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.4	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.6	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.8	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	1.0	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
Euclidean distance	0.1	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.2	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.4	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.6	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.8	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	1.0	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
Chebyshev distance	0.1	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.2	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.4	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.6	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.8	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	1.0	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
Average Wavelet	0.1	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.2	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.4	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.6	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.8	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	1.0	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
Haar Wavelet	0.1	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.2	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.4	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.6	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	0.8	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
	1.0	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX

Fig. 55 Inter-process Reduction: Retention of Performance Trends with Varying Thresholds for 1to1s_1024

		pmpl recv						sweep
no loss		EX	MP	CM	P2	LS	MO	EX
relative difference	0.1	EX	MP	CM	P2	LS	MO	EX
	0.2	EX	MP	CM	P2	LS	MO	EX
	0.4	EX	MP	CM	P2	LS	MO	EX
	0.6	EX	MP	CM	P2	LS	MO	EX
	0.8	EX	MP	CM	P2	LS	MO	EX
absolute difference	10	EX	MP	CM	P2	LS	MO	EX
	100	EX	MP	CM	P2	LS	MO	EX
	1000	EX	MP	CM	P2	LS	MO	EX
	10000	EX	MP	CM	P2	LS	MO	EX
	100000	EX	MP	CM	P2	LS	MO	EX
Manhattan distance	0.1	EX	MP	CM	P2	LS	MO	EX
	0.2	EX	MP	CM	P2	LS	MO	EX
	0.4	EX	MP	CM	P2	LS	MO	EX
	0.6	EX	MP	CM	P2	LS	MO	EX
	0.8	EX	MP	CM	P2	LS	MO	EX
Euclidean distance	0.1	EX	MP	CM	P2	LS	MO	EX
	0.2	EX	MP	CM	P2	LS	MO	EX
	0.4	EX	MP	CM	P2	LS	MO	EX
	0.6	EX	MP	CM	P2	LS	MO	EX
	0.8	EX	MP	CM	P2	LS	MO	EX
Chebyshev distance	0.1	EX	MP	CM	P2	LS	MO	EX
	0.2	EX	MP	CM	P2	LS	MO	EX
	0.4	EX	MP	CM	P2	LS	MO	EX
	0.6	EX	MP	CM	P2	LS	MO	EX
	0.8	EX	MP	CM	P2	LS	MO	EX
Average Wavelet	0.1	EX	MP	CM	P2	LS	MO	EX
	0.2	EX	MP	CM	P2	LS	MO	EX
	0.4	EX	MP	CM	P2	LS	MO	EX
	0.6	EX	MP	CM	P2	LS	MO	EX
	0.8	EX	MP	CM	P2	LS	MO	EX
Haar Wavelet	0.1	EX	MP	CM	P2	LS	MO	EX
	0.2	EX	MP	CM	P2	LS	MO	EX
	0.4	EX	MP	CM	P2	LS	MO	EX
	0.6	EX	MP	CM	P2	LS	MO	EX
	0.8	EX	MP	CM	P2	LS	MO	EX

Fig. 57 Inter-process Reduction: Retention of Performance Trends with Varying Thresholds for sweep3d_32p

	MPI Alltoall					do work
no loss	EX	MP	CM	CO	NN	EX
relDiff	EX	MP	CM	CO	NN	EX
absDiff	EX	MP	CM	CO	NN	EX
Manhattan	EX	MP	CM	CO	NN	EX
Euclidean	EX	MP	CM	CO	NN	EX
Chebyshev	EX	MP	CM	CO	NN	EX
iter_k	EX	MP	CM	CO	NN	EX
iter_avg	EX	MP	CM	CO	NN	EX
avgWave	EX	MP	CM	CO	NN	EX
haarWave	EX	MP	CM	CO	NN	EX

Fig. 58 Combined Reduction: Retention of Performance Trends with Default Thresholds for `dyn_load_balance`

	MPI Gather					do work
no loss	EX	MP	CM	CO	ER	EX
relDiff	EX	MP	CM	CO	ER	EX
absDiff	EX	MP	CM	CO	ER	EX
Manhattan	EX	MP	CM	CO	ER	EX
Euclidean	EX	MP	CM	CO	ER	EX
Chebyshev	EX	MP	CM	CO	ER	EX
iter_k	EX	MP	CM	CO	ER	EX
iter_avg	EX	MP	CM	CO	ER	EX
avgWave	EX	MP	CM	CO	ER	EX
haarWave	EX	MP	CM	CO	ER	EX

Fig. 59 Combined Reduction: Retention of Performance Trends with Default Thresholds for `early_gather`

	MPI Barrier					do work
no loss	EX	MP	SN	BA	WB	EX
relDiff	EX	MP	SN	BA	WB	EX
absDiff	EX	MP	SN	BA	WB	EX
Manhattan	EX	MP	SN	BA	WB	EX
Euclidean	EX	MP	SN	BA	WB	EX
Chebyshev	EX	MP	SN	BA	WB	EX
iter_k	EX	MP	SN	BA	WB	EX
iter_avg	EX	MP	SN	BA	WB	EX
avgWave	EX	MP	SN	BA	WB	EX
haarWave	EX	MP	SN	BA	WB	EX

Fig. 60 Combined Reduction: Retention of Performance Trends with Default Thresholds for `imbalance_at_barrier`

	MPI Bcast					do work
no loss	EX	MP	CM	CO	LB	EX
relDiff	EX	MP	CM	CO	LB	EX
absDiff	EX	MP	CM	CO	LB	EX
Manhattan	EX	MP	CM	CO	LB	EX
Euclidean	EX	MP	CM	CO	LB	EX
Chebyshev	EX	MP	CM	CO	LB	EX
iter_k	EX	MP	CM	CO	LB	EX
iter_avg	EX	MP	CM	CO	LB	EX
avgWave	EX	MP	CM	CO	LB	EX
haarWave	EX	MP	CM	CO	LB	EX

Fig. 61 Combined Reduction: Retention of Performance Trends with Default Thresholds for late_broadcast

	MPI Ssend					do work
no loss	EX	MP	CM	P2	LR	EX
relDiff	EX	MP	CM	P2	LR	EX
absDiff	EX	MP	CM	P2	LR	EX
Manhattan	EX	MP	CM	P2	LR	EX
Euclidean	EX	MP	CM	P2	LR	EX
Chebyshev	EX	MP	CM	P2	LR	EX
iter_k	EX	MP	CM	P2	LR	EX
iter_avg	EX	MP	CM	P2	LR	EX
avgWave	EX	MP	CM	P2	LR	EX
haarWave	EX	MP	CM	P2	LR	EX

Fig. 62 Combined Reduction: Retention of Performance Trends with Default Thresholds for late_receiver

	MPI Recv					do work
no loss	EX	MP	CM	P2	LS	EX
relDiff	EX	MP	CM	P2	LS	EX
absDiff	EX	MP	CM	P2	LS	EX
Manhattan	EX	MP	CM	P2	LS	EX
Euclidean	EX	MP	CM	P2	LS	EX
Chebyshev	EX	MP	CM	P2	LS	EX
iter_k	EX	MP	CM	P2	LS	EX
iter_avg	EX	MP	CM	P2	LS	EX
avgWave	EX	MP	CM	P2	LS	EX
haarWave	EX	MP	CM	P2	LS	EX

Fig. 63 Combined Reduction: Retention of Performance Trends with Default Thresholds for late_sender

	MPI Gather					do work
no loss	EX	MP	CM	CO	ER	EX
relDiff	EX	MP	CM	CO	ER	EX
absDiff	EX	MP	CM	CO	ER	EX
Manhattan	EX	MP	CM	CO	ER	EX
Euclidean	EX	MP	CM	CO	ER	EX
Chebyshev	EX	MP	CM	CO	ER	EX
iter_k	EX	MP	CM	CO	ER	EX
iter_avg	EX	MP	CM	CO	ER	EX
avgWave	EX	MP	CM	CO	ER	EX
haarWave	EX	MP	CM	CO	ER	EX

Fig. 64 Combined Reduction: Retention of Performance Trends with Default Thresholds for Nto1_32

	MPI Barrier						do work
no loss	EX	MP	SN	BA	WB	BC	EX
relDiff	EX	MP	SN	BA	WB	BC	EX
absDiff	EX	MP	SN	BA	WB	BC	EX
Manhattan	EX	MP	SN	BA	WB	BC	EX
Euclidean	EX	MP	SN	BA	WB	BC	EX
Chebyshev	EX	MP	SN	BA	WB	BC	EX
iter_k	EX	MP	SN	BA	WB	BC	EX
iter_avg	EX	MP	SN	BA	WB	BC	EX
avgWave	EX	MP	SN	BA	WB	BC	EX
haarWave	EX	MP	SN	BA	WB	BC	EX

Fig. 65 Combined Reduction: Retention of Performance Trends with Default Thresholds for NtoN_32

	MPI Bcast					do work
no loss	EX	MP	CM	CO	LB	EX
relDiff	EX	MP	CM	CO	LB	EX
absDiff	EX	MP	CM	CO	LB	EX
Manhattan	EX	MP	CM	CO	LB	EX
Euclidean	EX	MP	CM	CO	LB	EX
Chebyshev	EX	MP	CM	CO	LB	EX
iter_k	EX	MP	CM	CO	LB	EX
iter_avg	EX	MP	CM	CO	LB	EX
avgWave	EX	MP	CM	CO	LB	EX
haarWave	EX	MP	CM	CO	LB	EX

Fig. 66 Combined Reduction: Retention of Performance Trends with Default Thresholds for 1toN_32

	MPI Ssend					MPI Recv					do work
no loss	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
relDiff	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
absDiff	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
Manhattan	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
Euclidean	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
Chebyshev	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
iter_k	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
iter_avg	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
avgWave	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
haarWave	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX

Fig. 67 Combined Reduction: Retention of Performance Trends with Default Thresholds for 1to1r_32

	MPI Ssend					MPI Recv					do work
no loss	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
relDiff	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
absDiff	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
Manhattan	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
Euclidean	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
Chebyshev	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
iter_k	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
iter_avg	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
avgWave	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
haarWave	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX

Fig. 68 Combined Reduction: Retention of Performance Trends with Default Thresholds for 1to1s_32

	MPI Gather					do work
no loss	EX	MP	CM	CO	ER	EX
relDiff	EX	MP	CM	CO	ER	EX
absDiff	EX	MP	CM	CO	ER	EX
Manhattan	EX	MP	CM	CO	ER	EX
Euclidean	EX	MP	CM	CO	ER	EX
Chebyshev	EX	MP	CM	CO	ER	EX
iter_k	EX	MP	CM	CO	ER	EX
iter_avg	EX	MP	CM	CO	ER	EX
avgWave	EX	MP	CM	CO	ER	EX
haarWave	EX	MP	CM	CO	ER	EX

Fig. 69 Combined Reduction: Retention of Performance Trends with Default Thresholds for Nto1_1024

	MPI Barrier						do work
no loss	EX	MP	SN	BA	WB	BC	EX
relDiff	EX	MP	SN	BA	WB	BC	EX
absDiff	EX	MP	SN	BA	WB	BC	EX
Manhattan	EX	MP	SN	BA	WB	BC	EX
Euclidean	EX	MP	SN	BA	WB	BC	EX
Chebyshev	EX	MP	SN	BA	WB	BC	EX
iter_k	EX	MP	SN	BA	WB	BC	EX
iter_avg	EX	MP	SN	BA	WB	BC	EX
avgWave	EX	MP	SN	BA	WB	BC	EX
haarWave	EX	MP	SN	BA	WB	BC	EX

Fig. 70 Combined Reduction: Retention of Performance Trends with Default Thresholds for NtoN_1024

	MPI Bcast					do work
no loss	EX	MP	CM	CO	LB	
relDiff	EX	MP	CM	CO	LB	
absDiff	EX	MP	CM	CO	LB	
Manhattan	EX	MP	CM	CO	LB	
Euclidean	EX	MP	CM	CO	LB	
Chebyshev					LB	EX
iter_k	EX	MP	CM	CO	LB	EX
iter_avg	EX	MP	CM	CO	LB	EX
avgWave	EX	MP	CM	CO	LB	
haarWave	EX	MP	CM	CO	LB	

Fig. 71 Combined Reduction: Retention of Performance Trends with Default Thresholds for 1toN_1024

	MPI Ssend					MPI Recv					do work
no loss	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
relDiff	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
absDiff	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
Manhattan	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
Euclidean	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
Chebyshev	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
iter_k	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
iter_avg	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
avgWave	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
haarWave	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX

Fig. 72 Combined Reduction: Retention of Performance Trends with Default Thresholds for 1to1r_1024

	MPI Ssend					MPI Recv					do work
no loss	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
relDiff	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
absDiff	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
Manhattan	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
Euclidean	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
Chebyshev	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
iter_k	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
iter_avg	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
avgWave	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX
haarWave	EX	MP	CM	P2	LR	EX	MP	CM	P2	LS	EX

Fig. 73 Combined Reduction: Retention of Performance Trends with Default Thresholds for 1to1s_1024

	pmi recv						sweep
no loss	EX	MP	CM	P2	LS	MO	EX
relDiff	EX	MP	CM	P2	LS	MO	EX
absDiff	EX	MP	CM	P2	LS	MO	EX
Manhattan	EX	MP	CM	P2	LS	MO	EX
Euclidean	EX	MP	CM	P2	LS	MO	EX
Chebyshev	EX	MP	CM	P2	LS	MO	EX
iter_k	EX	MP	CM	P2	LS	MO	EX
iter_avg	EX	MP	CM	P2	LS	MO	EX
avgWave	EX	MP	CM	P2	LS	MO	EX
haarWave	EX	MP	CM	P2	LS	MO	EX

Fig. 74 Combined Reduction: Retention of Performance Trends with Default Thresholds for sweep3d_8p

	pmi recv						sweep
no loss	EX	MP	CM	P2	LS	MO	EX
relDiff	EX	MP	CM	P2	LS	MO	EX
absDiff	EX	MP	CM	P2	LS	MO	EX
Manhattan	EX	MP	CM	P2	LS	MO	EX
Euclidean	EX	MP	CM	P2	LS	MO	EX
Chebyshev	EX	MP	CM	P2	LS	MO	EX
iter_k	EX	MP	CM	P2	LS	MO	EX
iter_avg	EX	MP	CM	P2	LS	MO	EX
avgWave	EX	MP	CM	P2	LS	MO	EX
haarWave	EX	MP	CM	P2	LS	MO	EX

Fig. 75 Combined Reduction: Retention of Performance Trends with Default Thresholds for sweep3d_32p