Reducing the Network Latency and Bandwidth Requirements of Parallel and Distributed Applications

Lars Ailo Bongo

A dissertation for the degree of Philosophiae Doctor

UNIVERSITY OF TROMSØ
Faculty of Science
Department of Computer Science

July 2007

To Kristin.

Abstract

Latency and bandwidth requirements often dictate which platform an application must be run on to achieve acceptable performance. But often there is a cost or availability incentive for running an application on a platform with lower bandwidth or higher latency. This dissertation presents four approaches for reducing the network latency and bandwidth requirements of communication intensive parallel and distributed applications.

Our first approach for reducing the network latency requirements of parallel applications is to improve collective communication performance. The latency of collective operations can be reduced by adapting these to the application and platform in use. Such adaptation requires performance analysis of message traces collected internally in the communication system. For large-scale clusters, large volumes of trace data must be collected, analyzed, and transferred over the network. We propose a framework for building scalable runtime monitors. Our results show that monitors for collective operation analysis can be run on large-scale Ethernet and WAN multi-clusters without significantly perturbing the monitored application. The contributions are:

- A monitoring framework. It supports the development of a vide variety of trace based monitors.
- Approach for scalable message tracing with a very small memory footprint where message traces are processed at runtime by threads run on the cluster nodes.
- Approach for exploiting underutilized compute and network resources to run a monitor on a cluster with very low perturbation of the monitored application.

To further reduce the latency of collective operations used for global synchronization of parallel application threads on a WAN multi-cluster, we implemented new operations for evaluating global conditions. Measurements demonstrate that the operation has the same latency on a WAN multi-cluster as on a single cluster for most global condition evaluations. Our contribution is:

- An allreduce operation that can complete for most cases without WAN communication, and that does not change the application result.

Our third approach to reduce the network latency requirements of parallel applications is to overlap communication wait time with computation by overdecomposing a parallel application. The improvements and limitations of overdecomposition are documented by analyzing performance data collected for the NAS benchmarks run on a cluster composed of the first generation simultaneous multithreading (SMT) processors. The contributions are:

- Method for performance analysis of overdecomposed applications.
- Performance study of overdecomposed parallel applications run on processors supporting SMT.

Our approach for reducing the network bandwidth requirements of distributed applications is to divide the transferred data into segments and then eliminate redundant transfers of segments. Previous approaches do not work well for compressing multi-dimensional data, such as 2D pixels in remote data visualization and high-dimensional scientific datasets. In addition, large segments are required to achieve a high compression ratio. We propose a method to identify and eliminate redundant data transfers of complex data types over a network. The implemented prototype allows visualization of genomic data analysis applications interactively across WANs with relatively low available network bandwidths. Our contributions are:

- A framework for global compression using two-level fingerprinting and application specific segmentation, where redundancy detection is separated from redundancy elimination such that the same compression engine can be used with different application specific segmentation methods.
- Two-level fingerprinting protocol for efficiently encoding unique segments, such that smaller segments can be used to improve redundancy detection.
- A novel 2-dimensional content-based segmentation approach for remote visualization data.
- Design and implementation of a very large cache on disk for storing previously sent segments.
- A network bandwidth optimized, platform-independent remote visualization system using two-level fingerprinting to reduce end-to-end latency of screen updates.

These methods can be composed to improve the end-to-end communication performance for communication intensive parallel and distributed applications.

Acknowledgements

Many people have helped me with the work presented in this dissertation:

I would like to thank my advisor Professor Otto J. Anshus for his guidance, support, and encouragement during all my research. In addition, Otto has arranged for me to collaborate with many smart people.

Also, thanks to my co-advisor Professor Brian Vinter for his guidance and help with arranging access to the resources necessary for my work on parallel application scalability.

Prof. John Markus Bjørndalen has helped setting up many experiments and understanding the results; especially during Otto's sabbatical.

Prof. Tore Larsen have provided advice and helped write papers, especially during our stay at Princeton.

Also, thanks to the other Ph.D. students in our group (especially Daniel Stødle and Espen S. Johnsen) for their valuable comments, and our discussions.

Many thanks to Professors Kai Li and Olga Troyanskaya for arranging for me to stay a year at Princeton University, and for their guidance during the stay. Kai encouraged me to work on compression systems, while Olga introduced me to bio-informatics and showed me how great interdisciplinary work can be. Grant Wallace, William Josephson, and Christian Bienia have provided help and discussion about the design and implementation of the Canidae compression system, while Matthew Hibbs helped with the genomic applications. Also thanks to the other Ph.D. students in Kai's and in Olga's lab, and the other people at Princeton, who made my stay very enjoyable.

Professor Liviu Iftode arranged for me to work with his group, in their fun atmosphere, at Rutgers University. I very much enjoyed the work on monitoring for intrusion detection together with Arati Baliga.

Professor Xing Cai at Simula provided a parallel application with an interesting communication performance problem.

Professor Jonathan Walpole provided us access to a cluster at Oregon Graduate Institute. Prof. Anne E. Elster provided access to a cluster at the Norwegian University of Science and Technology, and Profs. Josva Kleist and Gerd Behrmann to a cluster at Aalborg University.

Our departments technical staff, especially Jon Ivar Kristiansen, but also Ken-Arne Jensen and Torfinn Holand, have been very helpful with technical support for all our equipment. Also, thanks to the departments and faculties administrative staff (especially Svein-Tore Jensen, Jan Fuglesteg, and Ola Marius Johnsen) for practical help, and the staff at the High Performance Computing Group.

Last, many thanks to the person giving me endless support; my wife Kristin.

My stay at Princeton and Rutgers were supported by grants from The Research Council of Norway (Project No. 164825), The University of Tromsø, and Princeton University. The experimental equipment is due to Research Council of Norway grants 159936/V30 and 155550/420.

Contents

List of Figures

List of Tables

List of Formulas

Chapter 1

Introduction

This chapter gives an overview of this dissertation. First, three important classes of parallel applications are described, and their communication performance requirements are defined. Then four approaches for utilizing available computation and storage resources to improve the end-to-end communication performance for these application classes are presented. The problems for each approach are defined, the methodology for solving the problems is presented, and contributions are stated. Finally, the organization of the rest of the dissertation is outlined.

1.1 Latency and bandwidth requirements

Distributed applications require communication over a network. For communication intensive applications the communication will be frequent and (or) involve large amounts of data. In these instances, the performance is limited either by the communication *latency*; the time to transfer an empty message, or the network *bandwidth*; the number of bytes that can be transferred in a given time. High latency or insufficient bandwidth may cause applications to fail to meet response time, quality, or resource utilization requirements. Some examples are given below.

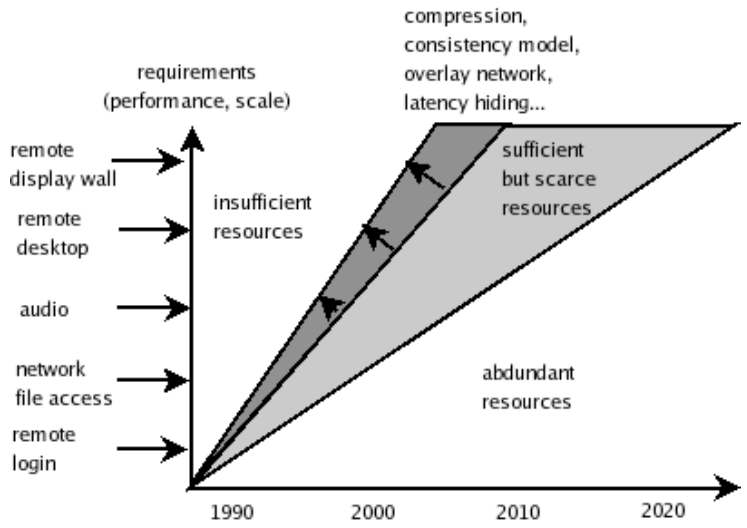


Figure 1: For the applications in the shaded area, resources must be carefully scheduled to meet the bandwidth requirements. This "window of scarcity" can be expanded by exploiting available computational and storage resources. The figure is based on figure 2 in [14].

During the last three decades network bandwidth has increased 1000x, and network latency has decreased 20x [164]. These improvements allow developing new applications that take advantage of the higher bandwidth and lower latency to provide new or improved services. However, these applications often operate in a "window of

scarcity" where network resources must be carefully managed to meet the applications requirements (Figure 1).

Processor performance and disk capacity have also had similar improvements [164]. In addition, processors and disks can be added incrementally to distributed computing platforms, making it much easier to improve computation or storage capacities than it is to achieve similar improvements for communication throughput or latencies. Therefore, distributed computing platforms typically have compute and storage resources that may be exploited to expand the window of scarcity, and hence may allow running applications with higher network performance requirements than is provided by the platform.

Below we describe three classes of communication intensive applications, and motivate why the network is the bottleneck for these applications.

1.1.1 Communication intensive application domains

First three application domains are described. Then the platforms these are typically run on are characterized.

1.1.1.1 Parallel applications

Parallel computing involves splitting time-consuming computations into tasks that are executed in less time simultaneously on multiple processors. Parallel computing has become an important tool in many scientific disciplines, and has transformed many disciplines [97]. The typical parallel application is a scientific simulation, such as fluid dynamics simulation or weather prediction. But parallel applications are also used in other domains, such as rendering of movie special effects.

The parallelization process can be divided into three steps [66]. First, the computation is decomposed into tasks that are assigned to threads (or processes). The goals of this step are to expose parallelism, distribute the tasks to achieve a good workload balance among the threads, and to reduce communication volume. Second, a programming model and language are chosen, and used to orchestrate data accesses, communication, and synchronization among threads. Important performance goals are to reduce communication and synchronization costs as seen by processors, and to reduce serialization caused by access to shared resources. Third, threads are mapped to processors such that network locality can be exploited.

However, parallelization does not improve the execution time of all applications for two reasons. First, not all parts of an application can be run in parallel, and hence the sequential parts will eventually limit the execution time reduction (Amdahl's law [13]). Second, and usually more important, the communication and synchronization necessary to coordinate all processors introduces an overhead, which typically increases with the number of parallel application threads.

The two most common parallel programming models are multi-threading and message passing. Multi-threading usually require fewer changes to a sequential application, but a shared address space platform must be used. These typically have only 2—16 processors, and hence limit the scalability of the applications [74]. Therefore, many of the parallel applications written today employ message passing. For message passing the Message Passing Interface (MPI) [148, 149] has become the *de facto* standard. MPI libraries provide both *point-to-point* and *collective* communication operations (Table 1). The collective operations implement coordinated communication involving

multiple processors, and can be used to broadcast data, gather data, scatter data, synchronize processes and execute reduce operations on distributed data.

Operation name	Communication pattern	Usage
Barrier	Many-to-many	Synchronize processes
Broadcast	One-to-many	Send data from one to many processes
Gather	Many-to-one	One process receives data from many processes
Scatter	One-to-many	Divide data from one process to many processes
Reduce	Many-to-one	Addition, min, max, multiplication, or another operation on distributed data. A single process receives the result.
Allreduce	Many-to-many	Reduce and then broadcast the result to all processes. Also synchronizes the processes.

Table 1: Commonly used collective communication operations provided by the Message Passing Interface (MPI).

Communication intensive parallel applications require low latency of communication operations (milliseconds to microseconds), and/or high network bandwidth (up to several gigabits per second). Failure to meet these requirements on a platform limits the number of processors that can be utilized efficiently, and hence the performance of the parallel application.

1.1.1.2 Remote visualization

Data analysis in scientific fields such as genomics is a collaborative process. Studies typically include multiple researches, often from different institutions, regions, and countries. Such collaboration requires interactive discussion of the data and its analysis, which is difficult to do without sharing visualizations. To make discussions truly effective, interactive exploration of the data should be provided in a seamless manner, independent of the choice of data analysis applications, platforms, and the users geographical location.

Remote desktop systems, such as VNC [184] or Microsoft Remote Desktop [67], can be used for remote collaboration, by allowing several users to share the visualization on a single desktop. In addition, a remote desktop system can be used to interact with an application running on a different platform than on a users machine.

Most recent desktop systems are *thin-client* systems, consisting of a server that runs the applications logic and stores most of the application state, and clients that only implement functionality to display received screen updates, and forward user input to the server. Screen updates and user input events are encoded using a remote display protocol. The protocol can either provide a rich set of high-level display commands (as in Microsoft Remote Desktop or the X window system [198]), or fewer low-level

commands (as in VNC, Sun Ray [200] or THINC [25]). For high-latency networks, low-level protocols have better performance [131], since they requires less synchronization.

For interactive remote visualization the latency of screen updates should be less than 150 milliseconds [204, 219]. The bandwidth requirements range from tens of megabytes up to hundreds of megabytes depending on the screen resolution and update frequency. Improvements in latency allows for smoother interaction, while increased bandwidth allows improving the quality of the visualization including a higher resolution.

1.1.1.3 Data Intensive Science

Current scientific instruments and simulations are creating peta-scale data volumes, and the amount of data produced is roughly doubled each year [94]. Examples include the Sloan Digital Sky Survey (SDSS) astronomical survey [201], the BaBar high energy physics experiment [21], the Entrez federated health sciences database [158], and the CERN Large Hadron Collider [56].

The amount of data stored, and the computation necessary for analyzing the data requires building a data storage and analysis infrastructure. The infrastructure may be used to access the data by thousands of scientists participating in the project working at hundreds of institutions. Building a distributed infrastructure has several advantages including no single point of failure, and load balancing of data, computation, and user support [57]. In addition the different parts of the infrastructure can be individually funded by the participating organizations.

A main challenge for such a distributed infrastructure is providing the necessary network bandwidth between the compute and storage resources. The bandwidth requirements are large, gigabits or higher per second. Available network bandwidth limits the data sets that can be transferred over the network, and hence the type of analysis scientists can do on their local resources.

1.1.2 Execution environments

The bandwidth and latency requirements often dictate which platform an application must be run on to achieve acceptable performance. But often there is a cost or availability incentive for running an application on a platform with lower bandwidth, higher latency, or both.

Parallel applications are run on a wide variety of platforms including: a single processor with multiple cores and multithreading support, Beowulf clusters were tens of commodity components computers are connected using Ethernet, large parallel systems with hundred thousands of processors connected using high performance interconnects, to a Grid that is a federation of Beowulf clusters connected using a wide area network (in this dissertation we refer to such systems as *WAN multi-cluster*s). The platforms mainly differ in the number of processors and the bandwidth and latency provided by the network interconnect (Table 2).

Many organizations use a Beowulf platform since it provides the best price-performance ratio. But, the relatively low bandwidth and high latency of the Ethernet network typically used in Beowulf clusters does not allow some parallel applications to be run efficiently. Recently, many organizations have connected their clusters to form a WAN multi-cluster in order to share their compute resources. The even higher

latency and lower bandwidth of WANs limits the usability of using such federation of Beowulf clusters.

Platform	Processors	Network interconnect	Network Bandwidth	Network Latency	Advantage
A single commodity processor [89, 110, 214]	1 (1-4 cores and 1—16 threads)	None			Cost, ease of programming
Beowulf cluster [92, 185, 216]	Tens	Gigabit Ethernet [90]	Tens of GB/s	Micro-seconds	Performance/ cost
WAN Multi-cluster (Grid) [46, 84, 85]	Hundreds	Ethernet and WAN	MB/s	Milli-seconds	Cost (resources are shared)
Large parallel system [11, 95, 132]	Hundred thousand	Myrinet [40], Quadrics [165], InfiniBand [109], proprietary…	GB/s	Micro-seconds	Highest performance
SETI@home [236]	Millions	WAN	MB/s	Days	Cost (free resources)

Table 2: Characteristics of parallel application platforms.

Network	Latency	Total bandwidth	TCP/IP throughput
Fast Ethernet	Microseconds	100 Mbit/sec	8 MByte/sec
Gigabit Ethernet	Microseconds	1000 Mbit/sec	80 MByte/sec
Tromsø-Odense WAN [42] (shared)	32 ms	155 Mbit/sec	0.32 MByte/sec
Tromsø-Princeton WAN [49] (shared)	120 ms	2500 Mbit/sec	0.2 MByte/sec
LambdaGrid [116] (dedicated network)	78 ms	10 Gbit/s	1.13 GByte/sec
CERN LHC Tier-0 [57] (dedicated network)	Microseconds	2x 10 000 Mbit/sec	1600 MByte/sec (our assumption)

Table 3: Characteristics of networks used in this dissertation and in related work.

Remote visualization may require specialized hardware and a dedicated network to access the raw data, and do the computation for the visualization [116]. But often the visualization must be sent over a shared wide area network to the user. Also, during the last decade more users have got access to specialized visualization platforms that

provide higher resolution than a typical desktop screen, such as display walls with very high resolution [135]. When such large displays are used, the bandwidth requirements become even higher. Current shared wide area networks do not provide the bandwidth and latency necessary for high-resolution interactive remote visualizations (Table 3).

The compute and storage resources in a *large data set* infrastructure are often connected using dedicated high bandwidth networks [57]. But, it is usually not economically feasible to build a dedicated network to all users. Many users must therefore access the data using a public wide area network, where the available bandwidth is shred among the users. The resulting application level throughput may be as low as a megabyte per second, limiting the size of datasets that can be transferred in a reasonable amount of time ((Table 3).

1.2 Reducing bandwidth and latency requirements

The previous section explained how the network bottleneck limits the scalability of parallel applications run on Beowulf and WAN multi-clusters, the data set sizes that can be transferred over a WAN, and the quality of remote visualization over a WAN. Network performance can be improved by modifying the network hardware, the software communication layer, or the application. Purchasing and deploying new network hardware is costly, but will improve the performance of many communication intensive applications. Rewriting applications is often not a practical solution due to the large number of applications that must be rewritten by the application programmers. Therefore a solution is needed that is both cheap to deploy, and that does not require application code changes.

Such a solution can be added to the software communication layer. Most distributed applications communicate using a well-defined protocol or by using operations specified by an interface. Usually the protocol or interface only specifies the semantics of the communication operations. Therefore, different implementations of the protocol or interface can use the available resources differently, and even add new functionality internally. Such solutions are described below.

1.2.1 Tuning collective communication operations

A collective operation spanning tree distributes the computation of the operation among the cluster hosts and specifies how the processors communicate and synchronize (Figure 2). The performance of the collective operations used by parallel applications depends on the spanning tree used, and the mapping of the spanning tree to the computation nodes and network topology [126, 205, 218, 229]. Adapting the algorithm for the resources in use [32, 80, 81] can therefore reduce collective operation latency.

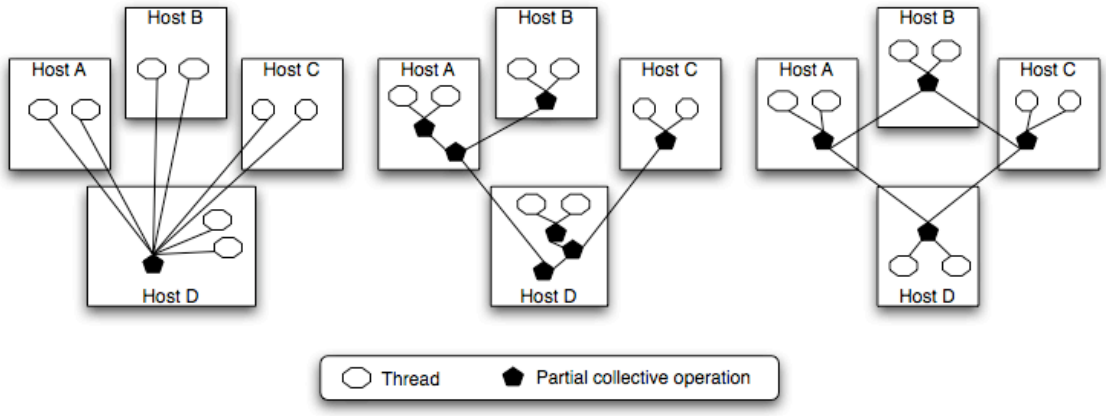


Figure 2: Three approaches for orchestrating the communication and synchronization for a many-to-many collective operation. The configurations shown are a flat tree (left), a hierarchy aware spanning tree (middle), and a configuration where four spanning trees are connected by an all-to-all communication graph (right).

The performance of a collective operation spanning tree depends on many factors including the cluster topology, cluster hosts, the load on hosts, the message size, and communication-computation overlap. It is therefore difficult to develop a mathematical model [12, 31, 65] or use simulation to find the best configuration [80, 81, 229]. Furthermore, on shared platforms such as a compute Grid, the resources allocated to an application may change for each time the application is run. It is therefore necessary to monitor and analyze the performance of collective operation spanning trees used by the application running on the actual platform to find the best configuration [32, 80, 81, 126, 205, 218, 229, 242].

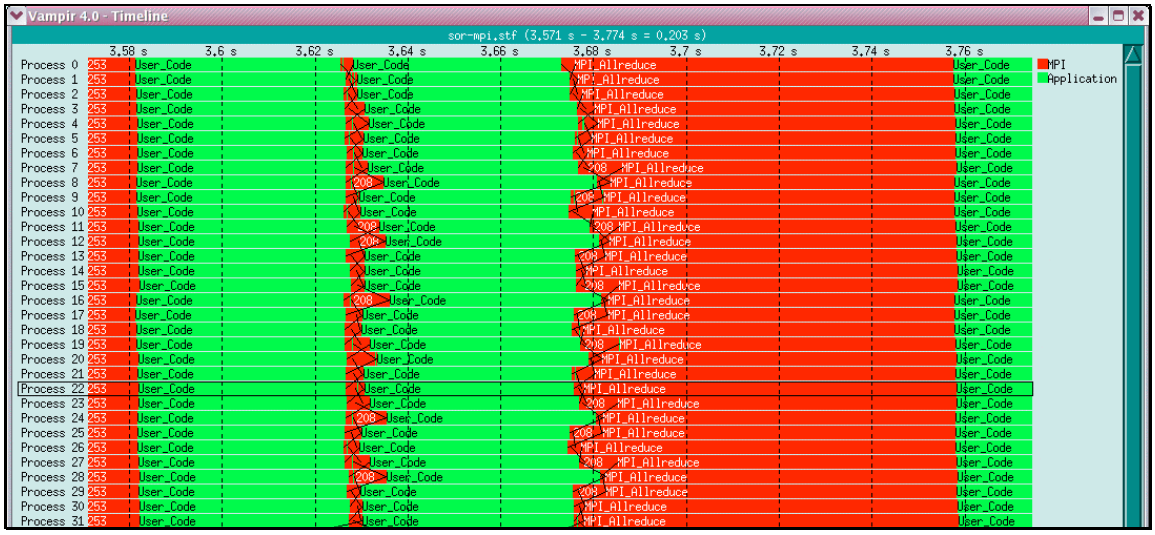


Figure 3: Timeline visualization from the Vampir tool [51, 162] (now Intel Trace Analyzer [111]) identifying the MPI_Allreduce as a bottleneck. There is one horizontal bar per thread that shows when the thread was computing (green) and communicating (red).

Existing commercial [51, 107, 111] and research [55, 99, 115, 119, 144, 155, 178, 194, 225, 241] monitoring and performance analysis tools for parallel applications have one or more of the following problems:

- The performance analysis can identify a certain collective operation call as the bottleneck (as shown in Figure 3). However, the tools are not able to provide insight into why the operation has poor performance and how the problem can be avoided.
- The monitor introduces overheads that can significantly perturb the monitored application such that a significant error is introduced to the results. The overheads are due to the monitor and monitored application sharing, and hence competing, for processor memory and network resources.
- The monitor does not scale to large clusters with hundreds of computers. The monitors may require storing hundreds of megabytes of data per node, which must often be transferred to a single node for analysis. Also, processing the data can take tens of minutes, and thereby not allowing runtime analysis, or even post-mortem analysis, between two application runs.
- The tools are difficult to use, since they may require application code modifications, recompiling or re-linking the application, or installing new software on the platform. Also, applications must usually be re-run in order to collect performance data.

1.2.2 Collective operations for WAN multi-clusters

Collective operations ease the programming of parallel applications. The small set of operations provided by for example MPI, can be used to implement most of the required collective communication for a parallel application. The semantics of these operations are chosen such that the result of an operation is predictable and repeatable. But, the latency of the operation is determined by the communication and synchronization necessary to satisfy the semantics requirements for the collective operation. For example if the hosts in Figure 2 are connected using WANs, the latency of using a flat spanning tree includes two WAN latencies (for the reduce and broadcast phase). To only include a single WAN latency, an all-to-all graph can be used for WAN communication [126]. But with this change the operation can no longer guarantee that the same results is always returned for operations on floating point numbers [126]. However, the latency of the collective operation may still be orders of magnitude larger than on a single cluster.

In order to run a tightly synchronized parallel application on a WAN multi-cluster more radical changes to the collective operations are necessary to achieve the necessary orders of magnitude reduction in latency. For the users of parallel applications it is important that the changed collective operations does not change the results produced by the parallel program. Also, application programmers should not be required to make changes to the applications source code.

1.2.3 Communication and computation overlap

Parallel applications are usually decomposed and mapped such that there is one thread per processor, since this minimizes the communication and synchronization overhead, and reduces the number of cache conflict misses. However, a processor will mostly be idle when its thread is blocked on a communication operation. Parallel application performance can be improved if the processor is more efficiently used. One approach is to overlap the wait time with computation for another thread [29, 70, 134, 173, 207]. In order to map multiple threads to each processor, the problem can be *overdecomposed* such that there are more tasks than processors (Figure 4). Such

overdecomposition requires no changes to application code, since most parallel applications are coded such that the number of threads can be specified at load time.

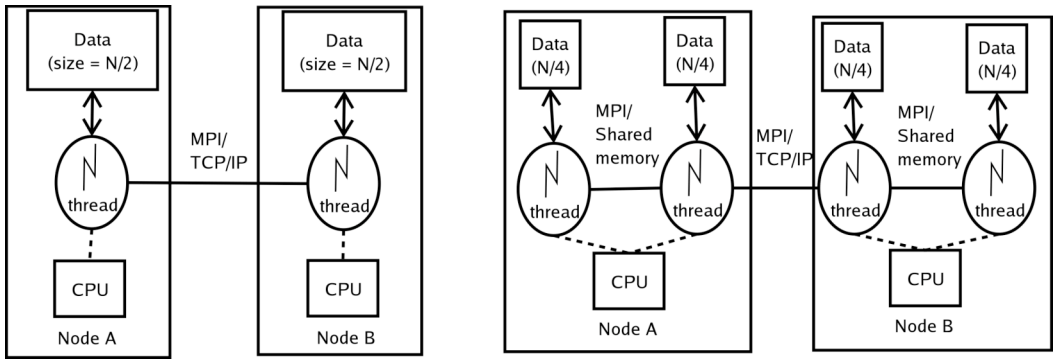


Figure 4: A parallel application decomposed and mapped with one thread per processor decomposition (left), and overdecomposed such that multiple threads are run on the same processor (right).

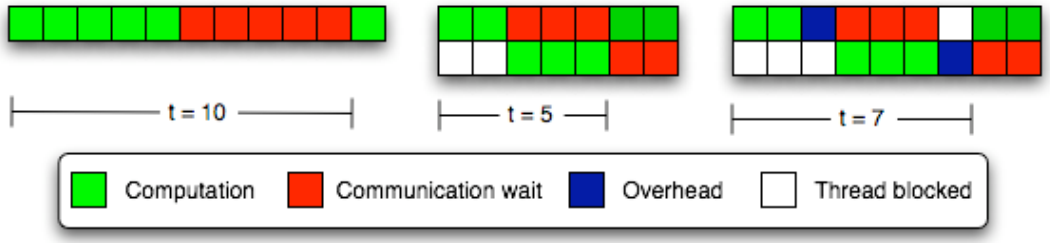


Figure 5: Communication-computation overlap can improve execution time even if some overhead is introduced.

Computation-communication overlap reduces parallel application execution time when the amount of wait-time used for computation is larger than the introduced overheads (Figure 5). Another advantage of overdecomposition is that it may provide the thread level parallelism (TLP) necessary to utilize modern processors with simultaneous multi-threading (SMT) and chip multiprocessors (CMP).

Overdecomposition is normally not used for parallel applications for two reasons. First, for most applications the number of communication events, synchronization events, and number of bytes communicated all increase [66]. Second, the computation in parallel applications is structured such that data accesses optimize processor cache usage [170]. Hence, each context switch may require reading a new working set into the caches.

Motivated by the introduction of processor with CMP or SMT processors, the benefits and limitations of overdecomposing parallel applications should be re-evaluated. In particular, the degree of computation-communication overlap, cache miss increase, communication overhead, and synchronization overhead should be quantified when applying overdecomposition. To provide the results, performance metrics collected at multiple software and hardware layers must be combined.

1.2.4 Compression for remote visualization data

A remote visualization system for collaborative scientific research should satisfy three requirements. First, the response time should be fast enough to allow collaborating parties to interact smoothly, even when using visualization-intensive software across a

relatively low-bandwidth wide area network (WAN). Second, collaborating parties should not be required to replicate data since datasets can be large, sensitive, proprietary, and potentially protected by privacy laws. Third, the system should allow collaborators to use any visualization and data analysis software running on any platform.

Adding remote collaboration capabilities to applications [63, 82, 150] may satisfy the first and the second requirements, but achieving universal adaptation is difficult due to then number of different applications in use. Web browser-based remote visualization software can satisfy the third requirement, but not the first two because usually these are single-user system and do not optimize the network bandwidth requirement. Most thin-client remote visualization systems, such as Sun Ray [200], THINC [25], Microsoft Remote Desktop [67] and Apple Remote Desktop [19] satisfy only the second requirement because they do not perform efficient data compression, and are platform-dependent.

The platform independent thin-client VNC [184] system, satisfies the second and third requirement. VNC has one graphics primitive: "put rectangle of pixels at position (x, y)" [184]. This allows separating the processing of application display commands from the generation of display updates. The client can therefore be stateless and hence easy to implement, maintain and port. The disadvantage of the protocol is that bandwidth requirements are high. To reduce bandwidth the screen updates can be compressed before being transferred over the network. However, existing compression algorithms for two-dimensional pixel segments do not provide the compression ratio and low compression time necessary for smooth interaction. In the following section we propose a novel method to compress a long visualization session.

1.2.5 Compression for large scale-scientific data sets

Distributed platforms have available computational and storage resources that can be used to compress network data in order to reduce the bandwidth requirements of distributed applications. This allows reducing the transfer time for large scientific data sets, and improving the resolution, color depth, and frame rate of remote visualizations.

Compression algorithms encode information using fewer bits than the original un-encoded representation uses. The compression can be either lossy or lossless. Lossy compression is often used to encode image [235], audio [113] or video data [88] since it typically achieves a higher compression ratio, and the reduced quality is acceptable. But for scientific data sets lossless compression is typically required. Similarly, for scientific remote visualization any changes to the provided visualizations are undesirable.

Network data is typically compressed using a local compression algorithm [9, 188, 202] which decouples compression from decompression, such that no communication between the server and client is necessary when doing the compression and decompression. A popular local compression algorithm is DEFLATE [73], used in the zlib/gzip library[9]. DEFLATE combines the Lempel-Ziv (LZ77) duplicate string elimination algorithm [244], with Huffman encoding for bit reduction [103]. LZ77 detects duplicate strings and replaces these with a back-reference to the previous location of the string. Huffman encoding replaces symbols with weighted symbols based on frequency of use. The problem with existing local compression algorithms,

is that the ratio achieved for scientific data is low, while compression time is to high for remote visualization.

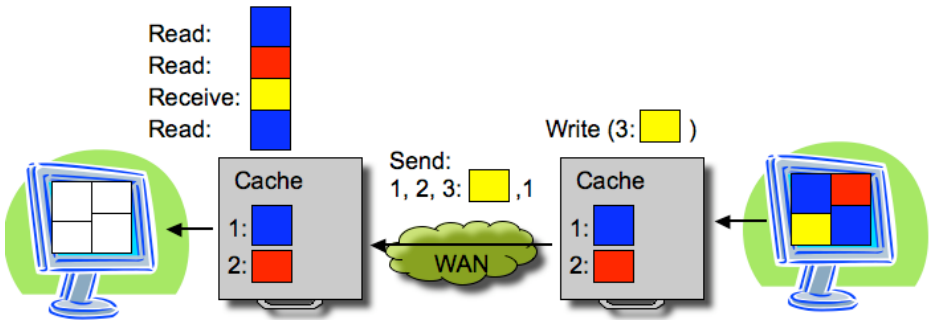


Figure 6: Global compression used to compress screen content. Previously sent pixel regions (segments) are stored in cooperating caches at the sender and receiver side. The data to be sent is segmented, and in place of replicated segments only the cache index is sent over the WAN.

During the last few years global compression has been suggested to improve compression ratio and to reduce compression time [156, 171, 209]. The sender and receiver cooperate to maintain caches of previously sent data. Data is compressed by eliminating transfer of redundant bytes (Figure 6). First, a *redundancy detection* algorithm divides the data to be sent into segments. Then, *redundancy elimination* is implemented by sending a fingerprint instead of replicated segments. The fingerprint is usually a hash value of the segment, and is used to retrieve the data from a segment cache. Such algorithm can detect redundancy in the entire data sets, while redundancy detection in local compression algorithms is within a local scope.

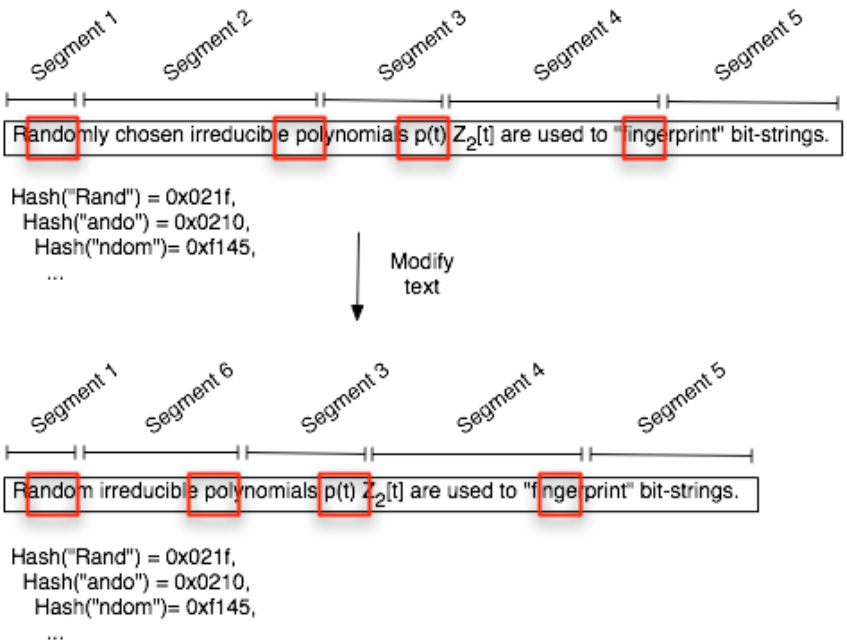


Figure 7: Content based 1-D anchoring. First hash values are calculated for fixed size substrings, including all overlaps. Anchorpoints are then selected based on the *k* least significant bits in the hash value. The anchorpoints divide the text into segments. Modifying the text does not change most anchor points, and hence most segments are identical.

The simplest redundancy detection approach is to anchor segments statically (such as an 8×8 pixel grid, used in the MPEG [88] compression algorithm). The problem with a static approach is that the anchoring is sensitive to data movements such as screen scrolls. To improve redundancy detection, a content-based technique introduced by Manber [145] is typically used to segment the data. His method applies a Rabin fingerprint filter [50, 175] over a byte data stream and identifies anchor points wherever the k least significant bits of the filter output are zeros. With a uniform distribution, an anchor point should be selected every 2^k bytes. The anchor points can then be used to either divide the data into segments [156] or as starting points for growing redundant regions [209]. Since the anchor points are selected based on the data content, they move with the data such that data insertion or removal in a datastream does not influence anchorpoint selection (Figure 7). Another advantage is that Rabin fingerprint calculation is very fast compared to the redundancy detection mechanisms typically used by local compression algorithms.

The compression ratio of global compression depends on the amount of redundancy found in the data, and the fingerprint to segment size.

Previously proposed segmentation approaches work well only with 1D data types, such as web content, documents, email and binaries, while many important applications use complex data types such as 2-D screen buffers for remote data visualization, and multidimensional scientific datasets. Therefore, using a redundancy detection algorithm that is aware of the data structure can improve the compression ratio.

Previous global compression systems [100, 156, 172, 180, 220, 222] typically chose a secure hash, such as 160-bit SHA-1 [7], as a fingerprint so that the probability of a fingerprint collision can be lower than a hardware bit error rate. However, since the global compression ratio is limited by the ratio of the average pixel segment size to the fingerprint size, using a large fingerprint size will reduce the compression ratio.

1.3 Approach and methodology

The methodology used in this work is to first build a prototype for the approaches proposed in the previous section, and then use the prototype to document the benefits and limitations of each approach.

1.3.1 Spanning tree monitoring

We attempt to reduce collective operation latency by adapting the collective operation spanning tree to the cluster in use. First a trace based method for collective operation spanning tree performance analysis is developed. Second, a runtime monitor system is built, and demonstrated to provide the low perturbation and data processing capability necessary for real-time analysis of the traces. Third, the monitor system is used to measure how the latency of globally synchronizing collective communication can be reduced on the WAN multi-cluster.

To get insight into where the bottlenecks of collective operation are and how these can be avoided, messages traces are collected internally in the communication system. These traces provide a detailed chronological view of the applications execution, and are used to calculate statistics and visualizations used to aid in adapting the spanning tree to the cluster resources in use.

To reduce monitor overhead and improve monitor scalability we use several novel techniques. Storing message traces in fixed size buffers, where old records are discarded when a buffer is full, reduces storage overhead. Coscheduling application and monitor threads to exploit underutilized compute and network resources reduces monitor computation and communication overhead, and distributes the analysis of message traces among the cluster nodes. Monitor scalability is further improved by using efficient collective operations for data gathering, and separating functional concerns of the monitor such that different parts can run at different speeds.

The research platform consists of Beowulf clusters used independently, and connected together using a WAN. For these platforms software modifications can significantly improve collective operation performance. We focus on latency bound collective operations since these are most commonly used [231]. Also, latency bound operations are good benchmarks for a monitor, since they are easily perturbed [118, 166], and more trace data is produced since the operations can be frequently called.

1.3.2 Exploiting application knowledge

```
delta = computation();
global_delta =
MPI_Allreduce(delta);
if (global_delta > epsilon)
  break;
```

```
delta = computation();
global_delta =
cond_allreduce(delta,
  LARGER_THAN_EPSILON, epsilon);
if (global_delta > epsilon)
  break;
```

```
MPI_Allreduce(v):
  t = lan_allreduce(v);
  r = wan_allreduce(t);
  return r;
```

```
cond_allreduce(v, type, epsilon):
  if (type==LARGER_THAN_EPSILON)
    t = lan_allreduce(v);
    if (t > epsilon)
      return t;
    else
      r = wan_allreduce(t);
      return r;
```

Figure 8: Standard allreduce (right) and conditional allreduce (left) as used by an application (top) and implemented in the communication system (bottom).

To reduce the latency of synchronizing collective operations used on a WAN multi-cluster, we implement a new operation that can be used for calculating a global value used to make global decisions. Such operations can be used for example to determine when a linear algebra computation has converged, and hence determine when all threads should move to the next stage in the applications algorithm. A call to this operation can return once the condition is determined to be true (Figure 8). This allows reducing the number of messages sent over a WAN, and hence the latency of the operation. The experiment methodology is similar as for spanning tree monitoring described above.

1.3.3 Overdecomposition

To evaluate the benefits of utilizing overdecomposition, the NAS parallel benchmark suite was used on a Beowulf cluster composed of computers with the first generation simultaneous multi-threaded processors (Intel Pentium 4). Also, we measured the benefits of using overdecomposition for WAN multi-clusters. To provide the data necessary for the performance analysis, we use our monitor tool developed for

collective operation analysis, hardware performance counters, operating system counters, and system level profilers.

1.3.4 Content based compression

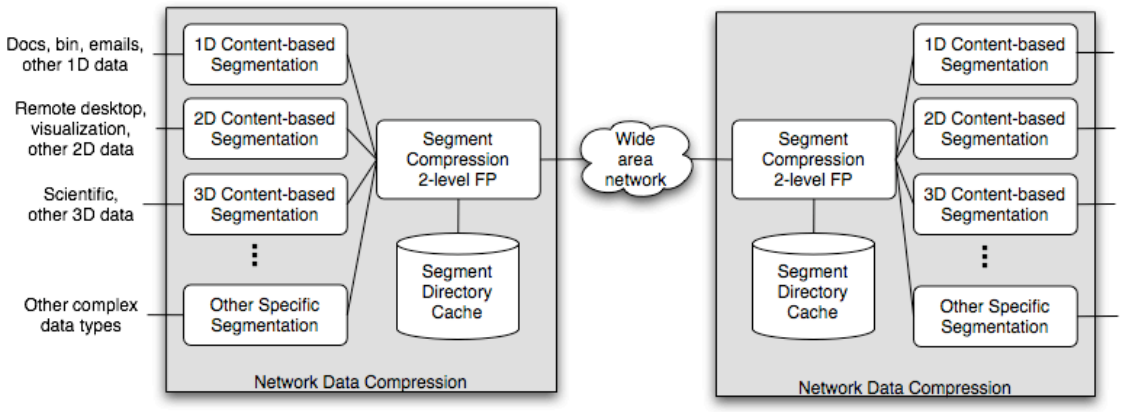


Figure 9: Architecture of proposed compression approach, consisting of components for content-aware segmentation, redundant segment elimination with two-level fingerprinting, and a large segment cache. Applications can choose their appropiate content-based segmemtation method according to their data type.

We propose a network data compression framework called *Canidae* that allows application users to build content-aware redundancy detection methods to improve the compression ratio (Figure 9). Our method is different from previous global compression approaches in four ways. First, data segmentation is separated from redundancy elimination such that specific content-based segmentation methods can be applied to complex data types. Second, we propose a 2-dimensional segmentation approach that works well with remote data visualization data transfers. Third, we employ a two-level fingerprinting method to optimize the encoding of unique data segments. Forth, we use a very large cache for storing segments that allows detecting redundancy in a larger scope.

Using the compression architecture described above, we built a remote visualization system called *Varg* that satisfies all three requirements defined in section 1.2.4. To satisfy the interactive performance requirement, the Varg system implements a novel method to compress redundant two-dimensional pixel segments over a long visualization session. To satisfy the no raw data sharing, and platform independence requirements, the Varg system is based on a platform-independent remote desktop system VNC, whose implementation allows remote visualization of multiple applications in a network environment.

To demonstrate the applicability of the framework for remote visualization, we used the framework to compress data sent by VNC when using several visualization intensive genomic applications.

1.4 Contributions

To improve parallel application scalability on Beowulf clusters we *improve collective operation performance*. Our contributions are:

- A new performance analysis method to identify bottlenecks in spanning trees using message traces collected internally in the communication system. Existing parallel analysis tools has treated the communication system as a black box.
- A monitoring framework that supports the development of tunable runtime monitors for parallel applications. Implementation of several monitors has lead to the discovery of:
    - New method to improve the scalability of message tracing. Trace records are stored in small buffers and processed at the same rate as they are produced by monitor threads distributed and run on the cluster computers.
    - Novel method for reducing the perturbation of a monitor by coscheduling monitor threads with parallel application threads, such that data can be analyzed when the processor is idle.

The implemented monitors were run on Ethernet and WAN multi-clusters with almost no perturbation of the parallel application, and using only a few megabytes of memory for storing the traces. The results demonstrate that trace based analysis is, even when more data produced than for earlier approaches, practical on large scale clusters. Using the monitors we were able to reduce collective operation latency up to 1.5.

Our contribution to further *reduce the latency of globally synchronizing collective operations* on WAN multi-clusters is:

- The novel conditional-allreduce operation used to implement global decisions. In most cases the condition used to make the decision can be evaluated without communication over WANs. Only minimal changes are required to application code, and the result produced by the application does not change.

Conditional-allreduce reduced the latency of the allreduce operations by several orders of magnitude, such that the threads in a parallel application run on a WAN multi-cluster can be synchronized with the same overhead as on a single cluster. This may allow running other than embarrassingly parallel applications on a computational grid.

To reduce parallel application execution time by *improving the parallel efficiency on Beowulf clusters*, we contribute with:

- Method for identifying improved TLP and overdecomposition overheads using data from multiple software and hardware layers.
- The first, to our knowledge, study of the utility of overdecomposed parallel applications run on Beowulf type clusters composed of hosts with SMT processors.

We demonstrate how overdecomposing the NAS parallel benchmarks can reduce execution time. But it may also increase execution time due to TLP not improving, and overheads caused by an increase in context switches and cache misses.

To *reduce the network bandwidth requirements* of distributed applications sending large scale scientific data sets and remote visualization data over a low-bandwidth WAN, we make the following contributions:

- A framework for global compression using two-level fingerprinting and application specific segmentation. Compared to previous systems segmentation is decoupled from redundancy detection such that the same compression engine can be used with different application specific segmentation methods. In addition:
  - A novel two-level fingerprinting protocol that improves redundancy detection by using smaller segments, while maintaining data consistency. Past work used large fingerprints, thus requiring large segments to maintain a high compression ratio.
  - Application specific segmentation method for multi-dimensional data that improves the redundancy detection for complex data types. Previous general- purpose 1-D segmentation algorithms do not take into account the structure and dimensionality of the data.
  - The design and implementation of a very large cache on disk for storing previously sent segments that improves compression ratio. Existing systems use much smaller segment caches.
- A network bandwidth optimized, platform-independent remote visualization system using two-level fingerprinting protocol, and:
  - A novel segmentation algorithm for 2-D pixel data that detects pixel movement on a screen. Existing compression methods do not provide the compression ratio and time necessary for interactive remote visualization over networks with low bandwidth.
  - A novel use of two-level fingerprinting and eventual consistency to reduce the end-to-end latency of segment messages.

Extending the VNC remote visualization system, we can support interactive visualization-intensive genomic applications in a remote environment by reducing bandwidth requirements from 30:1 to 289:1.

1.5 Organization of dissertation

This *first* chapter has presented the background for this dissertation, motivated the problems addressed, presented the methodology used to solve these, and summarized lessons learned solving these. The remaining parts are organized as follows.

The *second* chapter is about collective operation performance analysis, and is based on three papers. In the first we develop and use a post-mortem method for performance analysis (this paper is in Section 7.1). The second paper (Section 7.2) presents the design of a monitor that allows the analysis to be done in real-time. The third paper (Section 7.3) describes and evaluates conditional collective operations.

The *third* chapter is based on a paper (in Section 7.4) that evaluates how overdecomposition improves parallel application performance on clusters.

The *fourth* chapter is about global compression and consists of three papers. The first paper (Section 7.5) describes and evaluates an earlier implementation of the general-purpose compression framework designed for remote visualization of genomic applications. The second paper (Section 8.4) describes and evaluates our general-purpose two-level fingerprinting method. The third paper (Section 8.5) evaluates application specific segmentation methods for 2-D data.

The *fifth* chapter summarizes the dissertation and lists contributions, and the *sixth* chapter outlines future work.

In addition published papers are in Appendix A, while Appendix B contains unpublished papers and experiment results.

Chapter 2

Collective Operation Performance

In the previous chapter we motivated the need for improved collective operation performance. In this chapter three papers for improving collective operation performance are presented and discussed.

2.1 Introduction

The goal of parallel programming is to improve application performance by utilizing many processors. Previous work has found that the number of processors that can efficiently be utilized is for many parallel applications limited by collective operation performance [174, 231]. Therefore, during the last years there have been many attempts to improve collective communication performance including [23, 26, 27, 32, 80, 81, 105, 123, 126, 140, 143, 152, 163, 205, 213, 218, 229, 242]. Recent work has also compared the performance of different collective operation spanning tree configuration [80, 229]. However, a single best configuration has not been found. Instead, the spanning tree needs to be adapted to the application and the platform in use [32, 80, 81, 126, 163, 205, 218, 229].

```
if (msg_size > 100K)
  t=get_flat_tree();
else
  t=get_binary_tree();

use(t);
```

```
if (tested_algorithms
    == n_algorithms)
  t = get_best_tree();
  use(t);
else
  t = get_next_tree();
  s = get_time();
  use(t);
  e = get_time();
  update_perf_data(
    t, e-s);
  tested_algorithms++;
```

```
if (first_time)
  t = get_tree();

use(t);
r = do_perf_analy(t);
t = reconf_tree(t,r);
```

Figure 10: Three approaches for selecting a collective operation spanning tree configuration: static rule (left), latency measurements of predefined algorithms (middle), reconfiguration based on performance analysis (right).

There are at least three methods used to tune collective operation performance (Figure 10). The most common is to implement a few algorithms that create spanning trees of different shape, such as a balanced binary tree. The algorithm to use for a given collective operation call is then selected at run-time based on a few parameters such as the message size (as done in MPICH [5] and LAM/MPI [3]). The selection policy is specified at design time, but an algorithm may adapt the spanning tree to the cluster topology [36, 126] if a topology specification is provided. Recent work has shown that performance can be improved by selecting the algorithm dynamically based on performance measurements of different algorithms run on the actual platform in use [81, 229]. We present a third method in the paper in section 7.1. Rather than selecting an algorithm that was specified at design time, we reconfigure different parts of a

spanning tree based on the results of a performance analysis. The spanning tree can therefore be intelligently adapted to any cluster configuration, and the resulting spanning tree may have a shape that is difficult to express algorithmically. For example, some branches may be binary trees while others are oct-trees.

To detect bottlenecks in a spanning tree it is necessary to collect message traces internally in the communication system. Such traces provide a detailed chronological view of the applications execution, and can be used to calculate statistics and visualizations that provide insight to where time is spent during a collective operation call. The applicability of the method is demonstrated by adapting spanning trees to different Beowulf clusters and WAN multi-clusters based on the results.

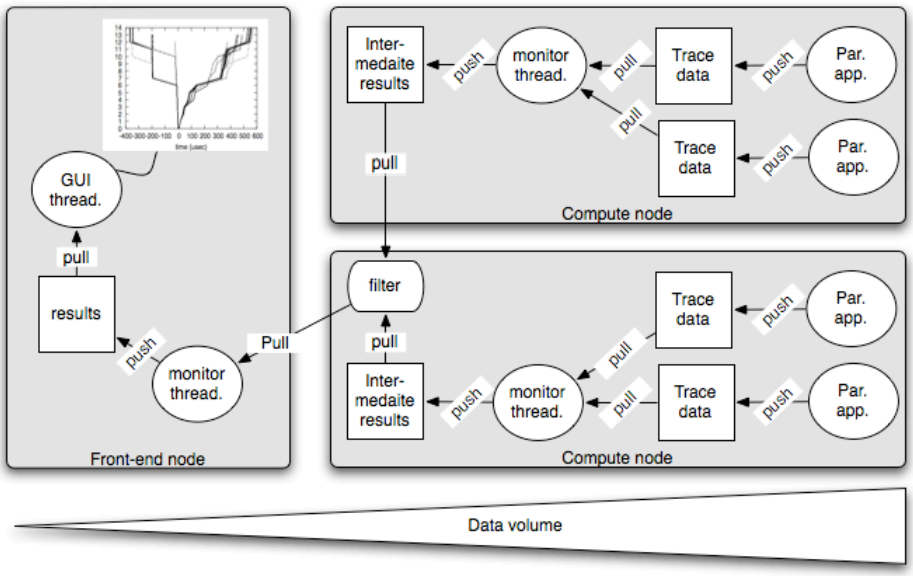


Figure 11: EventSpace architecture. A collective operation spanning tree is instrumented with multiple event collectors that store trace data in bounded buffers. Monitors read data from buffers using an event scope that also filters and reduces the data read.

The performance analysis method presented in section 7.1 is based on *post-mortem* analysis of message traces. Message tracing is a popular performance analysis approach and is used by many tools [55, 99, 111, 115, 119, 144, 178, 194, 241]. But message tracing does not scale to large-scale clusters for two reasons. First, hundreds of megabytes of trace data are collected even on small clusters. Maintaining and storing such large data sets can significantly perturb the monitored application [61, 160, 230]. Second, the data is typically processed sequentially, and the traces recorded on all cluster nodes must therefore be gathered to a single front-end computer for analysis. Transferring the data and running the analysis may take several minutes, and can only start after the application has exited.

To avoid the overhead of existing message trace monitors [55, 99, 111, 115, 119, 144, 178, 194, 241], alternative data reduction approaches have been used. These include: recording higher level data [15, 16, 91, 93, 112, 128, 230], recording data only on a few selected nodes in a cluster [181, 190], and only recording data for a small period of the applications execution time [151]. However, none of these approaches is able to

provide a complete chronological view of the applications execution, which may cause some performance problems not to be detected or understood [61].

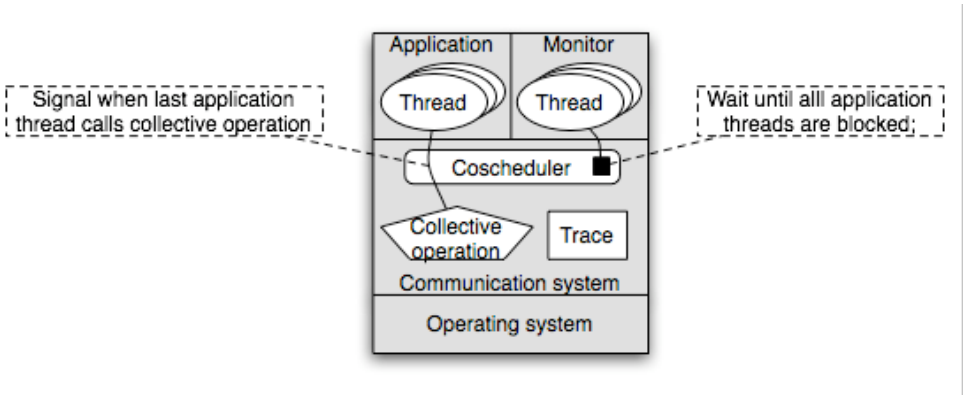


Figure 12: Coscheduling implementation. Message tracing is integrated to the communication system. Monitor threads can be blocked when accessing a trace buffer. Blocked threads are unblocked either before or after an application thread calls a communication operation. Scheduling policies are implemented by specifying when to unblock monitor threads (for example: unblock all monitor threads when all application threads are blocked on a collective operation call).

To solve the problem of high perturbation and storage requirements of existing trace based monitor systems, we developed the EventSpace system that is presented and evaluated in the paper in section 7.2. EventSpace is based on the following design ideas:

- A flexible framework is provided that supports the implementation of many different runtime trace-based monitoring tools.
- The functional concerns of the monitor are separated (Figure 11). Data collection is always enabled, but different analysis tools can use the data. These tools may produce temporarily results gathered and visualized by other tools. More importantly, the performance and perturbation concerns are also separated. This allows tuning each part separately to trade-off between introduced perturbation and data processing capability.
- To reduce memory usage we exploit underutilized compute and network resources to analyze the trace records at the rate they are produced. Trace records can be stored temporarily in small fixed size bounded buffers, since the monitors have sufficient compute power to analyze all records before they are discarded.
- The analysis of message traces is distributed among the cluster nodes. The combined compute power can be used to for compute intensive analysis of the recorded data before records are discarded (two examples of monitors implementing distributed analysis are shown in Figure 18 and Figure 19).
- To reduce perturbation, the monitor threads are coscheduled, as shown in Figure 12, such that monitors can use the processor and network when these are not used by the application.

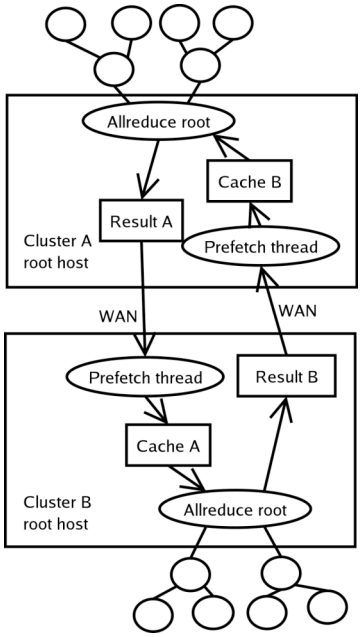


Figure 13: Conditional allreduce implementation for two clusters. First a local result is calculated on both clusters. If the condition is evaluated to be true, the local result is returned. Otherwise, the partial result received from the other cluster is read from the cache, combined with the local result, and returned.

Although spanning tree reconfiguration can significantly improve collective operation performance, the best WAN multi-cluster configuration will still be limited by the WAN latency, and hence be orders of magnitude slower than on a single cluster [126]. Therefore, it is necessary to make changes to the collective operation to further improve performance.

In the paper presented in section 7.3 we evaluate two novel changes to collective operations for further reducing the latency of globally synchronizing collective operations on a WAN multi-clusters:

- Conditional collective operations that can be used to implement global decisions. These are typically used to synchronize parallel application threads. Conditional-allreduce can return once the condition is determined to be true (or false). Many conditions can be evaluated using only a subset of the contributed values. Therefore it is often not necessary to wait for values sent over WANs, and hence the latency of the operation is not limited by the WAN latency (Figure 13).
- For large clusters, noise introduced by operating system activity and system daemons has been shown to contribute significantly to collective operation latency [118, 166, 226] (Figure 14). It may be possible to exploit the noise to overlap communication wait time with computation, and hence hide some of the WAN latency.

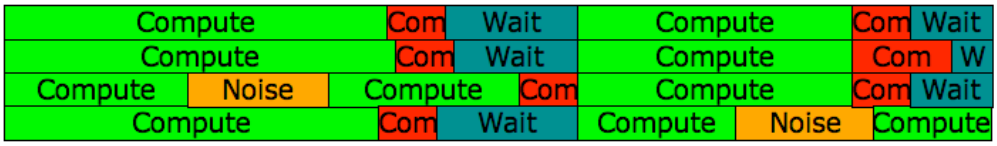


Figure 14: Noise delaying one thread in a parallel application causes all other threads to wait at synchronization points thus increasing the latency of the synchronizing collective operation.

The remaining sections in this chapter are as follows. Section 2.2 summarizes the results of the papers in sections 7.1, 7.2, and 7.3. The results are discussed in section 2.3. Section 2.4 presents related work published after our papers, and section 2.5 concludes.

In addition, a technical report about the WAN emulator used in the papers is in section 8.1, and a summary of an unpublished paper about operating system interference on Ethernet clusters is section 8.2.

2.2 Summary of papers

This section summarizes the results of the three papers.

2.2.1 Collective communication analysis

The first paper (section 7.1) answered the following questions:

- What type of statistics and visualization are useful to understand collective communication bottlenecks?
- Which factors influence the spanning tree performance?
- How should the spanning tree be reconfigured to avoid each type of bottleneck?
- How much can performance improve by reconfiguring the spanning tree?

2.2.1.1 Methodology

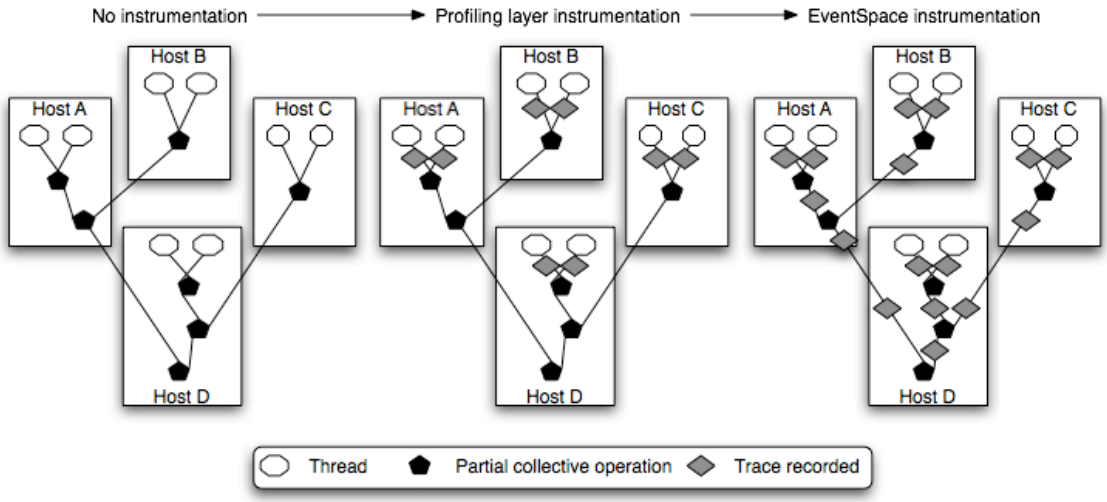


Figure 15: The spanning tree from Figure 10 instrumented using the MPI profiling layer (middle) and our EventSpace tool (right).

Many parallel application monitoring tools record message trace using the MPI profiling layer [149]. However, the collected trace data is not sufficient to understand collective communication problems, since the communication system is treated as a black box. We instrumented the communication system to record message traces internally in the communication system (Figure 15). This provides information about the time spent on each node for the computation and synchronization in the spanning tree. Also, the traces can be used to calculate the communication latency for each message sent. The data collection can be done with very low overhead (less than 3%), but storing the traces requires hundreds of megabytes even for short runs on clusters with tens of processors.

To answer the above questions, we used the collected message traces to analyze the performance of spanning trees on clusters with different number of cluster nodes, processors per node, network latency, and network bandwidth.

2.2.1.2 Experiment results

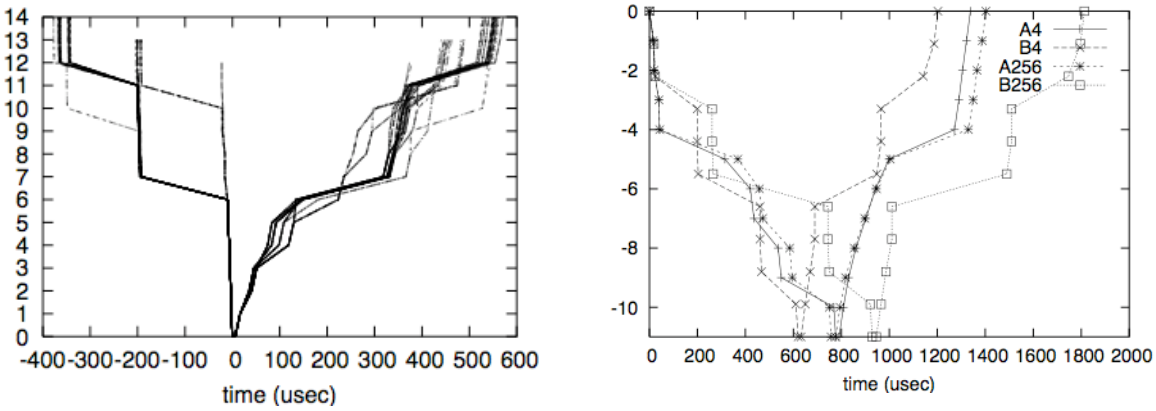


Figure 16: The *pathmap* visualization shows the computation time, wait time, and network latency for each thread (left). The measurement points are shown on the y-axis, while the time spent at each point is shown on the x-axis. The visualization can also be used to compare the performance of different spanning tree configurations (right).

We use the *pathmap* visualization (Figure 16) to detect load balance problems, sort branches in the spanning tree into groups with similar performance, and to detect bottlenecks. The visualization is created by calculating statistics for the time spent for synchronization computation, and communication for each thread.

The latency of the allreduce collective operations consists of three parts. First the *arrival time* is the time between the operation call of first and last thread. Second, the *reduce latency* is the time from the last thread called the operation until the result is at the root of the spanning tree. Third, the *broadcast latency* is the time from the root received the reduced value until the last thread received it. We found the performance to be mostly influenced by the arrival time, synchronization overheads during broadcast, and the network latency.

Predicting how a reconfiguration of the spanning tree will change each factor is difficult, since the factors are not independent. For example, the communication latency depends on the load on the communicating nodes. Therefore, we found it necessary to compare the spanning trees by plotting these in a single pathmap (Figure 16).

To configure a spanning tree, the depth and width of sub-trees are adjusted. If the bottleneck is due to the arrival time, either the application workload must be redistributed, or the spanning tree must be differently mapped to the cluster. If the bottleneck is due to the load on nodes, a deeper tree can be used such that multiple partial-reductions and broadcasts can be done in parallel. Similarly, if communication latencies contribute most, a broader tree can be used to reduce the number of network links. Our results show that performance could be improved up to 1.5 compared to collective communication configurations used by popular MPI implementations.

2.2.2 Scalable low overhead monitoring

In the second paper (section 7.2) the EventSpace monitor system is presented. The evaluation of the prototype answers the following questions:

- How much is the monitored application perturbed when traces are stored in buffers shared by a single writer and multiple readers, and how much memory should be allocated for the buffers used to store the traces?
- How should monitors be implemented and tuned to achieve the necessary performance to analyze the records in a buffer before they are discarded?
- How can monitors be adapted to reduce application perturbation or to increase monitor performance?
- Is perturbation reduced by coscheduling of monitor and application threads to utilize unused cluster resources?
- Does collective operation monitoring scale to very large clusters and WAN multi-clusters?

2.2.2.1 Methodology

We experimented with two collective communication monitors that implement runtime analysis using the performance analysis method described above. The *load balance monitor* provides an overview of how much time each thread spends on collective operations. Many message tracing libraries collect similar amount of data. The *synchronization point and network latency monitor* calculates the statistics required to draw the pathmap visualization (described above). These collect and process more data than most existing trace based analysis tools.

The *load balance monitor* analyses one message trace per thread, and provides results similar to other parallel application monitors (as discussed in section 1.2.1). The *synchronization point and network latency monitor* requires combining and analyzing more traces, and requires more computation.

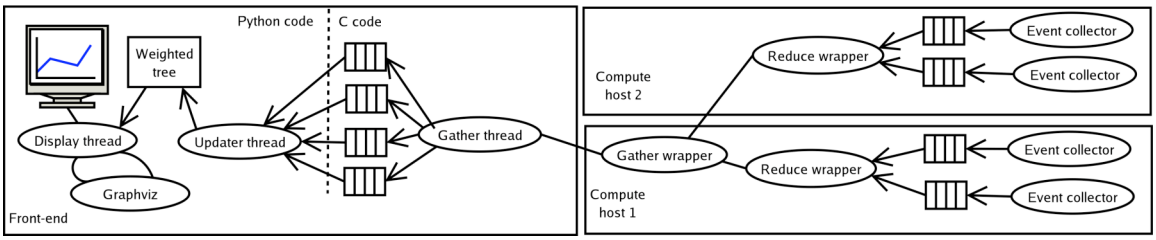


Figure 17: Load balance monitor with centralized trace analysis.

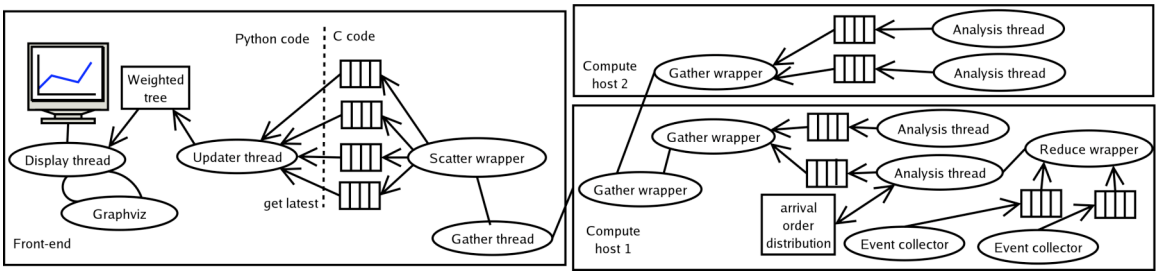


Figure 18: Load balance monitor with distributed trace analysis.

To experiment with different approaches for implementing monitors, the *load balance monitor* was implemented using a centralized and distributed approach. The centralized approach gathers data to a front-end node for analysis (Figure 17). The *event scope* used to gather the data does some filtering and reduction of the traces on the cluster. These are executed close to the data source, and are used to reduce the data volume transferred to the front-end node. In the distributed approach, analysis threads are run on the cluster nodes. These read data from the message traces, calculates derived metrics, and writes these to bounded buffers used to store intermediate results (Figure 18). The intermediate results are then gathered to a single node for presentation.

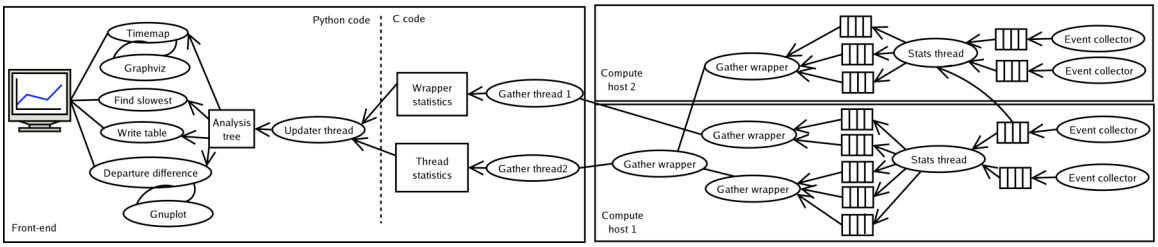


Figure 19: Synchronization point and network latency monitor.

Coscheduling is evaluated using the *synchronization point and network latency monitor* (*statsm* shown in Figure 19). This monitor is implemented using distributed analysis, since the derived metrics require more computation, and in addition some metrics are calculated using trace data collected on two nodes.

Experiment	LAN overhead	WAN overhead
Data collection only	None—1%	None—1%
Synchronization and latency analysis threads	5—9%	
…with co-scheduling	1%	
Centralized load balance monitor	0.4%	1%
Distributed load balance monitor	1%	1--3%
Synchronization and latency monitor	2%	None

Table 4: Application slowdown cause by different monitors.

To measure monitor performance and perturbation benchmarks with frequent collective operation calls were monitored. Such communication intensive benchmarks represent applications for which collective operation performance improvements are most interesting. In addition, monitoring frequent calls will introduce more perturbation and require the highest performance from the monitor. The benchmarks were run and monitored on a Beowulf cluster and a WAN multi-cluster.

2.2.2.2 Experiment results

The experiment results are summarized in Table 4. Data collection to bounded buffers does not significantly perturb the monitored application, since less than 1% execution time overhead is introduced to the monitored parallel application. Also, application execution time does not significantly increase when multiple monitors simultaneously read data from the buffer. The storage requirements are very low. Allocating about 10 megabyte of memory is typically enough to ensure that all trace records are processed before being discarded.

Both the centralized and distributed implementation of the *load balance monitor* is able to analyze over 99% of the data before trace-records were discarded (the buffer size can be increased to ensure that 100% of the data is analyzed before being discarded). The application execution time overhead introduced by the monitors ranged from 0—3%, with the centralized monitor having the largest overhead. With distributed analysis the functional concerns can be separated, such that all data is analyzed at the rate it is produced, but gathered to a node for presentation at a slower rate. Reducing the gather rate can also be used to reduce the perturbation of the monitor.

Without coscheduling the *synchronization point and network latency monitor* introduced 9% overhead to the application. But coscheduling reduced the overhead to 1%.

The monitors scale with cluster size, since the analysis workload is divided among threads that monitoring a fixed size subtree. For a larger cluster, the number of analysis threads is simply increased. The data gathering to a front-end node for presentation also scales well, since the analysis threads can reduce the number of cluster nodes from which data is gathered. For all monitors the time required to analyze the collected performance data is smaller than the latency of the operation. Thus, the monitors can be used even for micro-benchmark consisting of only collective communication.

Monitoring collective operations on WAN multi-cluster is simpler than on a single cluster of the same size, since the analysis computation time is the same but the collective operation latency is larger. Also, the performance for both the collective operation and monitor data gathering is limited by WAN latency.

2.2.3 Conditional allreduce

In the third paper (section 7.3) we experimented with two approaches for reducing allreduce execution time on WAN multi-clusters. The experiments answered the following questions:

- Can collective operation latency be hidden in system noise cause by operating system interference for a multi-cluster with WAN links?
- Does our conditional allreduce operation reduce allreduce latency compared to existing approaches?

2.2.3.1 Methodology

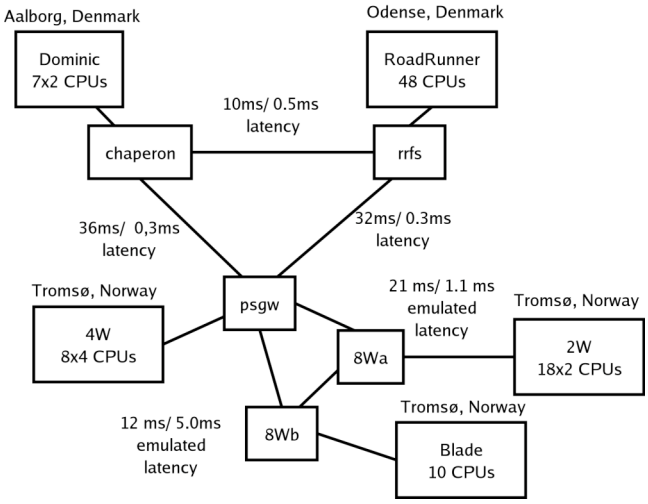


Figure 20: WAN multi-cluster topology used to measure conditional collective operation performance improvements.

For the evaluation four clusters with about hundred nodes in total were used. The clusters were located in Norway and Denmark, and the largest WAN latency was about 35 ms (the topology is shown in Figure 20). The SOR kernel was used for the evaluation, since it represents a typical usage of collective communication for synchronizing the threads in a parallel application.

An allreduce operation implemented according to the MPI standard was, as expected, limited by the two-way latency of the slowest WAN. The wide area network aware collective operation used in [126], only improved the latency by 10%. But other WAN topologies show the expected 50% improvement. But, the latency is still limited by the highest one-way WAN latency in the topology.

2.2.3.2 Experiment results

For most conditional allreduce calls, the condition can be evaluated using values contributed by threads on a single cluster. Hence, most calls can return without waiting for messages sent over WANs, and the latency of the operation is similar to the single-cluster latency. For the remaining algorithm iterations, a message from another cluster is typically enough to evaluate the condition. For these, the latency is limited by the smallest WAN latency (Figure 21). Finally, for the remaining iterations the latency is similar to the WAN aware algorithms.

We find that system noise does not significantly contribute to collective operation execution time for medium size Ethernet clusters (see section 8.2 for details).

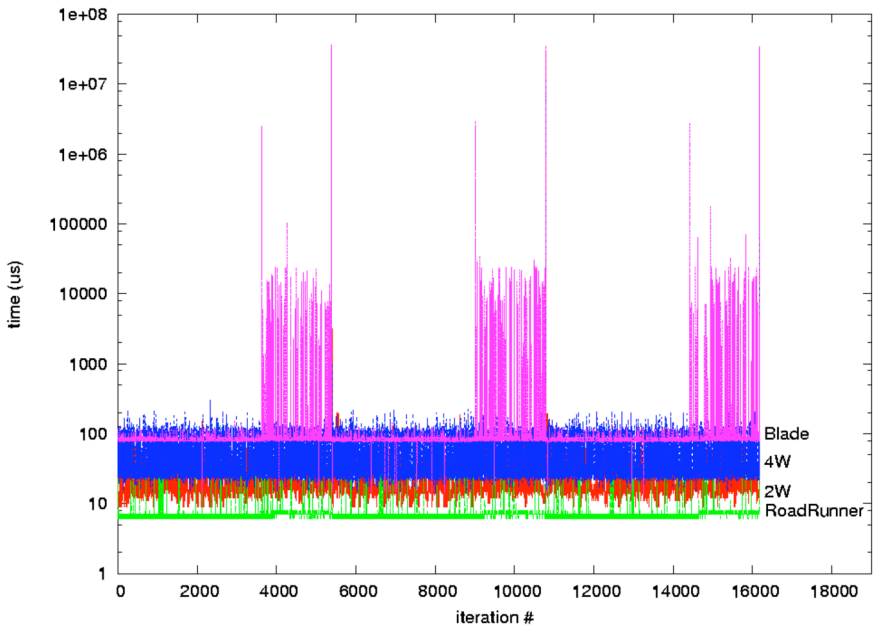


Figure 21: Conditional-allreduce latency for each cluster (results for Dominic are not shown). For most iteration the latency is equal to the LAN latency. But when the algorithm is close to converge, data from other clusters are needed and the latency includes the one-way WAN latency to these clusters.

2.3 Discussion

This section provides a high level discussion of the results presented in the previous section. In addition, the choice of benchmarks used in the papers is motivated at the end of this section.

2.3.1 Collective communication analysis

In a dynamic environment such as many compute Grids, the resources allocated to an application rapidly changes. It is therefore necessary to rapidly reconfigure the collective operation in order to adapt to the changes. We believe the approach demonstrated in the paper in section 7.1 provides the necessary insight to rapidly find the best spanning tree configuration. Our results demonstrate that for some cluster topologies the best spanning tree has a non-uniform shape that is difficult to predict and create using an algorithm. However, we have not documented that using performance analysis to reconfigure a spanning tree can be faster than searching through a repository of algorithms (as is done in [81]).

Collective operation performance analysis requires collecting data internally in the communication system. We used a research prototype communication system [32, 233] where adding such instrumentation was easy. But, in order for the method to be widely adapted the many communication systems implementing the MPI standard must be instrumented. Such instrumentation is realistic for three reasons. First, the recorded information is portable and minimal (recorded are: operation type, message size, start time, and end time). Second, the necessary data can be provided by the new Peruse monitoring interface (discussed in section 2.4). Third, as demonstrated by the EventSpace monitoring system, such data collection has very low overhead.

Another important requirement for the method to be applicable is that the communication system in use supports reconfigurable spanning trees. Recent MPI

implementations such as Open MPI [87] are module based, making it easy to add such functionality (as demonstrated in [35, 36, 117]).

To our knowledge this is the first study that attempts to identify the bottlenecks in collective operations rather than just comparing the performance of several algorithms [80, 229]. Our evaluation also differs in that we attempt to improve a spanning tree by configuring different parts of the tree.

In this work performance analysis was limited to the allreduce collective operation. Other operations that use similar spanning trees can also be analyzed using our method, as shown in section 7.3 were we extended the method to analyze all-to-all message exchange, and in Chapter 3 where we also monitor computation-communication overlap. The method itself is independent of synchronization variable implementations, and communication system implementation, since it is based on visualizations and statistical analysis of synchronization point wait time, network wait time, synchronization call overhead, and the wait time before a calculated result is returned to all callers. But in practice a monitoring system as described in the following section is required.

2.3.2 Scalable low overhead monitoring

To tune collective operations we analyzed low-level message traces. Even on our medium size clusters, a large amount of trace data is produced. But in contrast to earlier assumptions [189] and results [61, 230] we have demonstrated that all data collected for collective operations can be analyzed without significantly perturbing the monitored application, and that the approach is scalable. The approach can also be used on WAN multi-clusters as demonstrated in the paper in section 7.3.

The low perturbation and low storage requirements of our data collection approach suggest that it could be added to a parallel programming communication system and always be enabled. This would provide performance information about the communication system similar to the information provided by the performance counters on processors, the /proc filesystem in the Linux operating system, and the Ganglia [147] tool on clusters running the Rocks distribution [147]. Such an interface may increase the use of parallel application performance analysis tools, since it would no longer be necessary to re-compile or re-link the parallel application to enable data collection. Such an interface allows developing portable performance analysis tools.

Using EventSpace, both simple monitors doing centralized message trace analysis, and more complex monitors doing distributed analysis can be built. Both require gathering data from multiple nodes to a single node for presentation. The data is gathered using collective communication spanning trees. The spanning trees can do data reduction, filtering, and other computations. The advantage of such spanning trees is improved performance, and the approach has been used by many other tools [24, 71, 78, 147, 167, 189, 208].

Distributed analysis handles more data, scales better, and has lower perturbation than centralized monitoring. The reason for the improvements is that the threads are run closer to where the data is produced, and they introduce a level of separation between data collection and presentation. But the monitor is more complex to implement. The load balance monitor could be implemented using centralized analysis, and we believe other analysis tools that provide high-level performance information can be implemented using this simpler approach.

We did not find the monitors spanning tree shape to be important for monitor performance and perturbation (as speculated in [189]). The most efficient technique for reducing perturbation is to add a layer of separation, such that threads running on the compute nodes analyze data at the rate it is produced, but only intermediate results are gathered for presentation at a lower rate. To implement a level of separation, monitors must pull data. Pulling also makes scheduling of a multi part monitor threads easier, since each part can run at maximum speed, independent of the other parts.

Communication intensive applications run on Ethernet clusters are not able to fully utilize the compute and network resources, since parallel application threads are often blocked waiting for data from other nodes, or for collective operations to complete. The idle resources can be utilized for running monitor threads. Allocating separate nodes for monitor tasks (as suggested in [189]), can increase perturbation since on platforms similar to ours, communication requires more processor and network resources than just locally analyzing the data.

To utilize idle time it may be necessary to coschedule application and monitoring threads. The scheduler can be implemented by the communication system if both the monitor and application use it. Such an assumption is realistic since both must access the shared buffers used for storing the traces. The coscheduling results demonstrate that the unused processor resources can be exploited to run other tasks such as monitors, with perturbation similar to other (less powerful) data reduction approaches [230]. To our knowledge coscheduling has not been used for reducing monitor perturbation in previous work. In Chapter 3 we explore another approach for utilizing the idle time.

A limitation of the evaluation is that the monitor approach was evaluated by monitoring collective communication. Point-to-point communication monitoring may have higher perturbation, and require even better monitor performance; especially if a derived metrics is calculated using trace data from both nodes. Also, using a cluster with a faster interconnect can increase the communication operation call frequency and therefore increase the amount of data produced, and hence increase the perturbation and performance requirements. However, even for such clusters there are idle compute resources that can be used by the monitor [129]. In addition the monitor will also benefit from the better communication performance. Finally, the monitored collective operations were not implemented by an MPI communication system. But previous work [35, 36, 117] has shown that the communication system used in the evaluation has collective operation performance which is similar to the popular LAM/MPI communication system [52].

EventSpace provides a library used by analysis tools to access, reduce, and combine data from message traces. The analysis tools themselves are programmed using a combination of C and Python, but the results are presented to the user using a GUI. This is the approach used by most performance tools today. An alternative approach would be to provide a declarative language like SQL that could be used to access the trace data (as demonstrated by data stream management systems [22]).

2.3.3 Conditional allreduce

The latency of the conditional allreduce operation is orders of magnitude lower than ordinary allreduce. But only operations that return a single number, rather than an array, can be made conditional. Also, performance will probably only improve for

many-to-many or many-to-one communication, since the main advantage of conditional operations is that not all participating threads have to be synchronized. However, the remaining operations that either return an array, or have one-to-many communication, are typically used to distribute values used in the computation, and hence cannot be changed without affecting the result computed by the application (an exception is all-to-all where receivers can continue with its computation before the message transfer has completed [106].). Reduce operations are also the most commonly used collective operations [231].

Conditional operations require changes to collective operation semantics. In addition application code needs to be changed to specify an operation as conditional, and the condition to evaluate. We believe a pre-compiler could do this automatically.

Previous work has identified noise caused by operating system interference as an important scalability limitation for application with globally synchronizing operations run on large clusters with high performance interconnects [118, 166, 226]. We did not find noise to have a significant effect on parallel application performance on Ethernet clusters and WAN multi-clusters. Thus, these platforms require different performance improvement approaches. It is therefore important that research is also done on these platforms and not only on clusters with high performance interconnects. Conditional allreduce is one such contribution.

We have focused on WAN collective operation performance in this paper. In Chapter 3, we investigate an approach for tolerating the WAN latency for point-to-point operations. Other approaches for running parallel applications on a WAN are described in [168].

2.3.4 Collective operation benchmarks

In the tree papers presented in this chapter micro-benchmarks and benchmarks were experimented with. Unfortunately there does not exist a parallel benchmark suite targeted at collective operation performance. In this section we briefly examine the benchmarks used in related work [34-36, 80, 81, 123, 143, 163, 242].

The parallel programming community has a long tradition of using a small set of parallel benchmark suites such as Linpack [4], SPLASH-2 [240], NAS [157], the ASCI Purple benchmarks [133] , and the new HPCC benchmarks [1]. Of these, Linpack has been ported to many platforms, but it consists of only one kernel, while SPLASH-2 is oriented toward shared address space multiprocessors. The ASCI Purple benchmarks is a suite that can be used on large scale clusters [231, 232], but the benchmark are coded to use a combination of MPI and OpenMP. Unfortunately, there are few collective communication benchmarks that are widely in use (except the Pallas PMB micro-benchmarks [6]). In some of the recent papers on collective communication optimization [44, 105, 126, 143, 205, 213, 229] three papers [123, 140, 218] use two kernels from NAS. Both use the same type of collective communication. The other papers either use only micro-benchmarks or their own applications.

A collective communication benchmark should spend most of the execution time doing collective communication (Chen and Patterson use a similar argument for I/O benchmarks in [58]). The benchmark should also be realistic. Benchmark with 50% collective communication time relative to execution time, are realistic since it has been found that realistic parallel efficiency can be as low as 40% [129]. In [232] hundreds of processors are required before benchmark execution time was limited by

communication performance. But, many researchers do not have access to such large clusters. Instead the size of the problem computed by the benchmark can be reduced to increase the communication to computation ratio. A large enough reduction will make most benchmarks communication intensive. This does not only apply to our benchmarks, but also for the SPLASH-2 benchmarks [129]. We have verified that reducing the problem size does not change the communication behavior of the benchmarks used in our papers.

2.4 Additional related work

This section supplements the related work presented in the papers and in the discussion in section 2.3.

2.4.1 Collective communication analysis

Above we have presented many of the existing approaches for optimizing collective operation performance (the paper in section 7.1 provides additional details). Below we describe the new MPI Peruse interface, and relate our work to an approach published last year for tuning collective operation performance.

Peruse was motivated by the need for detailed information about the communication system internal activities triggered by MPI calls. The interface has been implemented in Open MPI [124, 211] and PACX-MPI [125]. The Peruse standard exposes events including: start and end timestamp of data transfers, and timestamps for when a message is added or removed from a queue. In addition, a communication system can expose events not in the specification. Peruse can therefore provide the performance data necessary for collective operation performance analysis. Our work complements Peruse in that we demonstrate how to use the exposed data, and we provide a mechanism for efficient collection, analysis, and gathering of the exposed data.

The importance of tuning collective communication operations to adapt to Ethernet clusters has been demonstrated in [80, 81]. In the first paper [80], the algorithms are statically tuned by using two techniques: topology aware spanning trees, and using linear search for empirically selecting a spanning tree algorithm from a repository. The second paper [81] reduces the overhead of finding the best algorithm, by reducing the number of collective operation calls necessary until the best algorithm is found in the repository. Our approach differs in that we attempt to select the next spanning tree configuration based on where the bottlenecks of the current spanning tree are. We have not measured which approach is able to find the best algorithm fastest.

2.4.2 Scalable low overhead monitoring

This section extends the *Related Work* section in the paper in section 7.2 that describes other scalable monitoring systems, data stream management systems, and other uses of coscheduling.

MapReduce [71] is a programming model for processing very large amounts of data on large clusters. The application programmers specify a map function that filters out data by computing a set of intermediate key/value pairs, and a reduce function that merges all intermediate values with the same key. In addition the MapReduce library simplifies parallel programming by handling load distribution, fault-tolerance, and locality optimization. Overall the tasks performed by a MapReduce program are very similar to EventSpace monitors. Both read raw data from a set of buffers/files, filter out data, and aggregate the result. But most EventSpace monitors can eliminate the

mapping stage, since all records in a buffer are typically read, and can therefore be directly reduced. Hence, the model is more similar to data stream management systems (discussed in section 7.2).

User defined reduction functions that produce derived metrics have also been used to reduce the data volume and control the overhead caused by writing trace data to files in the MPI tracing tool [61] developed for the Blue Gene / L system [11]. We have, in addition to providing a clearer separation of data reduction and data collection, demonstrated how to coschedule data reduction with parallel application threads.

Another approach for improving the scalability of message tracing is to compress traces to reduce data volume. Near constant size representation in a scalable manner is possible if only the temporal-ordering for communication events is stored [160]. However, such compressed traces do not contain information about the latency of communication events, and hence may miss many performance problems.

Many approaches have been suggested to utilize idle cluster resources. Cycle harvesters such as Condor [138] or the V System [215] or SETI@home [8, 236] use idle compute resources on remote computers for running jobs. These systems detect coarse-grained idle periods and then start the job as a low priority process. But, utilizing idle time on a cluster running a parallel application requires more fine-grained control over resource usage. Therefore, virtual machines have been used for space sharing parallel application with interactive applications on a cluster [136]. A recent approach with similar goals as our coscheduling is the kernel level idletime scheduler [77] that attempts to run a background job when the resource is idle. Idle time scheduling reduces the perturbation of the foreground job by waiting a short period until starting the background job, thus allowing foreground jobs that arrive during the period to be serviced immediately. The EventSpace coscheduler is simpler than these general-purpose virtual machines and schedulers, since the parallel application and its monitors are closely tied together, and the idle times are caused by communication operations with predictable latency.

2.4.3 Conditional allreduce

The paper in section 7.3 presents approaches for tolerating the higher latency and lower bandwidth of WANs, other architecture specific collective operations, and other distributed systems for group communication over WANs.

2.5 Conclusions

This chapter presented a post-mortem method for collective operation analysis, a framework for implementing scalable runtime monitors for such analysis, and the conditional allreduce operation.

Reconfiguring spanning trees based on performance analysis improved collective operation performance up to 1.5, without changing the application code or the communication system code. The method is based on analyzing message traces collected internally to the communication system, and can hence provide insight about why some collective operation is a bottleneck rather than just identifying it as a bottleneck. The novel *pathmap* visualizations can be used to group spanning tree paths with similar performance in order to reduce the number of paths to analyze, and to detect bottlenecks.

We have demonstrated that it is possible to combine message tracing with very low storage requirements, by storing traces only temporarily in small buffers in memory. The buffer content is analyzed by monitor threads distributed and run on the cluster nodes. The monitor can be used on very large scale cluster and Grids, since collective operation monitors scale better than the monitored collective operations. Monitoring activity can be coscheduled with application activity to utilize unused processor and network resources. Thus, the monitoring overhead is very low; only introducing 0—3% execution time overhead, and hence not significantly perturbing the monitored application.

When the result of a collective operation is used for testing a condition, many operations can complete without sending messages over a WAN. Thus the latency of the operation is reduced by orders of magnitudes and matches the latency of the operation on a single cluster. The changed collective operation will not introduce any changes to the result produced by the parallel application.

The presented methods can be composed to improve collective operation performance on clusters and multi-clusters with high latency networks, thereby allowing more parallel applications to be run efficiently on these platforms.

Chapter 3

Overdecomposition

This chapter presents results from a paper and some additional experiments, about the benefits and limitation of utilizing overdecomposition for parallel applications run on Beowulf clusters and WAN multi-clusters. Overdecomposition is most useful for overlapping point-to-point communication wait time, and hence complements the approaches for improving collective communication performance presented in the previous chapter.

3.1 Introduction

Through the last decade computational scientists have been used to a 1.8 annual performance growth (based on the top500 list [223]). The growth has been due to improved single processor performance, and supercomputers being built with an increasing number of processors. But the growth is limited by the relatively smaller improvements in memory access latency and inter-node communication latency, both leading to under-utilization of processor resources (today a 70% processor utilization is considered very good [166]).

One classic solution for hiding I/O or memory wait time is to overlap the wait time with computation for another task. Memory access latency can be overlapped with computation if a several threads can be executed in parallel on a processor. The technique is called thread level parallelism (TLP) and has since 2002 been supported by processors from Intel, Sun, IBM and others, either by simultaneous multi-threading (SMT) or by chip multiprocessors (CMP). Similarly, network latency for inter-process communication can be overlapped with computation for another thread if parallel applications are overdecomposed (as discussed in section 1.3.2). Overlapping computation with communication can improve parallel application performance up to 2.0 [29, 70, 173, 207]. Overdecomposing the problem into more tasks than there are processors is an easy way to achieve overlap, but it is usually not used since it introduces overheads due to increased communication volume [66], cache pollution [170], and coarse-grained context switches. With processors supporting CMP and SMT, these assumptions may no longer hold. A re-evaluation of the benefits and limitation of overdecomposition is therefore needed.

The paper in section 7.4 provides a performance analysis of overdecomposed NAS parallel benchmarks run on the first generation of SMT processors (Intel Pentium 4). The results provide insight into necessary system software changes to take advantage of the difference in context switch granularity, cache configuration and communication latencies for CMP with SMT nodes. In addition, section 8.3 provides additional WAN experiment results, and user-level scheduling to reduce the overheads.

The rest of the chapter consists of a summary and discussion of the results in respectively section 3.2 and section 3.3. Additional related work is discussed in section 3.4, and section 3.5 concludes.

3.2 Summary of paper

The paper in section 7.4 and the additional experiments in section 8.3 answers the following questions:

- Does applying overdecomposition reduce the execution time of parallel applications run on Ethernet clusters and WAN multi-clusters?
- What characterizes parallel applications for which overdecomposition improve performance?
- Can overdecomposition be used to exploit the multi-threading support of SMT processors?
- Which overheads significantly limit the performance of overdecomposed applications?
- Does system software limit the performance of overdecomposed parallel applications?

3.2.1 Methodology

To answer the questions above, we measured the benefits of utilizing overdecomposition for the SOR benchmark and the NAS parallel benchmarks. The benchmarks were run on an Ethernet cluster comprised of computers with the first generation simultaneous multi-threaded processors (Intel Pentium 4). SOR was chosen since its communication behavior should be well suited for overdecomposition. The NAS benchmarks were chosen since they represent a wide variety of communication behavior (as described in [239]).

The performance analysis is non-trivial since data collected from multiple software and hardware layers must be combined to provide a macro view of system behavior. TLP, communication wait time, and synchronization wait time are all calculated using message traces. Context switch and synchronization system call time are measured by the operating system, while hardware performance counters provide cache miss counts.

Several simplifications of system behavior are made when quantifying TLP and the different overheads. The error can be estimated, and we found that the calculated values for the cache miss overhead and operating system time tend to be larger than the real values with a high degree of overdecomposition. The overheads calculated for a specific node vary between experiment runs, since communication wait times for a given node change between runs. But this variation can be avoided by calculating average overheads for the entire cluster.

3.2.2 Experiment results

Applying overdecomposition improves performance for SOR, and some of the NAS parallel benchmarks (Table 5). The execution time changes range from a slowdown of 1.69, to a speedup of 1.8. In addition, we found that most of the NAS benchmarks have low parallel efficiency, even if they were written using non-blocking communication operations (the MPI standard does not require that the non-blocking operations must be overlapped with computation [237]). Hence, most have potential for computation-communication overlap.

Benchmark	Number and size of messages	Collective operations	Asynchronous messages	Best speedup
SOR (LAN)	Few large	Yes	No	1.4
SOR (WAN)	Few large	Yes	No	1.8
BT	Many small	No	Yes	0.98
CG	Many small, few large	Manual	Yes	0.59
EP	Few small	Yes	No	1.74
FT	Few large	Yes	No	1.11
IS	Few large	Yes	No	0.91
LU	Many small	No	No	1.07
MG	Many medium	No	Yes	0.70
SP	Many medium	No	Yes	0.64

Table 5: Communication behavior, and the overdecomposition improvements for the SOR and NAS benchmarks. Small messages are less than 1 KB, large more than 1 MB. For benchmarks with collective operations and asynchronous messages these typically contribute most to the communication time. Improvement is relative to the one thread per processor composition.

The benchmarks for which performance improved had a variety of communication behaviors. Best improvements were for benchmarks with few blocking operations, low cache miss penalty to execution time ratio, and low parallel efficiency. The improvements were better for the WAN multi-cluster. For all benchmarks a low degree of overdecomposition gave the best improvement, since increasing the number of threads may not significantly improve TLP, but increased cache miss and system call overheads.

Overdecomposition can be used to exploit SMT processors, since the performance improvements were better when SMT was enabled. But the improvements were limited by the lack of TLP. The main advantage of SMT is that some of the system activity could be run in parallel with the computation.

Benchmark	SOR	BT	CG	EP	FT	IS	LU	MS	SP
Processor saturated				✓				✓	
Lack of TLP	✓	✓	✓		✓		✓	✓	
Cache misses	✓	✓	✓				✓		✓
TLB misses									
System activity	✓	✓	✓				✓	✓	✓
Global synchronization	✓					✓			

Table 6: Overdecomposition performance limitations for the SOR, and the NAS benchmarks.

Table 6 summarizes the factors limiting performance improvement. The main limitations are:

- Some benchmarks have a globally synchronizing collective operation between a communication intensive and computation intensive phase in the application, making it impossible to overlap these phases.
- Overdecomposition does not always improve TLP. Often only a single thread is computing, while the others are blocked on communication operation. Thus,

the processor may not be fully utilized even with a high degree of overdecomposition.

- The overheads due to cache misses and system calls can increase more the computation-communication overlap.
- The largest increase in cache misses is for the L2 and L1-data caches, and is due to additional memory copies required for intra-node communication. The intra-node communication also causes additional context switches that increase the system time.

System software does influence the performance improvements achieved when utilizing overdecomposition. Especially important is the behavior of the intra-node synchronization mechanisms. We compared two Pthread libraries and found that TLP improves if the synchronization variables are implemented such that a unlock call is likely to cause a context switch. But this also increases system time. Due to the low TLP, user-level scheduling or operating system scheduling does not significantly influence performance. Section 8.3.4 provides additional details.

3.3 Discussion

Overdecomposing the problem is probably the simplest technique for computation-communication overlap for parallel applications, since it requires no changes to the application code or the communication system. Our results both confirm and contradict earlier results (and common knowledge) about the benefits of this technique. Overdecomposition increases cache misses [170], but not for all applications. Communication and synchronization overheads increase [66], but idle time can be exploited to tolerate the increase. A surprising result was that increasing the number of threads does not always improve TLP. Overdecomposition may improve parallel application performance but should not be used indiscriminately since performance can be unchanged or even decrease. Therefore performance analysis as demonstrated in the paper is useful for identifying the bottlenecks of overdecomposed application and the system software.

We found four main factors limiting overdecomposition improvements. Avoiding an increase in cache misses requires either rewriting the application or closing the memory gap on the processor. For the remaining three, changes to system software may reduce the problem:

- To improve computation-communication overlap the conditional collective operations described in section 7.3 can be used to relax the synchronization of threads.
- Intra-node communication and synchronization overhead can be reduced by using more efficient mechanisms as shown in [141, 177, 213, 218, 228].
- TLP can be increased by ensuring that threads blocked on synchronization variables are started immediately (at the cost of work conservation).

The performance improvements achieved for our parallel benchmarks run on medium size clusters are smaller than reported in earlier studies that used either a simulator [142, 206], or a single SMT processor [146, 227]. Running parallel applications on SMTs have two problems. First, the typical parallel application is memory intensive and uses floating point computations, which have been shown to perform worst on SMTs [108, 121]. Second, threads are often blocked on communication operations,

and thus there is not enough TLP to utilize the dual-threaded SMT processors on the cluster.

SMT support on the processors can be disabled. Related work [191] has shown that web server performance can decrease when enabling SMT, due to more synchronization in the kernel. Enabling SMT did not decrease the performance of any of our benchmarks; hence we believe parallel applications should be run with SMT enabled, even when applications are not overdecomposed.

Very few of the processors succeeding the Pentium 4 have supported SMT, and currently in the second quartile if 2007, none of the high-performance processors available from Intel and AMD have SMT support. However, almost all of the processors are CMP. The design of the CMP processors is still evolving, especially with regards to the cache hierarchy. Performance analysis as demonstrated in this work is therefore important to understand how parallel applications utilize the processors (and the cache hierarchy).

3.4 Additional related work

The *Discussion and related work* section in the paper in section 7.4 discusses other uses of overdecomposition, performance improvement results from other SMT studies, proposed system support for SMT, and alternatives to overdecomposition. This section summarizes additional related work for modeling communication-computation overlap, using one-sided communication operations to achieve the overlap, and schedulers for parallel applications.

3.4.1 Computation-communication overlap

A model for identifying potential computation-communication overlap in a parallel applications is presented and used to achieve speedups ranging from 1.1—2.0 in [134]. The limitations of overlapping are demonstrated in [173], where it is found that the best possible speedup is 2.0, but much smaller in practice. A compiler transforming a parallel application to achieve computation-communication overlap on Ethernet clusters improved benchmark performance from 0—33% when run in 8 processors [139]. Based on our experience, modeling the impact of overdecomposition is difficult due to the complex behavior of the cache hierarchy on SMT and CMP processors, and since many components in the underlying systems are black boxes with unknown system behavior. We complement the earlier models by presenting a method for performance analysis using data collected for real applications run on contemporary cluster hardware.

A recent study of large-scale scientific applications found that there is a large potential for computation-communication overlap that allows hiding most of the communication latencies in clusters equipped with fast interconnects [195]. Also, overlapping is most useful for latency bound applications, and allows reducing the requirements for network latency (with a few microseconds). Our results differ in that the communication latency we attempt to overlap is orders of magnitude larger due to the network interconnects used, and hence different approaches for overlap may be required.

The decomposition approach can have a large impact on cache utilization. A decomposition taking data dimensionality into account reduced the communication volume, but decreased application performance since data in non-continuous memory locations may be transferred [170]. Cache utilization can also be decreased when

applying overdecomposition. We find that overdecomposition increases cache misses for most benchmarks, but that for many the increase can be tolerated since it can be overlapped with communication wait time.

3.4.2 One-sided communication operations

MPI applications can be programmed for communication-computation overlap using the MPI *immediate* operations [148, 149]. But many MPI implementations does not support the required level of computation-communication overlap necessary to achieve the expected performance improvements [70, 237]. We experienced similar limitations for the NAS parallel benchmarks using the popular LAM/MPI implementation.

To improve the level of computation-communication overlap the communication activity can be offloaded to a separate processor, typically located on the network interface card. In addition, one-sided communication operations, such as *put* and *get* in MPI [148, 149] can be used to decouple communication from synchronization. Due to semantic limitations of MPI operations [41], recent work has used primitives provided by parallel programming languages such as UPC [29, 70].

The one-sided communication operations can hide the software overhead of synchronizing the sender and receiver required when sending large messages using a rendezvous communication protocol. For example, a parallel application can be transformed to overlap many-to-many collective operation communication activity with computation [70]. First, the application code is modified such that computation occurs in blocks. Second, the synchronous collective operation is replaced with asynchronous point-to-point operations that are called at the end of each block.

One-sided communication operations improved the performance of the NAS FT benchmark up to 1.9 [29]. The speedup we achieved with overdecomposition was smaller for this benchmark (1.1), but required no changes to application source code or system software. Overdecomposition can also be applied to any application, including applications with point-to-point communication, while the approach in [29] requires computation and communication to be structured as described above. Also, with overdecomposition there is no need for a separate processor for running communication activity, since communication wait time is overlapped, rather than the computation required for communication.

3.4.3 Schedulers for parallel applications

Often operating system schedulers are general purpose, and may not be optimized for parallel applications. For example the Linux 2.6.8 scheduler is optimized for fast responsive time at the cost of a larger operating system overhead [10]. Thus, a domain specific scheduler may be required. For parallel applications the scheduling can be:

- **Global**: where all application processes and system daemons on a cluster are scheduled together to reduce synchronization wait time [118, 166]. A fast global synchronization operation is required, which is not available on interconnects such as Ethernet, or WANs.
- **Between communicating nodes**: where communicating processes are scheduled to run simultaneously to improve latency by avoiding context switches [20, 59]. The protocol overhead of TCP/IP, typically used in Ethernet and WAN clusters, is too high for this type of coscheduling.

- **On a single node**: where the operating system scheduler is modified. For example the SMT aware SOS scheduler was able to improve NAS benchmark performance up to 17% compared to a non-SMT aware scheduler [206]. Also, such schedulers may provide performance isolation, such that several services can share the same server [130]. Performance isolation may also be provided by using virtual machines [136]. As demonstrated in this paper, performance isolation for parallel applications is more difficult since performance degradation due to cache pollution and increased latency of communications operations must also be taken into account. To our knowledge, these problems have not been addressed in related work on performance isolation. Another alternative is idle time scheduling [77] (also described in section 2.4.2) that runs low priority jobs when a resource is idle.
- **For all nodes on a cluster:** Overdecomposition can also be used for load balancing in heterogeneous environments by adding more threads to faster processors [33, 83].

3.5 Conclusions

We have measured how overdecomposing parallel applications into more threads than there are processors; can be used to overlap communication wait time with computation in order to reduce execution time. This was, to our knowledge, the first performance study of overdecomposition used on processors supporting simultaneous multi-threading (SMT). In addition, we describe three user-level scheduling approaches for overdecomposed parallel applications.

We find processors in Beowulf clusters to be underutilized due to communication wait time, even when the parallel applications are programmed to use non-blocking communication operations. Initial results using the SOR kernel were promising; with execution time improvements up to 1.8. The best improvements were for the WAN multi-cluster. However, execution time decreased for only two NAS benchmarks, and decreased for three. Performance improvements are limited by lack of TLP, and overheads due to context switches and cache misses. TLP is limited by application communication structure, and synchronization variable implementation. User-level scheduling did give a small performance improvement, but the effect is often limited by the lack of TLP.

Due to its simplicity overdecomposition can easily be applied for parallel applications with low parallel efficiency. But to understand the improvements and limitations for an application run on a given parallel platform, a performance analysis as used in this work is necessary. To fully utilize overdecomposition, we believe changes to underlying system are necessary to maintain a high degree of TLP and provide efficient intra-node communication and synchronization.

Chapter 4

Content Based Compression

In Chapter 1 the need for better compression methods for scientific and multi-dimensional network data were motivated, and the background necessary to understand global compression algorithms was presented. This chapter presents three papers describing a framework built to provide such compression.

4.1 Introduction

The goal of compressing network data is to reduce transfer time. The compression system should therefore reduce the number of bytes to transfer in less time than would be required to transfer the bytes. The local compression algorithms typically used for network data [9, 188, 202] do not efficiently compress scientific data, while the compression time is too high for interactive remote visualization. Therefore, global compression has been suggested to improve compression ratio and to reduce compression time.

Previous global compression methods [64, 72, 76, 153, 156, 171, 180, 196, 197, 209, 221, 222], are limited to deal with one-dimensional byte streams and have not addressed the issue of how to compress multi-dimensional data. In addition, these typically use large fingerprints to avoid data inconsistency caused by different segments having identical fingerprints. However, since the global compression ratio is limited by the ratio of the average pixel segment size to the fingerprint size, using large fingerprints reduces compression ratio.

To address these problems, we first built a remote visualization system called *Varg,* for which we propose a 2-dimensional segmentation approach that works well with remote data visualization data transfers. Then we generalized the Varg approach into a network data compression framework called *Canidae* that allows application users to build content-aware redundancy detection methods to improve compression ratio (the architecture is shown Figure 9 in section 1.3.4). In Canidae, data segmentation is separated from redundancy elimination such that specific content-based segmentation methods can be applied to complex data types. To solve the problem of fingerprint size limiting compression ratio, we employ a two-level fingerprinting method to optimize the encoding of unique data segments. Finally, to improve redundancy detection we use a segment cache capable of storing hundreds of GB of segment data.

The remaining of this section introduces the Varg system, application specific segmentation, two-level fingerprinting, and the segment cache. Then follows a summary of the papers, a discussion about the limitations and impact of this work, and description of related work. The final section concludes.

4.1.1 Varg remote visualization system

Our initial work was motivated by the need for remote collaboration tools to assist interactive collaborative analysis of microarray data in biology and bioinformatics [53, 68, 102, 137, 161, 199, 210]. Such tools should:

1. Provide fast response times for visualization-intensive genomics applications visualized over a low-bandwidth wide area network.
2. Eliminate replication of large and often sensitive datasets.
3. Work with any microarray analysis software.
4. Be platform-independent.

Most thin-client remote visualization systems [19, 25, 67, 200] satisfy the second and third requirements. In addition the open source VNC system also satisfies the last requirement, since implementations exists for most popular platforms. However, VNC does not provide interactive performance over a WAN.

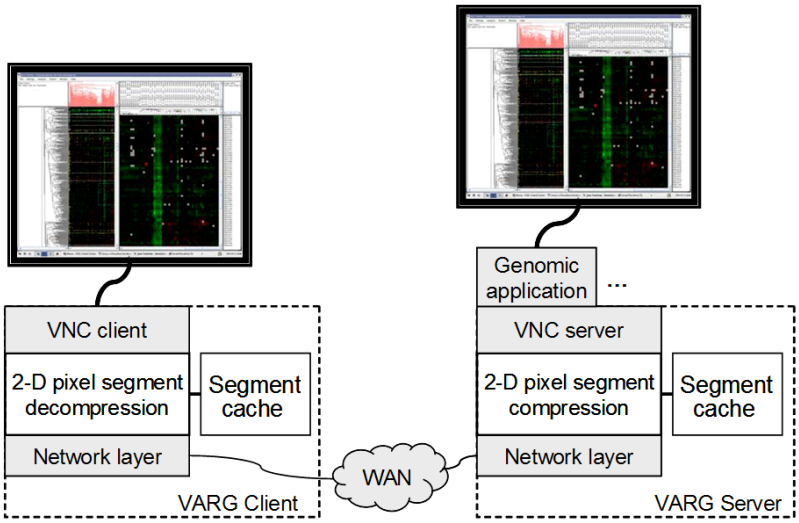


Figure 22: Compression system for remote visualization, consisting of a genomic application remotely visualized, the VNC remote desktop server, VNC client, 2-D bitmap aware redundancy detection, and 2-phase fingerprinting.

The paper in section 7.5 describes the design and implementation of a remote visualization system called Varg that implements a novel method to compress redundant two-dimensional pixel segments over a long visualization session (Figure 22). The Varg system is based on VNC, whose implementation allows remote visualization of multiple applications in a network environment.

The basic redundancy elimination algorithm is straightforward and its high-level idea is similar to previous studies on using fingerprints as identifiers to avoid transfer of redundant data segments (as described in Chapter 1).

The algorithm for segmenting a 2-D array on the Varg server is:

- Save a copy of the 2-D array.
- Receive a set of updated regions of pixels from the VNC server and apply the updates to a local 2-D array.
- Segment the 2-D array into 2-D pixel segments.

- Do region differencing by comparing the pixels in each region to the content in the saved 2-D array.
- For each segment, compute its fingerprint and use the fingerprint as the segment's identifier to lookup in the server cache. If the segment has not been sent to the Varg client previously, compress the segment with a local compression method and send the segment to the client. Otherwise, send the fingerprint instead.

The algorithm on the Varg client is:

- If the received data is a 2-D pixel segment, decompress it with a corresponding algorithm, write the fingerprint and segment to the cache, and then pass the segment to the VNC client
- If the received data is a fingerprint, retrieve the segment of the fingerprint from its cache and then pass the segment to the VNC client.

In addition we optimize the basic algorithm to reduce the user perceived end-to-end latency, by using a two-phase fingerprinting algorithm. With the optimization the server may send two sets of updates, the first based on optimistic fingerprints that can have collisions, and the second set of updates consisting of corrections in case of short fingerprint collisions. End-to-end latency is reduced since updates are sent before the more computation intensive check for collision has completed.

The redundancy detection method for anchoring 2-D pixel segments in the Varg system uses two important properties of genomic visualizations that create opportunities for content-based anchoring. First, microarray datasets tends to be very large. Second, due to the limitation of display scale and resolution, only a small part of the microarray can be viewed at a time, causing the frame to be frequently scrolled. Thus, the same set of pixels will be moved across the display multiple times. Our algorithm combines the statically divided screen approach used in MPEG [88], with Manber's technique [145] of applying a Rabin fingerprint filter [50, 175] for content based anchoring (both anchoring approaches were described in section 1.2.5). First we determine whether most content was moved vertically or horizontally. For predominately vertical motion we statically divide the screen into *m* columns (*m* times screen height) and divide each column into regions by selecting anchoring rows. The columns are then divided into regions by selecting anchoring rows based on Rabin fingerprints. If we detect predominately horizontal motion instead, the screen is transposed before the segmentation algorithm is run.

4.1.2 Canidae general purpose compression system

The Varg system has four problems. First, the segments are large in order to achieve high total compression ratio (due to zlib compression requiring large regions). Second, the two-phase fingerprinting protocol sends an optimistic fingerprint followed by a conservative fingerprint for each segment thereby reducing per segment compression ratio. Third, the segment cache is in memory, and its size is therefore limited by the main memory size on the computer. Fourth, the sender keeps track of which segments have been sent to the receiver, such that recovering after a crash requires synchronizing the caches on both sides.

The Canidae system, presented in the papers in sections 8.4 and 8.5, solves these problems. It consists of multiple segmentation components and a generic compression sub-system that handles the fingerprinting, transmission and caching of segments. To

solve the first and second problem, we employ a two-level fingerprinting method to optimize the encoding of small data segments. The third problem is solved by storing segments in a large cache on disk. The fourth problem is solved by making the sender stateless, and storing received segments and all other segment cache data structures on disk.

4.1.2.1 Application specific multi-dimensional segmentation

The Canidae architecture makes segmentation methods data specific. A segmentation component implementing a method can be configured to one or more ports of the system and to support a variable number of data streams of different data types. Each segmentation component is responsible for the segmentation of a specific class of data. The segmentation component implements the segmentation mechanisms for both send and receive data. For send data, the input data stream is divided into segments and passed to the segment compress component. For receive data, the segments are assembled into a data stream. In addition, the segment components must parse the application protocol to retrieve multi-dimensional data to be anchored.

The main challenge when implementing a segmentation module is to employ a segmentation strategy that will give the greatest likelihood of uncovering, and hence eliminating, redundancies within the data. This will require significantly different segmentation techniques depending on if the data is a 1-D bytestream, 2-D visual display or 3-D scientific data. One example is the Varg 2-D content-based segmentation algorithm. The paper in section 8.5 describes several other 1-D and 2-D segmentation algorithms.

4.1.2.2 Two-level fingerprinting

The two-level fingerprinting protocol in Canidae provides a solution to the problem of compression ratio being limited by data redundancy found and the segment size to fingerprint size ratio. Using smaller segments typically improves the amount of redundancy detected [100], but requires using smaller fingerprints to maintain a high compression ratio. However, to ensure data consistency the fingerprint size must be large enough to uniquely identify a segment. Therefore previous global compression systems [39, 69, 72, 100, 153, 156, 171, 172, 222] typically use a 160-bit such as SHA-1 [7], or even longer secure hash, as a fingerprint so that the probability of a fingerprint collision is far lower than a hardware bit error rate. But this also required using segments of several kilobytes in size. To allow smaller segments to be used in order to maximize the global compression ratio and maintain a low probability of fingerprint collision, we propose a two-level fingerprinting strategy.

The two-level fingerprinting organizes segments into groups. For each group of segments, a 160 bit SHA-1 hash is computed as the *conservative fingerprint* of the whole group. For each segment in the group, we compute a 40-bit FNV hash [86] as the *optimistic fingerprint*.

The two-level fingerprint algorithm is as follows:

- For a group of segments received from the segmentation component the *sender* computes an optimistic fingerprint for each segment, and a conservative fingerprint covering all segments in the group. The optimistic fingerprints and the conservative fingerprint are then sent to the receiver. The sender also stores the segments in a buffer for sent segments.

- The *receiver* uses optimistic fingerprints as segment identifier to look in the segment cache to see if it has received these segments previously. If there is an entry in the segment cache for a given fingerprint, it retrieves the segment from the cache and adds the segment data into an assembly buffer. Otherwise, a segment request message is sent to the sender.
- When the *sender* receives a segment request, it reads the segment from the sent segments buffer and sends the segment to the receiver.
- The *receiver* inserts received segment data to the segment cache, and copies the segment data to the assembly buffer.
- The *receiver* computes a conservative fingerprint for a group of segments when all has either been read from the cache, or received from the sender. This fingerprint is then compared to the received conservative fingerprint. If the conservative fingerprints do not match, all segments that were read from the cache are requested from the sender. If the conservative fingerprints are identical, or all requested segments have been received, an ACK message is sent to the sender, and all segments in the group are sent to the segmentation component to be assembled into the output data stream.
- When the *sender* receives the conservative fingerprint ACK message, it deletes all segments in the group from the sent segments buffer.

$$\text{Compression ratio} = \frac{\text{Uncompressed data}}{\text{Non redundant segments + fingerprints + collision segments}}$$

Figure 23: Factors influencing two-level fingerprinting compression ratio.

The compression ratio of two-level fingerprinting is determined by the data redundancy found, the number of fingerprint bytes sent, and the segment data sent due to collisions (Figure 23).

4.1.2.3 Segment cache

The basic operation of the segment cache is to read and write segments based on their optimistic fingerprint. The two main design goals are to make it large enough to hold all previously sent segments in a session, and fast enough not to limit the throughput of the compression pipeline. For a hundred gigabyte dataset, the total size of cached segments exceeds main memory size, such that segments must be stored on disk. In addition, an index is required to map optimistic fingerprints to the segments location on disk (or in a memory cache).

Our first approach to map optimistic fingerprints to segment data was to use a single large hash table with linear probing stored in memory, and all segments on disk. This naïve approach has two problems. First, the memory size limits the maximum number of segments that can be indexed by a single hash table resident in memory. Second, most segment accesses requires reading segments from disk since all available memory is used for the hash table. Therefore, the index should be split into multiple parts that can be stored in disk, and a large portion of the memory should be used to cache segments.

We propose using multiple small hash tables; each indexed using the first l bits of the fingerprint. Hash table entries are 64 bits, and contains the remaining fingerprint bits, the memory or disk offset of the segment, and the size of the segment. The hash table,

and the segments indexed by it are stored in a *container*. Each container is stored in a separate file on disk, but can also be cached in memory.

Segment accesses have no spatial locality with respect to fingerprint values, since the hashing function generates random fingerprints for segments. Segments can therefore not be efficiently cached if they are distributed to containers based on their fingerprint values. Instead we exploit the observation that segments written to the cache at the same time tend to be read together. Therefore, all new segments are written to the same container by inserting the fingerprint to the hash table and appending the segment to the end of the segment buffer. In case of a hash table collision the segment is written to the next container in memory. This clustering of segments allows read-ahead of segments from disk. The disadvantage of this approach is that a linear search is required to find the container containing a specific segment. Therefore, we propose multiple optimizations to reduce the number of containers on disk that has to be checked.

Segments accesses have temporal locality, so we cache recently accessed containers a in memory. When a container is accessed, the entire hash table is always read to memory, but the segment buffer is divided into several chunks, which are read on-demand from disk (similar to demand paging [127]). Writes are buffered such that modified segment chunks are only written to disk when the memory is full. To evict segment chunks or containers, we use a least recently used algorithm.

To further reduce disk accesses we use a Bloom filter [37]. A Bloom filter is a space efficient probabilistic data structure that we use to test whether an optimistic fingerprint is a member of the set of optimistic fingerprints stored in the segment cache. The test may return a false positive; hence an optimistic fingerprint in the Bloom filter may not be in the segment cache thus requiring all hash tables to be checked. But false positives are not possible. Therefore in case of a miss, it is not necessary to check the containers before requesting a segment from the sender, or writing a segment to the cache. In the Bloom filter can also be used to overlap network transmission time with disk read time.

4.2 Summary of papers

This section summarizes the paper in section 7.5, and the unpublished papers in sections 8.4 and 8.5.

4.2.1 Remote visualization

The paper in section 7.5 demonstrates that multi-dimensional content based anchoring can improve the performance of remote visualization. We have implemented and conducted an initial evaluation of the Varg prototype system. The goal of the evaluation was to answer the following questions:

1. Are screen update region sizes, and hence the bandwidth requirements, larger for genomic applications than for the Office applications normally used in remote collaboration?
2. What is the Varg compression ratio and time for network data sent for remote visualization of genomic applications?
3. Is the reduction in communication time larger when using Varg than when using the local compression algorithms typically used by remote visualization systems?

4.2.1.1 Methodology

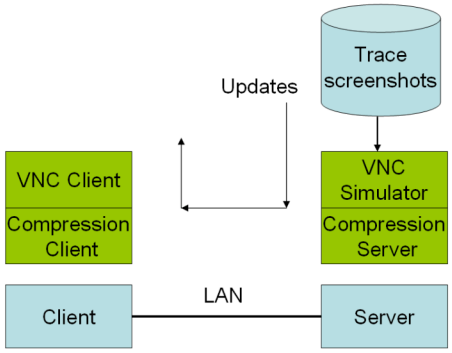


Figure 24: Experimental testbed used to evaluate the compression ratio and compression time of the Varg system.

In order to answer the above questions the Varg system was implemented and experimented with. A trace-driven approach was used (Figure 24). Traces were collected for different Office, and visualization intensive genomics applications (Java Treeview [193], TMeV [192], and GeneVaND [98]). The WAN latencies and bandwidths emulated during trace playback were based on measurements between nodes at different sites in USA and Norway (Table 7).

Network	Bandwidth (MB/sec)	Latency (msec)
Gigabit Ethernet	80.00	0.2
100 Mbps Ethernet	8.00	0.2
Princeton – Boston	2.13	11
Princeton – San Diego	0.38	72
Princeton (USA)– Tromsø (Norway)	0.20	120

Table 7: TCP/IP throughput and round-trip latency for different networks measured using Iperf [2].

4.2.1.2 Experiment results

The updated screen regions are larger for a genomic application than for two office applications (Figure 25). In addition, screen updates are more frequent. Combined these increase the bandwidth required.

For the genomic applications, transfer time is larger than the latency for about 50% of the updates. These updates are larger than 80x80 pixels, which we found to be segment size for which transfer time is larger than latency for all of the WANs in Table 7.

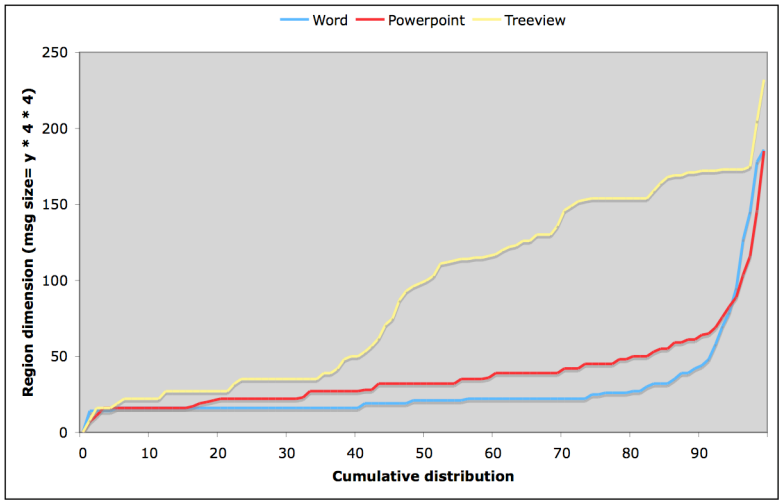


Figure 25: The size of updated screen regions is much larger for the Java Treeview genomic application, than for Office applications.

	Differencing	2D pixel segment compression	zlib	Total compression
TreeView	1.89	5.74	19.98	216.76
TreeView-Cube	2.87	4.05	24.88	289.19
TMeV	1.52	2.46	7.90	29.54
GeneVaND	3.15	2.72	10.85	92.96

Table 8: Compression ratio for four genomic data analysis applications.

The total compression ratios by our method are 217, 289, 30 and 93 for the four genomic application traces respectively (Table 8). These high compression ratios are due to the combination of the three compression methods used: segment differencing, 2D pixel segment compression, and zlib local compression. Zlib contributes the most in all cases, but zlib alone is not enough to achieve high compression ratios. The 2D pixel segment compression using fingerprinting contributes fairly significantly to the compression ratio ranging from 2.5 to 5.7. Without the differencing phase the ratio would be higher, since the differencing phase has already removed a large amount of redundant segments.

The total compression time ranges from 16 ms to 91 ms (Table 9). The most significant contributor is zlib, which consumes more than 10ms in all cases. But 2D pixel segment compression reduces the data volume to be compressed and hence the time spent in this stage. The second most significant contributor is anchoring, but it is below 8ms even for the display wall case. Although SHA-1 calculation contributes up to 8ms in the worst case, the 2-phase fingerprinting optimization allows computation to be overlapped with network communication.

	Differencing	2D pixel segment compression	Zlib	Diff. + Segment. + zlib	SHA-1
TreeView	0.9 ms	3.8 ms	11.1 ms	15.8 ms	3.5 ms
TreeView-Cube	2 ms	7.9 ms	30.2 ms	40.1 ms	7 ms
TMeV	1.3 ms	6.6 ms	83.4 ms	91.3 ms	7.7 ms
GeneVaND	1 ms	2.7 ms	10.1 ms	13.8 ms	1.5 ms

Table 9: Average compression time per screen update. The total compression time depends on the application window size, and how well the differencing and 2D pixel segment compression modules compress the data before zlib is run.

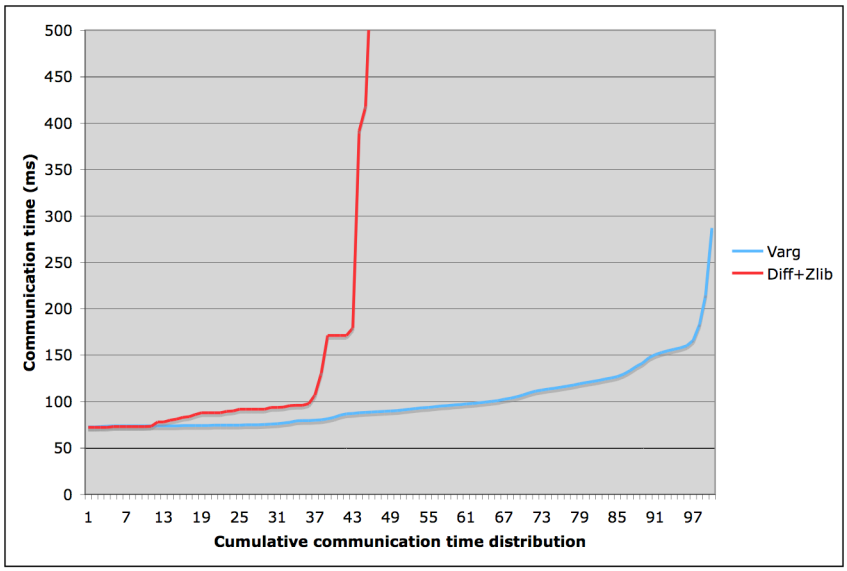


Figure 26: Cumulative communication time distribution for Treeview screen updates sent over the Princeton-Boston WAN.

The transmission time is significantly better for Varg than for the commonly used combination of region differencing and zlib (Figure 26). Without compression the communication overhead for the Princeton—Boston network is several seconds for the largest updates. With zlib the communication overhead is more than 300ms for about 50% of the messages. The communication overhead with Varg is less than 100ms for over 90% of the messages. Even for the cross-Atlantic Princeton—Tromsø network 80% of the updates have a communication overhead less than 200ms, of which the latency contributes to 112 ms. The communication time with Varg is also low for the other traces except for TMeV, where 2-D pixel segmentation did not work well (due to the movement estimation parameters not being tuned properly).

4.2.2 Two-level fingerprinting

The main challenges in implementing the compression sub-system is choosing appropriate fingerprint sizes for the 2-level fingerprint algorithm, choosing the number of optimistic fingerprints covered by conservative fingerprints, and implementing an efficient caching mechanism. These challenges are addressed in the paper in section 8.4. The following questions are answered.

1. For which segment sizes is compression ratio limited by the fingerprint size?
2. Where is the crossing point for when the number of bytes sent due to collisions is larger than the fingerprint bytes?
3. How many segments per conservative fingerprint give the best compression ratio?
4. How many additional bytes are necessary for encoding the two-level fingerprinting protocol messages?
5. Does a multi gigabyte cache improve compression ratio?

4.2.2.1 Methodology

To find the best parameters for the two-level fingerprinting protocol giving the best compression ratio, we model the number of segment bytes sent, the number of fingerprint bytes sent, and the number of collision bytes sent, using the formula in Equation 1, and the default workload parameters in Table 10. We compare the achieved ratio to a fingerprinting protocol using 20 byte fingerprints.

To evaluate the benefits of a large segment cache we use the traces collected for three genomic applications, as described in section 4.2.1.1.

$$compression\_ratio = \frac{S}{(S-R) + (\frac{k}{8} + \frac{l}{8p})\frac{S}{s} + ps \sum_{i=1}^{i=(\frac{S-R}{s})} \frac{i}{2^k}}$$

Equation 1 models compression ratio achieved using two-level fingerprinting. *S* is the data set size, *R* is the redundancy found, *k* is the number of optimistic fingerprint bits, *l* is the number of conservative fingerprint bits, *p* is the number of segments per conservative fingerprints, and *s* is the segment size. *S/s* is used to estimate the number of segments in the data set. The sum estimates the probability of a segment inserted to the cache having the same optimistic fingerprint as an existing segment. We assume each collision causes the entire group of segments to be resent.

Parameter	Value	Parameter explanation
S	100GB	Data set size
R	75GB (75%)	Data redundancy found
K	40 bits	Optimistic fingerprint size
L	160 bits	Conservative fingerprint size
P	20	Segments per conservative fingerprint
S	32 bytes	Segment size

Table 10: Default parameters used to model two-level fingerprint compression ratio.

The server is implemented using a multi-threaded event based model. The protocol handling is divided into several stages. The stages are connected using queues, used to store segment objects to be processed by the next stage. In addition some stages either

read from, or write to a socket. To support multicast, some stages produce output destined to several stages.

4.2.2.2 Experiment results

Protocol parameters

The fingerprint size significantly limits compression ratio for segments less than 1 Kbytes when the redundancy detection in the data is 75% (Figure 27). With lower redundancy detection even smaller segments are limited by the fingerprint size.

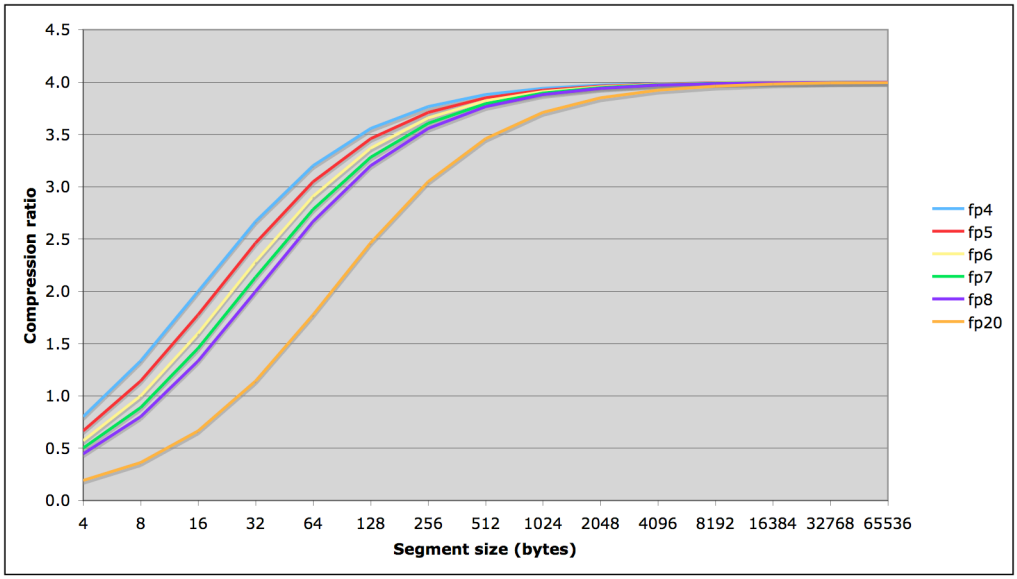


Figure 27: Compression ratio for different fingerprint and segment sizes. Data redundancy is 75% and collision bytes are ignored.

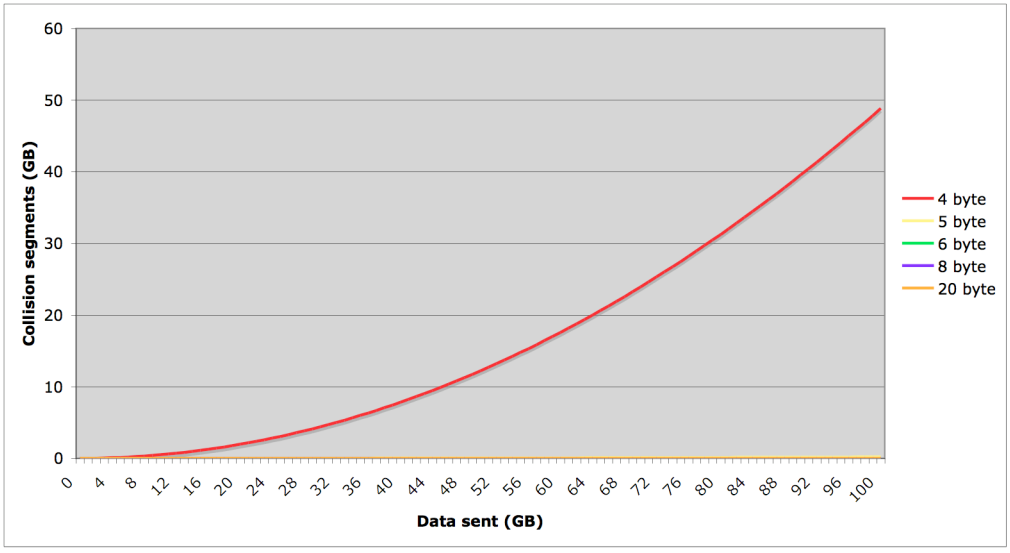


Figure 28: Miss penalty bytes sent for different optimistic fingerprint sizes. (for all but the 4 byte fingerprints the miss penalty is insignificant).

Choosing the conservative fingerprint size is relatively straightforward; it should be large enough to guarantee a collision rate smaller than the hardware error rate. Since 2^{160} is considered sufficient for data sets up to an exabyte in size [172], we use 160 bit SHA-1 hash values as conservative fingerprints.

The optimistic-fingerprint size is more challenging to choose because it affects two competing trends. Reducing the optimistic-fingerprint size will increase the maximum achievable compression ratio, but simultaneously increase the number of cache collisions that require entire segments to be resent. So we want to choose an optimistic-fingerprint size that is near the inflection point of the competing trends and that works across the many data types being transmitted.

If a 4 byte optimistic fingerprint size is chosen, then 50 GB of segment data will be sent due to collisions when transferring a 100 GB data set (Figure 28). Increasing the optimistic fingerprint size to 5 bytes, reduces the total number of bytes sent since the data sent due to collisions is reduced to 0.2 GB, while the increase in fingerprint bytes is only 6.1 GB. If the data set size is less than about 35 GB, does 4 byte fingerprints give the best compression ratio.

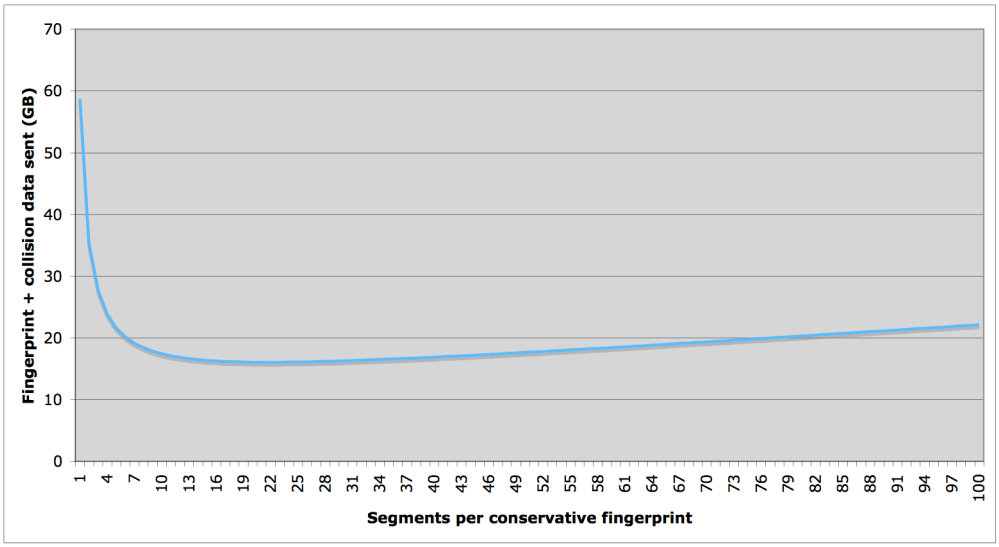


Figure 29: Fingerprint and collisions bytes sent for different segments per conservative fingerprint ratios.

The number of segments covered by a conservative fingerprints should be chosen such that the fingerprint bytes sent remains low, while keeping the bytes sent due to collisions low. With the default parameters in Table 10 the minimum number of bytes sent are for 22 segments per conservative fingerprint (Figure 29). Typically a ratio of 20—25 gives a good compression ratio, even if the segment size, redundancy ratio, or data set size is changed.

In conclusion, with 5 byte optimistic fingerprints and 20 segments per conservative fingerprint, the compression ratio is better than 4 byte and 20 byte fingerprints for most redundancy levels (Figure 30). Only when more than 95% redundancy is detected is the ratio better for 4 byte fingerprints.

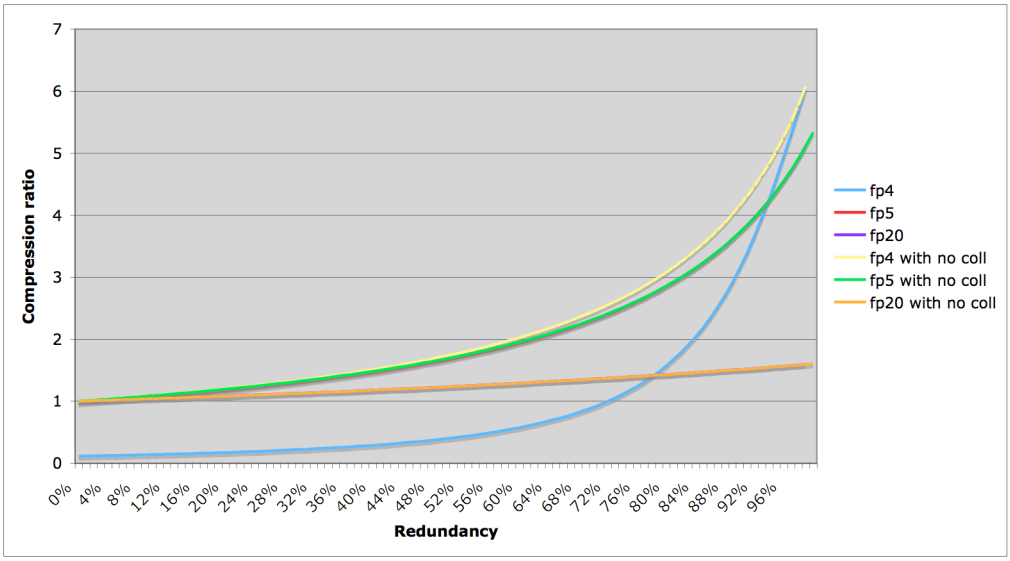


Figure 30: Compression ratio for different redundancy levels when using 4 byte, 5 byte, and 20 byte fingerprints. The 5 byte fingerprint compression ratios with and without collisions are almost identical. For 20 byte fingerprints these are identical since there are no collisions.

Protocol messages

The message headers used in the two-level fingerprinting protocol also limit compression ratio, and the messages have therefore been designed to use as few bytes as possible (Table 11). The first byte is used to identify the message type and to store the meta-data size in *optimistic fingerprint* messages.

Message type	Size (bytes)	Comment
Optimistic fingerprint	6 + M	M is implicitly set by the message type
Segment request	5	A 4 byte sequence number identifies the segment
Segment	7 + S	Includes the segments sequence number and size (2 bytes)
Conservative fingerprint	21	
Conservative fingerprint ACK	1	No sequence number since the ACKs are sent in the same order as conservative fingerprints
No-fingerprint segment	3 + S	Includes the segment size (2 bytes)
Multiplexing message	3	2 byte are used to identify the segmentation component that should receive the next batch of segments

Table 11: Two-level fingerprint messages. *M* is meta data size, and *S* is segment data size. Optimistic and conservative fingerprint sizes are respectively 5 and 20 bytes.

The *conservative fingerprint* message is always sent immediately after the last optimistic fingerprint message in a group. It is therefore not necessary to add any information to the message about which segments are covered, and thus the message only contains the message type and conservative fingerprint. *Conservative fingerprint*

ACK messages are always sent in the same order as the conservative fingerprints were received, and therefore it is not necessary to add a sequence number.

Advantages of a multi gigabyte segment cache

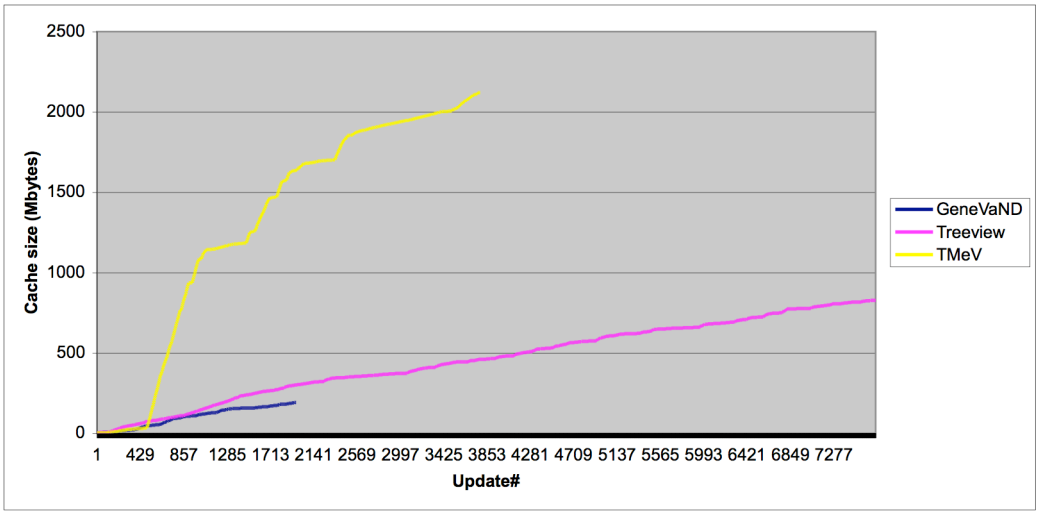


Figure 31: Cache size increase for remote visualization of three genomic applications.

The number of segments cached, and hence the size of the segment cache, depends on the redundancy detected. Redundancy detection stabilizes after a while, and can be up to 80%. But since the hit ratio never reaches 100% the cache size has a steady growth (Figure 31). Even for the short 10—15 minute traces the segment cache becomes too large to be stored in memory.

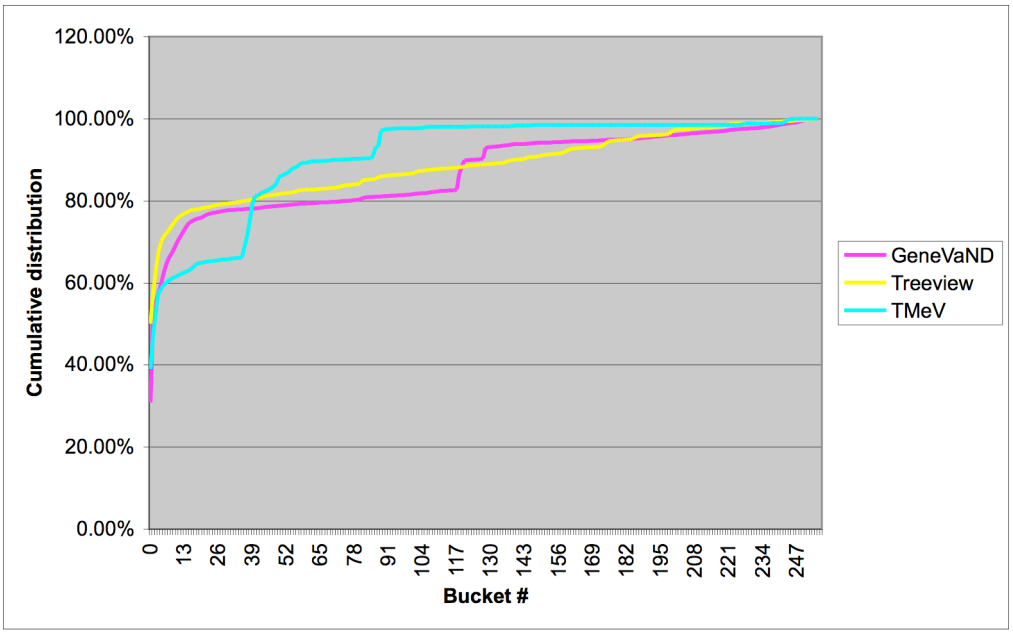


Figure 32: Cache hit entry age. Most cache hits are for recently inserted segments, but when execution time increases the number of hits for older entries increase. Note that the bucket size is 6021 for Treeview and 2445 for the other two.

A larger cache improves redundancy detection, as shown in Figure 32 where the age of accessed cache segments is plotted. Age is defined as the number of segments added to the cache since the given segment was added. Most hits are for recently added segments, but as the visualization session proceeds more hits are for older segments. Therefore we believe compression ratio will improve with a large cache for longer traces.

4.2.3 Multi-dimensional segmentation

The paper in section 8.5 describes and evaluates different algorithms for content based segmentation of 2-D data sets. The following questions about advantages of multi-dimensional segmentation, and the tuning of 2-D content-based algorithms are answered:

1. Does 2-D content-based segmentation improve the compression ratio compared to static 2-D segmentation?
2. Does 2-D application specific segmentation improve the compression ratio and time compared to general purpose 2-D segmentation
3. What region size should be used to get the best redundancy detection?
4. Does 2-D segmentation scale with respect to data set size?

4.2.3.1 Methodology

To answer the above questions we have used the traces collected for the Varg system (these were described in 4.2.1).

We have experimented with the following 2-D segmentation methods:

- The static 2-D segmentation algorithm used by VNC Hextile [183], and MPEG [88].
- Static 2-D segmentation combined with the popular zlib [9] local compression algorithm. This combination is often used by remote desktop systems.
- Our 2-D segmentation algorithm used in the Varg system (described above), which does static segmentation into columns and then content-based segmentation within columns. The algorithm has two optimizations for segmenting screenshot data:
    - The algorithm assumes that content movement is vertical. Redundancy detection can be improved if horizontal movement is also detected. To determine whether content has moved predominantly vertically or horizontally, we do movement estimation by comparing Rabin fingerprints for a subset of rows and columns on the screen, that are selected based on Manber's method [145]. If movement is predominantly horizontal we transpose the 2-D array containing the pixel data before running the algorithm.
    - Some screen regions consists of identical pixels. The row fingerprints calculated for such regions are identical, and hence either all or none will be selected. To improve compression ratio we always create a single large segment since it can efficiently be compressed using a local compression method (such as zlib).

- A novel 2-D content-based segmentation algorithm, that is similar to the Varg algorithm but uses content-based segmentation in both dimensions. First the 2-D array is statically divided into large *m x m* pixel tiles. Each tile is then divided into horizontal columns by selecting anchor-columns based on fingerprints calculated for each column. Finally, the horizontal strips are divided into regions by selecting anchor-rows based on fingerprints calculated for each row.
- Segmentation based on probabilistic 2D- pattern matching as suggested by Karp and Rabin [122]. A short fingerprint is calculated for all *m x m* regions including all overlaps. Then regions are selected based on the fingerprint value using Manber's method. The algorithm divides the 2-D data structure into fixed sized segments that can overlap.

4.2.3.2 Experiment results

Compression method	GeneVaND	TreeView	TIGR MeV
Hextile + Zlib	13.6	19.2	14.8
Static segmentation	15.9	24.3	16.1
Probabilistic 2-D segmentation	9.5		
Static + 2-D content based	18.3		
Varg (Static + 1-D content based)	24.0	90.9	29.7
Varg without movement estimation	23.8	89.6	17.3
Varg without similar region detection	22.9	82.9	17.1

Table 12: Compression ratio relative to Hextile for different segmentation methods for 2-D screenshot data.

The achieved compression ratios using the different methods are summarized in Table 12. This section details the results. First, we compare the Varg compression method against other widely used methods, and then find the parameters giving the best compression. Finally, the scalability of the Varg method is demonstrated.

Content-based 2-D segmentation

Compared to static segmentation, content-based segmentation improves the compression ratio up to 3.0 (Table 12). The improvement is due to content-based segmentation achieving higher redundancy detection when using larger segment that compress better with local compression algorithms, hence improving the total compression ratio (as discussed in section 4.2.1).

However, content-based segmentation in both dimensions does not improve redundancy detection compared to static segmentation. The problem is that if one of the pixels in an anchor-column changes, the fingerprint for the column also changes. The changed fingerprint may not be selected as an anchor-column. When the column boundaries change, all segment boundaries also change.

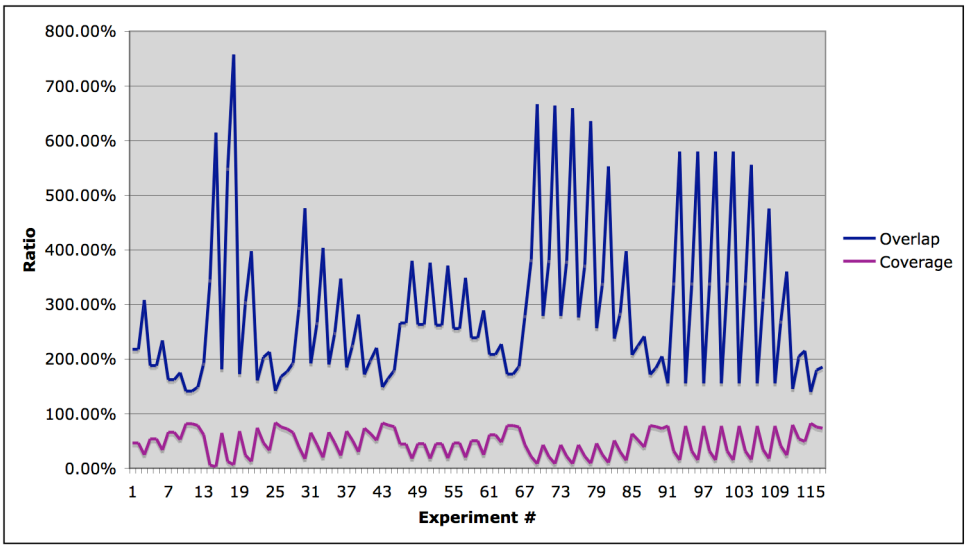


Figure 33: Probabilistic 2D pattern algorithm tuned to reduce the pixels in overlapping segments, or to reduce the number of pixels not covered by segments. Ideally both overlap and coverage should be 100%.

Probabilistic 2-D segmentation also does not provide better compression ratio than static segmentation, since pixels are either not covered or are in overlapping regions. Tuning the algorithm parameters either reduces both coverage and overlap, or increases both coverage and overlap (Figure 33). In addition, calculating Rabin fingerprints for all 2-D regions is computationally costly since a sliding window Rabin implementation cannot be used.

Segment size

With static segmentation the best total compression ratio when segments are not compressed with zlib is for 4x4 pixel regions (48 bytes), and 32x32 pixels (3072 bytes) if zlib is used (Figure 34). Similarly for Varg content-based segmentation, smaller segments improved redundancy detection, while larger segments improves zlib ratio and hence the total compression ratio.

The small segment sizes giving the best fingerprint redundancy detection are about 48—192 bytes. In the previous section we found that for such small segments compression ratio is limited by the fingerprint size, and that two-level fingerprinting will improve the compression ratio.

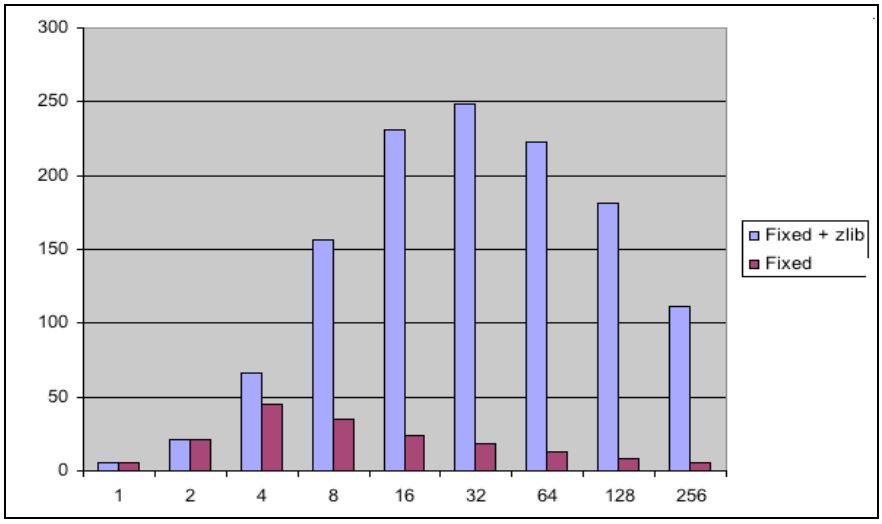


Figure 34: Compression ratio with fingerprinting and static segmentation.

There are four parameters in the Varg 2-D segmentation method that can be changed to adjust the average region size. Our results for the genomic application traces shows that these should be set as follows to achieve the best compression ratio:

- The static column width should be small. On our experiment platform 16 pixels worked well since horizontal scrolling often moved content 16 pixels at a time. But a width of 4 pixels gives the best redundancy detection. Small static columns increase horizontal redundancy detection, since multiple pixels are typically scrolled at a time. In addition redundancy detection may decrease if a column is wide enough to include content both inside and outside a scroll-pane.
- The number of bits used for fingerprint selection, depends on the visualization. For the Treeview and TMeV trace the best ratio is when every 8^{th} row is selected on the average. For GeneVaND selecting on the average every 32^{nd} row gives the best ratio.
- Minimum region height should be about 8-16 rows if zlib is used, and 4 pixels if not. A smaller minimum decreases the total compression ratio due to reduced zlib compression ratio. A larger minimum also decreases redundancy detection since a change to an anchor row may cause subsequent anchor rows not to be selected since they are within the minimum height.
- Specifying a maximum region height does not improve compression ratio, but may be necessary due to the fingerprint protocol messages having restrictions on the number of bits that can be used to store the segment size.

Application specific optimizations

The two pixel data specific optimizations used in the Varg segmentation algorithm improves compression ratio. Similar detection of column rows with identical content, improves compression ratio by 1—8% since zlib compression ratio improves. Movement estimation improves the compression ratio for TMeV with 72%, since about 30% of the updates have predominantly horizontal movement. The compression ratio improvement is smaller for the other traces, since few screen updates had horizontal movement.

Scalability

The Varg segmentation method scales with screen size. With a larger screen the same algorithm parameters give the best compression ratio, and the distribution of segment sizes do not change. However, the total compression ratio improves, due to improved fingerprinting and zlib compression ratio. It is therefore not necessary to tune the segmentation method for different screen sizes.

4.3 Discussion

In this section the results presented in the previous section are discussed.

4.3.1 Remote visualization

Our work was motivated by the need for interactive remote visualization tools to be used for collaborative analysis in bioinformatics and biology. Similar needs for collaborative visualization tools are required in other fields, such as meteorology, geosciences, and medicine. Provided that the interaction requires scrolling and zooming large 2-D visualizations, we believe using Varg will give similar improvements in compression ratio and time as for the genomics applications we have experimented with.

The redundancy detection algorithm provides a high compression ratio for 2-D visualizations. But, some scientific analysis tools provide 3-D visualizations. We do not expect content-based compression to work as well for these, since the interaction often involve rotation and zooming of objects such that the same set of pixels are rarely displayed on the screen. 3-D visualizations are often programmed using libraries that bypass the framebuffer typically used by remote visualization systems to detect changes to the screen. To solve this problem the remote visualization server can intercept the library function calls and forward these to the client using a protocol such as Chromium [104]. The interception and the protocol implementation may provide an opportunity for applying compression.

The 2-phase fingerprinting protocol uses eventual consistency to reduce user perceived end-to-end latency. VNC, and most other remote desktop systems, also uses eventual redundancy. But, the 2-phase fingerprinting protocol may cause the replicated screen content to become inconsistent. This can happen if an optimistic fingerprint collision is detected, and the segment with the corrected pixels is sent after another update for the same pixels. The Varg prototype is sequential; hence the problem is avoided since updates cannot be sent between the optimistic fingerprint and the segment data sent due to a collision.

Even with the very best compression ratio, the end-to-end update latency will be limited by the network latency. For WANs the latency can be large, often tens of milliseconds, and hence be noticeable to users. The lower bound for the network latency is limited by the speed of light. But it is important not to add a high transfer time on top of an already large latency. In addition to network latency, we found remote visualization performance to be limited by the VNC server implementation.

A remote visualization system could use a dedicated high bandwidth WAN. Also, we can expect WAN bandwidth to improve. Either way, we believe our compression method is still useful for two reasons. First, the requirements for the quality of visualization, and hence the network bandwidth requirements, will also increase. Second, with higher network bandwidth the compression time must be smaller in

order to reduce transfer time. Our 2-D pixel compression algorithm can provide the necessary compression ratio without the use of slow local compression algorithms.

Varg used VNC, since it has open source implementations for Linux, Mac and Windows. But the 2-D pixel compression algorithm is independent of VNC. In order to use another remote visualization system, all that is required is to parse the screen update protocol in order to extract the updated region data.

4.3.2 Two-level fingerprinting

We found that the best optimistic fingerprint size for compressing a 100 GB data set is 5 bytes, due to a reduction in collision bytes compared to 4 byte fingerprints. For a smaller data set, compression ratio improves with smaller fingerprints. The optimistic fingerprint size could be dynamically set at server startup time. But it is not possible to increase the optimistic fingerprint size without flushing the segment cache, or re-computing the optimistic fingerprint for all cached segments. Similarly, the size of the statically allocated Bloom filter and number of segments in the compression pipeline, could also be set at startup time based on the data set size.

The compression component multiplex data received from several segmentation components. An alternative would be to run a Canidae fingerprint component instance for each application. This would require dynamic memory and storage resource management, which complicate the system, and probably neither improves compression ratio nor the throughput of the system.

Most existing and new content based segmentation methods can be implemented to use the segment cache provided by Canidae. But, the Spring and Wetherall [209] method had to be modified since the redundancy detection requires a cache that stores the last N sent bytes. We believe segmentation methods for multi-dimensional data sets require a cache that stores a set of segments as in Canidae. The alternative is to use some multi-dimensional data structure to which data can be incrementally appended, and that efficiently allows comparison of stored data with a multi-dimensional segment by growing the region in all dimensions. We are not aware of any such data structure.

In order to simplify the implementation of the many segmentation components, the two-level fingerprint component provide in-order delivery of segments. For protocols supporting out-of-order message delivery, such as the VNC protocol compressed by Varg, the end-to-end latency can be reduced if the segmentation components on the receiver side can request messages to be delivered out-of-order before the conservative fingerprint is verified.

The two-level fingerprint components can also be modified to reduce the latency of segments sent through it. Currently, three messages may be sent over the WAN for a segment (optimistic fingerprint, segment request, and the segment message). For many segments the latency of sending segments can be reduced to a single WAN latency, by maintaining a Bloom filter with previously sent optimistic fingerprints. The sender checks the Bloom filter after sending an optimistic fingerprint, and immediately sends the segment if the optimistic fingerprint is not in the cache.

Canidae throughput may also be improved if the receiver can query multiple servers for segments not in its cache. This way, the segments may be received from a server to which the bandwidth is higher, or the latency is lower than to the sender.

Canidae is designed to run on a dedicated machine such that it can utilize all memory and disk on that machine. The same approach is used by several commercial WAN accelerators [38, 62, 79, 120, 186]. We believe it is realistic to allocate one machine in an organization for this purpose. The segmentation components can also be run on dedicated machines, or they can be integrated with the application if the communication latency, computational, or memory overhead of maintaining a separate multi-dimensional data structure is too large. The compression pipeline is implemented using multi-threading in order to utilize the many cores we expect future processors to have. But, currently the implementation only scales to 2 CPUs, due to cost of synchronizing the different parts in the compression pipeline.

We have not addressed security. If the application encrypts data to be sent through Canidae we should not to be able to find much redundancy. The problem is further complicated since applications are likely to use a wide variety of security (and privacy) mechanisms and policies. Related work [220, 221] has proposed to use convergent encryption [75] such that the fingerprints of segments are used to compress the data.

Several important questions remain unanswered. The advantage of two-level fingerprinting should be validated, by answering the following questions:

1. Does two-level fingerprinting improve the compression ratio achieved when using previous 1-D content-based segmentation methods?
2. Does redundancy detection improve with smaller segments?

The benefit of a hundred GB cache needs to be validated using a larger data set than the genomic visualization data sets.

In addition the system should be properly evaluated by measuring:

3. The throughput Canidae can support, and which cache parameters gives the best performance?
4. The scalability of the compression pipeline and segmentation methods on CMP and SMT systems?

To answer the first two questions the compression ratio of the implemented 1-D segmentation algorithms should be measured when using different segment sizes, with one level fingerprinting, and with two-level fingerprinting. A data set consisting of uncompressed flat files can be used for the experiments. The last two questions can be answered by measuring the throughput of the system on machines with processors supporting CMT and SMT.

4.3.3 Multi-dimensional segmentation

Implementing a general-purpose 1-D segmentation component is relatively straightforward. But, redundancy detection usually works better when protocol headers or file meta data is removed from the data to be segmented [145, 209]. This requires parsing the application messages. Typically only a few message types are used to send the data that need to be compressed, and the remaining message types can simply be forwarded uncompressed and unparsed. Implementing such limited parsing for a protocol is easy if an open source implementation of a server and client exists.

We have applied our segmentation on 2-D arrays of pixels. Data movement in such data sets is predictable, and neighboring elements tend to have similar values. For

other 2-D data sets, data movement may not be so easy to predict, and neighboring elements may be different. Compression may not work as well for these data sets. But, additional experiments are required to investigate these limitations.

Application specific optimizations improved the compression ratio for screenshot data. The need for such optimization can easily be detected by analyzing the segments produced by an algorithm. For the screenshot data we found it useful to play back the visualization with the segment boundaries shown.

In our experience, achieving high compression ratio, while keeping the computation cost low, is more difficult for 2-D segmentation methods than for the 1-D methods. Our best method for 2-D segmentation combined movement estimation, static segmentation, and content-based segmentation. Developing a similar algorithm for 3-D datasets, would require movement estimation in 3-D and keeping two of the three dimensions static. We do not expect such a method to achieve high redundancy detection, and alternative approaches are probably required.

To improve redundancy detection for 2-D data, a segment size could be selected at run-time based on the data type. But we are not aware of any methods that can be used to predict the segment size to use. In contrast to the optimistic fingerprint size, the segment size can be changed without flushing the segment cache.

Additional experiments are required to answer the following questions:

1. Does 2-D based segmentation methods improve the compression ratio and time compared to 1-D segmentation methods for 2-D data?
2. What is the redundancy detection of the 2-D based segmentation when used on scientific 2-D data?

To answer these question it is necessary to experiment with existing 1-D segmentation methods [156, 209], and the 2-D segmentation methods described above. The data sets could be from different scientific domains, such as geometric data, scientific simulation output, and satellite images.

4.4 Related work

The *Related Work* in the paper in section 7.5 discusses compression algorithms used in thin-client systems, MPEG compression and the Access Grid. This section describes related work in redundancy detection, two-level fingerprinting, disk cache design, commercial systems, remote visualization, and local compression.

4.4.1 Redundancy detection

Redundancy detection methods can be divided into four classes: duplicate detection, static segmentation, content-based segmentation, and delta encoding. Each class is described below. Examples of systems implementing the different methods are shown in Table 13.

Approach	Application domain/ data transferred	Redundancy detection	Median segment size	Fingerprint size
Mogul et. al. [153]	Web cache	Duplicate elimination	3 KB	128 bit
Santos and Wetherall [196]	TCP packet data	Duplicate elimination	536 bytes	128 bit
CAS (Tolia et al. [221])	Distributed file system	Static	4 KB	160 bits
Sapuntzakis et al. [197]	Virtual machine state	Static	4 KB	160 bits
CFS (Dabek et al. [69])	Peer-to-peer content distribution	Static	8 KB	160 bits
Hong et al. [100]	SAN file system	Static	4 KB	160 bits
Venti (Quinlan and Dorward [172])	Backup system	Static	8 KB	160 bits
rsync (Tridgell [224])	File synchronizer	Static + Overlapping static	300 bytes	32+128+ 128 bit (2-level)
Tolia and Satyanarayanan [222]	Database content	Static (row boundaries)		160 bits
Spring and Wetherall [209]	Network data	Spring and Wetherall	128 bytes	64 bits
LBFS (Muthitacharoen et.al [156])	Distributed file system	LBFS	4KB	160 bits
Rhea et al. [180]	Web objects	LBFS	2 KB	128 bit
Pastiche (Cox et al. [64])	Peer-to-peer backup system	LBFS	16 KB	160 bits
Shark (Annapureddy et al. [17])	Peer-to-peer content distribution	LBFS	16 KB	160 bits
Pucha et al. [171]	Multimedia files	LBFS	16 KB	160 bits
Tolia et al. [220]	Storage system for reference data	LBFS	4 KB and 128 byte	160 bits
Imrak and Suel [114]	Network data	LBFS (multi-resolution)	256 byte – 2 KB	64 bit + 128 bit (2-level)
Denehy and Hsu [72]	Storage system	Static, overlapping, LBFS	4 KB	160 bits
Bobbarjung et al. [39]	Storage system	LBFS + differencing	4 KB + 128-256 byte	160 bits + 160 bits
Housel and Lindquist [101]	Web cache	Delta encoding	no segments	[1]
Douglis and Iyengar [76]	HTML and web objects	Manber + delta encoding	No segments	[1,2]

Table 13: Overview of global compression systems.

[1] Delta encoding size depends on the difference between the compared files.
[2] The Manber feature set has a constant size that is independent of file size.

4.4.1.1 Duplicate elimination

Duplicate suppression identifies and eliminates transfers of exact duplicates. Mogul [153] use it at the document level for web data, such that the cache is indexed using a fingerprint of the document content instead of an URL. Santos and Wetherall [196] use it at the packet level. The disadvantage of duplicate elimination is that redundancy detection is typically low [169].

4.4.1.2 Static segmentation

A static segmentation algorithm divides a dataset into fixed size, non-overlapping, contiguous segments.

rsync (Tridgell [224]) is a Unix tool for copying a file directory over the network into an existing directory tree with similar files. Similarity between files is used to reduce bandwidth usage. The redundancy detection algorithm compares the content of files with identical filename in the two directory trees. First the receiver segments its version of the file using static segmentation, computes fingerprints, and sends these to the sender. The sender computes fingerprints for all overlapping fixed size segments in its file. If the fingerprint for a segment matches one of the received fingerprints, a description of the segments location is sent instead of the segment data.

Static segmentation can also be used to segment structured data. Tolia and Satyanarayanan [222] segment database content by exploiting row boundaries of relational database results. Their results shows that redundancy detection improved compared to content-based segmentation using the LBFS approach (described below). The improvement is due to the large content-based segments crossing row boundaries. Using the much smaller segments provided by Canidae may avoid this problem.

Static segmentation of large files has been used by peer-to-peer content distribution system, such as CFS [69], to improve data transfer speed and efficiency by supporting simultaneous download of segments from multiple sources. Sapuntzakis et al. [197] use static segmentation to reduce the bandwidth usage when sending memory content over a network.

Static segmentation algorithms, that for example take data structure into account, can be implemented as Canidae segmentation modules. These can take advantage of the two-level fingerprinting protocol and hence use smaller segments than has been used by existing systems.

4.4.1.3 Content-based segmentation

Content-based segmentation methods divide data to be transferred (or stored) into variable sized segments. First fingerprints (hash values) are computed for fixed size substrings in a 1-D bytestream, including overlaps. Then a deterministic random sample of these fingerprints is selected. The selected fingerprints are then either used as starting points for finding redundant segments in the data, or as anchorpoints for dividing the data into segments.

Many methods use Rabin fingerprints [175], since these can be efficiently computed using a sliding window implementation. A Rabin fingerprint is the polynomial representation of some data modulo an irreproducible polynomial. Broder [50] described how Rabin fingerprints can be computed 32 bits at a time using precomputed tables for the irreproducible polynomial. The sliding window implementation allows computing the fingerprint for the window content in terms of

the previous fingerprint. First the oldest byte in a window is subtracted using precomputed tables, then the window is advanced with 1 byte, and finally the fingerprint is updated by adding the terms of the new byte.

To select a random sample of Rabin fingerprints most segmentation methods use Manber's [145] approach. To find similar files, Manber selected fingerprints where the least *k* significant bits are zero, and then compared these. Since the fingerprints are selected based on their content rather than location, the same set of fingerprints will be selected even if the content has moved. With a uniform distribution one of every 2^k fingerprints will be selected.

Spring and Wetherall [209] adapted Manber's approach to select fingerprints to be used as starting points for finding redundant segments in a 1-D bytestream. Both the sender and receiver store previously sent data in a cache (the cache is a large buffer where data is appended). To segment a 1-D packet to be transferred, Rabin fingerprints are calculated, selected, and checked against fingerprints calculated for the data stored in the cache. For each match, the bytes covered by the fingerprint window have the same content in the cache and in the packet to be sent. The segment can then be expanded, to cover the bytes before and after the fingerprint bytes, by matching bytes in the packet and in the cache. Finally, the fingerprint and a description of the covered region are sent to the receiver.

In the Low Bandwidth File System (LBFS) [156] Manber's approach is also used to select a fraction of Rabin fingerprints. But instead of using the selected fingerprints as a starting point for growing a segment, these are used as anchorpoints in a 1-D bytestream. The anchorpoints divide the bytestream into segments, such that segment consists of all bytes in the Rabin fingerprint window, and all following bytes until the beginning of the next anchorpoint.

Many other fingerprint based systems have either used the Spring and Wetherall or the LBFS approach for segmenting 1-D data (some examples are give in Table 13). But these work well only with 1D data types, such as web content, documents, email and binaries. Our redundancy detection method is aware of the data structure, and works well with 2-D screen buffers. In addition the two-level fingerprinting protocol allows using smaller segments than are typically used by previous systems.

4.4.1.4 Delta encoding

Document updates often only make small modifications. These changes can be encoded using delta encoding (as done by the Unix diff utility). Delta encoding has been used for web data compression [101, 154], by only sending the difference between two version of a cached document. Similarly the CVS version management software [30] saves bandwidth by only sending patches describing required changes in order to update a set of files. However, these approaches are only able to detect differences at a single document level.

To apply delta encoding to a large set of files Douglis and Iyengar [76] first chose a representative base set of files by using Manber's technique for detecting similar files [145], and then send delta encoded differences between the files to send and the base files.

Delta encoding works well when the data is structured into files and the receiver already has a previous version of the data to be transferred. However, choosing a

representative base is more difficult for multi-dimensional data sets, especially if the data is streamed.

4.4.1.5 Evaluations

Policroniades and Pratt [169] measured the amount of redundancy detected when using whole-file duplicate elimination, static segmentation, and LBFS content-based segmentation on 1-D file sets. The data set sizes ranged from a few gigabytes to 100 GB. Overall, content-based segmentation detected most redundancy. But static segmentation also provided usable levels of redundancy; for some data sets content-based segmentation was only slightly better. Both were significantly better than whole-file duplicate elimination. Combining content-based segmentation with zlib compression of the segments can improve compression ratio. Content-based redundancy detection was best for code files (45—99%), but worse for more diverse files (20—25%). The index used to store fingerprint to segment location mapping was identified as a significant limitation to the achieved compression size.

You and Karamanolis [100] compared the compression ratio achieved using LBFS content-based segmentation versus Douglis and Iyengar's delta encoding. The data set experimented with consisted mostly of uncompressed 1-D files, but there were also some 2-D images in the TIFF format. Compression ratios ranged from none (content based on TIFF images) to 100x (Unix log data). For content-based segmentation with 128-bit fingerprints, the best redundancy detection was when using 128-256 byte segments. Content-based segmentation was best for large volumes of similar data. Delta encoding is more computation intensive, but better for less similar data due to the lower storage overhead of the meta-data (the feature set used to find similar files has a constant size, while the number of fingerprints depend on segment and file size). Both outperform gzip, especially when zlib is used to compress segments.

Pucha et al. [171] use the LBFS approach for content-based compression of compressed multimedia files in order to improve multi-source download performance for cases where the receiver is not able to utilize all its available network bandwidth. About 99% redundancy was detected for MP3 files, and about 15% for movies. These files contained identical content but differed in for example header data for MP3 files or the subtitle language for movies.

These previous evaluations have four limitations. First, only the LBFS approach was used for content-based segmentation. Second, large fingerprints were used to encode segments. Third, the average segment size was large. Content-based compression using Canidae can use smaller segments, and apply multi-dimensional segmentation methods. A new evaluation is therefore needed for multi-dimensional data.

4.4.2 Two-level fingerprinting

Most fingerprint based network data and storage systems use a single fingerprint to encode a segment. But some previous work has proposed using multiple levels of fingerprints as described below.

Rsync [224] uses three sets of fingerprints. However, in contrast to our two-level fingerprinting protocol the goal is not to efficiently encode small segments, but to reduce compression time. The three sets of fingerprints are used as follows. First, for each static segment, the receiver computes a sliding window Adler 32 bit fingerprint that is fast to compute, and a 128 bit MD4 hash [187] that has low probability of collision. These are then sent to the sender. The sender uses the sliding window

fingerprints when searching for a redundant segment, and only computes the low-collision probability fingerprint when the fast fingerprints match. The third set of fingerprints consists of MD4 hashes computed for each file, and are used to verify that the file assembled by the receiver is identical to the sender's file. In [224] Tridgell observed that since the per file MD4 hash ensures data consistency, the per segment MD4 hashes could be replaced with a smaller fingerprint. The resulting protocol would be similar to our two-level fingerprinting protocol. However, the improvements to compression ratio and transfer time were not evaluated.

Imrak and Suel [114] propose a hierarchical segmentation method, where the LBFS method is run over the same data multiple times with different expected segment sizes. The segments from the various levels are cached together in a multi-resolution cache in memory, where recently sent data is indexed using a fine granularity, and older data is indexed using a coarser granularity. To send a packet of data, the sender uses the cache to find the coarsest segments covering the packet data, by comparing fingerprints for the cached segments with the fingerprints calculated for the data to be sent (using multiple expected segment sizes). When a match has been detected an 8-byte fingerprint is sent, and then for all data in an object a 16-byte fingerprint is sent to ensure consistency. Our two-level fingerprinting protocol differs in three ways. First, our goal is not to improve compression ratio by detecting larger segments, but more efficiently encoding small segments. Second, segmentation is not integrated with the cache. Third, we use a much larger segment cache, and do not rely on information about which segments has previously been sent.

Bobbarjung, Jagannathan and Dubnicki [39] propose *fingerdiff* that combines fingerprinting and differencing to improve duplicate elimination in storage systems. First the LBFS content-based segmentation is used to divide the data into large chunks. If the chunk is updated, then the large chunk is divided into smaller segments, such that only the small chunks must be written to the storage. Fingerdiff allows using small 128-256 byte segments for changed regions, without significantly increasing the storage required for the fingerprint-to-segment index. The compression ratio improved with 13-40% compared to the best content-based approach. Canidae differs from storage systems in that segments are never updated, and will therefore not benefit from fingerdiff.

Tolia et al. [221] propose the fuzzy blocking approach for sending files over a network. It is an extension suggested to their content addressable storage (CAS), where the sender sends a set of fingerprints for statically segmented files, and the receiver requests segments not in its segment cache. With fuzzy blocking error correction codes are used to detect changes to 128 byte blocks within a 4 KB segment. The approach has not, to our knowledge, been implemented nor evaluated.

4.4.3 Segment cache

Few global compression systems have addressed the problem of implementing an index for mapping fingerprints to segments that that is stored on disk, since for most system the number of segments is small enough for the index to fit memory.

In the Venti system [172], the index is divided into fixed size buckets and stored in a separate *pointer* block on disk. A hash function maps fingerprints to index buckets, and then binary search is used to find the fingerprint. To optimize index lookups three optimizations are done. First pointer blocks and data blocks are cached in memory. Second, the index is stripped across multiple files. Third, writes are buffered. The

containers used by Canidae are also cached, and allows to buffer writes. In addition three additional optimizations are used. First, a Bloom filter is used to further reduce the number of times the disk must be accessed. Second, writes do not check the index, since we can tolerate duplicates causing inconsistencies in fingerprint to data mapping. Third, related segments are written to the same blocks allowing for read-ahead of segments.

The SAN file system of Hong et al. [100] avoids the fingerprint-to-block index overhead of Venti, since data is accessed by blocks (redundancy detection is done after writing the data). But a multi-level index of fingerprint to segment locations is maintained to improve the performance of fingerprint searches. The first level is 64 MB in size, and contains pointers to second level buckets indexed using the first 24 bits of the SHA-1 fingerprint. Each second level bucket contains a 32 bit segment location and the next 32 bits of the SHA-1 fingerprint. Since the maximum number of segments stored for an object is 2^{32}, each bucket index is on the average 256 blocks.

Lookup management in fingerdiff [39] is implemented using a variant of a binary search tree. Such a tree avoids the time and space overheads of hash tables, caused by the random distribution of fingerprint values leading to similar segments having completely different hash-key values, and non-similar segments having common SHA-1 hash substrings. The binary search tree reduces memory usage, since it avoids storing repeated SHA-1 substrings, and can be dynamically adjusted depending on the SHA-1 hash values in use, without increasing search time. For 20-byte SHA-1 fingerprints the tree has 20 levels. At the i^{th} level, the i^{th} byte in the SHA-1 fingerprint is used to decide which of the 256 possible children to check in the $i+1^{th}$ level. To further reduce storage different data structures are used for tree nodes depending on the number of children, and linear paths are only stored at the root. However, the design assumes that the entire tree fits in memory.

Denehy and Hsu [72] implemented a reliable storage system for reference data that stores multiple copies of redundant data segments. The primary data structure for accessing segments is a table containing the segment locations for each object. But there is also a secondary data structure used to maintain reference count and location of replicated segments. This data structure is indexed using fingerprints calculated for each segment. The fingerprints are clustered based on when the segment was created or updated. This allows read-ahead of fingerprint values by exploiting the observation that similar objects only differ in a few segments, such that for these sequences of similar fingerprints tend to be read. In addition, the temporal locality of data usage is exploited, such that entries are only kept for a short period of time in memory and then archived. Archived entries are only read from the archive when recovering from a read error.

The problem of using a two-level data structure where the second level “tables” are located using the fingerprint values is that these will be accessed uniformly, and hence cannot efficiently be cached in memory (assuming that fingerprints have the desired property of no locality). Canidae attempts to exploit fingerprint access locality with respect to segment creation time by storing segments created at the same time in the same container. This allows containers to be cached in memory, but requires linearly searching the containers to find a fingerprint. In addition the optimizations described above reduce the number of disks accesses.

4.4.4 Commercial WAN accelerators

Several commercial WAN accelerators have recently been introduced including Riverbed RiOS [186], Cisco WAAS [62], Juniper Networks WAN accelerator [120], F5 Networks BIG-IP [79], Blue Coat's MACH5 [38]. These are proxy systems, typically combining fingerprint based caching, local compression, protocol optimization, and TCP optimization.

The cache in RiOS and WAAS probably consist of fingerprinted content-based segments (in RiOS a 16 byte fingerprint is sent for segments that are on the average100 bytes in size, and cached on disk [186]). Juniper combines a custom compression method with fingerprinting, while MACH5 combines duplicate elimination with fingerprinting. All systems combine fingerprinting with Lempel-Ziv compression of non-redundant data.

In addition to compression, these systems apply WAN specific optimizations to application protocols such as the Common Internet File System (CIFS), HTTP, Microsoft Exchange, and the Message Application Programming Interface (MAPI). RiOS, WAAS, and Juniper also optimize TCP by using larger windows to allow more data in flight and repacking of data.

The BIG-IP WAN accelerator [79] runs a local compression engine on a proxy machine and dynamically tunes the compression ratio and compression time to achieve the best reduction in transfer time.

Canidae has the same architecture as these commercial systems, and use many of the same techniques, but it is designed to work well with multi-dimensional data sets.

4.4.5 Remote visualization

Varg was based on the VNC [184] thin client remote visualization system. Other thin client remote visualization systems include Microsoft Remote Desktop [67], Sun Ray [200] and THINC [25]. In such systems all graphical processing is at the server. The clients only forward mouse and keyboard events to the server and apply updates received from the server to a local framebuffer, and are therefore easy to implement, maintain, and port.

The display updates sent in VNC consist of raw pixels read from the server's framebuffer. Since, processing of application display commands is decoupled from the generation of display updates, the clients are stateless and hence very portable. In addition it is easy to send the updates through a compression engine such as Canidae. Other systems, such as Microsoft Remote Desktop, have a rich set of low-level commands for encoding the updates sent over the network. But, these commands do not compress well leading to decreased WAN performance [131]. Sun Ray achieves better WAN performance since a few low-level operations are used [131]. THINC provides an efficient mechanism, in the form of a device drive, to translate application display commands into a command set similar to that being used in Sun Ray. The disadvantage of THINC is that the device driver needs to be ported to different platform. In addition, our experience with deploying the Varg system is that users are often reluctant to install new software on their machines. Hence, we believe the VNC approach has advantages in that both the server and client can be implemented in Java, and therefore started by a single mouse click in a web browser. Interesting future work would be to use Varg redundancy detection on screenshots to detect the

same operations on pixels that THINC detects (for example a move of a region pixels).

Early remote visualization systems such as Unix X [198] have thick-clients doing all display processing, and high level graphics primitives for sending display updates. This approach has several disadvantages. First, the client is more complex and therefore harder to implement, maintain and port. Second, the high-level operations are not bandwidth efficient. [131, 200] Third, the client and server need rapid synchronization that decreases performance for high latency wide area networks [131, 200]. The cost of synchronization also decreases performance of replicated application-sharing systems [28].

Remote visualization systems for Grids are often application specific, and often the purpose is to provide access to powerful 3D graphics rendering machines such as in the SGI OpenGL Vizserver system [203] or the cluster based Chromium system [104]. The disadvantage of this approach is that it requires writing applications using specific graphics primitives.

The Scalable Adaptive Graphics Environment (SAGE) [116, 179] is a middleware system for streaming high-resolution graphics over local area-, and wide area networks. SAGE differs from Varg in that it is designed to utilize high bandwidth networks, and hence does not require the compression ratio necessary for shared WANs.

Thin-client systems have previously been benchmarked using video playback [159] and Web browsing [60, 159]. For both the user interaction rate will be very low, and hence the benchmark results will not be realistic for the type of interaction required by Genomics analysis tools.

4.4.6 Local compression

Local compression libraries such as zlib [9], bzip2 [202] and rar [188] combine several compression methods. This section describes the most commonly used.

The Lempel-Ziv (LZ77) [244] algorithm detects duplicate strings in a sliding window and replaces these with a back-reference to the previous location of the string. In zlib the default sliding window size is typically 32 KB, and the strings lengths are 3—258 bytes. Hence, redundancy detection is within a local scope. The search for strings is computationally expensive, but to reduce compression time the number of string lengths to check can be reduced. The algorithm complements global compression algorithms since it can be used to detect redundancy within a small segment.

Huffman [103] entropy encoding replaces symbols with weighted symbols based on frequency of use. The resulting Huffman tree provides prefix-free codes that express the most common symbols using fewer bits than less commonly used symbols. In zlib, Huffman encoding is run after the Lempel-Ziv algorithm, creating a tree with space for 288 symbols. Huffman encoding can also be applied to segment data, and it can significantly compress segments with few symbols such as pixel values.

Run-length encoding (RLE) replaces long runs of symbols with a single <data value, count> encoding. A variant of RLE is used by remote visualization systems (e.g. in the VNC Hextile [183] and RRE encodings [183]), to split a region into smaller regions that can be represented using a single <pixel value, region position, region size>. RLE is fast to compute and can therefore be applied to segments if the overhead of Lempel-Ziv and/or Huffman is too large.

Value-prediction can be used for lossless compression of floating point data [176]. This algorithm predicts the next value, XOR encodes the difference between the predicted value and the actual value, and then encodes the resulting value with a leading zero count and the remaining bits. The assumption is that the sign, exponent, and top mantissa bits, stored in the most significant bits, are easy to predict and hence will be zero after the XOR. This compression may work well with segments produced for a scientific dataset.

4.5 Conclusions

This chapter has presented the design, implementation, and initial evaluation of the Canidae and Varg systems that reduce the bandwidth requirements of distributed applications.

Canidae is a network data compression framework that allows multiple, data-specific segmentation methods to share a segment compression engine. A two-level fingerprinting protocol has been proposed to improve redundancy elimination by allowing using smaller segments than previous global compression systems. Also proposed is a novel method to compress 2-D pixel segments by using fingerprinting. In addition we propose a segment cache on disk used to store a hundred GB dataset, and optimized to reduce disk accesses by using a Bloom filter.

The requirements for a 100 GB data set were modeled. Our results shows that two-level fingerprinting is most useful for segment sizes ranging from 16 to 256 bytes. In order to get the best trade-off between fingerprint bytes, and collision bytes the optimistic fingerprint size should be 40 bits, and a conservative fingerprint should cover about 20—25 segments. In addition, we demonstrated the need for a large segment cache, and the benefits of such a cache.

Varg allows users to interactively visualize multiple remote genomic applications across a WAN. We found that genomic applications have much higher network bandwidth requirements than office applications, and hence require substantial compression of network data to achieve interactive remote data visualization on some WANs.

An initial evaluation of the Varg system shows that the proposed 2-D pixel segment compression method works well and imposes only modest overheads. By combining with zlib and differencing compression methods, the prototype system achieved compression ratios ranging from 30:1 to 289:1 for three genomic visualization applications that we have experimented with. Such compression ratios allow the Varg system to run remote visualization of genomic data analysis applications interactively across WANs with relatively low available network bandwidths.

The Canidae system provides a framework that can be used to implement application specific compression methods for large-scale data storage infrastructures, or between other sites where collaborative work requires sending large amounts of complex data, such as in remote visualization. The compression method complements the methods for reducing the latency requirements of parallel applications. Combined these allows to efficiently transfer a large dataset to a remote cluster, run a parallel application analyzing the data on a federation of clusters, and remotely visualize the results.

Chapter 5

Conclusions

This dissertation has presented and evaluated how end-to-end performance of distributed and parallel applications can be improved by better CPU utilization, reducing latency of communication primitives, overlapping computation-communication, and lowering bandwidth requirements.

Chapter 2 presented two approaches for improving parallel application scalability by reducing the latency of collective communication. First, performance monitoring and analysis was used to adapt collective communication to the application and platform in use. Collective communication performance analysis required message traces collected internally in the communication system. We proposed monitoring methods to reduce the high storage requirements of such data collection, and to satisfy the computation requirements of the data processing. Second, a commonly used collective operation was extended with knowledge about how the result was used in order to reduce the number of messages sent over WANs. The improvements due to the performance analysis, the performance and perturbation of the implemented monitors, and the improvements due the changed collective operation, were documented with benchmarks run on Ethernet and WAN multi-clusters. The following contributions were made:

1. A monitoring framework that supports the development of runtime monitors for parallel applications. Monitors can be tuned to trade-off between performance and perturbation. In particular:
   - Message traces are stored in small buffers and processed at the rate data is produced by monitor threads run on the cluster nodes to reduce the memory footprint.
   - Monitor and application threads are coscheduled to reduce monitor perturbation.
   - Documentation that data for collective communication performance analysis can be processed with insignificant perturbation.
2. A performance analysis method to identify bottlenecks due to network latency, synchronization overhead, and computation overhead, and:
   - Demonstration that such a method can be used to reduce collective operation latency.
3. We described how the allreduce operation can be modified to reduce the number of messages sent over high latency networks, without changing the application results, and:
   - Demonstrated that the conditional-allreduce operation has similar latency on a WAN multi-cluster as on an Ethernet cluster.

In *Chapter 3* parallel applications were overdecomposed to introduce thread level parallelism in order to reduce execution time by overlapping communication wait time with computation, and by utilizing SMT processors. The improvements and limitations of overdecomposition were documented by collecting data during the execution of benchmarks run on a cluster composed of the first generation SMT processors. The contributions are:

4. Method for identifying TLP improvements and overdecomposition overheads using data from multiple software and hardware layers.
5. Demonstration that overdecomposition can reduce the execution time of parallel applications, and identification of significant performance limitations.

Chapter 4 presented an approach for reducing the network bandwidth requirements of remote visualization, and for applications transferring large scientific data sets. We proposed a lossless global compression system for multi-dimensional network data. Performance improvements of data transfer are validated with experiments. The contributions are:

6. A framework for global compression using two-level fingerprinting and application specific segmentation, and:
   - Redundancy detection is separated from redundancy elimination, such that application specific segmentation algorithms can be implemented to improve redundancy detection.
   - Two-level fingerprinting protocol that improves redundancy detection by using smaller segments, while maintaining data consistency.
   - The design and implementation of a very large cache on disk for storing previously sent segments to improve compression ratio.
7. A network bandwidth optimized, platform-independent remote visualization system. In particular:
   - Two-phase fingerprinting protocol for reducing compression time.
   - Redundancy detection algorithm for 2-D pixel data that exploit data movement on screens to improve compression ratio.
   - Demonstration that the system allows visualization of genomic data analysis applications interactively across WANs with relatively low available network bandwidth.

Chapter 6

Future Work

The work in this dissertation has explored four approaches for reducing the bandwidth and latency requirements of distributed and parallel applications. Even for these there are unexplored paths, and room for improvement; some of which are described in this section.

6.1 Collective performance analysis and monitoring

Spanning tree reconfiguration for improving collective operation performance could be validated by additional experiments using benchmarks and applications with different communication structures.

To validate that spanning tree adaptation based on performance analysis is better than testing many different algorithms for creating spanning trees [80, 81, 229], measurements should be made to compare the performance of the tuned collective operations, and the time required to find the best configuration.

Better visualizations for presenting the performance results are needed to aid users. Developing scalable visualizations is regarded as an important research problem for performance analysis of applications run on large scale clusters [151, 178, 217, 238, 243]. High resolution displays may be of use to visualize such data as demonstrated in [96]. For non-expert users it may be necessary to automate the analysis and tuning of collective communication. Using our monitoring approach, automatic performance tuning tools [151, 182] could be implemented to use underutilized cluster resources.

Some parallel applications are implemented by decomposing an unstructured graph to threads, and using asynchronous point-to-point communication operations for exchanging data. For one such application [54], we were not able to understand a communication performance problem using traces collected using the MPI profiling layer. The necessary insight to understand the problem could be provided by using message traces collected internally in the communication system to create visualization that show which threads wait, and have waited, for which other threads during the execution.

6.2 Collective operations for WANs

Tools analyzing a parallel applications source code could be used to automatically detect which collective operations can be made conditional.

To validate the applicability of WAN multi-clusters for real parallel applications, experiments should be conducted to compare the performance of applications run on a single cluster, versus an application run on a multi-cluster that is optimized with all the methods presented in this dissertation.

6.3 Overdecomposition

The improvements and limitations of overdecomposition should be validated on CMP processors.

Multiprogramming can be used to also overlap collective communication wait time. The coscheduling method we developed for the EventSpace monitor system could be used to allow two parallel applications to share a cluster without performance degradation of the *primary* application. We believe this functionality can be implemented in the communication system. Such scheduling also needs to take into account cache pollution, and provide predictable and low latency communication operations.

System software could be changed to avoid the overdecomposition limitations identified for the NAS benchmarks. Especially, the following changes should be investigated; each motivated by the limitations for a benchmark:

- In BT and SP each thread communicates with multiple neighbors, and computation-communication overlap is implemented using the immediate functions provided by MPI. A better mapping of threads to processors may reduce the inter-node communication overhead.
- Applying overdecomposition for CG and MG decreases performance due to increased cache misses and operating system activity. Therefore, intra-node communication should use asynchronous communication operation that provides low memory-bandwidth user-level non-blocking synchronization.
- System software should be changed such that a high degree of TLP is maintained, even at the cost of work conservation.

6.4 Remote visualization

Performance comparison of remote visualization using Varg and other thin-client systems should be conducted by using the systems over real WANs.

A longer trace should be used to evaluate the compression ratio and time achieved using Varg.

The scalability of theVarg system should be demonstrated by compressing the data for cross Atlantic collaboration using display walls.

6.5 Two-level fingerprinting

Improvements to Canidae throughput, and the end-to-end latency of individual segment transfers, should be evaluated using a sender cache. This cache could be implemented either using a hash table or a Bloom filter.

The latency of segments sent through Canidae should be measured by compressing data by a Varg type remote visualization.

The effect on compression ratio and compression time of using bi-directional communication should be measured.

The benefit of a hundred GB cache should be validated using a larger data set than the genomic visualization data sets used in the initial evaluation.

A full evaluation of the advantage of two-level fingerprinting is needed to answer the following questions:

- Does two-level fingerprinting improve the compression ratio of previous 1-D content-based segmentation algorithms?
- Does redundancy detection improve with smaller segments?

The segment cache should be evaluated by answering the following:

- Are the assumptions that segment accesses have temporal and spatial locality true?
- How should the parameters be set to achieve the best performance, in particular: the number of hash table entries, segment buffer chunk size, and the memory allocated for hash tables versus segment chunks?
- Does the choice of container replacement algorithm significantly improve cache hit ratio?

In addition the system should be properly evaluated by measuring:

- The throughput of Canidae compression.
- The scalability of the Canidae compression pipeline on current CMP and SMT processors.

To answer the first two questions the compression ratio of the implemented 1-D segmentation algorithms with different segment sizes should be measured when using one level fingerprinting, and when using two-level fingerprinting. A data set consisting of flat uncompressed files could be used for the experiments. The last two questions can be answered by measuring the throughput of the system on machines with processors supporting CMT and SMT.

6.6 Segmenting multi-dimensional datasets

Segmentation algorithms for scientific 2-D data or 3-D data should be developed, and used to demonstrate network transfer time improvements when used to detect redundancy in multi-dimensional scientific datasets.

Methods could be developed for predicting the segmentation method and segment size to use for a given dataset.

Additional experiments are required to answer the following questions:

- Does 2-D based segmentation methods improve the compression ratio and time compared to 1-D segmentation methods?
- Does the 2-D based segmentation methods also work well with scientific 2-D data?

To answer these question it is necessary to experiment with existing 1-D segmentation methods [156, 209], and the 2-D segmentation methods we have developed. The data sets to use in the evaluation should be from different scientific domains, such as geometric data, scientific simulation output, and satellite images.

Chapter 7

Appendix A - Published papers

7.1 Collective Communication Performance Analysis Within the Communication System

This paper was published in the *Proceedings of Euro-Par 2004* [43].

An earlier version of the paper was published as a technical report [44].

Collective Communication Performance Analysis Within the Communication System

Lars Ailo Bongo, Otto J. Anshus, and John Markus Bjørndalen

Department of Computer Science, University of Tromsø, Norway
{larsab, otto, johnm}@cs.uit.no

Abstract. We describe an approach and tools for optimizing collective operation spanning tree performance. The allreduce operation is analyzed by collecting performance data at a lower level than traditional monitoring systems. We calculate latencies and wait times to detect load balance problems, find subtrees with similar behavior, do cost breakdown, and compare the performance of two spanning tree configurations. We evaluate the performance of different configurations and mappings of allreduce run on clusters of different size and with different number of CPUs. Monitoring overhead is low, and the analysis is simplified since many subtrees have similar behavior. However, the calculated values have large variations, and reconfiguration may affect unchanged parts. Despite these problems, we achieve a speedup of up to 1.49 for allreduce.

1 Introduction

Clusters are becoming an increasingly important platform for scientific computing. Many parallel applications run on clusters use a communication library, such as MPI [8], which provides collective operations to simplify the development of parallel applications. Of the eight scalable scientific applications investigated in [15], most would benefit from improvements to MPI's collective operations.

The communication structure of a collective operation can be organized as a spanning tree, with threads as leafs. Communication proceeds along the arcs of the tree and a partial operation is done in each non-leaf node. Essential for the performance of a collective operation is the shape of the tree, and the mapping of the tree to the clusters in use [6, 11, 12, 13].

We present a methodology and provide insight into performance analysis within the communication system. We demonstrate and evaluate the methodology by comparing and optimizing the performance of different allreduce configurations.

Performance monitoring tools for MPI programs [7] generally treat the communication system as a black box and collect data at the MPI profiling layer (a layer between the application and the communication system). To understand why a specific tree and mapping have better performance than others it is necessary to collect data for analysis inside the communication system. We describe

our experiences about what type of data is needed, how to do the analysis, and what the challenges are for collective communication performance analysis.

Usually, MPI implementations only allow the communication structure to be implicitly changed either by using the MPI topology mechanism or by setting attributes of communicators. To experiment with different collective communication configurations, we use the PATHS system [2], since it allows to inspect, configure and map the collective communication tree to the resources in use.

For a given tree configuration and mapping, our analysis approach and visualizations allows us to find performance problems within the communication system, and to compare the performance of several configurations. This allows us to do a more fine grained optimization of the configuration than approaches that only use the time per collective operation (as in [13]).

In addition to remapping trees, collective operation performance can be improved by taking advantage of architecture specific optimizations [11, 12], or by using a lower-level network protocol [5, 12]. However, the advantage of these optimizations depends on the message size. For example, for small message sizes, as used by most collective operations, point-to-point based communication was faster than Ethernet based broadcast in [5], and a shared memory buffer implementation for SMPs in [11]. Our approach allows comparing advantages of changing the communication protocol, or synchronization primitives for different message sizes.

On SMPs, collective communication performance can also significantly be reduced, by interference caused by system daemons [9]. Reducing the interference will make the performance analysis within the communication system even more important.

Mathematical models can be used to analyze the performance of different spanning trees (as in [1]), but these do not take into account the overlap and variation in the communication that occurs in collective operations [13].

For each thread, our monitoring system traces messages through a *path* in the communication system. We calculate latencies and wait times, and use these to detect load balance problems, find subtrees with similar behavior, do cost breakdown for subtrees, and compare the performance of two configurations.

Monitoring overhead is low, from nearly 0 to 3%. Analysis is simplified since many subtrees have similar behavior. However, the calculated values have large variation, reconfiguration may affect unchanged parts, and predicting the effect of reconfigurations is difficult. Despite these problems we achieved a speedup of up to 1.49 for an allreduce benchmark using our tools.

The rest of this paper proceeds as follows. PATHS is described in section 2. Our monitoring tool and analysis approach are described in section 3 and demonstrated in section 4. In section 5 we discuss our results, and finally, in section 6 we conclude and outline future work.

2 Reconfigurable Collective Operations

We use the PATHS system [2] to experiment with different collective operation spanning tree configurations and mappings of the tree to the clusters in use. In allreduce, each thread has data that is reduced using an associative operation, followed by a broadcast of the reduced value. The semantics differs from MPI in that the reduced value is stored in the PastSet structured shared memory [16].

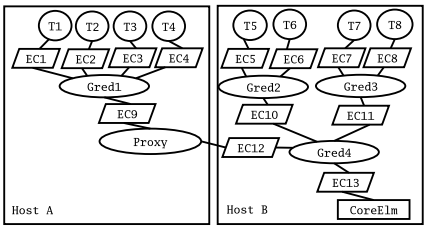


Fig. 1. An allreduce tree used by threads T1–T8 instrumented with event collectors (EC1–EC13). The result is stored in a PastSet element (CoreElm).

Using PATHS we create a spanning tree with all threads participating in the allreduce as leafs (figure 1). For each thread we specify a *path* through the communication system to the root of the tree (the same path is used for reduce and broadcast). On each path, several *wrappers* can be added. Each wrapper has code that is applied as data is moved down the path (reduce) and up the path (broadcast). Wrappers are used to store data in PastSet and to implement communication between cluster hosts. Also, some wrappers, such as allreduce wrappers, join paths and handle the necessary synchronization.

The PATHS/PastSet runtime system is implemented as a library that is linked with the application. The application is usually multi-threaded. The PATHS server consists of several threads that service remote clients. The service threads are run in the context of the application. Also, PastSet elements are hosted by the PATHS server. Each path has its own TCP/IP connection (thus there are several TCP/IP connections between PATHS servers). The client-side stub is implemented by a *proxy* wrapper. Wrappers are run in the context of the calling threads, until a wrapper on another host is called. These wrappers are run in the context of the threads serving the connection.

In allreduce, threads send data down the path by invoking the wrappers on their path. The allreduce wrappers block all but the latest arriving thread, which is the only thread continuing down the path. The final reduced tuple is stored in the PastSet element before it is broadcasted by awakening blocked threads that return with a copy of the tuple. For improved portability, synchronization is implemented using Pthread condition variables.

3 Monitoring and Analysis

To collect performance data we use the EventSpace system [3]. The paths in a collective operation tree are instrumented by inserting *event collectors*, implemented as PATHS wrappers, before and after each wrapper. In figure 1, an allreduce tree used by threads T1–T8 is instrumented, by event collectors EC1–EC13. For each allreduce operation, each event collector records a timestamp when moving down, and up the path. The timestamps are stored in memory and written to trace files when the paths are released. In this paper analysis is done post-mortem.

Depending on the number of threads and the shape of the tree, there can be many event collectors. For example, for a 30 host, dual CPU cluster, a tree has 148 event collectors collecting 5328 bytes of data for each call (36 bytes per event collector). The overhead of each event collector is low (0.5 μs on a 1.4 GHz Pentium 4) compared to the hundreds of microseconds per collective operation. Most event collectors are not on the slowest path, thus most data collecting is done outside the critical path.

There are three types of wrappers in an allreduce spanning tree: gred (partial allreduce), proxy (network) and core (PastSet). For core wrappers the two timestamps collected by the event collector above it (EC13 in figure 1) are used to calculate the *store latency*; the time to store the result in PastSet. For proxy, we calculate the two-way TCP/IP latency by $(t_4 - t_1) - (t_3 - t_2)$, where t_1 (down) and t_4 (up) are collected by the event collector above the proxy in a path, and t_2 (down) and t_3 (up) are collected by the event collector below. To achieve the needed clock synchronization accuracy for calculating one-way latencies (tens of μs) special hardware is needed [10].

Gred wrappers have multiple children that contribute with a value to be reduced. The contributor can be a thread or data from another gred wrapper (in figure 1 threads T5–T6 contribute to gred2, while gred1–3 contribute to gred4). There is one event collector on the path to the parent that collects timestamps t_2 and t_3, while the paths from the P parents each have an event collector collecting timestamps $t_{1,i}$, and $t_{4,i}$. We define the *down latency* for a gred wrapper to be $t_2 - t_{1,l}$, the down latency for the last arrival l. The *up latency* is $t_{4,f} - t_3$, the up latency for the first departurer f.

For a given number of collective operations we calculate for each participant the *arrival order distribution* and the *departure order distribution*; that is the number of times the contributor arrived and departed at the gred wrapper as the first, second, and so on. In addition we calculate: *arrival wait time* $t_{1,l} - t_{1,i}$; the amount of time the contributor i had to wait for the last contributor l to arrive, and *departure wait time* $t_{4,i} - t_{4,f}$; elapsed time since the first contributor f departed from the gred wrapper, until contributor i departed.

For the analysis we often divide the path from a thread to a PastSet element into several stages consisting of the latencies and wait times described above. To calculate the time a thread spent in a specific part of the tree (or a path), we add together the time at each stage in the tree (or path). Usually mean times are

used. Similarly, we can do a hotspot analysis of a tree by comparing the mean times.

For the performance analysis we (i) detect load balance problems, (ii) find paths with similar behavior, (iii) select representative paths for further analysis, (iv) find hotspots by breaking down the cost of a path into several stages, (v) reconfigure the path, and (vi) compare the performance of the new and old configuration.

For applications with load balance problems, optimizing collective operations will not significantly improve performance since most of the time will be spent waiting for slower threads. To detect load balance problems, we use the arrival order at each gred wrapper to create a weighted graph with gred wrappers and threads as nodes, and the number of last arrivals as weights on the edges. The part of the tree (or thread) causing a load imbalance can be found by searching for the longest path (i.e. with most last arrivals).

Usually, many threads have similar paths, with similar performance behavior. Thus, we can simplify the analysis and optimization by selecting a representative path for each type of behavior. Also, fast paths can be excluded since the allreduce operation time is determined by the slow paths. We have experimented with, but not succeeded in finding analysis and visualization approaches that allows for a detailed analysis of multiple paths at the same time.

When analyzing a representative path we break the cost of an allreduce operation into subtrees that can be optimized independently such as the subtree for an SMP host, a cluster, or TCP/IP latencies within a cluster and on a WAN. As for a gred wrapper, we calculate for a subtree the down latency, up latency, and departure wait time (using for the subtree on host B in figure 1 event collectors EC5–8 and EC13). A large down latency implies that the computation in the operation takes a long time. A large up latency indicates performance problems in the gred implementation, while a large departure wait time implies scalability problems of the synchronization variables, leading to unnecessary serialization.

4 Experiments

In this section we analyze the performance of different allreduce configurations for a blade cluster with ten uni-processor blades (Blade), a cluster of thirty two-way hosts (NOW), a cluster of eight four-way hosts (4W), and a cluster of four eight-way hosts (8W). The clusters are connected through 100 Mbps Ethernet, and the operating system is Linux. We designed experiments to allow us to measure the performance of different shapes and mappings of the spanning trees used to implement allreduce. A more detailed analysis can be found in [4].

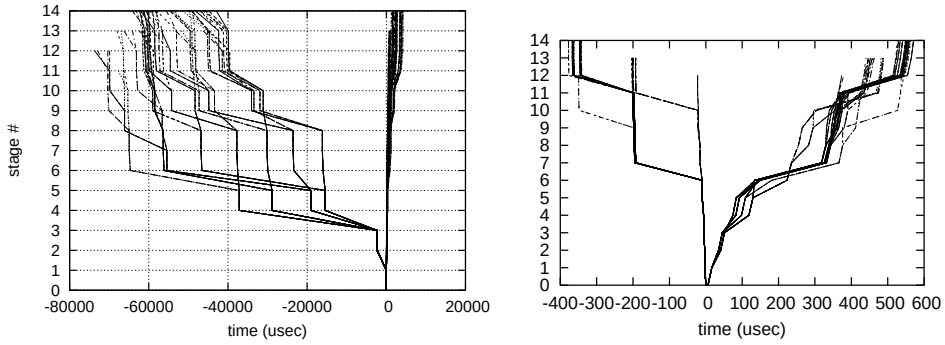
We use a microbenchmark, Gsum, which measures the time it takes T threads to do N allreduce operations. The allreduce computes a global sum. The number of values to sum is equal to the number of threads, T. Threads alternate between using two identical allreduce trees to avoid two allreduce calls to interfere with each other.

We also use an application kernel, SOR, a red-black checker pointing version of successive over-relaxation. In each iteration, black and red points are computed and exchanged using point to point communication, before a test for converge which is implemented using allreduce. A communication intensive problem size was used (57% of the execution time was spent communicating). The computation and point-to-point communication results in a more complex interaction with the underlying system than in Gsum.

Gsum was run ten times for 25000 iterations on each cluster, but only the results from one execution are used in the analysis. The Gsum execution time has a small standard deviation (less than 1%). Also, the slowdown due to data collection is small (from no slowdown up to 3%). For SOR the execution time variation is similarly low, and the monitoring overhead is lower since SOR has relatively less communication than Gsum.

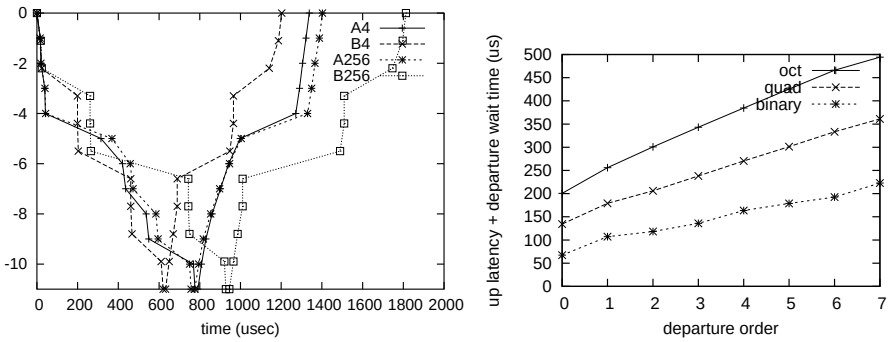
The standard deviation for the allreduce operation time is large. On the NOW cluster the mean is about 1000 μs and the standard deviation is about 200 μs. Also the variation is large for the computed latencies and wait times. For many stages both the mean and the standard deviation are about 10–25 μs. The only values with low standard deviation are the TCP/IP, and store latencies. Despite the large variation, using mean in the analysis gives usable results.

The large standard deviation for gred up, and down latency is caused by queuing in the synchronization variables, while the departure wait time distribution is a combination of several distributions since the wait time depends on the departure order (which, in our implementation, depends on the arrival order). We expect most implementation to have similar large variations for these stages. For Gsum, arrival wait time variation is caused by variations on the uppath, while for SOR and other applications there are additional noise sources such as system daemons [9].



(a) SOR configuration with arrival wait times. (b) Gsum configuration without arrival wait times.

Fig. 2. Timemap visualizations for NOW cluster spanning tree configurations.



(a) Timemaps for two 4W configurations when using 4 and 256 byte messages.

(b) Serialization introduced by 8W host subtree configurations.

Fig. 3. Visualizations for comparing the performance of different configurations.

For well balanced applications, such as Gsum and SOR, there is not one thread causing a load imbalance. SOR has a load imbalance when run on the 4W and 8W clusters caused by differences in point-to-point communication latency since two threads on each host communicate with a neighbor on a different host. Due to the load imbalance performance will not significantly be improved by optimizing allreduce as shown in figure 2a, where the down paths ($x < 0$) are dominated by arrival wait time (i.e. load imbalance). Based on our experiences most of the optimizations can be done on the up paths ($x > 0$). SOR can be reimplemented to hide the point-to-point communication latency, and thereby reducing the load imbalance.

Differences in network latency also cause a load imbalance within the communication system when Gsum is run on a multi-cluster. However, the problem is caused by the spanning tree and can be improved by reconfiguring the tree. For the remaining analysis we ignore the arrival wait times, since these hide the difference between fast and slow paths.

To get an overview of the communication behavior of the different threads we use a *timemap* visualization that shows the mean time spent (x-axis) in each stage of the path (y-axis) when moving down and up the tree. $X = 0$ is when the threads enter the bottommost wrapper. For more details we use tables with statistics for each thread, and for each stage. The timemap in figure 2b shows that the 60 threads run on the NOW cluster have similar behavior. Arrival wait times are not shown, and the up-path has variations due to the arrival-departure order dependency. The threads can roughly be divided into classes according to the number of TCP/IP connections in their paths. An optimized configuration for 2W has a more irregular shape complicating the analysis due to rather large variation for most stages.

For the mostly used allreduce message sizes (below 256 bytes [14]), a cost breakdown shows that broadcast is more expensive than reduce. The up, and down gred-latency are only a few μs, hence the time to do the reduce operation

is insignificant. For SMPs the departure wait time can be large, but for the best configuration the TCP/IP stages dominate the execution time. Also, the time spent storing the result in PastSet is insignificant. For some single-CPU cluster configurations we have a few outliers in the proxy stage that can significantly reduce performance. For one Blade Gsum configuration they caused a slowdown of 54.2.

When tuning the performance of a configuration it is important to find the right balance between load on *root* hosts and number of network links. Load on root hosts can be reduced by moving gred wrappers to other hosts. This will also improve potential parallelism, but it will introduce another TCP/IP connection to some threads paths. For SMPs, the performance of a host subtree can be improved by adding or removing an additional level in the spanning tree.

Reconfiguration improved the performance of Gsum up to a factor of 1.49. However, the effects of a reconfiguration can be difficult to predict. For a 4W Gsum configuration we doubled the number of TCP/IP connections on a threads path, but the time spent in these stages only increased by 1.55 due to the TCP/IP latency being dependent on load on the communicating hosts. A reconfiguration can also have a negative performance effect on unchanged subtrees. In addition, the best configuration for a cluster is dependent on CPU speed on hosts, LAN latency, number of hosts in cluster, and the message size used in the collective operation.

An overview of the differences between two configurations is provided by a timemap visualization that shows several configurations for one thread. Since paths in two configurations can have unequal length, the y-coordinates are scaled such that both have the same y_0 and y_{max}. Figure 3a shows that by moving a gred wrapper to another 4W host (B4 and B256), gives better performance for 4 byte messages, but worse for 256 byte messages, due to a trade-off between increased TCP/IP latencies, and single host subtree performance.

Figure 3b shows how adding additional levels to a subtree improves performance on an 8-way host. The figure shows the introduced latency (x=0), and the amount of serialization (slope of the curve, flatter is better). On the x-axis the order of departure is shown, and on the y-axis the up latency + departure wait time is shown. Notice that the up-latency is the same for all departures, and that for the first departurer (x=0) has zero departure wait time. The optimal height of the tree depends on the load on the hosts.

To find the fastest configuration, it is usually enough to compare the paths that are on the average slowest, but not always. A 4W Gsum configuration A, is slower than configuration B even if the 12 slowest threads are faster, because the remaining 20 threads in A are slower, and the large variation causes A to have the slowest thread for most allreduce calls.

5 Discussion

The timestamps collected by EventSpace allows us to analyze the performance of spanning trees and their mapping to the clusters in use. However, information

from within the operating system is needed to understand why a synchronization operation, or a TCP/IP connection is slow. Hence, information from within the operating system should be collected and used in the analysis.

For ease of prototyping we used our own parallel programming system (PATHS). Our analysis approach should be applicable for MPI runtime systems provided that we can collect timestamps that can be correlated to a given MPI collective operation call. Once monitoring tools using the proposed MPI PERUSE interface[?] are available these may be used to collect the data necessary for our analysis. However, other runtime systems will have performance problems not treated by this work, for example with regards to buffering.

The calculated values used in the analysis have large variation, and outliers can have a significant effect on performance. A complete trace of all messages sent within a given time period provides enough samples for statistical analysis, and can be used to detect any periodical performances faults (e.g. caused by system daemons [9]). EventSpace allows to collect such traces with a small overhead and memory usage (1 MB of memory can store 29.127 EventSpace events).

We only studied one application; SOR. Since MPI defines the semantics of collective operations the communication pattern of SOR is general and frequent. Since SOR has a load imbalance a better application for the study would have been one of the applications described in [9]. Also, only the allreduce operation has been analyzed. We believe the analysis will be similar for other collective operations with small message size such as reduce and barrier. Message arrival order, synchronization points, and network latencies are also important for the performance of operations with larger messages, such as alltoall, allgather, and the MPI-IO collective operations. For MPI-IO we can wrap the I/O operations using PATHS wrappers.

6 Conclusion and Future Work

We have described systems for monitoring and tuning the performance of collective operation within the communication system. For each thread we trace the messages through a *path* in the communication system. We demonstrated an analysis approach and visualizations by evaluating and optimizing different spanning-tree configurations and cluster mappings of the allreduce operation.

The monitoring overhead is low, from nearly 0 to 3%, and the analysis is simplified since many paths have similar behavior. However, the computed latencies and wait times have large variation, reconfiguration may affect unchanged parts, and it is difficult to predict the effect of some changes.

As future work, we will use the EventSpace system [3] for run-time analysis. Also, we will examine how data collected inside the operating system can be used in the analysis, and if some load balance problems can be avoided by reconfiguring the collective operation spanning trees. Finally, our long-term goal is to build a communication system where collective communication is analyzed, and adapted at run-time.

References

[1] Bernaschi, M., and Iannello, G. Collective communication operations: Experimental results vs.theory. *Concurrency: Practice and Experience 10*, 5 (1998).

[2] Bjørndalen, J. M. *Improving the Speedup of Parallel and Distributed Applications on Clusters and Multi-Clusters.* PhD thesis, Department of Computer Science, University of Tromsø, 2003.

[3] Bongo, L. A., Anshus, O., and Bjørndalen, J. M. EventSpace - Exposing and observing communication behavior of parallel cluster applications. In *Euro-Par* (2003), vol. 2790 of *Lecture Notes in Computer Science*, Springer, pp. 47–56.

[4] Bongo, L. A., Anshus, O., and Bjørndalen, J. M. Evaluating the performance of the allreduce collective operation on clusters: Approach and results, 2004. Technical Report 2004-48. Dep.of Computer Science, University of Tromsø.

[5] Karwande, A., Yuan, X., and Lowenthal, D. K. CC-MPI: a compiled communication capable MPI prototype for Ethernet switched clusters. In *Proc. of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming* (2003), ACM Press, pp. 95–106.

[6] Kielmann, T., Hofman, R. F. H., Bal, H. E., Plaat, A., and Bhoedjang, R. A. F. MagPIe: MPI's collective communication operations for clustered wide area systems. In *Proc. of the seventh ACM SIGPLAN symposium on Principles and practice of parallel programming* (1999), pp. 131–140.

[7] Moore, S., D.Cronk, London, K., and J.Dongarra. Review of performance analysis tools for MPI parallel programs. In *8th European PVM/MPI Users' Group Meeting, Lecture Notes in Computer Science 2131* (2001), Springer Verlag.

[8] MPI: A Message-Passing Interface Standard. *Message Passing Interface Forum* (Mar. 1994).

[9] Petrini, F., Kerbyson, D. J., and Pakin, S. The case of the missing supercomputer performance: Achieving optimal performance on the 8,192 processors of ASCI Q. In *Proc. of the 2003 ACM/IEEE conference on Supercomputing* (2003).

[10] Pásztor, A., and Veitch, D. Pc based precision timing without gps. In *Proceedings of the 2002 ACM SIGMETRICS international conference on Measurement and modeling of computer systems* (2002), ACM Press, pp. 1–10.

[11] Sistare, S., vandeVaart, R., and Loh, E. Optimization of MPI collectives on clusters of large-scale SMP's. In *Proc. of the 1999 ACM/IEEE conference on Supercomputing* (1999).

[12] Tipparaju, V., Nieplocha, J., and Panda, D. Fast collective operations using shared and remote memory access protocols on clusters. In *17th Intl. Parallel and Distributed Processing Symp.* (May 2003).

[13] Vadhiyar, S. S., Fagg, G. E., and Dongarra, J. Automatically tuned collective communications. In *Proceedings of the 2000 ACM/IEEE conference on Supercomputing* (2000).

[14] Vetter, J., and Mueller, F. Communication characteristics of large-scale scientific applications for contemporary cluster architectures. In *16th Intl. Parallel and Distributed Processing Symp.* (May 2002).

[15] Vetter, J. S., and Yoo, A. An empirical performance evaluation of scalable scientific applications. In *Proceedings of the 2002 ACM/IEEE conference on Supercomputing* (2002), IEEE Computer Society Press.

[16] Vinter, B. *PastSet a Structured Distributed Shared Memory System.* PhD thesis, Department of Computer Science, University of Tromsø, 1999.

7.2 Low Overhead High Performance Runtime Monitoring of Collective Communication

This paper was published in the *Proceedings of ICPP 2005* [47].

The first paper about the EventSpace system was published in the Proceedings of Euro-Par 2003 [45]. How to use EventSpace to analyze the performance of parallel applications was demonstrated in [18].

Low Overhead High Performance Runtime Monitoring of Collective Communication

Lars Ailo Bongo, Otto J. Anshus and John Markus Bjørndalen
Department of Computer Science, University of Tromsø, Norway
{larsab, otto, johnm}@cs.uit.no

Abstract

Scalability of parallel applications run on clusters and multi-clusters is often limited by communication performance. Message tracing can provide data for understanding bottlenecks, and for performance tuning. However, it requires collecting, storing, analyzing, and transferring potentially gigabytes of data. We have designed the EventSpace system for low overhead and high performance runtime collective communication trace analysis. EventSpace separates the perturbation and performance requirements of data collection, analysis, gathering and visualization. Data collection overhead is low since the minimum amount of data is recorded and stored temporarily in main memory. The recorded data is either discarded or analyzed on demand using available cluster resources. Analysis is distributed for high performance, and coscheduled with the computation and communication system threads for low perturbation. Gathering of analyzed data is done using extensible collective communication operations, which can be tuned to trade off between performance and monitoring overhead. EventSpace was used to do run-time monitoring and analysis of collective communication micro-benchmarks run on clusters, multi-clusters, and multi-clusters with emulated WAN links. Performance data was collected, analyzed and gathered with 0–3% monitoring overhead.

1 Introduction

In Grids rapid changes will be the norm. Hence, it is necessary for applications and the underlying systems to adapt, at run-time, to changes in the availability and performance of resources. An important part of the adaptation will be to reconfigure the point-to-point and collective communication structures used by parallel applications.

On large clusters, a much less dynamic environment than a Grid, communication system performance is important. Of eight scalable scientific application studied in [30], most would benefit from improvements to collective operations, and four would benefit from improvements in point-to-point communication performance. Improved communication performance is essential if Grids are to be used as a high performance computing platform.

Collective operation performance has been shown to improve by using better mappings of computation and data to the clusters in use [16, 24, 26, 27]. In earlier work, we have shown how to tune the mapping based on a performance analysis within the communication system [9]. We found that a global view of the system was needed to detect hotspots and simplify the hotspot analysis. Also, traces of all messages sent in a collective operation spanning tree were needed to understand some performance problems (as the problems described in [21]). Thus, we need to collect, store, analyze, gather, and visualize a large amount of performance data.

Monitoring tools need to collect data with minimal perturbation of the monitored application. For runtime analysis the performance data must be analyzed and often gathered to a single front-end host for use before the data becomes irrelevant. We have built the EventSpace system [8] for low overhead and high performance runtime collective communication trace analysis.

EventSpace is evaluated on clusters, multi-clusters, and multi-cluster with emulated WAN links. We demonstrate how data gathering performance can be tuned to either provide high performance or low perturbation. Our results show that performance data can be collected with less than 1% overhead. The data can be analyzed and gathered with 0–3% overhead, since collective communication intensive applications have low CPU utilization, and since analysis threads can be coscheduled with application and communication system threads.

2 Related Work

Generally performance monitoring tools for MPI programs [19] treats the communication system as a black box

and collect data at a layer between the application and the communication system (the MPI profiling layer). To understand why a specific collective operation spanning tree and mapping have better performance than others it is necessary to collect data for analysis inside the communication system, as EventSpace does.

MRNet [23] is the system most similar to EventSpace. Both use collective operations spanning trees to build scalable multi-cast/reduction overlay networks used by performance monitoring tools. MRNet shares the flexible organization and extensibility of EventSpace. In MRNet, communication is only between compute hosts and the front-end host, while EventSpace allows arbitrary communication structures resulting in more flexible and efficient analysis. EventSpace is also more tightly integrated with the underlying communication system, allowing the monitor activity to be coscheduled with the application. Our evaluation differs in that we use EventSpace for a different problem domain than used in [23], and we examine the performance of more complex spanning tree topologies than the balanced trees used in [23]. Another data aggregation tool for Grids is Yggdrasil [4].

PHOTON [28] allows monitoring point-to-point operations used by MPI applications run on large clusters. EventSpace is designed for collective operations, but share the same goals as PHOTON in reducing the monitoring overhead, perturbation and storage requirements of post-mortem trace analysis tools. PHOTON appends information to messages, which requires modifications to the MPI runtime system. This information is sampled and statistics are computed at runtime. Our experience in collective operation analysis [9] is that statistical profiling does not provide the necessary level of detail to understand all performance problems. Hence message tracing is necessary.

NetLogger [25] provides end-to-end application and system level monitoring of high performance distributed systems. It can provide similar performance data as EventSpace does. However, our focus is on how to aggregate and analyze the communication performance of collective operations. This requires monitoring more hosts than the single path usually monitored by NetLogger.

Data stream management systems (for an overview of DSMSs see [3]) have been used to implement network monitors [12]. DSMSs provide a relational/ query interface for the performance analyst. Such an interface could be useful for specifying EventSpace scopes as SQL queries. However, to achieve the desired performance and perturbation, it is still necessary to map, configure and tune the query plan to the clusters in use; as shown in this paper.

Astrolabe [22] is a system for collecting, aggregating and updating large scale system state. Astrolabe is targeted for widely distributed applications and the primary design goal was scalability. EventSpace uses some of the Astrolabe techniques for improving scalability such as hierarchies and aggregation. Other aggregation and filtering systems for Internet are publish-subscribe systems [10], and Grid monitoring and discovery services such as Remos [13]. The filtering and aggregation functions in EventSpace are more specialized towards performance analysis. Also, since Astrolabe and publish-subscribe systems are targeted at widely distributed applications run on the Internet, low latency aggregation is not important.

Cluster monitoring tools such as Ganglia [18], and Grid monitoring tools such as the Network Weather Service [32], does not support the high sample rate necessary for collective operation analysis.

To reduce monitoring overhead, EventSpace coschedule execution of monitoring threads with application and communication system threads. Coscheduling has traditionally been used to schedule communicating processes [1]. Our design is similar to [11], where coscheduling is used to boost the priority of communication threads doing collective communication to improve application performance. However, we do not modify kernel code since coscheduling can be added to the communication system.

Many research projects have optimized MPI collective operations. Some of the approaches used are: (i) using knowledge about the topology hierarchy, going from multi-cluster [16] to individual clusters of SMPs [24, 17] and uniprocessors. (ii) taking advantage of architecture specific optimizations [24, 26], (iii) using a lower-level network protocol [14, 26], and (iv) automatically trying different algorithms and buffer sizes [27].

3 Performance Analysis and Optimization

Applications monitored by EventSpace use the PATHS communication system [5], which is an extension to the PastSet structured shared memory system [31]. Threads communicate by reading and writing tuples to shared memory buffers.

The purpose of the analysis is to detect performance problems in a spanning tree and understand how the tree can be reconfigured to improve performance. We briefly describe the metrics computed for the allreduce operation. Other synchronizing collective operations will have similar metrics. For a more detailed description see [9].

Central to the analysis are communication *paths* through the communication system starting from a thread and ending in a PastSet buffer. Each path consists of several *wrappers*; each wrapper has code that is run before and after calling the next wrapper in the path. Wrappers are used to implement communication between hosts and for instrumentation. Also, some wrappers join paths used to implement collective operation spanning trees, and handle the necessary synchronizations. The spanning tree is configured by

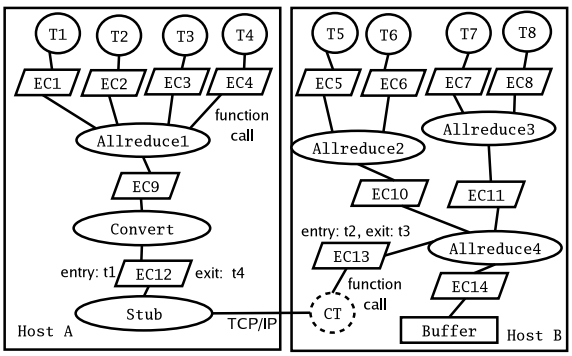


Figure 1. PATHS allreduce spanning tree.

specifying properties of the wrappers and the mapping of wrappers to cluster hosts [5, 9].

In summary, we do for the performance analysis the following steps: (i) detect load balance problems, (ii) find paths with similar behavior, (iii) select representative paths for further analysis, (iv) find hotspots by breaking down the cost of a path into several stages, (v) reconfigure the path, and (vi) compare the performance of the new and old configuration.

Figure 1 shows an allreduce spanning tree used by threads T1–T8 instrumented with *event collectors* (EC1–EC14). These collect entry and exit timestamps for each wrapper. The reduced value is stored is a PastSet buffer. CT is a communication thread serving one TCP/IP connection.

For inter-host communication we calculate the two-way TCP/IP latency by $(t_4 - t_1) - (t_3 - t_2)$, where t_1 and t_4 are collected by the event collector before the stub in a path (EC12), and t_2 and t_3 are collected by the first event collector called by the communication thread (EC13).

Allreduce wrappers are called by multiple threads each contributing with a value to be reduced. There is one event collector after the allreduce wrapper, that collects timestamps t_2 and t_3, while the paths from each contributor i have an event collector collecting timestamps $t_{1,i}$ and $t_{4,i}$. For each contributor three latencies are calculated: *down latency* $t_2 - t_{1,i}$, *up latency* $t_{4,i} - t_3$, and *total latency* $(t_{4,i} - t_{1,i}) - (t_3 - t2)$.

Also calculated for each contributor are the *arrival order distribution* and the *departure order distribution*; the number of times the contributor arrived, and departed, at the allreduce wrapper as the first, second, and so on. In addition we calculate: *arrival wait time* $t_{1,l} - t_{1,i}$; how long contributor i had to wait for the last contributor l to arrive, and *departure wait time* $t_{4,i} - t_{4,f}$; elapsed time since the first contributor f departed from the allreduce wrapper, until contributor i departed.

4 EventSpace

The architecture of the EventSpace system is given in figure 2. An application is instrumented by inserting *event collectors* into its communication paths. Each event collector record data about communication operations into a *trace tuple* and stores it in an *event space* consisting of PastSet bounded buffers. Different *views* of the communication behavior can be provided by extracting and combining trace tuples provided by different event collectors. Consumers use an *event scope*, an aggregation/gather network, to do this.

4.1 Design

Runtime monitoring tools need to provide the data necessary for analysis at high performance and without perturbing the monitored application. We describe the design choices made in EventSpace to achieve these goals.

Configurability and extensibility. Being a research tool, EventSpace is designed to be extensible and flexible in order to experiment with different approaches for tuning the trade-off between monitoring performance and perturbation. It is also possible to extend EventSpace by adding other event collectors, and event scopes.

Separation of functional concerns. The tasks of collecting, storing, analyzing, gathering and presenting data are clearly separated in order to allow each part to be implemented and tuned separately. Data is collected by communication system wrappers, and stored using the PastSet structured shared memory system. EventSpace provides mechanisms for distributed analysis and fast collective operations for gathering data from compute hosts to a front-end host, which is responsible for presentation or further analysis of the data.

Low overhead data collection. We expect the number of trace tuple writes to be much larger than the number of reads; hence an event collector only record the minimal information for each communication operation and stores it in binary format in memory using native byte ordering. For heterogeneous environments, the tuple content can be parsed to a common format when it is read. Due to separation of concerns all communication paths are instrumented, and data is recorded for each operation, since event collectors do not know what data monitors need and when they need it.

Temporal trace storage. The challenge for large scale message tracing is the amount of data produced [28]. EventSpace provides temporal storage requiring only a few megabytes of memory (each trace tuple is 28 bytes allowing about 37 450 tuples to be stored in one megabyte of memory). The event scopes used by monitors need to have sufficient performance to read the trace tuples before they

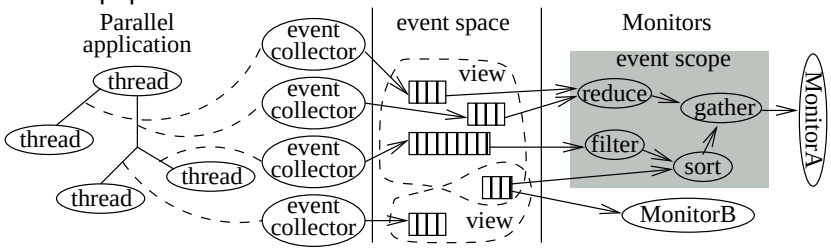


Figure 2. EventSpace architecture.

are discarded. Presently, the amount of tracing can not be dynamically be adjusted as in other monitoring systems (for example NetLogger [25]).

Distributed data analysis. Monitors use event scopes to analyze and gather data from compute hosts. The performance and perturbation of an event scope can be tuned by configuring the collective communication structures used by the event scope, and the mapping of these to the clusters. Data can be reduced or filtered close to the source, to avoid sending all data over a shared resource such as Ethernet, or a slow Internet link. Also some data preprocessing can be done on the compute clusters, thereby reducing the load on the front-end host.

Monitors using distributed analysis can be implemented either as a process on a front-end using an event scope or as a distributed application with several analysis threads. Each analysis thread can read and analyze trace tuples, and stores the result in a PastSet buffer. The results can then be gathered to a front-end for presentation.

Coscheduling. During a synchronizing collective operation all threads on a host must wait for data from other hosts. During the wait-time it is possible to run analysis threads if they are coscheduled with computation, and PATHS/PastSet communication threads. Coscheduling is possible since computation threads are blocked inside the communication system during collective operations and analysis threads also use the communication system for reading trace tuples. Hence, the release order of the different threads can be controlled by releasing all communication threads before computation threads, and finally any blocked analysis threads. No changes to the operating system scheduler are required.

On demand data gathering. Analyzing and gathering performance data comes at a cost. *Computation* is needed for the analysis, *communication* for moving data between hosts, and *storage* for intermediate results. Often these activities use the same resources as the monitored application. Pulling is used by monitors such that shared resources are not used until the data is needed.

Separation of performance concerns. Different parts of the monitoring system have different performance requirements. Event collectors run at the rate the application uses a collective operation. Some analysis threads must also run at this rate, but some lag is allowed due to the trace buffers. With distributed analysis, it is not necessary to gather all intermediate results; hence the gather rate can be lower than the event collecting rate. Further performance relaxation is allowed for presentation to users. The separation of performance concerns also makes it easier to trade-off between monitoring performance and perturbation.

4.2 Implementation

Event Space. An event space is implemented using PastSet buffers. Each trace buffer can have a different size and lifetime. The oldest tuple is automatically discarded when the number of tuples is above a specified threshold.

Event Collectors. An event collector writes a trace tuple to a trace buffer using the blocking PastSet write operation. During the write, the traced communication operation is blocked. As a result it is important to keep the introduced overhead low. The write consist of a mutex lock, a memory copy of 28 bytes, and a mutex unblock (a read is similar). The recorded information is: event collector identifier, PastSet operation type, tuple sequence number, return value, and the start and completion timestamps.

Event scopes. An event scope for a specific monitor is implemented as a spanning tree with PATHS wrappers for: (i) storage, (ii) data manipulation including aggregation, filtering and conversion, (iii) data gathering and scattering, and (iv) inter-host communication. Storage wrappers provide access to PastSet buffers, while inter-host communication wrappers allow setting properties of TCP/IP connections such as socket buffer size. Only the data manipulation wrappers are aware of tuple format and content.

Gather wrappers read tuples from several PastSet buffers, concatenate these and returns one large tuple. Scatter divides and writes a tuple into several PastSet buffers. The gathering and scattering is done in the context of the calling thread. It is also possible to specify that a given number of helper threads should be started for the wrapper. The helper threads allow parallel reads and writes on remote PastSet buffers.

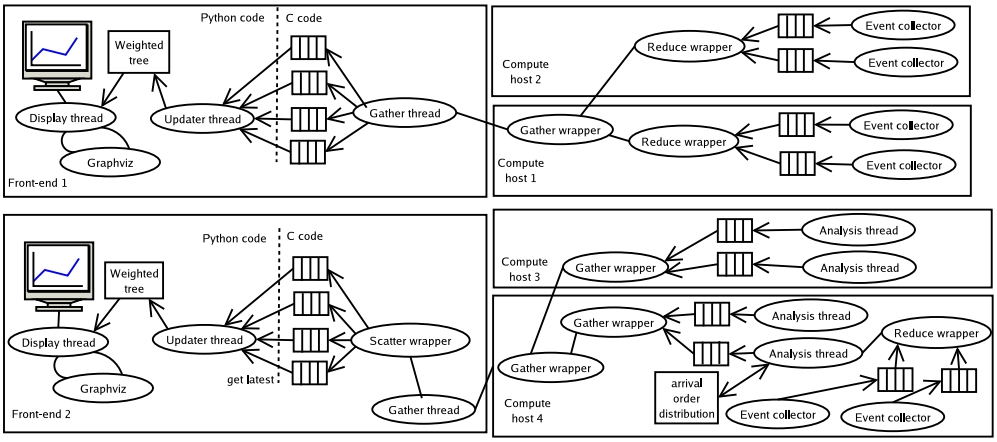


Figure 3. Load-balance monitor with a single event scope (top), and with distributed analysis (bottom).

4.3 Monitors

Load balance monitor. The load balance monitor is used to find load balance problems, which can be caused by workload imbalance, differences in point-to-point communication latency, or the mapping of a spanning tree to clusters. Two implementations are used. The first has a single event scope (figure 3). A gather thread uses the event scope to pull trace tuples produced by the event collectors on each compute host. A reduce wrapper is used to find the tuple with the largest down timestamp. All reduced tuples are then gathered to the front-end where they are scattered to PastSet buffers (one per allreduce wrapper). The tuples contain the number of last arrivals for each participant, and are read by a thread which applies updates to a weighted tree with the number of last arrivals for each participant. This tree is used to generate visualizations.

Distributed analysis reduces communication cost by increasing computation cost, but also complicates the monitor (figure 3). Each host has one analysis thread that counts the last arrivals for each participant by reading and reducing trace tuples as described above. After each read an *intermediate result* tuple is written to a PastSet buffer, containing the number of last arrivals for each participant. The gather thread gathers all intermediate result tuples from the compute hosts and scatters these to the local PastSet buffers. In the visualization we are only interested in the newest state of the system. Hence, not all intermediate result tuples need to be gathered since the arrival order state is maintained by the analysis threads.

Statistics monitor. The statistics monitor (statsm) is used to find paths with similar behavior and to detect hotspots. Computation is offloaded from the front-end by having on each compute host one or more analysis threads computing all statistics for the spanning tree wrappers on the host (figure 4). Our analysis assumes that all trace tuples are read before being discarded.

For each PATHS wrapper, statsm computes mean, minimum, maximum, standard deviation and median (using the sliding window median implementations from NWS [32] with window size set to 100) for the up, down and total latencies. For each wrapper, the results are stored in three 24 byte result tuples and written to three PastSet buffers. In addition, for allreduce wrappers similar results tuples are written for each arrival and departure order wait time. Also, for allreduce wrappers per thread arrival and departure wait time means are computed and stored in a PastSet buffer.

Two gather threads are used. The first gathers all up and down latencies in addition to the arrival and departure wait times. The second gathers per thread statistics (these are not always needed). Results are stored in two buffers at the front-end. These are used by an updater thread that maintains an analysis tree structure with statistics for each wrapper. The analysis tree is used by visualization threads.

5 Methodology

Two micro-benchmarks are monitored. In *Gsum* threads alternate between using two identical allreduce trees to compute a global sum. Gsum is run for 20 000 iterations using 8 byte messages (most scientific applications use small messages in allreduce [29]). *Compute-gsum* alternates between computing (integer sort) and calling allreduce. The benchmark can easily be perturbed since delaying one thread causes all others to wait for it [21]. Compute-gsum is run for either 10 000 or 20 000 iterations, and is tuned to spend 50% of its execution time computing and 50% in allreduce. Both have one computation thread per CPU. Each experiment is repeated at least three times and

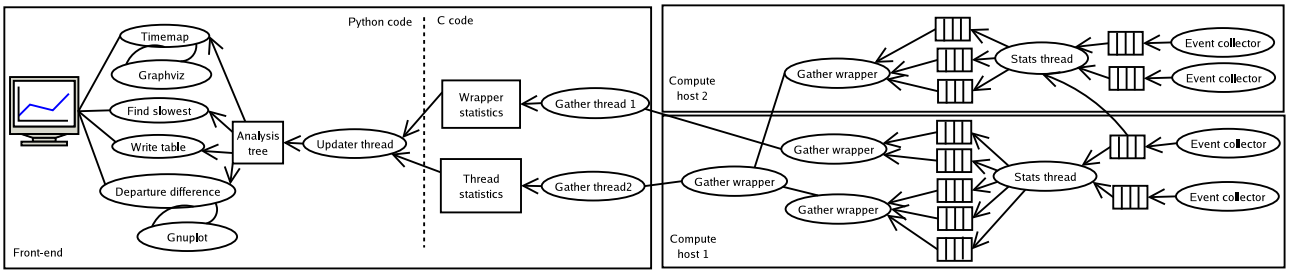


Figure 4. Statistics monitor threads and gather tree.

execution time averages are used to compute monitor overhead. Standard deviation is low (less than 1% of mean). To ensure fairness and experiment repeatability, all event scopes were set up and analysis threads were started before the monitored application.

Four clusters are used: *Copper* 18 dual-CPU Pentium II 300 MHz, 256 MB RAM, *Lead* 10 single-CPU Mobile Pentium III 900 MHz, 1024 MB RAM, *Tin* 51 single-CPU Pentium 4 Hyper-threaded 3.2 GHz, 2 GB RAM, *Iron* 39 singe-CPU Pentium 4 Hyper-threaded 3.2 GHz, 2 GB RAM with EM64T extension.

Copper and Lead share a two-way Pentium II 300 MHz with 256 MB RAM which is used as a gateway and which all communication to/from the cluster goes through. For Tin and Iron one host similar to the compute-hosts is used as gateway. A host outside the clusters, a Pentium 4 1.8 GHz with 2 GB RAM, is used as monitor front-end.

The Tin and Iron clusters have Gigabit Ethernet, while Copper, Lead, and all inter-cluster communication use 100 Mbit Ethernet. The operating system on all clusters is Linux, with the LinuxThreads Pthread library. Iron runs 32-bit code. Hyper-threading was enabled for Tin and Iron. On all TCP/IP connections the Nagle algorithm was disabled and default socket sizes were used.

We emulate WAN links between our clusters using the Longcut WAN emulator [7]. The design of Longcut is similar to the Panda WAN emulator [15]. Tin and Iron are each split into three sub-clusters. For each sub-cluster we select one host to act as a gateway. All communication to the sub-cluster is routed through its gateway, which adds delays to the routed messages to simulate the higher latency and lower bandwidth of a WAN TCP/IP connection. The emulator is implemented using PATHS wrappers.

To calculate the delay added to a message of a given size, we use a latency and bandwidth trace collected by running an instrumented communication intensive application on hosts in Tromsø, Trondheim, Odense and Aalborg. The largest latency is between Tromsø and Aalborg, and is about 36 milliseconds ([7] has additional details about the topology). The sub-clusters are assigned to these sites with two sub-clusters in Tromsø and Odense.

For each cluster we choose an allreduce spanning tree with, to our knowledge, the best performance. For Tin, Iron and Copper this is a hierarchy aware (as in [24, 17]), 8-way spanning tree, while for Lead it is a flat tree. For the LAN multi-clusters the cluster spanning trees are connected by adding an inter-cluster allreduce. For WAN multi-clusters the inter-cluster allreduce is replaced by an all-to-all for improved performance (as in MagPIe [16]). The average time per allreduce for the different topologies is about 0.5 ms for Tin with 32 hosts, 0.6 ms for Tin with 49 hosts, 1 ms for a LAN multi-clusters and 65 ms for a WAN multi-cluster (both multi-clusters with 43 Tin hosts and 39 Iron hosts).

6 Experiments

6.1 Data Collection

The overhead added to a PastSet operation by a single event collector is low (1.1 μs on a 3.2 GHz Pentium 4), compared to the hundreds of microseconds per collective operation. Thus, for the gsum and gsum-noise experiments presented below, the overhead due to event collectors range from 0–2%.

The storage requirement for temporal traces is small. For our 8-way allreduce, the hosts with most event collectors (9) stores 252 bytes per call. We use one megabyte memory for trace tuples and one megabyte for intermediate results. Thus, trace buffer size is set to 3750 tuples, and the intermediate result buffers have size set to 5000 tuples.

6.2 Event Scope

To experiment with the performance, perturbation and tuning of an event scope, we instrumented both allreduce trees used by gsum with event collectors, but only monitored one. The allreduce tree for 49 *Tin* hosts has 241 event collectors, but only data from 57 are needed to compute the arrival order at each allreduce wrapper. These are on 8 hosts, and due to the reduce wrapper only 28 bytes need to be gathered from each host. For a single cluster,

Event Scope	Overhead
Event collectors	none–1%
32 Tins, sequential	tuples discarded
32 Tins, parallel	0.4%
LAN multi-cluster, seq.	tuples discarded
LAN multi-cluster, par.	none
WAN multi-cluster, seq.	1%

Table 1. Load balance monitor with single event scope.

Event Scope	Overhead	Gather rate
49 Tins, sequential (gsum)	2%	51%
49 Tins, parallel (gsum)	2%	99%
49 Tins, sequential	1%	65%
49 Tins, parallel	1%	99%
LAN multi-cluster, seq.	none	45%
LAN multi-cluster, par.	3%	100%
WAN multi-cluster, seq.	1%	94%
WAN multi-cluster, par.	3%	100%

Table 2. Load balance monitor with distributed analysis.

the event scope has only one gather wrapper which is run on the cluster gateway. For multi-clusters the event scopes have a gather wrapper on each cluster gateway and a gather wrapper on the monitor front-end gathering from these.

Gsum. Adding event collectors to a 49 *Tin* spanning tree does not introduce a measurable overhead (monitored mean is within one standard deviation of un-monitored mean). Neither does the load balance monitor. To ensure that all trace tuples are read before being discarded, helper threads must be added to the gather wrappers such that data is gathered in parallel. LAN and WAN multi-clusters have similar results.

Compute-gsum. The largest monitoring overhead was for a multi-cluster with emulated WAN links with 49 *Tin*, 18 *Copper* and 10 *Lead* hosts (table 1). However, the overhead is caused by the WAN emulator becoming inaccurate when there are many emulated connections. As for gsum, sequential gathering has often not sufficient performance.

Scalability. For the event scope achieving sufficient performance is harder than keeping the overhead low. The event scope need to be hierarchy aware and do all intra-host reduces before inter-host gathers, and intra-cluster gathers before inter-cluster gathers. Further reconfiguration by for example moving gather wrappers to unused cluster hosts does not improve performance. Also, for the cluster sizes we had available a flat gather tree had sufficient performance. For larger clusters additional levels may be necessary.

Increasing the number of hosts by connecting clusters with LANs or WANs often lowers the performance requirements for the monitor, since the performance of the monitored operation decreases. Also, the event scopes used by monitors such as load balance scale better than allreduce trees, since data is not needed from all hosts.

The higher WAN latency is usually tolerated since the monitored operation is latency bound, and the messages sent by the event scope are small (a few hundred bytes) making them also latency bound. We believe most WAN links have enough bandwidth for concurrent transfers of application and monitor data.

The monitoring scales well with number of monitored spanning trees. Monitoring both spanning trees in gsum and gsum-compute does not increase monitoring overhead or reduce monitoring performance. Similarly modifying gsum to use four spanning trees and monitoring all trees did not increase overhead or reduce performance.

6.3 Distributed Analysis

Load balance monitor. Distributed analysis uses more resources than the single event scope. For each host with allreduce wrappers, 352 bytes are gathered (compared to 224). Also, there is additional computation cost for running the analysis threads, and storage must be allocated for intermediate results. Using distributed analysis increases monitoring overhead from none to about 2% for gsum on a single cluster (table 2). For compute-gsum the monitoring overhead has not changed.

Monitoring cost can be reduced since it is not necessary to gather all intermediate results to the front-end. Hence, the overhead on a LAN multi-cluster can be reduced from 3% to none, by removing the helper threads in all gather wrappers (parallel vs. sequential in table 2). The performance difference between sequential and parallel gather is smallest for the WAN multi-cluster, and largest for the LAN multi-cluster.

6.3.1 Statistics monitor

Gsum. The statistics monitor is a computation and communication intensive monitor; the analysis threads read data from all trace buffers on the host. Some are also read twice; when computing statistics for the wrapper before and after the associated event collector. Also, to compute TCP/IP latencies a trace tuple must be read from another host.

Initially we have one analysis thread per host. Running distributed analysis on a 32 *Tin* host spanning tree, has 9% monitoring overhead. We tried different approaches for reducing the overhead. Removing all statistics computation (but still reading trace tuples) did not reduce the overhead,

Event Scope	Overhead	Wrapper	Thread
Event collectors	none–1%	-	-
Analysis threads	5–9%	-	-
with coscheduling 1	3%	-	-
with coscheduling 2	1%	-	-
32 Tins, sequential	2%	50%	69%
32 Tins, parallel	2%	77%	99%
LAN multi, seq.	see text	43%	68%
LAN multi, par.	+1%	100%	100%
WAN multi, seq.	none	100%	100%

Table 3. Statsm overhead and gather rates.

showing that the slowdown is not caused by computation. Similarly, removing the read and computation of statistics for allreduce wrappers did not reduce the overhead. Thus the problem was not caused by synchronization in the many buffer reads. Removing statistics computation for TCP/IP connections reduced the overhead to 4%, showing that the slowdown was caused by reads on trace buffers on other hosts.

For TCP/IP connections we can choose whether statistics should be computed at the source or destination (the direction of a path is from the thread to a PastSet buffer). Moving the computation from the source to destination host reduced the overhead to 5%. However, the analysis thread was not able to read all trace tuples before they were discarded (since it reads from 8 hosts sequentially). Running two analysis threads on each host allowed reading all tuples, but increased the overhead to 6%.

Finally, we used two coscheduling strategies: (i) analysis threads are blocked until all participating threads have contributed and a message is sent to the next-level host, and (ii) analysis threads are blocked until all participating threads are unblocked. The first strategy tries to do the analysis while the host is idle waiting for the broadcasted reduced value. The second makes sure the broadcast is done before unblocking analysis threads. The first strategy reduced the overhead to 3%, while the second reduced it to 1%. For the remaining experiments the second coscheduling strategy is used.

Adding gathering increased the overhead to 2%. There was no difference in overhead when gather wrappers had helper threads, but with the latter more intermediate results could be gathered (table 3).

The allreduce spanning trees for a LAN multi-cluster with 43 *Tin* hosts and 39 *Iron* hosts had about 20% slower inter-cluster communication than expected. We were not able to reconfigure or remap the spanning tree to remove the problem. However, when data is gathered from the cluster, allreduce operation time *decreases* with up to 18%. Thus we cannot measure the gather overhead for the multi-cluster topology. But we can compare the performance of a gather tree with sequential and parallel gathering. The latter improved wrapper-, and per thread statistics gather rate, but increased monitoring overhead with 1% (table 3).

The larger latency of emulated WAN links hides the performance problem described above. With WAN links, analysis threads introduce a 1% overhead, but data gathering can be done without helper threads, without increasing the overhead, and with sufficient performance to gather all intermediate results.

Compute-gsum. For compute-gsum the execution time variation is larger than for gsum (about 2% of mean), hence we could not see any monitoring overhead. Also, the gather rate is better. Both are probably due to less communication, since compute-gsum has one less allreduce per iteration.

Scalability. Analysis thread performance is independent of cluster size, since each only monitors a subtree. However, the subtree is dependent on the spanning tree shape.

Gather scalability depends on how analysis threads are mapped to the cluster. For example in our initial configuration all hosts had analysis threads which produced intermediate results that had to be gathered, while the final configuration only had analysis threads on the hosts with allreduce wrappers.

Data gathering for multi-cluster with WAN links has better performance, relative to allreduce performance, than for a single-cluster. This could be due to the small cluster sizes used. The largest cluster had only 12 hosts, requiring only 4400 bytes to be sent over a WAN link. For larger clusters the message size would increase, probably decreasing the gather rate.

Monitoring both 32 *Tin* host allreduce spanning trees in gsum, increased the analysis thread overhead to 5%. We were not able to reconfigure the event scope or coschedule the monitoring to reduce it. The overhead is caused by increased communication activity in the monitor. Adding data gathering does not increase the overhead. Neither does increasing the number of allreduce trees to four, since the communication frequency does not increase neither for the benchmark nor the analysis threads. We have similar results for LAN multi-clusters. However, with emulated WAN links monitoring both allreduce trees does not increase the overhead, since the time between each allreduce operation call is larger (due to WAN latency), hence monitoring activity can be scheduled to run during the WAN communication part of the allreduce operation.

We also modified compute-gsum to alternate between using two and four different spanning trees. Monitoring overhead did not increase, since the number of computations, number of allreduce calls, and allreduce call frequency did not change (we reduced the size of all trace and intermediate PastSet buffers to reflect the fewer allreduce calls per spanning tree).

7 Discussion

The low monitoring overhead and high performance of EventSpace suggest that runtime analysis can be incorporated into a communication system for automatically tuning collective operation performance. In earlier work we have shown how our performance analysis approach can be used to improve allreduce performance up to 49% [9].

It is probably easier to reduce monitoring overhead and improve monitoring performance for real applications than the micro-benchmarks we used, which were designed to stress the monitoring system. We believe the benchmarks are representative for the type of applications interesting to monitor with EventSpace, but real applications will have a more complex interaction between computation, communication and I/O providing further challenges for the analysis and tuning of collective operations.

For the load balance monitor we achieved the same performance and scalability when using an aggregation network than with distributed analysis. Due to the increased complexity of distributed analysis aggregation networks should be used. However, for monitors such as the statistics monitor aggregation networks do not have the necessary performance. Event scope performance was tuned by allocating more resources to the collective operations used to implement them. Changing the spanning tree shape or mapping to clusters did not improve performance.

All our clusters use Ethernet for communication. Faster interconnects, such as Myrinet [6], will improve the performance of collective operations. Thus, application with high enough communication ratio to be interesting to monitor with EventSpace will have a higher communication frequency. This requires the analysis computation to be done in a shorter time, but the event scopes will benefit from the improved communication performance.

Even when using Ethernet, communication latency can be improved by using a lower level protocol than TCP. But, we believe it is easier to add distributed analysis than to implement an event scope with a non-reliable lower level protocol.

We have not measured, or focused, on the time to setup and initialize the event scopes (as in [23, 4]). Currently it can take seconds due to the implementation using Python and XML-RPC. A significant performance improvement is possible by using a more efficient implementation.

Coscheduling the computation threads, communication system threads and the analysis threads did reduce perturbation for one benchmark. We believe further reduction could be achieved by priority scheduling all inter-host communication such that the applications messages always had higher priority than EventSpace messages. This would require a reimplementation of the PATHS/PastSet communication system.

8 Conclusions

We have described the EventSpace system for runtime performance monitoring of collective operations within the communication system. EventSpace allows high-performance message tracing without a large perturbation of the monitored application. By combining distributed analysis with fast collective operations to gather and analyze performance data, temporal storage for only a few megabytes of data is required. Separation of performance concerns allows us to tune the different parts of the system to achieve the required monitoring overhead and performance. Close integration with the communication system allows to coschedule analysis activity with the computation and communication of the monitored application.

We evaluated different monitors for collective operation performance analysis. Our findings were as follows: (i) monitor overhead was low, from none to maximum 3%, (ii) for many monitors it is harder to get sufficient performance than low perturbation, (iii) coscheduling allowed to reduce monitoring overhead from 9% to 1% for one benchmark, (iii) the monitoring has good scalability both with regards to the number of cluster hosts, number of clusters, and number of monitored spanning trees, (iv) high performance monitoring of a WAN multi-cluster is often easier than a single cluster, and (v) performance tuning should be done by allocating more threads to a monitor rather than reconfiguring its communication structure.

9 Future Work

Our long term goal is to build automatically reconfigurable collective operations. We will build and evaluate such a system based on the data provided by the monitoring tools in this paper.

Presently we are porting the NAS parallel benchmarks [20] to PATHS/PastSet to be able to use our tools. EventSpace may also be used to monitor other type of communication systems, for example to optimize global work scheduling in distributed work queues [2]. For data Grid applications large data sets are accessed. For such applications communication performance is important, making them interesting to monitor with EventSpace.

Also, important for the usability of EventSpace are graphical tools to simplify the building and tuning of event scopes.

References

[1] A. C. Arpaci-Dusseau. Implicit coscheduling: coordinated scheduling with implicit information in distributed systems. *ACM Transactions on Computer Systems*, 19(3):283–331, 2001.

[2] R. H. Arpaci-Dusseau. Run-time adaptation in River. *ACM Transactions on Computer Systems*, 21(1):36–86, 2003.

[3] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *Proceedings of the twenty-first Symposium on Principles of database systems*, pages 1–16. ACM Press, 2002.

[4] S. M. Balle, J. Bishop, D. LaFrance-Linden, and H. Rifkin. Ygdrasil: Aggregator network toolkit for the grid. In *Proceedings of PARA'04 - Workshop on State-of-the-Art in Scientific Computing*, volume To appear of *Lecture Notes in Computer Science*. Springer, June 2004.

[5] J. M. Bjørndalen. *Improving the Speedup of Parallel and Distributed Applications on Clusters and Multi-Clusters*. PhD thesis, Department of Computer Science, University of Tromsø, 2003.

[6] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W.-K. Su. Myrinet: A gigabit-per-second local area network. *IEEE Micro*, 15(1):29–36, 1995.

[7] L. A. Bongo. The Longcut wide area network emulator: Design and evaluation. Technical Report 2005-53, Dep.of Computer Science, University of Tromsø, 2005.

[8] L. A. Bongo, O. Anshus, and J. M. Bjørndalen. EventSpace - Exposing and observing communication behavior of parallel cluster applications. In *Euro-Par*, volume 2790 of *Lecture Notes in Computer Science*, pages 47–56. Springer, 2003.

[9] L. A. Bongo, O. Anshus, and J. M. Bjørndalen. Collective communication performance analysis within the communication system. In *Euro-Par*, volume 3149 of *Lecture Notes in Computer Science*, pages 163–172. Springer, August 2004.

[10] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf. Design and evaluation of a wide-area event notification service. *ACM Trans. Comput. Syst.*, 19(3):332–383, 2001.

[11] G. S. Choi, J.-H. Kim, D. Ersoz, A. B. Yoo, and C. R. Das. Coscheduling in clusters: Is it a viable alternative? In *Proceedings of the 2004 ACM/IEEE conference on Supercomputing*. IEEE Computer Society Press, 2004.

[12] C. Cranor, T. Johnson, O. Spataschek, and V. Shkapenyuk. Gigascope: a stream database for network applications. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 647–651. ACM Press, 2003.

[13] P. Dinda, T. Gross, R. Karrer, B. Lowekamp, N. Miller, P. Steenkiste, and D. Sutherland. The architecture of the Remos system. In *Proc. 10th IEEE Symp. on High Performance Distributed Computing*, 2001.

[14] A. Karwande, X. Yuan, and D. K. Lowenthal. CC-MPI: a compiled communication capable MPI prototype for Ethernet switched clusters. In *Proc. of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 95–106. ACM Press, 2003.

[15] T. Kielmann, H. E. Bal, J. Maassen, R. van Nieuwpoort, L. Eyraud, R. Hofman, and K. Verstoep. Programming environments for high-performance grid computing: the Albatross project. *Future Generation Computer Systems*, 18(8):1113–1125, 2002.

[16] T. Kielmann, R. F. H. Hofman, H. E. Bal, A. Plaat, and R. A. F. Bhoedjang. MagPIe: MPI's collective communication operations for clustered wide area systems. In *Proceedings of the seventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 131–140. ACM Press, 1999.

[17] LAM-MPI homepage. http://www.lam-mpi.org/.

[18] M. Massie, B. Chun, and D. E. Culler. The Ganglia distributed monitoring system: Design, implementation, and experience. *Parallel Computing*, 30, 2004.

[19] S. Moore, D.Cronk, K. London, and J.Dongarra. Review of performance analysis tools for MPI parallel programs. In *8th European PVM/MPI Users' Group Meeting, Lecture Notes in Computer Science 2131*. Springer Verlag, 2001.

[20] NASA. NAS Parallel Benchmarks.

[21] F. Petrini, D. J. Kerbyson, and S. Pakin. The case of the missing supercomputer performance: Achieving optimal performance on the 8,192 processors of ASCI Q. In *Proc. of the 2003 ACM/IEEE conference on Supercomputing*, 2003.

[22] R. V. Renesse, K. P. Birman, and W. Vogels. Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining. *ACM Transactions on Computer Systems (TOCS)*, 21(2):164–206, 2003.

[23] P. C. Roth, D. C. Arnold, and B. P. Miller. MRNet: A software-based multicast/reduction network for scalable tools. In *Proceedings of the 2003 ACM/IEEE conference on Supercomputing*. IEEE Computer Society Press, 2003.

[24] S. Sistare, R. vandeVaart, and E. Loh. Optimization of MPI collectives on clusters of large-scale SMP's. In *Proceedings of the 1999 ACM/IEEE conference on Supercomputing*. ACM Press, 1999.

[25] B. Tierney, W. E. Johnston, B. Crowley, G. Hoo, C. Brooks, and D. Gunter. The NetLogger methodology for high performance distributed systems performance analysis. In *Proc. 7th IEEE Symp. On High Performance Distributed Computing*, pages 260–267, 1998.

[26] V. Tipparaju, J. Nieplocha, and D. Panda. Fast collective operations using shared and remote memory access protocols on clusters. In *17th Intl. Parallel and Distributed Processing Symp.*, May 2003.

[27] S. S. Vadhiyar, G. E. Fagg, and J. Dongarra. Automatically tuned collective communications. In *Proceedings of the 2000 ACM/IEEE conference on Supercomputing*, 2000.

[28] J. Vetter. Dynamic statistical profiling of communication activity in distributed applications. In *Proceedings of the 2002 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 240–250. ACM Press, 2002.

[29] J. Vetter and F. Mueller. Communication characteristics of large-scale scientific applications for contemporary cluster architectures. In *16th Intl. Parallel and Distributed Processing Symp.*, May 2002.

[30] J. S. Vetter and A. Yoo. An empirical performance evaluation of scalable scientific applications. In *Proceedings of the 2002 ACM/IEEE conference on Supercomputing*. IEEE Computer Society Press, 2002.

[31] B. Vinter. *PastSet a Structured Distributed Shared Memory System*. PhD thesis, Department of Computer Science, University of Tromsø, 1999.

[32] R. Wolski, N. T. Spring, and J. Hayes. The network weather service: a distributed resource performance forecasting service for metacomputing. *Future Generation Computer Systems*, 15(5–6), 1999.

7.3 Extending Collective Operations with Application Semantics for Improving Multi-cluster Performance

This paper was published in the *Proceedings of HeteroPar 2004* [46].

Extending Collective Operations With Application Semantics for Improving Multi-cluster Performance

Lars Ailo Bongo, Otto Anshus, John Markus Bjørndalen and Tore Larsen
Department of Computer Science, University of Tromsø, Norway
Email: {larsab, otto, johnm, tore}@cs.uit.no

***Abstract*— We identify two ways of increasing the performance of allreduce-style of collective operations in a multi-cluster with large WAN latencies: (i) hiding latency in system noise, and (ii) *conditional-allreduce* where knowledge about the application is used to reduce the number of WAN messages. In our multi-cluster, system noise was not large enough to hide the WAN latency. But, the latency could be hidden using conditional-allreduce, since on many iterations only cluster-local values were needed, and many of the values needed from other clusters were prefetched. A speedup of 2.4 was achieved for a microbenchmark. Prefetching introduced a small overhead in the cluster with the slowest hosts.**

I. Introduction

Computational Grids is an emerging platform for computational science [1]. In a grid, multiple computers and clusters are connected using wide-area networks (WAN). Ideally, applications developed for more tightly connected platforms (e.g. SMPs, clusters) should run effectively without modifications on grids. However, for many applications, modifications are required to tolerate the higher latencies and lower bandwidths of WAN links [2].

Many applications are written using a communication library, such as MPI [3], which provides operations for point-to-point and collective communication. Examples of collective operations are broadcast, reduce, and allreduce. In allreduce, the reduced value is broadcasted to all threads that contributed with a value.

For clusters, the performance of collective operations is an important factor in determining application performance [4]. For grids, we expect collective operation performance to be even more critical. Sensitivity to WAN latency has been shown to be the primary cause for poor collective operation performance on grids [5].

If the provided operations can be made to tolerate WAN latencies and bandwidths, many applications can run on Grids with only minor modifications. In this paper we evaluate two approaches for improving the performance of the allreduce collective operation on Grids: (i) latency hiding, and (ii) extending collective operations with application semantics.

We propose a novel algorithm, conditional-allreduce, where we apply application knowledge to reduce the number of WAN messages exchanged. Many algorithms, such as converging iterative algorithms for linear algebra, use the reduced value only to test whether a particular condition is true. In many cases where multiple clusters communicate over a WAN link, each of the clusters may have enough information locally to determine that the condition is true. In these cases, time-consuming WAN communication can be avoided by returning the result of the cluster-local operation.

Another performance problem is caused by system activities causing 'noise' that takes resources (e.g. CPU) from individual threads and, by implication, delays both the thread itself as well as all other threads participating in a synchronous operation [6], [7]. We evaluate whether some of the WAN latencies can be hidden in the noise.

We describe a micro-benchmark for analyzing noise on clusters, as well as systems for configuring and monitoring the performance of different allreduce algorithms. The performance analysis is based on traces from actual runs on an available multi-cluster.

Our results show that the system noise in our multi-cluster is too low to allow us to hide the WAN latency. Using conditional-allreduce, the WAN latency was avoided for most operations, since these only required values from one cluster. For the remaining operations the required values were often already prefetched. Conditional-allreduce only introduced overhead on the cluster with the slowest hosts. Thus applications using conditional-allreduce can be run on a grid with good performance.

The rest of this paper proceeds as follows. Related work is discussed in section II. Our parallel programming and monitoring systems are described in section III. The design and implementation of conditional-allreduce is described in section IV. Section V describes the clusters and benchmarks used in section VI to compare the performance of conditional-allreduce with other algorithms. Section VII concludes and outlines future work.

II. Related Work

Improving the performance of collective operations is the focus of this paper. However, three additional techniques were applied in [2] to enable applications to tolerate the high latency and low bandwidth associated with WANs. These techniques were (i) distributed work queue implementation, (ii) message combination, and (iii) exploiting asynchronicity in applications.

Typically, collective operations are implemented using a spanning tree. [5] identifies two requirements for collective operations to be wide area optimal: (1) 'every sender-receiver path used by an algorithm contains at most one wide area link', and (2) 'no data item travel multiple times to the same

cluster'. Our work is complementary in that we evaluate how we can avoid sending messages over a WAN, or hide the WAN latency.

For clusters, many implementations apply SMP aware spanning trees [8]–[11]. Many implementations also use fast interconnects [12] or applies special features of the selected interconnect, such as native broadcast in Ethernet [13] or fast remote memory operations [14]. Our implementation is SMP-aware but uses TCP/IP for intra-cluster communication. With faster local interconnects; WAN latencies become even more important. Also, the overhead introduced by the different WAN algorithms measured by us are valid even with faster interconnects.

In [15] it was shown that for barrier operations on an SMP, most of the time was spent waiting for the last thread to arrive. Even for highly balanced applications, noise caused by e.g. system daemons may cause random processes to be delayed [6], [7]. Noise can be reduced by leaving one processor on each SMP idle, by eliminating unnecessary system daemons [7], or by modifying the scheduler to implement co-scheduling [6]. In a Grid, many clusters have either single or dual CPU hosts, and eliminating daemons and modifying the scheduler may be difficult due to administrative issues. Hence, we believe the noise cannot be avoided, and algorithms and systems should be designed to take the noise into account. Conditional-allreduce does so, as fewer threads need to be synchronized, thereby reducing the impact of a delayed thread.

Relaxing the restrictions on a collective operation, as in conditional-allreduce and MagPIe [5], can be regarded as the same approach as using a weaker consistency models to improve the scalability of distributed shared memory systems [16]. Weaker consistency models generally introduce a more complex programming model. However, we believe the relaxation is necessary to get efficient collective communication performance in Grids.

Astrolabe [17] is a recent system for collective (or group) communication in WANs. The primary design goal in Astrolabe was scalability. For collective communication in scientific computing applications, the focus is often on the latencies of operations.

III. Systems

A. PATHS

Usually, MPI implementations only allow the communication structure to be implicitly changed either by using the MPI topology mechanism or by setting attributes of communicators. The PATHS system [18] allows inspecting, configuring and mapping the collective communication structure to the resources in use. PATHS is an extension to the PastSet structured shared memory system [19], where threads communicate by reading and writing tuples to named *elements*.

Using PATHS, we create a sequential spanning tree with all threads participating in the allreduce as leafs (figure 1). For each thread we specify a *path* through the communication system to the root of the tree (the same path is used for reduce and broadcast). On each path, several *wrappers* can be added.

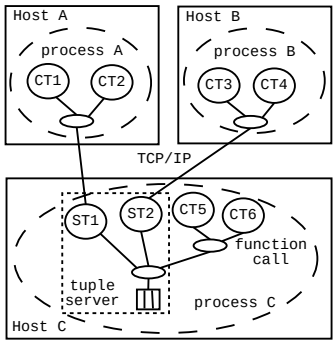


Fig. 1. An application with six computational threads (CT) and two TCP/IP service threads (ST) using a collective operation tree implemented using allreduce wrappers (small ovals). Results are stored in a PastSet element.

Each wrapper has code that is applied as data is moved down the path (reduce) and up the path (broadcast). Wrappers are used to store data in PastSet and to implement communication between cluster hosts. Also, some wrappers, such as allreduce wrappers, join paths and handle the necessary synchronization.

Figure 1 shows the PATHS/PastSet runtime system. It is implemented as a library that is linked with the application. The application is usually multi-threaded. The PATHS server consists of several threads that service remote clients. The service threads are run in the context of the application. Also, PastSet elements are hosted by the PATHS server. Each path has its own TCP/IP connection (thus there are several TCP/IP connections between PATHS servers). Wrappers are run in the context of the calling threads, until a wrapper on another host is called. These wrappers are run in the context of the threads serving the connection.

The allreduce wrappers block all but the latest arriving thread, which is the only thread continuing down the path. The final reduced tuple is stored in the PastSet element before it is broadcasted by awakening blocked threads that return with a copy of the tuple.

B. EventSpace

To collect performance data we use the EventSpace system [20]. The paths in a spanning tree are instrumented by inserting *event collectors*, implemented as PATHS wrappers, before and after each wrapper. For each allreduce operation, each event collector records a timestamp when moving down and up the path. The timestamps are stored in memory and written to trace files when the paths are released. In this paper, analysis is done post-mortem.

Depending on the number of threads and the shape of the tree, there can be many event collectors. For example, for a 30 host, dual CPU cluster, a tree has 148 event collectors collecting 5328 bytes of data for each call (36 bytes per event collector). The overhead of each event collector is low (0.5 μs on a 1.4 GHz Pentium 4) compared to the hundreds of microseconds per collective operation. Most event collectors are not on the slowest path, thus most data collecting is done outside the critical path. Hence, even for the noise-allreduce

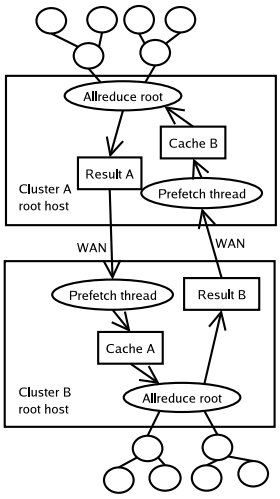


Fig. 2. Conditional-allreduce implementation for two clusters.

microbenchmark the overhead due to data collection is less than 1%.

IV. Conditional-allreduce

Many parallel applications, such as iterative algorithms, use the result of an allreduce operation to check for convergence (one such application is described in section V-A). Hence, the result value is only needed in the last iteration of the algorithm. For all others it is only necessary to reduce enough values until it can be determined whether the convergence condition is true or not. To determine if the condition is true, only values from a subset of the threads may be required. If these threads are on the same cluster, no WAN communication is necessary.

There are some limitations to how the allreduce can be used: (a) the value should only be used for the convergence test and perhaps debugging, (b) the allreduce should not be used as a barrier, and (c) only positive (or only negative) values should be contributed. We believe many applications meet these requirements.

The implementation of conditional-allreduce is based on a wide-area optimal algorithm used in MagPIe [5], but with some differences. As shown in figure 2, we have a sequential allreduce tree on each cluster (as described in section III-A). Between the clusters an all-to-all is implemented using a fully connected graph. An allreduce is done on each cluster and the result is stored in a PastSet element. On each root node there are prefetch threads that pull[1] tuples from the result elements on other clusters, and store these tuples in caches implemented using PastSet elements. The pulled tuples and the local result are reduced, and broadcasted to all threads on the cluster.

To use conditional-allreduce, the application programmer specifies that an allreduce should be conditional, the type of evaluation to use (greater than, less than or equal), and the constant to evaluate against. The operation type (sum, max or min) is already specified for the allreduce. As the PATHS system allows us to set properties of individual nodes in the allreduce tree at initialization time, we have set the condition and constant as properties of the allreduce tree nodes.

The condition check is done after storing the result for the cluster in the local PastSet element. After that, a new check is made every time a tuple is read from a cache. If the condition is found to be true, a broadcast is initiated for the local cluster, and no more caches are accessed.

Since the allreduce operation no longer synchronizes all participants, some clusters (or allreduce trees) may get ahead of others. To reduce the amount of buffering needed for the result values, a sequence number is stored with the result. If allreduce tree A pulls a tuple from allreduce tree B, and the tuple has a larger sequence number than A's result tuple, then B must have found the condition to be true for the iteration A is at (otherwise B would have needed A's result tuple). Hence, the condition must also be true for A. The sequence number allows the memory for the caches on a host to be limited to only one tuple for each remote cluster.

As described in section III-A, there are multiple threads that are synchronized by the allreduce root wrapper. To reduce the introduced overhead, and simplify the implementation, only the thread arriving latest reads tuples from the caches. The read operation is non-blocking, since a tuple from any of the remote clusters can be enough to make the condition true, and we do not know which tuple will arrive first. Between each pull there is a yield call to allow other threads to run.

On each root host there is one prefetch thread per remote cluster. Each thread only fetches the newest tuple from the remote cluster. Hence some tuples are not fetched if the difference between the WAN latency and the time per local allreduce on the remote cluster is large. The read operation blocks on the remote cluster if there are no new result tuples.

V. Methdology

A. Noise-allreduce Microbenchmark

To measure the performance of the different allreduce algorithms, and the system noise in our clusters, we use a benchmark that imitates the behavior of medium grained parallel applications (which are realistic to run on a Grid [2]). Each thread independently sorts a list of integers, a task that is automatically tuned to take 30ms (about the same as the largest WAN latency). The benchmark is run for about 15.000 iterations. It has been shown that system noise resonating with the computation granularity of a synchronous application will cause a substantial performance loss [7]. Thus, for our benchmark the worst kind of noise delays the computation for about 30 ms [7].

We only use 8 byte messages. Most scientific applications have message sizes of less than 256 bytes for most collective operations [21]. Also, we are mostly interested in avoiding the WAN latency.

B. Input Data

The performance of conditional-allreduce depends on the values used in the operation, which depend on the input data.

[1]We can easily implement pushing also (as in MagPIe).

Each noise-allreduce thread reads the values it contributes with from a file. We use five sets of input files. Two sets are the unrealistic *best-case* and *worst-case* allreduce values for conditional-allreduce. The three others are traces of actual values used in Successive Over-Relaxation (SOR), when using different data sets. The data sets have different convergence rates.

SOR is a well known iterative converging linear algebra algorithm that approximates each element in a matrix to its neighbors until the sum of all changes in an iteration converges below a given value. We have traced a Red-Black implementation of SOR. Each worker-process updates all its red points and then exchanges red border point values with its neighbors using point-to-point communication. Then the black points are updated and exchanged. Each process calculates a delta, by summing, for all its matrix elements, the absolute value of the new value subtracted from the old value. At the end of each iteration there is a check for convergence. First, the sum of all deltas is calculated using MPI_Allreduce. Then the resulting global delta is compared to a constant *epsilon*. The algorithm terminates if the global delta is smaller than *epsilon*.

A 1380×1380 matrix was divided among 138 processes. *Epsilon* is 0.01904. The first data set, *frosty*, is from a heat distribution simulation where the top row is set to 27760 degrees Celsius[2], while the remaining elements are set to -273.15 degrees Celsius[3]. SOR converges after 5403 iterations.

The second data set, *tridiagonal*, uses a tridiagonal matrix where all dialog elements, and all elements on the three sub-diagonals and super-diagonals are set to a random value between 0 and 10000. The remaining values are zero. Convergence is after 1737 iterations.

For the third data set, *random*, the matrix elements are initialized with random values between zero and 10.000. The computation converges after 273 iterations.

C. Clusters

The hardware platform comprises six clusters:

- RoadRunner 48 single-CPU Celeron 1700 MHz, 256 MB RAM. Odense, Denmark.
- Dominic 7 dual-CPU Pentium III 733 MHz, 2 GB RAM. Aalborg, Denmark.
- Blade 10 single-CPU Mobile Pentium III 900 MHz, 1024 MB RAM. Tromsø, Norway.
- 2W 18 dual-CPU Pentium II 300 MHz, 256 MB RAM. Tromsø, Norway.
- 4W Eight four-CPU Pentium Pro 166 MHz, 128 MB RAM. Tromsø, Norway.
- 8W Four eight-CPU Pentium Pro 200 MHz, 2 GB RAM. Tromsø, Norway.

The clusters are not directly accessible from the Internet. Communication through and from the Tromsø clusters goes through a two-way Pentium II 300 MHz with 256 MB RAM.

[2]The surface temperature of a blue star.

[3]Zero Kelvin, or absolute zero.

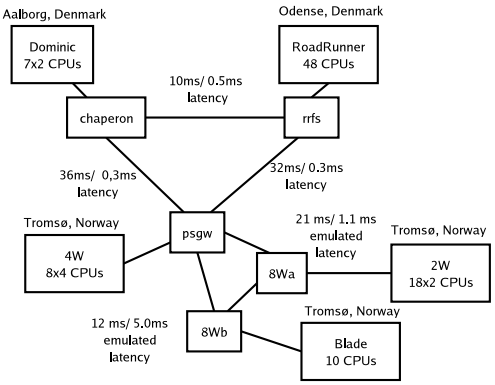


Fig. 3. Clusters, gateway hosts and WAN link emulator hosts of the multi-cluster used in the experiments. For each WAN link the average and standard deviation of the two-way TCP/IP latency is given.

For Roadrunner, a Pentium III 1400 MHz with 1 GB RAM is used as a gateway host. The gateway host for Dominic is a dual-CPU Pentium III 733 MHz with 640 MB RAM. The clusters use TCP/IP over a 100 Mbps Ethernet for intra-cluster communication. Inter-cluster communication uses the Nordic interconnection of national research networks (NORDUnet).

There was no background workload on the cluster hosts. However, there was other traffic on the department networks, and on the Internet. On all TCP/IP connections the Nagel algorithm is disabled to ensure that even small data packets are sent immediately. The operating system on all clusters is Linux.

D. Wide-area Network Emulator

To increase the number of WAN links we emulate WAN links between the Tromsø clusters. The emulator is inspired by the Panda WAN emulator [22]. We use two of the 8W hosts as gateways for Blade and 2W. Thus, a message from a 2W host to a Blade host is first sent to the 2W's gateway, which forwards it to Blade's gateway, which finally forwards it to the Blade host. Figure 3 shows the topology of the multi-cluster.

The emulator is implemented using PATHS wrappers that emulate a WAN link. These wrappers are run on the gateway hosts. For all messages a delay time is calculated by using the latency and bandwidth of the emulated WAN link, and the message length. The latency and bandwidth are read from a file. For each WAN connection we have one trace file for each direction consisting of latency and bandwidth traces.

We have collected the WAN traces using the Unix ping tool. The ping latency is similar to the TCP latency due to the small message size used in the experiments (8 bytes). Also, bandwidth is not measured; instead the maximum bandwidth of the link is used. Bandwidth is not important for the small messages used.

The measured WAN connections were between the University of Tromsø and: (i) Norwegian University of Science and Technology in Trondheim, Norway, and (ii) Finnmark University College in Alta, Norway. The average two-way latencies are given in figure 3.

VI. Experiments

In this section we analyze the performance of different allreduce implementations using the benchmark and clusters described in section V. Also, for each allreduce implementation we measure the noise in the system.

A. Sequential Allreduce

To identify a baseline, we analyze the performance of a sequential multi-cluster allreduce tree implemented as described in section III-A. The algorithm is similar to the algorithms used in LAM-MPI [11] and MPICH [23]. However, our spanning tree is SMP and WAN aware. The noise-allreduce benchmark was run on the five clusters described in section V-C, with the root of the spanning tree on a 4W host. Two of the WAN links were emulated, as described in section V-D. For each sender-receiver path there is one WAN link, but two messages are sent over the link (one for reduce, and one for broadcast). For 15.000 iterations the execution time was 1412 seconds.

As the sequential spanning three synchronizes all threads, one slow cluster may delay all others. By analyzing the message arrival order at the spanning tree root, we find that the two slowest clusters are 2W and Dominic, arriving last 69% and 23% of the times respectively. The many last arrivals for Dominic were expected since the WAN link between Dominic and 4W has the highest latency.

The 2W cluster has a performance problem caused by the interaction between the allreduce spanning tree and the workload. As described in section III-A, the broadcast of a reduced value is implemented by unblocking a set of server threads that return the value to their clients. The broadcast may unblock a worker thread that uses the CPU, causing server threads to wait. Hence, the last message may be sent up to 30 ms later than the first. The spanning tree on the other cluster with 2-way SMPs (Dominic) has a similar, but smaller, problem. For the 2W send-receive paths, 58% of the time spent in an allreduce was as a result of the WAN link, compared to 87–89% for the paths on the other clusters (expect for 4W where the paths do not have a WAN link). This shows that the spanning tree on a cluster may have a significant effect of the multi-cluster allreduce performance. Possibly, a re-mapping or re-implementation may improve the spanning tree performance.

For some RoadRunner hosts we had unexpected performance irregularities, increasing the computation time from 30 ms to 36 ms for most iterations. A similar increase in computation time was observed on other RoadRunner hosts in other experiments. We do not believe the problem is caused by other background workload, nor the spanning tree implementation. Also, the disturbances occur too frequently to be caused by system daemons. However, the increase is overlapped by the larger WAN latencies and the performance problems on 2W, demonstrating that the sequential spanning tree tolerates noisy hosts as long as the noise doesn't occur in a cluster with the largest WAN latency to the root.

For the 15.000 iterations, only in 41 iterations at least one of the threads was delayed for more than 30 ms compared to the average computation time[4]. In 223 iterations at least one thread was more than 10 ms delayed, in 359 iterations some thread was more than 5 ms delayed, and in all iterations at least one thread was 1 ms delayed. Thus the potential benefit of hiding the WAN latency in the system noise is limited.

Earlier we have documented that there are large variations in execution time per allreduce, and where within the communication system time is spent [24]. The multi-cluster spanning tree exhibits even larger variations. However, the standard deviation for the WAN links is low (figure 3). Thus, for our system, variations in the communication systems have larger impacts than variations in computation time.

To conclude, for a sequential spanning tree the WAN latency is the primary cause of poor performance. However, the implementation of the spanning tree on a cluster may also cause performance problems. The potential for latency hiding is small.

B. MagPIe Allreduce

When using the *worst-case* data set for conditional-allreduce, the condition is never true and hence every iteration requires an all-to-all exchange. This behavior is similar to the MagPIe allreduce algorithm [5]. However, due to differences in the underlying systems, the implementation differs[5]. The MagPIe algorithm should improve performance as each allreduce operation introduces just a single one-way latency. As we do not have global clock synchronization, we assume the one-way latency to be half of the measured two-way latency.

For 15 000 iterations, the execution time was 1474 seconds, which is slower than for the sequential configuration. The potential speedup of MagPIe is dependent on the multi-cluster topology, in particular the difference between the largest two-way and one-way WAN latency. For our case, the expected speedup was 1.1[6]. However, when running the benchmark on a multi-cluster with an emulated topology where the largest two-way latency was twice the largest one-way latency and there was 50% communication, we achieved speedups of around 2.0.

In our implementation a potential bottleneck are the pre-fetch threads, as we assume the time to send the read request is overlapped with computation. The performance data confirms this assumption as the largest two-way WAN latency is around 60 ms indicating that the send request latency (30 ms) is overlapped with computation.

To analyze the performance of conditional allreduce, we compare for each cluster-root host, the order, and wait time until tuples where read from the pre-fetch thread caches. Wait times longer than the one-way latency indicate that the

[4]By comparing with the average value, we can ignore the performance faults on RoadRunner.

[5]MagPIe is implemented on top of MPICH.

[6]The largest one-way WAN latency is in the all-to-all graph is 30 ms, and the largest two-way latency for the sequential tree is 36 ms giving a speedup of 1.2. However, only 63% is spent communicating reducing the potential speedup to 1.1.

cluster must wait for another cluster to complete its sequential allreduce. Smaller wait times indicate that tuples where either in the cache or already sent (but not yet arrived).

For all cluster-roots, most last arrivals are either from RoadRunner or from 2W, indicating that these are the slowest clusters. Also, the wait times on 4W, Blade and Dominic are larger than the one-way latency for these two clusters. On 2W and RoadRunner all wait times are smaller than the one-way latency, except for 2W waiting for Roadrunner and vice versa. Hence no single cluster is especially slow.

As for the sequential experiment, the 2W cluster has performance problems caused by the spanning tree. The difference between the first and last send in broadcast is larger, probably due to the increased load due to the pre-fetch threads on the root host. On RoadRunner, some hosts still compute for 36 ms in most iterations.

The MagPIe algorithm allows some of the WAN latency to be hidden in the noise since the allreduce time for the slowest cluster may not include WAN latencies as messages can be exchanged while waiting for the slowest thread. If the probability of two cluster being slowest are equal, the clusters will alternate being slowest. However, due to the performance problems on 2W and RoadRunner, these were slowest for most iterations. Due to the large variations within the communication system, it is difficult to determine whether these actually allowed some of the WAN latency to be hidden.

In conclusion: The potential for speedup was limited due to the multi-cluster topology used, and we were unable to demonstrate significant speedups due to problems with the workload-balancing on RoadRunner and the sequential spanning tree implementation on the 2W cluster.

C. Conditional-allreduce

1) Best-case: For the *best-case* data set, inter-cluster communication is only necessary in the last of the 15 000 iterations. Compared to the sequential spanning tree, the speedup is 2.4. Average time per iteration is 38.6 ms, which is close to the computation time for the slowest thread. The performance improvement is due to all but the latest iterations not needing any results from the other clusters.

There is no problem with the broadcast on the 2W cluster, but some RoadRunner threads still have a computation time of 36 ms for most iterations. Also, the computation time for the 4W root host threads has increased to 34 ms. The other cluster roots are unaffected (these hosts are much faster than the 4W hosts). Due to the performance problems on 4W and RoadRunner, the three other clusters wait 53 and 102 seconds for results from these clusters in the last iteration.

The amount of computation noise is about the same as for the worst-case data set. But the variation of the measured performance within the communication system is lower, since fewer threads are synchronized on each iteration, and there is no broadcast problem on 2W.

To conclude, the best-case data set for conditional-allreduce allows the WAN latency to be completely hidden. Also, the overhead introduced by the prefetch threads is low on fast hosts.

2) Frosty: The frosty heat distribution was simulated three times; hence all threads had to contribute in at least 3 of the 16210 allreduce operations. The average time per operation is comparable to *best-case* (39.6 ms) even if the data set has more operations requiring results from other clusters. As for the *best-case* experiment, some RoadRunner threads compute for 36 ms, while the 4W root host threads compute for 34 ms.

For 4W and Dominic, only four operations required values from other clusters (the spikes at each 5403rd iteration in figure 4). Both clusters waited longest for the results from RoadRunner due to the difference in the computation time between RoadRunner and the other clusters (13 and 26 seconds respectively). For the other clusters, 4W waited between 1 ms (for Dominic) and 21 ms (for 2W), and Dominic waited between 99 ms (for Blade) and 14 seconds (for 4W).

RoadRunner has more threads than 4W, which provides it with more local results to check the condition for. However, 14 operations need remote results due to the input data dependency[7]. For nine of these operations, only one remote result was required to determine the condition to be true. All required values were prefetched, so the wait time was only a few microseconds (figure 4).

2W required values from other clusters for 165 operations. For 161 of these, only prefetched values from Blade were needed, thus the wait time for these operations were only a few microseconds.

On Blade there were 5291 operations that required values from other clusters, due to the cluster having only 10 threads. The number of operations requiring remote cluster values increase as the computation is close to convergence (figure 4). The average wait time ranges from 2.5 ms (for Dominic) to 133 ms (for 4W). However, the median wait times were only a few μs indicating that for most iterations prefetched values could be used.

The results show that, even if there are more operations that require values from other clusters, performance is not degraded compared to the *best-case* experiment as most values are prefetched, resulting in a median wait time of a few microseconds. Furthermore, only values from one or a few clusters are required for most operations that require values from remote clusters.

3) Tridiagonal: Using the *tridiagonal* data set, the average time for the 15635 iterations was 38.6 ms. For 4W and Dominic, only the 9 convergence iterations required values from other clusters. For the other clusters, more remote values where required: 55 for RoadRunner, 254 for 2W, and 6013 for Blade. The wait times are as in the *frosty* data set.

4) Random: For the *random* data set, the average time per iteration was 38.3 ms. The computation converges after 273 iterations and is repeated 55 times. As for the other conditional-allreduce experiments, some threads on RoadRunner compute for 36 ms, and the 4W root threads compute for 34 ms. Figure

[7] 4W has the top row that initially has different values than the other rows.

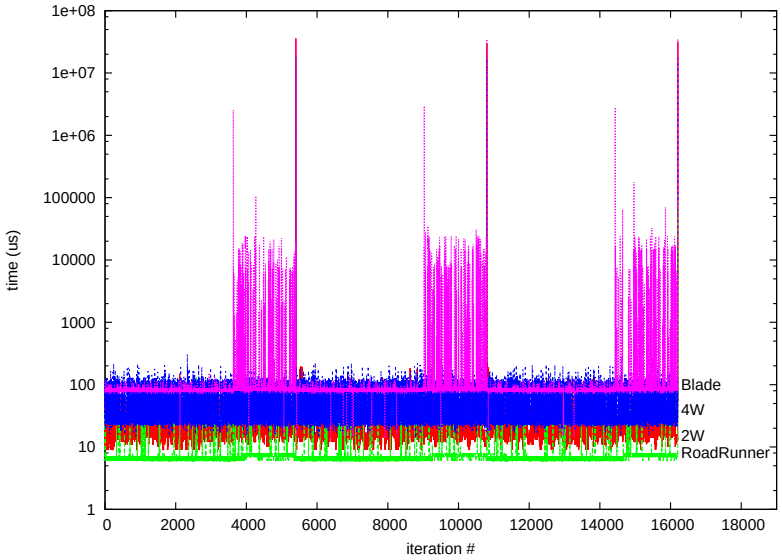


Fig. 4. For each cluster root, the time to determine whether the condition is true using the *frosty* data set. For clarity the graph for Dominic is not shown (it is similar to the 4W graph).

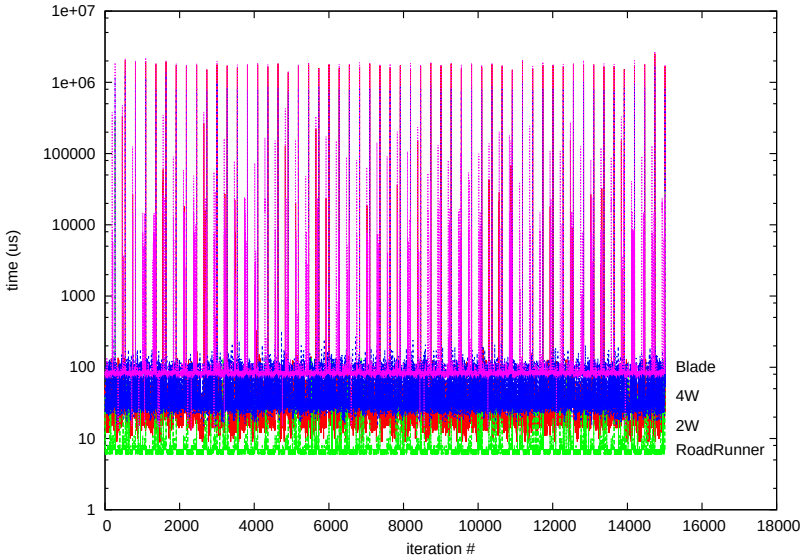


Fig. 5. For each cluster root, the time to determine whether the condition is true using the *random* data set. For clarity the graph for Dominic is not shown.

5 shows the time to determine if the condition is true for all clusters.

Dominic and 4W has fewest (55) operations that require values from other clusters. The average wait times ranged from 0.5 ms (4W from Blade) to 1.8 seconds (Dominic from RoadRunner).

On RoadRunner, 235 operations required remote cluster results. The wait time was low with most operations waiting only a few microseconds. For the 759 cache reads on 4W, the medians were 4–64 μs. But the means were larger for RoadRunner (64 ms) and 2W (21 ms).

2W has 3267 operations that require results from other clusters, of which 112 required values from 4W. The mean wait time for values from 4W was 411 ms (median 205 ms). The median values for the other clusters were lower since prefetched values could be used for most operations.

As for the other data-sets, Blade has many operations requiring results from other clusters (4388). However, for most operations prefetched values could be used.

To conclude, even with a data set that converges after 273 iterations we get similar performance results as for a data set with converge after 5403 iterations. Hence, we believe conditional-allreduce allows the WAN latency to be hidden for many converging iterative algorithms.

VII. Conclusion and Future Work

Collective operations for Grids containing multiple clusters should be designed to tolerate the high latency and low

bandwidth of WANs. We have evaluated two approaches for improving the performance of the allreduce collective operation on Grids of this kind: (i) latency hiding, and (ii) extending collective operations with application semantics.

We have described conditional-allreduce, a novel allreduce algorithm that applies application knowledge to reduce the number of WAN messages exchanged. The performance of conditional-allreduce was compared to other allreduce algorithms by running a benchmark on a real multi-cluster.

We proposed hiding some of the WAN latency in system noise, which delays the arrival of threads at synchronizing collective operations. However, our results demonstrate that the system noise in our multi-cluster is too low to allow a significant part of the WAN latency to be hidden.

For our setup, a wide area optimal allreduce algorithm did not perform significantly better than a sequential allreduce spanning tree. This is due to the multi-cluster topology, workload tuning problems on one cluster, and competition for resources between the communication system and the workload on another cluster.

Using conditional-allreduce, WAN latency was avoided for most operations since these require values from only one cluster. For the remaining operations, only values from a few clusters were needed, and these where often pre-fetched. There was no difference in performance when using a data set from an iterative converging algorithm that converged after 5403 iterations, or a data set from another algorithm which converges after 273 iterations. Conditional-allreduce only introduced overhead on the cluster with the slowest hosts.

Applications using conditional-allreduce can be run on a grid without performance degradation, provided that the point-to-point and other collective operations can tolerate the WAN latency and bandwidth problems. For many applications asynchronous point-to-point communication can be used [2]. We will as future work evaluate algorithms and communication systems for Grids using other types of collective operations with larger messages, such as all-to-all. We believe pre-fetching and replication may improve the performance of these operations. An open question is whether and how the semantics of these operations can be relaxed, or if other programming models may be required for applications using these operations.

ACKNOWLEDGMENT

Thanks to Brian Vinter for providing us access to the clusters in Denmark, and helpful discussions about the experiments. Also thanks to Josva Kleist and Gerd Behrmann for allowing us to use the cluster in Aalborg.

REFERENCES

[1] I. Foster and C. Kesselman, Eds., *The Grid 2: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 2003.

[2] A. Plaat, H. E. Bal, R. F. Hofman, and T. Kielmann, "Sensitivity of parallel applications to large differences in bandwidth and latency in two-layer interconnects," *Future Generation Computer Systems*, vol. 17, no. 6, pp. 769–782, 2001.

[3] "MPI: A Message-Passing Interface Standard," *Message Passing Interface Forum*, Mar. 1994.

[4] J. S. Vetter and A. Yoo, "An empirical performance evaluation of scalable scientific applications," in *Proceedings of the 2002 ACM/IEEE conference on Supercomputing*. IEEE Computer Society Press, 2002.

[5] T. Kielmann, R. F. H. Hofman, H. E. Bal, A. Plaat, and R. A. F. Bhoedjang, "Magpie: Mpi's collective communication operations for clustered wide area systems," in *Proceedings of the seventh ACM SIGPLAN symposium on Principles and practice of parallel programming*. ACM Press, 1999, pp. 131–140.

[6] T. Jones, W. Tuel, L. Brenner, J. Fier, P. Caffrey, S. Dawson, R. Neely, R. Blackmore, B. Maskell, P. Tomlinson, and M. Roberts, "Improving the scalability of parallel jobs by adding parallel awareness to the operating system," in *Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, 2003.

[7] F. Petrini, D. J. Kerbyson, and S. Pakin, "The case of the missing supercomputer performance: Achieving optimal performance on the 8,192 processors of ASCI Q," in *Proc. of the 2003 ACM/IEEE conference on Supercomputing*, 2003.

[8] S. Sistare, R. vandeVaart, and E. Loh, "Optimization of mpi collectives on clusters of large-scale smp's," in *Proceedings of the 1999 ACM/IEEE conference on Supercomputing*. ACM Press, 1999.

[9] H. Tang and T. Yang, "Optimizing threaded mpi execution on smp clusters," in *Proceedings of the 15th international conference on Supercomputing*, 2001.

[10] P. Husbands and J. C. Hoe, "Mpi-start: delivering network performance to numerical applications," in *Proceedings of the 1998 ACM/IEEE conference on Supercomputing*. IEEE Computer Society, 1998, pp. 1–15.

[11] LAM-MPI homepage. http://www.lam-mpi.org/.

[12] J. Liu, B. Chandrasekaran, J. Wu, W. Jiang, S. Kini, W. Yu, D. Buntinas, P. Wyckoff, and D. K. Panda, "Performance comparison of MPI implementations over InfiniBand, Myrinet and Quadrics," in *Proc. of the 2003 ACM/IEEE conference on Supercomputing*, 2003.

[13] A. Karwande, X. Yuan, and D. K. Lowenthal, "CC-MPI: a compiled communication capable MPI prototype for Ethernet switched clusters," in *Proc. of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming*. ACM Press, 2003, pp. 95–106.

[14] V. Tipparaju, J. Nieplocha, and D. Panda, "Fast collective operations using shared and remote memory access protocols on clusters," in *17th Intl. Parallel and Distributed Processing Symp.*, May 2003.

[15] S. Kumar, D. Jiang, R. Chandra, and J. P. Singh, "Evaluating synchronization on shared address space multiprocessors: methodology and performance," in *Proceedings of the 1999 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*. ACM Press, 1999, pp. 23–34.

[16] S. V. Adve and K. Gharachorloo, "Shared memory consistency models: A tutorial," *IEEE Computer*, vol. 29, no. 12, pp. 66–76, 1996.

[17] R. V. Renesse, K. P. Birman, and W. Vogels, "Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining," *ACM Transactions on Computer Systems (TOCS)*, vol. 21, no. 2, pp. 164–206, 2003.

[18] J. M. Bjørndalen, "Improving the speedup of parallel and distributed applications on clusters and multi-clusters," Ph.D. dissertation, Tromsø University, 2003.

[19] B. Vinter, "PastSet a Structured Distributed Shared Memory System," Ph.D. dissertation, Tromsø University, 1999.

[20] L. A. Bongo, O. Anshus, and J. M. Bjørndalen, "EventSpace - Exposing and observing communication behavior of parallel cluster applications," in *Euro-Par*, ser. Lecture Notes in Computer Science, vol. 2790. Springer, 2003, pp. 47–56.

[21] J. Vetter and F. Mueller, "Communication characteristics of large-scale scientific applications for contemporary cluster architectures," in *16th Intl. Parallel and Distributed Processing Symp.*, May 2002.

[22] T. Kielmann, H. E. Bal, J. Maassen, R. van Nieuwpoort, L. Eyraud, R. Hofman, and K. Verstoep, "Programming environments for high-performance grid computing: the albatross project," *Future Generation Computer Systems*, vol. 18, no. 8, pp. 1113–1125, 2002.

[23] W. Gropp, E. Lusk, N. Doss, and A. Skjellum, "A high-performance, portable implementation of the MPI message passing interface standard," *Parallel Computing*, vol. 22, no. 6, pp. 789–828, September 1996.

[24] L. A. Bongo, O. Anshus, and J. M. Bjørndalen, "Collective communication performance analysis within the communication system," 2004, to appear in Proceedings of Euro-Par 2004.

7.4 Using Overdecomposition to Overlap Communication Latencies with Computation and Take Advantage of SMT Processors

This paper was published in the *Proceedings of ICPP Workshops 2006* [48].

Using Overdecomposition to Overlap Communication Latencies with Computation and Take Advantage of SMT Processors

Lars Ailo Bongo[1], Brian Vinter[2], Otto J. Anshus[1],
Tore Larsen[1] and John Markus Bjørndalen[1]
[1)] Department of Computer Science, University of Tromsø, Norway
{larsab, otto, tore, johnm}@cs.uit.no
[2)] DIKU, University of Copenhagen, Denmark
vinter@diku.dk

Abstract

Parallel programs running on clusters are typically decomposed and mapped to run with one thread per processor each working on its disjoint subset of the data. We evaluate performance improvements and limitations for a micro-benchmark and the NAS benchmarks, by using overdecomposition to map multiple threads to each processor to overlap computation with communication. The experiment platform is a cluster with Pentium 4 symmetric multithreading (SMT) processor nodes interconnected through Gigabit Ethernet. Micro-benchmark results demonstrate execution time improvements up to 1.8. However, for the NAS benchmarks overdecomposition and SMT provides only slight performance gains, and sometimes significant performance loss. We evaluated improvement and limitation sensitivity to problem size, communication structure and whether SMT is enabled or not. We found that performance improvements are limited by: applications having communication dependencies that limit thread-level parallelism, increase in cache misses, or increased systems activity. Our study contributes a better understanding of these limitations.

1 Introduction

In this paper we investigate when and how overdecomposition may be applied to improve performance without any changes to source-code for MPI-based [17] parallel scientific applications running on clusters of simultaneous multithreading (SMT) enabled single-processor Pentium 4 nodes interconnected through low-cost Gigabit Ethernet.

As shown in figure 1, scientific parallel applications are typically decomposed such that one processor in the cluster runs one thread for a disjoint subset of the data.

Increasing the decomposition of the data will increase the number of threads and may allow for overlapping computation with communication to improve single-application performance. However, increasing the decomposition will typically also increase the number of messages exchanged and the latencies and other costs associated with those message transfers. Our goal is to identify when and how we may increase the decomposition to achieve the performance benefits of overlapping computation and communication while not incurring communication costs that alleviate the increased performance.

The paper makes three contributions:

- We provide an experimental evaluation of the performance benefits of overdecomposition for parallel application with a wide range of communication characteristics.
- We also provide an experimental evaluation of the benefit of SMT for parallel applications implemented using MPI.
- We provide insight into system software issues that effect overdecomposition improvements by describing and using an analysis methodology that combines message traces, operating system counters and hardware performance counters.

2 Experiment setup

2.1 Hardware platform

All experiments were run on a cluster of 44 nodes interconnected over Gigabit Ethernet. Each node is a single processor system with 2 GB RAM and local disk. The processor used is a 90 nm 3.2 GHz version of the Intel Pentium 4. This is an SMT processor applying the second iteration

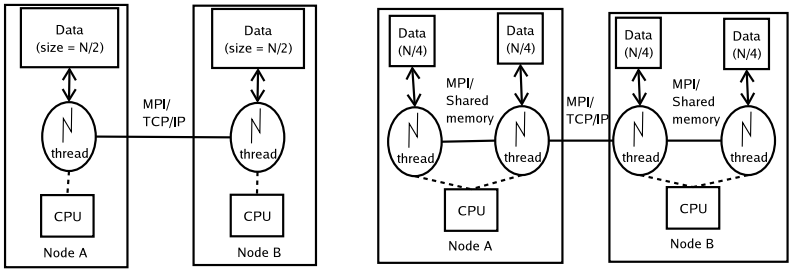


Figure 1. A parallel application without (left), and with overdecomposition (right).

of Intel's Hyperthreading (HT) Technology [1] which offers several improvements over previous implementations in terms of increased or enhanced resources and more dynamic resource allocation.

Each processor has a 12 KB L1 execution trace cache for microoperations, 16 KB 8-way L1 data cache, and a 1 MB 8-way unified L2 cache. Memory access latencies measured using Cachebench [13] are: L1 data: 1.25 ns, L2 unified: 8.78 ns, and main memory: 36.6 ns.

2.2 Software platforms

The cluster nodes run the Linux 2.4.18 uni-processor kernel for the experiments where SMT is disabled, and Linux 2.4.18smp or 2.6.9smp for the SMT experiments. The 2.4 kernel was the first Linux kernel with explicit support for Intel HT Technology. The 2.6 kernel further improve the handling of HT.

The Native POSIX Thread Library (NPTL) [5] was used. NPTL synchronization variables are implemented using the fast user-space locking system call (futex) which handles any non-contended case without requiring a system call.

The communication runtime system used was LAM/MPI version 7.1.1 [11]. LAM/MPI supports hierarchy aware collective operation and shared memory intra-node communication. But when applying overdecomposition multiple processes must be used. For the SOR experiments PastSet [25] was used instead of LAM/MPI. PastSet differs from LAM/MPI in that it supports multi-threading, buffers are explicitly allocated, the communication system has helper threads, and the same protocol is used for all message sizes.

2.3 Benchmarks

The successive over-relaxation (SOR) kernel was chosen under the assumption that the latency of its blocking point-to-point communication operations can easily be overlapped with computation. The benchmark is run for three problem sizes: large, medium and small. For these communication operations contribute to respectively 25%, 50% and 75% of the execution time, when run on 32 nodes with SMT disabled. SOR is compiled with gcc 3.2.3.

Benchmark	Messages	Coll.	Asynch.
BT	Many small	No	Yes
CG	Many small, few large	Manual	Yes
EP	Few small	Yes	No
FT	Few large	Yes	No
IS	Few large	Yes	No
LU	Many small	No	No
MG	Many medium	No	Yes
SP	Many medium	No	Yes

Table 1. NAS benchmark communication behavior. Small message is less than 1 KB, large more than 1 MB. Yes for collectives if execution time is dominated by them. Yes for asynchronous if asynchronous operations are used.

The NAS benchmarks [18] are widely used to evaluate different aspects of parallel architectures. They represent a variety of communication behaviors as shown in table 1. We use the NAS 2.4 MPI implementation with the class B and C problem sizes. The benchmarks were compiled using the Intel Fortran 8.1, and Intel C++ 8.1 compilers.

2.4 Data collection

PAPI [4] is used to access the Intel Pentium performance counters. The Linux kernel is patched with *perfctr* 2.6.9 to provide *virtual performance counters*. These are per-thread counters that increase only when the thread runs user level code. Since this release of *perfctr* lacks SMT support, we have no hardware counter data for the SMT experiments.

Linux maintains process statistics including user level time and system level time per thread, and idle and interrupt handling time per processor context.

For runtime monitoring we use runtime statistical pro-

filing (as described in the next section). For the SOR experiments we use EventSpace [2] to collect message traces for post-mortem analysis. EventSpace allows us to record timestamps inside the communication system, such as before and after writing a message to a buffer.

2.4.1 Overhead

For SOR, the overhead for reading the OS resource counters was less than the variation in execution time.

Message tracing overhead depends on the communication characteristics of the application. For our experiments, the overhead is typically in the 0–4% range. The PAPI overhead due to the in-kernel collection of data is in the 0–2% range. The perturbation introduced by the data collection may influence which mappings shows best performance, favoring mappings with fewer threads per processor. Similarly, execution time improvements due to overdecomposition may also be negatively affected. Still, we believe the data collected demonstrates important trends such as reduction in idle time and increased overhead.

3 Analysis methodology

We characterize each benchmark by: (i) thread-level parallelism (TLP): number of threads ready to run (or running) application computation code, (ii) memory-wait: time the processor is stalled due to cache misses, (iii) system overhead: number of cycles used for running operating system code, (iv) communication overhead: number of cycles for communication activity, (v) network-wait: time waiting due to network latency, and (vi) synchronization-wait: time waiting for data arrival or thread synchronization.

TLP is estimated from the thread count by subtracting the number threads blocked on communication, assuming the remaining threads are compute ready. To characterize the distribution of TLP over a benchmark run we define TLP_N as the ratio of execution time where TLP is larger than or equal to N. Thus, TLP_1 is the percentage of execution time when at least one thread was, our could have been, computing. Without operating system instrumentation we cannot distinguish between these two states.

Memory wait is calculated based on the recorded number of cache misses and the miss penalties determined previously using Cachebench.

System overhead includes operating system activity for inter-node communication, synchronization overhead, context switches, and TLB misses. System time statistics are maintained by Linux.

Communication overhead was typically either to small to be significant, or accounted for elsewhere. The main sources are thread synchronization and memory copying. Both are already accounted for respectively as system overhead and memory wait.

Network wait, the time between sending a request and receiving a response, excluding request processing time on the other node, and *synchronization wait*, the time between a receive operation blocked until a send is initiated, are determined from message traces. Wait time at synchronization points is calculated as described in [3].

Threads:processor	2:1	4:1	8:1	16:1	32:1
Idle	1435	1565	1417	1644	976
System activity	70	130	250	500	1040
Memory wait	226	603	1492	3147	6576
TLB wait	10	20	41	81	165
Unknown	97	163	483	958	1991
in % of exec	1.0%	2.0%	5.7%	10.3%	18.9%

Table 2. Breakdown of SOR overhead increases relative to the one thread per processor mapping. The measurements are for the medium problem size run on 32 nodes with SMT disabled. Unknown is the difference between estimated and measured execution time reduction. All times are in ms.

3.1 TLP and overhead variation

During our analysis we assume that the metrics are similar on all nodes if the benchmark is load balanced. Using the SOR benchmark we measured the variation for the calculated metrics for SOR run on 32 nodes with the medium problem size. The benchmark was run five times.

TLP and data cache miss averages are similar for all nodes for all runs, with standard deviations less than 5% of mean. L1-instruction cache misses and system time have more variation (standard deviation is about 10% of mean).

SOR has non-deterministic waiting pattern where most nodes waits for other nodes, due to a small load imbalance in the communication workload since two of the nodes only have one neighbor. Therefore the variation is large for network wait and synchronization wait (and hence idle time). Which nodes have large synchronization wait change when rerunning an experiment, while network wait is similar for all runs. We believe we still can use the average synchronization wait time for all cluster nodes in the analysis, since the average has less variation.

3.2 Overhead accuracy

The overhead metrics combine data from several sources and abstraction levels. Also, we make several simplifications for system behavior. Here, we evaluate the accuracy

of the estimated overheads. In addition we have verified that TLP results correspond with idle ratio statistics collected by the operating system.

Subtracting all overheads from the reduction in idle time should give the reduction in execution time. The sum of overheads is usually overestimated (table 2). There are two sources of error. First the memory miss penalty is too large. Probably since overlapped cache misses are not taken into account. For SOR increasing the number of threads increases the number of cache misses and hence the miss penalty overestimation. If we assume computation time does not increase, then we can find the overestimation by comparing the sum of memory wait and user level time. The second source of error is system time which is too high for frequently communicating benchmarks.

3.3 Runtime monitor implementation

Our MPI runtime monitor intercepts all communication operations. Statistics about operation times and TLP are updated for each operation. In addition OS statistics and PAPI counters are read at selected collective operations (usually when calling MPI_Init and MPI_Finalize).

Since we do not have any tracing inside the communication system we cannot distinguish between network wait and synchronization wait. TLP counters are in a shared memory map, and these are updated before and after calling a blocking communication operation.

Usually metrics results are presented as statistics over of all nodes over all iterations. But for applications with load balance problems per node statistics are useful. Similarly for applications with several phases, per phase statistics should be used.

4 Performance improvement

Results from running the benchmarks with one thread (or process) per processor with SMT disabled provides insight into which benchmarks have communication wait that can be overlapped with computation. We do similar experiment with SMT enabled, to verify that SMT does not slow down the benchmarks.

We measure overdecomposition execution time improvements with SMT disabled to get insight into the degree of overlapping, and overhead increase we can achieve when threads are not run in parallel on a processor. Then we enable SMT to measure how TLP and the overheads increase when threads can run and compete for resources in parallel.

For all experiments we first analyze the simpler SOR benchmark, before analyzing how the different communication behavior of the NAS benchmarks influence the results. All experiments are repeated ten times and the mean is reported. The standard deviation for the execution times was low if not otherwise noticed, usually less than 2% of mean.

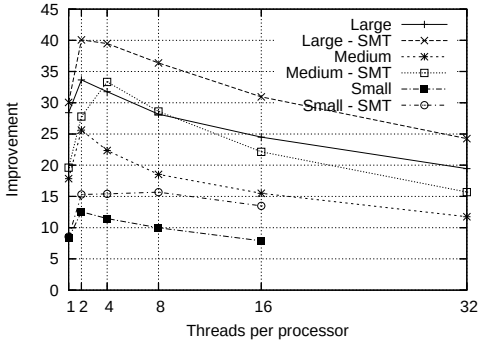


Figure 2. SOR execution time improvements relative to sequential code.

4.1 Baseline

For problem constrained scaling with SMT disabled, execution time is reduced for SOR for all three problem sizes when increasing the number of nodes from 1 to 44. Similarly, execution time is reduced for all NAS benchmarks with both problem sizes when increasing the number of nodes from 1 to 32 or 36 (BT and SP can only be run with a square number of processes). For the remaining experiments we use either 32, 36, or 44 nodes.

The SOR problem sizes were chosen such that 25%, 50% and 75% of the execution time is spent blocked in communication operations. For these respectively 20%, 40% and 55% is due to network latency, the remaining is for synchronization wait.

For most NAS benchmarks wait operations, collective operations or blocking receiving operations contribute significantly to the execution time (table 3).

In conclusion, all benchmarks scale to the cluster size used, and most have operations that can partially or totally be overlapped with computation by using overdecomposition.

4.2 Overdecomposition

SOR was run with 2, 4, 8, 16 and 32 threads per processor (below we use 2:1 when referring to a mapping with two threads per processor core). Execution time improves compared to the one thread per processor mapping for all problem sizes (figure 2). The large problem size has best parallel efficiency, but the relative reduction in execution time is largest for the small problem size (1.5). The best mappings have few threads; 2:1 with SMT disabled. The

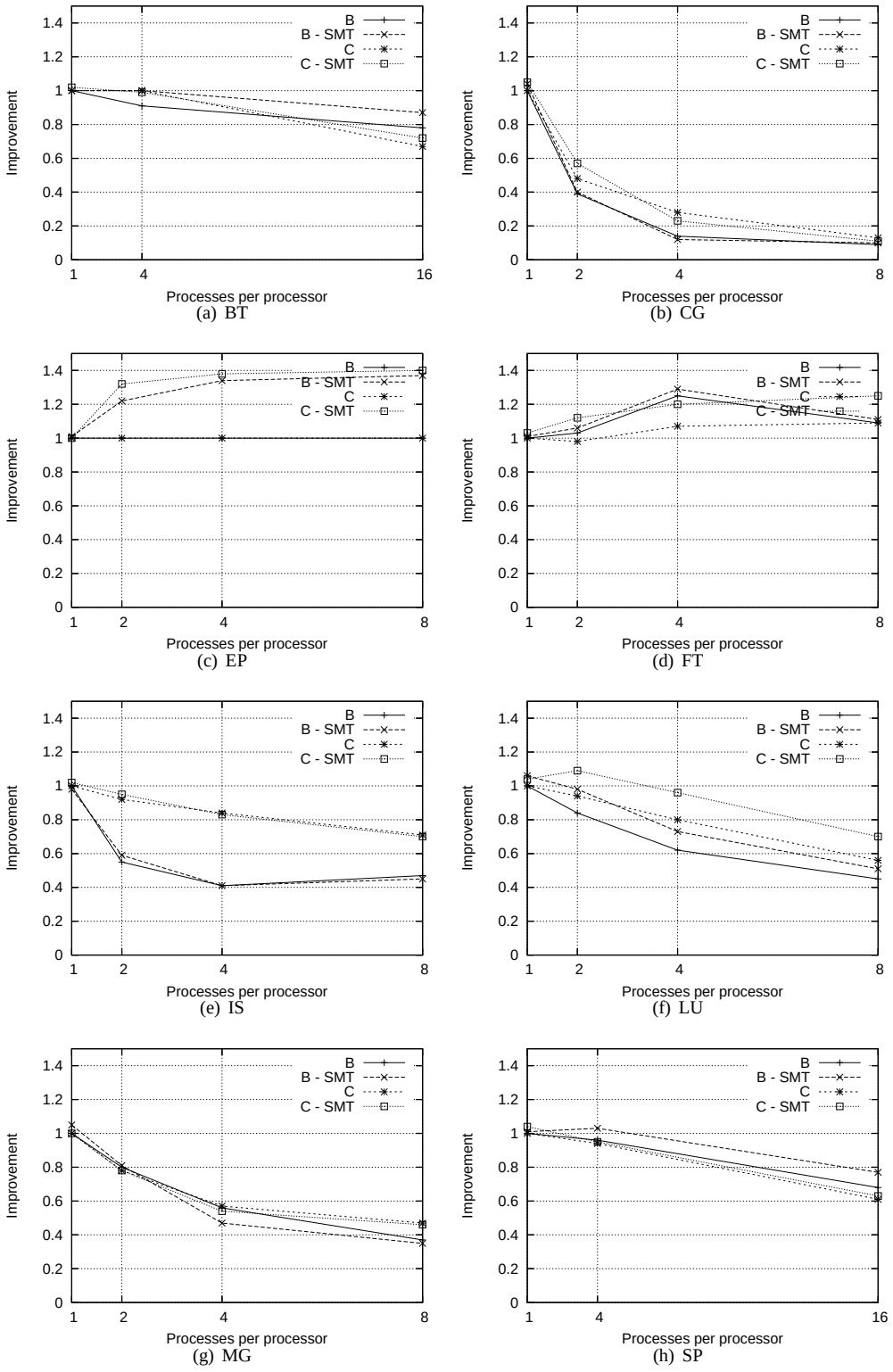


Figure 3. NAS benchmark execution time improvements relative to one thread per processor mapping. Experiments were run on 32 or 36 (BT and SP) nodes.

Benchmark	class B	class C
BT	wait (54%), waitall (10%)	wait (32%), waitall (8%)
CG	wait (53%), send (24%)	send (40%), wait (32%)
EP	none	none
FT	alltoall (62%)	alltoall (53%)
IS	alltoallv (54%), allreduce (29%)	alltoallv (47%), allreduce (20%)
LU	recv (12%), send (10%), wait (7%)	recv (9%), send (8%), wait (4%)
MG	wait (24%), send (16%)	send (18%), wait (8%)
SP	waitall (80%)	waitall (60%)

Table 3. MPI operations contributing to more than 4% of the execution time.

results shows that overdecomposition can improve application performance even on uni-processors.

Figure 3 shows that with SMT disabled overdecomposition improves performance significantly only for FT for both class B and class C. However, performance decreased for for CG, IS and MG.

4.3 Overdecomposition with SMT

Enabling SMT does not change 1:1 mapping execution time, but the improvements with overdecomposition are better. For SOR the best improvement is 1.81 compared to the 1:1 mapping. For the large problem size the parallel efficiency is improved from 30 to 40 (figure 2). The best performance is for mappings with more threads than processor contexts (four threads per 2-way SMT core).

Figure 3 shows that for the NAS benchmarks, enabling SMT gives performance improvement for EP, FT, LU and SP (only for class B). For BT and LU performance was unchanged, while CG and IS got a significant slowdown. For most experiments overdecomposition had best performance with two processes per processor.

The benchmarks for which performance improves have: (i) few and small messages, (ii) few large collective operation messages, and (iii) many small blocking point-to-point messages.

Performance is either not changed or decreased for benchmarks with many asynchronous point-to-point operations with medium or small sized messages. The IS benchmark has two execution phases with almost all communication taking place in the second phase, as well as a global synchronize operation between the phases preventing any overlap.

In conclusion, applying overdecomposition demonstrates a potential performance gain for some application characteristics, but should not be applied indiscriminately as it may result in unchanged or reduced performance for other applications. The mappings with best performance have few threads per processor, but some have multiple threads per processor context. Also, the best performance improvements are for problem sizes where more than 50% of the 1:1 execution time is due to communication.

5 Performance limitations

In this section we analyze how many threads are running at the same time, and which overheads increase most for the different benchmarks. Finally, we measure the effect of synchronization variable implementations, user-level schedulers and operating system kernels.

5.1 Thread level parallelism

When run with SMT disabled, SOR does not have enough TLP_1 to fully utilize the single processor context even with 32 threads per processor. The TLP limitation is not due to system code using the processor, since with more than four threads the idle ratio increases. Rather the limitation is due to data dependencies in the application and scheduling policies in the system software.

Enabling SMT improves TLP_1 for SOR, but still TLP_1 decrease when there are too many threads per processor. Also, when the problem size gets smaller the ratio of execution time where at least two threads are runnable decreases. Often it can be as low as 5%, even for configurations with 32 threads.

With SMT disabled, increasing the number of processes per processor does not always increase TLP_1 for the NAS benchmarks. For EP and MG the processor is saturated, but for the other benchmarks processor utilization is usually less than 76% (table 4).

Enabling SMT may increase TLP_1 with a few percentages. Also, as shown in table 5 TLP_2 is low for BT, CG, LU and SP.

5.2 Overhead increases

For SOR, all overheads increase. The increase in data cache misses is most significant for the medium problem

Benchmark	B	C
BT	44–76%	70–67%
CG	21–7%	27–9%
EP	100-99%	100-99%
FT	32–64%	33–75%
IS	19–32%	31–42%
LU	67–38%	77–56%
MG	20–53%	68–41%
SP	23–35%	47–48%

Table 4. TLP_1 increase when the number of processes per processor is increased from 1 to 8 or 16 (BT and SP). SMT is disabled.

Benchmark	B, 2	C, 2	B, 4	C, 4
BT	12%	15%	9%	6%
CG	8%	3%	8%	3%
EP	99%	99%	97%	99%
FT	35%	64%	20%	39%
LU	5%	7%	4%	2%
MG	40%	39%	39%	37%
SP	8%	21%	6%	16%

Table 5. Maximum TPL_2 and TLP_4 for the class B and class C problem sizes (minimum is always zero). SMT is enabled (for some benchmark these numbers are higher when SMT is disabled).

size, but with the small problem size system activity becomes the most significant overhead. Also, network wait which is the overhead we are trying to overlap, increase when the processor load increase with more threads.

Enabling SMT does not increase per thread user level time or system level time for SOR. Thus, we can assume that cache miss penalties and system activity increase are similar. However, the reduction in idle time is larger, giving a larger reduction in execution time.

Table 6 shows that for most NAS benchmarks either memory wait or system activity dominate the increase in overheads. Usually, the dominating overhead does not depend on problem size, but on the process to processor ratio. With four or less processes per processor, cache miss penalty increase most. But with more processes, system activity increase more. Also, cache misses may not always increase with more processes, but system activity always increases.

Of the cache misses the largest penalty is due to L1-D or L2 caches misses. However, with class B; L1-I and TLB miss penalty may be significant.

For most benchmarks the increase in user and system

Benchmark	Class B	Class C
BT	Memory, system	System, memory
CG	Memory, system	Memory, system
EP	None	None
FT	(System)	(System)
IS	None	None
LU	Memory, system	Memory, system
MG	System	System
SP	Memory, system	Memory, system

Table 6. Significant overheads.

time is similar with and without SMT. But for CG and SP both are lower, and for MG system time increase is lower.

Table 7 summarizes which parts of the platform limits overdecomposition performance for the NAS benchmarks.

5.3 System software

Using oprofile [19] we find that most kernel samples for SOR with 1:1 mapping are for the Ethernet driver, while for 32:1 most are for synchronization and context switches. Since synchronization may cause a context switch, we cannot differentiate between these.

We evaluated system software effect on TLP and the system activity overhead using two synchronization variable implementations, two user-level schedulers, and two operating system kernels. The results are for SOR run with the medium problem size.

We replaced NPTL [5] with LinuxThreads [12], and as expected system overhead increased, due to more system calls for synchronization. However, for small messages sizes TLP improved. LinuxThreads improved TLP_2 two threads were runnable from 2% to 34%. The reduction is caused by difference in scheduling policy. With LinuxThreads, synchronization variable calls are likely to cause a context switch.

We implemented two user level schedulers in the PastSet communication system. The first attempts to reduce cache misses by only allowing one or two threads to run computation code at the same time. The second attempts to better overlap inter-node communication by reordering the computation order of the threads in one node. However, due to TLP limitations most of the time there is only one runnable thread, and hence user-level scheduling will not work.

Replacing the 2.4 SMP Linux kernel with the SMT optimized 2.6 kernel does not significantly improve TLP for SOR. Also, system overhead does not significantly change.

6 Discussion and related work

Overlapping I/O wait time with computation to achieve higher CPU utilization is a well known and widely used

Benchmark	BT	CG	EP	FT	IS	LU	MG	SP
Processor idle	Yes			Yes				Yes
Processor saturated			Yes				Yes	
Lack of TLP	Yes	Yes		Yes		Yes	Yes	
Cache misses	Yes	Yes				Yes		Yes
TLB misses								
System activity	Yes	Yes				Yes	Yes	Yes
Comm. phases					Yes			

Table 7. Overdecomposition performance limitations for the NAS benchmarks.

technique. For parallel applications overdecomposition has been described in text books [7], and has been for load balancing by running more threads on underutilized processors [7, 6], and to mask communication latency in a Grid environment [10]. To our knowledge this is the first study on overdecomposition performance improvements on Ethernet clusters with SMT processors. In [10] experiments were conducted to measure application slowdown when the WAN latency between clusters was increased. Our experiments differ in that we attempt to improve the performance of an applications run on a network with a fixed LAN latency. We have unpublished results showing that overdecomposition improvement becomes better for SOR in a WAN environment.

Early simulator results have shown that SMTs [15] can improve parallel application performance [15, 22]. However, recent studies show that SMT has best performance on the POWER5 [9] when cache performance is at its worst, and SMT is not well suited for floating-point workloads and memory bandwidth bound applications [8]; all typical characteristics of parallel scientific applications. Our results show that only four of the NAS benchmarks had significant increase in memory wait time.

A thorough study of SMT on the HT Technology enabled Pentium 4 processors used in our cluster is [23]. The average multithreaded speedup recorded is 1.20 for multithreaded workloads and 1.24 for parallel workloads running on a single node. The applications that were worst affected by running with SMT enabled were those that had the lowest instructions per cycle ratio. Another study [16] on Intel Xeon, shows speedups ranging from 1.05 to 1.28 for data-parallel numerically intensive benchmarks. Intel Xeon performance improvements for web servers were found to depend on the server design and implementation, and could get worse when enabling SMT due to more synchronization in the operating system kernel [21]. Our results for SMT improvement shows smaller improvements for our message passing parallel applications run on a cluster, than the single node shared memory applications in [23, 16]. We do not experience slowdown when using a SMP kernel rather than an uni-processor kernel.

Proposed system support for SMT includes: (i) new synchronization mechanism that permits cheaper synchronization [24], (ii) compiler optimizations including new approaches for inter-thread data-sharing, application of latency-hiding, and loop distribution [14] (iii) kernel mode behavior [20], and (iv) operating system schedulers [22] attempting to benefit from possible constructive inter-thread behavior.

Our results shows that synchronization contributes significantly to system overhead, which is the overhead that increase mostly when the number of threads increase. In addition to using more efficient hardware mechanisms, synchronization variable improvements should also attempt to improve TLP by minimizing the time between unblocking and running a thread. Due to the TLP limitations kernel mode behavior and operating system schedulers are less important, since there usually are few runnable threads. Similarly compiler optimizations and schedulers designed for minimizing competition for processor resources will probably not improve performance since our benchmarks have low TLP, in addition to being memory intensive and hence have low instructions per cycle.

Alternatives to overdecomposition are to rewrite the application to either use both message passing and shared memory, or to use asynchronous communication operations. Both increase the complexity of the parallel program.

7 Conclusion and future work

We evaluated if parallel application performance can be improved by overdecomposition the data into more pieces than there are processors in order to overlap communication operation latencies with computation and taking advantage of SMT processors.

Microbenchmark results are promising with execution time improvements up to 1.8. However, performance improved for only two NAS benchmark, and decreased for three, showing that improvements are sensitive to applications communication structure, cache miss behavior, the problem size used, and also of the underlying system components. The best results were for applications with few

blocking communication operations, and low cache miss penalty to execution time ratio.

Performance improvements are better when SMT is enabled, and never significantly worse. Hence for Pentium 4 based cluster SMT can be enabled as default. But changes to system software are necessary for fully utilizing SMT enabled processors. Especially intra-node communication must be designed to reduce system calls and cache misses, and synchronization primitives must strive to keep the number of runnable processors high.

As future work, we will investigate if the techniques described in this paper can be used with multiprogramming to overlap globally synchronizing operations with computation, without decreasing single application performance. Also, we intend to investigate system changes tailored for the NAS benchmarks for which performance did not improve.

The monitoring tool used for measuring TLP and overhead increase is available at:

http://www.cs.uit.no/l̃arsab/minim/

Acknowledgment

Thanks to Jon Ivar Kristiansen for help configuring the cluster.

References

[1] D. Boggs, A. Baktha, J. Hawkins, D. T. Marr, J. A. Miller, P. Roussel, R. Singhal, B. Toll, and K. S. Venkatraman. The microarchitecture of the intel pentium 4 processor on 90nm technology. *Intel Technology Journal*, 8, February 2004.

[2] L. A. Bongo, O. Anshus, and J. M. Bjørndalen. EventSpace - Exposing and observing communication behavior of parallel cluster applications. In *Euro-Par*, volume 2790 of *Lecture Notes in Computer Science*, pages 47–56. Springer, 2003.

[3] L. A. Bongo, O. Anshus, and J. M. Bjørndalen. Collective communication performance analysis within the communication system. In *Euro-Par*, volume 3149 of *Lecture Notes in Computer Science*, pages 163–172. Springer, 2004.

[4] S. Browne, J. Dongarra, N. Garner, K. London, and P. Mucci. A scalable cross-platform infrastructure for application performance tuning using hardware counters. In *Proc. of the 2000 ACM/IEEE conference on Supercomputing*, 2000.

[5] U. Drepper and I. Molnar. Native POSIX thread library for linux. http://people.redhat.com/drepper/nptl-design.pdf.

[6] R. J. O. Figueiredo and J. A. B. Fortes. Impact of heterogeneity on dsm performance. In *Proc. of Sixth International Symposium on High-Performance Computer Architecture*, 2000.

[7] G. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon, and D. Walker. *Solving Problems on Concurrent Processors, volume Volume I: General Techniques and Regular Problems*. Prentice Hall, Englewood Cliffs, New Jersey, 1988.

[8] IBM systems & technology group. how DB2 exploits IBM @serverp5 and AIX 5L simultaneous multithreading, October 2004. www-1.ibm.com/servers/eserver/pseries/hardware/whitepapers/p5_db2.pdf.

[9] R. N. Kalla, B. Sinharoy, and J. M. Tendler. Ibm power5 chip: A dual-core multithreaded processor. *IEEE Micro*, 24(2):40–47, 2004.

[10] G. A. Koenig and L. V. Kalé. Using message-driven objects to mask latency in grid computing applications. In *In Proc. of 19th International Parallel and Distributed Processing Symposium*. IEEE Computer Society, 2005.

[11] LAM-MPI homepage. http://www.lam-mpi.org/.

[12] X. Leroy. LinuxThreads. http://pauillac.inria.fr/x̃leroy/linuxthreads/.

[13] LLCbench. http://icl.cs.utk.edu/projects/llcbench/.

[14] J. L. Lo, S. J. Eggers, H. M. Levy, S. S. Parekh, and D. M. Tullsen. Tuning compiler optimizations for simultaneous multithreading. *International Journal of Parallel Programming*, 27(6):477–503, 1999.

[15] J. L. Lo, J. S. Emer, H. M. Levy, R. L. Stamm, D. M. Tullsen, and S. J. Eggers. Converting thread-level parallelism to instruction-level parallelism via simultaneous multithreading. *ACM Trans. Comput. Syst.*, 15(3):322–354, 1997.

[16] D. Marr, F. Binns, D. Hill, G. Hinton, D. Koufaty, J. Miller, and M. Upton. Hyper-threading technology architecture and microarchitecture. *Intel Technology Journal*, February 2002.

[17] MPI: A Message-Passing Interface Standard. *Message Passing Interface Forum*, Mar. 1994.

[18] NASA. NAS parallel benchmarks. http://www.nas.nasa.gov/Software/NPB/.

[19] Oprofile system-wide profiler for linux. http://oprofile.sourceforge.net.

[20] J. A. Redstone, S. J. Eggers, and H. M. Levy. An analysis of operating system behavior on a simultaneous multithreaded architecture. In *ASPLOS-IX: Proceedings of the ninth international conference on Architectural support for programming languages and operating systems*. ACM Press, 2000.

[21] Y. Ruan, V. S. Pai, E. Nahum, and J. M. Tracey. Evaluating the impact of simultaneous multithreading on network servers using real hardware. In *SIGMETRICS '05: Proceedings of the 2005 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*. ACM Press, 2005.

[22] A. Snavely and D. M. Tullsen. Symbiotic jobscheduling for a simultaneous multithreaded processor. In *ASPLOS-IX: Proceedings of the ninth international conference on Architectural support for programming languages and operating systems*. ACM Press, 2000.

[23] N. Tuck and D. M. Tullsen. Initial observations of the simultaneous multithreading pentium 4 processor. In *12th International Conference on Parallel Architectures and Compilation Techniques*. IEEE Computer Society, 2003.

[24] D. M. Tullsen, J. L. Lo, S. J. Eggers, and H. M. Levy. Supporting fine-grained synchronization on a simultaneous multithreading processor. In *Proc. of the Fifth International Symposium on High-Performance Computer Architecture*, 1999.

[25] B. Vinter. *PastSet a Structured Distributed Shared Memory System*. PhD thesis, Department of Computer Science, University of Tromsø, 1999.

7.5 Systems Support for Remote Visualization of Genomics Applications over Wide Area Networks

This paper was published in *Proceedings of GCCB 2006* [49].

Systems Support for Remote Visualization of Genomics Applications over Wide Area Networks

Lars Ailo Bongo[1], Grant Wallace[2], Tore Larsen[1],
Kai Li[2], Olga Troyanskaya[2,3]

[1] Department of Computer Science, University of Tromsø, N-9037 Tromsø, Norway.

[2] Department of Computer Science, Princeton University, Princeton NJ 08544, USA.

3 Lewis-Sigler Institute for Integrative Genomics, Princeton University,

Princeton NJ 08544, USA.

{larsab, tore}@cs.uit.no

{gwallace, li, ogt}@cs.princeton.edu

Abstract. Microarray experiments can provide molecular-level insight into a variety of biological processes, from yeast cell cycle to tumorogenesis. However, analysis of both genomic and protein microarray data requires interactive collaborative investigation by biology and bioinformatics researchers. To assist collaborative analysis, remote collaboration tools for integrative analysis and visualization of microarray data are necessary. Such tools should: (i) provide fast response times when used with visualization-intensive genomics applications over a low-bandwidth wide area network, (ii) eliminate transfer of large and often sensitive datasets, (iii) work with any analysis software, and (iv) be platform-independent. Existing visualization systems do not satisfy all requirements. We have developed a remote visualization system called Varg that extends the platform-independent remote desktop system VNC with a novel global compression method. Our evaluations show that the Varg system can support interactive visualization-intensive genomic applications in a remote environment by reducing bandwidth requirements from 30:1 to 289:1.

Keywords: Remote visualization, genomics collaboration, Rabin fingerprints, compression.

1. Introduction

Interactive analysis by biology and bio[1]informatics researchers is critical in extracting biological information from both genomic [1], [2] and proteomic [3], [4], [5], [6], [7] microarrays. Many supervised and unsupervised microarray analysis techniques have

[1] To be published in Springer-Verlag LNBI 4360.

been developed [8], [9], [10], [11], and the majority of these techniques share a common need for visual, interactive evaluation of results to examine important patterns, explore interesting genes, or consider key predictions and their biological context.

Such data analysis in genomics is a collaborative process. Most genomics studies include multiple researchers, often from different institutions, regions, and countries. Of the 20 most relevant papers returned by BioMed Central with the query "microarray," 14 had authors located at more than one institution, and 7 had authors located on either different continents or cross continents. Such collaboration requires interactive discussion of the data and its analysis, which is difficult to do without sharing a visualization of the results. To make such discussions truly effective, one in fact needs not just static images of expression patterns, but an opportunity to explore the data interactively with collaborators in a seamless manner, independent of the choice of data analysis software, platforms, and of researchers' geographical locations.

We believe that an ideal collaborative, remote visualization system for genomic research should satisfy three requirements. First, synchronized remote visualization should have a fast response time to allow collaborating parties to interact smoothly, even when using visualization-intensive software across a relatively low-bandwidth wide area network (WAN). Second, collaborating parties should not be required to replicate data since microarray datasets can be large, sensitive, proprietary, and potentially protected by patient privacy laws. Third, the system should allow collaborators to use any visualization and data analysis software running on any platform.

Existing visualization systems do not satisfy all three requirements above. Applications with remote visualization capabilities may satisfy the first and the second requirements, but typically not the third as require universal adoption among participating collaborators. Thin-client remote visualization systems, such as VNC [12], Sun Ray [13], THINC [14], Microsoft Remote Desktop [15] and Apple Remote Desktop [16] satisfy only the second requirement because they do not perform intelligent data compression and all except VNC are platform-dependent. Web browser-based remote visualization software can satisfy the third requirement, but not the first two because these systems are not interactive and do not optimize the network bandwidth requirement.

This paper describes the design and implementation of a remote visualization system called Varg that satisfies all three requirements proposed above. To satisfy the first requirement, the Varg system implements a novel method to compress redundant two-dimensional pixel segments over a long visualization session. To satisfy the second and the third requirements, the Varg system is based on a platform-independent remote desktop system VNC, whose implementation allows remote visualization of multiple applications in a network environment.

The main contribution of the Varg system is the novel method for compressing 2-D pixel segments for remote genomic data visualization. Genomic data visualization has two important properties that create opportunities for compression. The first is

that datasets tends to be very large. A microarray dataset typically consists of a matrix of expression values for thousands or tens of thousands of genes (rows). The second is that due to the limitation of display scale and resolution, researchers typically view only tens of genes at a time by frequently scrolling visualization frames up and down. As a result, the same set of pixels will be moved across the display screen many times during a data analysis and visualization session. We propose a novel method to identify, compress and cache 2-D pixels segments. Not sending redundant segments across the WAN greatly improves the effective compression ratio reducing network bandwidth requirements for remote visualization.

Our initial evaluation shows that the prototype Varg system can compress display information of multiple genomic visualization applications effectively, typically reducing the network bandwidth requirement by two orders of magnitude. We also demonstrate that this novel method is highly efficient and introduces a minimal overhead to the networking protocol; and that the Varg system can indeed support multiple visualization-intensive genomic applications in a remote environment interactively with minimal network bandwidth requirement.

2. System Overview

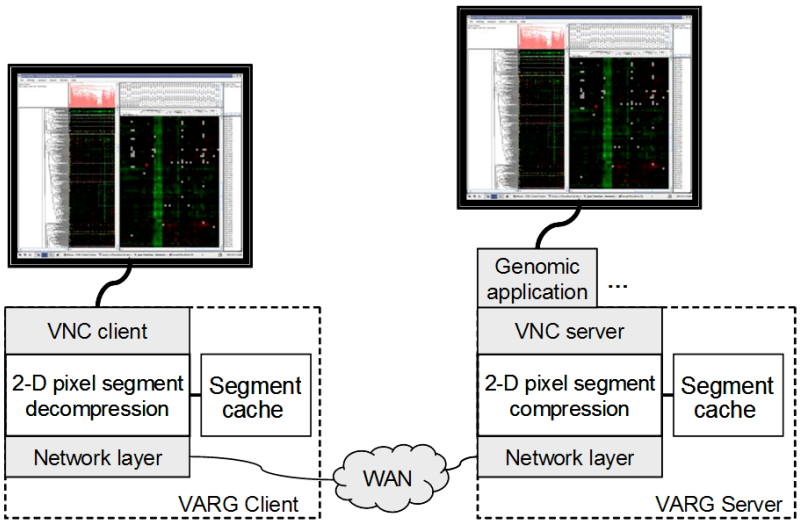


Figure 1: Remote visualization overview. The VNC server sends screen updates to the VNC client. The Varg system caches updates and provides compression by replacing updates already in the client cache with the cache index.

Varg is a network bandwidth optimized, platform-independent system that allows users to interactively visualize multiple remote genomic applications across a WAN. The architecture of Varg is based on a client-server model as shown in Figure 1. Varg leverages the basic VNC protocol (called RFB) to implement platform-independent

remote visualization and extends it with a high-speed 2-D pixel segment compression module with a cache in the server and a decompression module with a cache in the client. The Varg server runs multiple visualization applications, compresses their two-dimensional pixel segments, and communicates with the remote Varg client. The client decompresses the data utilizing a large cache and performs remote visualization.

The caches of the Varg server and client cooperate to minimize the required network bandwidth by avoiding redundant data transfers over the network. Unlike other global compression methods for byte data streams [17, 18], Varg is designed to optimize network bandwidth for remote data transfers of 2-D pixels segments generated by genomic visualization applications on the VNC server.

Since Varg is built on the VNC protocol, it allows multiple users to conveniently visualize and control a number of applications in a desktop across a network. When an owner of some sensitive or very large data set wants to collaborate with a remote collaborator, she can run one or more analysis programs that access her sensitive data on her Varg server, which connects with a Varg client on her collaborator's site. The researchers can then use these programs in a synchronized fashion across the network. Although the collaborator can visualize and control the application programs in the same way as the owner, the Varg client receives only visualization pixels from the Varg server; no sensitive data is ever transferred across the network. We expect that this feature may be especially useful to researchers working with clinical data due to privacy and confidentiality concerns.

3. Compressing 2-D Pixel Segments

The main idea in the Varg system is to compress visualization pixel data at a fine-grained 2-D pixel segment level. The system compresses 2-D pixel segments by using a global compression algorithm to avoid sending previously transferred segments and by applying a slow, but efficient, local compression [19] on the unique segments. This section describes Varg's basic compression algorithm, explains our novel content-based anchoring algorithm for 2-D pixel segments, and outlines an optimization of the compression algorithm using a two-level fingerprinting scheme that we developed.

3.1. Basic Compression Algorithm

The basic compression algorithm uses fingerprints together with cooperative caches on the Varg server and client to identify previously transferred pixel segments, as shown in Figure 2.

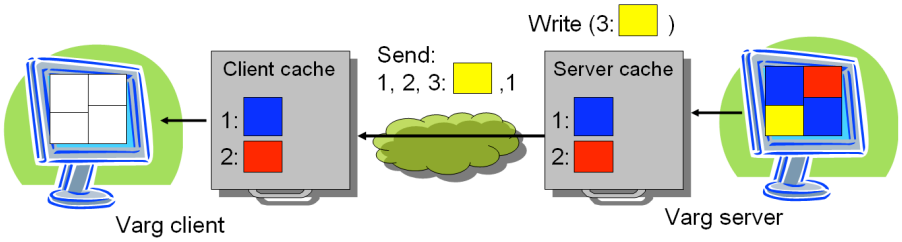


Figure 2: Compression scheme. The screen is divided into regions which are cached at both ends of the low-bandwidth network. Fingerprints are sent in place of previously sent regions.

The algorithm on the Varg server is:

- Process an updated region of pixels from the VNC server
- Segment the region into 2-D pixel segments
- For each segment, compute its fingerprint and use the fingerprint as the segment's identifier to lookup in the server cache. If the segment has not been sent to the Varg client previously, compress the segment with a local compression method and send the segment to the client. Otherwise, send the fingerprint instead.

The algorithm on the Varg client is:

- If the received data is a 2-D pixel segment, decompress it with a corresponding algorithm, write the fingerprint and segment to the cache, and then pass the segment to the VNC client
- If the received data is a fingerprint, retrieve the segment of the fingerprint from its cache and then pass the segment to the VNC client.

The basic algorithm is straightforward and its high-level idea is similar to previous studies on using fingerprints (or secure hashes) as identifiers to avoid transfer of redundant data segments [20, 21], [22], [17], [18]. The key difference is that previous studies are limited to deal with one-dimensional byte streams and have not addressed the issue of how to anchor 2-D pixel segments. In a later section, we will also present an algorithm to use short fingerprints to compress repeated 2-D pixel segments.

3.2. Content-Based Anchoring of 2-D Pixel Segments

One basis of our approach is content-based anchoring where the 2-D region of display-pixels is divided into segments based on segment content. A simpler approach would be to anchor segments statically (such as an 8×8 pixel grid, used in MPEG compression algorithms). The problem with a static approach is that the anchoring is sensitive to screen scrolls. When a user scrolls her visualization by one pixel, the segmentation of 2-D pixels will be shifted by one pixel relative to the displayed image. Even if the entire scrolled screen has been transferred previously,

the content of segments will typically have changed, giving a new fingerprint and requiring a new transfer across the WAN.

Our approach is to perform content-based anchoring instead of static anchoring. The anchoring algorithm takes its input from the frame-buffer, and returns a set of rectangular segments which subdivide the screen. The goal of the algorithm is to consistently anchor the same groups of pixels no matter where they are located on the screen. The main difficulty in designing a content-based anchoring algorithm for a screen of pixels is that the data is two dimensional.

Manber introduced a content-based technique to anchor one-dimensional data segments for finding similar files [22]. His method applies a Rabin fingerprint filter [23] over a byte data stream and identifies anchor points wherever the k least significant bits of the filter output are zeros. With a uniform distribution, an anchor point should be selected every 2^k bytes.

Our algorithm combines the statically divided screen approach with Manber's technique. The algorithm is based on the observation that content motion in microarray analysis is often due to vertical or horizontal scrolling. However, it is not practical to do redundancy detection both horizontally and vertically due to the computational cost and reduced compression ratio caused by overlapping regions. Therefore, we estimate whether the screen has moved mostly horizontally or mostly vertically using Manber's technique. We generate representative fingerprints for every 32nd row, and every 32nd column for the screenshot, and compare how many fingerprints are similar to the row and column fingerprints of the previous screenshot. Assuming that horizontal scrolling or moving will change most row fingerprints, but only a few column fingerprints, we can compare the percentage of similar row and column fingerprints to estimate which movement is dominant.

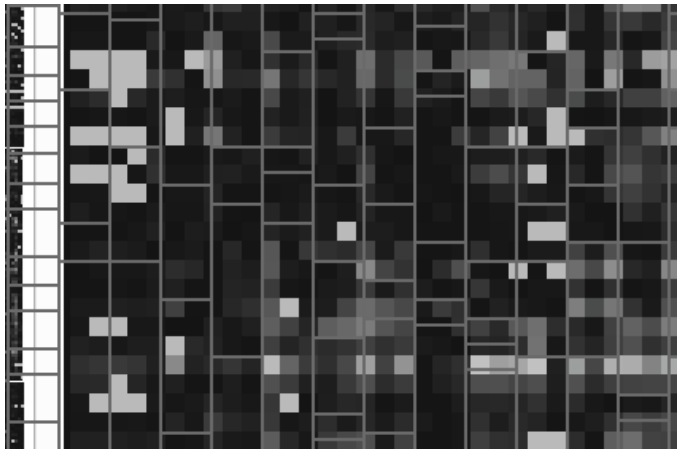


Figure 3: A portion of the screen that is divided into segments that move with the content.

For predominately vertical motion we statically divide the screen into *m* columns (*m* times screen height) and divide each column into regions by selecting anchoring rows. The anchoring rows are selected based on their fingerprint calculated using a

four byte at a time Rabin fingerprint implementation. The column segmentation is ideal for scrolling because the regions move vertically with the content. If we detect predominately horizontal motion instead, we run the same algorithm but divide the screen into rows first and then divide each row into regions by selecting anchoring columns.

Screen data can include pathological cases when large regions of the screen have the same color. For such regions, the fingerprints will be identical. Thus, either all or no fingerprints will be selected. To avoid such cases, our algorithm does fingerprint selection in three steps. First all fingerprints are calculated. Second, we scan the fingerprints and mark fingerprints as *similar* if at least *s* subsequent fingerprints are identical. Third, we select fingerprints using the *k* most significant bits, while imposing a minimum distance *m* between selected fingerprints. Also, the first and last rows are always selected.

Empirically we have found that the best results are achieved for s = 8, m = 16 or 32, and k such that each 64th row on the average is selected. Also, such similar regions compress well using a local compression algorithm such as zlib [24] due to their repeated content. We have found empirically that imposing a maximum distance does not improve the compression ratio or compression time.

3.3. An Optimization with Two-Level Fingerprinting

An important design issue in using fingerprints as identifiers to detect previously transferred data segments is the size of a fingerprint. Previous systems typically chose a secure hash, such as 160-bit SHA-1 [25], as a fingerprint so that the probability of a fingerprint collision can be lower than a hardware bit error rate. However, since the global compression ratio is limited to the ratio of the average pixel segment size to the fingerprint size, increasing the fingerprint size reduces this limit on the compression ratio.

To maximize the global compression ratio and maintain a low probability of fingerprint collision, we use a two-level fingerprinting strategy. The low-level fingerprinting uses 32-bit Rabin fingerprint of fingerprints, one for each 2-D pixel segments. Although using such short fingerprints will have a higher probability of a fingerprint collision, they can be computed quickly using the fingerprints already computed for the anchoring, thereby maintaining a high global compression ratio.

The high-level fingerprinting uses SHA-1 hashes as fingerprints. It computes a 160-bit fingerprint for each of the transferred pixel segments. The server computes such a long fingerprint as a strong checksum to detect low-level fingerprint collisions. When a low-level fingerprint collision is detected, the server resends the pixel segment covered by the long fingerprint.

Another way to look at this method is that the server may send two sets of updates, the first based on short fingerprints that can have collisions, and the second set of updates consisting of corrections in case of short fingerprint collisions. This method reduces the user perceived end-to-end latency.

4. Implementation

We have implemented a prototype system (called Varg) consisting of a sequential server and a client, as described in Section 2. The Varg server implements the 2-D pixel segment compression algorithm and Varg client implements the corresponding decompression algorithm described in the previous section.

The integration of Varg compression, decompression, and cache modules with the VNC client and server are simple. VNC has only one graphics primitive: "Put rectangle of pixels at position (x, y)" [12]. This allows separating the processing of the application display commands from the generation of display updates to be sent to the client. Consequently the server only needs to detect updates in the frame-buffer, and can keep the client completely stateless.

Varg employs a synchronized client and server cache architecture that implements an eventual consistency model using the two-level fingerprinting mechanism. The client and server caches are initialized at Varg system start time. The client cache is then synchronized by the updates sent from the server. The compression algorithm requires the client cache to maintain the invariant that whenever the client receives a fingerprint, its cache must have the fingerprint's segment. Since short fingerprints may have collisions, our prototype allows the client cache to contain any segment of the same short fingerprint at a given time. The long fingerprint will eventually trigger an update to replace it with a recently visualized segment.

5. Evaluation

We have conducted an initial evaluation of the Varg prototype system. The goal of the evaluation is to answer the following two questions:

- What are the network communication requirements for remote visualization of genomic applications?
- How much compression of network communication data can the Varg prototype system achieve for remote visualization of genomic applications?

To answer the first question, we have measured the difference between available bandwidth on current WANs and the required bandwidth for remote visualization of Genomic applications. To answer the second question, we used a trace-driven VNC emulator to find how much the Varg system can reduce the communication time for three genomic applications. In the rest of this section, we will present our experimental testbed and then our evaluation results to answer each question.

5.1. Experimental Testbed

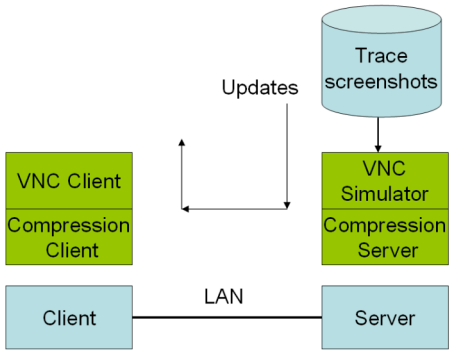


Figure 4: Experimental testbed for the bandwidth requirements and compression ratio evaluation.

In order to compare compression ratios of various compression algorithms, our experimental testbed (Figure 4) employs two identical Dell Dimension 9150, each with one dual-core 2.8 GHz Pentium D processor and 2 GB of main memory. Both computers run Fedora Core 4, with Linux kernel 2.6.17SMP.

The server runs with a screen resolution of 1280x1024 pixels and with a color depth of 32 bits per pixel. We also run an experiment on a display wall with a resolution of 3328x1536 pixels.

To compare different systems, an important requirement is to drive each system with the same remote visualization workloads. To accomplish this goal, we have used a trace-driven approach. To collect realistic traces, we used the Java AWTEventListener interface to instrument three genomic microarray analysis applications. We used these to record a 10-minute trace containing all user input events for each case. Later the traces were used to create a set of screenshots, each taken after playing back a recorded mouse or keyboard event that changes the screen content. The screenshots are used by a VNC simulator that copies a screenshot to a shadow framebuffer, and invokes the Varg server, which does change detection and compression before sending the updates to the client.

5.2. Network Communication Requirements

In order to answer the question about the network communication requirements for remote visualization of genomic applications, we need to answer several related questions including the composition of communication overhead, the characteristics of available networks, the behavior of remote visualization of genomic applications, and the required compression ratio to meet certain interactive requirements. Our finding is that genomic applications require high compression ratio to compress the pixel data to use existing WAN connections.

The network communication overhead can be expressed with a simple formula:

$$2L + \frac{S}{B \cdot R} + C \quad (1)$$

where L is the network latency, S is the data to be transferred, B is the network bandwidth, R is compression ratio, and C is compression time. The formula considers compression a part of the network communication mechanism, thus the total communication overhead includes the round-trip network latency plus the time to compress and transfer the data. This formula ignores the overheads of several software components such as the VNC client and server. Also, we usually ignore decompression time since it is low compared to the compression time (less than 1msec).

Based on this formula, it is easy to see that different network environments have different implications for remote visualization. Conventional wisdom assumes that WANs have low bandwidth. To validate this assumption we used Iperf [26] to measure the TCP/IP throughput between a server and a client connected using various local and wide area networks. The following table shows that the WAN throughput ranges from 0.2 to 2.13 Mbytes/sec (Table 1). This is up to 400 times lower than for Gigabit Ethernet. Also, the two-way latency is high, ranging from 11—120 ms.

Table 1: TCP/IP bandwidth and latency for client-server applications run on local area and wide area networks.

Network	Bandwidth (Mbytes/sec)	Latency (msec)
Gigabit Ethernet	80.00	0.2
100 Mbps Ethernet	8.00	0.2
Princeton – Boston	2.13	11
Princeton – San Diego	0.38	72
Princeton (USA(– Tromsø (Norway)	0.20	120

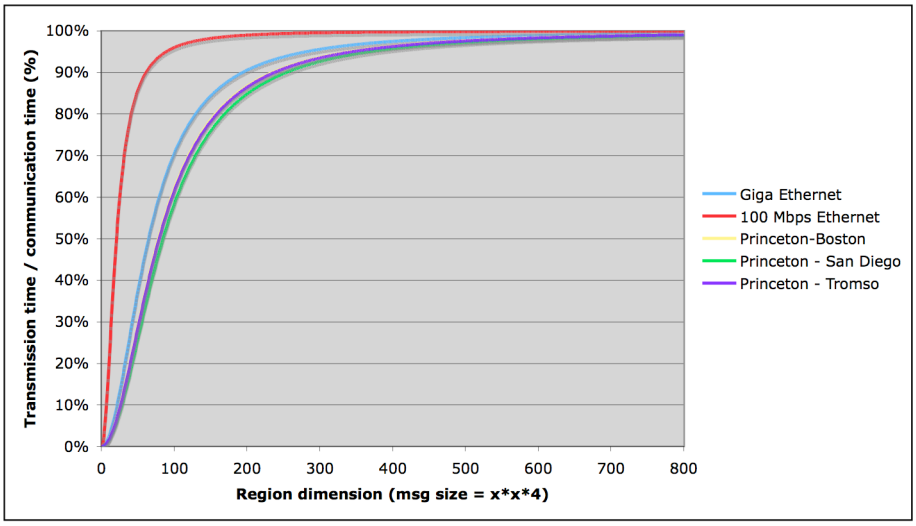


Figure 5: For regions larger than 80×80 pixels, the transmission time dominates the total communication overhead.

Based on the characteristics of the available networks, an interesting question is what size of network transfers contribute significantly to the total communication overhead. Figure 5 shows how much transmission contributes to the communication time depending on the amount of pixel data sent over the network connection. For all WAN networks, the ratio of transmission time to communication time is more than 50% for regions more than about 80×80 pixels or 25 Kbytes.

Two natural questions are, how frequent are screen updates larger than 80×80 pixels for genomics applications, and are the update sizes different compared to Office applications usually used in remote collaboration. To answer these questions, we measured the average VNC update size for three sessions using three applications on Windows XP:

1. Writing this paper in Microsoft Word.
2. Preparing the figures for this paper in Microsoft PowerPoint.
3. Microarray analysis using the popular Java Treeview software [27].

For each application, we recorded a session lasting about 10 minutes. We instrumented the VNC client to record the time and size of all screen updates received. We correlated these to when the update requests were sent, to get an estimate for the size of each screen update.

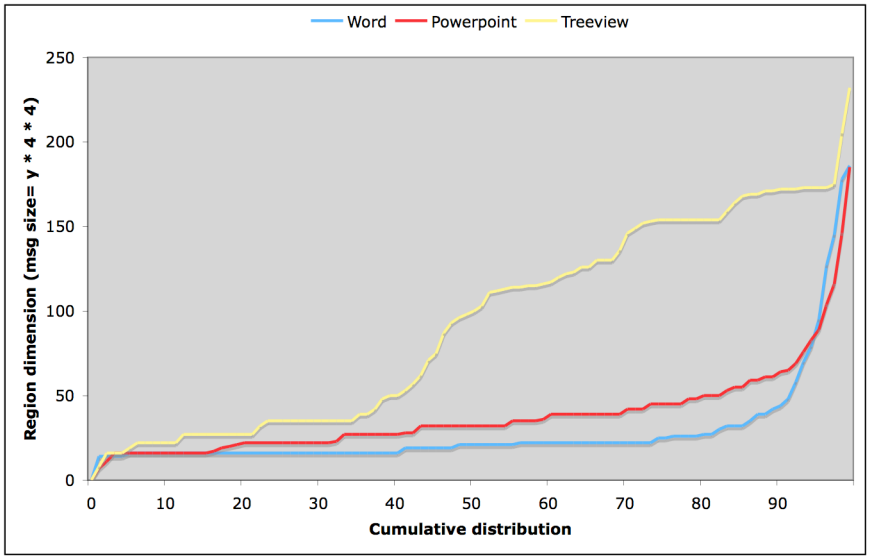


Figure 6: The screen regions update sizes for the Java Treeview application are much larger than for the Office applications. About 50% of the messages are more than 80x80 pixels.

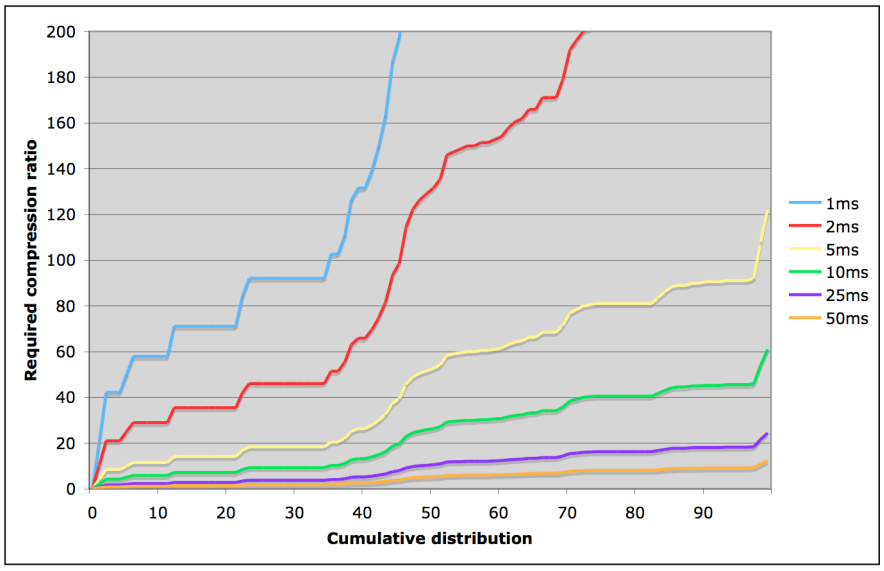


Figure 7: Compression ratio required to keep transmission overhead below a given threshold for the Princeton-San Diego network connection. The x-axis shows the percentiles for the Treeview message sizes in Figure 6**.** Compression time is not taken into account.

The results show that updated regions are much larger for the genomic application than for the two office applications (Figure 6). About 50% of the messages are larger

than 80×80 pixels, and hence for these the transmission time will be longer than the network latency for the WANs. Another observation is that the genomic application has a higher update frequency than office applications. Combined these increase the required bandwidth.

To see the impact of compressing pixel data for remote visualization, we have calculated the compression ratio necessary to maintain the transmission time below a given threshold for a cross-continent WAN (Figure 7). We have several observations from the results. First, it requires a compression ratio of about 25:1 to keep the transmission time below 10 msec for most of the network traffic. Second, the compression ratio required to maintain the same transmission time increases rapidly for the top two percentiles. Third, as the message size increases, the compression ratio required for the different transmission times increases.

The following section examines which compression ratio and compression time gives the best transmission time.

5.3. Compression results

To answer the question about what compression ratios the Varg system can achieve for remote visualization of genomic applications, we have measured compression ratios, compression cost and the reduction of transmission time.

Table 2: Compression ratio for four genomic data analysis applications.

	Differencing	2D pixel segment compression	Ziv-Lempel (zlib)	Total compression
TreeView	1.89	5.74	19.98	216.76
TreeView-Cube	2.87	4.05	24.88	289.19
TMeV	1.52	2.46	7.90	29.54
GeneVaND	3.15	2.72	10.85	92.96

To measure the compression ratios the Varg system can achieve, we have used four 15-minute traces recorded using: Java Treeview [27], Java Treeview on the display wall [28], TMeV [29], and GeneVaND [30]. For Treeview, the visualizations mostly are scrolling and selecting regions from a single bitmap. GeneVaND has relatively small visualization windows and the trace includes 3D visualizations as well as some 2D visualizations. TMeV trace includes different short visualizations.

The total compression ratios by our method are 217, 289, 30 and 93 for the four traces respectively (Table 2). These high compression ratios are due to a combination of three compression methods: Region differencing, 2D pixel segment compression, and zlib local compression. We have several observations based on these data. First, the combined compression results are excellent. Second, zlib contributes the most in all cases, but zlib alone is not enough to achieve high compression ratios. Third, the 2D pixel segment compression using fingerprinting contributes fairly significantly to

the compression ratio ranging from 2.5 to 5.7. This is due to the fact that the differencing phase has already removed a large amount of redundant segments.

Table 3: Average compression time per screen update. The total compression time depends on the application window size, and how well the differencing and 2D pixel segment compression modules compress the data before zlib is run.

	Differencing	2D pixel segment compression	Ziv-Lempel (zlib)	SHA-1
TreeView	0.9 ms	3.8 ms	11.1 ms	3.5 ms
TreeView-Cube	2 ms	7.9 ms	30.2 ms	7 ms
TMeV	1.3 ms	6.6 ms	83.4 ms	7.7 ms
GeneVaND	1 ms	2.7 ms	10.1 ms	1.5 ms

To understand the contribution of different compression phases to the compression time, we measured the time spent in each module (Table 2). The most significant contributor is zlib, which consumes more than 10 ms in all cases. In TMeV it consumes more than 83ms, since more data is sent through this module due to the low compression ratios for the differencing and 2D pixel segment compression modules. The second most significant contributor is anchoring, but it is below 8 ms even for the display wall case. Although SHA-1 calculation contributes up to 8ms in the worst case, its computation overlaps with network communication.

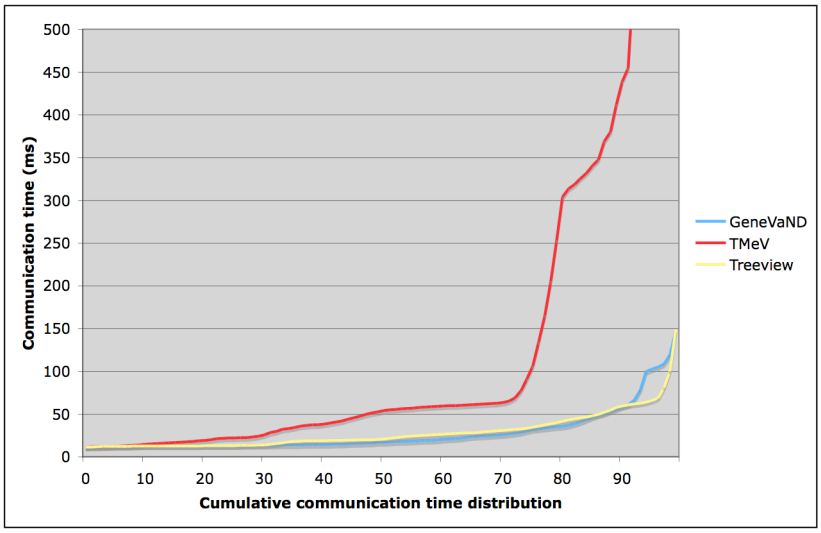


Figure 8: Communication time distribution for update messages over the Princeton—Boston network. For Treeview and GeneVaND more than 90% of the communication overheads are less than 100ms. The update size distribution differs from Figure 5, since a more accurate tracing tool was used to capture the trace.

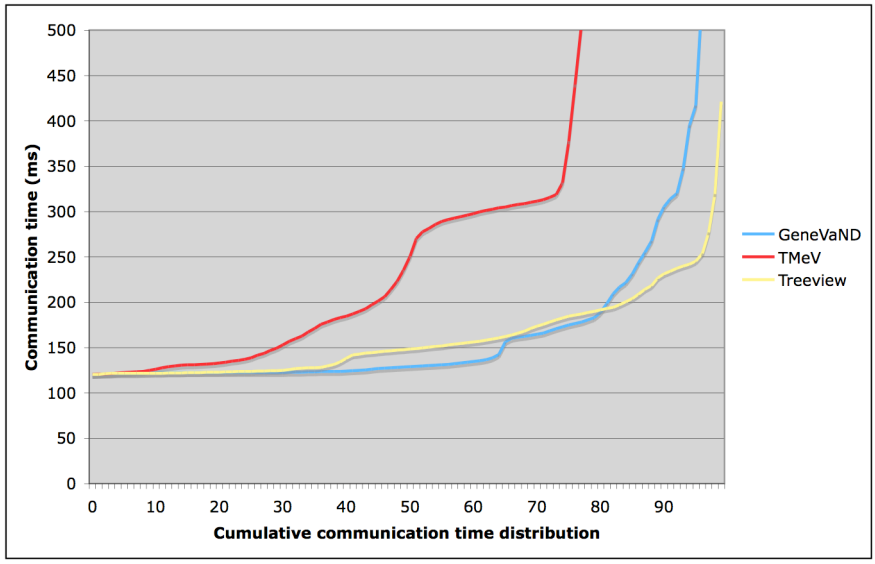


Figure 9: Communication time distribution for the Princeton—Tromsø network. For Treeview and GeneVaND more than 80% of the communication overheads are less than 200ms.

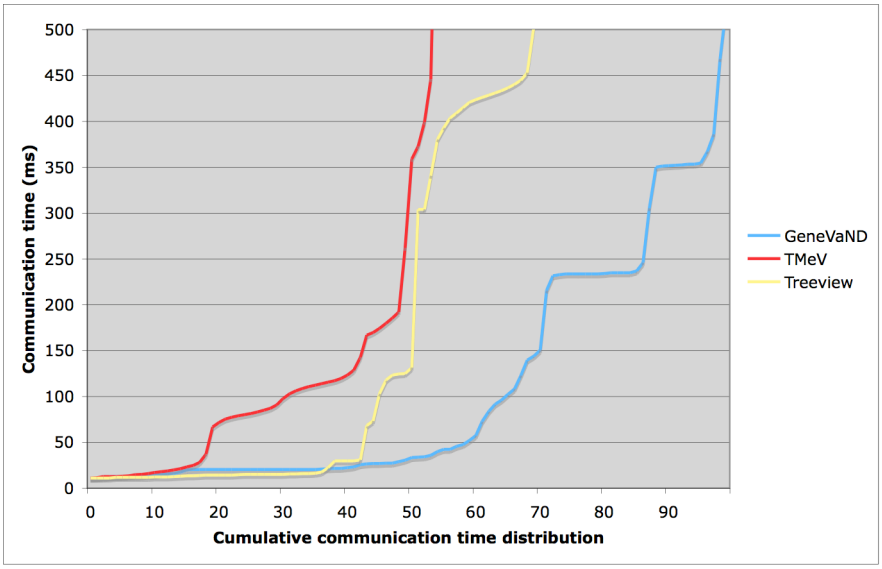


Figure 10: Communication time distribution with VNC compression for the Princeton—Boston network. Compared to Varg the communication time shown in Figure 9 significantly increases for large messages.

To understand the reduction in communication time, we recorded for each update the number of compressed bytes returned by each module, and the compression time for each module. This allows us to use Formula 1 to estimate the communication time for each of the WANs in Table 1. The cumulative distribution of communication times for the highest and lowest bandwidth networks are shown in Figure 8 and Figure 9. Without compression the communication overhead for the largest updates is several seconds. For the Princeton—Boston network the communication overhead with Varg is less than 100ms for over 90% of the messages (except for TMeV). On the Princeton—Tromsø network, for 80% of the update operations the communication overhead is less than 200ms, of which the latency contributes to 112 ms.

To compare our compression against the zlib compression used in many VNC implementations for low-bandwidth networks, we disabled the 2D pixel segment compression module in Varg, and did a similar calculation as above (Figure 10). The results show a significant increase in communication time, especially for Treeview where the communication overhead is more than 300ms for about 50% of the messages.

6. Related Work

Compression algorithms used by VNC [12] implementations either take advantage of neighboring region color similarities, use general purpose image compression [31] such as JPEG [32], or general purpose compression such as zlib [19]. Neighboring region redundancy compression is fast but has low compression ratio. Therefore zlib is usually used for WANs. Our results show that the compression time for zlib is high. JPEG is lossy, and is not suited for Microarray analysis, since it may introduce visual artifacts that may influence the biologist's interpretation of the data.

Remote visualization systems that use high level graphics primitives for communication, such as Microsoft Remote Desktop [15], are able to cache bitmaps used for buttons and other GUI components. However, the high-level graphics primitives do not compress well leading to performance problems in WANs [13, 33].

Encoders used for streaming video, such as MPEG [34], compress data by combining redundancy detection and JPEG type compression. Usually a static pixel grid is used, which we have shown gives worse performance than our approach. In addition the MPEG compression is lossy and there are no real time encoders available. TCC-M [35] is a block movement algorithm designed for thin-client visualization that use unique pixels in an image (feature sets) to detect 2D region movement. However, redundancy is only detected between the two latest screen updates thus reducing the compression ratio.

Earlier one-dimensional fingerprinting approaches [17, 18] require the two-dimensional screen to be converted to some one-dimensional representation. This will split up two-dimensional regions on the screen causing the size of the redundant regions to decrease, hence reducing the compression ratio.

Access Grid [36] provides multiple collaborators with multimedia services by multicasting audio, video and remote desktop displays such as VNC. However, Access Grid does not provide compression to reduce the network bandwidth requirement for specific data visualization such as genomic data exploration. Since Varg extends the VNC protocols to compress 2D segments for genomic data visualization, it can effectively work together with Access Grid systems to support multi-party collaborations.

7. Conclusion

This paper presents the design and implementation of the Varg system: a network bandwidth optimized, platform-independent system that allows users to interactively visualize multiple remote genomic applications across a WAN. The paper has proposed a novel method to compress 2-D pixel segments by using fingerprinting and proposed a two-level fingerprinting method to improve global compression, and to reduce compression time.

We also found that genomic applications have much higher network bandwidth requirements than office applications. They require substantial compression of network data to achieve interactive remote data visualization on some examples of existing WAN.

An initial evaluation of our prototype system shows that the proposed 2-D pixel segment compression method works well and imposes only modest overheads. By combining with zlib and differencing compression methods, the prototype system achieved compression ratios ranging from 30:1 to 289:1 for four genomic visualization applications that we have experimented with. Such compression ratios allow the Varg system to run remote visualization of genomic data analysis applications interactively across WANs with relatively low available network bandwidths.

8. Acknowledgments

This work was done while LAB and TL were visiting Princeton University and was supported in parts by Princeton University, The University of Tromsø, The Research Council of Norway Project No. 164825 159936/V30, 155550/420, NSF grants CNS-0406415, EIA-0101247, CNS-0509447, CCR-0205594, and CCR-0237113, and NIH grant R01 GM071966. OGT is an Alfred P. Sloan Research Fellow

9. References

1. Lipshutz RJ, Fodor SPA, Gingeras TR, Lockhart DJ: **High density synthetic oligonucleotide arrays**. *Nature Genetics* 1999, **21**:20-24.

2. Schena M, Shalon D, Davis RW, Brown PO: **Quantitative Monitoring of Gene-Expression Patterns with a Complementary-DNA Microarray**. *Science* 1995, **270**(5235):467-470.
3. Cahill DJ, Nordhoff E: **Protein arrays and their role in proteomics**. *Adv Biochem Eng Biotechnol* 2003, **83**:177-187.
4. Sydor JR, Nock S: **Protein expression profiling arrays: tools for the multiplexed high-throughput analysis of proteins**. *Proteome Sci* 2003, **1**(1):3.
5. Oleinikov AV, Gray MD, Zhao J, Montgomery DD, Ghindilis AL, Dill K: **Self-assembling protein arrays using electronic semiconductor microchips and in vitro translation**. *J Proteome Res* 2003, **2**(3):313-319.
6. Huang RP: **Protein arrays, an excellent tool in biomedical research**. *Front Biosci* 2003, **8**:d559-576.
7. Cutler P: **Protein arrays: the current state-of-the-art**. *Proteomics* 2003, **3**(1):3-18.
8. Kerr MK, Churchill GA: **Bootstrapping cluster analysis: assessing the reliability of conclusions from microarray experiments**. *Proc Natl Acad Sci U S A* 2001, **98**(16):8961-8965.
9. Yeung KY, Haynor DR, Ruzzo WL: **Validating clustering for gene expression data**. *Bioinformatics* 2001, **17**(4):309-318.
10. Mendez MA, Hodar C, Vulpe C, Gonzalez M, Cambiazo V: **Discriminant analysis to evaluate clustering of gene expression data**. *FEBS Lett* 2002, **522**(1-3):24-28.
11. Datta S, Datta S: **Comparisons and validation of statistical clustering techniques for microarray gene expression data**. *Bioinformatics* 2003, **19**(4):459-466.
12. Richardson T, Stafford-Fraser Q, Wood KR, Hopper A: **Virtual network computing**. *Ieee Internet Computing* 1998, **2**(1):33-38.
13. Schmidt BK, Lam MS, Northcutt JD: **The interactive performance of SLIM: a stateless, thin-client architecture**. In: *Proceedings of the seventeenth ACM symposium on Operating systems principles*. Charleston, South Carolina, United States: ACM Press; 1999.
14. Baratto RA, Kim LN, Nieh J: **THINC: a virtual display architecture for thin-client computing**. In: *Proceedings of the twentieth ACM symposium on Operating systems principles*. Brighton, United Kingdom: ACM Press; 2005.
15. Cumberland BC, Carius G, Muir A: **Microsoft Windows NT Server 4.0, Terminal Server Edition: Technical Reference**. In. Edited by Press M. Redmond, WA; 1999.
16. **Apple Remote Desktop** [http://www.apple.com/remotedesktop/]
17. Spring NT, Wetherall D: **A protocol-independent technique for eliminating redundant network traffic**. In: *Proceedings of the conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*. Stockholm, Sweden: ACM Press; 2000.
18. Muthitacharoen A, Chen B, Mazières D: **A low-bandwidth network file system**. In: *Proceedings of the eighteenth ACM symposium on Operating systems principles*. Banff, Alberta, Canada: ACM Press; 2001.

19. Ziv J, Lempel A: **A universal algorithm for sequential data compression**. *IEEE Transactions on Information Theory* 1977 **23**(3):337 - 343.
20. Broder A: **Some applications of Rabin's fingerprinting method**. In: *Sequences II: Methods in Communications, Security, and Computer Science: 1993*; 1993.
21. Broder A: **On the Resemblance and Containment of Documents**. In: *Proceedings of the Compression and Complexity of Sequences 1997*. IEEE Computer Society; 1997.
22. Manber U: **Finding similar files in a large file system**. In: *Proceedings of the Winter 1994 USENIX Technical Conference*. San Francisco, CA; 1994.
23. Rabin MO: **Fingerprinting by random polynomials**. In: *Technical Report TR-15-81*. Center for Research in Computing Technology, Harvard University; 1981.
24. **DEFLATE Compressed Data Format Specification version 1.3**. In: *RFC 1951*. The Internet Engineering Task Force; 1996.
25. **Secure Hash Standard**. In: *FIPS PUB 180-1*. National Institute of Standards and Technology; 1995.
26. **Iperf webpage** [http://dast.nlanr.net/Projects/Iperf/]
27. Saldanha AJ: **Java Treeview--extensible visualization of microarray data**. *Bioinformatics* 2004, **20**(17):3246-3248.
28. Wallace G, Anshus OJ, Bi P, Chen HQ, Clark D, Cook P, Finkelstein A, Funkhouser T, Gupta A, Hibbs M *et al*: **Tools and applications for large-scale display walls**. *Ieee Computer Graphics and Applications* 2005, **25**(4):24-33.
29. Saeed AI, Sharov V, White J, Li J, Liang W, Bhagabati N, Braisted J, Klapa M, Currier T, Thiagarajan M *et al*: **TM4: a free, open-source system for microarray data management and analysis**. *Biotechniques* 2003, **34**(2):374-378.
30. Hibbs MA, Dirksen NC, Li K, Troyanskaya OG: **Visualization methods for statistical analysis of microarray clusters**. *BMC Bioinformatics* 2005, **6**:115.
31. Richardson T: **The RFB Protocol version 3.8**. In.: RealVNC Ltd; 2005.
32. Gregory KW: **The JPEG still picture compression standard**. *Commun ACM* 1991, **34**(4):30-44.
33. Lai AM, Nieh J: **On the performance of wide-area thin-client computing**. *ACM Trans Comput Syst* 2006, **24**(2):175-209.
34. Gall DL: **MPEG: a video compression standard for multimedia applications**. *Commun ACM* 1991, **34**(4):46-58.
35. Christiansen BO, Schauser KE: **Fast Motion Detection for Thin Client Compression**. In: *Proceedings of the Data Compression Conference (DCC '02)*. IEEE Computer Society; 2002.
36. **Access Grid** [http://www.accessgrid.org/]

Chapter 8

Appendix B - Unpublished papers

8.1 The Longcut Wide Area Network Emulator: Design and Evaluation

This paper has been published as a technical report [42].

The Longcut Wide Area Network Emulator: Design and Evaluation

Lars Ailo Bongo
Department of Computer Science
University of Tromsø
Norway
Email: larsab@cs.uit.no

***Abstract*— Experiments run on a Grid, consisting of clusters administered by multiple organizations connected by shared wide area networks (WANs), may not be reproducible. First, traffic on the WAN cannot be controlled. Second, allocating the same resources for subsequent experiments can be difficult. Longcut solves both problems by splitting a single cluster into several parts, and for each part having one node emulating a WAN link by delaying messages sent through it. The delay is calculated using latency and bandwidth measurements collected using the Network Weather Service and a parallel application monitor. We evaluate the precision, usability for WAN collective operation research, and scalability of Longcut.**

I. Introduction

A Grid consists of clusters administered by multiple organizations connected by shared wide area networks. Two factors make system research difficult in such an environment. First, experiments may not be reproducible since the traffic on shared WANs cannot be controlled [8]. Second, allocating exclusive access, at the same time, to several clusters is usually not supported by Grid middleware. To avoid these problems a large cluster (or several small cluster) at a single site can be used with emulated WAN links.

As input to the emulator we use latency and bandwidth traces of real WAN links collected using the Network Weather Service (NWS) [16] and the EventSpace parallel application monitor [2].

The reaming of this paper proceeds as follows. Section II describes related work. Section III describes the design and implementation of the Longcut WAN emulator. The trace collection tools are described, and the collected traces evaluated, in section IV. Longcut is evaluated in section V, by doing experiments measuring the precision, scalability, and usability for WAN collective operation research of the collected traces. Finally section VI concludes.

II. Related Work

The design of Longcut is inspired by the Panda WAN emulator [8]. Both use sub-cluster gateway nodes to run WAN emulation code. Also, both are closely integrated with the communication system. Our experiments differ from the distributed work queue experiments in [8], in that we use applications with higher communication frequency.

Other emulators are Netbed [15]. Dummynet [13], nse [7], Trace Modulation [10] and ModelNet [14]. Most use low-level rerouting which requires adding a module to the operating system. Longcut runs unmodified applications on unmodified operating systems.

Alternatives to emulation are simulation [5], [17] and live-network experimentation. Simulation provides a controlled, easy to change, and repeatable environment. However, higher level abstraction must be used due to the scale of the system; thus accuracy is lost. Live-network experimentation using environments such as PlanetLab [11] is most realistic, but often these are not designed for performance experiments. For example PlanetLab uses virtualization to share resources and protect services from each other, which makes it difficult to control the load on resources.

There are several network monitoring tools [6], [9], [16] that can be used to collect the traces used by Longcut. However, most of the existing traces do not have the high sample rate required for our experiments.

III. Design and Implementation

In many clusters a gateway node provides the single entry point to the compute nodes, to the benefit of cluster users and administrators.

The design of Longcut is similar to the Panda WAN emulator [8]. A cluster is split into several sub-clusters. For each sub-cluster we select one node to act as a gateway. All communication to the sub-cluster is routed through its gateway, which delays messages to emulate the higher latency of WAN connections.

To implement Longcut, we need to change the communication paths used by applications, such that messages are routed through the gateway where the emulation code is run. Being a research tool, Longcut should be extensible such that users can add their own emulation code, and configurable such that the emulated topology can be easily changed. Our communication system, PATHS [1], supports all this.

PATHS provide configurable *paths* though the communication system. A path consists of several *wrappers* that can run arbitrary code. Figure 1 shows how we reconfigure a path between two nodes to include a gateway node which runs the emulation code in form of a wrapper. Extending Longcut with other emulation approaches requires writing a new wrapper (consisting of 3 functions).

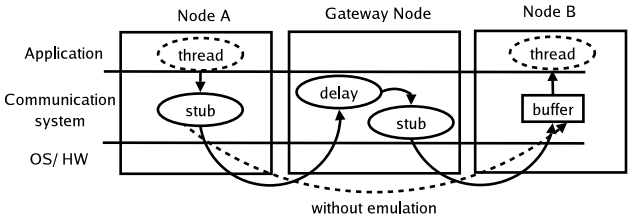


Fig. 1. Communication path with WAN emulation wrapper.

All communication paths used by an application are specified in a *pathmap* [2], which is created using three data structures: cluster topology, application communication information, and a mapping of application threads and communication buffers to the clusters. To re-route messages the cluster topology is changed. To add emulation wrappers, scripts are run that reconfigure the pathmap.

We have implemented two types of WAN emulation where the delay is calculated using: (i) constant WAN latency-, and bandwidth, and (ii) latency and bandwidth time series read from trace files. The first type is useful for simple experiments where different topologies are evaluated. The tools used to collect the traces are described in the following section.

On the gateway there is one thread per TCP/IP connection. In our initial implementation the threads waited either by blocking (by calling *usleep*) or spinning. Spinning had to be used since it was not possible to sleep for less than 30 ms. This approach does not scale well, since gateways emulating many WAN links can have many threads spinning at the same time causing loss of accuracy (as reported in [4]).

To make sure only one thread spins at a time, we reimplemented the delay code as shown in figure 2. Threads block if there is already one thread spinning. Threads are unblocked when the currently spinning thread exists, or when they are done waiting. The scalability of Longcut is evaluated in section V.

IV. Trace Collection

Five cluster gateways were monitored:

vvgw.cs.uit.no	: Pentium 4 3.2 GHz in Tromsø, Norway.
psgw.cs.uit.no	: dual Pentium II 300 MHz in Tromsø, Norway.
clustis.idi.ntnu.no	: dual Athlon MP 1.6 GHz in Trondheim, Norway.
roadrunner.imada.sdu.dk	: Pentium III 1.4 GHz in Odense, Denmark.
benedict.aau.dk	: dual-CPU Pentium III 733 MHz in Aalborg, Denmark.

The topology, ping latency and link bandwidth of the WANs between the monitored nodes are shown in figure 3.

A. Monitoring Tools

1) Network Weather Service: A widely used network monitoring tools is the Network Weather Service (NWS) [16].

```
done_time = current_time() + wait_time;

if (somebody spinning)
  // signaled by spinning thread
  condition_wait();

while (1) {
  current_time = timestamp();
  if (current_time() > done_time) {
    if (thread blocked)
      // unblocked thread will do the
      // spinning
      condition_signal();
    break;
  }

  for (each blocked thread)
    if (current_time() > thread done_time)
      // unblocked thread will exit
      condition_signal();

  // allow others to run
  yield();
}
```

Fig. 2. Pseudo code for the delay function.

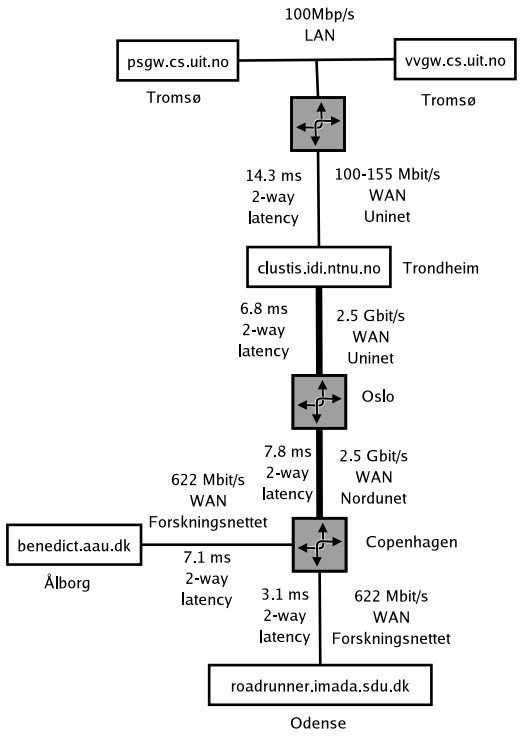


Fig. 3. The monitored topology (all intermediate routers are not shown).

TABLE I
NWS TWO-WAY LATENCY IN MILLISECONDS.

	benedict	clustis	psgw	roadrunner	vvgw
benedict		22.43	36.70	7.11	36.52
clustis	22.14		17.88	18.81	14.88
psgw	36.71	15.03		34.84	0.31
roadrunner	7.03	18.83	34.00		32.95
vvgw	36.29	14.52	0.54	32.82	

TABLE II
NWS MEAN BANDWIDTH IN MBITS/SEC.

	benedict	clustis	psgw	roadrunner	vvgw
benedict		3.15	2.58	8.60	2.32
clustis	3.50		5.57	3.96	5.63
psgw	2.28	4.85		2.58	81.73
roadrunner	6.27	2.70	1.67		1.87
vvgw	2.28	4.78	79.64	2.56	

NWS has low monitoring overhead, and has been shows to provide measurements accurate enough to predict future TCP/IP latencies and bandwidth [16]. It is easy to install and use, but three ports need to be opened on firewalls.

Latency is measured by sending a four byte message. Bandwidth is measured by sending four 16 Kbytes messages using a socket buffer size of 32 Kbytes each 60th second. We tried using a shorter sample period (1 second), but the rate was too high for the monitored WAN connections.

2) EventSpace: Using the EventSpace monitoring tool [2] we can trace the latencies of TCP/IP connections as used by a communication system for WANs. EventSpace allows low-overhead monitoring of the actual communication rate of the applications we are interested in. However, installing EventSpace can be difficult due to a large number of libraries used (e.g. Python). Also, firewalls need to be opened for the PATHS server ports.

We collected traces for two benchmarks. The first was collected for a collective operation micro-benchmark run on a multi-cluster (the experiment is described in [3]). As only small message were used, we do not report bandwidth results. In the second experiment, we used a benchmark designed for latency and bandwidth measurements. For each iteration it sends an eight byte message, followed by two 32 Kbytes messages. Sends were blocking, hence one must complete before a new one can be initiated.

We did one experiment were the Nagle algorithm was disabled on all TCP/IP connections, to ensure that even small messages are sent immediately, but it did not significantly reduce the latency.

B. Collected Traces

Tables I, III and IV shows the mean two-way TCP/IP latency measured for the different links (in both directions). The NWS trace has smaller mean latencies for small latency links than the EventSpace traces. Tables II and V shows the TCP/IP throughput. The EventSpace trace has higher bandwidth than the NWS trace.

TABLE III
EVENTSPACE COLLECTIVE OPERATION TRACE TWO-WAY LATENCY (MILLISECONDS).

	benedict	psgw	roadrunner
benedict		35.76	9.16
psgw			32.49
roadrunner		32.35	

Increasing the sample rate lowers the observed variation both in bandwidth and latency. Also, the bandwidth differs in two directions, while the latency usually does not. Conclusions should not be drawn from the above results since we have only collected one trace for each link.

V. EXPERIMENTS

For the experiments we use a cluster with 44 nodes, each with a single-CPU Pentium 4 3.2 GHz with Hyper-threading (2-way SMT) enabled. The nodes are connected using Gigabit Ethernet, and all run Linux with kernel version 2.4.21. We use NPTL threads for the experiments. On all TCP/IP connections the Nagle algorithm was disabled and default socket sizes were used. The delay is implemented with the single-thread-spinning approach described above.

A. Precision

To investigate the precision of Longcut, we measured application level ping-pong latency and bandwidth between a cluster in Tromsø and Trondheim using PingPong from the Pallas Microbenchmark suite (PMB) [12] (ported to PATHS). We also traced the link by using EventSpace to monitor the latency-bandwidth micro-benchmark, and used the captured trace to emulate the link on our cluster. Each experiment was repeated twice. For most message sizes the real and emulated links have similar latency and bandwidth (figure 4).

We also measured how different traces influence the latency and bandwidth of PingPong. Two traces were used; the NWS and EventSpace latency-bandwidth microbenchmark traces presented in section IV. Also we did one experiment with constant latency and bandwidth values (means from the EventSpace trace).

Figure 5 shows the difference in latency and bandwidth for the WAN link between Odense and Aalborg. The PingPong results differ for the NWS and EventSpace traces since the NWS trace has lower latency and lower latency than the EventSpace trace. However, using constant values does not differ significantly from using the EventSpace trace, even if it has smaller variation. We have similar results for other links.

We also measured how the different traces influence the performance of collective communication using Allreduce from PMB. The cluster was split into four parts with 10 nodes in each part in addition to the node selected as gateway. The four clusters were emulated to be in Tromsø (behind vvgw), Trondheim, Odense and Aalborg. The difference between the constant value trace and the EventSpace trace is smaller than for PingPong (figure 6). However, the difference in NWS

TABLE IV

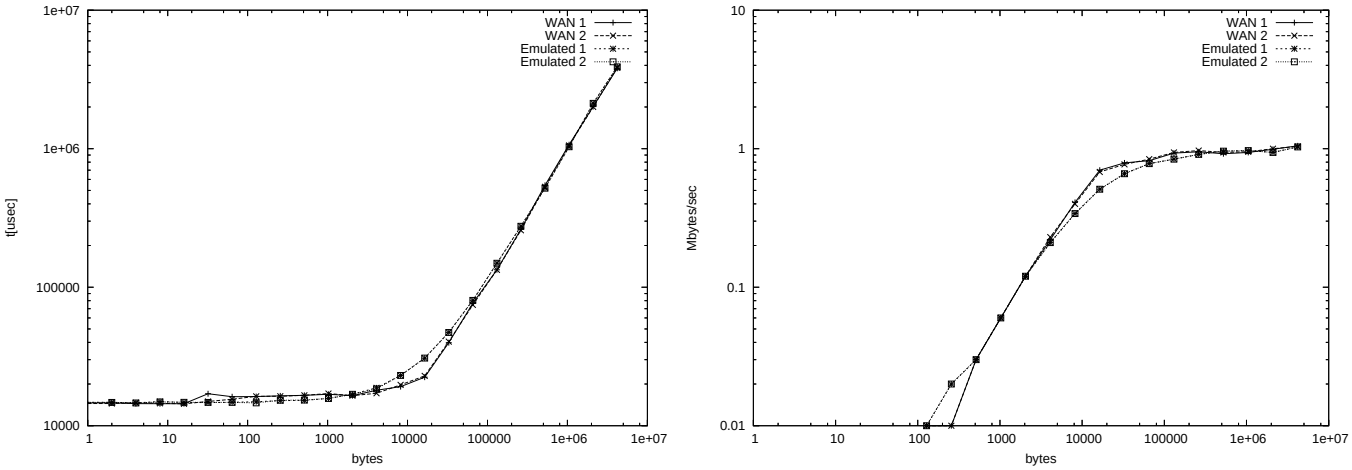
EVENTSPACE LATENCY-BANDWIDTH MICROBENCHMARK TRACE LATENCY (MILLISECONDS). STANDARD DEVIATION IN PARENTHESIS.

	benedict	clustis	psgw	roadrunner	vvgw
benedict		23.18 (3.31)	37.19 (1.66)	12.70 (27.03)	36.98 (1.48)
clustis	22.82 (3.41)		14.80 (2.47)	23.87 (30.67)	14.91 (3.84)
psgw	37.13 (1.57)	15.00 (1.17)		36.90 (28.51)	1.81 (1.52)
roadrunner	9.85 (4.02)	20.43 (3.12)	33.93 (3.75)		34.02 (4.38)
vvgw	36.95 (1.46)	15.00 (1.19)	1.84 (1.34)	36.49 (26.84)	

TABLE V

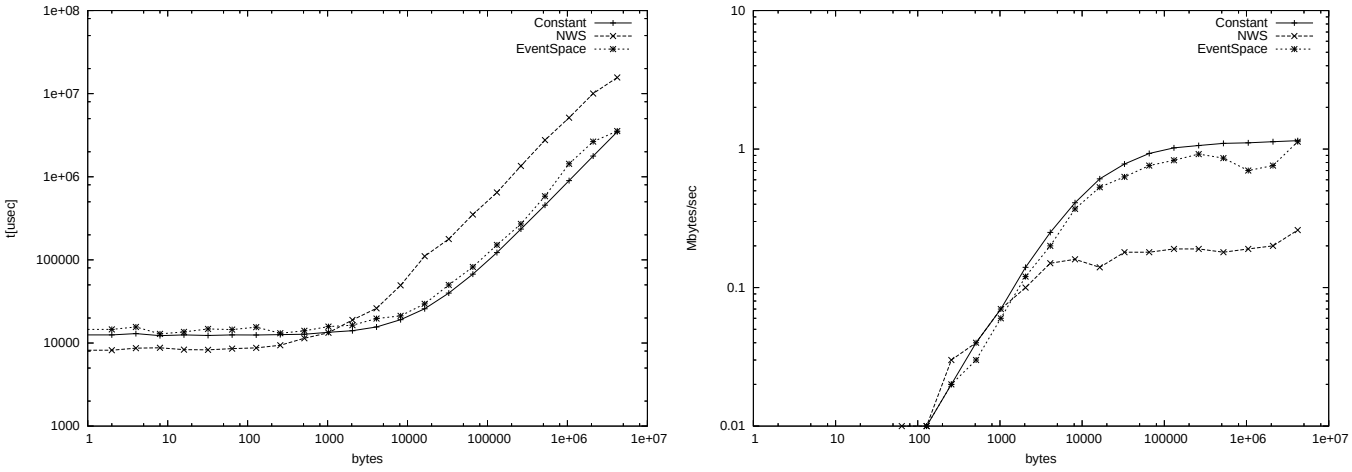
EVENTSPACE LATENCY-BANDWIDTH MICROBENCHMARK TRACE BANDWIDTH (MBITS/SEC). STANDARD DEVIATION IN PARENTHESIS.

	benedict	clustis	psgw	roadrunner	vvgw
benedict		8.80 (0.45)	5.75 (0.66)	10.60 (3.22)	3.19 (0.08)
clustis	4.97 (0.50)		9.74 (3.01)	6.76 (2.35)	7.78 (0.69)
psgw	3.18 (0.10)	12.73 (0.75)		4.86 (1.64)	44.49 (10.99)
roadrunner	10.08 (1.28)	8.18 (2.30)	5.49 (1.48)		3.46 (0.26)
vvgw	3.19 (0.09)	12.79 (0.76)	51.50 (9.68)	4.75 (1.63)	



(a) Latency (b) Bandwidth

Fig. 4. Measured and emulated PingPong latency and bandwidth between nodes in Tromsø and Trondheim.



(a) Latency (b) Bandwidth

Fig. 5. Emulated PingPong latency and performance using different traces for Odense and Aalborg link.

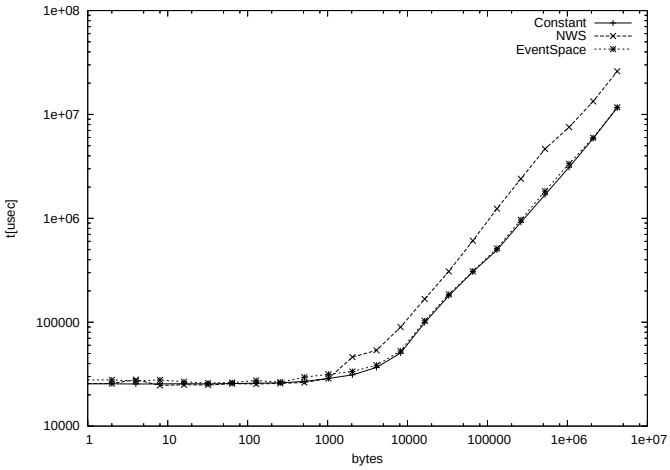


Fig. 6. Emulated Allreduce latency using different traces.

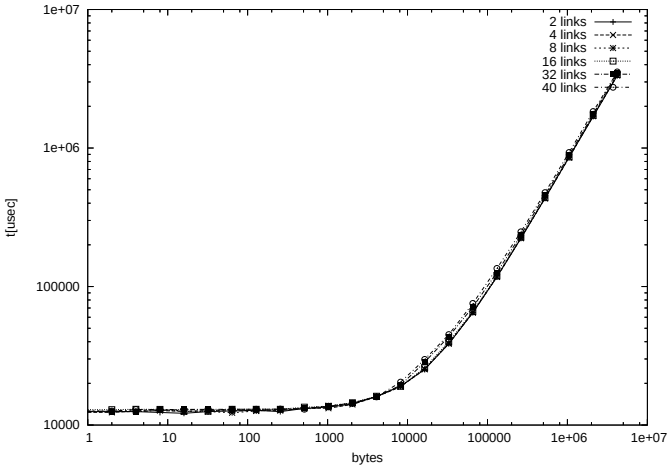


Fig. 7. Longcut scalability measured using PingPong with increasing number of emulated connections per gateway.

and EventSpace latency and bandwidth do influence Allreduce performance.

B. Scalability

To evaluate the scalability of Longcut, we measured the number of TCP/IP connections each gateway can emulate without loss in precision. The cluster was divided into two parts with 20 nodes in each part, and we run several instances of PingPong, all communicating over emulated WAN links (Aalborg–Odense). For each instance of PingPong each gateway handles two TCP/IP connections. Figure 7 shows the maximum latency observed for each experiment. PingPong latency does not differ when emulating 2 and 40 connections.

C. Usability

In our final experiment, we measure the execution time of an application kernel. The kernel is Successive Over-Relaxation (SOR). We use a Red-Black checker pointing version of SOR, with a matrix size of 48000×48000. The cluster was divided into four parts as described above. Each worker-process is assigned 1200 rows, and each updates all its red points and then exchanges red border point values by sending a 19800 bytes message to each neighbor. Then black points are updated and the communication is repeated. At the end of each iteration the global change in the system is calculated using allreduce (with and 8 byte message). For the problem size chosen about 70% of the execution time is spent communicating when using the NWS trace. Table VI shows the execution time, and the slowdown compared to the constant value trace.

TABLE VI
SOR PERFORMANCE WITH DIFFERENT TRACES.

Trace	Exec. time	Slowdown
Constant	383.8 sec.	
EventSpace	390.3 sec.	2%
NWS	461.9 sec.	17%

VI. CONCLUSION

We have described the design and implementation of the Longcut WAN emulator, shown the emulation precision using traces collected by different tools, and evaluated the scalability of Longcut.

We learned the following lessons:

- For most traces, bandwidth differs in two directions, while latency does not.
- Traces with finer granularity have higher latency.
- The difference for point-to-point communication performance does not significantly differ when using constant and traced latency and bandwidth values.
- For synchronizing collective communication, such as allreduce, there are small differences between using latency-bandwidth traces and constant values.

The collected traces are available at http://www.cs.uit.no/∼larsab/longcut/.

ACKNOWLEDGMENTS

Thanks to Brian Vinter for providing us access to the clusters in Denmark. Also thanks to Josva Kleist and Gerd Behrmann for allowing us to use the cluster in Aalborg, and Anne C. Elster for allowing us to use the cluster in Trondheim. Thanks to Otto J. Anshus, John Markus Bjørndalen and Espen S. Johnsen for discussions, and to the MNF-8000 students who were referees for this paper.

REFERENCES

[1] BJØRNDALEN, J. M. *Improving the Speedup of Parallel and Distributed Applications on Clusters and Multi-Clusters*. PhD thesis, Department of Computer Science, University of Tromsø, 2003.

[2] BONGO, L. A., ANSHUS, O., AND BJØRNDALEN, J. M. EventSpace - Exposing and observing communication behavior of parallel cluster applications. In *Euro-Par* (2003), vol. 2790 of *Lecture Notes in Computer Science*, Springer, pp. 47–56.

[3] BONGO, L. A., ANSHUS, O., BJØRNDALEN, J. M., AND LARSEN, T. Extending collective operations with application semantics for improving multi-cluster performance. In *ISPDC/HeteroPar* (July 2004), IEEE Computer Society, pp. 320–327.

[4] BONGO, L. A., ANSHUS, O. J., AND BJØRNDALEN, J. M. Low overhead high performance runtime monitoring of collective communication, 2005. To appear in Proc. of ICPP'05.

[5] BRAKMO, L. S., AND PETERSON, L. L. Experiences with network simulation. In *SIGMETRICS '96: Proceedings of the 1996 ACM SIGMETRICS international conference on Measurement and modeling of computer systems* (1996), ACM Press, pp. 80–90.

[6] DINDA, P., GROSS, T., KARRER, R., LOWEKAMP, B., MILLER, N., STEENKISTE, P., AND SUTHERLAND, D. The architecture of the Remos system. In *Proc. 10th IEEE Symp. on High Performance Distributed Computing* (2001).

[7] FALL, K. Network emulation in the vint/ns simulator. In *In Proc. IEEE ISCC '99* (1999).

[8] KIELMANN, T., BAL, H. E., MAASSEN, J., VAN NIEUWPOORT, R., EYRAUD, L., HOFMAN, R., AND VERSTOEP, K. Programming environments for high-performance grid computing: the Albatross project. *Future Generation Computer Systems 18*, 8 (2002), 1113–1125.

[9] Network performance tools. http://www.caida.org/tools/taxonomy/.

[10] NOBLE, B. D., SATYANARAYANAN, M., NGUYEN, G. T., AND KATZ, R. H. Trace-based mobile network emulation. In *SIGCOMM '97: Proceedings of the ACM SIGCOMM '97 conference on Applications, technologies, architectures, and protocols for computer communication* (1997), ACM Press, pp. 51–61.

[11] PETERSON, L., CULLER, D., ANDERSON, T., AND ROSCOE, T. A blueprint for introducing disruptive technology into the internet, 2002.

[12] PMB - Pallas MPI Benchmarks, http://www.pallas.com/e/products/pmb/.

[13] RIZZO, L. Dummynet and forward error correction. In *In Proc. of the 1998 USENIX Annual Technical Conf* (June 1998).

[14] VAHDAT, A., YOCUM, K., WALSH, K., MAHADEVAN, P., KOSTIC, D., CHASE, J., AND BECKER, D. Scalability and accuracy in a large-scale network emulator. In *In Proc. 5th OSDI* (Dec 2002).

[15] WHITE, B., LEPREAU, J., STOLLER, L., RICCI, R., GURUPRASAD, S., NEWBOLD, M., HIBLER, M., BARB, C., AND JOGLEKAR, A. An integrated experimental environment for distributed systems and networks. In *Proc. of the Fifth Symposium on Operating Systems Design and Implementation* (December 2002), pp. 255–270.

[16] WOLSKI, R., SPRING, N. T., AND HAYES, J. The network weather service: a distributed resource performance forecasting service for metacomputing. *Future Generation Computer Systems 15*, 5–6 (1999).

[17] ZENG, X., BAGRODIA, R., AND GERLA, M. Glomosim: a library for parallel simulation of large-scale wireless networks. In *PADS '98: Proceedings of the twelfth workshop on Parallel and distributed simulation* (1998), IEEE Computer Society, pp. 154–161.

8.2 Impact of Operating System Interference on Ethernet Clusters

This section summarizes the results of an unpublished paper.

The work was done primarily in collaboration with Otto J. Anshus.

Abstract. Operating system interference has been identified as an important reason for parallel application scalability problems on large-scale clusters with fast interconnects. In this paper we measure and characterize operating system interference in an Ethernet cluster. Our results shows that operating system interference is not a significant factor for Ethernet clusters.

8.2.1 Introduction

Operating system interference has been identified as an important reason scalability problems for applications with globally synchronizing operations on large clusters with fast interconnects [118, 166, 226]. The problem was caused by lack of coordination between the operating systems on the different nodes. Thus, when one operating system scheduler decides to run daemon code instead of application code, all other nodes must wait at the globally synchronizing operation for data from this node. When the number of nodes increases the probability for one node running daemon code between synchronization points increases. A solution to this problem is removing unnecessary daemons and modifying the operating system scheduler such that scheduling is globally controlled.

Previous work [118, 166, 226] has mostly been done on clusters with SMPs and high speed interconnect such as QsNet [165]. Our environment differs in that we have:

1. Nodes with only one CPU
2. Nodes with simultaneous multi-threading (SMT)
3. GigaBit Ethernet
4. Fewer nodes

We believe many clusters have similar characteristics. We are interested in answering the question: will operating system interference increase the execution time of parallel applications run on medium size Ethernet clusters?

8.2.2 Methodology

Two clusters are used:

- Tin: 51 single-CPU Pentium 4 Hyper-threaded 3.2 GHz, 2 GB RAM
- Iron: 39 single-CPU Pentium 4 Hyper-threaded 3.2 GHz, 2 GB RAM with EM64T extension.

The processors in the Tin and Iron clusters support SMT, which was enabled during the experiments. The interconnect on both clusters is Gigabit Ethernet, while inter-cluster communication uses the departments 100 Mbps Ethernet. The parallel communication system used is LAM/MPI [3, 52].

We assume the combined size of the Tin and Iron clusters is above the *critical mass* for when noise becomes a problem. For example, in [166] the difference before and after removing noise for 128 CPUs is about 50%.

To measure the noise introduced by system activities we use a similar micro-benchmarks as in [166], but with a different configuration in order to adapt to the higher network latency in our cluster. The benchmark consists of a matrix multiplication that can be tuned to take *n* milliseconds. The computation is repeated *m* times such that the computation takes 1000 seconds. To also measure the communication system noise we add an allreduce call after the computation.

We use three different computation times per iteration: (a) 1 ms (used in [166]), (b) 10 ms which is the classical operating system time slice length [212, 226], and (c) the latency of the allreduce operation on the cluster.

8.2.3 Results

Experiment	Mean	Stdev	+1.35ms	+2ms	+5ms	+10ms
Computation	1.08 ms	0.01 ms	14	0	0	0
Communication	1.37 ms	0.27 ms	1196	812	322	0
Iteration	2.46 ms	0.27 ms	1198	813	322	0

Table 14: Number of iterations where at least one of 50 threads is delayed for 1.35 ms, 2 ms or 5 ms.

In the first experiment we run the benchmark on 50 Tin hosts for 450.000 iterations. The time per computation has a very low variation, but the variation for allreduce latency is larger.

To estimate the delay caused by system interference, we calculated for each thread the median time per computation, allreduce and per iteration. Then for each thread we counted the number of iterations where the thread was delayed for more than *x* ms, where *x* was the mean time per allreduce operation (1.35 ms), 2 ms, 5 ms and 10 ms. The results are shown in Table 14.

For only 0.3% of all iterations was at least one thread more than 1.35 ms delayed. The delay is usually caused by variation in the time per allreduce operation. The time per computation is only delayed with more than 1.35 ms in 14 iterations. Together these delay the benchmark with about 7.4 seconds, which is insignificant compared to an execution time of 1106 seconds. With a larger computation time the impact of noise was even smaller. Running the benchmarks on both clusters gives similar results.

Per thread delay counts shows an even distribution of iterations with delay, hence there are no particular nodes causing the nodes. Which is not surprising since the nodes are homogenous both in hardware and software. Also, the workload is evenly distributed with the exception of the allreduce mapping to the cluster. The allreduce introduces some work to some nodes, but it is not shown in the delay distribution.

8.2.4 Conclusion

Operating system interference does not have significant effect on parallel application performance on Ethernet clusters with about 100 nodes. These results suggest that

Ethernet clusters have different performance issues than the high performance interconnect clusters used in previous work.

8.3 Additional overdecomposition experiments

This section summarizes unpublished experiment results.

The work was done in collaboration with Brian Vinter, Otto J. Anshus, and John Markus Bjørndalen.

8.3.1 Introduction

This section presents additional experiment results for overdecomposing parallel applications run on a WAN multi-cluster. Also three user-level schedulers are evaluated.

The paper in section 7.4 evaluated how overdecomposition can improve single parallel application performance on Ethernet clusters. The network latency we attempt to overlap with computation is larger on a WAN multi-cluster than on an Ethernet cluster. Larger overheads are easier to overlap for two reasons. First, a larger overhead increase is tolerated. Second, the time spent computing has increased relatively to the system call and context switch overheads.

User-level scheduling can easily be added to the communication system, since we are scheduling the threads (or the processes) of a single application. In the paper in section 7.4 we found that user-level schedulers could not significantly improve benchmark performance due to TLP limitations. But we also found that using a different synchronization variable implementation improved TLP. The improved TLP allows measuring performance improvements due to three-user level scheduling approaches designed for overdecomposed parallel applications.

The remaining of this section proceeds as follows. The experiment setup is presented in section 8.3.2. WAN results are presented in section 8.3.3. The design, implementation, and performance measurements of three user-level schedulers are presented in section 8.3.4. Section 8.3.5 concludes.

8.3.2 Methodology

For the experiments we used a cluster with 3.2 GHz Hyper-threaded Pentium 4 nodes, connected using Gigabit Ethernet. The Linux kernel version used was 2.4.26SMP with LinuxThreads. Version 3.3.3 of the gcc compiler was used. Hyper-threading (SMT) was enabled for all experiments. Additional details are provided in section 7.4.

To experiment with different WAN latencies and bandwidths we emulate WAN links between our clusters using the Longcut WAN emulator [42] (or section 8.1). The cluster is split into three sub-clusters. For each sub-cluster we select one host to act as a gateway. All communication to the sub-cluster is routed through its gateway, which adds delays to the routed messages to simulate the higher latency and lower bandwidth of a WAN TCP/IP connection. The delay for a given message size is calculated based on a latency and bandwidth trace collected by running an instrumented communication intensive application on hosts in Tromsø, Odense and Aalborg [42]. Table 15 and Table 16 show respectively the average latency and bandwidth between the nodes. For the experiments we emulated a topology with 14 nodes in Odense, 13 in Aalborg, and 17 in Tromsø.

The SOR kernel is used in the experiments. The problem size is scaled such that 50% of the execution time is spent communicating. The improvements reported in this section

are calculated using the mean execution time of ten experiments. The largest standard deviation was 6.1% of mean. However, for most experiments the standard deviation was less than 2% of mean.

	Aalborg	Odense	Tromsø
Aalborg		12.70 ms	36.98 ms
Odense	9.85 ms		34.02 ms
Tromsø	36.95 ms	26.49 ms	

Table 15: Average round trip latency in milliseconds between cluster sites in the emulated WAN multi-cluster topology.

	Aalborg	Odense	Tromsø
Aalborg		10.60 Mbit/s	3.19 Mbit/s
Odense	10.08 Mbit/s		3.46 Mbit/s
Tromsø	3.19 Mbit/s	4.75 Mbit/s	

Table 16: Average bandwidth between cluster sites in the emulated WAN multi-cluster topology.

8.3.3 WAN multi-cluster experiments and discussion

The execution time improvements are about 45% better on the WAN multi-cluster than on a single cluster, due to better computation-communication overlap.

The overheads are larger for the WAN experiment than for the LAN experiment, since the problem size increased to maintain a 50% communication-execution time ratio. But the overhead increase is smaller relative to the one thread per processor mapping (Figure 35).

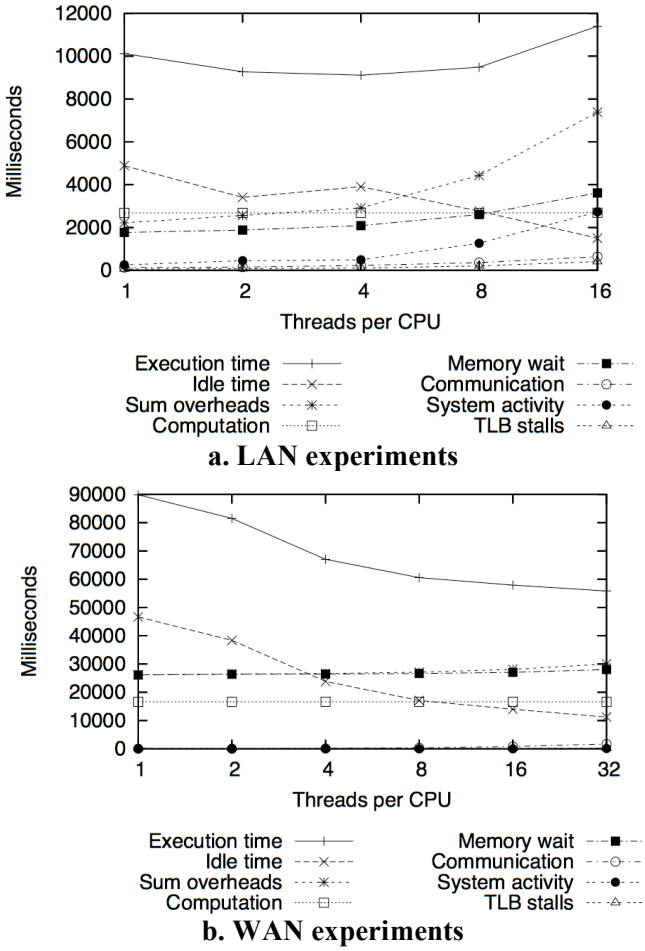


Figure 35: SOR execution time and estimated overheads for the Ethernet and WAN multi-cluster experiments.

8.3.4 User-level scheduler design and evaluation

User-level scheduling for overdecomposition can be implemented in a layer that intercepts communication operations calls using a similar approach as parallel application profiling layers (Figure 36). The interception can block a thread, or implement different scheduling decisions by controlling the release order of blocked threads. Compared to general-purpose schedulers, the implemented schedulers can take advantage of application knowledge, and ignore fairness since high priority threads will eventually block waiting for data from lower priority threads. Below we describe the design and implementation of three schedulers.

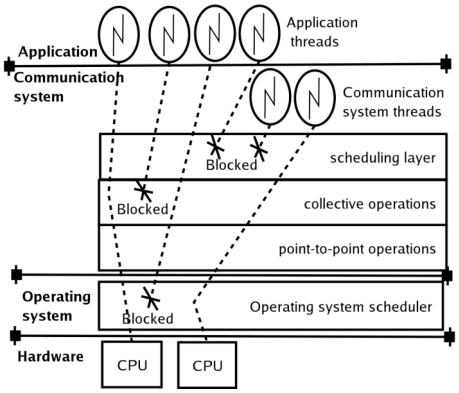


Figure 36: A user-level scheduler layer is added above the communication system layers and the operating system scheduler. Application threads and helper threads in the communication system can be blocked and unblocked at this layer.

8.3.4.1 Serialization

Cache pollution and the number of context switches may be reduced if threads are not interrupted during computation. The scheduling layer can serialize threads by only releasing one thread at a time. The simplest implementation for a multi-threaded application is to have a *global user-level lock* that must be acquired when returning from a communication operation call.

8.3.4.2 Priority Scheduling

The communication structure of some parallel applications may have dependencies dictating which messages must be received before a thread can proceed with its computation. For some benchmarks computation-communication overlap may improve if the threads sending messages over a high latency connection are scheduled to arrive first at the communication operations. This information can be used to set the priority of threads, and hence specify the computation order.

Threads are assigned a fixed priority at load time based on the applications communication pattern, and the mapping of threads to processors. Threads doing inter-node communication have the highest priority, while threads only doing intra-node communication have the lowest priority. A more advanced implementation could assign priorities based on measurements of communication time for all threads (using techniques described in Chapter 2). For each priority class the scheduling layer has a condition variable. Similarly to the serialization approach, the scheduling layer blocks threads upon return from communication operation calls, but the highest priority thread is released first to user-level.

8.3.4.3 Computation and communication coscheduling

Parallel application performance can be improved by coscheduling application threads and communication activity threads [59]. For example, if the operating system scheduler chooses to do computation before communication the latency of the communication operation may increase with tens of milliseconds.

Our coscheduling is similar to hybrid coscheduling [59], where the priority of communication threads is boosted when doing collective communication. However, our implementation is simpler since we use a communication system (PastSet [233]) with separate threads for collective communication activity. The scheduling layer prioritizes these by ensuring that application threads cannot return to user-level if there are collective operation threads to be unblocked.

Point-to-point communication activity is served immediately, since it is run in the context of an application thread. These are only blocked after returning from a communication operation call.

8.3.4.4 Experiment results and discussion

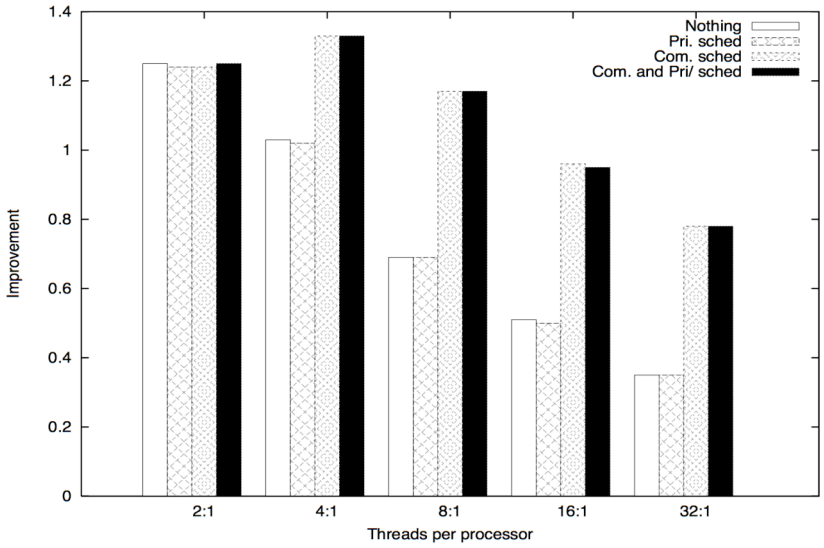


Figure 37: User-level scheduling performance improvements for SOR with allreduce run on an Ethernet cluster. All numbers are relative to the one thread per processor mapping.

Overdecomposition improves SOR performance up to 1.25 without user-level scheduling (Figure 37). Coscheduling computation and communication threads reduces communication wait time and improves TLP, resulting in a 6% speedup improvement. Serialization and priority scheduling of application threads does not significantly influence performance.

TLP is higher on the WAN multi-cluster, with at least two runnable threads in 42% of the execution time. Thus the priority scheduler can change the computation order, such that threads sending messages to other nodes arrive mostly first to communication operations. Performance improves with 6%. Further performance improvements were limited by the inability to overlap the allreduce operation with computation. Hence, coscheduling communication and computation threads does not improve performance. Using the conditional allreduce operation described in section 7.3, may improve the overlap.

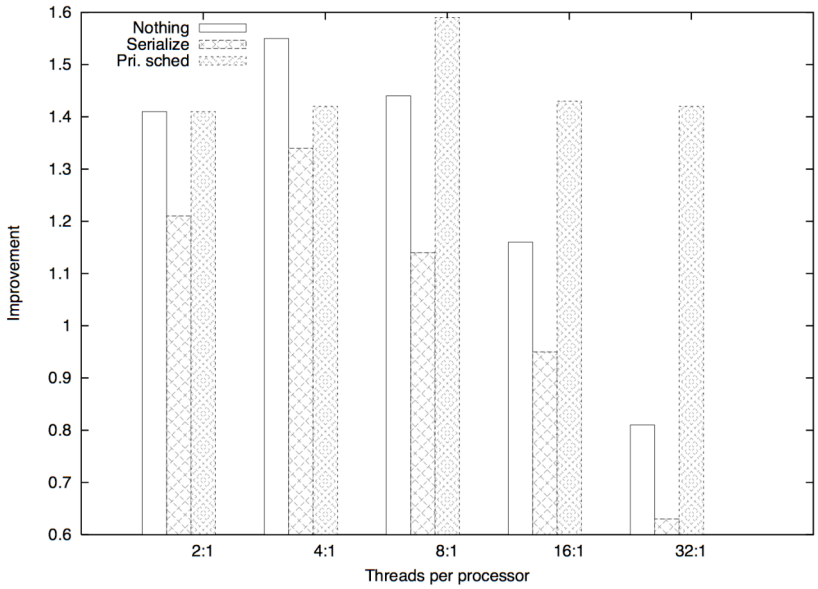


Figure 38: User-level scheduling performance improvement for SOR without allreduce run on an Ethernet cluster. The improvement is relative to the one thread per processor mapping.

The globally synchronizing allreduce operation used in SOR limits the potential for communication-computation overlap (as described in section 7.4). Removing the allreduce operation increases improvements due to overdecomposition up to 1.55, without user-level scheduling. Serializing computation did not improve performance. But priority scheduling application threads improved performance when more than eight threads were mapped to a processor. With fewer threads the scheduler was not able to change thread execution order, since most of the time few threads were runnable.

8.3.5 Conclusions

The achieved performance improvements on the WAN multi-cluster demonstrate that overdecomposition can be a useful technique for tolerating the high network latency of the WANs.

Three user-level schedulers were described, and performance measurements were done on an Ethernet cluster and a WAN multi-cluster. The impact on parallel application execution time is often limited by the lack of TLP, but a small improvement was achieved. Hence, we believe such scheduling is of limited use unless other changes are able to improve TLP.

8.4 Compression of Network Data Using 2-level Fingerprinting

This section presents an extended abstract that has not been submitted for publication.

The work was done in collaboration with Kai Li and Olga Troyanskaya.

Abstract. Previously proposed techniques for eliminating redundant network traffic are based on integrated anchoring and analysis of fine-grained one-dimensional data segments in data streams. The main limitation of such methods is that they require large segments in order to provide high compression ratio, and they do not work well with transferring multi-dimensional data such as 2D pixels in remote data visualization. This paper presents a method to identify and eliminate redundant data transfers of complex data types over a network. Our method is different from the previous approaches in four ways. First, the method separates data segmentation from redundancy elimination such that specific content-based segmentation methods can apply to complex data types. Second, we propose a 2-dimensional segmentation approach that works well with remote data visualization data transfers. Third, we employ a two-level fingerprinting method to optimize the encoding of unique data segments. Fourth, we propose a large segment cache on disk that improves redundancy detection by examining a larger scope.

8.4.1 Introduction

Current scientific instruments and simulations are creating peta-scale data volumes, and the amount of data produced is roughly doubled each year [94]. Examples include the Sloan Digital Sky Survey (SDSS) astronomical survey [201], the BaBar high energy physics experiment [21], the Entrez federated health sciences database [158], and the CERN Large Hadron Collider [56].

The amount of data stored, and the computation necessary for analyzing the data requires building a data storage and analysis infrastructure. The infrastructure may be used to access the data by thousands of scientists participating in a project working at hundreds of institutions. A distributed infrastructure has several advantages including no single point of failure, and load balancing of data, computation, and user support [57]. In addition the different parts of the infrastructure can be individually funded by the participating organizations.

A main challenge for such a distributed infrastructure is providing the necessary bandwidth between the resources. In particular gigabytes of scientific data must be reliably moved over wide area networks, and scientist must remotely interact with applications used to analyze the data at a remote site. Compressing the network data can reduce the bandwidth requirements for such data movement and remote visualization.

Network data is typically compressed using a local compression algorithm [9, 188, 202] which decouples compression from decompression.. A popular local compression algorithm is DEFLATE [73], used in the zlib/gzip library[9]. DEFLATE combines the Lempel-Ziv (LZ77) duplicate string elimination algorithm [244], with Huffman encoding for bit reduction [103]. LZ77 detects duplicate strings and replaces these with a back-reference to the previous location of the string. Huffman encoding replaces symbols with weighted symbols based on frequency of use. The problem with existing local compression algorithms is that they only detect redundancy within a local scope, such

that the ratio achieved for scientific data is low, while compression time is to high for remote visualization.

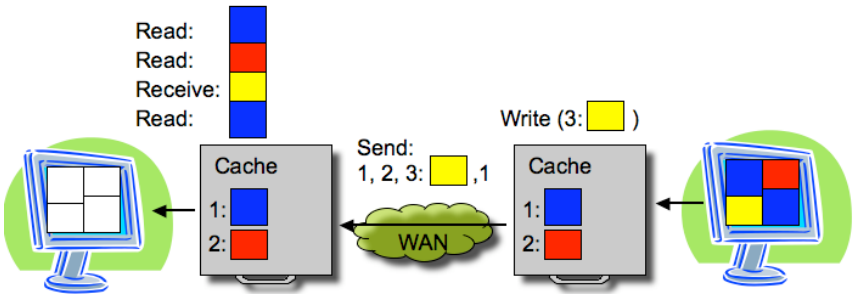


Figure 39: Global compression used to compress screen content. Previously sent segments are stored in cooperating caches at the sender and receiver side. The data to be sent is segmented, and in place of replicated segments only the cache index is sent over the WAN.

During the past few years, global compression has been proposed to eliminate redundant network traffic data [156, 209]. The sender and receiver cooperate to maintain a shared cache of previously sent data. To eliminate transfer of redundant bytes, the sender divides the data to be sent into segments, and sends fingerprints instead of replicated segments over the network. The receiver uses the fingerprints to retrieve the data from its cache (Figure 39).

Global compression ratio is limited by data redundancy and the segment size to fingerprint size ratio. Using smaller segments improves the redundancy found, but requires using smaller fingerprints to maintain a high compression ratio. However, to ensure data consistency the fingerprint size must be large enough to uniquely identify a segment. Previous global compression systems [39, 69, 72, 100, 153, 156, 171, 172, 222] typically chose a secure hash, such as 160-bit SHA-1 [7], as a fingerprint so that the probability of a fingerprint collision can be lower than a hardware bit error rate. But this also required using segments of several kilobytes in size.

Our approach to this problem is to propose a new framework that allows application users to build content-aware anchoring mechanisms to significantly improve the network data compression. We propose a two-level fingerprinting method to further improve encoding for fine-grained data segments, and a prototype system to show the proposed methods are effective.

This paper makes three contributions:

- A novel two-level fingerprinting protocol that improves redundancy detection by using smaller segments, while maintaining data consistency. Past work used large fingerprints, and hence required large segments to maintain a good compression ratio.
- The design and implementation of a very large cache on disk for storing previously sent segments that improves compression ratio. Most previous systems stored segments in memory only.

The requirements for a 100 GB data set were modeled. Our results shows that two-level fingerprinting is most useful for segment sizes ranging from 16 to 256 bytes. In order to

get the best trade-off between fingerprint bytes, and collision bytes the optimistic fingerprint size should be 40 bits, and a conservative fingerprint should cover about 20—25 segments. In addition we demonstrate the need for a large segment cache.

8.4.2 Proposed approach

We propose a network data compression framework that solves the problem of compression ratio being limited by having to use a long fingerprint to represent a data segment in the compression protocol. This section first describes the architecture of the proposed approach and then a two-level fingerprinting method to compress fine-grained data segments.

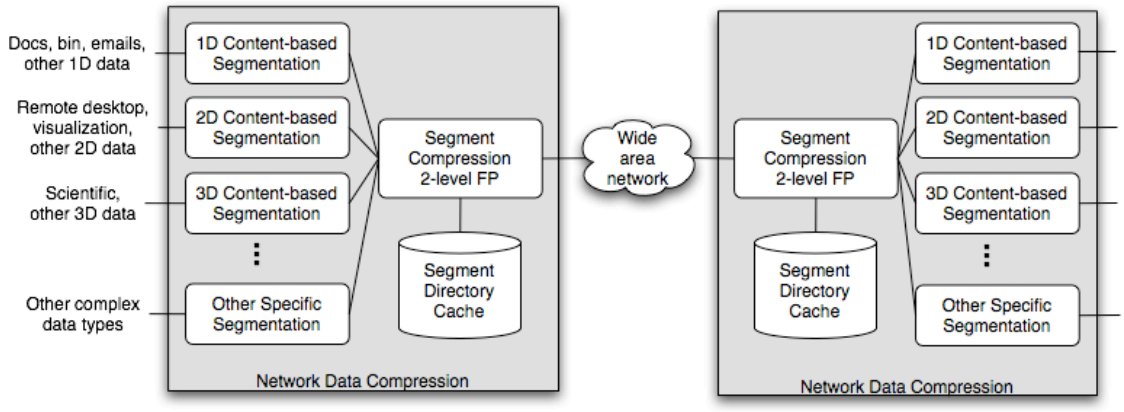


Figure 40: Architecture of proposed compression approach, consisting of components for context-aware segmentations, redundant segment elimination with two-level fingerprinting, and segment directory cache. Applications can choose their appropriate content-based segmentation method according to their data type.

8.4.2.1 Architecture

The architecture of the proposed framework for network data compression engine, as shown in Figure 40, includes multiple, data-specific segmentation methods and a shared segment compression engine.

The key idea of the architecture is to make segmentation methods data specific. The segmentation methods can be configured to one or more ports of the system and to support a variable number of data streams of different data types. Each content-based segmentation component is responsible for the segmentation of a specific class of data. For example, text documents, emails, binary executables, and other one-dimensional data can use a content-based segmentation component using the Manber/ LBFS segmentation method [145, 156], whereas a remote terminal application or a remote collaborative data visualization application can use a specific content-based segmentation component that can anchor 2D segments based on screen pixel contents [49].

The content-based segmentation component implements the segmentation mechanisms for both send and receive data. For send data, it anchors its input data stream into segments and passes them to the segment compress component. For receive data, it assembles segments into a data stream.

The segment compression engine is responsible for fingerprinting and caching data segments from the segmentation modules. It maintains a segment directory cache to eliminate redundant segments. The basic algorithm is:

- The *sender* computes a fingerprint for each incoming data segment, and sends the fingerprint to the receiver.
- The *receiver* uses the fingerprint as the segment's identifier to look in the segment cache to see if it has received this segment previously.
- If there is an entry in the segment cache for the given fingerprint, it retrieves the segment of the fingerprint from the cache and passes it to the segmentation component to assemble into a data stream.
- If the fingerprint is not in the cache a segment request is sent to the sender.
- When the *sender* receives a segment request, it compresses the segment with a local compression method, and sends the segment to the receiver.
- The *receiver* decompress the segment with a corresponding local decompression algorithm, inserts the data segment to the segment cache, and pass the segment to the segmentation component to assemble into a data stream.

This basic algorithm is straightforward and its high-level idea is similar to previous studies on using fingerprints (or secure hashes) as identifiers to avoid transfer of redundant data segments. The key difference is that in the previous studies detected redundancy is limited by the large fingerprints necessary to uniquely represent data segments, which require large segments to achieve a good compression ratio. We address the issue of how to represent fine-grained data segments. Below, we present a 2-level fingerprinting method to compress fine-grained segments to optimize the basic algorithm.

8.4.2.2 Application specific segmentation

The segmentation modules main goal is to divide the application data into segment that are likely to be repeated throughout the data. Since such the segmentation methods are very data and application specific, we have separated the segmentation module from the Canidae subsystem to allow applications to use their own specialized segmentation algorithms. A segmentation module implements three tasks: segmentation, reassembly, and protocol handling. How these tasks are implemented depends on the data type and which segmentation algorithm is used, but in general the following is done:

- The input data stream is copied to a local buffer. This may include parsing the input data stream to extract the application data from the meta-data such as protocol headers.
- A segmentation algorithm is run when the buffer is full or the application protocol requires data to be sent. This algorithm divides the data into segments. The segmentation can either be static or based on the buffer content.
- The segments and the meta-data necessary to reassemble the segments is sent to the compression engine.

- On the receiver side, the segments are received from the compression engine, and the meta-data is used to reassemble the segments. The segment data is then copied to the output stream.

For multi-dimensional data the buffer is a multi-dimensional data structure, and the resulting segments may be multi-dimensional. Not all messages types have to be parsed, since the data is typically sent using a few message types. The remaining non-data carrying messages can be forwarded unparsed.

8.4.2.3 Two-level fingerprinting

An important design issue in using fingerprints as identifiers to detect previously transferred data segments is the size of a fingerprint. Therefore previous global compression systems typically use a 160-bit such as SHA-1 [7], or even longer secure hash, as a fingerprint so that the probability of a fingerprint collision is far lower than a hardware bit error rate. But, since the global compression ratio is limited to the ratio of the average pixel segment size to the fingerprint size, the larger the fingerprint size, the smaller the compression ratio.

To maximize the global compression ratio and maintain a low probability of fingerprint collision, we propose a two-level fingerprinting strategy. The two-level fingerprinting views data as groups of segments. For each group of segments, a 160 bit SHA-1 hash is computed as the *conservative fingerprint* of the whole group. For each segment in the group, we compute a 40-bit FNV hash [86] as the optimistic fingerprint.

The two-level fingerprinting protocol extends the basic protocol described above. To send a group of data segments, the sender sends short optimistic fingerprints for all data segments, and the conservative fingerprint for the group. The segments are also added to the senders segment cache. Upon arrival, the receiver buffers the whole group of segments, re-computes a conservative fingerprint using the same hash function as the sender, and compares it with the received conservative fingerprint. If the two conservative fingerprints are identical, the receiver sends an ACK message back to the sender. If they are different, it requests all segments read form the cache from the sender, and only sends the ACK once all requested segments have been received. When the ACK message have been sent, the receiver updates its segment cache with the group of segments and send all segments over to a segmentation component such that the segments can be assembled and sent to the receiving application.

An important design decision is to choose the sizes of the conservative and optimistic fingerprints, and the number of optimistic fingerprints per conservative fingerprint. The conservative fingerprint should be long enough so that the probability of a collision is far smaller than a hardware error. The optimistic fingerprint should be short enough to maximize the compression ratio of network data, but long enough to minimize the events of resending groups of segments. Also, the number of optimistic fingerprints should be high enough to maximize the number of fingerprints and low enough to keep the probability of collision low.

8.4.2.4 Segment cache

The basic operation of the segment cache is to read and write segments based on their optimistic fingerprint. The two main design goals for the segment cache are to make it large enough to hold all previously sent segments in a session, and fast enough not to limit the throughput of the compression pipeline. For a hundred gigabyte dataset, the total size of cached segments exceeds main memory size, such that segments must be stored on disk. In addition, an index is required to map optimistic fingerprints to the segments location on disk (or in a memory cache).

The naïve approach of using a large hash table in memory as the index, and storing all segments on disk has two problems. First, the memory size limits the maximum number of segments that can be indexed by a single hash table resident in memory. Second, most segment accesses requires reading segments from disk since all available memory is used for the hash table. Therefore, the index should be split into multiple parts that can be stored in disk, and a large portion of the memory should be used to cache segments.

We propose using multiple small hash tables; each indexed using the first l bits of the fingerprint. Hash table entries are 64 bits, and contains the remaining fingerprint bits, the memory or disk offset of the segment, and the size of the segment. The hash table, and the segments indexed by it are stored in a *container*. Each container is stored in a separate file on disk, but can also be cached in memory.

Segment accesses have no spatial locality with respect to fingerprint values, since the hashing function generates random fingerprints for segments. Segments can therefore not be efficiently cached if they are distributed to containers based on their fingerprint values. Instead we exploit the observation that segments written to the cache at the same time tend to be read together. Therefore, all new segments are written to the same container by inserting the fingerprint to the hash table and appending the segment to the end of the segment buffer. In case of a hash table collision the segment is written to the next container in memory. This clustering of segments allows read-ahead of segments from disk. The disadvantage of this approach is that a linear search is required to find the container containing a specific segment. Therefore, we propose multiple optimizations to reduce the number of containers on disk that has to be checked.

Segments accesses have temporal locality, hence recently accessed containers are cached in memory. When a container is accessed, the entire hash table is always read to memory, but the segment buffer is divided into several chunks, which are read on-demand from disk (similar to demand paging [127]). Writes are buffered such that modified segment chunks are only written to disk when the memory becomes full. To evict segment chunks or containers, we use a least recently accessed algorithm.

To further reduce disk accesses we use a Bloom filter [37]. A Bloom filter is a space efficient probabilistic data structure that we use to test whether an optimistic fingerprint is a member of the set of optimistic fingerprints stored in the segment cache. The test may return a false positive; hence an optimistic fingerprint in the Bloom filter may not be in the segment cache requiring all hash tables to be checked. But false positives are not possible. Therefore in case of a miss, it is not necessary to check the containers before requesting a segment from the sender, or writing a segment to the cache.

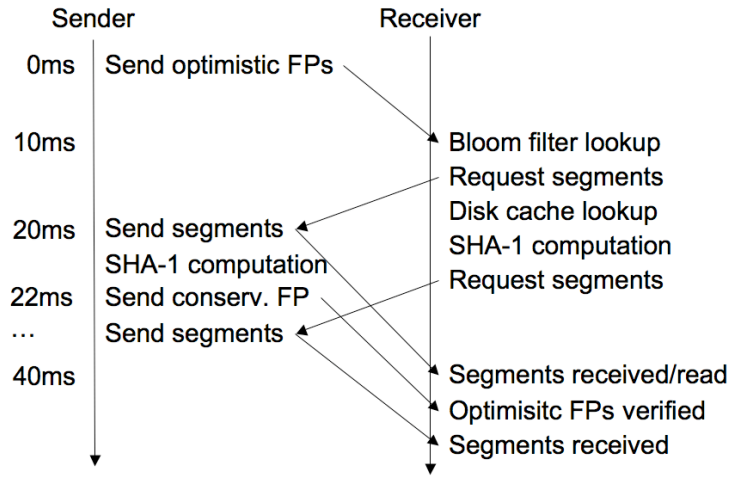


Figure 41: Timeline for an update operation, were the network latency and segment transmission times are both assumed to be 10ms. The disk lookup time is overlapped with the time to send and receive segments not in the segment cache.

In addition to reducing the number of disk accesses, the Bloom filter can also be used to overlap network transmission time with disk accesses (Figure 41). For a group of segments the receiver checks the Bloom filter, and sends request messages for the segments not in the Bloom filter. Disk lookups for other segments in the group can then be overlapped with the time to send and receive the non-cached segments.

8.4.3 Protocol and system implementation

We have implemented the 2-level fingerprinting protocol proposed in section 8.4.1 in a system called *Canidae*, which is designed to support multiple simultaneous hardware and software clients from each Canidae installation. It consists of multiple segmentation components and a generic compression sub-system that handles the fingerprinting, transmission and caching of segments.

The compression engine and segment cache are implemented as a server and run on a dedicated machine. The server is designed to use all available memory, and hundreds of gigabytes of disk storage. To avoid dynamic memory and storage resource management all share the same compression component. Using multiple compression components, and dynamic resource management will complicate the system but probably neither improve compression ratio, nor improve the performance of the system.

To establish a connection over Canidae the application server connects to a segmentation component instance on a predefined port. Applications then communicate with Canidae using their usual communication protocol. On the receiver side, the application clients connect to their Canidae server, which creates and initializes a connection to the application server site.

The main goal of Canidae is to improve bi-directional network transfer time between sites where Canidae servers are installed. This is achieved by using the 2-level fingerprints protocol that saves bandwidth, but provides in-order strict-consistency message delivery for applications. However, the server is implemented to utilize out-of-

order messages, eventual consistency, and to allow parallel compression and decompression of segments.

In this section we describe the protocol and address implementation implications. Although Canidae supports bi-directional compression and multi-cast, we will below for clarity usually assume that communication is uni-directional and point-to-point.

8.4.3.1 Segmentation components and segment protocol

The segmentation components are either integrated with the Canidae server, run as a separate process on another machine, or integrated with the application. The *segment protocol* is used for encoding segments sent by the segment component to the compression engine. If both are run in the context of the same process, the segment component can send segments by calling functions in the compression component.

Space efficient encoding of segment protocol messages is not necessary, since these are only sent over a high bandwidth local area network. However, efficient encoding is necessary for the meta-data used by the receiving segmentation component to assemble the segments. The meta-data is sent uncompressed during the two-level fingerprinting protocol and its size will therefore limit the achieved compression ratio. For 1-D bytestreams only the meta-data may only specify the segment size, but for a multi-dimensional data sets the position in the data-structure may also have to be specified. But with careful design, only 1—4 bytes are required.

8.4.3.2 Two-level fingerprinting parameters

The main challenges in implementing the compression sub-system is choosing appropriate fingerprint sizes for the 2-level fingerprint algorithm, and choosing the number of optimistic fingerprints covered by conservative fingerprints, and implementing an efficient caching mechanism.

The optimistic fingerprint should be small enough to provide good compression ratio, but still large enough to have few collisions even for large data sets. The factors influencing the optimistic fingerprint size are:

- The data set size. The number of unique segments, and hence the fingerprint collision probability increases with data set size.
- Data redundancy. With higher redundancy, fewer segments are stored in the cache, and hence the collision probability is reduced.
- Segment size. Reducing the segment size increases the number of segments sent and stored in the cache. Thus, the collision probability increases.
- Optimistic fingerprints covered by a conservative fingerprint. A single optimistic fingerprint collision requires sending all segments read from cache, thus increasing the collision penalty.

In this section we answer the following questions:

1. For which segment sizes is compression ratio limited by the fingerprint size?
2. Where is the crossing point for when the number of bytes sent due to collisions is larger than the fingerprint bytes?

3. How many segments per conservative fingerprint give the best compression ratio?

Parameter	Value	Parameter explanation
S	100GB	Data set size
R	75GB (75%)	Data redundancy found
K	40 bits	Optimistic fingerprint size
L	160 bits	Conservative fingerprint size
P	20	Segments per conservative fingerprint
S	32 bytes	Segment size

Table 17: Default parameters used to model two-level fingerprint compression ratio.

Canidae is designed to compress very large datasets, so for the analysis we set the data set size to 100GB. Segment size is conservatively set to 32 bytes for which 20-byte fingerprints gives some compression. The optimistic to conservative fingerprint ratio is set to 20, since it usually gives high compression ratio (as shown below). Finally, the redundancy is set to 75%, such that the maximum compression ratio is 4.0. The parameters are summarized in Table 17.

$$compression\_ratio = \frac{S}{(S-R) + (\frac{k}{8} + \frac{l}{8p})\frac{S}{s} + ps \sum_{i=1}^{i=(\frac{S-R}{s})} \frac{i}{2^k}}$$

Equation 2: Formula for modeling compression ratio achieved using two-level fingerprinting.

To find the best parameters for the two-level fingerprinting protocol giving the best compression ratio, we model the number of segment bytes sent, the number of fingerprint bytes sent, and the number of collision bytes sent. We assume that the dataset is segmented into *S/s* segments, where *S* is the data set size, and *s* is the average segment size. To find the number of fingerprints bytes sent we multiply the number of segments with $(\frac{k}{8} + \frac{l}{8p})$, where *k* is the number optimistic fingerprint bits, *l* is the number of conservative fingerprint bits, and *p* is the number of segments per conservative fingerprints. The optimistic fingerprints should be random, and hence the probability of a new fingerprint having the same value as an existing fingerprint in the segment cache is given by $\frac{n}{2^k}$, where *n* is the number of segments in the segment cache, and *k* is the number of bits in the fingerprint. To estimate the number of bytes sent due to collisions, we find the number of collisions for all cache writes. Then, we assume that each collision causes one group of segments to be resent. Putting it all together gives the formula in Equation 2. Below we compare the achieved compression ratio to one-level fingerprinting using 20 byte fingerprints.

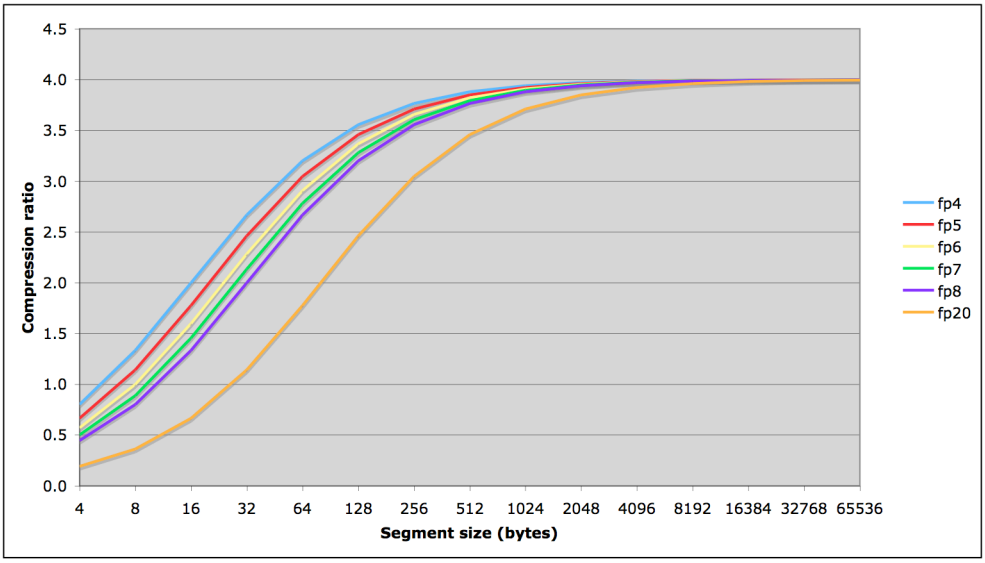


Figure 42: Compression ratio for different fingerprint and segment sizes. Data redundancy is 75% and collision bytes are ignored.

To find the segment sizes that are limited by the fingerprint size, we plot the compression ratio achieved for different segment size (Figure 42). We find that the fingerprint size significantly limits compression ratio for segments less than 1 Kbytes if the data has 75% redundancy, but even for smaller segments if the detected redundancy is lower.

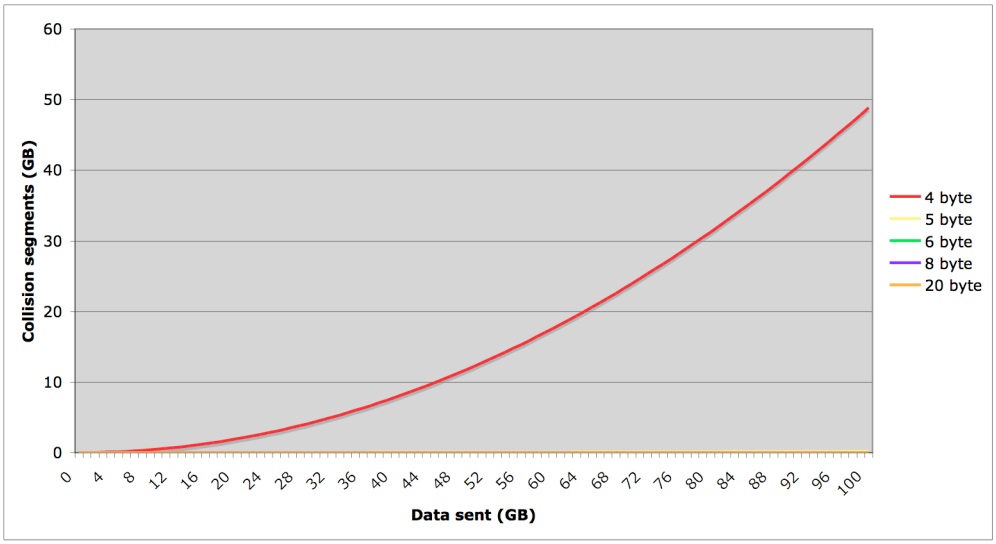


Figure 43: Miss penalty bytes sent for different optimistic fingerprint sizes.

Choosing the conservative fingerprint size is relatively straightforward; it should be large enough to guarantee a collision rate smaller than the hardware error rate. Since 2^{160} is considered sufficient for data sets up to an exabyte in size [172], we use 160 bit SHA-1 hash values as conservative fingerprints.

The optimistic-fingerprint size is more challenging to choose because it affects two competing trends. Reducing the optimistic-fingerprint size will increase the maximum achievable compression ratio, but simultaneously increase the number of cache collisions

that require entire segments to be resent. So we want to choose an optimistic-fingerprint size that is near the inflection point of the competing trends and that works across the many data types being transmitted.

If a 4-byte optimistic fingerprint size is chosen, then 50 GB of segment data will be sent due to collisions when transferring a 100 GB data set (Figure 43). Increasing the optimistic fingerprint size to 5 bytes reduces the total number of bytes sent since the data sent due to collisions is reduced to 0.2 GB, while the increase in fingerprint bytes is only 6.1 GB. Only when the data set size is less than about 35 GB, does 4 byte fingerprints give the best compression ratio. Further increasing the fingerprint size to 6 bytes does not improve compression ratio since the reduction in collision bytes (about 0.2 GB) is much smaller than the increase in fingerprint bytes (6.1 GB).

For a smaller data set, the compression ratio will improve if 4 byte fingerprints are used. The optimistic fingerprint size could be dynamically set at server startup time. But it is not possible to increase the optimistic fingerprint size without flushing the segment cache, or re-computing the optimistic fingerprint for all cached segments.

The number of segments covered by a conservative fingerprints should be chosen such that the fingerprint bytes sent remains low, while keeping the bytes sent due to collisions low. With the default parameters in Table 17, the minimum number of bytes sent is for 22 segments per conservative fingerprint (Figure 44). Typically a ratio of 20—25 gives a good compression ratio, even if the segment size, redundancy ratio, or data set size is changed.

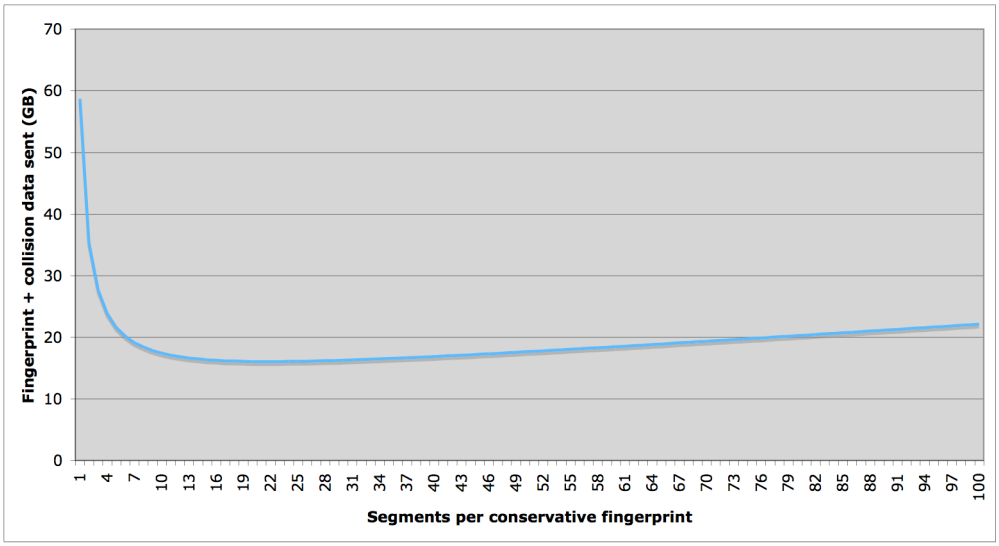


Figure 44: Bytes sent for fingerprint and collisions, when the number of segments per conservative fingerprint is changed.

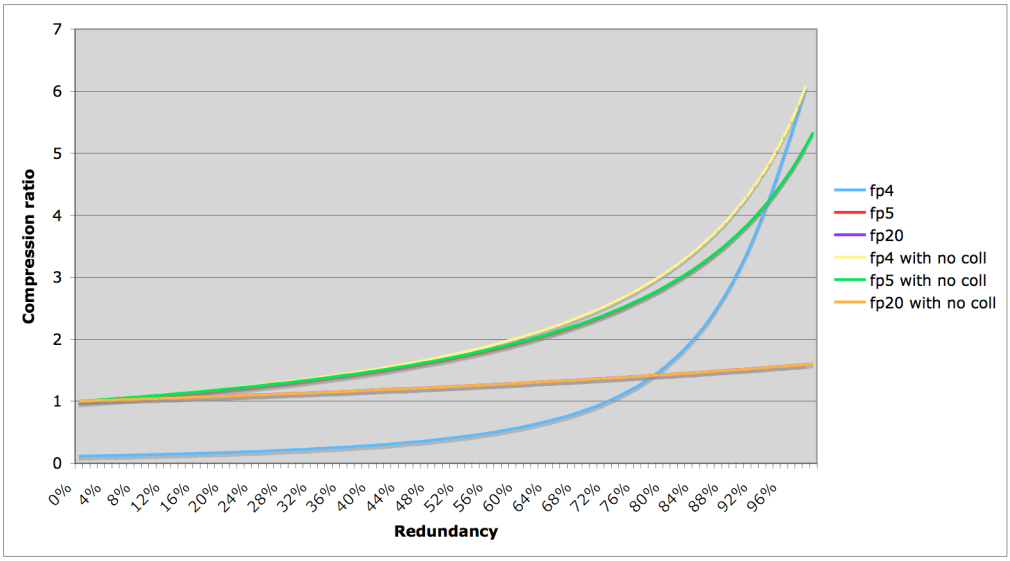


Figure 45: Compression ratio for different redundancy levels when using 4 byte, 5 byte and 20 byte fingerprints. The 5 byte fingerprint compression ratios with and without collisions are almost identical, and identical for 20 byte fingerprints that have no collisions.

Above we have assumed that the redundancy level is 75%. Figure 45 shows that using 5-byte optimistic fingerprints with 20 segments per conservative fingerprint, gives the best compression ratio for most redundancy levels. Only if redundancy detection is lower than 15%, or higher than 95%, is the best compression ratio achieved using other optimistic fingerprint sizes.

8.4.3.3 Two-level fingerprint protocol messages

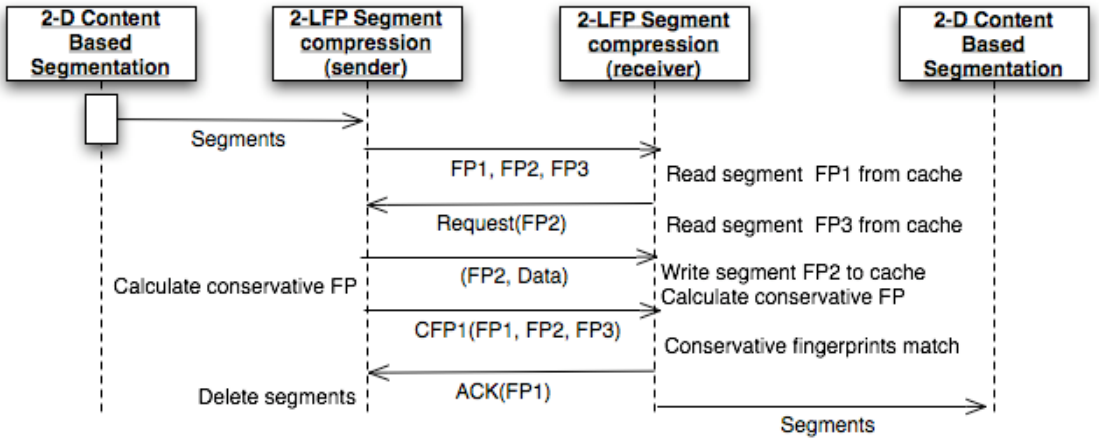


Figure 46: Two-level fingerprinting messages (FP*i* is an optimistic fingerprint message, and CFP*i* is a conservative fingerprint message).

Message type	Size (bytes)	Comment
Optimistic fingerprint	6 + M	M is implicitly set by the message type
Segment request	5	A 4 byte sequence number identifies the segment
Segment	7 + S	Includes the segments sequence number and size (2 bytes)
Conservative fingerprint	21	
Conservative fingerprint ACK	1	No sequence number since the ACKs are sent in the same order as conservative fingerprints
No-fingerprint segment	3 + S	Includes the segment size (2 bytes)
Multiplexing message	3	2 byte are used to identify the segmentation component that should receive the next batch of segments

Table 18: Two-level fingerprint messages. M is meta data size, and S is segment data size. Optimistic and conservative fingerprint sizes are respectively 5 and 20 bytes.

The two-level fingerprinting messages have been designed to use as few bytes as possible, since additional bytes sent reduce compression ratio (Table 18). For each message, the first byte is used to identify the message type.

The most frequently sent message is the *optimistic fingerprint* message. For these messages, the message type implicitly specifies the meta-data size (it can be from 0—12 bytes).

Each segment is assigned a sequence number that is set by the sender when the optimistic fingerprint messages have been sent, and by the receiver when the message is received. *Segment request*, and *segment* messages must include the sequence number, since it is used respectively to identify the requested segment, and to match the segment data to meta-data sent with the optimistic fingerprint message.

The *conservative fingerprint* message is always sent immediately after the last optimistic fingerprint message in a group. It is therefore not necessary to add any information to the message about which segments are covered, and thus the message only contains the message type and conservative fingerprint. *Conservative fingerprint ACK* messages are always sent in the same order as the conservative fingerprints were received, and therefore adding a sequence number is not necessary.

Segments are addressed to segmentation components by inserting a *multiplex* message between the fingerprint and segment messages. A multiplex message specifies the segmentation component that should receive the next batch of segments. Multiplex messages are used since adding addressing information to optimistic fingerprint messages would reduce the compression ratio. We assume the overhead of multiplex messages is small since segmentation components typically sends large burst of data.

The remaining messages types are the *no-fingerprint* message used to send data that should not be stored in the segment cache, and messages sent during connection initialization and closing.

8.4.3.4 Data consistency

Computing and comparing conservative fingerprints gives a very strong guarantee that all segments read from the cache are identical to the segments that were not sent over the network. However, in order for the fingerprints to match, the secure hash function on both sides must be run over the segments in the same order. The segments are ordered using the sequence numbers added to optimistic fingerprint messages as described above. For simplicity the conservative fingerprint is calculated for all segments even when the segment is sent to the receiver.

The two-level fingerprinting protocol allows disk cache reads, and segment requests to be issued out of order, while maintaining consistency and in order delivery of segments. To further overlap network latency and transfer time with computation, Canidae implements additional optimizations. First, optimistic fingerprints for the next group can be sent before the conservative fingerprint ACK message for the previous group has been received. Second, the receiver streams the conservative fingerprint computation, such that the fingerprint is continuously updated when requested segments are received, or segments have been read from the segment cache. Third, multiple segment cache reads can be issued in parallel.

8.4.3.5 Bidirectional communication and multiple clients

To support bidirectional communication, a separate instance of the protocol is run for each direction with one side acting as sender and the other as receiver. The sender writes all sent segments to its segment cache, allowing redundancy to be detected for both directions, since the same segment cache is used. Any inconsistencies in the segment cache are detected during conservative fingerprint comparison.

To send the same data to multiple clients, Canidae use multi-cast. The two-level fingerprinting protocol is not modified, but the segment protocol needs to be changed such that recipients for each segment are specified. Since the receiver specification is only sent to the local Canidae server, an efficient address encoding is not necessary. In the current implementation we use a bitmap with bits set for the receivers of a segment. The advantage of multi-cast over multiple point-to-point messages is that optimistic fingerprint computation, and local compression is only done once. Conservative fingerprints must still be computed for each receiver, since all may not receive the same set of segments. The advantage over broadcast is that the amount of data sent to each client can be different, and hence the total amount of data sent is not limited by network with the lowest throughput.

Canidae servers can be connected to multiple other Canidae servers, but many-to-one communication operations are not supported. All connections are handled by the same compression component, and all share the same segment cache. It is therefore possible for a server to have received data to be sent from another server. The conservative fingerprints takes care of all consistency issues.

8.4.3.6 Compression pipeline

The server is implemented using a multi-threaded event based model. The protocol handling is divided into several stages. The stages are connected using queues that are

used to store segment objects to be processed by the next stage. In addition some stages are either read from, or write to a socket. To support multicast, some stages produce output destined to several stages.

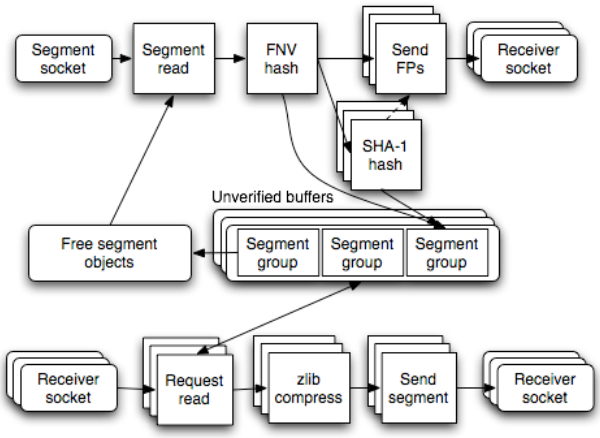


Figure 47: Stages and data structures used in the fingerprint components send path.

The compression pipeline consist of a send path and a receive path. For clarity we describe these separately.

There are multiple independent send paths (Figure 47). The path for sending optimistic fingerprints consists of: (i) a data *read* stage that reads segments from the segment component socket, (ii) a *FNV hash* stage that calculates the optimistic fingerprint for the segment, (iv) a *SHA-1 hash* stage for each client that computes the conservative fingerprint for a group of segments, and (v) a *send fingerprint* stage for each client that write optimistic fingerprint message to the clients socket.

The path for handling segment requests consist of a *request read* stage that reads segment request message from multiple receiver sockets. The *zlib compress* stage that does local compression, and the *send segment* stage that writes compressed segment to the receiver socket. In addition for bidirectional communication the send path is extended with a stage for writing segments to its segment cache.

There is one segment object for each segment, even if it is going to be sent to multiple receivers. The memory allocated for the segment object is reused to avoid allocating and freeing memory for each sent segment. Also, to simplify (memory) resource management the number of segment objects is statically set, by allocating memory for the segment objects and adding all to the *free queue*, to which segment objects are also added when the reference count becomes zero. In addition there is a buffer for storing a group of segments until the receiver has acknowledged the group's conservative fingerprint.

The two-level fingerprinting protocol does not implement rate limitation. But, only a limited number of segments are buffered while waiting for a conservative ACK. If the buffer is full no more segments are inserted into the compression pipeline. Additional rate limitation must be implemented either by segmentation components or the application.

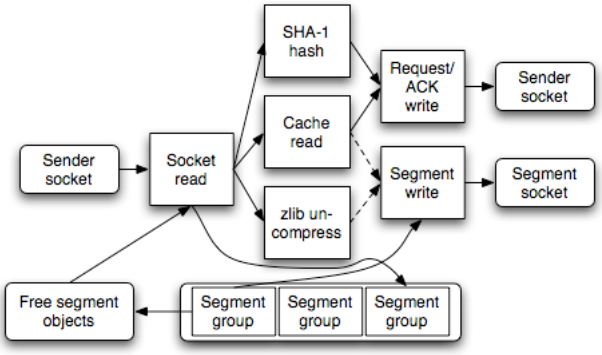


Figure 48: Stages and data structures used in the fingerprint components receive path.

The receive path uses the same data structures as the send path (Figure 48). The *socket reader* stage reads messages from the sender socket. Depending on the message type, either the *SHA-1 hash* stage updates a conservative fingerprint, the *cache read* stage reads a segment from the cache, or the *zlib un-compress* stage uncompress a received segment. Then the *request/ ACK write* stage either sends a segment request if the segment was not in the cache, or a conservative fingerprint ACK message. Finally, when all segments have been read from the cache, or received from the sender, and the conservative fingerprints match, the *segment write* stage writes the group of to a socket read by the segmentation component.

Both the send and receive path are implemented to exploit parallelism for respectively reducing compression time and overlapping disk accesses with computation. Send path parallelism is most useful for scaling the number of receivers, since message ordering and conservative fingerprint computation require all stages except *FNV hash* and *zlib compress* to be run sequentially. But, the stage instances for multiple receivers can be run in parallel.

8.4.4 Segment cache

A hash table and the buffer storing the segments indexed by the hash table form a *container*. We assume that segments are accessed with temporal and spatial locality with respect to segment creation time. Each container is stored in a separate file on disk. The entire hash table is always read to memory, but only parts of the segment buffer may be in memory as explained below.

A hash table entry is 64 bits and contains the segments optimistic fingerprint, the segment buffer offset, and the segment size (Figure 49). The first *i* bits of an optimistic fingerprint are used to index the hash table. The remaining fingerprint bits are compared with the ID bits stored in the hash table entry. If they match the segment can be located in the segment buffer using the segment offset and size stored in the hash table entry. The offset may be multiplied with a constant to allow a larger segment buffer to be used.

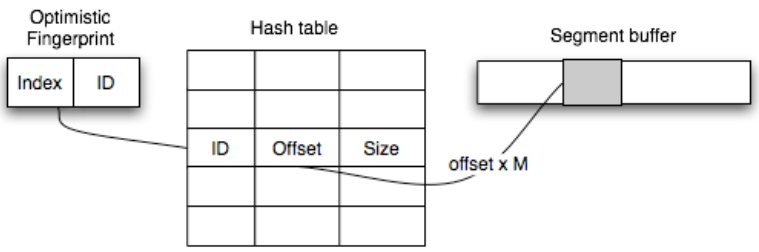


Figure 49: A container consists of a hash table used to map fingerprints to segments in the containers segment buffer.

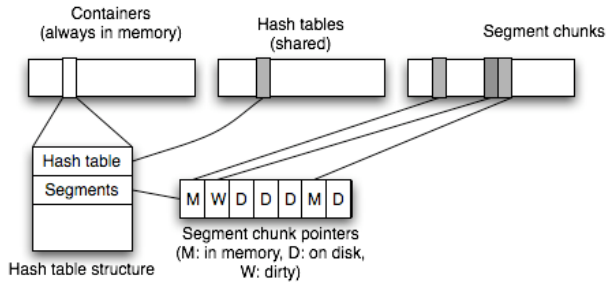


Figure 50: Container data structures.

Container meta-information is always stored in memory. This data structure consists of a handle for the file where the container is stored, a pointer to the hash table (if in memory), a small bitmap of which hash table entries are in use, a table of segment chunks currently in memory, and the timestamps required by the container replacement algorithm (Figure 50). The size of the hash table depends on the number of entries. We have set it to 512KB. Segment chunks are always 1MB.

Hash tables and segment chunks are stored in two large arrays with a static size. If one of the arrays becomes full, a container is evicted (even when only a single segment chunk is required). To select a container a simple working set replacement algorithm is used that selects the least recently accessed container for eviction. The algorithm is implemented by maintaining a timestamp for each container that is updated each time a segment is read from, or written to the container. Since the time to write a container to disk is large, and the number of containers in memory is relatively small, all containers are scanned linearly to find the container with the smallest timestamp.

8.4.4.1 Operations

The segment cache supports three operations: read, write, and update.

The most important design goal for read operations is to reduce disk accesses. Therefore, the Bloom filter is first checked. In case of a miss, the segment is not in the cache and the operation can return. Otherwise the hash tables are linearly checked starting with the hash tables already in memory, and then the hash tables on disk. The search terminates when the segment is found and returned, or all hash tables have been checked.

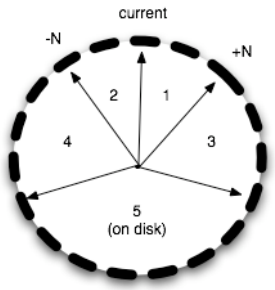


Figure 51: For segment writes the last accessed (current) container is first checked. If there is a hash table collision, writes are attempted to the N subsequent containers, and then the N previous containers. If all collide, writes to the remaining containers in memory are attempted, before the containers on disk.

Write operations are implemented to cluster segments such that spatial locality can later be utilized. First, the segment write is attempted to the container last accessed. If the fingerprint maps to an entry in use, the write is attempted to hash tables in memory, and then to hash tables on disk as illustrated in Figure 51. Finally, if an unused entry is still not found a new container is created. The segment data is always appended to the end of the containers segment buffer.

The write operation does not check the Bloom filter, or check whether the hash tables already contain a segment with the same optimistic fingerprint. This may create duplicate segments with the fingerprint value, and thereby increase the storage used for segments. But, the conservative fingerprints calculated during the two-level fingerprinting protocol ensure data consistency. Avoiding duplicates would increase the number of disk accesses due to searches necessary for each Bloom filter positive, and would require serializing write operations (to avoid a race condition where two segments with the same optimistic fingerprint are written simultaneously).

The update operation is only called when a conservative fingerprint collision is detected. First the Bloom filter is checked as described for the read operation. When the segment has been found it is updated. If the new segment is larger than the old segment, a new segment is appended to the end of the segment chunk array, and the memory for the old segment is left unused.

8.4.4.2 Bloom filter parameters

The most important goal when setting the Bloom filter parameters is to minimize the probability of false positives. A Bloom filter is a bitmap. To add an optimistic fingerprint k different combinations of the optimistic fingerprint bits are used to set m bits in the bitmap. For lookup the same k combinations are used to check whether the m bits have been set in the Bloom filter. If one of the bits is not set, the fingerprint is guaranteed not to be in the disk cache. If all are set, any of the n segments in the cache could have set the bits.

A formula for calculating the false positive probability is given by the formula in Equation 3. The probability can be minimized with respect to the number of lookups

independent of the number of entries. If there are 8 bits per entry, then using 6 lookups gives the minimum false positive ratio of 1.56%.

$f = (1 - e^{\frac{-nk}{m}})^k$	$k = (\frac{m}{n}) \ln 2 \Rightarrow f = (\frac{1}{2})^k$

Equation 3: Formula for calculating the false positive ratio for Bloom filters (left), and the same formula reduced with respect to *k* (right). *n* is the maximum number of entries, *k* is the number of lookups, *m* is the number of bits per entry.

Bitmap size (MB)	Required redundancy
64	98%
128	96%
256	92%
512	84%
1024	68%
2048	36%

Table 19: Required redundancy for a cache of a given size used to store a 100GB data set filled with 32 byte segments. If the segment size is doubled, or the data set size is reduced by two, 92% of redundancy is required for a 128 MB Bloom filter, 84% for a 256 MB Bloom filter, and so on.

Allocating memory for a Bloom filter with 8 bits for each of the 2^{40} optimistic fingerprints would require 1 TB of memory. But since only a small portion of the fingerprints are in use, we can set the size based on the data set size and expected redundancy as shown in Table 19.

8.4.4.3 Memory allocation

	1 GB DS 2 GB RAM	10 GB DS 2 GB RAM	100 GB DS 2 GB RAM	100 GBDS 4 GB RAM	100 GB DS 8 GB RAM
Bloom filter	128 MB	256 MB	1024 MB	1024 MB	1024 MB
Compression pipeline	200 MB	200 MB	200 MB	200 MB	200 MB
Hash tables	650 MB	600 MB	290 MB	1110 MB	2750 MB
Cache segments	970 MB	900 MB	430 MB	1660 MB	4120 MB

Table 20: Memory allocation for largest data structures (in addition 100 MB of memory is allocated for other data structures, executables, OS, etc).

The memory on the computer allocated for the Canidae server is shared between the Bloom filter, hash tables, segments, and the compression pipeline (Table 20). Memory is statically allocated for the Bloom filter and the compression pipeline. The container data structures dynamically manage memory such that about 40% is used for hash tables, and 60% for segments (a similar distribution is used in [209]).

8.4.4.4 Discussion

Most existing and new content based segmentation methods can be implemented to use the segment cache provided by Canidae. But, the Spring and Wetherall [209] method had

to be modified since the redundancy detection requires a cache that stores the last *N* sent bytes. We believe segmentation methods for multi-dimensional data sets require a cache that stores a set of segments, since implementing a data structure that support incremental appending of data, and that supports efficient comparison of multi-dimensional regions is an unsolved problem.

8.4.5 Segmentation methods

Several segmentation methods implemented by Canidae are described in section 8.5.3.

Table 21 summarizes the meta-data size required for each method. The meta-data is sent with the optimistic fingerprints and will therefore limit the achieved compression ratio.

Method	Meta-data size	Maximum segment size
1-D Overlapping static	1 byte	Fixed (but about 256 bytes)
1-D Spring and Wetherall [209]	2 bytes	558 bytes
1-DLBFS [156]	2 bytes	64 KB
2-D static	4 bytes	Fixed
2-D Varg [49]	6 bytes	2^{16} x fixed width

Table 21: Meta-data size for different segmentation methods implemented in Canidae.

8.4.6 Initial evaluation

To evaluate whether a large segment cache will improve compression ratio, we use the traces collected for three microarray analysis genomic applications: Java Treeview [193], TMeV [192], and GeneVaND [98]. The applications were instrumented using the Java AWTEventListener interface. The screen resolution was 1280x1024 pixels and the color depth was 32 bits per pixel. We used these to record a 15-minute trace containing all user input events for each case. Later the traces were used to create a set of screenshots, each taken after playing back a recorded mouse or keyboard event that changed the screen content.

The screenshots are read by a VNC emulator that sends these to a Canidae component that updates a local buffer, and uses the Varg method to segment the data. The segments are then sent using the two-level fingerprinting protocol. On the receiver side we have instrumented the cache such that each segment has been given a sequence number when written to the cache. The sequence number is then used to calculate the age of segments read from the cache (we define age as the number of segments that have been written since this segment was written to the segment cache).

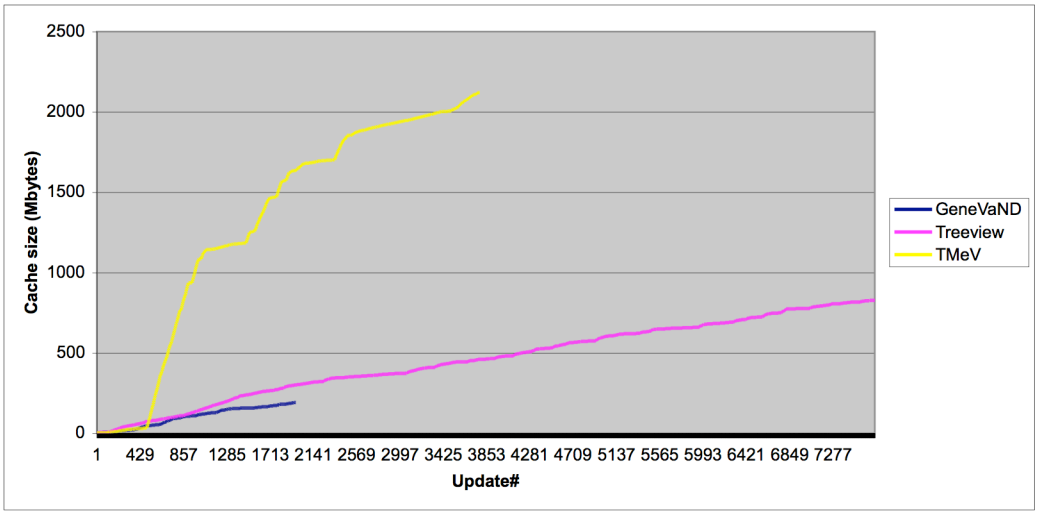


Figure 52: Cache size increase for remote visualization of three genomic applications.

Our first observation is that the number of segments cached, and hence the size of the segment cache, depends on the redundancy detected. Redundancy detection stabilizes after a while, and can be up to 80%. But since the hit ratio never reaches 100% the cache size has a steady growth (Figure 52). Even for the short 10—15 minute traces the segment cache becomes too large to be stored in memory.

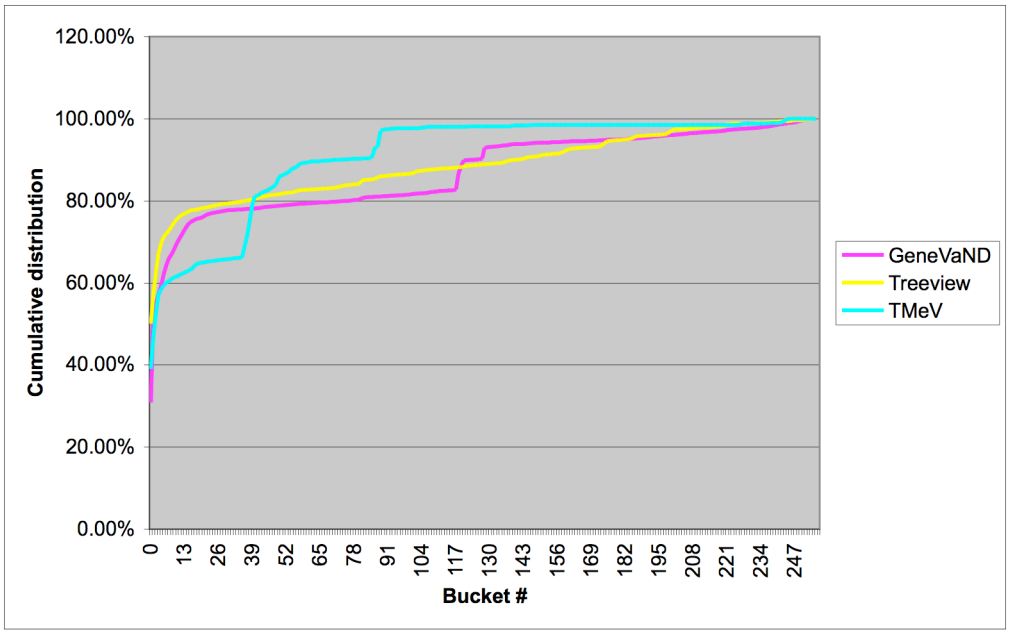


Figure 53: Cache hit entry age. Most cache hits are for recently inserted segments, but when execution time increases the number of hits for older entries increase. Note that the bucket size is 6021 for Treeview and 2445 for the other two.

A larger cache improves redundancy detection, as shown in Figure 32 where the age of the cache segments read is plotted. Most hits are for recently added segments, but as the visualization session proceeds more hits are for older segments. Therefore we believe

compression ratio will improve with a large cache for longer traces. However, the advantage needs to be evaluated using larger data sets with different segmentation methods.

8.4.7 Future work

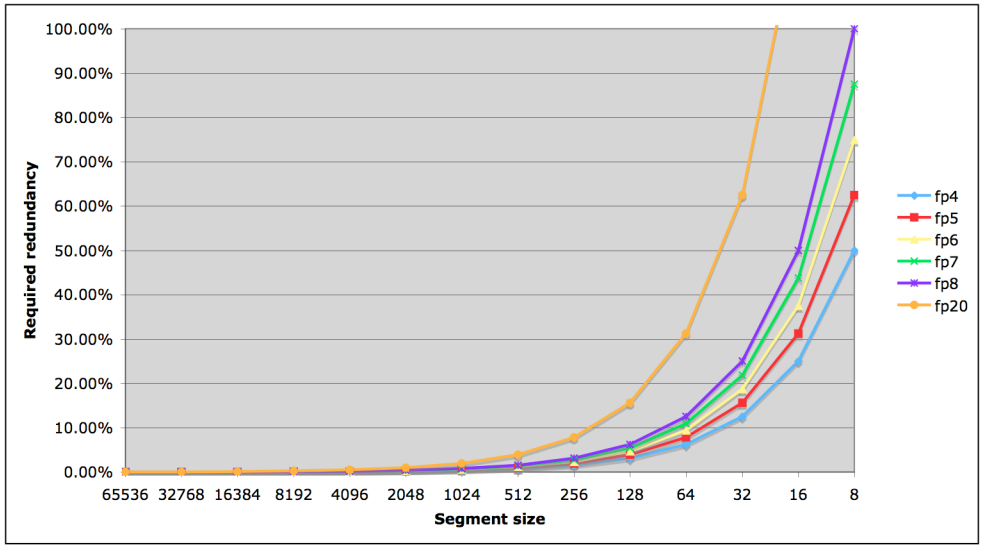


Figure 54: The minimum increase in redundancy detection for which reducing the segment size increases compression ratio (bytes sent due to collisions are ignored).

A full evaluation of the advantage of two-level fingerprinting is needed to answer the following questions:

1. Does two-level fingerprinting improve the compression ratio of previous 1-D content-based segmentation algorithms?
2. Does redundancy detection improve with smaller segments (the required increase is shown in Figure 54)?
3. Is the model used to select two-level fingerprinting parameters realistic?

The segment cache should be evaluated by answering the following:

4. Are the assumptions that segment accesses have temporal and spatial locality true?
5. How to set the parameters to achieve the best performance, in particular: the number of hash table entries, segment buffer chunk size, and the memory allocated for hash tables versus segment chunks?
6. Does the choice of container replacement algorithm significantly improve cache hit ratio?
7. Would using binary search for container tables instead of hashing improve cache hit ratio?

In addition the system performance should evaluated by measuring:

8. The throughput of Canidae compression.

9. The scalability of the Canidae compression pipeline on CMP and SMT processors.

To answer the first two questions the implemented 1-D segmentation algorithms should be used to segment a large data set consisting of uncompressed flat files. The compression ratio should then be measured when using respectively one level fingerprinting, and two levels fingerprinting for redundancy elimination. The last two questions can be answered by measuring the throughput of the system on machines with processors supporting CMT and SMT.

8.4.8 Related work

Related work was presented in section 4.4.

8.4.9 Conclusions

This chapter has presented the design and implementation of the Canidae system. Canidae is a network data compression framework that allows multiple, data-specific segmentation methods to share a segment compression engine. A two-level fingerprinting protocol has been proposed to improve redundancy elimination, and hence compression ratio by using smaller segments than previous global compression systems. In addition we propose storing a large segment cache on disk that is optimized to reduce disk accesses by using a Bloom filter.

The requirements for a 100 GB data set were modeled. Our results shows that two-level fingerprinting is most useful for segment sizes ranging from 16 to 256 bytes. In order to get the best trade-off between fingerprint bytes, and collision bytes the optimistic fingerprint size should be 40 bits, and a conservative fingerprint should cover about 20—25 segments. In addition, we demonstrated the need for a large segment cache, and how it can improve the achieved compression ratio.

8.5 Multi-level Content-Aware Segmentation for Compression of Network Data

This section presents an extended abstract that has not been submitted for publication.

The work was done in collaboration with Kai Li and Olga Troyanskaya.

Abstract. Previously proposed techniques for eliminating redundant network traffic are based on integrated anchoring and analysis of one-dimensional fine-grained data segments in data streams. The main limitation of such methods is that they are effective only for simple data types such as web contents, documents, email and binaries. They do not work well with transferring multi-dimensional data such as 2D pixels in remote data visualization and high-dimensional scientific datasets. This paper presents a method to identify and eliminate redundant data transfers of complex data types over a network. Our method is different from the previous approaches in two ways. First, the method separates data segmentation from redundancy elimination such that specific content-based segmentation methods can apply to complex data types. Second, we use a 2-dimensional segmentation approach that allows using smaller segments.

8.5.1 Introduction

Transferring multi-gigabyte datasets over a wide area network is necessary for many scientific and commercial applications. Since wide area network bandwidth is limited, compressing the transferred data is necessary to get acceptable performance for the applications.

During the past few years, global compression [156, 171, 209] has been proposed to eliminate redundant network traffic data. We will outline how such an algorithm works and then explains why previously proposed approaches work well only with 1D data types, whereas many important applications use complex data types (such as remote terminals, remote data visualization, and multidimensional datasets).

Our approach to this problem is to propose a new framework that allows application users to build content-aware anchoring mechanisms to significantly improve the network data compression. We propose a content-based anchoring method for 2D pixel segments, and a prototype system to show the proposed methods are effective. The prototype is used to compare our method to several existing 1D and 2D segmentation algorithms.

The primary contribution of this paper is:

- Application specific segmentation method for multi-dimensional data that improves the redundancy detection for complex data types. Previous general-purpose 1-D segmentation algorithms does not take into account the structure and dimensionality of the transferred network data.

8.5.2 Proposed approach

The architecture of the proposed framework for network data compression was described in section 8.4.2.

8.5.3 Segmentation methods

Canidae implements several 1-D and 2-D segmentation algorithms that are described in this section. The 1-D algorithms are all general purpose and can therefore be used on any 1-D datastream. However, the algorithms work better if protocol headers or file meta data is removed [145, 209]. The segmentation components can therefore be implemented to parse the protocol messages in order to remove protocol headers. Such parsing is necessary for multi-dimensional segmentation algorithms as described in the next section.

8.5.3.1 Manber's approach used for global compression

The basic method for segmenting 1D data streams was proposed by Manber [145]. The method computes a Rabin fingerprint [50, 175] for a window of a fixed number of bytes in a rolling fashion over a byte data stream and selects fingerprints wherever the *k* least significant bits of the fingerprint are zeros. With a uniform distribution, a fingerprint will be selected every 2^k bytes.

Manber used the fingerprints to compare the similarity of files. But the fingerprints can also be used for global compression. The disadvantage of this method is that it needs to divide data into tiny segments in order to find redundancy, segments partially overlap, and the segments may not cover all bytes. Therefore, this type of segmentation is usually combined with static segmentation (as in rsync [224] that is described in *Related Work*).

The algorithm produces segments with fixed size equal to the Rabin window size. Increasing the window size increases the average compression ratio per region, but reduces the detected redundancy. Reducing *k* will increase the redundancy found, but will also increases segment overlap, and hence reduce the segment bytes to fingerprint bytes ratio.

8.5.3.2 Spring and Wetherall's 1-D content based segmentation

Spring and Wetherall [209] adapted Manber's approach to find redundant segments in a 1-D stream of network packets. A cache is used to store previously sent data. To segment a packet, Rabin fingerprints are calculated, selected, and checked against fingerprints calculated for the data stored in the cache. For each match, the bytes covered by the fingerprint window have the same content in the cache and in the packet to be sent. The segment can then be expanded, to the left and to the right, by matching bytes in the packet and in the cache. Finally, the fingerprint and a description of the covered region are sent to the receiver.

The Canidae segment cache differs from the FIFO buffer used by Spring and Wetherall. In order to use the Canidae segment cache, the approach must be slightly modified such that a segment can be used to store the data contained in a fingerprint window, and the data to the right and left. We allocate one segment per fingerprint. All segments are fixed in size; *s* bytes. The *w* bytes covered by the fingerprint are stored in the middle of the segment, and the *(s-w)/2* bytes to the left contains the bytes preceding the fingerprint window in the last update, and the *(s-w)/2* bytes to the right contains the bytes following the fingerprint window.

In the modified algorithm the sender does the following:

1. Select fingerprints in the data to be transferred as described above.
2. Read the segments indexed by the fingerprints from the local segment cache.
3. Search for redundant region as described, but limit the segment size to *s* bytes.
4. Send redundant segments using *S&W optimistic fingerprint* messages (described below), and non-covered bytes using no-fingerprint messages.
5. Update the segment cache with the transferred data.

The receiver does the following:

1. Assemble received segment data, and calculate fingerprints for the data as described above.
2. Use the assembled data to update the segments indexed by the selected fingerprints.

The Canidae two-level fingerprinting protocol described in section 8.4.3 is extended with two new messages. In the *S&W optimistic fingerprint* message the segment size is defined as bytes before and after the fingerprint bytes. The segment cache is extended such that only the specified range is read (or written) to the cache for segment data indexed by such messages. In addition the *S&W segment message* only includes the data covered by the fingerprint, but the segment size in the cache is set to the predefined fixed segment size.

The main drawback of the modification is that segments may overlap. Therefore the segment size must be limited to reduce the number of bytes stored in multiple segments (but overlapping bits are not sent over the network). We use 256 byte fixed segments. This size is larger than the 128 byte average segment size found in [209] (the maximum segment size was limited by the number of bits allocated for storing the segment size in the protocol messages and was about 4Kbyte). The smaller maximum segment size can reduce the compression ratio. However, the Canidae cache allows storing a larger trace, which may improve compression ratio. Also, compression throughput may decreases, since the cached segments must be updated each time they are accessed.

8.5.3.3 Anchorpoint content-based segmentation

In the Low Bandwidth File System (LBFS) [156] Manber's approach is also used to select a fraction of Rabin fingerprints. But, instead of using the selected fingerprints as a starting point for growing a segment, these are used as anchorpoints in a 1-D bytestream. The anchorpoints divide the bytestream into segments, such that segment consists of all bytes in the Rabin fingerprint window, and all following bytes until the beginning of the next anchorpoint.

The median size of the segments found by the algorithm can be set to 2^k, where k is the number of fingerprint selection bits. In addition, it is necessary to specify minimum and maximum segment size to avoid regions that are either too small to get a good compression ratio, or too large to find redundancy. In [156], these were set such that the average segment size was 8KB, the minimum 2KB, and the maximum 64KB.

8.5.3.4 2-D Static Segmentation

A naïve way to anchor 2-D segments is to divide 2-D data into fixed-size grids statically. This approach is used by the VNC desktop systems [184] and by MPEG [88] encoders typically use a static 2-D grid to segment screen buffer content or movie frames (typically with 8x8 or 16x16 pixel regions). The problem with a static approach is that the anchoring is sensitive to data movement. For example if the 2-D data set is a screenshot sent by a remote desktop system then scrolling the visualization by one pixel, then segmentation of 2-D pixels will be shifted by one pixel relative to the displayed image. Even if the entire scrolled screen has been transferred previously, the content of segments will typically have changed, giving new fingerprints and hence reducing redundancy.

8.5.3.5 2-D static and content-based segmentation

Our approach [49] is to perform content-based anchoring instead of static anchoring. Since it is not practical to anchor both dimensions simultaneously due to the high computational cost, our algorithm uses Manber's technique to detect data shift first and then use the result to anchor 2D segments. Using screen pixel data as an example, we estimate whether the screen has moved mostly horizontally or vertically using Manber's technique. We generate representative fingerprints for every k-th row, and every k-th column for the screen (k is a small integer), and compare how many fingerprints are similar to the row and column fingerprints of the previous screen. Assuming that horizontal scrolling or moving will change most row fingerprints, but only a few column fingerprints, we can compare the percentage of similar row and column fingerprints to estimate which movement is dominant.

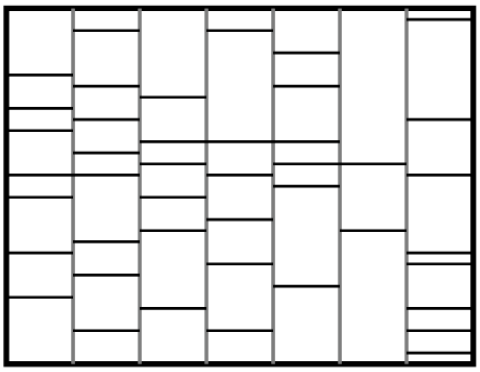


Figure 55: A 2-D array is first divided into fixed size columns. Then for each column, content-based anchor rows divide the column into segments.

For predominately vertical shift we statically divide the data into *m* columns and divide each column into regions by selecting anchoring rows (Figure 56). The anchoring rows are selected based on their fingerprint calculated using a four byte at a time Rabin fingerprint implementation. The column segmentation is ideal for scrolling because the regions move vertically with the content. If we detect predominately horizontal shift instead, we transpose the 2-D array before running the algorithm.

2-D data can include cases when large regions of the data have the same value (e.g. portions of a screen have the same color). For such regions, the row fingerprints will be identical. Thus, either all or no fingerprints will be selected. To avoid such cases, our

algorithm does fingerprint selection in three steps. First all fingerprints are calculated. Second, we scan the fingerprints and mark fingerprints as *similar* if at least *s* subsequent fingerprints are identical. Third, we select fingerprints using the *x* most significant bits, while imposing a minimum distance *m* between selected fingerprints. Also, the first and last rows are always selected.

8.5.3.6 2-D content based anchoring

This method extends the algorithm in the previous section, such that both column and row boundaries are selected based on their content. First, the he 2-D array statically into large *m x m* pixel tiles. Each tile is then divided into horizontal strips by using Manber's method to select anchor-columns based on fingerprints calculated for each column. Finally, the columns are divided into regions by selecting anchor-rows as described in the previous section (Figure 56).

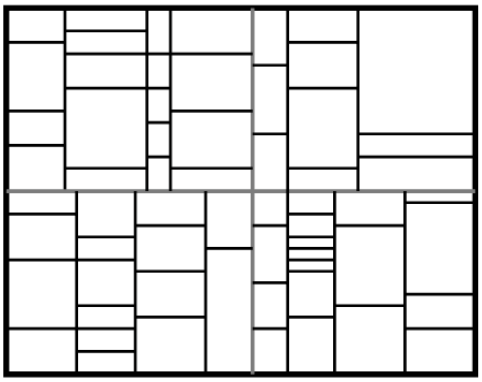


Figure 56 The 2-D array is divided into large tiles (4 tiles in this case). Each tile is segmented by first selecting anchor-columns, and then within each column selecting anchor-rows.

8.5.3.7 Rabin and Karp probabilistic 2-D segmentation

To detect all data movement in a 2-D data structure, we use an algorithm similar to the probabilistic 2-D pattern matching suggested by Karp and Rabin [122]. A short fingerprint is calculated for all *m x m* regions including all overlaps. Then regions are selected based on the fingerprint value using Manber's approach. The resulting segments divide the 2-D data structure into fixed sized segments that can overlap, and that may not cover all data.

8.5.4 Segment component implementation

The segmentation component is usually implemented as a standalone server, but can also be integrated with the fingerprint component, or application server for improved performance. In this section we describe how Canidae implements segmentation for the VNC remote desktop protocol.

8.5.4.1 VNC

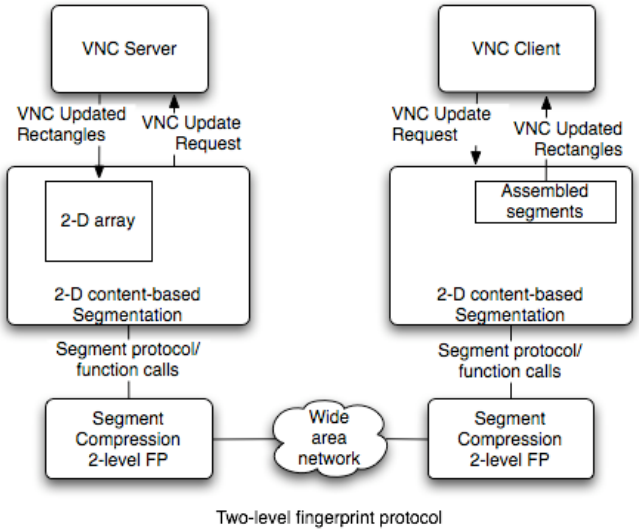


Figure 57: VNC updates segmented using Canidae. The VNC protocol is used for communication between the VNC components and the segmentation components.

Virtual Network Computing (VNC) [183, 184] is a pixel based remote visualization protocol, where the screen content on the VNC server is sent to VNC clients in the form of rectangles of pixels to be updated. Typically the framebuffer that is replicated is relatively small but is frequently updated; about 5MB for a 1280x1024 screen, but for large scale display walls [135, 234] the framebuffer can be hundreds of megabytes. Since it is an interactive application, low response time is important. Visualization intensive applications typically have large bandwidth requirements than can be provided by existing wide area networks [49].

The VNC server detects changes to the framebuffer and encodes the changes using the Remote Frame Buffer (RFB) protocol [183]. RFB is based on a single graphics primitive: “put rectangle of pixels at position (x, y)”. Also, RFB defines several compression algorithms such as: copy a region of pixels, run-length encoding, JPEG encoding, and zlib compression (as discussed above these local compression algorithms either do not provide the necessary compression ratio, are slow, or are lossy). In addition RFB provides messages for forwarding mouse and keyboard input, authentication, and server and client capability negotiation.

The segmentation components must intercept update request messages sent by the VNC client, and screen update messages sent by the VNC server (Figure 57). In Canidae the VNC server and VNC client are decoupled. The sender side segmentation component sends its own update request message to the VNC server. The received update messages are then parsed, and applied to a local 2-D array. The 2-D array is then segmented and the segments are sent using the two-level fingerprinting protocol. Simultaneously, the receiver side segmentation component intercepts update requests, and responds by encoding the received segments in RFB and sending these to the VNC client.

During initialization the VNC client connects to the VNC server, authenticates itself, and negotiates about the screen size, pixel depth, and protocol to be used. The segmentation

components must parse the screen size and pixel depth messages to initialize their local 2-D array.

The implementation of protocol handling is simplified since the message parsing code can be copied from open source VNC implementations. In addition all RFB messages not described above, can be forwarded unparsed. Also, it is not necessary to support all compression protocols, since the basic *Raw, Copy* and *Hextile* provide sufficient performance on a LAN network.

8.5.5 Initial Evaluation

We evaluate different algorithms for content-based segmentation of 2-D data sets. The following questions about the advantages of multi-dimensional segmentation, and the tuning of 2-D content-based algorithms are answered:

1. Does 2-D content-based segmentation improve the compression ratio compared to static 2-D segmentation?
2. Does 2-D application specific segmentation improve the compression ratio and time compared to general purpose 2-D segmentation?
3. What region size should be used to get the best redundancy detection?
4. Does 2-D segmentation scale with respect to data set size?

8.5.5.1 Methodology

We use four VNC data sets, which are 2-D datasets with a high degree of redundancy. Each update consists of a screenshot with 1280x1024 or 3328x1536 pixels (display wall), and 24 bits per pixel:

- TMeV: 3786 updates resulting in 14 198 MB of uncompressed data.
- GeneVaND: 1756 updates resulting in 5910 MB of uncompressed data.
- Treeview: 7693 screen updates resulting in 28 848 MB of uncompressed data.
- Treeview-display wall: 994 updates resulting in 14 537 MB of uncompressed data.

8.5.5.2 2-D segmentation

Compression method	GeneVaND	TreeView	TIGR MeV
Hextile + Zlib	13.6	19.2	14.8
Static segmentation	15.9	24.3	16.1
Probabilistic 2-D segmentation	9.5		
Static + 2-D content based	18.3		
Varg (Static + 1-D content based)	24.0	90.9	29.7
Varg without movement estimation	23.8	89.6	17.3
Varg without similar region detection	22.9	82.9	17.1

Table 22: Compression ratio for different segmentation methods for 2-D screenshot data.

The achieved compression ratios using the different methods are summarized in Table 22. This section details the results. First, we compare the Varg compression method against other widely used methods, and then find the parameters giving the best compression. Finally, the scalability of the Varg method is demonstrated.

Compared to static segmentation, content-based segmentation improves the compression ratio up to 3.0 (Table 20). The improvement is due to content-based segmentation achieving higher redundancy detection when using larger segment that compress better with local compression algorithms, and hence the total compression ratio improves.

However, content-based segmentation in both dimensions does not improve redundancy detection compared to static segmentation. The problem is that if one of the pixels in an anchor-column changes, the fingerprint for the column also changes. The changed fingerprint may not be selected as an anchor-column, and when the column boundaries are changed, all segment boundaries also change.

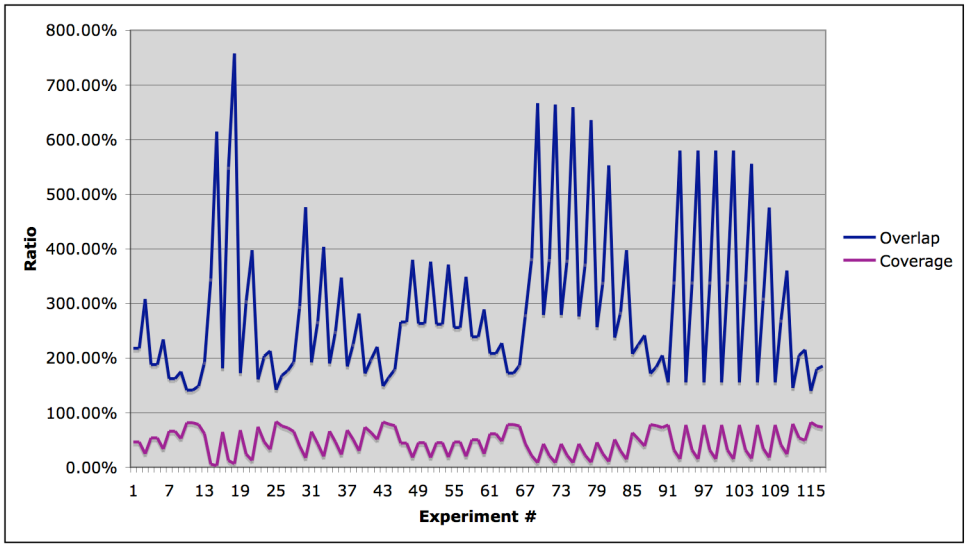


Figure 58: Probabilistic 2D pattern algorithm tuned to reduce the pixels in overlapping segments, or to reduce the number of pixels not covered by segments. Ideally both overlap and coverage should be 100%.

Probabilistic 2-D segmentation also does not provide better compression ratio than static segmentation, since pixels are either not covered or are in overlapping regions. Tuning the algorithm parameters either reduces both coverage and overlap, or increases both coverage and overlap (Figure 58). In addition, calculating Rabin fingerprints for all 2-D regions is computationally costly since a sliding window Rabin implementation cannot be used.

8.5.5.3 Algorithm parameters

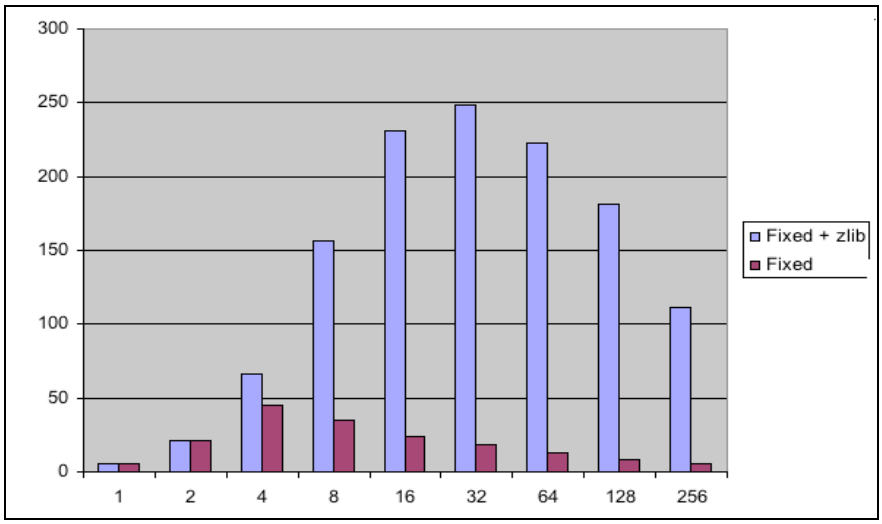


Figure 59: Compression ratio with fingerprinting and static segmentation.

With static segmentation the best total compression ratio when segments are not compressed with zlib is for 4x4 pixel regions (48 bytes), and 32x32 pixels (3072 bytes) if zlib is used (Figure 59). Similarly for Varg content-based segmentation, smaller segments improve redundancy detection, while larger segments improve zlib ratio and hence the total compression ratio.

The small segment sizes giving the best fingerprint redundancy detection are about 48—192 bytes. In section 8.4 we found that for such small segments compression ratio is limited by the fingerprint size, and that two-level fingerprinting will improve the compression ratio.

There are four parameters in the Varg 2-D segmentation method that can be changed to adjust the average region size. Our results for the genomic application traces shows that these should be set as follows to achieve the best compression ratio:

- The static column width should be small. On our experiment platform 16 pixels worked well since horizontal scrolling often moved content 16 pixels at a time. But a width of 4 pixels gives the best redundancy detection. Small static columns increase horizontal redundancy detection, since multiple pixels are typically scrolled at a time. In addition redundancy detection may decrease if a column is wide enough to include content both inside and outside a scroll-pane.
- The number of bits used for fingerprint selection, depends on the visualization. For the Treeview and TMeV trace the best ratio is when every 8th row is selected on the average. For GeneVaND selecting on the average every 32nd row gives the best ratio.
- Minimum region height should be about 8-16 rows if zlib is used, and 4 pixels if not. A smaller minimum decreases the total compression ratio due to reduced zlib compression ratio. A larger minimum also decreases redundancy detection since a change to an anchor row may cause subsequent anchor rows not to be selected since they are within the minimum height.

- Specifying a maximum region height does not improve compression ratio, but may be necessary due to the fingerprint protocol messages having restrictions on the number of bits that can be used to store the segment size.

8.5.5.4 Application specific segmentation

Setting the parameters as discussed above gives a region size distribution as shown in Figure 60. About 10% of the segments have height less than the minimum distance. There are three classes of such regions: (i) the segments at the bottom of each column, (ii) segments just above a similar region, and (iii) segments below a similar region. Each trace has a few segments where the height is above 512 rows. But these often have similar content, which compresses well with zlib.

The last two cases can be avoided by disabling similar detection of column rows with identical content (Figure 61). But the total compression ratio is reduced by 1—8% since zlib compression ratio decreases.

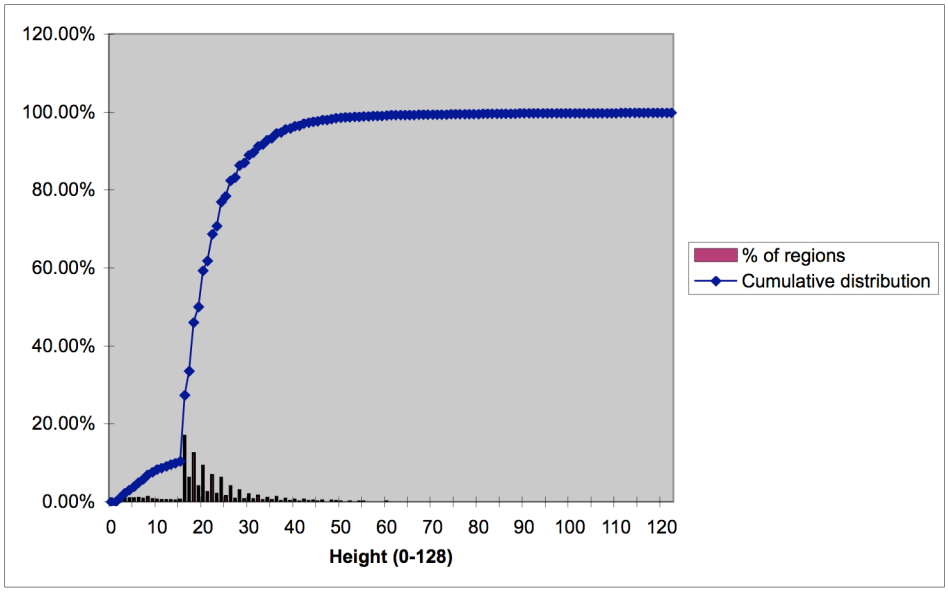


Figure 60: Segment height distribution for the Treeview trace. Minimum height is 16, and the median is 19.

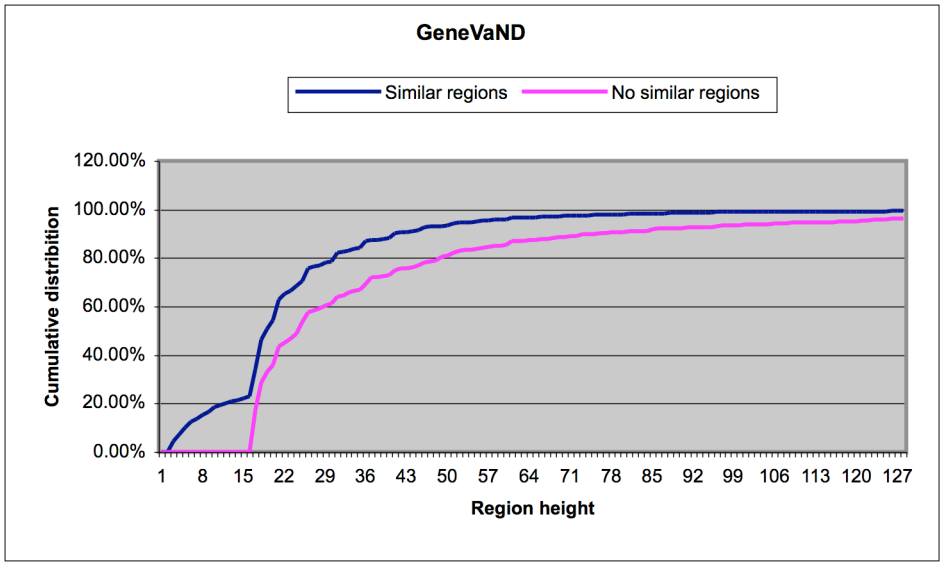


Figure 61: Segment height distribution for the GeneVaND trace with and without similar region detection (the other traces are similar).

Movement estimation improves the compression ratio for TMeV with 72%, since about 30% of the updates have predominantly horizontal movement. The compression ratio improvement is smaller for the other traces, since few updates had horizontal movement.

8.5.5.5 Scalability

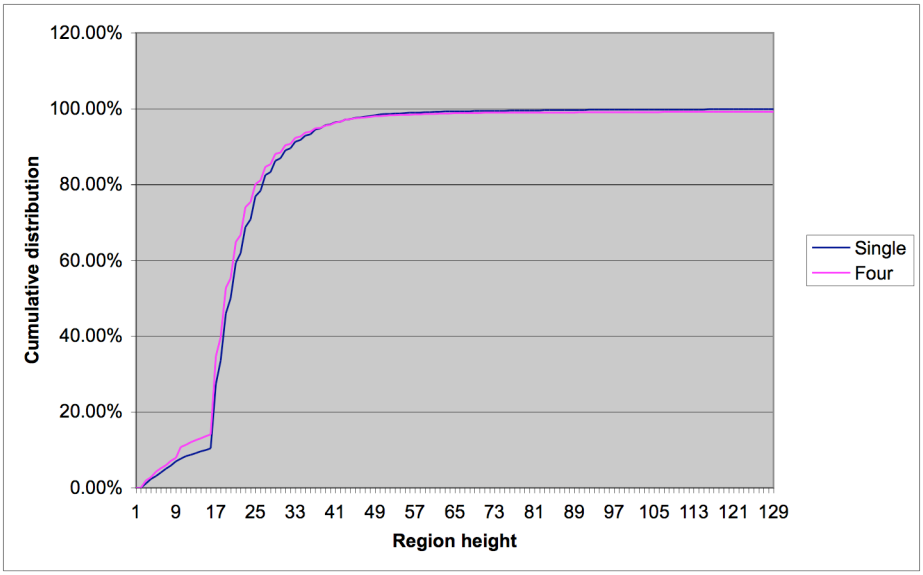


Figure 62. Cumulative distribution of segment heights shows that these do not change when the screen size increases.

The Varg segmentation method scales with screen size. With a larger screen the same algorithm parameters give the best compression ratio, and the distribution of segment sizes do not change (Figure 62). However, the total compression ratio improves, due to

improved fingerprinting and zlib compression ratio. It is therefore not necessary to tune the segmentation method for different screen sizes.

8.5.6 Future Work

A full evaluation is required to answer the following questions:

1. Does 2-D based segmentation methods improve the compression ratio and time compared to 1-D segmentation methods for 2-D data?
2. Does the 2-D based segmentation methods also work well with scientific 2-D data?
3. Does our modified Spring and Wetherall segmentation algorithm achieve similar compression ratio as the original algorithm?

To answer these questions it is necessary to experiment with the 1-D and the 2-D segmentation methods described above. The data sets to use in the evaluation could be from different scientific domains, such as geometric data, scientific simulation output, and satellite images.

8.5.7 Conclusions

This paper has evaluated different segmentation methods for 2-D data. The segmentation methods are implemented in a network data compression framework called Canidae. It allows multiple, data-specific segmentation methods to share a segment compression engine. A two-level fingerprinting protocol is used to provide high compression ratio with smaller segments than previous global compression systems. Also proposed is a novel method to compress 2-D pixel segments by using fingerprinting.

We found that 2-D content-based segmentation algorithms improve compression ratio up to 3.0, compared to static compression methods. In addition we found that applying screenshot specific optimizations to the segmentation algorithm, improves screenshot compression up to 1.7. In addition we demonstrate that the algorithm parameters can be set independent of the visualization and the screen size.

References

1. *HPC Challenge Benchmark*. 2007. http://icl.cs.utk.edu/hpcc/index.html.
2. *Iperf webpage*. 2007. http://dast.nlanr.net/Projects/Iperf/.
3. *LAM-MPI homepage*. http://www.lam-mpi.org/.
4. *LINPACK Benchmark*. http://www.netlib.org/linpack/.
5. *MPICH home page*. http://www.mcs.anl.gov/mpi/mpich/.
6. *PMB - Pallas MPI Benchmarks*. http://www.pallas.com/e/products/pmb/.
7. *Secure Hash Standard*. FIPS PUB 180-1 National Institute of Standards and Technology. 1995.
8. *SETI@home - The Search for Extraterrestrial Intelligence*. http://setiathome.ssl.berkeley.edu/.
9. *Zlib/ gzlib library home page.* http://www.zlib.net/.
10. Aas, J. *Understanding the Linux 2.6.8.1 CPU Scheduler*. Silicon Graphics, Inc. (SGI). 2005.
11. Adiga, N.R., Almasi, G., Almasi, G.S., Aridor, Y., Barik, R., Beece, D., Bellofatto, R., Bhanot, G., Bickford, R., Blumrich, M., et al., An overview of the BlueGene/L Supercomputer. in *Proc. of the 2002 ACM/IEEE conference on Supercomputing*, (Baltimore, Maryland, 2002), IEEE Computer Society Press, pages 1-22.
12. Alexandrov, A., Ionescu, M.F., Schauser, K.E. and Scheiman, C., LogGP: incorporating long messages into the LogP model—one step closer towards a realistic model for parallel computation. in *Proc. of the seventh annual ACM symposium on Parallel algorithms and architectures*, (Santa Barbara, California, United States, 1995), ACM Press, pages 95-105.
13. Amdahl, G.M., Validity of the single processor approach to achieving large-scale computing capabilities. in *Proc. of the AFIPS Spring Joint Computer Conference*, (Atlantic City, New Jersey, USA, 1967), AFIPS Press, pages 483-485.
14. Anderson, D.P., Tzou, S.-Y., Wahbe, R., Govindan, R. and Andrews, M., Support for continuous media in the DASH system. in *Proc. of 10th International Conference on Distributed Computing Systems (ICDCS)*, (Paris, France, 1990), IEEE Computer Society, pages 54-61.
15. Anderson, J.M., Berc, L.M., Dean, J., Ghemawat, S., Henzinger, M.R., Leung, S.-T.A., Sites, R.L., Vandevoorde, M.T., Waldspurger, C.A. and Weihl, W.E. Continuous profiling: where have all the cycles gone? *ACM Trans. Comput. Syst.*, *15* (4): 357-390. 1997.
16. Anderson, T.E. and Lazowska, E.D., Quartz: a tool for tuning parallel program performance. in *Proc. of the 1990 ACM SIGMETRICS conference on Measurement and modeling of computer systems*, (Boulder, Colorado, USA, 1990), ACM Press, pages 115-125.
17. Annapureddy, S., Freedman, M.J. and Mazieres, D., Shark: scaling file servers via cooperative caching. in *Proc. of the 2nd conference on Symposium on Networked Systems Design Implementation*, (2005), USENIX Association.
18. Anshus, O.J., Bjørndalen, J.M. and Bongo, L.A., Parallelizing, Configuring, and Monitoring of Parallel Applications on Clusters. in *Proc. of ParCo 2003*

(Dresden, Germany, 2003), Elsevier, Advances in Parallel Computing 13, pages 879-886.
19. Apple. *Apple Remote Desktop*. http://www.apple.com/remotedesktop/.
20. Arpaci-Dusseau, A.C. Implicit coscheduling: coordinated scheduling with implicit information in distributed systems. *ACM Transactions on Computer Systems*, *19* (3): 283-331. 2001.
21. BABAR Collaboration. *BaBar high energy physics experiment*. 2006. http://www-public.slac.stanford.edu/babar/.
22. Babcock, B., Babu, S., Datar, M., Motwani, R. and Widom, J., Models and issues in data stream systems. in *Proc. of the twenty-first Symposium on Principles of database systems*, (Madison, Wisconsin, 2002), ACM Press, pages 1-16.
23. Bala, V., Bruck, J., Cypher, R., Elustando, P., Ho, A., Ho, C.-T., Kipnis, S. and Snir, M. CCL: a portable and tunable collective communication library for scalable parallel computers. *IEEE Trans. Parallel Distrib. Syst.*, *6* (2): 154-164. 1995.
24. Balle, S.M., Bishop, J., LaFrance-Linden, D. and Rifkin, H., Ygdrasil: aggregator network toolkit for the Grid. in *Proc. of PARA'04 - Workshop on State-of-the-Art in Scientific Computing*, (Lyngby, Denmark, 2004), Springer.
25. Baratto, R.A., Kim, L.N. and Nieh, J., THINC: a virtual display architecture for thin-client computing. in *Proc. of SOSP '05: the twentieth ACM symposium on Operating systems principles*, (Brighton, United Kingdom, 2005), ACM Press, pages 277-290.
26. Barnett, M., Gupta, S., Payne, D., Shuler, L., van de Geijn, R.A. and Watts, J., Interprocessor collective communication library (Intercom). in *Proc. of Scalable High Performance Computing Conference 1994*, (1994).
27. Barnett, M., Gupta, S., Payne, D.G., Shuler, L., Geijn, R.v.d. and Watts, J., Building a high-performance collective communication library. in *Proc. of the 1994 ACM/IEEE conference on Supercomputing*, (Washington, D.C., 1994), ACM Press, pages 107-116.
28. Begole, J., Rosson, M.B. and Shaffer, C.A. Flexible collaboration transparency: supporting worker independence in replicated application-sharing systems. *ACM Trans. Comput.-Hum. Interact.*, *6* (2): 95-132. 1999.
29. Bell, C., Bonachea, D., Nishtala, R. and Yelick, K., Optimizing bandwidth limited problems using one-sided communication and overlap
. in *Proc. of 20th International Parallel and Distributed Processing Symposium (IPDPS)*, (2006), IEEE.
30. Berliner, B., CVS II: Parellizing software development. in *Proc. of the Winter 1990 USENIX Technical Conference*, (Colorado Springs, CO, USA, 1990).
31. Bernaschi, M. and Iannello, G. Collective Communication Operations: Experimental Results vs.Theory. *Concurrency: Practice and Experience*, *10* (5). 1998.
32. Bjørndalen, J.M. *Improving the Speedup of Parallel and Distributed Applications on Clusters and Multi-Clusters*. Ph.D. dissertation. Department of Computer Science. University of Tromsø. 2003.
33. Bjørndalen, J.M., Anshus, O., Larsen, T., Bongo, L.A. and Vinter, B., Scalable processing and communication performance in a multi-media related context. in *Proc. of 28th EUROMICRO Conference 2002*, (Dortmund, Germany, 2002), IEEE Computer Society, pages 200-206.
34. Bjørndalen, J.M., Anshus, O., Larsen, T. and Vinter, B., PATHS - Integrating the principles of method-combination and remote procedure calls for run-time

configuration and tuning of high-performance distributed applications. in *Proc. of Norsk Informatikk Konferanse (NIK 2001)*, (Tromsø, Norway, 2001).

35. Bjørndalen, J.M., Anshus, O., Vinter, B. and Larsen, T., Configurable collective communication in LAM-MPI. in *Proc. of Communicating Process Architectures 2002*, (Reading, UK, 2002).
36. Bjørndalen, J.M., Anshus, O., Vinter, B. and Larsen, T., The performance of configurable collective communication for LAM-MPI in clusters and multi-clusters. in *Proc. of NIK 2002, Norsk Informatikk Konferanse*, (Kongsberg, Norway, 2002).
37. Bloom, B.H. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, *13* (7): 422-426. 1970.
38. Blue Coat Systems. *Caching and compression – key complementary technologies for application acceleration*. White paper 2007.
39. Bobbarjung, D.R., Jagannathan, S. and Dubnicki, C. Improving duplicate elimination in storage systems. *ACM Trans. on Storage*, *2* (4): 424-448. 2006.
40. Boden, N.J., Cohen, D., Felderman, R.E., Kulawik, A.E., Seitz, C.L., Seizovic, J.N. and Su, W.-K. Myrinet: A Gigabit-per-Second Local Area Network. *IEEE Micro*, *15* (1): 29-36. 1995.
41. Bonachea, D. and Duell, J. Problems with using MPI 1.1 and 2.0 as compilation targets for parallel language implementations. *Int. J. High Performance Computing and Networking*, *1* (1/2/3): 91-99. 2004.
42. Bongo, L.A. *The Longcut wide area network emulator: design and evaluation*. Technical report 2005-53. Dep. of Computer Science, University of Tromsø. 2005.
43. Bongo, L.A., Anshus, O. and Bjørndalen, J.M., Collective communication performance analysis within the communication system. in *Proc. of Euro-Par 2004*, (Pisa, Italy, 2004), Springer, Lecture Notes in Computer Science 3149, pages 163-172.
44. Bongo, L.A., Anshus, O. and Bjørndalen, J.M. *Evaluating the performance of the allreduce collective operation on clusters: approach and results*. Technical Report 2004-48. Dep.of Computer Science, University of Tromsø. 2004.
45. Bongo, L.A., Anshus, O. and Bjørndalen, J.M., EventSpace - exposing and observing communication behavior of parallel cluster applications. in *Proc. of Euro-Par 2003*, (Klagenfurt, Austria, 2003), Springer, Lecture Notes in Computer Science 2790, pages 47-56.
46. Bongo, L.A., Anshus, O., Bjørndalen, J.M. and Larsen, T., Extending collective operations with application semantics for improving multi-cluster performance. in *Proc. of ISPDC/HeteroPar*, (Cork, Ireland, 2004), IEEE Computer Society, pages 320-327.
47. Bongo, L.A., Anshus, O.J. and Bjørndalen, J.M., Low overhead high performance runtime monitoring of collective communication. in *Proc. of the 2005 International Conference on Parallel Processing*, (Oslo, Norway, 2005), IEEE Computer Society, pages 455-464.
48. Bongo, L.A., Vinter, B., Anshus, O.J., Larsen, T. and Bjørndalen, J.M., Using overdecomposition to overlap communication latencies with computation and take advantage of SMT processors. in *Proc. of International Conference on Parallel Processing Workshops (ICPP Workshops 2006)*, (Columbus, Ohio, USA, 2006), IEEE Computer Society, pages 239-247.
49. Bongo, L.A., Wallace, G., Larsen, T., Li, K. and Troyanskaya, O., Systems support for remote visualization of genomics applications over wide area

networks. in *Proc. of GCCB 2006*, (Eilat, Israel, 2007), Springer, Lecture Notes in Bioinformatics 4360, pages 157-174.
50. Broder, A., Some applications of Rabin's fingerprinting method. in *Proc. of Sequences II: Methods in Communications, Security, and Computer Science*, (1993), Springer-Verlag.
51. Brunst, H., Hoppe, H.-C., Nagel, W.E. and Winkler, M., Performance optimization for large scale computing - The scalable VAMPIR approach. in *Proc. of International Conference on Computational Science (2)*, (San Francisco, CA, USA, 2001), Springer, Lecture Notes in Computer Science 2074, pages 751-760.
52. Burns, G., Daoud, R. and Vaigl, a.J., LAM: An Open Cluster Environment for MPI. in *Proc. of Supercomputing Symposium 1994*, (Toronto, Canada, 1994).
53. Cahill, D.J. and Nordhoff, E. Protein arrays and their role in proteomics. *Adv Biochem Eng Biotechnol*, *83*177-187. 2003.
54. Cai, X., Improving the performance of large-scale unstructured PDE applications. in *Proc. of PARA'04 Workshop*, (Lyngby, Denmark, 2006), Springer, Lecture Notes in Computer Science, pages 699–708.
55. Caubet, J., Gimenez, J., Labarta, J., Rose, L.D. and Vetter, J.S., A dynamic tracing mechanism for performance analysis of OpenMP applications. in *Proc. of the International Workshop on OpenMP Applications and Tools: OpenMP Shared Memory Parallel Programming*, (2001), Springer-Verlag, pages 53-67.
56. CERN. *Large Hadron Collider (LHC)* 2006. http://lhc.web.cern.ch/lhc/.
57. CERN. *The LCG infrastructure built for LHC*. http://lcg.web.cern.ch/LCG/.
58. Chen, P.M. and Patterson, D.A. A new approach to I/O performance evaluation: self-scaling I/O benchmarks, predicted I/O performance. *ACM Transactions on Computer Systems (TOCS)*, *12* (4): 308-339. 1994.
59. Choi, G.S., Kim, J.-H., Ersoz, D., Yoo, A.B. and Das, C.R., Coscheduling in clusters: is it a viable alternative? in *Proc. of the 2004 ACM/IEEE conference on Supercomputing*, (Pittsburgh, PA, 2004), IEEE Computer Society Press.
60. Christiansen, B.O. and Schauser, K.E., Fast Motion Detection for Thin Client Compression. in *Proc. of the Data Compression Conference (DCC '02)*, (2002), IEEE Computer Society.
61. Chung, I.H., Walkup, R.E., Wen, H.-F. and Yu, H., MPI performance analysis tools on Blue Gene/L. in *Proc. of the 2006 ACM/IEEE conference on Supercomputing*, (Tampa, Florida, 2006), ACM Press.
62. Cisco Systems. *Cisco wide area application services (WAAS) V4.0 technical overview*. Whitepaper 2006.
63. Cisco Systems Inc. *WebEx homepage*. http://www.webex.com/.
64. Cox, L.P., Murray, C.D. and Noble, B.D., Pastiche: making backup cheap and easy. in *Proc. of the 5th symposium on Operating systems design and implementation*, (Boston, Massachusetts, 2002), ACM Press, pages 285-298.
65. Culler, D., Karp, R., Patterson, D., Sahay, A., Schauser, K.E., Santos, E., Subramonian, R. and Eicken, T.v., LogP: towards a realistic model of parallel computation. in *Proc. of the fourth ACM SIGPLAN symposium on Principles and practice of parallel programming*, (San Diego, California, United States, 1993), ACM Press, pages 1-12.
66. Culler, D.E. and Singh, J.P. *Parallel Computer Architecture: A Hardware / Software Approach*. Morgan Kaufmann, 1999.
67. Cumberland, B.C., Carius, G. and Muir, A. *Microsoft Windows NT Server 4.0, Terminal server edition: technical reference*. Microsoft Press 1999.
68. Cutler, P. Protein arrays: the current state-of-the-art. *Proteomics*, *3* (1): 3-18. Jan, 2003.

69. Dabek, F., Kaashoek, M.F., Karger, D., Morris, R. and Stoica, I., Wide-area cooperative storage with CFS. in *Proc. of the eighteenth ACM symposium on Operating systems principles*, (Banff, Alberta, Canada, 2001), ACM Press, pages 202-215.
70. Danalis, A., Kim, K.-Y., Pollock, L. and Swany, M., Transformations to parallel codes for communication-computation overlap. in *Proc. of the 2005 ACM/IEEE conference on Supercomputing*, (2005), IEEE Computer Society.
71. Dean, J. and Ghemawat, S., MapReduce: simplified data processing on large clusters. in *Proc. of OSDI 2004*, (2004), Usenix, pages 137-150.
72. Denehy, T.E. and Hsu, W.W. *Reliable and efficient storage of reference data*. Technical Report RJ10305. IBM Research. October, 2003.
73. Deutsch, P. *DEFLATE Compressed Data Format Specification version 1.3*. Network Working Group Request for Comments 1951. The Internet Engineering Task Force. May, 1996.
74. Dongarra, J., Foster, I., Fox, G., Gropp, W., Kennedy, K., Torczon, L. and White, A. (eds.). *Sourcebook of parallel computing*. Morgan Kaufmann Publishers Inc., 2003.
75. Douceur, J.R., Adya, A., Bolosky, W.J., Simon, D. and Theimer, M., Reclaiming space from duplicate files in a serverless distributed file system. in *Proc. of the 22 nd International Conference on Distributed Computing Systems (ICDCS'02)*, (2002), IEEE Computer Society.
76. Douglis, F. and Iyengar, A., Application-specific delta-encoding via resemblance detection. in *Proc. of 2003 USENIX Technical Conference*, (2003).
77. Eggert, L. and Touch, J.D., Idletime scheduling with preemption intervals. in *Proc. of the twentieth ACM symposium on Operating systems principles*, (Brighton, United Kingdom, 2005), ACM Press, pages 249-262.
78. Evensky, D.A., Gentile, A.C., Camp, L.J. and Armstrong, R.C., Lilith: scalable execution of user code for distributed computing. in *Proc. of Sixth IEEE International Symposium on High Performance Distributed Computing*, (Portland, Oregon, 1997), pages 306-314.
79. F5 Networks. *The BIG-IP system with intelligent compression: cutting application delivery time and optimizing bandwidth*. White Paper 2007.
80. Faraj, A. and Yuan, X., Automatic generation and tuning of MPI collective communication routines. in *Proc. of the 19th annual international conference on Supercomputing*, (Cambridge, Massachusetts, 2005), ACM Press, pages 393-402.
81. Faraj, A., Yuan, X. and Lowenthal, D., STAR-MPI: self tuned adaptive routines for MPI collective operations. in *Proc. of the 20th annual international conference on Supercomputing*, (Cairns, Queensland, Australia, 2006), ACM Press, pages 199-208.
82. Feibush, E. *ElVis homepage*. http://w3.pppl.gov/elvis.
83. Figueiredo, R.J.O. and Fortes, J.A.B., Impact of Heterogeneity on DSM Performance. in *Proc. of Sixth International Symposium on High-Performance Computer Architecture*, (2000).
84. Foster, I. and Kesselman, C. (eds.). *The Grid 2: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers Inc., 2003.
85. Foster, I. and Kesselman, C. (eds.). *The grid: Blueprint for a new computing infrastructure*. Morgan Kaufmann Publishers Inc., 1999.
86. Fowler, G., Vo, P. and Noll, L.C. *Fowler / Noll / Vo (FNV) Hash webpage*. http://www.isthe.com/chongo/tech/comp/fnv/.

87. Gabriel, E., Fagg, G.E., Bosilca, G., Angskun, T., Dongarra, J.J., Squyres, J.M., Sahay, V., Kambadur, P., Barrett, B., Lumsdaine, A., et al., Open MPI: goals, concept, and design of a next generation MPI implementation. in *Proc. of 11th European PVM/MPI Users' Group Meeting*, (Budapest, Hungary, 2004).
88. Gall, D.L. MPEG: a video compression standard for multimedia applications. *Commun. ACM*, *34* (4): 46-58. 1991.
89. Geppert, L. Sun's Big Splash. *IEEE Spectrum*, *42* (1): 56-60. January, 2005.
90. Gigabit Ethernet Alliance. *Gigabit Ethernet: Accelerating the Standard for Speed*. White paper May, 1999.
91. Goldberg, A.J. and Hennessy, J.L. Mtool: an integrated system for performance debugging shared memory multiprocessor applications. *IEEE Trans. Parallel Distrib. Syst.*, *4* (1): 28-40. 1993.
92. Gordon, B. and Jim, G. What's next in high-performance computing? *Commun. ACM*, *45* (2): 91-95. 2002.
93. Graham, S.L., Kessler, P.B. and Mckusick, M.K., Gprof: A call graph execution profiler. in *Proc. of 1982 SIGPLAN symposium on Compiler construction*, (Boston, Massachusetts, United States, 1982), ACM Press, pages 120-126.
94. Gray, J., Liu, D.T., Nieto-Santisteban, M., Szalay, A., DeWitt, D.J. and Heber, G. Scientific data management in the coming decade. *SIGMOD Rec.*, *34* (4): 34-41. 2005.
95. Habata, S., Umezawa, K., Yokokawa, M. and Kitawaki, S. Hardware system of the Earth Simulator. *Parallel Comput.*, *30* (12): 1287-1313. 2004.
96. Hagen, T.-M.S. *The PATHS Visualizer*. Master thesis. Department of Computer Science. University of Tromsø. 2006.
97. Hayes, E.F. and al, e. *Report of the Task Force of the Future of the NSF Supercomputer Centers Program*. National Science Foundation Report 9646. September, 1995.
98. Hibbs, M.A., Dirksen, N.C., Li, K. and Troyanskaya, O.G. Visualization methods for statistical analysis of microarray clusters. *BMC Bioinformatics*, *6*. 2005.
99. Hoeflinger, J., Kuhn, B., Nagel, W.E., Petersen, P., Rajic, H., Shah, S., Vetter, J.S., Voss, M. and Woo, R., An integrated performance cisualizer for MPI/OpenMP programs. in *Proc. of the International Workshop on OpenMP Applications and Tools: OpenMP Shared Memory Parallel Programming*, (2001), Springer-Verlag, pages 40-52.
100. Hong, B., Plantenberg, D., Long, D.D.E. and Sivan-Zimet, M., Duplicate data elimination in a san file system. in *Proc. of the 21st IEEE/12th NASA Goddard Conference on Mass Storage Systems and Technologies (MSST)*, (2004), pages 301-314.
101. Housel, B.C. and Lindquist, D.B., WebExpress: a system for optimizing Web browsing in a wireless environment. in *Proc. of Proceedings of the 2nd annual international conference on Mobile computing and networking*, (Rye, New York, United States, 1996), ACM Press, pages 108-116.
102. Huang, R.P. Protein arrays, an excellent tool in biomedical research. *Front Biosci*, *8*d559-576. May 1, 2003.
103. Huffman, D.A. A method for the construction of minimum-redundancy codes. *Proc. of the Institute of Radio Engineers*, *40* (9): 1098-1101. September, 1952.
104. Humphreys, G., Houston, M., Ng, R., Frank, R., Ahern, S., Kirchner, P.D. and Klosowski, J.T., Chromium: a stream-processing framework for interactive rendering on clusters. in *Proc. of the 29th annual conference on Computer graphics and interactive techniques*, (San Antonio, Texas, USA, 2002), ACM Press, pages 693-702.

105. Husbands, P. and Hoe, J.C., MPI-StarT: delivering network performance to numerical applications. in *Proc. of the 1998 ACM/IEEE conference on Supercomputing*, (San Jose, CA, 1998), IEEE Computer Society, pages 1-15.
106. Iancu, C.C. and Strohmaier, E., Optimizing communication overlap for high-speed networks. in *Proc. of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*, (San Jose, California, USA, 2007), ACM Press, pages 35-45.
107. IBM Advanced Computing Technology Center. *MPI Tracer/Profiler*. http://domino.research.ibm.com/comm/research_projects.nsf/pages/actc.mpitracer.html.
108. IBM Systems & Technology Group. *How DB2 exploits IBM @serverp5 and AIX 5L Simultaneous Multithreading*. IBM Whitepaper October, 2004.
109. InfiniBand Trade Association. *InfiniBand Architecture Specification, Release 1.0*. October, 2000.
110. Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 1: Basic Architecture*. 2006.
111. Intel Corporation. *Intel Trace Analyzer and Collector 7.0*. http://www3.intel.com/cd/software/products/asmo-na/eng/306321.htm.
112. Intel Corporation. *Intel VTune Performance Analyzer*. 2007. http://www.intel.com/cd/software/products/asmo-na/eng/239144.htm.
113. International Standard. *Information Technology - Generic Coding of Moving Pictures and Associated Audio Information - Part 3: Audio*. ISO/IEC 13818-3 1994.
114. Irmak, U. and Suel, T., Hierarchical substring caching for efficient content distribution to low-bandwidth clients. in *Proc. of the 14th international conference on World Wide Web*, (Chiba, Japan, 2005), ACM Press, pages 43-53.
115. Jeffrey, V., Performance analysis of distributed applications using automatic classification of communication inefficiencies. in *Proc. of the 14th international conference on Supercomputing*, (Santa Fe, New Mexico, United States, 2000), ACM Press, pages 245-254.
116. Jeong, B., Renambot, L., Jagodic, R., Singh, R., Aguilera, J., Johnson, A. and Leigh, J., High-performance dynamic graphics streaming for scalable adaptive graphics environment. in *Proc. of the 2006 ACM/IEEE conference on Supercomputing*, (Tampa, Florida, 2006), ACM Press.
117. Johnsen, E.S., Bjørndalen, J.M. and Anshus, O., CoMPI - Configurable collective operations in LAM/MPI using the scheme programming language. in *Proc. of PARA 2006*, (Umeaa, Sweden, 2006), Springer, LNCS (in publication).
118. Jones, T., Tuel, W., Brenner, L., Fier, J., Caffrey, P., Dawson, S., Neely, R., Blackmore, R., Maskell, B., Tomlinson, P., et al., Improving the scalability of parallel jobs by adding parallel awareness to the operating system. in *Proc. of the 2003 ACM/IEEE conference on Supercomputing*, (Phoenix, Arizona, 2003).
119. Jost, G., Jin, H., Labarta, J., Gimenez, J. and Caubet, J., Performance analysis of multilevel parallel applications on shared memory architectures. in *Proc. of the 17th International Symposium on Parallel and Distributed Processing*, (2003), IEEE Computer Society.
120. Juniper Networks. *Juniper Networks WAN acceleration platforms*. 2007. http://www.juniper.net/products_and_services/application_acceleration/wan_acceleration/index.html.
121. Kalla, R.N., Sinharoy, B. and Tendler, J.M. IBM Power5 chip: a dual-core multithreaded processor. *IEEE Micro*, *24* (2): 40-47. 2004.

122. Karp, R.M. and Rabin, M.O. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, *31* (2): 249-260. 1987.
123. Karwande, A., Yuan, X. and Lowenthal, D.K., CC-MPI: a compiled communication capable MPI prototype for Ethernet switched clusters. in *Proc. of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming*, (San Diego, California, USA, 2003), ACM Press, pages 95-106.
124. Keller, R., Bosilca, G., Fagg, G.E., Resch, M.M. and Dongarra, J., Implementation and Usage of the PERUSE-Interface in Open MPI. in *Proc. of PVM/MPI*, (Bonn, Germany, 2006), Springer, Lecture Notes in Computer Science 4192, pages 347-355.
125. Keller, R., Gabriel, E., Krammer, B., Müller, M.S. and Resch, M.M. Towards efficient execution of MPI applications on the Grid: porting and optimization issues. *Grid Computing*, *1* (2): 133-149. 2003.
126. Kielmann, T., Hofman, R.F.H., Bal, H.E., Plaat, A. and Bhoedjang, R.A.F., MagPIe: MPI's collective communication operations for clustered wide area systems. in *Proc. of the seventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, (Atlanta, Georgia, United States, 1999), pages 131-140.
127. Kilburn, T., Edwards, D.B.G., Lanigan, M.J. and Sumner, F.H. One-level storage system. *IRE Trans. Electronic Computers*. April, 1962.
128. Kohn, J. and Williams, W. ATExpert. *Parallel and Distributed Computing*, *18* (2): 205-222. June, 1993.
129. Kumar, S., Jiang, D., Chandra, R. and Singh, J.P., Evaluating synchronization on shared address space multiprocessors: methodology and performance. in *Proc. of the 1999 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, (Atlanta, Georgia, United States, 1999), ACM Press, pages 23-34.
130. Kvalnes, A., Johansen, D., Arnesen, A. and Renesse, R.v. *Vortex: an event-driven multiprocessor operating system supporting performance isolation*. Technical report 2003-45. Dep.of Computer Science, University of Tromsø. 2003.
131. Lai, A.M. and Nieh, J. On the performance of wide-area thin-client computing. *ACM Trans. Comput. Syst.*, *24* (2): 175-209. 2006.
132. Lawrence Livermore National Laboratory. *ASCI Purple*. http://www.llnl.gov/asc/computing_resources/purple/purple_index.html.
133. Lawrence Livermore National Laboratory. *ASCI Purple Benchmark*. http://www.llnl.gov/asci/platforms.
134. Leu, J.S., Agrawal, D.P. and Mauney, J., Modeling of parallel software for efficient computation communication overlap. in *Proc. of the 1987 Fall Joint Computer Conference on Exploring technology: today and tomorrow*, (Dallas, Texas, United States, 1987), IEEE Computer Society Press, pages 569-575.
135. Li, K., Chen, H., Chen, Y., Clark, D.W., Cook, P., Damianakis, S., Essl, G., Finkelstein, A., Funkhouser, T., Timoth, H., et al. Building and using a scalable display wall system. *IEEE Comput. Graph. Appl.*, *20* (4): 29-37. 2000.
136. Lin, B. and Dinda, P.A., VSched: mixing batch and interactive virtual machines using periodic real-time scheduling. in *Proc. of the 2005 ACM/IEEE conference on Supercomputing*, (2005), IEEE Computer Society.
137. Lipshutz, R.J., Fodor, S.P.A., Gingeras, T.R. and Lockhart, D.J. High density synthetic oligonucleotide arrays. *Nature Genetics*, *21*20-24. Jan, 1999.
138. Litzkow, M., Livny, M. and Mutka, M., Condor - A hunter of idle workstations. in *Proc. of the 8th International Conference of Distributed Computing Systems*, (1988).

139. Liu, G. and Abdelrahman, T.S., Computation-communication overlap on network-of-workstation multiprocessors. in *Proc. of International Conference on Parallel and Distributed Processing Techniques and Applications*, (Las Vegas, Nevada, USA, 1998), CSREA Press, pages 1635-1642.
140. Liu, J., Chandrasekaran, B., Wu, J., Jiang, W., Kini, S., Yu, W., Buntinas, D., Wyckoff, P. and Panda, D.K., Performance comparison of MPI implementations over InfiniBand, Myrinet and Quadrics. in *Proc. of the 2003 ACM/IEEE conference on Supercomputing*, (Phoenix, Arizona, 2003).
141. Lo, J.L., Eggers, S.J., Levy, H.M., Parekh, S.S. and Tullsen, D.M. Tuning compiler optimizations for simultaneous multithreading. *International Journal of Parallel Programming*, *27* (6): 477-503. 1999.
142. Lo, J.L., Emer, J.S., Levy, H.M., Stamm, R.L., Tullsen, D.M. and Eggers, S.J. Converting thread-level parallelism to instruction-level parallelism via simultaneous multithreading. *ACM Trans. Comput. Syst.*, *15* (3): 322-354. 1997.
143. Lowekamp, B. and Beguelin, A., ECO: efficient collective operations for communication on heterogeneous networks. in *Proc. of International Parallel Processing Symposium*, (Honolulu, HI, 1996), pages 399-405.
144. Malony, A.D. and Reed, D.A. Visualizing parallel computer system performance. in *Instrumentation for future parallel computing systems*, ACM Press, 1989, 59-90.
145. Manber, U., Finding similar files in a large file system. in *Proc. of the Winter 1994 USENIX Technical Conference*, (San Francisco, CA, 1994).
146. Marr, D., Binns, F., Hill, D., Hinton, G., Koufaty, D., Miller, J. and Upton, M. Hyper-threading technology architecture and microarchitecture. *Intel Technology Journal*. February, 2002.
147. Massie, M.L., Chun, B.N. and Culler, D.E. The Ganglia distributed monitoring system: design, implementation, and experience. *Parallel Computing*, *30*. 2004.
148. Message Passing Interface Forum MPI-2: A Message-Passing Interface Standard. *International Journal of Supercomputer Applications and High Performance Computing 12* (1/2). 1998.
149. Message Passing Interface Forum MPI: A Message-Passing Interface Standard. *International Journal of Supercomputing Applications*, *8* (3/4). Fall/Winter, 1994.
150. Microsoft Corporation. *Microsoft Office Live Meeting homepage*. 2007. http://office.microsoft.com/en-us/livemeeting/default.aspx.
151. Miller, B.P., Callaghan, M.D., Cargille, J.M., Hollingsworth, J.K., Irvin, R.B., Karavanic, K.L., Kunchithapadam, K. and Newhall, T. The Paradyn parallel performance measurement tool. *IEEE Computer*, *28* (11): pp. 37-46. 1995.
152. Mitra, P., Payne, D., Shuler, L., Geijn, R.v.d. and Watts, J., Fast collective communication libraries, please. in *Proc. of Intel Supercomputing Users' Group Meeting 1995*, (1995).
153. Mogul, J.C., Chan, Y.M. and Kelly, T., Design, implementation, and evaluation of duplicate transfer detection in HTTP. in *Proc. of the 1st conference on Symposium on Networked Systems Design and Implementation - Volume 1*, (San Francisco, California, 2004), USENIX Association.
154. Mogul, J.C., Douglis, F., Feldmann, A. and Krishnamurthy, B. *Potential benefits of delta encoding and data compression for HTTP*. Technical Report 97/4. Compaq Computer Corporation. July, 1997.
155. Mohr, B. and Wolf, F., KOJAK: a tool set for automatic performance analysis of parallel applications. in *Proc. of Euro-Par 2003*, (Klagenfurt, Austria, 2003), Springer-Verlag, Lecture Notes in Computer Science 2790.

156. Muthitacharoen, A., Chen, B. and Mazieres, D., A low-bandwidth network file system. in *Proc. of the eighteenth ACM symposium on Operating systems principles*, (Banff, Alberta, Canada, 2001), ACM Press, pages 174-187.
157. NASA. *NAS Parallel Benchmarks*. http://www.nas.nasa.gov/Software/NPB/.
158. NCBI. *Entrez federated health sciences database* 2006. http://www.ncbi.nlm.nih.gov/gquery/gquery.fcgi.
159. Nieh, J., Yang, S.J. and Novik, N. Measuring thin-client performance using slow-motion benchmarking. *ACM Trans. Comput. Syst.*, *21* (1): 87-115. 2003.
160. Noeth, M., Mueller, F., Schulz, M. and Supinski, B.R.d., Scalable compression and replay of communication traces in massively parallel environments. in *Proc. of International Parallel and Distributed Processing Symposium (IPDPS)*, (Long Beach, CA, 2007).
161. Oleinikov, A.V., Gray, M.D., Zhao, J., Montgomery, D.D., Ghindilis, A.L. and Dill, K. Self-assembling protein arrays using electronic semiconductor microchips and in vitro translation. *J Proteome Res*, *2* (3): 313-319. May-Jun, 2003.
162. Pallas GmbH. *The Vampirtrace/ Vampir tool.* http://www.pallas.com/e/products/vampir/.
163. Patarasuk, P. and Yuan, X., Bandwidth efficient all-reduce operation on tree topologies. in *Proc. of IEEE IPDPS Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS 2007)*, (Long Beach, CA, 2007).
164. Patterson, D.A. Latency lags bandwith. *Commun. ACM*, *47* (10): 71-75. 2004.
165. Petrini, F., Feng, W.-c., Hoisie, A., Coll, S. and Frachtenberg, E. The Quadrics network: high-performance clustering technology. *IEEE Micro*, *22* (1): 46-57. 2002.
166. Petrini, F., Kerbyson, D.J. and Pakin, S., The case of the missing supercomputer performance: achieving optimal performance on the 8,192 processors of ASCI Q. in *Proc. of the 2003 ACM/IEEE conference on Supercomputing*, (Phoenix, Arizona, 2003).
167. Pike, R., Dorward, S., Griesemer, R. and Quinlan, S. Interpreting the data: Parallel analysis with Sawzall. *Sci. Program.*, *13* (4): 277-298. 2005.
168. Plaat, A., Bal, H.E., Hofman, R.F.H. and Kielmann, T. Sensitivity of parallel applications to large differences in bandwidth and latency in two-layer interconnects. *Future Generation Computer Systems*, *17* (6): 769-782. 2001.
169. Policroniades, C. and Pratt, I., Alternatives for detecting redundancy in storage systems data. in *Proc. of the USENIX Annual Technical Conference 2004 on USENIX Annual Technical Conference*, (Boston, MA, 2004), USENIX.
170. Prieto, M., Llorente, I.M. and Tirado, F. Data locality exploitation in the decomposition of regular domain problems. *IEEE Trans. Parallel Distrib. Syst.*, *11* (11): 1141-1150. 2000.
171. Pucha, H., Andersen, D.G. and Kaminsky, M., Exploiting similarity for multi-source downloads using file handprints. in *Proc. of 4th USENIX Symposium on Networked Systems Design & Implementation*, (Cambridge, MA, USA, 2007), USENIX, pages 15–28.
172. Quinlan, S. and Dorward, S., Venti: A new approach to archival data storage. in *Proc. of the 1st USENIX Conference on File and Storage Technologies*, (Monterey, CA, 2002), USENIX Association, pages 7.
173. Quinn, M.J. and Hatcher, P.J. On the utility of communication-computation overlap in data-parallel programs. *J. Parallel Distrib. Comput.*, *33* (2): 197-204. 1996.

174. Rabenseinfner, R., Automatic MPI counter profiling of all users: First results on CRAY T3E900-512. in *Proc. of the Message Passing Interface Developer's and User's Conference 1999 (MPIDC'99)*, (Atlanta, USA, 1999), pages 77-85.
175. Rabin, M.O. *Fingerprinting by random polynomials*. Technical Report TR-15-81 Center for Research in Computing Technology, Harvard University. 1981.
176. Ratanaworabhan, P., Ke, J. and Burtscher, M., Fast Lossless Compression of Scientific Floating-Point Data. in *Proc. of Data Compression Conference (DCC'06)*, (2006), IEEE Computer Society, pages 133-142.
177. Redstone, J.A., Eggers, S.J. and Levy, H.M., An analysis of operating system behavior on a simultaneous multithreaded architecture. in *Proc. of ASPLOS-IX: the ninth international conference on Architectural support for programming languages and operating systems*, (2000), ACM Press.
178. Reed, D.A., Aydt, R.A., Noe, R.J., Roth, P.C., Shields, K.A., Schwartz, B.W. and Tavera, L.F., Scalable performance analysis: The Pablo performance analysis environment. in *Proc. of Scalable Parallel Libraries*, (1993), IEEE.
179. Renambot, L., Rao, A., Singh, R., Jeong, B., Krishnaprasad, N., Vishwanath, V., Chandrasekhar, V., Schwarz, N., Spale, A., Zhang, C., et al., SAGE: the scalable adaptive graphics environment. in *Proc. of the Workshop on Advanced Collaborative Environments 2004*, (Nice, France, 2004).
180. Rhea, S.C., Liang, K. and Brewer, E., Value-based web caching. in *Proc. of the 12th international conference on World Wide Web*, (Budapest, Hungary, 2003), ACM Press, pages 619-628.
181. Ribler, R.L., Simitci, H. and Reed, D.A. The Autopilot performance-directed adaptive control system. *Future Generation Comp. Syst.*, *18* (1): 175-187. 2001.
182. Ribler, R.L., Vetter, J.S., Simitci, H. and Reed, D.A., Autopilot: Adaptive Control of Distributed Applications. in *Proc. of the 7th IEEE International Symposium on High Performance Distributed Computing*, (1998), pages 172-179.
183. Richardson, T. *The RFB Protocol - Version 3.8*. RealVNC Ltd. 5 October, 2006.
184. Richardson, T., Stafford-Fraser, Q., Wood, K.R. and Hopper, A. Virtual Network Computing. *IEEE Internet Computing*, *2* (1): 33-38. 1998.
185. Riesen, R., Brightwell, R., Fisk, L.A., Hudson, T., Otto, J. and Maccabe, A.B., Cplant. in *Proc. of Second Extreme Linux workshop at the 1999 USENIX Annual Technical Conference*, (Monterey, California, 1999).
186. Riverbed Technology Inc. *The Riverbed Optimization System (RiOS) - A Technical Overview of Version 4.0*. A Riverbed Technology White Paper 2007.
187. Rivest, R. *The MD4 message-digest algorithm*. Request for Comments 1320. Network Working Group. April, 1992.
188. Roshal, A. *RAR homepage*. 2007. http://www.rarlab.com/.
189. Roth, P.C., Arnold, D.C. and Miller, B.P., MRNet: A software-based multicast/reduction network for scalable tools. in *Proc. of the 2003 ACM/IEEE conference on Supercomputing*, (Phoenix, Arizona, 2003), IEEE Computer Society Press.
190. Roth, P.C. and Miller, B.P., On-line automated performance diagnosis on thousands of processes. in *Proc. of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, (New York, New York, USA, 2006), ACM Press, pages 69-80.
191. Ruan, Y., Pai, V.S., Nahum, E. and Tracey, J.M., Evaluating the impact of simultaneous multithreading on network servers using real hardware. in *Proc. of the 2005 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, (Banff, Alberta, Canada, 2005), ACM Press.

192. Saeed, A.I., Sharov, V., White, J., Li, J., Liang, W., Bhagabati, N., Braisted, J., Klapa, M., Currier, T., Thiagarajan, M., et al. TM4: a free, open-source system for microarray data management and analysis. *Biotechniques*, *34* (2): 374-378. Feb, 2003.
193. Saldanha, A.J. Java Treeview- extensible visualization of microarray data. *Bioinformatics*, *20* (17): 3246-3248. Nov 22, 2004.
194. Sameer, S., Allen, D.M., Janice, C., Peter, B., Steve, K. and Kathleen, L., Portable profiling and tracing for parallel, scientific applications using C++. in *Proc. of the SIGMETRICS symposium on Parallel and distributed tools*, (Welches, Oregon, United States, 1998), ACM Press, pages 134-145.
195. Sancho, J.C., Barker, K.J., Kerbyson, D.J. and Davis, K., Quantifying the potential benefit of overlapping communication and computation in large-scale scientific applications. in *Proc. of the 2006 ACM/IEEE conference on Supercomputing*, (Tampa, Florida, USA, 2006), ACM Press.
196. Santos, J. and Wetherall, D., Increasing effective link bandwidth by suppressing replicated data. in *Proc. of USENIX Annual Technical Conference*, (1998).
197. Sapuntzakis, C.P., Chandra, R., Pfaff, B., Chow, J., Lam, M.S. and Rosenblum, M., Optimizing the migration of virtual computers. in *Proc. of the 5th symposium on Operating systems design and implementation*, (Boston, Massachusetts, 2002), ACM Press, pages 377-390.
198. Scheifler, R.W. and Gettys, J. The X window system. *ACM Trans. Graph.*, *5* (2): 79-109. 1986.
199. Schena, M., Shalon, D., Davis, R.W. and Brown, P.O. Quantitative monitoring of gene-expression patterns with a complementary-DNA microarray. *Science*, *270* (5235): 467-470. Oct 20, 1995.
200. Schmidt, B.K., Lam, M.S. and Northcutt, J.D., The interactive performance of SLIM: a stateless, thin-client architecture. in *Proc. of the seventeenth ACM symposium on Operating systems principles*, (Charleston, South Carolina, United States, 1999), ACM Press, pages 32-47.
201. SDSS. *Sloan Digital Sky Survey (SDSS)*. 2006. http://sdssdp47.fnal.gov/sdsssn/sdsssn.html.
202. Seward, J. *bzip2 homepage*. 2007. http://www.bzip.org/.
203. SGI. *OpenGL Vizserver*. 2006. http://www.sgi.com/products/software/vizserver/.
204. Shneiderman, B. *Designing the User Interface*. Addison Wesley, 1997.
205. Sistare, S., vandeVaart, R. and Loh, E., Optimization of MPI collectives on clusters of large-scale SMP's. in *Proc. of the 1999 ACM/IEEE conference on Supercomputing*, (Portland, Oregon, United States, 1999).
206. Snavely, A. and Tullsen, D.M., Symbiotic jobscheduling for a simultaneous multithreaded processor. in *Proc. of ASPLOS-IX: the ninth international conference on Architectural support for programming languages and operating systems*, (Cambridge, Massachusetts, United States, 2000), ACM Press.
207. Sohn, A., Ku, J., Kodama, Y., Sato, M., Sakane, H., Yamana, H., Sakai, S. and Yamaguchi, Y., Identifying the capability of overlapping computation with communication. in *Proc. of the 1996 Conference on Parallel Architectures and Compilation Techniques*, (1996), IEEE Computer Society.
208. Sottile, M.J. and Minnich, R., Supermon: A high-speed cluster monitoring system. in *Proc. of IEEE Cluster*, (2002), pages 39-46.
209. Spring, N.T. and Wetherall, D., A protocol-independent technique for eliminating redundant network traffic. in *Proc. of the conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM'00)*, (Stockholm, Sweden, 2000), ACM Press, pages 87-95.

210. Sydor, J.R. and Nock, S. Protein expression profiling arrays: tools for the multiplexed high-throughput analysis of proteins. *Proteome Sci*, *1* (1): 3. Jun 10, 2003.
211. T. Jones et. al. *MPI PERUSE: An MPI extension for revealing unexposed implementation information (Version 2.0)*. www.mpi-peruse.org
212. Tanenbaum, A. *Modern Operating Systems (2nd Edition)*. Prentice Hall, 2001.
213. Tang, H. and Yang, T., Optimizing threaded MPI execution on SMP clusters. in *Proc. of the 15th international conference on Supercomputing*, (Sorrento, Italy, 2001).
214. The Wikipedia community. *Athlon 64 X2*. 2007. http://en.wikipedia.org/wiki/Athlon_64_X2.
215. Theimer, M.M., Lantz, K.A. and Cheriton, D.R., Preemptable remote execution facilities for the V-system. in *Proc. of the tenth ACM symposium on Operating systems principles*, (Orcas Island, Washington, United States, 1985), ACM Press, pages 2-12.
216. Thomas, L.S., John, S., Donald, J.B. and Daniel, F.S. *How to build a Beowulf: a guide to the implementation and application of PC clusters*. MIT Press, 1999.
217. Tierney, B., Johnston, W.E., Crowley, B., Hoo, G., Brooks, C. and Gunter, D., The NetLogger methodology for high performance distributed systems performance analysis. in *Proc. of the 7th IEEE Symp. On High Performance Distributed Computing*, (1998), pages 260-267.
218. Tipparaju, V., Nieplocha, J. and Panda, D., Fast collective operations using shared and remote memory access protocols on clusters. in *Proc. of 17th Intl. Parallel and Distributed Processing Symp.*, (2003).
219. Tolia, N., Andersen, D.G. and Satyanarayanan, M. Quantifying interactive user experience on thin clients. *Computer*, *39* (3): 46-52. 2006.
220. Tolia, N., Kaminsky, M., Andersen, D.G. and Patil, S., An architecture for internet data transfer. in *Proc. of the 3rd conference on 3rd Symposium on Networked Systems Design Implementation*, (San Jose, CA, 2006), USENIX Association, 3.
221. Tolia, N., Kozuch, M., Satyanarayanan, M., Karp, B., Perrig, A. and Bressoud, T., Opportunistic use of content addressable storage for distributed file systems. in *Proc. of 2003 USENIX Annual Technical Conference*, (San Antonio, TX, USA, 2003), pages 127-140.
222. Tolia, N. and Satyanarayanan, M., Consistency-preserving caching of dynamic database content. in *Proc. of Proceedings of the 16th international conference on World Wide Web*, (Banff, Alberta, Canada, 2007), ACM Press, pages 311-320.
223. Top500.org. *TOP500 Supercomputer Sites*. 2007. http://www.top500.org/.
224. Tridgell, A. *Efficient Algorithms for Sorting and Synchronization*. PhD thesis. Australian National University. 2000.
225. Truong, H.L. and Fahringer, T. SCALEA: a performance analysis tool for parallel programs. *Concurrency and Computation: Practice and Experience*, *15* (11-12): 1001-1025. 2003.
226. Tsafrir, D., Etsion, Y., Feitelson, D.G. and Kirkpatrick, S., System noise, OS clock ticks, and fine-grained parallel applications. in *Proc. of the 19th annual international conference on Supercomputing*, (Cambridge, Massachusetts, 2005), ACM Press, pages 303-312.
227. Tuck, N. and Tullsen, D.M., Initial observations of the simultaneous multithreading Pentium 4 processor. in *Proc. of 12th International Conference on Parallel Architectures and Compilation Techniques*, (2003), IEEE Computer Society.

228. Tullsen, D.M., Lo, J.L., Eggers, S.J. and Levy, H.M., Supporting fine-grained synchronization on a simultaneous multithreading processor. in *Proc. of the Fifth International Symposium on High-Performance Computer Architecture*, (1999).
229. Vadhiyar, S.S., Fagg, G.E. and Dongarra, J., Automatically tuned collective communications. in *Proc. of the 2000 ACM/IEEE conference on Supercomputing*, (Dallas, Texas, United States, 2000).
230. Vetter, J., Dynamic statistical profiling of communication activity in distributed applications. in *Proc. of the 2002 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, (2002), ACM Press, pages 240-250.
231. Vetter, J.S. and Mueller, F. Communication characteristics of large-scale scientific applications for contemporary cluster architectures. *Journal of Parallel and Distributed Computing*, *63*853-865. September, 2003.
232. Vetter, J.S. and Yoo, A., An empirical performance evaluation of scalable scientific applications. in *Proc. of the 2002 ACM/IEEE conference on Supercomputing*, (Baltimore, Maryland, 2002), IEEE Computer Society Press.
233. Vinter, B., Anshus, O.J. and Larsen, T., PastSet - a distributed structured shared memory system. in *Proc. of High Performance Computers and Networking*, (Amsterdam, 1999).
234. Wallace, G., Anshus, O.J., Bi, P., Chen, H., Chen, Y., Clark, D., Cook, P., Finkelstein, A., Funkhouser, T., Gupta, A., et al. Tools and applications for large-scale display walls. *IEEE Computer Graphics and Applications*, *25* (4): 24-33. July/August, 2005.
235. Wallace, G.K. The JPEG still picture compression standard. *Commun. ACM*, *34* (4): 30-44. 1991.
236. Werthimer, D., Cobb, J., Lebofsky, M., Anderson, D. and Korpela, E. SETI@HOME - massively distributed computing for SETI. *Comput. Sci. Eng.*, *3* (1): 78-83. 2001.
237. White III, J. and Bova, S., Where's the Overlap? An analysis of popular MPI implementations. in *Proc. of MPI Developer's and User's Conference (MPIDC'99)*, (1999).
238. Wolfgang, B., Michael, K. and Martin, S., Visualizing structural properties of irregular parallel computations. in *Proc. of the 2005 ACM symposium on Software visualization*, (St. Louis, Missouri, 2005), ACM Press, pages 125-134.
239. Wong, F.C., Martin, R.P., Arpaci-Dusseau, R.H. and Culler, D.E., Architectural requirements and scalability of the NAS parallel benchmarks. in *Proc. of Supercomputing '99: the 1999 ACM/IEEE conference on Supercomputing*, (Portland, Oregon, United States, 1999), ACM Press.
240. Woo, S.C., Ohara, M., Torrie, E., Singh, J.P. and Gupta, A., The SPLASH-2 programs: characterization and methodological considerations. in *Proc. of the 22nd annual international symposium on Computer architecture*, (S. Margherita Ligure, Italy, 1995).
241. Wu, C.E., Bolmarcich, A., Snir, M., Wootton, D., Parpia, F., Chan, A., Lusk, E. and Gropp, W., From trace generation to visualization: a performance framework for distributed parallel systems. in *Proc. of the 2000 ACM/IEEE conference on Supercomputing*, (Dallas, Texas, United States, 2000), IEEE Computer Society.
242. Wu, M.-S., Kendall, R.A., Wright, K. and Zhang, Z., Performance modeling and tuning strategies of mixed mode collective communications. in *Proc. of the 2005 ACM/IEEE conference on Supercomputing*, (2005), IEEE Computer Society.
243. Zhou, C., Summers, K.L. and Caudell, T.P., Graph visualization for the analysis of the structure and dynamics of extreme-scale supercomputers. in *Proc. of the*

2003 ACM symposium on Software visualization, (San Diego, California, 2003), ACM Press, pages 143-149.

244. Ziv, J. and Lempel, A. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, *23* (3): 337 - 343. May 1977