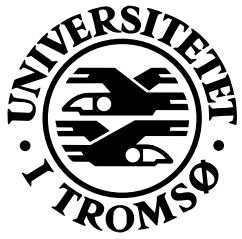
Improving the Speedup of Parallel and Distributed Applications on Clusters and Multi-Clusters

John Markus Bjørndalen

February 3, 2003

A dissertation submitted in partial fulfillment for the degree of Doctor Scientiarum



Department of Computer Science
Faculty of Science
University of Tromsø

To my wife Karen, and my children Christopher, Elisabeth, Rebecca and Miriam.

Abstract

In parallel and distributed computing, clusters are increasingly used for compute- and I/O-intensive applications. As we add computing resources to a parallel application, one of the fundamental questions is how well the application scales, both with regards to speedup and to increasing the problem size.

This dissertation reports on two main issues impacting scaling. The first is the end-to-end communication latency. The other is the configuration and mapping of the application onto a cluster topology and architecture.

Several factors were studied to determine their impact on end-to-end latency, including procotols, workload, and locating communication endpoints at user-, kernel- and interrupt-level.

The dominating contribution to latency comes from complex protocols, such as TCP/IP, which do not take advantage of properties in the interconnect to reduce the amount of processing for communication. The latency reduction from choosing a protocol with lower overhead is approximately an order of magnitude larger than the reduction gained from moving communication endpoints from user-space to the operating system kernel.

When blocking communication is used, a hardware supported implementation of Virtual Interface Architecture (VIA) using a 1.25 Gbit interconnect has approximately the same latency for small messages as a software implementation of VIA using 100 Mbit Ethernet. The performance advantage of the hardware supported VIA implementation is masked by the operating system overhead for interrupt handling, suspending, and resuming the communicating threads.

To avoid this overhead, polling communication can be used. This results in a significant reduction of latency on the hardware supported gigabit VIA implementation. Even in this case, hardware supported user-space communication reduces latency significantly less than using simpler protocols.

To experiment with an applications configuration, an approach and an instrument, PATHS, was developed. Using PATHS, a number of configurations were specified and their performance measured, in order to identify significant behaviours impacting the performance.

A number of benchmarks were used, including a simulation of the Colombia river and estuary. The platform used for experiments was comprised of three clusters, each with 32 processors. The clusters were used individually or together in a number of multi-cluster configurations.

Tuning the performance of distributed and parallel applications is complex. Simple and small changes to a configuration influence scaling and latency. It is hard to compute good configurations. Instead, it is demonstrated that starting with what is believed to be a good configuration, a number of experiments can be run to find configurations with better performance.

The LAM-MPI implementation of the Message Passing Interface (MPI) standard is documented to use configurations which do not scale well, and lacks mechanisms for changing the configuration. A configuration mechanism has been added to LAM-MPI

and used to double the performance of the MPI Allreduce function.

Acknowledgements

The road to a doctorate degree is long and laborious, sprinkled with frustration, despair and lack of sleep. But it is also a road filled with joy, satisfaction, curiosity, and interesting challenges. During such a journey, the people around you become a vital part of being able to finish. There are too many to include here, but these must be mentioned by name:

I would like to thank my advisor, Otto Anshus, for his support and guidance. His advice, and the discussions we have had during this project are greatly appreciated.

Brian Vinter and Tore Larsen for their support and input, and for the long hours we spent writing papers. Brian: I certainly hope we can continue the tradition of setting aside time for interesting hacks. They inspire, fuel creativity, and have even lead to publications.

Otto, Brian, Tore, in addition, I would like to thank you for the discussions I had with the three of you, and for your contribution to my education, not only in computer science, but also in academic life.

Lars Ailo Bongo for input, forwarding interesting papers and asking questions. Also, thanks for working on performance visualization and profiling tools for the PATHS system.

My brother, Ole Martin, for reading and suggesting improvements to the dissertation. Also for discussions, and for taking part in the group's activities.

The other doctoral students and faculty at the computer science department in Tromsø, too many to list them all, for providing an environment with interesting discussions. You were part of the reason I wanted to do this in the first place.

Our technical staff, especially Ken-Arne Jensen, but also Jon Kristiansen and Torfinn Holand for their support over the years. As I understand, it is quite unique to have a technical staff that is not only capable with the technology we use, but also one that you can discuss research with.

Jan Fuglesteg and Svein Tore Jensen for practical help, and for their help in navigating the local bureaucracy.

Antonio Batista and Jonathan Walpole at Oregon Graduate Institute for providing us with ELCIRC, which turned out to be an interesting challenge.

The Department of Mathematics and Computer Science at the University of Southern Denmark, for access to one of their clusters.

Erik Naggum for getting me interested in Common Lisp, a language that put the word 'fun' back into programming when I was tired of C++.

Last, and most important, my parents, my wife Karen, and my children Christopher, Elisabeth, Rebecca and Miriam for having patience with me, and for supporting me over the years.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

A current trend in parallel and distributed computing is that compute- and I/O-intensive applications are increasingly run on cluster and multi-cluster architectures.

As we add computing resources to a parallel application, one of the fundamental questions is how well the application scales. There are two main ways of scaling an application when processors are added: *speedup*, where the goal is to solve a problem faster, and *scaling up the problem*, where the goal is to solve a larger problem (or get a more fine-grained solution to a given problem) in a fixed time by adding computing resources (See also Amdahl [1] vs. Gustafson [24]).

One of the recognized factors in scaling parallel applications is end-point to end-point communication latency.

Scaling an application when it is mapped onto different cluster and multi-cluster architectures involves controlling factors such as balancing the workload between the processes in the system, controlling inter-process communication latency, and managing interaction between the processes. In doing so, one of the main questions is understanding how an application is mapped to the given architecture. This requires an understanding of *what* computations are done *where*, where data is located, and *when* control and data flow through the system.

Trying to understand this in a "black box" design, where the application programmer and profiler only sees a high-level API with object references and operations, is difficult as information about what is happening inside the box is not available.

An example of this is the Message Passing Interface (MPI) [39] collective functions, which have scaling problems if the algorithms of the functions are not mapped properly to the cluster architecture. Understanding why the functions do not scale as well in some configurations as in others is difficult without an exact knowledge of how the functions are implemented and mapped to the cluster architecture.

Even if we discover the reason for the scaling problems, we may not be able to remedy the problem without modifying the source code of the communication library, as mechanisms intended to aid the mapping of the application to the cluster are either insufficient or not implemented [53].

In other cases, implementations of a communication layer or middleware may not have been tested on a cluster of the same size or configuration as the cluster an ap-

plication is deployed at. This may expose new problems which also require intimate knowledge of the implementation to find and resolve [6].

1.1 Research issues

Problem definitions:

1. Investigate the architecture and implementation of communication abstractions with the purpose of determining the end-point to end-point latencies.
2. Investigate the performance of different configurations of applications on clusters with the purpose of determining the importance and sensitivity of the configurations with regards to scaling.

To solve these two problems an approach and an instrument, PATHS, was developed. The approach is to try a subset of the possible configurations, searching for significant behaviours impacting the performance. New configurations are created after analyzing results.

The PATHS instrument was built to facilitate this approach in an experimental situation.

1.2 Limitations

This work started in the context of PastSet [5, 57], a structured distributed shared memory system in the tradition of Linda [15], and builds on local expertise with the PastSet system.

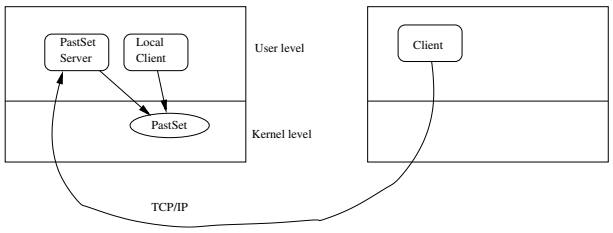


Figure 1.1: PastSet organization: A local process accesses PastSet through a kernel API. Remote processes use a stub-library to access PastSet data on a given node through a user-level server on the node hosting the data.

Figure 1.1 shows the organization of a system using PastSet. PastSet, as implemented in [57], is located in the operating system kernel. Processes access local data through a kernel API. Remote clients access data through a stub library that forwards operations to a user-level PastSet server on the node hosting the data.

Rather than changing the PastSet API by, for instance, introducing asynchronous operations, the synchronous nature of the PastSet API is preserved. The focus is instead

on factors that contribute to improving the performance of the synchronous operations. One of the reasons for this is that a synchronous API is often simpler to use than an asynchronous API.

The location of PastSet and the PastSet servers have not been restricted to the original configuration from [57], which is shown in figure 1.1. Part of the motivation for experimenting with user-level servers is that it is usually easier to deploy them in clusters owned and managed by other organizations.

1.3 Methodology

The project has taken an experimental approach to studying scalability issues. Systems have been built, instrumented and experimented with, and data based on executions of the systems have been analyzed.

Most of the experiments used the following three clusters:

- 2W cluster - 16 * 2-Way (Dual) Pentium III 450 MHz, 256MB RAM. Location: Odense, Denmark
- 4W cluster - 8 * 4-Way (Quad) Pentium Pro 166 MHz, 128MB RAM. Location: Tromsø, Norway
- 8W cluster - 4 * 8-Way Pentium Pro 200 MHz, 2GB RAM. Location: Tromsø, Norway

Most of the visualization and analysis of performance data used ad-hoc tools. Some performance analysis is described in section 3.5 and also in the chapter about using PATHS with ELCIRC in chapter 5.

1.4 Main Contributions

1.4.1 Latency

Several factors were studied to determine their impact on end-to-end latency, including procotols, workload on the server host, and locating communication endpoints at user-, kernel- and interrupt-level.

The dominating contribution to latency comes from complex protocols, such as TCP/IP, which do not take advantage of properties in the interconnect to reduce the amount of processing for communication. The latency reduction from choosing a protocol with lower overhead is approximately an order of magnitude larger than the reduction gained from moving communication endpoints from user-space to the operating system kernel.

There is also a significant potential for improvement by avoiding the operating system through using hardware-supported implementations of API's such as the Virtual Interface Architecture (VIA) [56] for sending and receiving messages.

When blocking communication is used, a hardware supported implementation of VIA using a 1.25 Gbit interconnect has approximately the same latency for small messages as a software implementation of VIA using 100 Mbit Ethernet. The performance advantage of the hardware supported VIA implementation is masked by the operating system overhead for interrupt handling, suspending, and resuming the communicating threads.

To avoid this overhead, polling communication can be used. This results in a significant reduction of latency on the hardware supported gigabit VIA implementation. Even in this case, hardware supported user-space communication reduces latency significantly less than using simpler protocols.

PastSet servers are located on the compute nodes in a cluster. It is shown that using less complex protocols leaves more time for the workload on the compute nodes to proceed with their computation. A benchmark with the communication endpoints at kernel level exchanged 5.3 times as many messages as the benchmark using TCP/IP, while at the same time it didn't slow down the workload any more than the TCP/IP benchmark.

When simpler protocols are used, the latency for remote clients is less influenced by the workload on the server nodes. Kernel-based servers are hardly influenced by workload on the server node.

1.4.2 Configuration

A number of benchmarks, including a simulation of the Colombia river and estuary (see section 1.4.4), were mapped to different cluster and multi-cluster architectures, and their sensitivity to the architectures and configurations were examined.

Small and simple changes to a configuration influence scaling and latency. It is hard to find good configurations analytically or by computation. Instead, it is demonstrated that by starting with what is believed to be a good configuration, one can run a number of experiments to find configurations with better performance.

The LAM-MPI implementation [14] of the MPI standard is documented to use configurations which do not scale well, and lacks mechanisms for changing the configuration. A configuration mechanism has been added to LAM-MPI and used to double the performance of the MPI Allreduce function.

Thus, the ability to tune the configuration with knowledge about the application and the cluster topology, as opposed to relying on the implementation to do important optimization choices, is found to be important.

1.4.3 PATHS

The PATHS system provides a method of specifying how the application is mapped onto clusters, focusing on the location of computations and data. The specification can be changed by modifying meta-data and meta-code, allowing an applications mapping to be studied, tuned, and remapped to a given cluster without recompiling the application code.

PATHS allows the user to specify what type of instrumentation should be used where. The user can also add new types of instrumentation to the system.

PATHS simplifies studying how different configurations influence the performance of an application when it is mapped onto a cluster or multi-cluster. This simplifies experimenting with multiple factors and configurations.

1.4.4 ELCIRC

A simulation model for river and estuary circulations, ELCIRC [20], has been parallelized, and PATHS has been used to control and study the parallel program. Initial experiments show how PATHS can be used to determine bottlenecks, study the applications behaviour, and configure some of the communication and computation.

1.5 Organization of the dissertation

This document is organized as a collection of papers, where the first part includes a synthesis of the results from the papers, and some additional chapters with complimentary work and information. The second part includes the papers.

Chapter 2 provides a synthesis of the results. Chapter 3 describes the PATHS system. Chapter 4 presents ELCIRC, and the parallelization of ELCIRC. Chapter 5 describes experiences from using PATHS with ELCIRC. Chapter 6 presents additional related work. Chapter 7 concludes, and chapter 8 describes future work.

The papers are included in appendix A. Additional information about the current PATHS implementation is presented in appendix B.

Chapter 2

Synthesis of Results

The papers in this thesis focus on two classes of factors. The first class is factors that contribute to the latency of remote operations (primarily in PastSet systems).

The second class is mapping applications to different cluster topologies by experimenting with different parameters (such as communication protocols and data location), and with controlling where computation is done in the PastSet subsystem, and how communication is done.

The papers are included in appendix A.

2.1 Reducing remote operation latency

The PastSet servers are intended to be run on the same nodes as those that run the application workloads. Thus, they must satisfy two main goals: minimizing latency for servicing remote clients, and minimizing the impact on the application processes running on the server's host.

The first goal is important as PastSet has a synchronous API where the calling thread is blocked until the operation returns. Reducing the latency results in less time wasted while the client waits for remote requests to complete.

The second goal is important because servicing remote requests on a host will spend resources which could have been used to execute the application processes on the host. Thus, time spent servicing remote requests must be minimized, and the server should generally be suspended while waiting for remote requests.

The papers presented in this section examine some of the design and implementation options for PastSet servers, and aim at determining which factors should be considered, and which factors contribute most to reducing latency in the design of PastSet servers.

2.1.1 User-space Communication APIs

The main focus of user-space communication APIs is to remove the kernel from the communication path when sending and receiving messages over the network interface.

This removes the overhead of invoking the operating system for sending and receiving messages, but also allows optimizations such as application specific buffer management [58], which can be difficult to support in the kernel.

Early examples of userlevel communication APIs include UNET [58] and VMMC [11]. The two research projects influenced the development of the Virtual Interface Architecture (VIA) industry standard [13] which was backed by a consortium that includes, among others, Intel and Microsoft.

In paper 1 and paper 2, we compare the performance of PastSet remote operations using TCP/IP over Fast Ethernet with two implementations of the VIA API: Giganet cLan [22] and M-VIA [40]. The former is a gigabit (1.25 Gb/s full-duplex) hardware supported VIA implementation. The latter provides a software implementation of VIA on 100 Mbit/s Fast Ethernet. Since it has no hardware support, it uses a fast kernel trap mechanism and a low-overhead driver for the Network Interface Card (NIC) to implement the API. Thus, it does not avoid invoking the kernel upon sending and receiving messages, but the amount of kernel code executed for message passing is small.

Both VIA implementations provide a significant reduction in latency compared to TCP/IP, but in comparing the two VIA implementations we observe that the hardware implementation only provides a marginal improvement in latency for small packets, and only shows an improvement in latency compared to the software VIA implementation when the packet size increases so much that the bandwidth of the network starts to dominate the latency.

The main reason that we fail to see an improvement from userspace communication is that to fully utilize hardware supported VIA, the application needs to use nonblocking communication operations that can poll message queues directly from userlevel.

Blocking operations first poll for a short while to check whether the operation can be completed in userspace, but once a decision is made to block the calling thread the kernel is invoked to suspend the thread. The kernel also needs to enable interrupts on the NIC to provide for waking up the blocked thread when an incoming message arrives. Thus, unless a packet is already available, receiving messages involves both interrupt handling and the operating systems scheduler. Tuning the amount of polling before blocking the process may improve the performance of the server for some conditions[19].

For sending operations, initiating the actual sending of the message can be done from userspace, but the process is likely to block while waiting for the completion of the send operation, or block while waiting for the next message to be received.

A weakness with these experiments was that we only used a single client process for the benchmarks. This means that when a request had been serviced and a reply message sent by the server, it would take a little while for the client to send a new request. Thus, there would be no new request available in the incoming queue, and the server would block, ensuring that the hardware supported VIA implementation does not benefit from polling before blocking. Processing the interrupt and waking up the server when a new request arrives would then add to the overhead of servicing a request.

On the other hand, if the common case is that a new request is available after a request is serviced, it may be that a significant share of the hosts resources is used to service remote clients. In that case, moving the server over to dedicated nodes in the cluster should be considered to reduce the servers impact on the hosts workload.

In conclusion, the overhead introduced by blocking operations masked any advantages that hardware support for the VIA API provided.

2.1.2 Communication Protocols

In paper 2, we observed that since both TCP/IP and M-VIA invoked the kernel to send and receive messages, there had to be other factors than user space vs. kernel space protocols that explained the difference between the performance of the two protocols. A number of factors contributing to the lower latency of M-VIA message passing was found by studying the source code.

One of the more important factors was that the implementation made use of a low-overhead Ethernet-optimized protocol. As such, it avoided most of the processing necessary in Internet-protocols such as TCP/IP. M-VIA was also able to queue packets directly to the network interface card, and provide hooks which allows the interrupt handler to dispatch incoming VIA packets to the M-VIA system instead of sending them to the ordinary Ethernet layer in the kernel.

Thus, comparing M-VIA to TCP/IP does little to evaluate the effect of using the VIA API compared to the BSD socket API, or compare userlevel vs. kernel level protocols.

This observation prompted the development of an experimental low-overhead protocol implemented in the kernel, which used the Linux /proc file system as an interface for sending and receiving messages. The performance was measured to be comparable to M-VIA for sending and receiving messages. This protocol implementation was later used in paper 9, where we compare it with other protocols.

The experiments in paper 9 shows that by using a simpler protocol, we both reduce the latency and the influence that a workload executing on the server host has on the latency (see section 2.1.5).

2.1.3 Thread models

The PastSet server from [57] used a thread pool to service requests. Paper 1 compares the effect of replacing the thread pool with a dedicated thread per client connection. Not only is the latency of remote accesses over TCP/IP reduced, but the variance of the latency is reduced as well.

We attributed this to two factors: firstly, for the thread pool implementation, we iterate over the worker threads (round robin), which, among other things, result in a new stack being used for every operation issued by a remote client. We suspected that this might influence the cache footprint of the server.

Secondly, the thread pool implementation also executes more code than the dedicated thread implementation. The extra code was mainly in the *select()* call used to wait for incoming data on the client sockets, and the mutex protecting the *select()* call[1].

[1] An informal experiment on the same system showed that compared to the *poll()* system call, *select()* had both higher latency and higher variance.

2.1.4 User Level vs. Kernel Level

One method of reducing server access time for remote clients is to move the server or parts of the server into kernel space. Recent examples of doing this include HTTP servers such as Tux, kHTTPd and Microsoft's Scalable Web Cache (a comparison of the performance of these three servers compared with user-level servers can be found in Joubert et. al. [28]).

One of the motivations for locating the server in kernel space is to avoid copying data from the file system buffers up to user level, and then back down to kernel space to send the data through the TCP/IP stack. It also opens up for optimizations such as coordination of disk and network buffering.

An alternative approach to this is to coordinate buffering at kernel level from user-level applications through buffer management APIs. Examples of such functionality is the Linux sendfile system call, IO Lite [41], and the zero-copy datapath in INSTANCE [25].

In paper 9, we benchmark different locations of a simple server (and different protocols) to determine a base-line performance which different PastSet server implementations can be compared against, and to provide information about some of the design choices for PastSet servers.

In addition to locating the server at user-level and kernel-level, we also measure the latency of sending messages between simple servers located in the interrupt-handler. The rationale behind the interrupt-handler server is that the basic PastSet operations only include a very modest amount of computation (mostly lookups in a few data structures) and copying of data in and out of the PastSet elements. This should be comparable overhead to, or less than, the processing currently done in the interrupt handler to allocate a message buffer, copy the incoming message into the buffer and queue the buffer with the Ethernet layer in the kernel.

Compared to the improvements we see from choosing protocols specialized for Ethernet communication (rather than Internet communication), the advantage of moving the server to kernel space is rather modest.

There are two main limitations to this paper: first, the experiments were only made with a single client. It would be interesting to see the effect of adding multiple clients to the experiment. The second limitation is that the experiments were run by moving both endpoints of the benchmark from user-level down to kernel (and interrupt) level. This is representative of a situation where two PastSet server communicate with each other, but the paper should be extended with experiments where one of the endpoints is kept at user-level, corresponding to a user-level client process accessing a server.

2.1.5 Workload on the server host

Paper 9 also introduces another aspect: the influence of protocol and server location on the execution time of workload on the server node, and the influence of the workload on the latency observed by the remote client.

By adding a workload (a matrix multiplication program) to the server end of the benchmark, we observed that the protocols that executed less code were also less influenced by the workload on the computer. The kernel level benchmarks were hardly

influenced by the workload on the computer.

By measuring the execution time of matrix multiplications while executing the benchmarks, we find that the two benchmarks that influence the execution time the least are the TCP/IP and interrupt-based benchmarks. The main reason for this is that the TCP/IP benchmark sends fewer round-trip messages, and that it takes longer time for the client side of the TCP/IP benchmark to receive an answer and send a new request. Thus, the matrix multiplication benchmark is allowed to execute for a longer time between each message.

2.2 Mapping computation and communication

The PATHS system (introduced in paper 5 and described in chapter 3) is used to control how a PastSet application is mapped onto a given cluster or multi-cluster architecture. It allows a user to experiment with different mappings to tune an application to a given architecture, and to study how different choices influence the scalability of a given application. The system is described in more detail in chapter 3.

The PATHS system was first used in the PastSet benchmarks in the URMS (User Redefinable Memory Semantics) paper (paper 4), where we compared the URMS-version of PastSet with LAM-MPI. We found that we could create mappings which allowed us to outperform LAM-MPI on a global reduction function (MPI Allreduce) by a factor 1.83.

Having studied the mappings and verified that the reductions were computed correctly, we concluded that we were able to find a better mapping for the cluster than LAM-MPI had done. An understanding of why LAM-MPI did not perform as well as expected was found in paper 7.

The good results from the URMS paper and the flexibility we believe lies in the PATHS approach prompted us to explore the PATHS idea further.

2.2.1 Mapping applications to different clusters and multi-clusters

Global reduction and Monte Carlo Pi

To experiment with the PATHS configuration system, we ran experiments with two benchmarks: the global reduction benchmark (a benchmark of PastSet's equivalent of MPI Allreduce), and the Monte Carlo Pi benchmark (an emberassingly parallel benchmark).

The two benchmarks were mapped onto three different clusters, consisting of 2-, 4- and 8-way SMP nodes. We also ran a multi-cluster configuration where the PATHS system was used to bind the three clusters together, as the nodes in each of the clusters didn't have direct connectivity to nodes in the other clusters.

Some of the main observations from these benchmarks were:

- Experimenting with multiple factors and configurations can help expose the factors that are most important in a particular cluster, and which combination of factors lead to performance bottlenecks that should be avoided.

An example was found in paper 5, where the sum wrapper was found to scale badly and become a performance bottleneck once it was used by more than 3 or 4 threads. This resulted in more time spent in the sum wrapper than sending messages between the 8-way nodes.

- Performance can be improved significantly without removing or modifying components.

  Instead of rewriting the sum-wrapper, sum-wrappers were arranged hierarchically, to allow groups of 3-4 threads to compute a partial sum which was then forwarded down to a sum-wrapper further down in the hierarchy. This improved the performance of summing partial sums from threads on a node significantly.

- For some configurations, sending more messages on the network than the minimum required may improve the performance. One of the reasons for this is that there is sometimes a tradeoff between the number of messages sent and the parallelism in handling these messages.

  An example is found in paper 5, where sending more messages than the minimum required to implement a global reduction sum reduced the latency by nearly a factor 2 (on the 2W cluster) compared to sending the minimum number of messages.

- Different clusters may need different configuration strategies.

  The results in paper 5 also shows that a strategy which performed best in one cluster was not the best strategy for another cluster.

Wind Tunnel

The *Wind tunnel* application is a Lattice Gas Automaton particle simulation. The application was parallelized by Lars Ailo Bongo, and used for paper 6 and for the Steps paper (paper 10).

The application has a linear speedup for each of the 3 clusters, and for a multi-cluster configuration using both Tromsø clusters. Combining the 2W cluster, which is located in Odense, Denmark, with any of the Tromsø clusters gave us sub-linear speedups. We tried a number of experiments and located some of the factors contributing to the bottlenecks, but did not solve the problem. This was partly due to time limitations, and partly because of a lack of support for variable size PastSet tuples at the time, which limited our ability to experiment with mechanisms such as compression and decompression wrappers.

Video distribution

The *Video distribution* application (paper 6) was an experiment to see whether we could use PATHS and PastSet for real-time video distribution. One of the applications for this is to use PastSet and PATHS for visualizing the output of simulations at run-time.

Feeding images from a frame-grabber into a PastSet element, and using a new *last-observe* wrapper, we scaled the application up to 2016 client processes located in

Tromsø and Odense, Denmark. At that time, the processes closest to the sever started dropping frames as the cluster computers in Tromsø started having problems coping with the amount of data and processes. The processes furthest away from the server (Odense) didn't see any degradation in frame rate.

As we used a hierarchical approach, the video server host was not influenced by increasing the number of clients.

ELCIRC

ELCIRC is a simulation of river and estuary water flows used for the Colombia river system.

The application is parallelized, and PATHS is used to study the performance aspects of the parallelized application. This project is not yet at a stage where any papers have been written. ELCIRC and parallelization of ELCIRC is described in chapter 4. Experiences with using PATHS on ELCIRC and on studying some of the performance aspects of ELCIRC is described in chapter 5.

2.2.2 Adding configurable collective communication to LAM-MPI

In paper 7, we address the hypothesis that PATHS allowed PastSet to scale better than LAM-MPI on a global reduction benchmark due to a better mapping to the cluster architecture.

LAM-MPI implements the reduce and broadcast phase of the Allreduce operation by organizing the processes into static operation trees. For the broadcast phase, the root process in the tree sends messages to its direct children. Internal nodes in the tree then forward messages to their children until all processes in the three have received a message.

LAM-MPI uses different Broadcast and Reduce trees, neither of which are modified to take into account the topology of the cluster. Figure 3 in paper 7 shows part of the reason LAM-MPI did not scale well: the Reduce operation tree maps badly to the topology of the 4W cluster, and ends up sending one message per client over the network.

To verify that our understanding was correct, and to experiment with optimizations of the trees, LAM-MPI was modified to allow the broadcast and reduce trees to be configured at load time.

Using this configuration mechanism, we were first able to closely match the performance of LAM-MPI when scaling the application (to verify that adding the configuration mechanism did not impact the performance). Secondly, we were able to close the gap between LAM-MPI and PastSet on Allreduce by improving the performance of LAM-MPI by a factor 1.79 (compared to the factor 1.83 difference between unmodified LAM-MPI and PastSet with PATHS).

On the 8-way cluster, we showed that the best configuration does not always involve minimizing the number of messages in the network. By sending twice as many messages on the network as strictly necessary, we improved the performance of Allreduce on the 8-way cluster by a factor 1.98 compared to the native implementation.

This result correspond to our experiences with PATHS that shows that as the application is scaled, minimizing the use of what is intuitively considered the highest cost factor (messages on the network) does not always minimize the latency.

None of these experiments were carried out to find the optimal configuration. Rather, the goals were to see whether we could find better configurations than the unmodified LAM-MPI, and whether we would replicate the performance of PastSet using PATHS.

Multi-cluster LAM-MPI

Paper 8 extends this work further by running experiments on a multi-cluster configuration, with two clusters over a WAN link between Odense, Denmark and Tromsø, Norway. The WAN link has a round-trip latency for small messages (measured with Unix ping) of 30-50 milliseconds. The clusters were connected by using an IP tunnel between Odense and Tromsø.

These experiments showed that a bad configuration (all processes in one cluster communicating directly with processes in the other cluster rather than minimizing the number of messages across the WAN link to 1 per reduce or broadcast) can be masked by properties of the WAN link. For a limited number of small messages, the WAN link is working like a pipeline that is able to accept new messages while the first message is in transit. Thus, the latency of the Allreduce operation is not influenced much when scaling up the number of processes.

In these experiments, we did not make the same observations as in [32], where they showed a reduction of latency by minimizing the number of messages across a WAN link. One possible explanation for this is that their experiments were run on multi-clusters where the clusters were of different sizes. Our experiences from experimenting with mapping the LAM-MPI operations down to different multi-cluster configurations show that it is easy to get configurations where a path from a leaf node in the operation tree to the root node crosses the WAN link multiple times. The number of times such a path crosses the WAN link will limit the latency of the operation more than the number of messages crossing the WAN link.

The peak at the end of figure 8 in paper 8 shows a situation where we got such a configuration by scaling up the number of processes beyond the number of processors in the multi-cluster.

2.3 Cluster Components

In paper 3, 6 benchmark applications are used to examine properties of two-, four- and eight-way multiprocessor hosts used as cluster components. For large problem sizes, two of the applications favor large nodes, two of the applications are indifferent to node sizes, and two favor small node sizes.

Chapter 3

PATHS

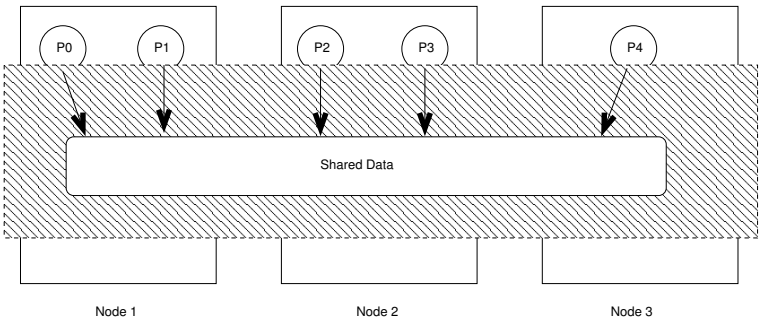


Figure 3.1: A distributed system with multiple processes (circles) mapped onto three nodes (rectangles). The processes communicate and share information using a communication library or middleware (shaded box) that hides details about communication and location of objects. In this example, the processes access shared data.

A middleware system or communication library usually provides the user with an API that abstracts away low-level details about how communication is implemented and where objects and shared data is located. Figure 3.1 shows an example where a number of processes located on three hosts access shared data. The location of the data and how the data is accessed is transparent to the application programmer.

Although this simplifies programming distributed applications, it is difficult for the programmer to determine why some of the functionality provided by the API does not scale well, and where bottlenecks occur. Even if the user discovers a bottleneck or the reason for scaling problems, which may require intimate understanding of the API implementation [6], it may be difficult or impossible to solve the problems without modifying the implementation.

The PATHS system allows a user to specify how the functionality behind an API is implemented and mapped to a cluster by controlling *what* is computed *where* and where data is stored. PATHS also provides instrumentation to identify *when* computation and

communication occurs.

The user experiments with different specifications to study and tune performance aspects of an applications mapping. This helps the user identify which factors are most important for scaling the application.

The PATHS specification is used both for setting up the system and for inspecting the system, reducing the chances of in-coherency between the specification used for building and studying the system.

PATH specifications can be changed without modifying the application code.

3.1 The Path Specification

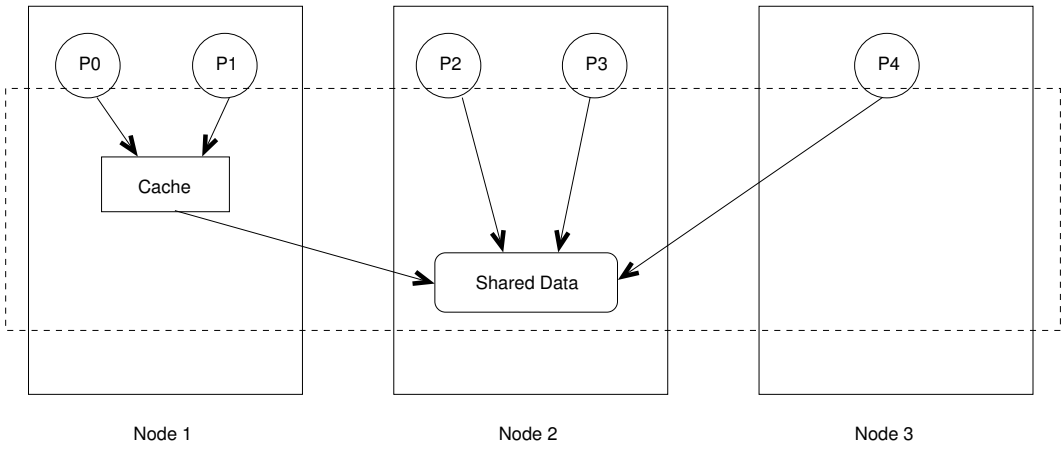


Figure 3.2: A mapping of the system from figure 3.1. The location of the shared data is exposed, and process 0 an 1 share a cache of values read from the shared data.

Figure 3.2 shows a mapping of the processes and the shared data from figure 3.1 to the cluster. The data is located in node 2. In addition, the two processes on node 1 share a cache of values read from the shared data.

The *path* from process p_0 to the shared data can be described as a sequence of stages, where each stage identifies the computation done in that stage, and parameters that control the execution of that computation. In this case, the path goes from p_0, through a cache and on to the shared data.

In PATHS, a *path specification* is a sequence of *stages*, where each stage identifies a *Wrapper class* and the parameters used to initialize the wrapper instance at that stage. A process gains access to remote data by asking the PATHS system to build a path to the data according to a path specification. The application can then invoke operations through the topmost wrapper in the path.

Paths can be joined to share resources. An example in figure 3.2 is process p_0 and p_1, which share the cache and thus the path from the cache to the shared data. The

PATHS system automatically joins paths by comparing path specifications[1]. Current implementations compare the specifications from the last stage towards the first, sharing only the last portion of the paths that have matching specifications.

By combining all path specifications in an application, a specification of how the application is mapped to the cluster is created. This is called a *path map*.

3.1.1 The Remote Operation Wrapper

Remote operations are reified in the PATHS system with *Remote operation* wrappers. The remote operation wrappers are specified with parameters identifying which protocols to use and which server the next wrapper in the path is hosted by[2].

The remote operation wrappers have two purposes in the path specifications: the first is to provide a mechanism for calling operations on a remote server (using RPC).

The second purpose is to add the spatial (*where*) information to the path. By following the path from the process to the shared data and paying attention to the remote operation wrappers, an analysis tool can deduce the server hosting each of the wrappers in the path. Each time a remote operation wrapper is encountered, the tool knows that the next wrapper in the path is located in the server that the remote operation wrapper points at. The location of this server can be found by inspecting the initialization parameters of the remote operation wrapper.

An advantage of doing this is that the same information in the specification is used both to deduce the location of the wrappers and to implement the RPC calls. This reduces the possibilities of inconsistencies between the application and the specification.

3.1.2 Path map example

Figure 3.3 shows a path map of the application from figure 3.2 when the remote operation wrappers are included in the paths. The two processes on node 1 access a cache, which is located in a server on the node. The shared data is located in a server on node 2.

The path from process p_0 to the shared data is comprised of the following stages:

- A remote operation wrapper. Parameters include which protocol to use and the address to the server (in this case to the server on the same host).
- A cache wrapper, which caches values read from the shared data. Parameters may be size of the cache, or caching policy.
- A remote operation wrapper, pointing to the server on node 2.
- The shared data.

[1] Unique tokens can be attached to stages to prevent this default behaviour whenever a process needs a unique path

[2] Other parameters such as service requirements may be specified as well

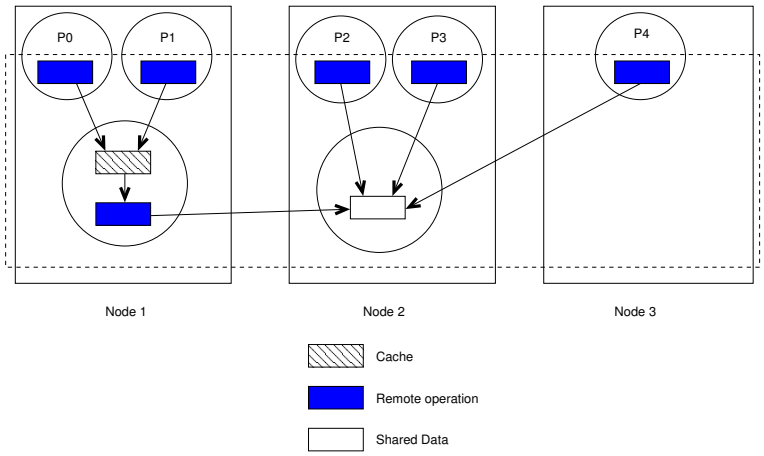


Figure 3.3: A path map of the configuration from figure 3.2.

3.1.3 Reasoning about "what", "where" and "when"

Using the path specification and paying attention to the remote operation wrappers, it is possible to reason about *what* is done *where* for a given mapping.

To reason about *when* operations are called and data flows through the system, we need to add another factor in the PATHS system: instrumentation. Instead of adding a specific tracing functionality to the PATHS runtime system, we use instrumentation wrapper classes, which opens up for experimenting with different types of instrumentation. The tracing wrappers are described in section 3.4.

3.2 PATHS Architecture

Figure 3.4 shows the organization of a client application using PATHS. The current PATHS implementation is based on the PastSet[3] system from [57], where PATHS is used to orchestrate the mapping of a PastSet application to a cluster.

The PATHS-enabled PastSet implementation contains a library of wrappers and PastSet Elements (the shared data entities in PastSet). The Remote operation wrappers communicate with PastSet servers using a simple RPC library that supports the use of multiple communication protocols, such as TCP/IP and VIA.

Paths are built by handing a path specification to the *Path management library*, which uses the specification to instantiate the wrappers (and elements) and combine the instances to form a path.

Remote operation wrappers are bound to the next stage in the path by handing the path specification (minus the part up to and including the remote operation wrapper), to the remote server. The remote server builds the rest of the path and returns an identifier

[3] PastSet is a structured distributed shared memory system in the tratidion of Linda[15]

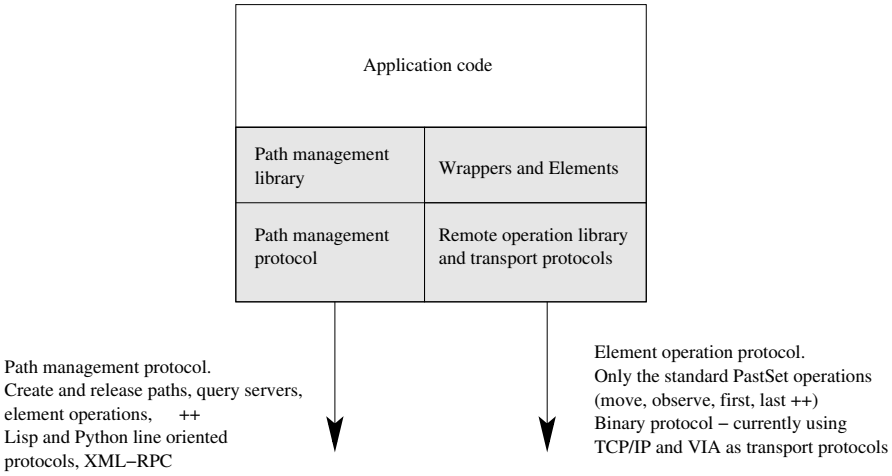


Figure 3.4: Organization of a PATHS enabled client application

to the first wrapper of the path in the remote server. This identifier is used by the remote operation wrapper to invoke operations on the remote wrapper.

Path management is supported through a *path management protocol*, which is mainly used to create and release paths, and to inspect the path specifications of existing paths in the system. The path management protocol also includes ordinary PastSet operations, allowing tools without a full PastSet and PATHS implementation to create and release paths, and execute operations on them.

3.3 Using PATHS

Setting up access from a thread to a PastSet element involves two steps: 1) retrieving a path specification, and 2) building the path according to the specification.

For the PastSet PATHS system, an application can also choose to use the standard PastSet API, which hides the retrieval and building of a path behind the PastSet *Enter* function[4]. In this case, the use of the PATHS system is not visible in the application source code.

Alternatively, a process can retrieve a path specification and ask the PATHS system to build the path directly. There are currently three ways to retrieve path specifications:

- The application program can specify a path directly. This is rarely used, and has for the most part been used in benchmarks.
- A path specification module is queried for a specification.

  The path specification module is loaded when PATHS is initialized. Which module to use can be specified by the user. The current implementation loads Python

[4]Enter is used to get a reference to a named PastSet Element

modules. Users may write their own modules to implement their own methods of creating specifications.

- A *path server* can be queried for a path specification. This is equivalent to querying the specification module, but avoids having to distribute updated path specification modules to the nodes in the system when changes are made to the configuration.

After retrieving a path specification, the path is built from the specification. This involves creating wrappers with parameters from the specification and binding the created wrappers together.

3.3.1 Path specification example

Process	Path specification created in Python

p_0	`spec = make_path(stage("remote", proto=TCP),`
	`stage("cache", size=64),`
	`stage("remote", proto=TCP,`
	`host="node2"),`
	`stage("SharedData"))`

p_2	`spec = make_path(stage("remote", proto=TCP),` `stage("SharedData"))`

p_4	`spec = make_path(stage("remote", proto=TCP,`
	`host="node2"),`
	`stage("SharedData"))`

Figure 3.5: Example path specification in Python

Figure 3.5 shows how path specifications can be created manually for some of the processes in figure 3.3. The path created by calling `build_path` with the specification and the name of the PastSet Element:

```
path = build_path("Zebras", spec)
```

Operations can then be called on the reference returned by `build_path`.

3.4 Instrumentation

Instrumentation is supported in PATHS by adding instrumentation wrappers rather than adding a special facility in the run-time system. Two trace-wrappers are included in the current PATHS implementation:

- A *trace wrapper*, which logs the start and completion time (using Pentium timestamp counters) of operations called through it, as well as the operation type.

  Traces are recorded in memory, and stored to disk when the trace wrapper is not used any more.

  The overhead of calling operations through this wrapper has been measured to be around 100-120 CPU clock cycles [9].

- A *debug wrapper*, which is a trace wrapper that has been extended to additionally log arguments to operations and returned data.

An advantage of using wrappers to add instrumentation is that new types of instrumentation can be added without changing the PATHS system itself. An example of this is the EventScope system [12], where new tracing and monitoring wrappers are added to support run-time monitoring of PATHS applications.

3.5 Performance data analysis

By combining the path map with trace wrapper data, we have the information necessary to implement profiling and performance analysis tools. An example use of the performance data is shown in section 5.1, where trace data and the path specifications were used to pinpoint bottlenecks in the ELCIRC application.

Further work on tools for analysing performance aspects of applications is in progress. Visualization and profiling tools are presented in paper 10 and in [12].

3D visualization of performance data is created by drawing a 2D path map in the XY plane and extending a time-line from each trace wrapper parallel to the Z axis. Operations going down (start time) and up (completion time) are marked on the time-lines. VRML files are generated, allowing a user with a standard VRML browser to fly through the time history of the application, inspecting patterns in its behaviour.

Initial experiments suggest that this may be a useful method of visualizing an application's behaviour. We plan to experiment more with 3D-visualizations in the future. More extensive 3D visualizations can be found in the Avatar system [42][43].

3.6 Debugging

The debug wrappers have, for the most part, only been used to debug other wrappers by making use of the ability to inspect how the data flows and how data is modified along the paths.

The path specifications also provide information that can be used for inspecting some of the structural and communication properties of the system. As an example, section 5.2 describes an experience where a configuration error was found by discovering that a small island of two processes had no communication channels (or paths to shared elements) with any of the other processes in the system.

3.7 Summary

The PATHS system allows a user to specify how the functionality behind an API is implemented and mapped to a cluster by controlling *what* is computed *where* and where data is stored. PATHS also provides instrumentation to identify *when* computation and communication occurs.

The user experiments with different specifications to study and tune performance aspects of an applications mapping. This helps the user identify which factors are most important for scaling the application and try alternative mappings which may resolve the problems.

A *path specification* is a sequence of stages, where each stage is a description of the wrapper type used to implement that stage and the parameters used to initialize the wrapper.

PATH specifications can be changed without modifying the application code. Thus, the PATHS system is orthogonal to the API provided by the middleware or communication library.

Tracing and profiling is supported through trace wrappers rather than building tracing within the PATHS system. This opens up for adding new types of tracing and monitoring to the PATHS system (as has been done in [12]).

Chapter 4

ELCIRC

ELCIRC [20] is a finite volume model that is used to simulate the coastal and estuary circulations for the Colombia River and the west coast of the USA. It can forecast parameters such as water speeds, elevation, temperature, and salinity. The ELCIRC model is used in the CORIE project [16], which is a pilot environmental observation and forecasting system for the Columbia River. ELCIRC and CORIE are developed at Oregon Graduate Institute (OGI).

Figure 4.1 shows an example visualization of one of the parameters, salinity, produced by ELCIRC. The color coding shows the salinity of the water for a given point in time. The output of the model can be used for tasks such as studying habitat conditions for wildlife in the region.

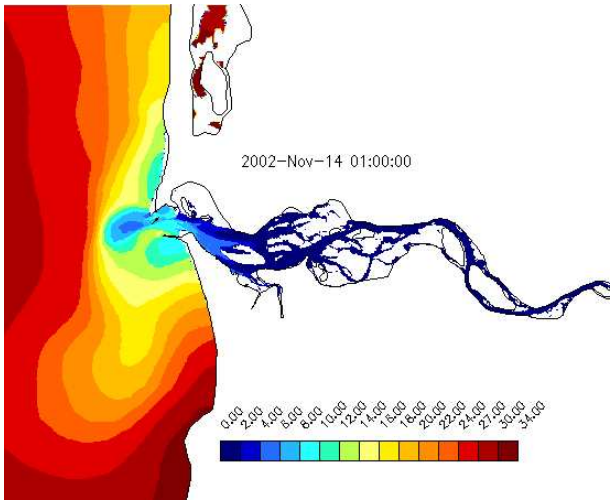


Figure 4.1: ELCIRC sample visualization: salinity in the Columbia River and coastal waters outside the river estuary. The figure is copied with permission from the CORIE homepage.

To reduce its execution time, ELCIRC was first tuned by experimenting with code optimization techniques. The model was then parallelized to reduce the execution time further. PATHS was used to implement and control the communication between the processes in the parallel model. The PATHS instrumentation wrappers were used to study some of the performance aspects of the application.

The parallelization experiments used version 3.10 of ELCIRC along with a sample data set, both provided to us by OGI. The ELCIRC source code is written in Fortran 77 (with some GNU F77 extensions).

This chapter, and chapter 5, present some of the work done on parallelizing ELCIRC, and on some of the experiences from using PATHS with ELCIRC. Some alternative approaches and implementations have been left out since they didn't provide any extra insight into using PATHS.

4.1 Overview and terminology

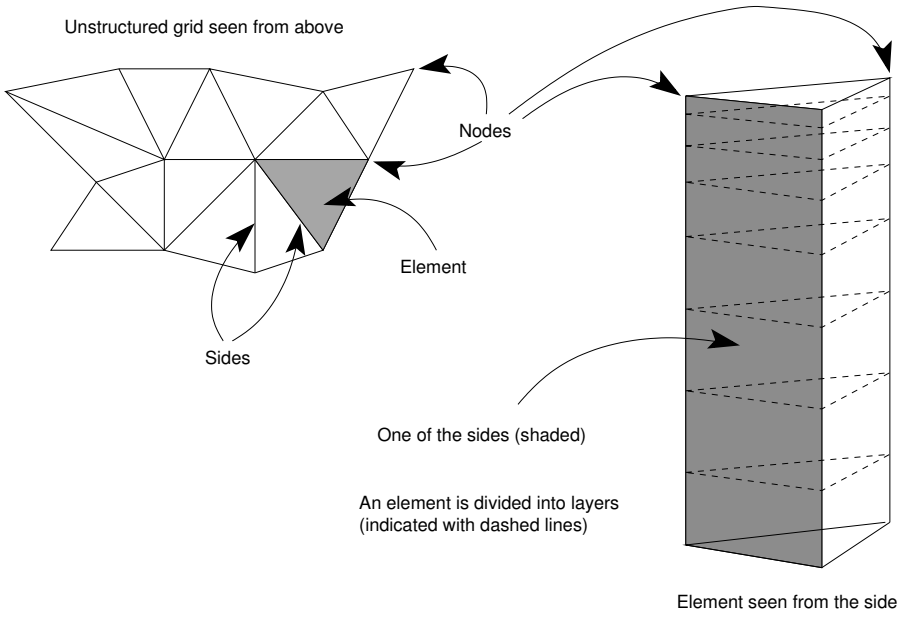


Figure 4.2: Unstructured grid with elements, sides and nodes in ELCIRC

ELCIRC simulates the flow of water in a 3-dimensional grid which is unstructured in the horizontal plane. In an unstructured grid, the control points are not required to be positioned at regular intervals. The density can be different in different regions of the grid, allowing higher density in important regions of the simulation. An example of an unstructured grid is shown in figure 4.2. The grid in ELCIRC is divided vertically in a number of layers that are defined globally.

The following terms are defined in ELCIRC (see figure 4.2):

Element A column of water in the grid. The Element is described by a polygon in the horizontal plane. In the current data set all polygons are 3-sided.

The element is divided vertically into a number of *layers*. The vertical position of each layer is the same for all elements in the data set.

A *wet* element is an element with water in it.

Nodes are the vertices of the polygon defining the Element in the horizontal plane.

Sides are the edges of the polygon defining the Element in the horizontal plane. The sides extend down vertically.

Water and currents as well as other parameters are introduced into the 3D model with *boundary conditions*, which are tables and functions that reflect the change of a parameter over time.

Boundary conditions are defined for sides and elements, and can be used for introducing parameters such as tidal waves.

Elements and sides which have defined boundary conditions are handled specially at various points in the model, potentially overwriting values computed earlier in the time step.

4.2 Tuning

4.2.1 Sequential code optimizations

The simulation loop was restructured into stages to simplify experimentation with the application code.

The execution times of the stages change during the first few iterations of the model, but stabilize within a few percent after a few iterations. The two exceptions to this is the backtracking stage, which increases and decreases execution time slowly[1], and the output stage which stores output from the model only every nth round (the current data set stores output every second round). Figure 4.3 shows an approximation of the relative times taken in the sequential application during an iteration where ELCIRC stores output data to disk.

Working with the ELCIRC source code, it was found that some of the terms computed by the program could be removed when particular combinations of parameters canceled out terms in the model. This suggests that specializing the program for a given data set (using techniques such as dead code removal and partial evaluation) may be worthwhile to investigate for future versions of the ELCIRC model.

The execution time of ELCIRC can also be reduced by improving cache utilization, and by loop restructuring[2].

[1] This is probably linked to the tidal waves, as the tops seems to come about 12 hours apart, but whether this is the real cause has not been verified

[2] As an example, moving calls to *sin* and *cos* a few levels out from the inner loops and inverting a few of the loops in the *nodalvel* function reduced the number of cycles spent in this function from 2607 million cycles to 1336 million cycles. Some of this may be realized automatically by a good compiler.

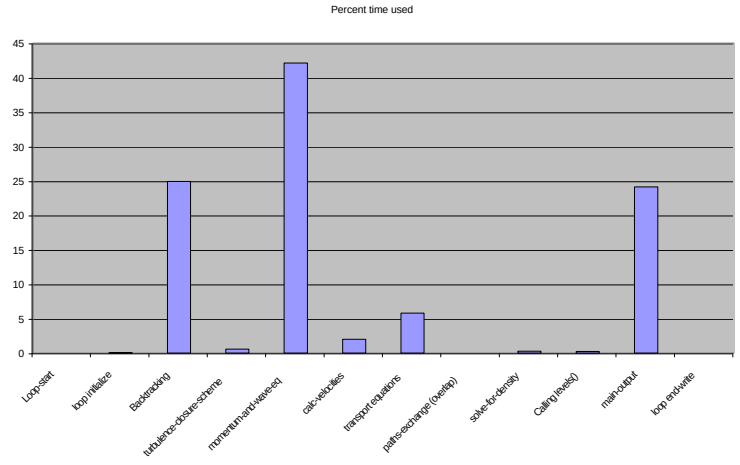


Figure 4.3: Time spent in the stages of the ELCIRC simulation loop. The bars reprecent the percentage of the total time spent in the simulation loop.

Compiler	Execution time in minutes:seconds
f2c + gcc 2.95.4	15:22
f77 (gcc 2.95.4)	13:54
f77 (gcc 3.2)	9:20
Intel FC	6:50

Table 4.1: Execution time (including initialization and exit) of ELCIRC when simulating only 26 iterations using different compilers on the sequential code.

4.2.2 Compiler experiments

Table 4.1 shows the execution time of ELCIRC benchmarked with different versions of GCC and with Intel's Fortran compiler, version 6.0. In addition, the f2c (Fortran-to-C) translator was used to convert the ELCIRC source code to C, and the C code was compiled using one of the GCC versions.

The table shows that the execution time of ELCIRC can be improved significantly by replacing GCC 2.95.4 with a newer GCC or with Intel's Fortran compiler.

4.3 Parallelizing ELCIRC

To create a parallel version of ELCIRC, the grid is partitioned geometrically and the partitions assigned to processors in the cluster[3]. An advantage of this is that most of the communication for any process in the system is communication with the processes handling the neighbour geometry.

By studying the ELCIRC source code with respect to the data exchange needed between partitions, three classifications of computations in the model were found:

basic Most of the expressions in the model only use information from the element it currently computes for, or from the element and the immediate neighbour elements which it share an side with.

To support this type of computation, two neighbour partitions only need to share information about elements which use any of the sides that divide the two partitions.

backtracking The backtracking stage of ELCIRC determines where water that flowed into a given element came from. An example use of this information is to determine the temperature of the water within a given element by looking at the temperature of the water that flowed into the element.

To run the backtracking algorithm, each partition needs information about the grid in an area around the partition (called ghost region) which is large enough to include the water that might flow into the partition during one iteration of the model. Information about water speeds, and some other parameters (such as water temperature and salinity) is also needed for the backtracking algorithm and for later computations.

Figure 4.4 shows two partitions and the ghost regions for the two partitions.

solver During the momentum-wave phase of the application, a large matrix equation of the form $Ax = b$ is built up and solved for x (which is a vector representing the elevation of the water in each element). Each row in the equation represents an element, and can be built up using information about the element and the neighbours of the element (it is based on a relation between the element and the elements that share sides with the element).

The equation can easily be built up in parallel, where each partition is responsible for building up the rows for its local elements. The question then is just how to run the solver (and which solver to use).

The ghost region that is exchanged between neighbour partitions thus corresponds to the body of water which may flow between the partitions. We currently set an assumed maximum water speed and multiply it with the length of the iteration time step to determine how large the overlap regions must be.

[3]This technique is called Domain Decompositioning [18]. Using parallelized functions was evaluated, but found to have a lower potential for scaling than Domain Decompositioning

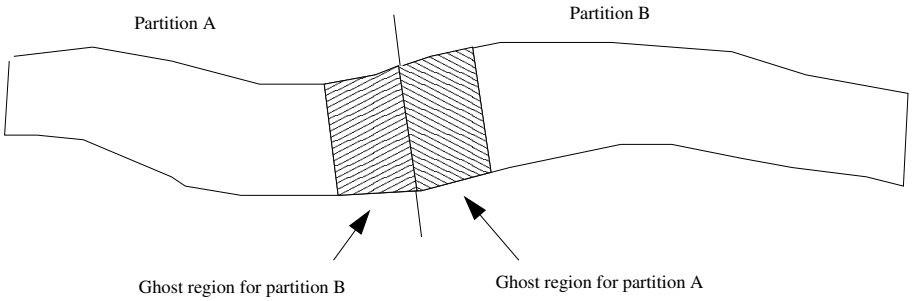


Figure 4.4: Partitioning a small river in two subdomains. The shaded areas are ghost regions, which define bodies of water which may flow into the other partition during one iteration of the model. The size of the ghost regions depend on the expected maximum speed of the water in the region and length of the time step in the model.

4.4 Implementation of Domain Decomposition

The original data set was partitioned by generating, for each partition, a smaller data set which only include the information necessary for the process computing for that partition (including the ghost regions). The sequential source code, with modifications to update ghost regions from neighbour partitions, was then run for each of the partitions.

The two main advantages of this approach are: first, that ELCIRC can be run nearly unmodified. The main changes involves adding the code to update the ghost regions. Second, less memory is required for each process since the processes only need to allocate memory for the partition and ghost regions instead of the entire system. This provides us with an option to scale to large data sets which may not fit in one computer.

A drawback with this approach is that we end up running the model for the ghost regions of a partition since the model doesn't differentiate between elements internal in the partition and the ghost elements. The values computed for the ghost region are then replaced when we update the ghost regions from the neighbour partitions.

Thus, the relative size of the ghost regions and the partitions limits the scalability of the application since large ghost regions relative to small partitions may result in a high communication to computation ratio. It may also limit scalability as the number of wasted computations (ghost regions) relative to productive computations may be large.

4.5 Reducing the number of exchanged arrays

Some of the arrays in ELCIRC are computed based solely on other arrays computed earlier in the loop. Recomputing these arrays in each partition instead of transmitting them reduces the size of the updates transmitted for each ghost region. By arranging the arrays changed in the main loop of ELCIRC in a dependency graph, the number of arrays to transmit for ghost regions is reduced to 4.

One problem that may appear is that a partition does not have any information about the geometry outside the ghost regions. Thus, some computations that are based

Experiment	Execution time in hours:minutes:seconds	Speedup	Utilization
8-way Pentium Pro, sequential	13:44:30		
4 x 8-way Pentium Pro, parallel	58:50	14.01	0.539
2-way P4, sequential	2:24:04		
11 x 2-way P4, parallel	9:54	14.55	0.661

Table 4.2: Execution time of 24 hours simulated time in the ELCIRC model (including initialization time).

on backtracking and information about neighbour elements may result in less accurate results close to the outer edges of the ghost regions.

Areas for future study is whether these inaccuracies are large enough to cause any problems, and in that case, how to compensate for this.

4.6 Performance results

To measure the speedup using the parallelized ELCIRC application, we compared the wall clock time (measured with the Unix *time* command) of the processes running in parallel. This time was compared to the wall clock time of running the sequential version of the code. In both the parallel and sequential executions, the model was simulating a 24-hour period.

26 partitions were created with a manual partitioning tool. The tool was too cumbersome to use to get the wanted number of partitions[4], and to get a balanced number of elements in each partition (the smallest partition had 689 elements, the largest partition had 2779 elements, including ghost elements).

Thus, these experiments were run to get an initial understanding of how well the parallel ELCIRC version scales. It also provided us with an opportunity to examine a badly balanced application.

The manual tool has since been replaced by an automated tool which uses Metis [29] to partition the grid.

4.6.1 Speedup with 26 partitions

Two scaling experiments were run. One experiment was run on a cluster of 4 8-way HP Netserver nodes (Pentium Pro 200 MHz with 2 GB Ram each) interconnected with 100 MBit Ethernet. When partitions were assigned to nodes, care was taken to limit the communication between each of the 8-way nodes. The speedup on this cluster was approximately 14 (see table 4.2).

The second experiment was run on a cluster of 11 Dual Pentium 4 nodes[5] interconnected with gigabit Ethernet. Since the cluster had fewer than 26 processors, and

[4]The goal was 32 partitions, assigning one partition to each of the 32 processors in the 8-way cluster

[5]This cluster is located at the computer science department at OGI, which kindly allowed us to use it for benchmarking ELCIRC.

repartitioning with the manual tool was a cumbersome task, processes were placed such that two of the nodes had 4 of the smallest partitions each, and the rest had 2 partitions each. No attempt was made to place neighbour partitions close to each other in the cluster.

The speedup on this cluster was approximately 14.6 (see table 4.2).

Utilization

The utilization column of table 4.2 is computed by dividing the gained speedup by the number of processors used (26 in the 8-way cluster, and 22 in the P4 cluster). A utilization of 1 means that an application computes N times faster when using N processors.

The numbers show that by mapping a badly balanced application to a cluster with fewer processors than processes (the P4 cluster), we can overlap some of the waiting time with computation, which utilizes the cluster better.

4.7 Effect of domain decompositioning on the accuracy of ELCIRC output

4.7.1 Comparing output of sequential ELCIRC using different compilers

Modifying the order that floating point operations are applied to numbers may affect the result of those operations. As an example, summing a list of floating point numbers head to tail may produce different results than summing the same list of numbers tail to head.

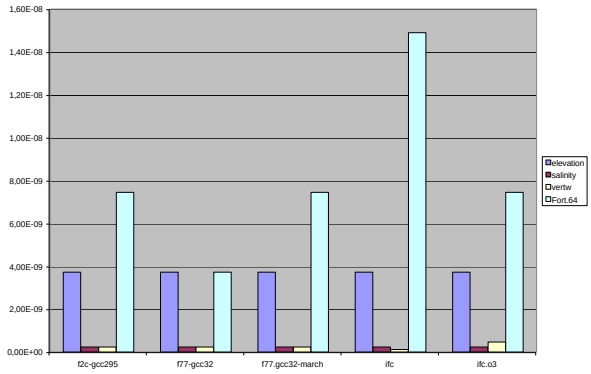
This problem was observed when modifying the sequential ELCIRC code. Although the differences were small and only in the least significant bits of a few floating point numbers, the end result was that differences propagated throughout the model after every iteration.

Knowing that our code restructuring and sequential code optimization may introduce small differences in the output, it is interesting to see how much effect using different optimizing compilers have on the output from the simulation, and whether our changes to the code result in differences larger than the ones observed by using different compilers and optimization flags.

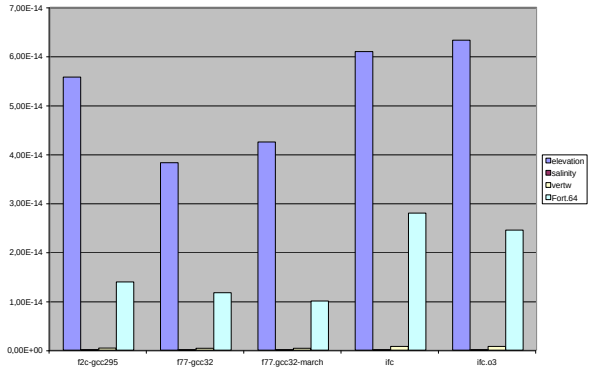
The differences observed also serve as a baseline for comparing differences introduced when a parallel implementation of ELCIRC is run.

Figure 4.5 show the maximum and average difference observed when comparing every value over every dumped timestep of the sequential code compiled with a reference compiler with the output from other compilers. The compilers and flags used were:

- The reference compiler, GNU Fortran 77, version 2.95.4, optimization flags -O5 -funroll-loops -fomit-frame-pointer



(a) Maximum difference



(b) Average difference

Figure 4.5: Maximum and average difference between elevation, salinity and horizontal/vertical speeds in output files from sequential model using different compilers and compiler options. The reference compiler is the GNU Fortran 77 compiler, version 2.95.4.

- f2c (Fortran to C translator), C code compiled with gcc 2.95.4, optimization flags -O2 -funroll-loops
- Fortran 77 compiler, gcc 3.2, optimization flags -O5 -funroll-loops -fomit-frame-pointer
- Fortran 77 compiler, gcc 3.2, optimization flags -O5 -funroll-loops -fomit-frame-pointer -mcpu=pentium4 -msse2 -march=pentium4

- Intel Fortran compiler, version 6.0, optimization flags -tpp7 -xKW -ipo
- Intel Fortran compiler, version 6.0, optimization flags -tpp7 -xKW -ipo -O3

As seen in the figure, both the maximum and average differences introduced are small. The maximum differences are less than 2.0E-8, while the average differences are less than 7.0E-14. As an example, the elevation is given in meters, which means that the largest difference in elevation is 3.7E-9 meters, or about 3.7 nanometers.

A limitation of the measurements presented in this and the next section is that they are only the result of 29 iterations of the simulation. This corresponds to a simulated time of approximately 3 hours 40 minutes, compared to the normal simulated time of 48 hours.

4.7.2 Accuracy of the Parallel ELCIRC model

Comparing the output of the parallel model using 4 partitions with the output from the reference sequential model (see Figure 4.6), we see that the maximum and average difference is several orders of magnitude larger than with any of the compiler experiments above. For example, the maximum difference in elevation for an element is 4.3cm, and the average difference over all iterations and elements is about 0.8cm.

A simple visualization was created to study where differences occur, how large they are, and how effects from these differences spread throughout the model over time. Figure 4.7 shows a visualization, where each pixel in a 400 by 100 image represents an element in the model. The color of each pixel ranges from black (no difference between the elements in the reference sequential model and the parallel model), to white (the absolute difference between the element in the sequential and parallel model is larger than or equal to 4 cm).

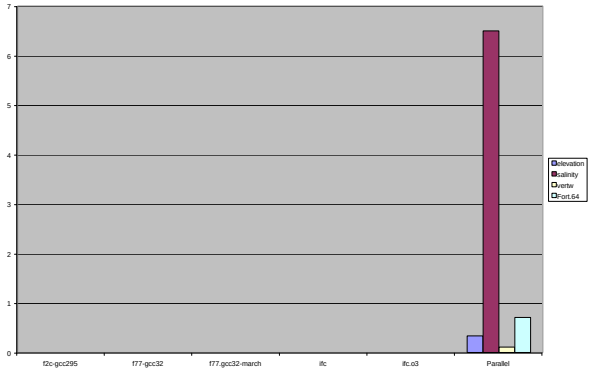
The figure consists of 13 panels, where each panel is an image representing the difference between the sequential and parallel ELCIRC implementations in one dumped iteration The uppermost panel is the first dumped iteration. Since the model only dumps every second iteration, the panels below represent every second iteration of the simulation.

The figure shows that the differences are introduced in a few localized regions, and that these differences propagate throughout the system over time. By inspecting which elements are included in the white bands, we should be able to find out what sets these elements apart from other elements. This is work in progress.

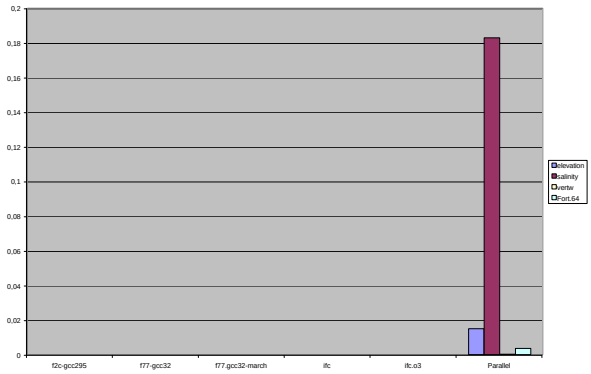
Two of the main factors that are expected to potentially introduce errors in the system are:

- In the sequential model, all elements, sides and nodes at the edges of the grid are governed by boundary conditions. The tool that generates the partition data files does not create any new boundary conditions along the grid edges that occur where we cut the original grid to create the partitions.

  ELCIRC may not properly handle grids with edges that are not controlled by boundary conditions. This needs to be investigated further.



(a) Maximum difference



(b) Average difference

Figure 4.6: Maximum and average difference between elevation, salinity and horizontal/vertical speeds in output from sequential model using different compilers and compiler options, and parallel model

- One of the differences between the sequential model and the parallel model shown above is that the solver was run only internally in each partition. Section 4.7.3 describes experiments with running the solver for the entire grid in the parallel model.

Another potential cause of errors lies in the comparison tool itself. The tool does not yet distinguish between wet and dry elements for all parameters. The first indication that this might be a problem came when we observed a maximum difference in salinity of about 114. The reason for this was that the salinity output file tags dry elements

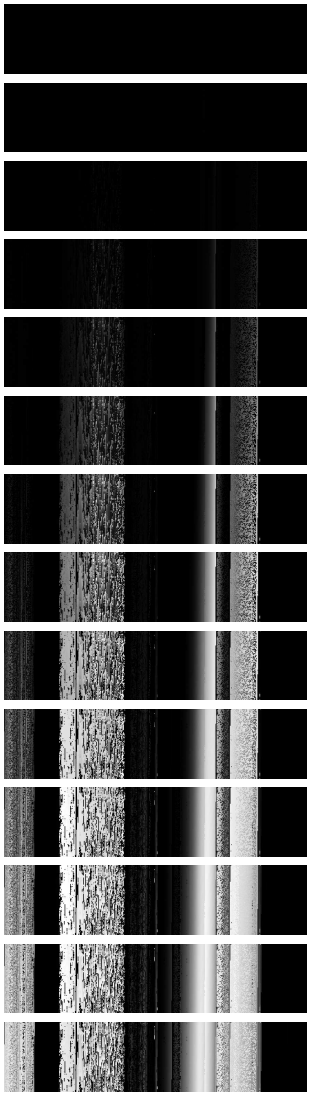


Figure 4.7: Comparison of elevation output files for parallel (with 4 partitions) vs. sequential model. The absolute difference in elevation for each element is translated to pixel intensities ranging from black (no difference) to white (a difference larger than or equal to 4 cm). The maximum difference elevation over 29 iterations was 4.3 cm, the average difference for all elements over all iterations was 0.8 cm.

with a salinity of -99. When comparing a dry element with a wet element (which has a positive salinity), a large difference occurs. The differences shown above are the results of omitting dry elements when comparing the salinity of elements.

4.7.3 Global or subdomain based solver

During the momentum-wave equation phase, a large matrix equation of the form $Ax = b$ is set up, and the equation is solved for x using a Jacobi[6] solver.

Each row in x represents the elevation of the water in an element in the model. The entries for the same row in A and b are built up using information from that element and the immediate neighbour elements.

Most of the experiments have been run with a separate solver for each partition (including its ghost regions). This may result in some information not being transmitted properly between the partitions as pressure waves may be traveling faster than the speed of the water flowing between the partitions[7].

To examine the sensitivity of the application to the region the solver is working on, a "global solver" has been added. The global solver relies on the property that rows in the equation can be set up in parallel (see section 4.3). Rows are forwarded from each partition to a centralized process, which combines the rows, solves the equation for the full grid, and forwards the result vector (x) back to each of the processes.

Initial experiments with the global solver shows that the accuracy of the parallel version using the global solver was roughly an order of magnitude worse than the one without the global solver.

A problem which complicates comparison between the parallel and sequential model is that the Jacobi solver does not converge within the maximum specified (2000) iterations in either of the sequential or parallel models. An interesting question is whether running the solver for a smaller problem (a partition) produces a more accurate solution to the equation than running the solver for the entire system.

4.8 Summary

ELCIRC has been parallelized using domain decompositioning. Initial measurements have shown a speedup of a factor 14.5 using 22 processors for a configuration where the workload was badly balanced. Additional factors have been identified which must be considered to provide better balancing of the workload.

Current work on ELCIRC is focused on improving the accuracy of the parallel model. Experiments have been conducted to compare the results of the parallel model with the sequential model and locating regions where errors are introduced in the model.

A "global solver" has also been introduced, and used to experiment with the size of the geometric regions that a matrix equation in ELCIRC needs to be solved for. Initial experiments have not shown an improvement by solving for a larger region.

[6] The type of solver used is configurable, but for the current data set, the Jacobi solver is used

[7] This concern was raised at a meeting at OGI in May 2001

Chapter 5

ELCIRC and PATHS

This chapter describes experiences with using PATHS with ELCIRC.

The parallel and sequential version of ELCIRC use the same source code. To create a parallelized ELCIRC executable, annotations in the source code are expanded (by a preprocessor) to call functions that read updates from neighbour partitions and apply them to the ghost regions. Updates are sent to neighbour partitions by extracting, for each neighbour, the values in the arrays that are used by the neighbour and storing these values in a PastSet Element.

The paths created to send and receive updates are instrumented using trace wrappers to inspect some of the performance aspects of the application.

5.1 Locating a performance bottleneck in the parallel ELCIRC

Figure 5.1 shows a visualization of timestamp data from the processes running on one of the nodes in the 8-way cluster. The visualization resembles a Gantt chart, with one time line per trace wrapper. The tick lines represent the execution time for move or observe operations.

Since some of the operations take a relatively short time and are difficult to see in the graph, a small circle is drawn above the start timestamp for each operation. Color coding (not visible in black and white prints) are used to separate read and write operations.

The traces show that sending ghost region updates to the neighbour partitions does not delay the processes much, as the thick lines of the write operations are generally not visible in the graph.

Reading updates from the neighbours, though, severely delay some of the processes after a few iterations. The reason for this delay is that the read operation is waiting for a neighbour that either has a higher workload or is delayed by one of its other neighbours.

Knowing which process each of the processes are delayed by in an execution, we can organize the processes in a dependency tree. A process that is not delayed by other processes will be the root of the dependency tree. This root process dominates the

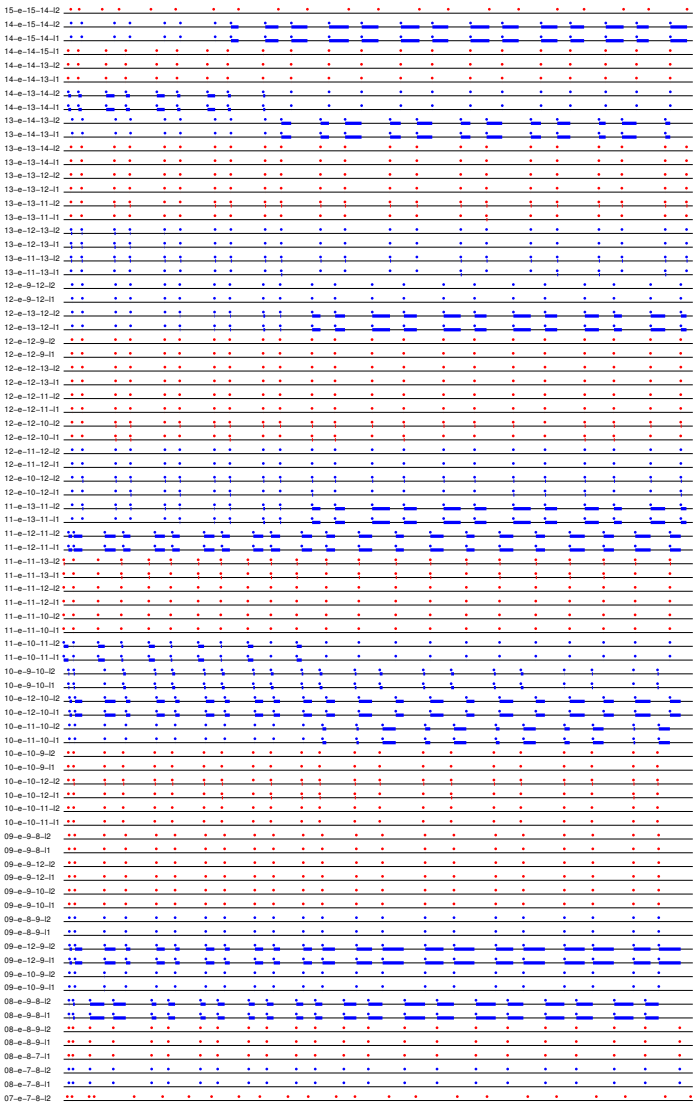
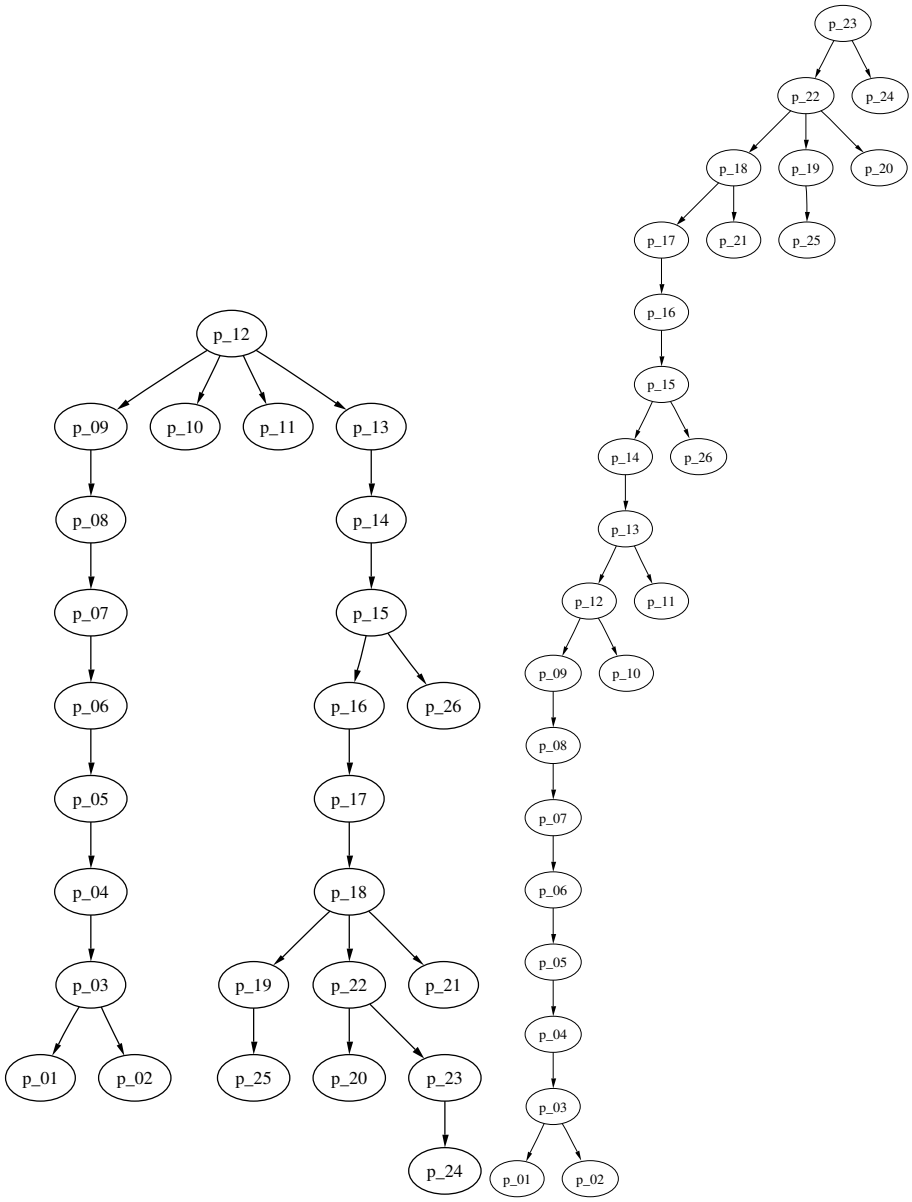


Figure 5.1: Trace data for processes running on one of the 8-way nodes. The text to the left identifies the trace wrapper. The horizontal line is the time line from the start of the first operation on the host. The thick lines represents read or write operations. The figure only shows the first 21-23 iterations of the application.

execution time of the other processes in the tree. Figure 5.2 shows two dependency trees described below.

Figure 5.2-(a) shows the dependency trees from an experiment on the P4 cluster

where we ran the parallel ELCIRC application using NFS for storing data files. One of the nodes in the cluster was set up as an NFS server for the other nodes. The delay graph shows that the execution time of the parallel application was dominated by a single process in the system. This process (process 12) was also the partition which had the largest number of elements.



(a) Original analysis with NFS traffic

(b) Analysis without NFS traffic

Figure 5.2: Delayed-by trees. The circles represent processes in the system. The arrows point from a process to the processes that depend on this process.

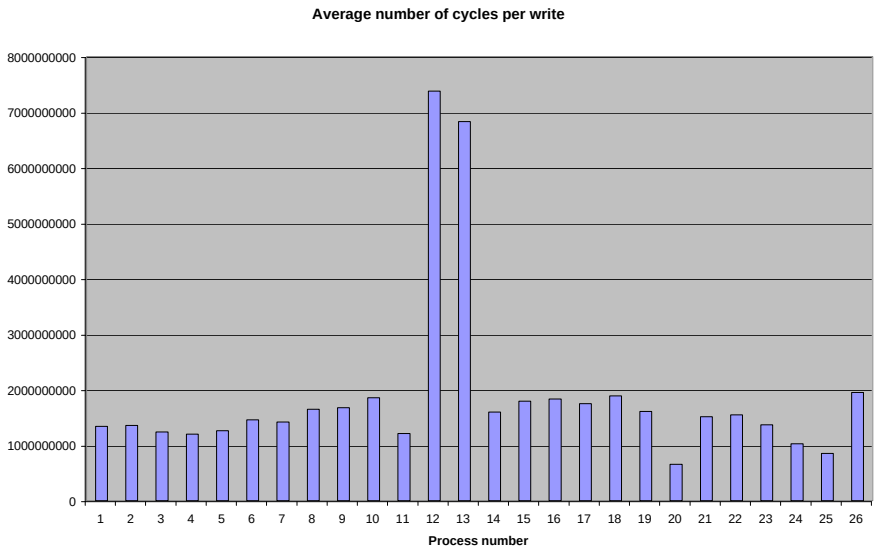


Figure 5.3: Average time for writing output data in an iteration for each process. Process 12 and 13 are located on the node acting as NFS server for the other nodes.

5.1.1 Identifying the cause of the bottleneck

Using one of the cluster nodes as an NFS server creates an unbalanced configuration of the nodes. Figure 5.3 shows that the two processes residing on the server node (process 12 and 13) have a much higher overhead in writing to local disk than the other processes that write to NFS.

Observing that the process that was the applications bottleneck (process 12) was located on the NFS server, another experiment was set up to examine whether it was the configuration of the system rather than the size of the partitions that was the cause of the bottleneck.

The data files were mirrored out to local disks in each node and the experiment was rerun using only local disks on each node. The times reported from the P4 cluster in the previous chapter (table 4.2) are the results from this second execution.

The bottleneck analysis for the second run shows that the process which dominates the execution time is now process 23 (see figure 5.2-(b)), which is only the 16th largest partition (at 1789 elements). This suggests that the number of elements is not the only factor that dominates the computation time of a partition.

This experiment emphasizes the importance of not only looking at the single factor that one expects to dominate the performance. Other factors that may or may not coincide with the expected dominating factor may be more important.

5.1.2 Additional factors contributing to a partitions execution time

The following factors were found by examining the source code, and should be examined further for their influence on the computation time:

- The number of *wet* elements[1] in the partitions grid. Most of the computation in the model is done only on wet elements in the grid. Dry elements (such as land elements) are only involved in some of the computations.
- The depth of the water in the wet elements. Elements with shallower water have fewer wet layers, which results in less computation for the element.
- The density of the grid combined with the water speed in the region. One of the largest contributions to run time in the simulation loop is the backtracking stage. With a dense grid, high water speed and long time step, each water particle may be tracked through several elements before finding the originating point. Tracking particles through multiple elements increase the execution time of the backtracking algorithm.

5.1.3 Communication overhead

Using the data from the trace-wrappers, the communication overhead of the dominating process (process 23) was examined. The time spent communicating was less than 1% of the execution time of each iteration.

5.2 Partitioning bug

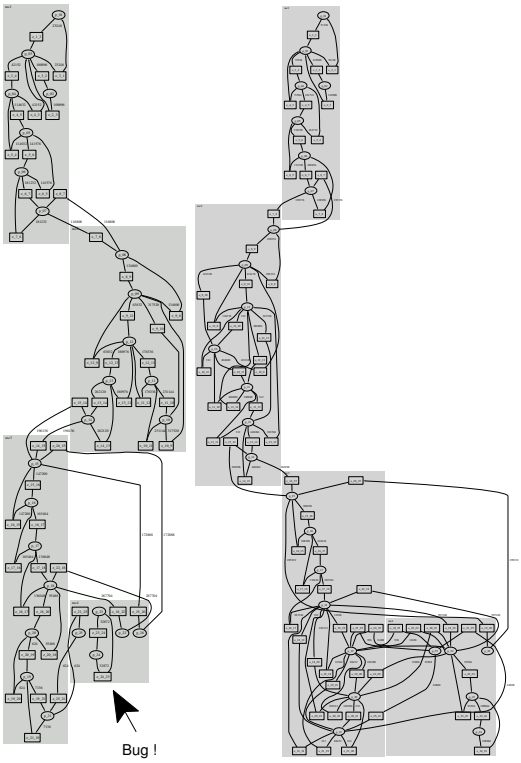
The path maps can also be used to find some configuration-related bugs.

Figure 5.4-(a) shows a visualization of the communication topology of ELCIRC that was created from the path specifications. This configuration has a bug where two of the processes are only connected to each other, and not to any other processes in the system (indicated by an arrow). The problem manifested itself as the two processes finishing much earlier than the rest of the processes in the system.

The problem occurred due to a problem with the manual partitioning tool which didn't correctly handle the ghost regions in the coarsest areas of the grid, such as in the ocean, resulting in ghost regions with no elements. The problem was fixed by changing the rules for including elements in the ghost regions.

The new configuration after running the corrected partitioning tool is shown in figure 5.4-(b).

[1] A wet element is an element that contains water.



(a) Original configuration with a bug

(b) Modified configuration

Figure 5.4: Visualization of processes, paths and elements in the parallelized EL-CIRC system. Processes (circles) are mapped onto 4 8-way computers (shaded boxes). Tuples are read and written to elements (rectangles) using paths (lines).

5.3 Controlling ghost region updates with PATHS

One of the problems with the parallel ELCIRC version described earlier was the amount and complexity of the code added to ELCIRC to update the ghost regions. Additionally, when new strategies for updating ghost regions were tried out, both the support tools generating partition information and the support code in ELCIRC had to be changed.

Instead of implementing the necessary code for sending and receiving updates in the ELCIRC application code, the responsibility for implementing what is essentially a *send_update* and *receive_update* function is moved down into the PATHS system. This corresponds to one of the goals of the PATHS system: using PATHS to control how communication is done between processes, and to implement the functionality of the communication subsystem.

This allows us to experiment with different update strategies and keep a record, using the path specifications, of the strategies and configurations used when multiple experiments are run.

5.3.1 Introducing new PATHS wrappers

Sending and receiving of updates can be implemented using two wrappers:

- The *subsample* wrapper, which is used to extract a subset of the entries in the array when the array is written to a path, and to apply updates to an array when updates are read from a path.

  Which entries to extract or apply are specified as parameters to the wrapper[2].

- The *fork* wrapper, which has a list of paths that it applies an operation to. The path list is provided as one of the parameters to the wrapper.

  Once an operation is called on the fork wrapper, it applies the same operation and parameters to each of the paths in the path list.

An example is shown in Figure 5.5. Process i sends an update to its neighbours by sending the eta2 array[3] down the path. The fork wrapper then sends the eta2 array further down to each of its sub-paths. The subsample wrappers extract only the values needed by the corresponding neighbour, and stores the extracted values in a location shared with the neighbour.

To receive updates, process $i-1$ calls the *receive_update* function on the path using its copy of eta2 as one of the arguments. The fork wrapper then calls *receive_update* on each of its sub-paths. The subsample wrappers retrieve the updates from the shared locations, extracts the values and applies them to the eta2 array[4].

[2] To make path descriptions more readable for humans, the index list is stored in a file and the wrapper is handed the file name.

[3] eta2 is one of the arrays in ELCIRC

[4] The current subsample wrappers can also do some transformations on the values extracted and applied, as some vectors need to be transformed before they can be copied between partitions

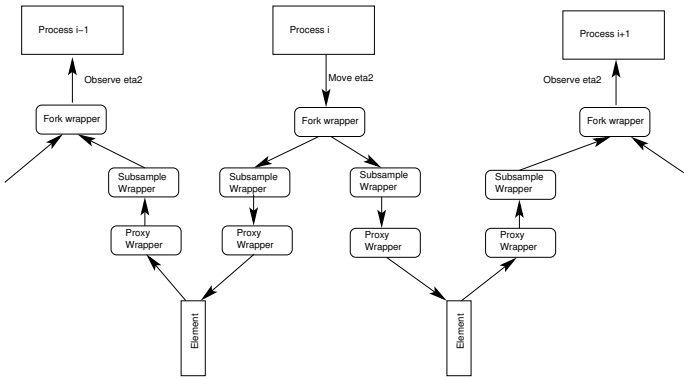


Figure 5.5: ELCIRC using fork and subsample wrappers - sharing *eta2*

5.3.2 Experiences using new wrappers

An ELCIRC implementation using the new wrappers was used in some of the experiments described in 4.7.2. Some of the initial observations are:

- We can experiment with different ways of updating ghost region simply by changing path descriptions or introducing new wrapper types along the paths.
- The support tools used to generate partitions and path configurations are simpler and the source code is shorter.
- This version needs less source code than any of the other versions. The support code which is added to the ELCIRC application is about 120 lines of C code including the initialization code.
- Which variables are updated are now selected by modifying one or a few isolated places in the support tools rather than in the ELCIRC source code *and* the support tools.

5.4 Summary

PATHS has been used to implement the communication between the processes in the parallelized ELCIRC application.

PATHS was used to measure some of the performance aspects of the application such as locating the process dominating the execution time of the simulation, and to find a bug with the configuration which caused two of the processes to be isolated from the rest.

PATHS was also used to simplify updating of ghost region data, as the functions sending, receiving and applying updates to ghost regions were moved down to the PATHS system. This allows us to experiment with different strategies for updating the

ghost regions without modifying or recompiling the ELCIRC source code, and to use the PATHS system for logging and tracing updates applied to ghost regions.

Chapter 6

Additional related work

This chapter presents additional related work, and expands on some of the related work presented in the papers.

6.1 Configuration and adapting to cluster architectures

MPI

The Message Passing Interface (MPI) [39] standard hides the architecture of the system a distributed application runs on from the application programmer. It is up to the implementation to provide efficient point-to-point and collective operations, allowing the user to focus on these abstractions instead of concentrating on the under-laying architecture. As an example, the MPICH [23] implementation allows clusters of SMP nodes to use shared memory to optimize message passing internally on the nodes.

Efficient implementation of collective operations has been studied for various architectures [27, 7, 32, 31, 26, 46, 49, 54]. A fixed strategy or operation tree is not necessarily optimal for all architectures and topologies, however [54]. We have also observed this in [8], where we found that in some situations, going against common sense and doubling the amount of messages over the network improved the performance of the Allreduce operation.

The MPI standard [39] includes Process Topologies, which is a mechanism for re-mapping the ranks of processes according to a logical arrangement of communication specified as a graph. The standard states that this may be used by the runtime system to aid the mapping of processes onto hardware, and that it may be used as an advice for implementing optimized communication.

We did not see any provision in the LAM-MPI [14, 35] implementation to optimize the collective operations based on information in the Process Topologies[1]. According to Träff [53], current MPI implementations seems to make little use of the Process

[1] In fact, since the reduce and broadcast operations used different trees for reduction and broadcast, it is hard to see how any re-mapping of process ranks would succeed in optimizing one of the operations without penalizing the other.

Topology mechanism: "current MPI implementations rarely go beyond the most trivial implementations, and simply performs no process re-mapping".

Vadhiyar et. al [54] searches (using modified hill-descent heuristics) for an optimal mapping of collective operation algorithms from a set of predefined algorithms. In contrast, we do not presuppose any particular algorithms, but allow free experimentation with different tree structures in both the PATHS system and the MPI extensions. They also experiment with tuning buffer sizes in the MPI implementation. We have not done this.

Infopipes

Infopipes [10, 34] is an abstraction for information flow with particular focus on distributed streaming applications.

The abstraction models an application as a pipeline and focuses on how information flows down the pipeline, and on the computations and manipulations done on the information on the way. It provides components such as pipes, filters, buffers, and pumps, which are combined to build a pipeline. The main flow of information is in one direction, though they provide a control feedback mechanism which can be used to control components earlier in the pipeline.

An example is a filter which receives information from a sensor further down in the pipeline. The filter may use this information to control what is filtered (for instance, to control the amount of data sent down the pipeline).

The infopipe abstraction also supports splitting and merging of information flows (through components called tees), as we do with path merging and fork wrappers.

Our focus, however, is on the flow of an *operation* through the middleware (and, hence, also the flow of the return value from the operation), and on optimization of computation along the path.

Reflective middleware

One of the key design points of PATHS was to allow inspection of a running systems configuration through the path specifications. The system also allows run-time configuration by creating new path specifications and building new paths.

One of the goals behind Reflective Middleware (an overview of some current efforts is available in [33]) is to allow the middleware to adapt to changing conditions. This ability is introduced by adding reflection, the ability to inspect and adapt the behaviour of objects [30]. [17] defines reflective middleware as:

> "reflective middleware is simply a middleware system that provides inspection and adaptation of its behaviour through an appropriate CCSR."

The current PATHS implementations do not support inspection of dynamic properties of wrappers, or changing of parameters to wrappers after the wrappers are initialized. Changing the configuration of a system is instead done by creating a new path.

As such, we support some of the structural aspects [2] of reflective middleware. Dynamic reconfiguration of the paths has been considered, but is left as future work.

Active networking and Overlay Networks

Active networking [51] introduces user-controlled customized computing in the switches of a network. [50] describes two approaches to realization of active networks: In the *programmable switch* a mechanism is provided to download programs to the switches which may manipulate ordinary network packets. In the *capsule* approach, a program is sent along with each message. The program is executed in each switch, and may modify the message.

Switches that can execute code may be interesting hosts for running PATHS wrappers if wrappers can be instantiated and bound in the switch. Packets must be tagged to identify which path they belong to.

Our current approach of using user-level servers on intermediate nodes in the clusters is probably closer in that respect to Overlay Networks [45, 21, 52], which make use of user-level routers to provide functionality such as Resilient Overlay Networks (detecting and recovering from routing outages and periods of degraded performance) [3], reliable multicasting of time-critical data [47], and emulation of dedicated networks. The PATHS system also makes use of intermediate servers, but current experiments have mostly used this to set up paths between hosts which do not have direct network connectivity (an example is the 4-way cluster which is hidden behind a firewall – nodes in the cluster can only be reached from nodes outside of the firewall by setting up a path which goes through a PastSet server running on the firewall).

Scout

The Scout operating system [38] uses an abstraction called *routers*, where each router implements functionality such as the IP protocol, an MPEG decompression algorithm, or a driver for a particular SCSI adapter. Routers may use lower-level routers to implement their functionality. As an example, the IP router may need an ETH and an ARP router[2] to send or receive IP traffic over an Ethernet device. Routers are connected in a graph, which is initialized at boot time.

To optimize communication in the operating system, a *path* can be created where each stage of the path corresponds to a router. Attributes specified when the path is created can be used to associate optimized service implementations for each of the involved routers with the path. As an example, given the right set of attributes (which specify invariants of the traffic over the path), a path through multiple routers may share buffers, avoiding copying of data.

As in the PATHS system, application specific information may be used to optimize the communication path. Computations cannot be introduced along the paths however (other than selecting paths through the existing routing tree). Furthermore, the Scout paths do not extend beyond the operating system, which means that the Scout system can only be used for internal optimization in the node.

Packets arriving at a network device cannot be associated with a Scout path immediately. To resolve this, the scout system adds a demux operation to the routers. The demux operation maps packets into a path which can be used to process that data. Packets may need to traverse through multiple routers before the classification of the

[2] The ARP router uses the ETH router again

packet is refined enough to find a unique path. In contrast, operations traveling along a PATHS path contain information to uniquely identify the next stage in the path.

6.2 Monitoring and profiling

The PATHS system has no built-in profiling or monitoring system. Instead, profiling and monitoring is implemented using PATHS wrappers, allowing different types of instrumentation to be selected or added by a user.

Tracing and monitoring tools for distributed systems can generally be classified by the following categories:

- Extending profilers that instrument sequential programs to work with parallel and distributed applications.
- Monitoring communication events such as sending and receiving messages by instrumenting the communication API or middleware API.
- Passive monitoring of messages in the network or in the network interfaces .

An example of the first form of profiling tools is Quartz [4]. The PATHS system is only able to add instrumentation in the PATHS paths, and thus cannot capture events that occur in code that does not make use of a path.

The second class of tracing and monitoring includes tracing of MPI messages and operation calls [55]. Instrumentation in MPI can be added using the MPI profiling interface [39]. Monitoring for PVM (Parallel Virtual Machine) [48] is also provided through the XPVM graphical console and monitor.

An operation traveling down (invoke) or up (return) a path can be viewed as a sequence of communication events where an invocation or a return value is a message. This view resembles the use of the term *sending messages* to refer to the activity of calling methods on objects [44].

The PATHS trace wrappers are inserted along the path to time stamp these communication events. As such, the PATHS trace wrappers resemble the communication event monitors.

PATHS has an advantage that communication event systems generally don't have: the path specification makes some of the communication paths explicitly specified. The operation sent through the path can also be followed through intermediate hops along the path.

An example of the third class monitoring listed above, monitoring messages in a network or in the network interfaces, can be found in [37, 36]. Current implementations of the PATHS system do not support this type of monitoring.

Chapter 7

Conclusions

In parallel and distributed computing, clusters are increasingly used for compute- and I/O-intensive applications. As we add computing resources to a parallel application, one of the fundamental questions is how well the application scales, both with regards to speedup and to increasing the problem size.

This dissertation is based on 10 papers reporting on two main issues influencing scaling. The first is the end-to-end communication latency. The other is the configuration and mapping of the application onto a cluster topology and architecture.

The project has taken an experimental approach to studying scalability issues. Systems have been built, instrumented and experimented with, and data based on executions of the systems have been analyzed.

A detailed synthesis of the papers (chapter 2) and the main contributions (section 1.4) has been presented.

In a "black box" design, it is hard to understand the performance impacts of an application's configuration and mapping to a cluster, as information about what is happening inside the box is not available.

Small and simple changes to a configuration influence scaling and latency. It is hard to find good configurations analytically or by computation. Instead, it is demonstrated that starting with what is believed to be a good configuration, a number of experiments can be run to find configurations with better performance.

Thus, the ability to tune the configuration, and use knowledge about the application and the cluster topology, is found to be important.

One of the key tools for finding good configurations is a specification focusing on what is done where in the clusters, and where data is stored.

Chapter 8

Future work

The complexity of creating PATHS specifications for large applications is currently managed by using path-generation functions based on rules, or by using coarse-grained path maps that are expanded to full path specifications using various "path expansion" functions. An area of future study is how to make generalized higher level abstractions for defining path maps which can then be translated into path specifications.

The current approach to adapting to changing conditions, is to replace an existing path with a new path. Dynamic reconfiguration of existing paths has not been examined, and may be useful in some application settings.

The PATHS system has been used with PastSet, and a PATHS-inspired system was used to configure LAM-MPI. The LAM-MPI project is currently in progress, and further experiments with configuration mechanism for MPI will be examined. We also plan to look at using PATHS-inspired configuration systems for other communication APIs and middleware.

References

[1] AMDAHL, G. Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities. In *Proceedings of the AFIPS Spring Joint Computer Conference, Atlantic City, New Jersey, USA* (1967), AFIPS Press, Reston, Virginia, USA, pp. 483–485.

[2] ANDERSEN, A. *OOPP, A Reflective Middleware Platform including Quality of Service Management*. Dr. sci. thesis, Department of Computer Science, University of Tromsø, Tromsø, Norway, Feb. 2002.

[3] ANDERSEN, D., BALAKRISHNAN, H., KAASHOEK, F., AND MORRIS, R. Resilient overlay networks. In *Proceedings of the eighteenth ACM symposium on Operating systems principles* (2001), ACM Press, pp. 131–145.

[4] ANDERSON, T. E., AND LAZOWSKA, E. D. Quartz: a tool for tuning parallel program performance. In *Proceedings of the 1990 ACM SIGMETRICS conference on Measurement and modeling of computer systems* (1990), ACM Press, pp. 115–125.

[5] ANSHUS, O. J., AND LARSEN, T. MacroScope: The Abstractions of a Distributed Operating System. *Norsk Informatikk Konferanse* (Oct. 1992).

[6] ARPACI-DUSSEAU, A. C., ARPACI-DUSSEAU, R. H., CULLER, D. E., HELLERSTEIN, J. M., AND PATTERSON, D. A. Searching for the Sorting Record: Experiences in Tuning NOW-Sort. In *Proceedings of the SIGMETRICS symposium on Parallel and distributed tools (SPDT 98), USA* (1998), pp. 124–133.

[7] BERNASCHI, M., AND RICHELLI, G. MPI Collective Communication Operations on large Shared memory Systems. *Proceedings of the Ninth Euromicro Workshop on Parallel and Distributed Processing (EUROPDP.01)* (2001).

[8] BJØRNDALEN, J. M., ANSHUS, O., LARSEN, T., BONGO, L. A., AND VINTER, B. Scalable Processing and Communication Performance in a Multi-Media Related Context. *Euromicro 2002, Dortmund, Germany* (September 2002).

[9] 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 Application. In *Norsk Informatikk Konferanse* (Nov. 2001), pp. 164–175.

[10] BLACK, A. P., HUANG, J., KOSTER, R., WALPOLE, J., AND PU, C. Infopipes: An abstraction for multimedia streaming. *Multimedia Systems, special issue on multimedia middleware* (2002). Volume 8 Issue 5 pp 406-419, Springer Verlag.

[11] BLUMRICH, M., LI, K., ALPERT, R., DUBNICKI, C., FELTEN, E., AND SANDBERG, J. A virtual memory mapped network interface for the SHRIMP multicomputer. In *Proceedings of the 21st Annual Symposium on Computer Architecture* (April 1994), pp. 142–153.

[12] BONGO, L. A. EventScope: Configurable On-line Monitoring of Parallel and Distributed Applications. Master's thesis, Department of Computer Science, University of Tromsø, Dec. 2002.

[13] BUONADONNA, P. Implementation and Analysis of the Virtual Interface Architecture. *Supercomputing '98, Orlando, FL* (Nov. 1998).

[14] BURNS, G., DAOUD, R., , AND VAIGL, J. LAM: An Open Cluster Environment for MPI. www.lam-mpi.org, 1994.

[15] CARRIERO, N., AND GELERNTER, D. Linda in Context. *Commun. ACM 32*, 4 (Apr. 1989), pp. 444–458.

[16] CORIE project homepage. http://www.ccalmr.ogi.edu/CORIE/.

[17] COULSON, G. What is reflective middleware?, Dec. 2001. See computer.org/dsonline/middleware/RMarticle1.htm.

[18] CULLER, D. E., AND SINGH, J. P. *Parallel Computer Architecture - A hardware / software approach*. Morgan Kaufmann, 1999.

[19] DAMIANAKIS, S. N., CHEN, Y., AND FELTEN, E. Reducing Waiting Costs in User-Level Communication. In *11th International Parallel Processing Symposium (IPPS '97)* (April 1997).

[20] ELCIRC homepage. http://www.ccalmr.ogi.edu/CORIE/modeling/elcirc.html.

[21] ERIKSSON, H. MBONE: The Multicast Backbone. *Communications of the ACM, Vol.37, No. 8* (Aug. 1994).

[22] Giganet cLAN. http://www.giganet.com/products/.

[23] GROPP, W., AND LUSK, E. Installation Guide to MPICH, a Portable Implementation of MPI, Version 1.2.4.

[24] GUSTAFSON, J. L. Reevaluating Amdahl's law. *Communications of the ACM 31*, 5 (1988), pp. 532–533.

[25] HALVORSEN, P. *Improving I/O Performance of Multimedia Servers*. PhD thesis, University of Oslo, 2001.

[26] HUSBANDS, P., AND HOE, J. C. MPI-StarT: delivering network performance to numerical applications. *Proceedings of the 1998 ACM/IEEE conference on Supercomputing* (1998). San Jose, CA.

[27] JACUNSKI, M., SADAYAPPAN, P., AND PANDA, D. All-to-All Broadcast on Switch-based Clusters of Workstations. *13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing* (12 - 16 April 1999). San Juan, Puerto Rico.

[28] JOUBERT, P., KING, R., NEVES, R., RUSSINOVICH, M., AND TRACEY, J. High-Performance Memory-Based Web Servers: Kernel and User-Space Performance. *Proceedings of the 2001 USENIX Annual Technical Conference* (2001), pp. 175–188.

[29] KARYPIS, G., AND KUMAR, V. *MeTis: Unstructured Graph Partitioning and Sparse Matrix Ordering System, Version 2.0*, 1995.

[30] KICZALES, G., DES RIVIERES, J., AND BOBROW, D. G. *The Art of the Metaobject Protocol*. MIT Press, 1991.

[31] 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. *Proceedings of the seventh ACM SIGPLAN symposium on Principles and practice of parallel programming* (1999). Atlanta, Georgia, United States.

[32] KIELMANN, T., HOFMAN, R. F. H., BAL, H. E., PLAAT, A., AND BHOEDJANG, R. A. F. MPI's Reduction Operations in Clustered Wide Area Systems. *Proceedings of the seventh ACM SIGPLAN symposium on Principles and Practice of parallel programming* (1999). Atlanta, Georgia, United States.

[33] KON, F., COSTA, F., BLAIR, G., AND CAMPBELL, R. H. The Case for Reflective Middleware. *Communications of the ACM, Vol. 45, No. 6* (June 2002).

[34] KOSTER, R., BLACK, A. P., HUANG, J., WALPOLE, J., , AND PU, C. Thread Transparency in Information Flow Middleware. In *Middleware 2001 – IFIP/ACM International Conference on Distributed Systems Platforms, Heidelberg, Germany* (November 2001), pp. 121–140.

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

[36] LIAO, C., MARTONOSI, M., AND CLARK, D. W. Performance monitoring in a Myrinet-connected SHRIMP cluster. In *Proceedings of the SIGMETRICS symposium on Parallel and distributed tools* (1998), ACM Press, pp. 21–29.

[37] MARTONOSI, M., CLARK, D. W., AND MESARINA, M. The SHRIMP performance monitor: design and applications. In *Proceedings of the SIGMETRICS symposium on Parallel and distributed tools* (1996), ACM Press, pp. 61–69.

[38] MOSBERGER, D., AND PETERSON, L. L. Making Paths Explicit in the Scout Operating System. In *Operating Systems Design and Implementation* (1996), pp. 153–167.

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

[40] M-VIA Home Page, NERSC center at Lawrence Berkeley National Laboratory, http://www.nersc.gov/research/ftg/via/.

[41] PAI, V. S., DRUSCHEL, P., AND ZWAENEPOEL, W. IO-Lite: A Unified I/O Buffering and Caching System. *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation* (1999).

[42] REED, D. A., MADHYASTHA, T. M., AYDT, R. A., ELFORD, C. L., SCULLIN, W. H., AND SMIRNI, E. I/O, Performance Analysis, and Performance Data Immersion. In *MASCOTS* (1996), pp. 5–15.

[43] REED, D. A., SHIELDS, K. A., SCULLIN, W. H., TAWERA, L. F., AND ELFORD, C. L. Virtual Reality and Parallel Systems Performance Analysis. *IEEE Computer 28*, 11 (1995), 57–67.

[44] SEBESTA, R. W. *Concepts of Programming Languages*. Addison Wesley, 2002. ISBN 0-201-75295-6.

[45] SIMPSON, W. IP in IP Tunneling, October 1995. Internet RFC 1853, Daydreamer.

[46] SISTARE, S., VANDEVAART, R., AND LOH, E. Optimization of MPI collectives on clusters of large-scale SMP's. *Proceedings of the 1999 conference on Supercomputing* (1999). Portland, Oregon, United States.

[47] SNOEREN, A. C., CONLEY, K., AND GIFFORD, D. K. Mesh-based content routing using XML. In *Proceedings of the eighteenth ACM symposium on Operating systems principles* (2001), ACM Press, pp. 160–173.

[48] SUNDERAM, V. PVM: A Framework for Parallel Distributed Computing. In *Concurrency: Practice and Experience, Vol. 2, No. 4* (Dec. 1990).

[49] TANG, H., AND YANG, T. Optimizing threaded MPI execution on SMP clusters. *Proceedings of the 15th international conference on Supercomputing* (2001). Sorrento, Italy.

[50] TENNENHOUSE, D. L., SMITH, J. M., SINCOSKIE, W. D., WETHERALL, D. J., AND MINDEN, G. J. A Survey of Active Network Research. *IEEE Communications Magazine 35*, 1 (1997), pp. 80–86.

[51] TENNENHOUSE, D. L., AND WETHERALL, D. J. Towards an Active Network Architecture. *Computer Communication Review 26*, 2 (1996).

[52] TOUCH, J., AND HOTZ, S. The X-Bone. *Proc. Third Global Internet Mini-Conference at Globecom '98 Sydney, Australia Nov. 8-12* (1998), pp 75–83 (listed as pp. 44–52 of the miniconference).

[53] TRAFF, J. L. Implementing the MPI Process Topology Mechanism. *Supercomputing 2002*.

[54] VADHIYAR, S. S., FAGG, G. E., AND DONGARRA, J. Automatically Tuned Collective Communications. *SuperComputing* (2000), pp. 46. Dallas, Texas.

[55] VETTER, J. Performance analysis of distributed applications using automatic classification of communication inefficiencies. In *International Conference on Supercomputing* (2000), pp. 245–254.

[56] Virtual Interface (VI) Architecture - Implementation Guide. Draft revision 0.6., February 1998. Intel Corporation.

[57] VINTER, B. *PastSet a Structured Distributed Shared Memory System*. PhD thesis, Tromsø University, 1999.

[58] VON EICKEN, T., BASU, A., BUCH, V., AND VOGELS, W. U-Net: A User-Level Network Interface for Parallel and Distributed Computing. *The 15th ACM Symposium on Operating Systems Principles (SOSP)* (December 1995).

[59] XML-RPC Home Page:. http://www.xml-rpc.com/.

Bibliography

[60] ALNAES, K., KRISTIANSEN, E., GUSTAVSON, D., AND JAMES, D. Scalable Coherent Interface. In *IEEE CompEuro 90* (1990).

[61] AMDAHL, G. Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities. In *Proceedings of the AFIPS Spring Joint Computer Conference, Atlantic City, New Jersey, USA* (1967), AFIPS Press, Reston, Virginia, USA, pp. 483–485.

[62] ANDERSEN, A. *OOPP, A Reflective Middleware Platform including Quality of Service Management*. Dr. sci. thesis, Department of Computer Science, University of Tromsø, Tromsø, Norway, Feb. 2002.

[63] ANDERSEN, D., BALAKRISHNAN, H., KAASHOEK, F., AND MORRIS, R. Resilient overlay networks. In *Proceedings of the eighteenth ACM symposium on Operating systems principles* (2001), ACM Press, pp. 131–145.

[64] ANDERSON, T. E., AND LAZOWSKA, E. D. Quartz: a tool for tuning parallel program performance. In *Proceedings of the 1990 ACM SIGMETRICS conference on Measurement and modeling of computer systems* (1990), ACM Press, pp. 115–125.

[65] ANSHUS, O. J., AND LARSEN, T. MacroScope: The Abstractions of a Distributed Operating System. *Norsk Informatikk Konferanse* (Oct. 1992).

[66] ARPACI-DUSSEAU, A. C., ARPACI-DUSSEAU, R. H., CULLER, D. E., HELLERSTEIN, J. M., AND PATTERSON, D. A. Searching for the Sorting Record: Experiences in Tuning NOW-Sort. In *Proceedings of the SIGMETRICS symposium on Parallel and distributed tools (SPDT 98), USA* (1998), pp. 124–133.

[67] BAL, H., KAASHOEK, M., AND TANENBAUM, A. Orca: A Language For Parallel ProgrammingOf Distributed Systems. In *IEEE Computer 25(8)*. Aug. 1992, pp. 10–19.

[68] BAL, H. E., AND TANENBAUM, A. S. Orca: A Language for Distributed Object-Based Programming. In *SIGPLAN Notices* (May 1990), vol. 25, pp. 17–24.

[69] BERNASCHI, M., AND RICHELLI, G. MPI Collective Communication Operations on large Shared memory Systems. *Proceedings of the Ninth Euromicro Workshop on Parallel and Distributed Processing (EUROPDP.01)* (2001).

[70] BHOEDJANG, R., RUHL, T., AND BAL, H. E. Efficient Multicast On Myrinet Using Link-Level Flow Control. In *International Conference on Parallel Processing* (Minneapolis, MN, Aug. 1998), pp. 381–390.

[71] BILAS, A., IFTODE, L., AND SINGH, J. P. Evaluation of Hardware Support for Automatic Update in Shared Virtual Memory Clusters. In *12th ACM International Conference on Supercomputing* (July 1998).

[72] BIRREL, A. D., AND NELSON, B. Implementing Remote Procedure Calls. *ACM Transactions on Computer Systems* (Feb. 1984), pp. 39–59.

[73] BIRRELL, A. D., AND NELSON, B. J. Implementing Remote Procedure Calls. In *Proceedings of the ninth ACM Symposium on Operating Systems Principles* (1983).

[74] BJØRNDALEN, J. M., ANSHUS, O., LARSEN, T., BONGO, L. A., AND VINTER, B. Scalable Processing and Communication Performance in a Multi-Media Related Context. *Euromicro 2002, Dortmund, Germany* (September 2002).

[75] BJØRNDALEN, J. M., ANSHUS, O., VINTER, B., AND LARSEN, T. Configurable Collective Communication in LAM-MPI. *Proceedings of Communicating Process Architectures 2002, Reading, UK* (September 2002).

[76] BJØRNDALEN, J. M., ANSHUS, O., VINTER, B., AND LARSEN, T. The latency of user-to-user, kernel-to-kernel and interrupt-to-interrupt level communication. *NIK 2002, Norsk Informatikk Konferanse, Kongsberg, Norway* (Nov. 2002).

[77] BJØRNDALEN, J. M., ANSHUS, O., VINTER, B., AND LARSEN, T. The Performance of Configurable Collective Communication for LAM-MPI in Clusters and Multi-Clusters. *NIK 2002, Norsk Informatikk Konferanse, Kongsberg, Norway* (November 2002).

[78] 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 Application. In *Norsk Informatikk Konferanse* (Nov. 2001), pp. 164–175.

[79] BJØRNDALEN, J. M., ANSHUS, O., VINTER, B., AND LARSEN, T. Comparing the Performance of the PastSet Distributed Shared Memory System using TCP/IP and M-VIA. In *Proceedings of WSDSM'00, Santa Fe, New Mexico* (May 2000).

[80] BJØRNDALEN, J. M., ANSHUS, O., VINTER, B., AND LARSEN, T. The Impact on Latency and Bandwidth for a Distributed Shared Memory System Using a Gigabit Network Supporting the Virtual Interface Architecture. In *Norsk Informatikk Konferanse* (Nov. 2000).

[81] BLACK, A. P., HUANG, J., KOSTER, R., WALPOLE, J., AND PU, C. Infopipes: An abstraction for multimedia streaming. *Multimedia Systems, special issue on multimedia middleware* (2002). Volume 8 Issue 5 pp 406-419, Springer Verlag.

[82] BLUMRICH, M., LI, K., ALPERT, R., DUBNICKI, C., FELTEN, E., AND SANDBERG, J. A virtual memory mapped network interface for the SHRIMP multicomputer. In *Proceedings of the 21st Annual Symposium on Computer Architecture* (April 1994), pp. 142–153.

[83] BODEN, N., COHEN, D., FELDERMAN, R., KULAWIK, A., SEITZ, C., SEIZOVIC, J., AND SU, W. Myrinet: A Gigabit-per-Second Local Area Network. *IEEE Micro 15*, 1 (Feb. 1995), pp. 29–38.

[84] BONGO, L. A. EventScope: Configurable On-line Monitoring of Parallel and Distributed Applications. Master's thesis, Department of Computer Science, University of Tromsø, Dec. 2002.

[85] BONGO, L. A. Steps: A Performance Monitoring and Visualization Tool for Multicluster Parallel Programs, June 2002. Large term project, Department of Computer Science, University of Tromsø.

[86] BRADSHAW, M. K., WANG, B., SEN, S., GAO, L., KUROSE, J., SHENOY, P., AND TOWSLEY, D. Periodic Broadcast and Patching Services - Implementation, Measurement, and Analysis in an Internet Streaming Video Testbed. ACM MM'01, Ottawa, Canada.

[87] BUONADONNA, P. Implementation and Analysis of the Virtual Interface Architecture. *Supercomputing '98, Orlando, FL* (Nov. 1998).

[88] BURNS, G., DAOUD, R., , AND VAIGL, J. LAM: An Open Cluster Environment for MPI. www.lam-mpi.org, 1994.

[89] BURNS, G., AND DAOUD, R. Robust Message Delivery with Guaranteed Resources. In *Proceedings, MPIDC'95* (May 1995).

[90] BUTLER, R., AND LUSK, E. User's guide to the p4 parallel programming system. Tech. Rep. ANL-92/17, Argonne National Laboratory, October 1992.

[91] CARRIERO, N., AND GELERNTER, D. Linda in Context. *Commun. ACM 32*, 4 (Apr. 1989), pp. 444–458.

[92] CHEN, J. B., ENDO, Y., CHAN, K., MAZIERES, D., DIAS, A., SELTZER, M., AND SMITH, M. The Measured Performance of Personal Computer Operating Systems. *ACM Transactions on Computer Systems* (February 1996).

[93] CHENG, D., AND HOOD, R. A portable debugger for parallel and distributed programs. In *Proceedings of the 1994 conference on Supercomputing* (1994), IEEE Computer Society Press, pp. 723–732.

[94] CHTCHELKANOVA, A., GUNNELS, J., MORROW, G., OVERFELT, J., AND VAN DE GEIJN, R. A. Parallel Implementation of BLAS: General Techniques for Level 3 BLAS. Tech. rep., The University of Texas at Austin. Austin, Texas 78712, 1995.

[95] CORIE project homepage. http://www.ccalmr.ogi.edu/CORIE/.

[96] COULSON, G. What is reflective middleware?, Dec. 2001. See computer.org/dsonline/middleware/RMarticle1.htm.

[97] CULLER, D. E., AND SINGH, J. P. *Parallel Computer Architecture - A hardware / software approach*. Morgan Kaufmann, 1999.

[98] DAMIANAKIS, S. N., CHEN, Y., AND FELTEN, E. Reducing Waiting Costs in User-Level Communication. In *11th International Parallel Processing Symposium (IPPS '97)* (April 1997).

[99] DEWDNEY, A. Computer Recreations. *Scientific American 250* (1984), pp. 22–34.

[100] DICKENS, P., HEIDELBERGER, P., AND NICOL, D. Parallel Direct Execution Simulation of Message-Passing Parallel Programs. *IEEE Transactions on Parallel and Distributed System* (1996).

[101] DRUSCHEL, P., AND PETERSON, L. L. Operating Systems and Network Interfaces. In *Foster, Ian and Kesselman, Carl (Eds.), The Grid: Blueprint for a New Computing Infrastructure* (1999), Morgan Kaufmann.

[102] ELCIRC homepage. http://www.ccalmr.ogi.edu/CORIE/modeling/elcirc.html.

[103] ERIKSSON, H. MBONE: The Multicast Backbone. *Communications of the ACM, Vol.37, No. 8* (Aug. 1994).

[104] FOSTER, I., AND KESSELMAN, C. Globus: A Metacomputing Infrastructure Toolkit. *The International Journal of Supercomputer Applications and High Performance Computing 11*, 2 (Summer 1997), pp. 115–128.

[105] FOSTER, I., AND KESSELMAN, C. Computational Grids. In *The Grid: Blueprint for a New Computing Infrastructure*, I. Foster and C. Kesselman, Eds. Morgan Kaufmann, San Francisco, CA, 1999, pp. 15–51. Chap. 2.

[106] Allegro Common Lisp 6.2 documentation. Franz Inc. http://www.franz.com/.

[107] FREEMAN, E., AND GELERNTER, D. Lifestreams: A Storage Model for Personal Data. *ACM SIGMOD Bulletin 2*, 2 (March 1996).

[108] FREEMAN, E., HUPFER, S., AND ARNOLD, K. *JavaSpaces(TM) Principles, Patterns and Practice*. Addison-Wesley, 1999.

[109] FREEMAN, E. T. *The Lifestreams Software Architecture*. PhD thesis, Yale University, 1997.

[110] 1,000-Pentium Beowulf-Style Cluster Computer for Genetic Programming. *http://www.genetic-programming.com/machine1000.html* (1999).

[111] Gigabit Ethernet: Accelerating the Standard for Speed, May 1999. Gigabit Ethernet Alliance. Technical report. White paper.

[112] Giganet cLAN. http://www.giganet.com/products/.

[113] GOVIL, K., TEODOSIU, D., AND YONGQIANG HUANG AND, M. R. Cellular Disco: resource management using virtual clusters on shared-memory multiprocessors. In *ACM Symposium on Operating Systems Principles (SOSP'99), published in Operating Systems Review 34(5)* (December 1999), pp. 154–169.

[114] GROPP, W., AND LUSK, E. Installation Guide to MPICH, a Portable Implementation of MPI, Version 1.2.4.

[115] GROPP, W., LUSK, E., DOSS, N., AND SKJELLUM, A. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing, Volume 22, Issue 6* (September 1996).

[116] GUSTAFSON, J. L. Reevaluating Amdahl's law. *Communications of the ACM 31*, 5 (1988), pp. 532–533.

[117] HALVORSEN, P. *Improving I/O Performance of Multimedia Servers*. PhD thesis, University of Oslo, 2001.

[118] HELME, A. Scheduling of Processes in a Distributed System using a Multi Dimensional Algorithm (in Norwegian). Master's thesis, Dept. of Computer Science, University of Tromsø, Tromsø, Norway, 1992.

[119] HENTY, D. S. Performance of Hybrid Message-Passing and Shared-Memory Parallelism for Discrete Element Modeling. In *Supercomputing* (2000).

[120] HUSBANDS, P., AND HOE, J. C. MPI-StarT: delivering network performance to numerical applications. *Proceedings of the 1998 ACM/IEEE conference on Supercomputing* (1998). San Jose, CA.

[121] HWANG, K., AND XU, Z. *Scalable Parallel Computing: Technology, Architecture, programming*. WCB/McGraw-Hill Co, 1998.

[122] MultiProcessor Specification version 1.4. Tech. rep., Intel Corp, 1997.

[123] JACUNSKI, M., SADAYAPPAN, P., AND PANDA, D. All-to-All Broadcast on Switch-based Clusters of Workstations. *13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing* (12 - 16 April 1999). San Juan, Puerto Rico.

[124] JOHNSON, K. L., KAASHOEK, M. F., AND WALLACH, D. A. CRL: High-Performance All-Software Distributed Shared Memory. In *Proceedings of the 15th Symposium on Operating Systems Principles (15th SOSP'95), Operating*

Systems Review (Copper Mountain, CO, Dec. 1995), ACM SIGOPS, pp. 213–228. Published as Proceedings of the 15th Symposium on Operating Systems Principles (15th SOSP'95), Operating Systems Review, volume 29, number 5.

[125] JOUBERT, P., KING, R., NEVES, R., RUSSINOVICH, M., AND TRACEY, J. High-Performance Memory-Based Web Servers: Kernel and User-Space Performance. *Proceedings of the 2001 USENIX Annual Technical Conference* (2001), pp. 175–188.

[126] JUURLINK, B. H., AND WIJSHOFF, H. A. A Quantitative Comparison of Parallel Computation Models. *ACM Transactions on Computer Systems Vol. 16*, No. 3 (August 1998), pp. 271–318.

[127] KARYPIS, G., AND KUMAR, V. *MeTis: Unstructured Graph Partitioning and Sparse Matrix Ordering System, Version 2.0*, 1995.

[128] KEENE, S. E. *Object-Oriented Programming in Common Lisp: A Programmer's Guide to CLOS*. Addison Wesley, 1988. ISBN 0-201-17589-4.

[129] KELEHER, P., COX, A., DWARKADAS, S., AND ZWAENEPOEL, W. TreadMarks: Distributed shared memory on standard workstations and operating systems. In *Proceedings of the Winter USENIX Conference* (January 1994), pp. 115–132.

[130] KICZALES, G., DES RIVIERES, J., AND BOBROW, D. G. *The Art of the Metaobject Protocol*. MIT Press, 1991.

[131] 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. *Proceedings of the seventh ACM SIGPLAN symposium on Principles and practice of parallel programming* (1999). Atlanta, Georgia, United States.

[132] KIELMANN, T., HOFMAN, R. F. H., BAL, H. E., PLAAT, A., AND BHOEDJANG, R. A. F. MPI's Reduction Operations in Clustered Wide Area Systems. *Proceedings of the seventh ACM SIGPLAN symposium on Principles and Practice of parallel programming* (1999). Atlanta, Georgia, United States.

[133] KOGGE, P. M. EXECUBE - A New Architecture for Scalable MPPs. In *In 1994 International Conference on Parallel Processing* (Washington - Brussels - Tokyo, Aug. 1994), IEEE, pp. I77–I84.

[134] KON, F., COSTA, F., BLAIR, G., AND CAMPBELL, R. H. The Case for Reflective Middleware. *Communications of the ACM, Vol. 45, No. 6* (June 2002).

[135] KOSTER, R., BLACK, A. P., HUANG, J., WALPOLE, J., , AND PU, C. Thread Transparency in Information Flow Middleware. In *Middleware 2001 – IFIP/ACM International Conference on Distributed Systems Platforms, Heidelberg, Germany* (November 2001), pp. 121–140.

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

[137] LI, K. Shared Virtual Memory on Loosely Coupled Multiprocessors. In *Proc. IEEE CS 1986, Int. Conf. on Computer Languages* (Miami, FL, Oct. 1986), pp. 98–106. YALEU/DCS/RR-492 September 1986, Yale University, New Haven, CT.

[138] LIAO, C., MARTONOSI, M., AND CLARK, D. W. Performance monitoring in a Myrinet-connected SHRIMP cluster. In *Proceedings of the SIGMETRICS symposium on Parallel and distributed tools* (1998), ACM Press, pp. 21–29.

[139] LUMETTA, S. S., MAINWARING, A. M., AND CULLER, D. E. Multi-Protocol Active Messages on a Cluster of SMP's. In *Proceedings of the 1997 ACM/IEEE SC97 Conference* (San Jose California, USA., Nov. 1997), ACM Press and IEEE, pp. 15–21.

[140] LUO, Y. MPI Performance Study on the SGI Origin 2000. *Pacific Rim Conference on Communications, Computers and Signal Processing* (1997), pp. 269–272.

[141] MALY, K. J., GUPTA, A. K., AND MYNAM, S. BTU: A Host Communication Benchmark. IEEE Computer, pp. 66-74, (May 1998).

[142] MARTONOSI, M., CLARK, D. W., AND MESARINA, M. The SHRIMP performance monitor: design and applications. In *Proceedings of the SIGMETRICS symposium on Parallel and distributed tools* (1996), ACM Press, pp. 61–69.

[143] MEIER, M. S., MILLER, K. L., PAZEL, D. P., RAO, J. R., AND RUSSELL, J. R. Experiences with building distributed debuggers. In *Proceedings of the SIGMETRICS symposium on Parallel and distributed tools* (1996), ACM Press, pp. 70–79.

[144] 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 (1995), pp. 37–46.

[145] MOSBERGER, D., AND PETERSON, L. L. Making Paths Explicit in the Scout Operating System. In *Operating Systems Design and Implementation* (1996), pp. 153–167.

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

[147] M-VIA Home Page, NERSC center at Lawrence Berkeley National Laboratory, http://www.nersc.gov/research/ftg/via/.

[148] NEVIN, N. J. The Performance of LAM 6.0 and MPICH 1.0.12 on a Workstation Cluster. Tech. Rep. OSC-TR-1996-4, Ohio Supercomputing Center, Columbus, Ohio, 1996.

[149] NIEPLOCHA, J., HARRISON, R., AND LITTLEFIELD, R. Global Arrays: A Portable Shared-Memory Programming Model for Distributed Memory Computers. In *Proceedings of the conference on Supercomputing '94* (1994), pp. 340–349.

[150] NORVIG, P. *Paradigms of Artificial Intelligence Programming: Case Studies in Common Lisp*. Morgan Kaufmann, 1992. ISBN 1-55860-191-0.

[151] O'HALLARON, D. Spark98: Sparse matrix kernels for shared memory and message passing systems, October 1997. Technical Report CMU-CS-97-178, School of Computer Science, Carnegie Mellon University.

[152] PAI, V. S., DRUSCHEL, P., AND ZWAENEPOEL, W. IO-Lite: A Unified I/O Buffering and Caching System. *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation* (1999).

[153] PANDA, D. Issues in Designing Efficient and Practical Algorithms for Collective Communication in Wormhole-Routed Systems. *Proc. ICPP Workshop Challenges for Parallel processing* (1995), pp. 8–15.

[154] PATTERSON, D., ANDERSON, T., CARDWELL, N., FROMM, R., KEETON, K., KOZYRAKIS, C., THOMAS, R., AND YELICK, K. Intelligent RAM (IRAM): Chips that remember and compute. In *1997 IEEE International Solids-State Circuits Conference. Digest of Technical Papers* (Washington - Brussels - Tokyo, Feb. 1997), IEEE, pp. 224–225.

[155] PATTERSON, D., ASANOVIC, K., BROWN, A., FROMM, R., GOLBUS, J., GRIBSTAD, B., KEETON, K., KOZYRAKIS, C., MARTIN, D., PERISSAKIS, S., THOMAS, R., TREUHAFT, N., AND YELICK, K. Intelligent RAM (IRAM): the Industrial Setting, Applications, and Architectures. In *International Conference on Computer Design: VLSI in Computers and Processors (ICCD '97)* (Washington - Brussels - Tokyo, Oct. 1997), IEEE, pp. 2–9.

[156] PATTERSON, D., CARDWELL, N., FROMM, R., KEETON, K., KOZYRAKIS, C., THOMAS, R., AND YELICK, K. A case for intelligent RAM. *IEEE Micro 2*, 2 (March-April 1997), 34–44.

[157] RAMACHANDRAN, U., NIKHIL, R. S., HAREL, N., REHG, J. M., AND KNOBE, K. Space-time memory: a parallel programming abstraction for interactive multimedia applications. In *Proceedings of the seventh ACM SIGPLAN symposium on Principles and practice of parallel programming* (1999), ACM Press, pp. 183–192.

[158] RANGARAJAN, M., AND IFTODE, L. Software Distributed Shared Memory over Virtual Interface Architecture: Implementation and Performance. Tech. Rep. DCS-TR-413, Rutgers University, Department of Computer Science, April 2000. To appear in Proceedings of The Third Extreme Linux Workshop, October 10-12, Atlanta.

[159] REED, D. A., MADHYASTHA, T. M., AYDT, R. A., ELFORD, C. L., SCULLIN, W. H., AND SMIRNI, E. I/O, Performance Analysis, and Performance Data Immersion. In *MASCOTS* (1996), pp. 5–15.

[160] REED, D. A., SHIELDS, K. A., SCULLIN, W. H., TAWERA, L. F., AND ELFORD, C. L. Virtual Reality and Parallel Systems Performance Analysis. *IEEE Computer 28*, 11 (1995), 57–67.

[161] REINHARDT, S., HILL, M. D., LARUS, J., LEBECK, A., J.C, LEWIS, AND WOOD, D. The Wisconsin Wind Tunnel: Virtual Prototyping of Parallel Computers. *Proceedings of the 1993 ACM SIGMETRICS Conference* (May 1993).

[162] RITCHIE, D. M. A Stream Input-Output System. Tech. Rep. 8, AT'&'T Bell Laboratories Techical Journal, October 1984.

[163] ROSENBLUM, M., BUGNION, E., DEVINE, S., AND HERROD, S. Using the SimOs Machine Simulator to Study Complex Computer Systems. *ACM Trans. On Modeling and Computer Simulation Vol. 7*, No. 1 (January 1997), pp. 78–103.

[164] SAMANTA, R., BILAS, A., IFTODE, L., AND SINGH, J. P. Home-based SVM protocols for SMP clusters: Design and Performance. In *Proc. of the 4th IEEE Symp. on High-Performance Computer Architecture (HPCA-4)* (Feb. 1998).

[165] SCALES, D. J., AND GHARACHORLOO, K. Design and Performance of the Shasta Distributed Shared Memory Protocol. In *Proceedings of the 11th International Conference on Supercomputing (ICS-97)* (New York, July 7–11 1997), ACM Press, pp. 245–252.

[166] SEBESTA, R. W. *Concepts of Programming Languages*. Addison Wesley, 2002. ISBN 0-201-75295-6.

[167] SIMPSON, W. IP in IP Tunneling, October 1995. Internet RFC 1853, Daydreamer.

[168] SINGH, J. P., WEBER, W.-D., AND GUPTA, A. SPLASH: Stanford Parallel Applications for Shared-Memory. *Computer Architecture News 20*, 1 (Mar. 1992), 2–12. Technical Report CSL-TR-91-469 1991 FTP mojave.stanford.edu, Computer Systems Laboratory, Stanford University.

[169] SISTARE, S., VANDEVAART, R., AND LOH, E. Optimization of MPI collectives on clusters of large-scale SMP's. *Proceedings of the 1999 conference on Supercomputing* (1999). Portland, Oregon, United States.

[170] SNOEREN, A. C., CONLEY, K., AND GIFFORD, D. K. Mesh-based content routing using XML. In *Proceedings of the eighteenth ACM symposium on Operating systems principles* (2001), ACM Press, pp. 160–173.

[171] SPEIGHT, E., ABDEL-SHAFI, H., AND BENNETT, K. Realizing the Performance Potential of the Virtual Interface Architecture. In *International Conference on Supercomputing* (June 1999).

[172] SQUYRES, J. M., LUMSDAINE, A., GEORGE, W. L., HAGEDORN, J. G., AND DEVANEY, J. E. The Interoperable Message Passing Interface (IMPI) Extensions to LAM/MPI. In *Proceedings, MPIDC'2000* (March 2000).

[173] STABELL-KULØ, T. A Partial Implementation of the MacroScope Distributed Operating System (in Norwegian). Master's thesis, Dept. of Computer Science, University of Tromsø, Tromsø, Norway, 1992.

[174] STONE, H. S. A Logic-in-Memory Computer. *IEEE Transactions on Computers 19*, 1 (January 1970), pp. 73–78.

[175] SUNDERAM, V. PVM: A Framework for Parallel Distributed Computing. In *Concurrency: Practice and Experience, Vol. 2, No. 4* (Dec. 1990).

[176] JavaSpaces Specification, revision 1.0, January 1999. Sun Microsystems.

[177] TAKAHASHI, T., O'CARROLL, F., TEZUKA, H., HORI, A., SUMIMOTO, S., HARADA, H., ISHIKAWA, Y., AND BECKMAN, P. H. Implementation and Evaluation of MPI on an SMP Cluster. In *IPPS/SPDP Workshops* (1999), pp. 1178–1192.

[178] TANG, H., AND YANG, T. Optimizing threaded MPI execution on SMP clusters. *Proceedings of the 15th international conference on Supercomputing* (2001). Sorrento, Italy.

[179] TENNENHOUSE, D. L., SMITH, J. M., SINCOSKIE, W. D., WETHERALL, D. J., AND MINDEN, G. J. A Survey of Active Network Research. *IEEE Communications Magazine 35*, 1 (1997), pp. 80–86.

[180] TENNENHOUSE, D. L., AND WETHERALL, D. J. Towards an Active Network Architecture. *Computer Communication Review 26*, 2 (1996).

[181] TOUCH, J., AND HOTZ, S. The X-Bone. *Proc. Third Global Internet Mini-Conference at Globecom '98 Sydney, Australia Nov. 8-12* (1998), pp 75–83 (listed as pp. 44–52 of the miniconference).

[182] TRAFF, J. L. Implementing the MPI Process Topology Mechanism. *Supercomputing 2002*.

[183] TSpaces homepage. http://www.almaden.ibm.com/cs/TSpaces/.

[184] VADHIYAR, S. S., FAGG, G. E., AND DONGARRA, J. Automatically Tuned Collective Communications. *SuperComputing* (2000), pp. 46. Dallas, Texas.

[185] VAN RENESSE, R., BIRMAN, K. P., FRIEDMAN, R., HAYDEN, M., AND KARR, D. A. A Framework for Protocol Composition in Horus. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing* (Ottawa, Ontario, Canada, 2–23 1995), pp. 80–89.

[186] VENUGOPAL, S., AND NAIK, V. K. Effects of Partitioning and Scheduling Sparse Matrix Factorization on Communication and Load Balance. *Proceedings of the 1991 conference on Supercomputing* (1991), pp. 866–875.

[187] VETTER, J. Performance analysis of distributed applications using automatic classification of communication inefficiencies. In *International Conference on Supercomputing* (2000), pp. 245–254.

[188] Virtual Interface (VI) Architecture - Implementation Guide. Draft revision 0.6., February 1998. Intel Corporation.

[189] VINTER, B. *PastSet a Structured Distributed Shared Memory System*. PhD thesis, Tromsø University, 1999.

[190] VINTER, B., ANSHUS, O., AND LARSEN, T. Data Distribution Models for a Structured Distributed Shared Memory System. In *Proc. Of the international conference on Parallel and Distributed Programming Techniques and Applications, PDPTA 99* (Las Vegas, June 1999).

[191] VINTER, B., ANSHUS, O. J., AND LARSEN, T. PastSet - A Distributed Structured Shared Memory System. In *Proc. of High Performance Computers and Networking* (Amsterdam, April 1999).

[192] VINTER, B., ANSHUS, O. J., LARSEN, T., AND BJØRNDALEN, J. M. Extending the Applicability of Software DSM by Adding User Redefinable Memory Semantics. *Parallel Computing (ParCo) 2001, Naples, Italy* (Sept. 2001).

[193] VINTER, B., ANSHUS, O. J., LARSEN, T., AND BJØRNDALEN, J. M. Using Two-, Four- and Eight-Way Multiprocessors as Cluster Components. *CPA, Communicating Process Architectures* (September 2001).

[194] VINTER, B., LARSEN, T., AND ANSHUS, O. J. Improving Cluster Performance using a Causally Ordered Structured Distributed Shared Memory System. *Norsk Informatik Konferense* (1999).

[195] VON EICKEN, T., BASU, A., BUCH, V., AND VOGELS, W. U-Net: A User-Level Network Interface for Parallel and Distributed Computing. *The 15th ACM Symposium on Operating Systems Principles (SOSP)* (December 1995).

[196] VON EICKEN, T., CULLER, D. E., GOLDSTEIN, S. C., AND SCHAUSER, K. E. Active messages: A mechanism for integrated communication and computation. In Proceedings of 19th International Symposium on Computer Architecture, pages 256-266, May 1992.

[197] WALKER, D. W. The design of a standard message passing interface for distributed memory concurrent computers. *Parallel Computing 20*, 4 (Mar. 1994), pp. 657–673.

[198] WILSON, L. F., AND NICOL, D. M. Experiments in Automated Load Balancing. *Proceedings of the 10th Workshop on Parallel and Distributed Simulation (PADS '96)* (1996).

[199] WYCKOFF, P., MCLAUGHRY, S. W., LEHMAN, T. J., AND FORD, D. A. T Spaces. *IBM Systems Journal, 37(3):454474* (1998).

[200] XML-RPC Home Page:. http://www.xml-rpc.com/.

Appendix A

Papers

2000

Paper 1 *Comparing the Performance of the PastSet Distributed Shared Memory System using TCP/IP and M-VIA*
John Markus Bjørndalen, Otto Anshus, Brian Vinter, Tore Larsen
WSDSM'00, The Second International Workshop on Software Distributed Shared Memory, Santa Fe, New Mexico, 2000-05-07

Paper 2 *The Impact on Latency and Bandwidth for a Distributed Shared Memory System Using a Gigabit Network Supporting the Virtual Interface Architecture*
John Markus Bjørndalen, Otto Anshus, Brian Vinter, Tore Larsen
NIK 2000, Norsk Informatikk Konferanse, Bodø, Norway, Autumn 2000

2001

Paper 3 *Using Two-, Four- and Eight- Way Multiprocessors as Cluster Components*
Brian Vinter, Otto Anshus, Tore Larsen, John Markus Bjørndalen
CPA 2001, Communicating Process Architectures 2001, September 16-19, Bristol, UK

Paper 4 *Extending the Applicability of software DSM by adding user redefinable memory semantics*
Brian Vinter, Otto Anshus, Tore Larsen, John Markus Bjørndalen
ParCo 2001, Parallel Computing 2001, 4 - 7 September, Via Cintia, I-80126 NAPLES, Italy

Paper 5 *PATHS - Integrating the Principles of Method-Combination and Remote Procedure Calls for Run-Time Configuration and Tuning of High-Performance Distributed Applications*
John Markus Bjørndalen, Otto Anshus, Tore Larsen, Brian Vinter
NIK 2001, Norsk Informatikk Konferanse, Tromsø, Norway, 2001

2002

Paper 6 *Scalable Processing and Communication Performance in a Multi-Media Related Context*
John Markus Bjørndalen, Otto Anshus, Tore Larsen, Lars Ailo Bongo, Brian Vinter
Euromicro 2002, Dortmund, Germany, September 2002

Paper 7 *Configurable Collective Communication in LAM-MPI*
John Markus Bjørndalen, Otto Anshus, Brian Vinter, Tore Larsen
CPA 2002, Communicating Process Architectures 2002, September 2002, Reading, UK

Paper 8 *The Performance of Configurable Collective Communication for LAM-MPI in Clusters and Multi-Clusters*
John Markus Bjørndalen, Otto Anshus, Brian Vinter, Tore Larsen
NIK 2002, Norsk Informatikk Konferanse, Kongsberg, Norway, November 2002

Paper 9 *The latency of user-to-user, kernel-to-kernel and interrupt-to-interrupt level communication*
John Markus Bjørndalen, Otto Anshus, Brian Vinter, Tore Larsen
NIK 2002, Norsk Informatikk Konferanse, Kongsberg, Norway, November 2002

Submitted

Paper 10 *Cluster Monitoring with Steps: Making the Application Behaviour Visible*
Lars Ailo Bongo, John Markus Bjørndalen, Otto J. Anshus

A.1 Comparing the Performance of the PastSet Distributed Shared Memory System using TCP/IP and M-VIA

Comparing the Performance of the PastSet Distributed Shared Memory System using TCP/IP and M-VIA

John Markus Bjørndalen, Otto J. Anshus, Brian Vinter[1], Tore Larsen
Department of Computer Science
University of Tromsø
[1]Department of Mathematics and Computer Science
University of Southern Denmark
johnm@cs.uit.no, otto@cs.uit.no, vinter@imada.sdu.dk, tore@cs.uit.no

Abstract

Using TCP/IP or M-VIA, the performance of the structured distributed shared memory system PastSet is measured and compared to a reference single-node implementation (excluding all intra-node communication). The latencies of PastSet-operations are measured using several micro-benchmarks. For the experiment setup used, M-VIA latencies are shown to be between 1.4 and 2.2 times lower than the comparable latencies using TCP/IP. For a data size of 31KB, this corresponds to a difference of more than one millisecond. Depending on the thread-allocation policy applied in the PastSet server, PastSet latencies using TCP/IP may exhibit increased variance compared to the corresponding latencies using M-VIA. The increased latency and variance may mask the performance characteristics of the PastSet system.

1. Introduction

The application-to-application performance of a Distributed Shared Memory (DSM) system depends on the performance and interaction of the DSM and the underlying network subsystems. The key challenge [16] is to preserve the performance characteristics of the physical network (bandwidth, latency, QoS) while making effective use of host resources. Network bandwidths have been increasing and latencies through these networks have been decreasing. Unfortunately, applications have not been able to take full advantage of these performance improvements due to the many layers of user level and kernel level software required to use the network. A detailed breakdown of hardware and software costs of remote memory operations is discussed in [5]. The Virtual Interface Architecture was developed to significantly reduce the software overhead between a high performance CPU/memory subsystem and a high performance network.

In this paper, we study the application-to-application performance of the PastSet DSM system using either M-VIA, a software VIA implementation for Linux [6], or TCP/IP. The report describes the functionality of PastSet, the organization of the implementation; and the interactions between PastSet user-level components, M-VIA, and a PastSet-modified Linux kernel. An experiment configuration with micro-benchmarks and metrics is described before presenting and analyzing benchmark results.

2. PastSet

2.1. Model

PastSet was first introduced as a structured shared memory in [1]. Early partial implementations of PastSet include [2, 3]. [4] develops and demonstrates an extended PastSet programming model, shows how parallel applications are written using PastSet, and documents the performance of these applications on the authors implementation of PastSet. The PastSet paradigm resembles that of Linda [18], but with added structuring of the shared memory and different functionality of the operations provided. Comparable, efforts within industry include IBM TSpaces [15] and JavaSpaces [14].

All PastSet operations are synchronous, returning only when the operation is completed, or an error has been detected. Processes using PastSet dynamically generate tuples based on tuple templates that may also be generated dynamically. A tuple template specifies a list of data types. A tuple is a list of values matching the data types specified in the template upon which the tuple is based.

The 'elements' of PastSet are lists of tuples; one list per unique template used. The `enter` operator takes a tuple template as parameter and establishes a binding from that template to the associated element in PastSet. If the template is unique (i.e. no identical template has been specified for previously executed `enter`-operations), a new element is created. If the template is not unique, the binding is established with the element already associated with the identical templates. As with Linda, PastSet sup-

ports writing (called `move`) tuples into PastSet and reading (called `observe`) tuples that reside in PastSet. A tuple that is moved into PastSet is added to the associated element in PastSet and remains in that element as a unique tuple. For each element, tuples are added and observed in FIFO or program-specified order as described in [4]. Two identifiers `First` and `Last` are associated with each element in PastSet. `First` refers to the elements oldest unobserved tuple. `Last` refers to the tuple most recently added to the element. A parameter, `DeltaValue`, associated with each element in PastSet defines the maximum number of tuples allowed between `First` and `Last` for that element. A process may change `DeltaValue` at any time. `Move` and `observe` update `First` and `Last`, and obey the restrictions imposed by `DeltaValue` for each element in PastSet. PastSet preserves the sequential order among `move` and `observe` operations on tuples based on identical templates. There is no ordering among operations on tuples based on different templates.

A combined `move-observe` operation, `mob`, is provided to support efficiently the commonly used sequence of a `move` immediately followed by an `observe`. `Mob` takes two tuples as parameters and operates on two elements if the tuples are based on different templates; if not, `mob` operates on one element.

Contrary to similar systems, PastSet `observe` does not remove tuples from PastSet, observed tuples are tagged 'observed' but remain in PastSet and may be observed again later. A mechanism is provided to coarsely truncate PastSet on a per element basis, permanently removing all tuples that are older than a given tuple. There is no mechanism to remove individual tuples from PastSet.

2.2. Organization

One or more nodes may host PastSet. Each hosting node runs a PastSet kernel, a server, and an application library (Fig. 1). Currently, each element in PastSet is stored on one host only. There is no distribution, replication, or migration of individual elements. Nodes that use PastSet without hosting it will run the application library and TCP/IP or M-VIA only.

The organization supports PastSet operations on elements that are located on the same node as the initiating process (local operations), as well as operations on elements that are located on other nodes (remote operations). Remote operations are wrapped and communicated to the PastSet server on a remote node. `Mob` operations work on one or two elements, each of which may be local or remote.

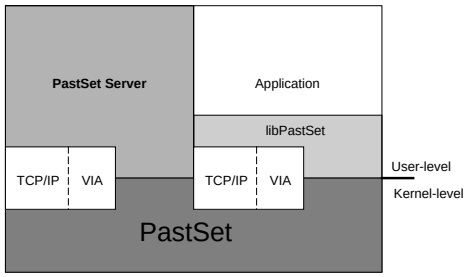


Fig. 1. Layout of a node that supports PastSet

THE PASTSET APPLICATION LIBRARY handles access to PastSet, including multiplexing between local and remote execution paths.

All PastSet operations check to determine whether the element that is to be operated on is hosted locally or remotely. If the element is hosted locally, the operation is executed using the local PastSet Kernel. If the element is hosted remotely, the operation is redirected to the PastSet Server on the appropriate host. The server, upon completion, returns the answer through the application library to the initiating process.

New tuples with additional space for communication headers are also allocated via the application library.

THE PASTSET SERVER executes remotely issued PastSet operations on local elements using the local PastSet Kernel. Since operations are blocking, the server must be able to service several operations concurrently.

THE PASTSET KERNEL is a modified Linux kernel that stores PastSet elements and services operations issued by the local PastSet Server or the application library.

3. Implementing the PastSet Server and Application Library

Two approaches to handling communication were used in the PastSet server. The *Single Thread* approach spawns a new thread for each new client connection. The thread is given a pointer to the new connection and handles that connection exclusively.
The *Thread Pool* approach uses a pool of threads, which multiplexes handling of multiple connections.

The synchronous nature of PastSet necessitates a reply with the result of an operation before the client can issue a new operation. Consequently, there are two messages for each remote operation.

The current implementation of the PastSet Application Library can not handle multithreaded clients.

3.1. TCP/IP implementation

The client library uses ordinary TCP/IP connections to remote servers. If an element is located on a remote node, the operation and its parameters are sent over the TCP/IP socket to the remote PastSet server. The client library then blocks waiting for the reply from the server.

When using the Single Thread approach, the PastSet server creates a new thread when a new socket arrives. The newly created thread is given the file descriptor of the socket and immediately tries to read data from the socket. It blocks if there is no data available.

In the Thread Pool approach, when a connection is accepted, the file descriptor is added to the list of active connections. A set of threads use the *select()* system call to multiplex themselves between the active sockets. To avoid race conditions, the *select()* call and the subsequent read of a socket with data is protected with a mutex.

We disable the Nagle algorithm on all sockets to ensure that data is sent immediately.

3.2. M-VIA implementation

The PastSet server and the application library were implemented using the M-VIA 1.0 [6] implementation of the VIA API. We have used the message passing model of VIA since the port from the TCP/IP implementation was straightforward.

The NICs used do not support the doorbell mechanism, and M-VIA has to emulate this in software. This results in traps to the Linux kernel.

The Thread Pool model was implemented using VIA Completion Queues. Because M-VIA is not thread safe per VI we protected the calls to the Completion Queue and the per VI operations using mutexes.

To reduce the CPU use of the PastSet server we used blocking calls to wait for completed descriptors. M-VIA implements these blocking calls by first spinning a few times with the respective non-blocking functions to avoid going to the kernel if the descriptor is already completed.

Tuples used in the application are allocated in memory registered with the VIA NICs to reduce copying on send and receive.

4. Methodology

This section documents the hardware and software details of the experiments, how the timing measurements were done, the micro-benchmarks, and the metrics used.

4.1. Hardware and Software

All experiments were done using one, two, or three HP LX-Pro Net-servers, each having four 166MHz Pentium Pro CPUs and 128MB main-memory, and dual peer 33MHz, 32 bit PCI buses. The level 2 cache size is 1MB per processor. The computers were interconnected using either Intel 82255 or Trendnet TE100-PCIA (with Tulip chip set) NIC-cards connected to a hub. Both NICs, and one 100VG NIC connected to the outside network, were connected to PCI bus no. 0.

Linux v. 2.2.14 with PastSet functionality added to the kernel was installed on each node participating in the experiments. M-VIA version 1.0 with a minor patch to the connection management was used. The benchmarks and the PastSet Application Library were compiled with gcc 2.95.2 using optimization flags "-O6 -m486 -mjumps=2 -malignloops=2 –malignfunctions=2." M-VIA, The PastSet Server, the PastSet Kernel, and the Linux operating system were compiled with egcs 1.1.2 using default flags.

For some experiments we could not get M-VIA running on the SMP-configurations. To circumvent this problem, we had to resort to compiling the Linux kernel to run as a single processor system rather than using all four processors in a node.

4.2. Time Measurements

The Intel Pentium Pro RDTSC (read time-stamp counter) instruction and the Linux gettimeofday system call were used to determine PastSet operation latencies.

Using RDTSC, as in [17], the cycle count was recorded for every `move` and `observe` operation. Elapsed time in microseconds was calculated by dividing the registered cycle count by the specified processor frequency of 166 MHz. No attempts were made to verify the actual frequency of each individual computer, leaving open the possibility that the computed time may deviate slightly, but consistently, from the performance measured in cycles spent. Care was taken to avoid potential problems with register overwrites and counter overflow.

The *gettimeofday()* system call was used for aggregate measurements over many operation calls. Checks were made to ensure that RDTSC and *gettimeofday()* measurements were consistent.

Cache effects are not eliminated, but measurements are averaged over 1000 iterations.

4.3. Micro-benchmarks and Metrics

Two micro-benchmarks that measure operation and ping-pong latencies of the PastSet system were designed:

- *`Move latency (mvlat)`*, *`observe latency (oblat)`*: Time to invoke, complete and return from a `move` or `observe` operation.
- *Ping-pong latency (pplat)*: Time to exchange data between two processes using `moveobserve`.

The benchmarks were executed inside client processes running both on the same computer as PastSet *("Local Latencies")* and on remote computers. When using more than one computer TCP/IP or M-VIA were used for communication.

When doing the performance measurements each node supported no other workload except for the operating system and its various artifacts.

5. Micro-benchmark Results

5.1. Operation Latency Experiment Design

PastSet "operation latency" is defined to be the time elapsed from a `move`, `observe`, or `mob` operation is called until it has completed and returned successfully. For `observe` operations it is assumed that enough tuples are available in PastSet to prevent the operations from blocking for lack of tuples. All necessary initializations are done before starting time- or cycle measurements.

```
for(i=0; i<1000; i++)
{
    save_timestamp;
    mv();
}
save_timestamp;
```

Fig. 2: The Mvlat benchmark

The client process executes `move` or `observe` operations. Data size per operation call is varied from zero to 31KB. The elapsed time is measured for each operation call. Each call is repeated 1.000 times. This is repeated five times, and the average is computed.

Due to space constraints, results are shown only for the move operation. The observe operation exhibit slightly different behavior, but is close in performance.

Three experiments were designed to measure the latency of the move operation:

- Local Move Latency: The client process and PastSet are on the same node.
- TCP/IP Move Latency: The client process and PastSet are on different nodes. TCP/IP is used for intra-node communication.
- M-VIA Move Latency: The client process and PastSet are on different nodes. M-VIA is used for intra-node communication.

The latency experiments were conducted on the following configurations, using both SMP and uniprocessor versions of Linux:

- A pool of threads is used in the PastSet server to serve all connections ("Thread Pool").
- A single thread is used in the PastSet server per connection ("Single Thread").

Because of problems experienced with M-VIA, the M-VIA Move Latency experiment was conducted only for the "single thread" version .

The configurations used for the experiments are summed up in table 1.

Table 1: Configurations

	Local	**Using M-VIA**		**Using TCP/IP**	
SMP	(Fig. 3 & 4)	Intel NIC, Thread Pool (Fig.3 & 4)		Intel NIC, Thread pool (Fig. 3 & 4)	
Uni-proc essor	(Not shown)	TREN Dnet NIC, Single thread (Fig. 5)	TREND-net NIC, Thread Pool (Not done)	TREND-net NIC, Single thread (Fig. 5)	TREND-net NIC, Thread pool (Fig. 5)

5.2. Move Latency Results

Fig. 3 shows move latencies for local (one node only) communication, and for intra-node communication using TCP/IP or M-VIA. Tuple sizes are varied from one byte to 31KB. The results are measured using Linux in SMP mode, a thread pool in the PastSet Server for data in/out servicing, and using the Intel NICs.

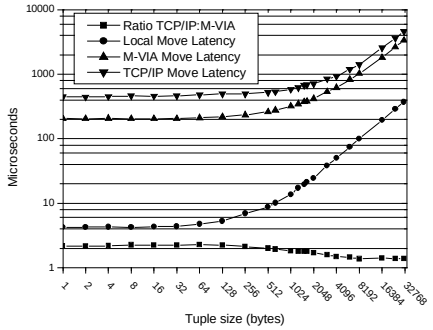


Fig. 3. *The mvlat benchmark*: Operation latency for `move`

Fig. 3 shows that local move latency (clients and server on same node) changes from about two to one order of magnitude better than M-VIA or TCP/IP as tuple size is varied from 1 byte to 32 KB. The difference in performance is due to the extra overhead caused by network communication and using the PastSet Server.

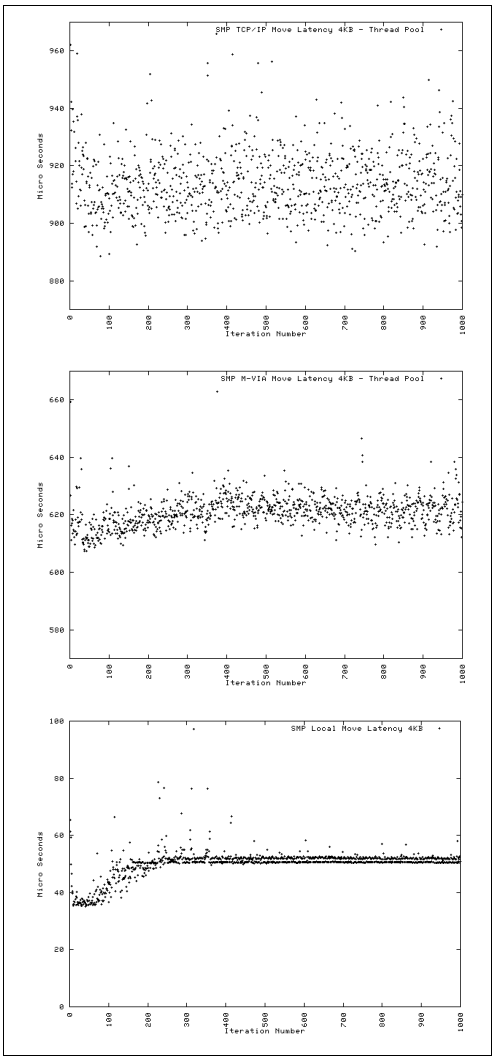


Fig. 4. *The mvlat benchmark*: `move` latency for tuple size 4KB for 1000 measurements, SMP, Thread

For small tuple sizes M-VIA `move` latency is less than half the latency when using TCP/IP. With increasing tuple size, the M-VIA advantage decreases to about 2/3 for 31 KB tuples. This is a large difference in absolute numbers. For one byte tuples, the difference between using M-VIA and TCP/IP is 242 microseconds, while at 31KB the difference is 1256 microseconds. We explain the advantage of M-VIA over TCP/IP partly by the user level communication used by M-VIA and the faster traps to the operating system. Other contributing factors are that M-VIA does not compute checksums of the incoming packets, taking advantage of the properties of a local net. However, TCP/IP uses the operating system much more heavily, and does more copying than M-VIA. Fig. 4 shows the `move` latency of 1000 `move` operations for a single tuple size, 4KB. The results are measured using Linux in SMP mode, using a thread pool in the PastSet Server for data in/out servicing, and using Intel NICs.

We have plotted three cases: local, TCP/IP and M-VIA. The TCP/IP measurements include a few very high (factor 10) values that we have removed from the plot. We believe that these values are the result of 10ms time slice events.

The results for the local `move` latency clearly show an effect coming from the way PastSet implements receiving and storing of tuples. The storage structure of PastSet seeks to make it efficient to access the newest tuples. Three levels of indirection are used to achieve this ("reverse I-nodes"). When a stream of tuples are sent to PastSet the cost of inserting them grows steadily until we are at level three in the datastructure. This effect can be seen in the local move latency plot, and to a lesser extent also in the M-VIA plot.

The measurements show that M-VIA has a lower variance while TCP/IP gives a much more unpredictable latency. We believe that the TCP/IP latency is long enough to include relatively more events (including interrupts and scheduling) happening in the total system resulting in a high variance. Also, the thread pool in the PastSet Server uses the *select()* system call when using TCP/IP. This seems to be more expensive than using the VIA completion queue mechanism.

The results suggest that M-VIA and the way we use it, even when using a slow 100Mbit network, is just fast enough to let the basic behavior of the PastSet system become visible in the measurements. This is due both to the better latency of M-VIA and less variance.

TCP/IP does not reveal the behavior of PastSet, while M-VIA does.

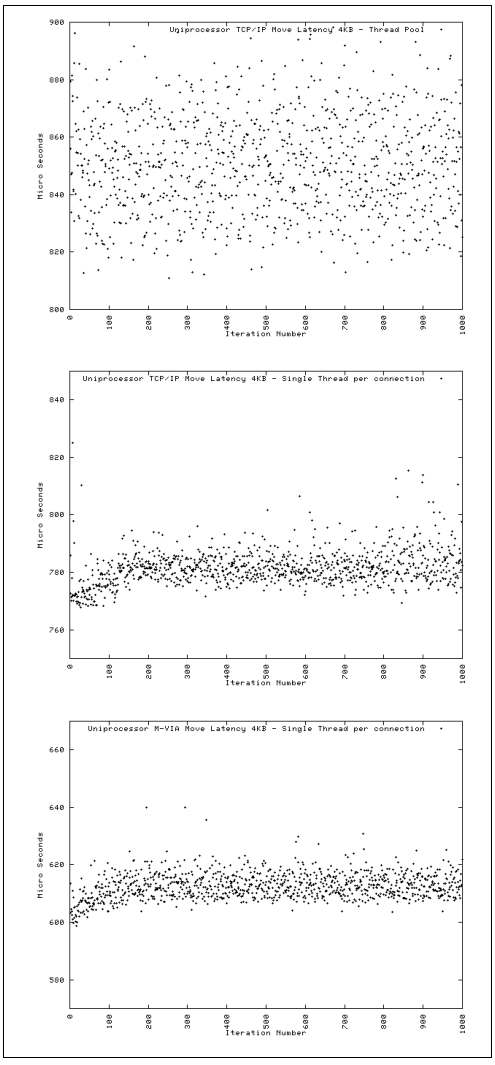


Fig. 5: *The mvlat benchmark*: `move` latency for tuple size 4KB for 1000 measurements, Uniprocessor and SMP, Single Thread and Thread Pool, TRENDnet NIC.

Fig. 5 shows the move latency of 1000 move operations for a single tuple size, 4KB. The results are measured using Linux in uniprocessor mode, and TRENDnet NICs are used. The PastSet Server uses a single thread for each connection for the M-VIA, and a single thread or a thread pool for the TCP/IP. We have not been able to use a thread pool for the M-VIA measurements using TRENDnet NICs due to problems with M-VIA.

The results from fig. 5 show that TCP/IP improves significantly when using a single thread to handle the benchmark connection as compared to using a thread pool. TCP/IP is still slower than M-VIA, but the variance has improved, and is just slightly worse than for M-VIA. We explain this with the way the PastSet server handles TCP/IP connections. In the single thread per connection configuration there will almost always be a thread ready to read incoming data, and this thread will be the same every time. In the thread pool configuration a new thread will serve each incoming packet. We believe this has impact on the cache footprint giving more variance and worse results. Also, the thread pool configuration executes more instructions.

We have not been able to measure the move latency when using M-VIA, TRENDnet NICs, and a thread pool on a Linux uniprocessor configuration. If we assume that M-VIA will behave about the same or better than TCP/IP, then we can conclude that M-VIA is less influenced than TCP/IP on whether a single thread or a thread pool is used in the PastSet server. TCP/IP is much more sensitive to this as can be seen by comparing the two TCP/IP results in fig. 5. We explain this difference in sensitivity to the M-VIAs better utilization of user level communication.

Generally, TRENDnet NICs are faster than the older Intel NICs.

5.3. Ping pong latency

PastSet "ping pong latency" is the time period elapsed between the repeated lock stepped exchanges of a value between two processes. What we measure will include potential waiting by the two processes for each other to rendezvous. All initializations have been done before we start counting time.

```
gettimeofday();
for(i=0; i<1000; i++){
  mob(); // Exchange a value with
the other process
}
gettimeofday();
```

Fig. 6. The PPlat Benchmark

For the ping-pong latency performance measurements, we execute the micro-benchmark using two processes. Both processes loops doing a given number of mob() operations. One process moves data to element e_1, and tries to pick up data from element e_2. The other process moves data to element e_2 and tries to get data from e_1. In this way, the processes exchange data in a lock step fashion.

The data units range in size from zero to 31KB. The elapsed time to exchange all data is measured, and the time per exchange is computed. The step locked operations are repeated 1000 times to eliminate noise.

We did five ping-pong experiments (in the legend tag, L means local, R means remote relative to the location of PastSet):

- LL: The two processes and PastSet are on the same computer.
- M-VIA LR: One process is on one computer, the other process and PastSet are on another computer. VIA is used for communication between the two computers.
- TCP/IP LR: One process is on one computer, the other process and PastSet are on another computer. TCP/IP is used for communication between the two computers.
- M-VIA RR: The two processes and PastSet are all on different computers. M-VIA is used for communication between the three computers.
- TCP/IP RR: The two processes and PastSet are all on different computers. M-VIA is used for communication between the three computers.

The results presented in fig. 7 show that the ratio between using TCP/IP vs. M-VIA when the communication processes are on two different nodes behaves as the same ratio for the move latency. The advantage of using M-VIA decreases when the tuple size increases. However, M-VIA has a significantly better latency at all tuple sizes, and especially for tuple sizes between 0-256 bytes.

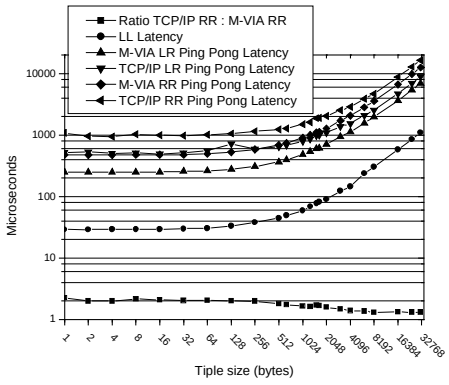


Fig. 7. *The Pplat Benchmark*

Fig. 7 also shows that for tuple sizes up to 2KB, using M-VIA on a three-node configuration (RR) gives about the same latency as when using TCP/IP on a two-node configuration (LR). Thus, M-VIA is fast enough to make up for the extra communication taking place.

6. Related work

Much effort has been put into cluster communication using either a shared memory model or an explicit communication model.

When comparing our results with the results reported in [19] we find that the ratio between TCP/IP vs. M-VIA latency is about the same and around 2.25-2.26.

Distributed Shared Memory implementations include Princeton Shrimp SVM [8] and Rice TreadMarks [9]. Object based Distributed Shared Memory systems include Orca [10]. Noteworthy examples of message-passing systems include Message Passing Interface, MPI [11], and Parallel Virtual Machines, PVM [12]. Less work has been done using Structured Distributed Shared Memory. The most well known systems include Linda [18], and more recently, Global Arrays [13].

7. Conclusions

Porting PastSet from TCP/IP to M-VIA proved straightforward. However, we have identified several bugs or limitations of the M-VIA implementation we used, and we still do not have a stabile system available.

When using a DSM system a predictable performance is desirable, and in addition to being faster, the latency of M-VIA is more predictable than the latency of TCP/IP. Operations on tuples of 256 or fewer bytes are twice as fast when using M-VIA.

By using a standard API such as VIA, system designers can achieve the benefits of user-level communication, while still maintaining portability.

8. Acknowledgements

Ken Arne Jensen and Jon Ivar Kristiansen provided invaluable support under the preparation of this paper.

9. References

[1] Anshus, O.J., Larsen, T.: “MacroScope: The Abstractions of a Distributed Operating System”. Norsk Informatikk Konferanse 1992, October 1992.

[2] Helme, A., “Scheduling of Processes in a Distributed System using a Multi Dimensional Algorithm” (in Norwegian), Master Thesis, Dept. of Computer Science, University of Tromsø, Tromsø, Norway, 1992.

[3] Stabell-Kulø, T., “A Partial Implementation of the MacroScope Distributed Operating System (in Norwegian)”, Master Thesis, Dept. of Computer Science, University of Tromsø, Tromsø, Norway, 1992.

[4] Brian Vinter, “PastSet a Structured Distributed Shared Memory System”, Dr. Scient. Thesis, Tromsø University, 1999.

[5] Bilas, A., Iftode, L., Singh, J. P., “Evaluation of Hardware Support for Automatic Update in Shared Virtual Memory Clusters”. 12th ACM International Conference on Supercomputing, July, 1998
[6] http://www.nersc.gov/research/ftg/via/
[7] Brian Vinter, Tore Larsen and Otto J. Anshus, “Improving Cluster Performance using a Causally Ordered Structured Distributed Shared Memory System”, Norsk Informatik Konferense ‘.99, 1999.
[8] Blumrich, M., Li, K., Alpert, R., Dubnicki, C., Felten, E., Sandberg, J.: “A virtual memory mapped network interface for the shrimp multicomputer”. In Proceedings of the 21st Annual Symposium on Computer Architecture, pages 142–153, Apr. 1994.
[9] Keleher, P., Cox, A., Dwarkadas, S., Zwaenepoel, W.: “TreadMarks: Distributed shared memory on standard workstations and operating systems”. In Proceedings of the Winter USENIX Conference, pages 115–132, Jan.1994.
[10] Bal, H.E., Kaashoek, M.F., Tanenbaum, A.S.: ‘Orca: A Language For Parallel ProgrammingOf Distributed Systems”. IEEE Computer 25(8), pp. 10-19, Aug. 1992.
[11] Walker, D.W.: “The Design of a Standard Message-Passing Interface for Distributed Memory Concurrent Computers”. Parallel Computing, Vol. 20, No. 4, pages 657-673, April 1994
[12] Sunderam, V.S.: “PVM: A Framework for Parallel Distributed Computing”. Concurrency: Practice and Experience, Vol. 2, No. 4, Dec. 1990.
[13] Nieplocha, J., Harrison, R.J., Littlefield, R.J.: “Global Arrays: A Portable Shared-Memory Programming Model for Distributed Memory Computers”. Proceedings of the conference on Supercomputing ‘94, pages 340-ff., 1994
[14] Eric Freeman, Susanne Hupfer, Ken Arnold, “JavaSpaces(TM) Principles, Patterns and Practice”, SUN MicroSystems
[15] http://www.almaden.ibm.com/cs/TSpaces/
[16] Druschel, Peter and Peterson, Larry L. “Operating Systems and Network Interfaces,” In Foster, Ian and Kesselman, Carl (Eds.), The Grid: Blueprint for a New Computing Infrastructure, Morgan Kaufmann, 1999
[17] Chen, J. B., Endo, Y., Chan, K. Mazieres, D., Dias, A., Seltzer, M., Smith, M.D.: “The Measured Performance of Personal Computer Operating Systems”. ACM Transactions on Computer Systems, February 1996.
[18] Carriero, N., Gelernter, D., “Linda in Context”, Commun. ACM, (April 1989), Vol. 32, No. 4, pp. 444-458]
[19] Speight, E., Abdel-Shafi, H., Bennett, K., “Realizing the Performance Potential of the Virtual Interface Architecture”, International Conference on Supercomputing, June 1999
[20] Vinter, B., Anshus, O.J., Larsen, T., “Data Distribution Models for a Structured Distributed Shared Memory System”, Proc. Of the international conference on Parallel and Distributed Programming Techniques and Applications, PDPTA 99, Las Vegas June 1999

A.2 The Impact on Latency and Bandwidth for a Distributed Shared Memory System Using a Gigabit Network Supporting the Virtual Interface Architecture

The Impact on Latency and Bandwidth for a Distributed Shared Memory System Using a Gigabit Network Supporting the Virtual Interface Architecture

John Markus Bjørndalen[1], Otto J. Anshus[1], Brian Vinter[1,2], Tore Larsen[1]

Department of Computer Science [1]
University of Tromsø

Department of Mathematics and Computer Science [2]
University of Southern Denmark

Abstract

Previous studies have shown significant performance advantages in using Virtual Interface Architecture (VIA) instead of TCP/IP for handling network communication in the structured distributed shared memory system, PastSet. With the availability of network hardware that supports VIA, we wish to examine whether, and to what extend, an available hardware supported VIA implementation outperforms the software-only implementation for PastSet DSM. To do this, PastSet has been ported to two VIA implementations: M-VIA, which is a software implementation that we use on a 100 Mbit Fast Ethernet, and Giganet cLAN, which uses dedicated VIA hardware. The two implementations are tested, and performance results are compared with the reference TCP/IP implementation on the 100 Mbit Fast Ethernet.

For the experiment setups used, M-VIA latencies are between 1.1 and 2.6 times faster than corresponding latencies using TCP/IP.

For large packets, Giganet cLAN latencies are about 2.7 times faster than corresponding M-VIA latencies. However, for small packets, cLAN latencies are only about 1.04 times faster than corresponding M-VIA latencies, indicating that the current software design and implementation does not fully benefit from the improved performance of Giganet cLAN over Fast Ethernet. Further experiments demonstrate that significantly improved small-packet latencies on cLAN are possible, and may be achieved through a software redesign carefully considering the use of polling versus interrupts.

1 Introduction

The latency and bandwidth performance of a Distributed Shared Memory (DSM) system depends on the performance and interaction of the DSM and the underlying network subsystems. The key challenge [8] is to preserve the performance characteris-

tics of the physical network (bandwidth, latency, QoS) while making effective use of host resources. Network bandwidths and latencies are constantly improving. Unfortunately, applications have not been able to take full advantage of these performance improvements due to the interactions of layers of user and kernel level software. A detailed breakdown of hardware and software costs of remote memory operations is discussed in [3]. The Virtual Interface Architecture (VIA) was developed to significantly reduce the software overhead between a high performance CPU/memory subsystem and a high performance network.

In this paper, we study the latency and bandwidth performance of PastSet DSM using either M-VIA[10], a software VIA implementation for Linux; Giganet cLAN[9], a VIA implementation with hardware VIA support; or TCP/IP. The paper briefly describes the functionality of PastSet, the organization of the implementation, the interactions between PastSet components, and the VIA implementations. Experiment configurations with micro-benchmarks and metrics are described before presenting and analyzing benchmark results.

2 Implementing the PastSet Server and Application Library

PastSet is a structured distributed shared memory system. PastSet memory objects are tuples. Operations exist to create tuples, copy tuples to DSM, and read tuples in DSM. The DSM is structured in that tuples are organized in disjoint elements, and that an ordering of tuples is maintained within each element. Operations exist to create elements and define ordering criteria for each element. A synchronization mechanism is included in the memory model, and a synchronization criterion may be set for each element. Operations are provided to set synchronization criteria.

All PastSet operations are blocking. The PastSet memory model complies with sequential consistency.

For this paper, the PastSet operation `move` is used in determining PastSet latencies. `Move` takes a tuple as parameter and copies the content of the tuple into a specified element in DSM, maintaining tuple order and synchronization criterion. The `move` operation blocks in the sense that it returns only after confirmation has been received from the PastSet server that the operation is completed.

The design, applicability, and performance of PastSet DSM is demonstrated in [1] and [12].

The synchronous nature of PastSet operations implies that each operation request requires a reply message with the result of the operation before the client may continue execution; consequently, two messages are required for each remote operation.

The version of the PastSet server used for the experiments reported on in this paper creates a new thread for each new client connection. Each thread is exclusively responsible for servicing it's associated connection. The threads loop, reading requests, performing operations on behalf of the client and returning results to the caller.

This is a simple approach, with low overhead for a small number of connections. However, the single-thread-per-connection approach is not well suited for multi-threaded clients where several client threads may need to share the same connection. We have developed alternatives to using a sin-

gle thread per connection, but we will not report on these in this paper.

2.1 TCP/IP implementation

When a PastSet operation requests non-local data, the operation and its parameters are sent via a TCP/IP connection to the remote PastSet server, and the caller is blocked awaiting the reply from the PastSet server.

All connections disable the Nagle algorithm to ensure that even small data packets are sent immediately.

2.2 M-VIA implementation

The PastSet server and the application library were implemented using the M-VIA 1.0 [10] implementation of the VIA API. By using the message passing model of VIA, we got a simple port from the TCP/IP implementation. The alternative, using remote DMA, is complicated by the way PastSet operations can manipulate and address PastSet distributed shared memory.

The PastSet server and the application library use blocking calls to M-VIA in order to reduce the processor usage. M-VIA first check to see if the data already has arrived. If not, a block is done.

The 100Mbit network interface cards (NICs) we used do not support the "doorbell" mechanism of the VIA. Instead, this is done in software in M-VIA, making traps to the Linux kernel necessary.

The tuples that are used by the micro benchmarks we use are allocated in parts of the memory that are registered with the M-VIA NICs in order to reduce copying on send and receive. However, M-VIA first copies the data from the NIC to kernel level memory, and then from kernel memory to the user level application memory.

2.3 cLAN implementation

With hardware support, VIA is intended to enable applications to send and receive packets over a Virtual Interface without trapping to the operating system kernel. The kernel is basically only involved in setting up and tearing down connections, and in other book-keeping tasks. In particular, the incoming data is directly written to the user level application memory.

The PastSet server and the application library using hardware supported VIA are otherwise basically identical to the one using M-VIA. In particular, blocking calls are used doing a little spinning to check if data already have arrived before doing the actual blocking.

3 Methodology and Experiment Design

This section describes the hardware and software details of the experiments, how the timing measurements were done, the micro-benchmarks, and the metrics used.

3.1 Hardware and Software

All experiments reported on in this paper were done using two HP LX-Pro Netservers, each having four 166MHz Pentium Pro CPUs. Each computer had 128MB main-memory, and dual peer 33MHz, 32 bit PCI buses. The level 2 cache size is 1MB per processor.

For the experiments, the computers were interconnected using either Giganet cLAN 1.25Gb/s [9] or Trendnet TE100-PCIA

(DEC Tulip 21143 chip set) 100 Mb/s network interface cards (NIC) connected to a hub. Both NICs were on PCI bus no. 0 on each server. In addition, a 100VG 100Mb/s NIC, also on PCI bus no. 0, was used to connect to the local area network of the Department of Computer Science. This network was used to manage the experiments and the servers.

Linux v. 2.2.14 with PastSet functionality added to the kernel was installed on each node participating in the experiments. We used M-VIA version 1.0 with a minor patch to the connection management.

We compiled M-VIA, cLAN, the PastSet Server, the PastSet Kernel, the PastSet Application Library, the Linux operating system, and the benchmarks using egcs 1.1.2.

Default compiler flags were used for M-VIA, cLAN, The PastSet Server, the PastSet Kernel, and the Linux operating system. We used the optimization flags "`-O6 -m486 -mjumps=2 -malignloops=2 -malignfunctions=2`." for the benchmarks and the PastSet library.

Because we experienced problems with M-VIA when using four processors, we redesigned the experiments to use only one processor per server, and we recompiled the Linux kernel to run as a single processor system.

3.2 Micro benchmarks and Metrics

To measure the latency of the PastSet operations, we used several micro benchmarks. In this paper we will only report on the `move` latency, that is, the time to invoke, complete and return from a `move` operation. The `move` operation blocks when waiting for an acknowledgement message from PastSet.

The client process calls `move` operations.

```
for (i = 0; i < 1000; i++) {
   save_timestamp;
   mv();
}
save_timestamp;
```

Figure 1: The `move` latency (Mvlat) benchmark

The client process running the benchmark and the PastSet server are on two different computers.

To determine the effect of using blocking vs. spinning when waiting for data, we used the `vnettest` micro benchmark taken from the M-VIA 1.0 distribution. This micro benchmark is a low level roundtrip ping-pong of data. We modified `vnettest` so we could choose to use either blocking or spinning when waiting for data.

To determine the effect of the underlying network technology, each micro benchmark used TCP/IP, software supported VIA (M-VIA), and hardware supported VIA (Giganet cLan).

Data size for the messages was varied from one to 31KB. The elapsed time for 1000 transmissions is measured for each packet size and then divided by 2000 to get the average latency of a message from address space to address space. We repeated each run of 1000 transmissions five times.

When doing the performance measurements each node supported no other workload except for the operating system and its various artifacts.

All necessary initializations were done before starting time- or cycle measurements.

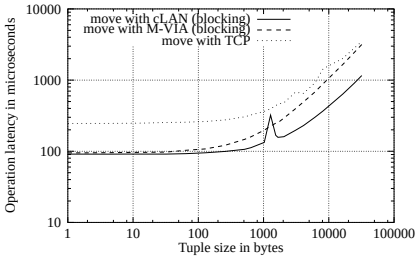
3.3 Time Measurements

The Intel Pentium Pro RDTSC (read time-stamp counter) instruction and the Linux *gettimeofday()* system call were used to determine PastSet operation latencies.

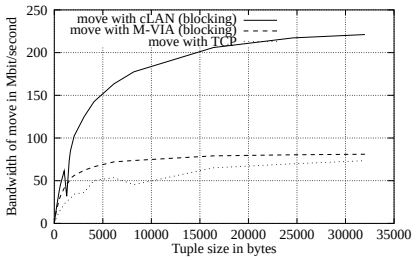
Using RDTSC, as in [6], the cycle count was recorded for every `move` operation. Elapsed time in microseconds was calculated by dividing the registered cycle count by the specified processor frequency of 166 MHz. We did not verify the actual frequency of each individual computer, leaving open the possibility that the computed time may deviate slightly, but consistently, from the performance measured in cycles spent. Care was taken to avoid potential problems with register overwrites and counter overflow.

The *gettimeofday()* system call was used for aggregate measurements over many operation calls. Checks were made to ensure that RDTSC and *gettimeofday()* measurements were consistent.

Cache effects are not eliminated, but measurements are averaged over five runs of one-thousand iterations each, and no other workload is present.



(a) latency



(b) bandwith

Figure 2: The Mvlat benchmark results: operation latency and bandwidth of the PastSet `move` operation

4 Micro-benchmark Results

4.1 Move Latency Results

Figure 2 shows `move` latencies and bandwidth for intra-node communication using TCP/IP, M-VIA and cLan. Tuple sizes are varied from one byte to 31KB. The bandwith is computed from the latency since PastSet requires one operation to complete before the next can be initiated.

For small tuple sizes `move` latency using M-VIA is about 2.6 times faster than TCP/IP. M-VIA latency is about 1.1 times faster than TCP/IP for 31 KB tuples. For one byte tuples, the difference between using M-VIA and TCP/IP is 152 microseconds, while at 31KB the difference is 330 microseconds.

cLan performs slightly better than M-VIA on small tuple sizes (about 1.04 times faster than M-VIA). At larger tuple sizes, the performance of cLan is 2.7 times faster than M-VIA and 3 times faster than TCP/IP.

The observed bandwith using M-VIA is about 70 percent of the potential 100 Mbit/s that the hardware can support, while using cLan we achieve about 18 percent of the potential 1.25 Gbit/s.

Most of the performance improvement of M-VIA over TCP/IP comes from the implementation drawing advantage of local network properties. M-VIA skips ethernet checksums (done in hardware) and handles much of the protocol in the interrupt handler while TCP/IP has to send the data through several layers and compute checksums for ethernet frames, IP headers and TCP packets. M-VIA also uses faster traps to the kernel than TCP/IP.

The performance advantage with cLan is first visible at larger tuple sizes where the higher bandwith (1.25 Gbit vs 100 Mbit) becomes more important. At smaller packet sizes, the benefit from hardware support is masked by the overhead in the management of the blocking calls. As such, the current PastSet implementation does not show much of a performance benefit due to the hardware implementation of VIA.

4.2 Latencies of Polling and Blocking Message Passing

Figure 3 shows message passing latencies of cLan and M-VIA measured with vnettest using spinning (polling) and blocking VIA calls.

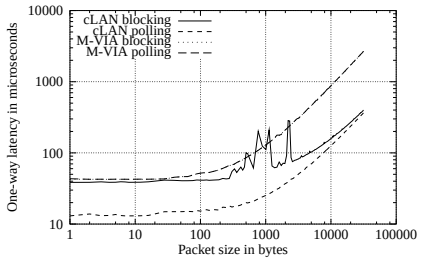


Figure 3: One-way latency over cLAN and M-VIA measured with vnettest

For basic ping-pong communication there is little difference in latency between spinning and blocking communication when using M-VIA, resulting in the graphs overlapping in the figure. This effect comes from the fact that the software M-VIA implementation has to handle interrupts and protocol implementation both for polling and blocking operations.

The extra overhead from the kernel traps (up to 2 ioctl calls per send or receive operation) are overlapped with the pysical transmission of data. This might hurt the performance of M-VIA during high load from multiple clients.

Using polling on cLAN gives a clear advantage, reducing the latency with 20-30 microseconds over all tested packet sizes compared to the blocking version.

4.3 Implications for PastSet implementation

For small tuple sizes the latency of PastSet move operations is about 100 microseconds. Using spinning on cLan achieves a one-way latency improvement of 20-30 microseconds as compared to blocking. This translates into a potential move latency improvement of 40-60 microseconds.

Achieving this requires modifications to the PastSet server and application library. The use of spinning must be carefully applied due to its CPU usage[7].

5 Related work

How to reduce waiting costs in user-level communication has been reported on in several papers, including [7]. This paper describes a mechanism for reducing the cost of waiting for messages in architectures that allow user-level communication libraries.

They document how blocking and spinning can affect the performance, and correlates well with our results.

VIA is currently being introduced for various message passing sub-systems. Systems that are based on the p4[5] communication library are candidates for porting to the VIA API, e.g. the M-VIA team have ported MPICH to use M-VIA instead. Distributed shared memory systems which uses VIA includes the page based HLRC DSM system [11].

Work on building DSM systems on top of user level communication libraries includes the Virtual Memory Mapped Communication system, VMMC [4].

The Orca object based DSM system has an associated communication library, PANDA, which also provides a high performance communication system that runs on Myrinet. PANDA is specifically designed for Orca which is highly dependent on multicast[2].

6 Conclusions

Based on the performance results we can conclude that:

- Hardware supported VIA gives a non-significant improvement in PastSet operation latency over software M-VIA for small tuple sizes. This is because the PastSet operation is blocking, and the cost of blocking is much higher than the advantage of the small protocol overhead in hardware supported VIA
- Hardware supported VIA gives a significant improvement in PastSet operation latency over software M-VIA for large tuple sizes. This is because the hardware supported VIA is a gigabit network versus the megabit network used by the software M-VIA
- Hardware supported VIA benefits significantly from using spinning instead of blocking when waiting for data. This is because the cost of blocking is avoided
- Software supported M-VIA does not benefit significantly from using spinning instead of blocking. This is because the protocol implies several traps to the kernel per data transfer, and this is much more expensive than the benefit coming from spinning
- By using spinning and hardware supported VIA, the PastSet move latency may be cut in half. However, carefully combining spinning and blocking seems to be needed to benefit from gigabit networks with hardware support for VIA while at the same time not using too much processor cycles

7 Acknowledgements

Giganet Inc. kindly provided us with a cLan environment. Ken Arne Jensen and Jon Ivar Kristiansen provided invaluable support under the preparation of this paper.

References

[1] ANSHUS, O. J., AND LARSEN, T. Macroscope: The abstractions of a distributed operating system. *Norsk Informatikk Konferanse* (October 1992).

[2] BHOEDJANG, R., RUHL, T., AND BAL, H. E. Efficient multicast on

myrinet using link-level flow control. In *International Conference on Parallel Processing* (Minneapolis, MN, August 1998), pp. 381–390.

[3] BILAS, A., IFTODE, L., AND SINGH, J. P. Evaluation of hardware support for automatic update in shared virtual memory clusters. In *12th ACM International Conference on Supercomputing* (July 1998).

[4] BLUMRICH, M., LI, K., ALPERT, R., DUBNICKI, C., FELTEN, E., AND SANDBERG, J. A virtual memory mapped network interface for the shrimp multicomputer. In *Proceedings of the 21st Annual Symposium on Computer Architecture* (April 1994), pp. 142–153.

[5] BUTLER, R., AND LUSK, E. User's guide to the p4 parallel programming system. Tech. Rep. ANL-92/17, Argonne National Laboratory, October 1992.

[6] CHEN, J. B., ENDO, Y., CHAN, K., MAZIERES, D., DIAS, A., SELTZER, M., AND SMITH, M. The measured performance of personal computer operating systems. *ACM Transactions on Computer Systems* (February 1996).

[7] DAMIANAKIS, S. N., CHEN, Y., AND FELTEN, E. Reducing waiting costs in user-level communication. In *11th International Parallel Processing Symposium (IPPS '97)* (April 1997).

[8] DRUSCHEL, P., AND PETERSON, L. L. Operating systems and network interfaces. In *Foster, Ian and Kesselman, Carl (Eds.), The Grid: Blueprint for a New Computing Infrastructure* (1999), Morgan Kaufmann.

[9] http://www.giganet.com/.

[10] http://www.nersc.gov/research/ftg/via/.

[11] RANGARAJAN, M., AND IFTODE, L. Software distributed shared memory over virtual interface architecture: Implementation and performance. Tech. Rep. DCS-TR-413, Rutgers University, Department of Computer Science, April 2000. To appear in Proceedings of The Third Extreme Linux Workshop, October 10-12, Atlanta.

[12] VINTER, B. *PastSet a Structured Distributed Shared Memory System*. PhD thesis, Tromsø University, 1999.

A.3 Using Two-, Four- and Eight- Way Multiprocessors as Cluster Components

Communicating Process Architectures – 2001
Alan Chalmers, Majid Mirmehdi and Henk Muller (Eds.)
IOS Press, 2001

Using Two-, Four- and Eight-Way Multiprocessors as Cluster Components

Brian Vinter[1,2], Otto J. Anshus[2], Tore Larsen[2], John M. Bjørndalen[2]
[1] *University of Southern Denmark,* [2] *Tromsø University*

Abstract. This work considers the pros and cons of different size SMPs as nodes in clusters. We investigate the Intel SMP architecture and consider the potential of and some problems with larger node-sizes in clusters of multiprocessors. Six applications that represent different classes of parallel applications are developed in versions that support both shared and distributed memory. Performance measurements are done on three different clusters of multiprocessors, with the purpose of identifying how the number of processors in each SMP node impacts the cluster performance. Our results show that clusters using higher order SMPs do not give a clear performance benefit compared to clusters using two-way SMPs. Off the benchmark-suite of six applications, the performance of two turn out to be independent of node-size, two show an advantage of larger node-sizes, as much as 34% improvement of eight-way nodes over a dual-system, while the remaining two show an advantage of dual-processor nodes as big as 11% over an eight-way cluster.

1 Introduction

There is currently an unprecedented proliferation of low cost, mass market; small-scale shared memory multiprocessors (SMPs). Dual processor PCs are used as engineering workstations, four processor systems are applied as departmental servers, and low-cost six- to eight-way systems using PC-type processors are used for database applications. It remains to be seen if mass-market, small-scale shared-memory multiprocessors will remain commercially and technically viable. Current speculations go both ways, using both technical and commercial arguments. For research on clusters, the mix of shared memory communication within SMPs and network-based communication among SMPs raises a set of interesting questions.

In this work, we characterize the performance of clusters using two-, four- and eight-way SMP nodes respectively. Six CPU-bound applications are implemented using a combination of shared memory and Message Passing Interface, MPI. We describe how the applications are parallelized and measure their relative performance on clusters of two-, four- and eight-way SMPs respectively. The purpose is to investigate and compare performance characteristics of clusters with a the same number of processors using two-way, four-way or eight-way SMP nodes.

1.1 Clusters of Multiprocessors

Loosely coupled clusters of computers are being investigated both in academia and in industry. Currently these ”poor man’s supercomputers” typically uses from eight to 64 CPU’s, while some employ as many as 1000 CPUs[1]. The interest in Commercial Of The Shelf, COTS, clusters as alternative multicomputers has been stimulated by several developments in addition to the obvious price advantages. Most fundamental are the recent advancements

in high-speed, low-cost interconnects, i.e. 100Mb Ethernet at the low end, Myrinet[2] and SCI-bus[3] at the high-end, and, more recently, cLan[4] and Gigabit Ethernet[5] in between.

High-performance inter-process communication mechanisms for loosely coupled clusters, mainly the standardized Message Passing Interface, MPI[6], have come of age in stable and efficient implementations. The source-code compatibility of MPI applications between architectures has made it possible to use shrink-wrap software and public domain applications on clusters, independent of the specific cluster architecture. The latter has promoted clusters heavily in fields of science outside computer science. Computer scientists have been attracted by cluster architectures for a series of reasons, including application, operating-system and distributed shared memory research.

Most research on clusters has been on platforms using a high number of uniprocessor workstations, or very few servers, typically sharing secondary storage. Shared memory multiprocessors (SMPs), are currently at price points that make them very attractive as nodes in clusters, these are often refered to as CLUMPS, Cluster of Multiprocessors[7]. Dual processor workstations cost only marginally more than similar uniprocessor systems plus the cost of the extra CPU. Each dual processor workstation is typically cheaper than two similarly equipped uniprocessors, with the same amount of memory and disk capacity. Entry-level servers using four CPUs also add other attractive features like redundant power supplies and RAID-disks. Eight-way SMPs currently carry a price premium that is explained partly by expensive vendor-specific implementations and partly by lack of competition in the marketplace.

One interesting aspect of using multiprocessor cluster nodes is to determine how the traditional multicomputer programming models perform when running on clusters of SMPs rather than on clusters of uniprocessors. Traditionally shared memory application programming interfaces (API), and distributed memory APIs has been quite different in both syntax and semantics. Shared memory APIs, such as Sys V Interprocess Communication (Sys V IPC) and APIs found in thread packages such as POSIX threads (Pthreads), are kept simple to support both flexibility and good performance for most common cases. Distributed APIs, such as MPI and Remote Procedure Calls (RPC)[8], have more complex and powerful functionalities freeing the programmer from building his own operations using simpler ones. The question remains how the two classes of programming API will work together.

From here we go on to briefly investigate the Intel SMP architecture in section 2 and the MPI API in section 3, section 4 motivate and describe the applications we have chosen to test the impact of node-size on the scalability of the application. The results from our experiments are discussed in section 5. We take a brief look at related work in section 6 and finally we draw our conclusions in section 7.

2 SMP node architecture

The Intel multiprocessor architecture is one of the most widely available multiprocessors, and most of the control hardware that is needed to implement symmetrical-multi-processing is placed on the CPU itself, with the result that a dual-processor Intel machine is only marginally more expensive than the cost of a uni-processor and the additional CPU. As one can imagine this makes these machines attractive from a cluster point-of-view since a 32 CPU cluster made up of 16 dual-processor machines is notably less expensive than a similar cluster made from 32 standard PCs. Beyond two CPUs the Intel Standard High Volume, SHV, program has made four- and eight-way SMP machines fairly cheap too. However, these are marketed only for the server segment of the market and thus never becomes truly commodity and as a consequence they don't become quite as cheap, on a per CPU basis, as the

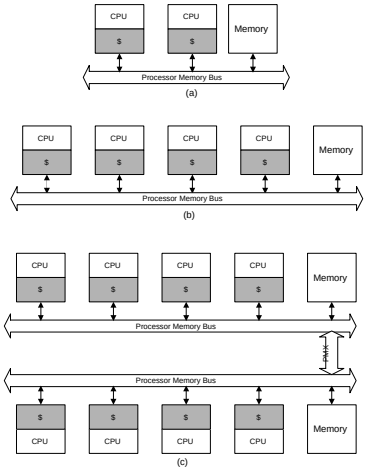


Figure 1: The Intel MP architecture in (a) two, (b) four and (c) eight CPU versions

dual-processor setup.

Intel's SMP architecture is a shared bus architecture, which is one reason why the machines are fairly cheap, but this is also the reason why the machines only scale to very few processors in each machine. Each CPU has two levels of cache and both levels are kept coherent in compliance with the MESI protocol. The Intel MP Specification as of revision 1.4[9] supports at most four CPU on the same bus. Thus to scale to eight CPUs, a system needs to use two busses and add extra hardware support to cache coherence of the two busses.

The purpose of using fewer nodes with more CPUs is to lower the average communication time between processors and limit the load on the cluster interconnect. These advantages should result in a better CPU utilization when running parallel applications on the cluster. On the downside, more processors on the same bus also means less available bandwidth per processor. Table 3 in section 5 shows the consequences of this for the three clusters used in this investigation.

If clusters of multiprocessors are intended to be 'poor-mans-supercomputers', a more serious problem is the premium cost that is associated with higher degree SMP servers. The higher cost of larger servers may be attributed to several issues; first and foremost is the lower volume in sales of these systems, which automatically increases the price per unit. In addition the higher degree servers require more hardware, larger casing, etc.

The fairly high price of four and eight way Intel based machines has resulted in a scenario where these architectures are only interesting to the segments of the market that heavily depend on the higher degree of multi-processing, e.g. the server market. Thus, four and eight CPU nodes are only available as servers with the associated high performance I/O subsystems and redundant power supplies, all of which add to the price of the machines. In addition the available four and eight way servers all seek to compensate for the limited memory bandwidth by using processors with large high-performance cache systems (Intel's Xenon class processors). The result is that it is very hard to compare the price of clusters based on different SMP sizes without ending up comparing apples to oranges. Figure 2 show the prices of a number of clusters all with 32 CPUs and 4GB memory, based on a variety of node types. While none of the two CPU per node clusters compares directly in configuration to the four and eight way versions, it is obvious that there is a significant premium associated

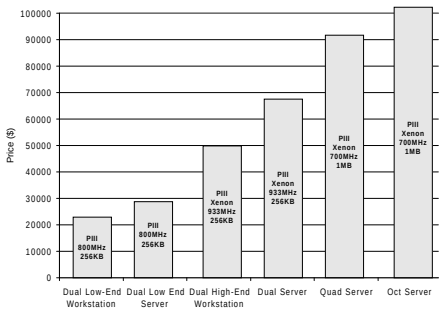


Figure 2: Price of a 32 CPU cluster using various machines as nodes

with using higher degree SMPs as cluster nodes.

It is clear that a 32 CPU cluster with 4GB RAM cluster made up of four eight-way machines is close to five times more costly than the cheapest dual-processor solution (700MHz / 1MB PIII Xenon vs. 800MHz / 256KB PIII) and 50% more expensive than the most expensive dual solution (700MHz / 1MB PIII Xenon vs. 933MHz / 256KB PIII Xenon). All the configurations in figure 2 are machines from the same brand-name producer. If the cheapest dual processor configuration were made up from home-assembled nodes, the price would be approximately 50% lower still.

The prices in figure 2 make it evident that for CLUMPS of node-size larger than two, the performance of these systems should out-perform the dual-systems in a ratio similar to the 50%-500% difference in price.

3 Message Passing Interface

The Message Passing Interface, MPI[6], is a controlled API standard for programming a wide array of parallel architectures. Though MPI was originally intended for classic distributed memory architectures, it is used on various architectures from networks of PCs via large shared memory systems, such as the SGI Origin 2000, to massive parallel architectures, such as Cray T3D and Intel paragon. The complete MPI API offers 186 operations, which makes this is a rather complex programming API. However, most MPI applications use only six to ten of the available operations.

MPI is intended for Single Program Multiple Data, SPMD programming paradigm, e.g. all nodes run the same application-code. The SPMD paradigm is efficient and easy to use for a large set of scientific applications with a regular execution pattern. Other, less regular, applications are far less suited to the MPI programming model and implementation in MPI is tedious.

MPI's point-to-point communication comes in four shapes, standard, ready, synchronous and buffered. A standard-send operation does not return until the send buffer has been copied, either to another buffer below the MPI layer or to the network interface, (NIC). The ready-send operations are not initiated until the addressed process has initiated a corresponding receive. The synchronous call sends the message, but does not return until the receiver has initiated a read of the message. The fourth model, the buffered send, copies the message to a buffer in the MPI-layer and then allows the application to continue. Each of the four models also comes in asynchronous, in MPI called non-blocking, modes. The non-blocking calls return immediately, and it is the programmer's responsibility to check that the send has com-

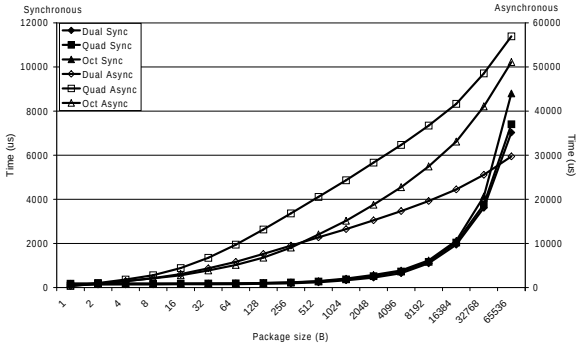


Figure 3: Point-to-point latency of synchronous and asynchronous messages in us

pleted before overwriting the buffer. Likewise a non-blocking receive exist, which returns immediately and the programmer needs to ensure that the receive operation has finished before using the data.

MPI supports both group broadcasting and global reductions. Being SPMD, all nodes have to meet at a group operation, i.e. a broadcast operation blocks until all the processes in the context have issued the broadcast operation. This is important because it turns all group-operations into synchronization points in the application. The MPI API also supports scatter-gather for easy exchange of large data-structures and virtual architecture topologies, which allow source-code compatible MPI applications to execute efficiently across different platforms.

3.1 Micro-benchmarks

Since the performance of the interconnect and the MPI layer dictates the scalability of the applications we will investigate in this work, we first identify the basic cost of synchronous (blocking) and asynchronous (non-blocking) point to point operations and of the inherently synchronous group operations, broadcast and global reductions. The MPI implementation we use here is the LAM-MPI distribution[10], version 6.5, which is commonly used in cluster-computing.

Figure 3[1] shows the latency of both synchronous and asynchronous point-to-point communication. In the applications we seek to use asynchronous messages as much as possible since these provides latency-hiding, in the cases where the application is able to do other work while the communication takes place.

It is obvious from figure 3 that the synchronous operation performance is dictated by the interconnect, while the performance of the asynchronous operations differ much more. The specifications of the clusters are listed in table 3, and it is evident that the performance of the asynchronous operations is heavily dependent on the processor speed. This is even clearer if we look at some of the raw numbers, listed in table 1. The specifications for the clusters are found in table 3 in section 5.

The broadcast latency of the dual-processor cluster with varying number of nodes and varying package size is shown in figure 4. The graph show that the latency of broadcasts depend heavily on both package size and the number of nodes. It is quite interesting that the

[1]Notice different scales for synchronous and asynchronous messages.

	Dual	Quad	Oct
1B	356	436	396
1KB	13236	24333	15136
64KB	29728	56949	51113

Table 1: Asynchronous message-latency in μs, on three different platforms

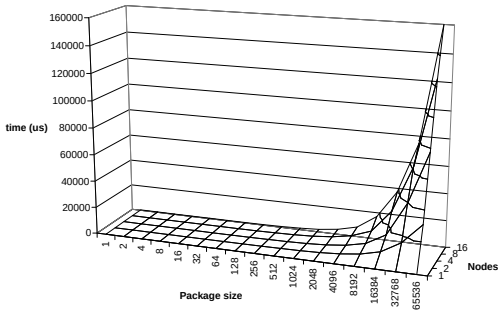


Figure 4: Broadcast latency of the dual-CPU cluster.

number of nodes has such a significant impact on performance, given that the interconnect[2] supports broadcast at the media layer.

Table 2 lists the broadcast latency for all three clusters in figure 3, and shows that while the eight-way cluster, with its faster CPUs, is faster than the four-way cluster at asynchronous operations, the four-way cluster is faster at synchronous operations. This is also visible in figure 3. Whether this is due to differences in the node architecture or due to the different 100Mb networks is unknown, though the different network-technology is the most likely reason.

Global reductions performance is very similar to the broadcast performance, though the latency is slightly higher. We chose not to present the graph and numbers here since the pattern is almost identical to that of the broadcast benchmark.

4 Applications

All the applications we present here are chosen to represent different classes of processor-bound hard-to-compute problems. The problems are chosen because they are well known and thus should allow the reader to focus on the performance characteristics that we observe, similar or close to identical performance behavior should be observed on other, more realistic, applications in the same classes.

The included applications are, an embarrassingly parallel Monte Carlo simulation; a global optimization problem, the Traveling Salesman Problem; a grid application, Successive Over Relaxation; a simple linear algebra problem, matrix multiplication; an automaton model, WATOR; and finally the classic super-computer benchmark, Gaussian elimination.

[2] Switched Ethernet

Nodes	*Dual*			*Quad*			*Oct*		
	1	**1k**	**64k**	**1**	**1k**	**64k**	**1**	**1k**	**64k**
1	0,5	0,5	0,5	1,2	1,2	1,2	1,0	1,0	1,0
2	151	676	30497	205	798	40987	202	835	44551
4	213	988	77504	280	1202	97817	260	1429	102916
8	302	1589	125511	546	1767	156712			
16	377	2028	159971						

Table 2: Broadcast-latency in μs with varying nodesize, number of nodes and packagesize

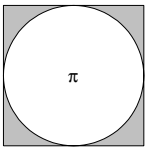


Figure 5: Monte Carlo Estimation of π

4.1 Monte Carlo Estimate of π

Monte Carlo methods are a widely used group of numerical methods, which involve sampling from random numbers. The Monte Carlo method can be used to solve otherwise intractable problems. Since Monte Carlo applications are based on random events a large number of such events must typically be processed to ensure a realistic result. The Monte Carlo method we use here, Monte Carlo π, is utterly uninteresting in and of itself but exhibits the same behavior as real world Monte Carlo applications as well as the related Las Vegas methods and random walks, all of which are frequently used in physics, biology and finance.

The Monte Carlo method of estimating π uses a unitary circle, inscribed inside a square. The application repeatedly picks a random point within the square, the fraction of the points that are inside the unitary circle then represents $\pi/4$, this is sketched in figure 5.

The Monte Carlo approach depends heavily on a good random number generator and a large number of guesses, still it will never provide a particular good estimate of π. The sequential solution picks all the numbers and estimates π using a single processor. The parallel solution distributes the required samples amongst the nodes. Each node spawns one thread per processor and each thread test a fraction of the tests that the node is responsible for.

The Monte Carlo estimation of π is embarrassingly parallel, and is included as a sanity-check, if we do not get perfect speedup on this application there is an error somewhere[3]. Since the problem is embarrassingly parallel it only include one protected function call amongst the threads in a single node to collect the total circle hits on the node and a single synchronous MPI operation to collect the global number of circle hits.

4.2 Traveling Salesman Problem

The Traveling Salesman Problem, TSP, is a classic representative for the class of global optimization problems. The TSP solution we use in this work is a depth-first branch-and-bound algorithm which makes the parallel version different from the other applications we use by the fact that a static division of the work would result in a highly unbalanced execution.

[3]The Monte Carlo π test actually detected various problem during the testing

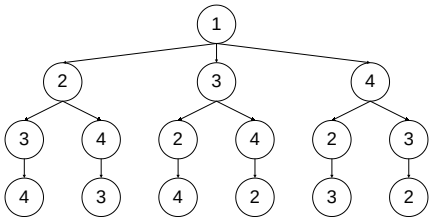


Figure 6: A complete TSP search-tree for a four city problem

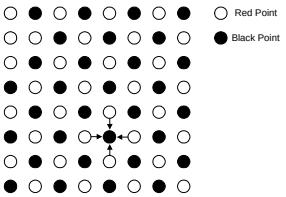


Figure 7: Red-Black Successive Over-Relaxation

Thus the parallel TSP is implemented as a bag-of-tasks application, a paradigm that does not come natural to the SPMD programming paradigm that MPI is designed around.

The parallel TSP is implemented as a global master process and set of worker-threads on each node, each thread communicates with the master to retrieve jobs and submit results. A job is represented as a set of cities that have already been placed and a set that need to be placed, i.e. a sub-tree. Each job that is sent from the master has the length of the shortest known route piggy-backed and each node maintains one shared instance of the bound value. Since the application is so highly unbalanced in the workload this application use synchronous messages for communication. A scheme for applying asynchronous messages could be developed, but two things talk against it, first of all an asynchronous scheme would place an extra job at each node, which is likely to increase the load-unbalance. The second reason why asynchronous messages are unfavorable is that the messages a quite small and the additional overhead of asynchronous messages, as show in figure 3, are very likely to be bigger that the potential gain from asynchronicity.

4.3 SOR

Successive Over-Relaxation, SOR, is a frequently used technique for solving very large systems of partial differential equations by successive approximations. The general idea is to approximate each element in a matrix to its neighbors until the sum of all changes within one iteration converges below a given value.

The Red-Black checker pointing version of SOR, shown in figure 7, returns identical results for the same system of equations; independent of the actual computing environment, while at the same time providing sufficient parallelism that real speedup can be achieved. With Red-Black checker pointing, the equation system is divided into alternating red and black points in a chess-board fashion. Updating a red point depends only on black neighboring points and vice versa. Using this, an algorithm is derived where each worker-process updates all its red points and then exchanges red border point values with its neighbors. Each worker then updates its black points and repeats the communication for the black points. At the end of each iteration the global change in the system is calculated and the process

```
do {
  if THREAD_ID=0 then {
    ASYNC_SEND(top and bottom black rows)
    ASYNC_RECV(top and bottom neighbor black rows)
  }
  UPDATE(red points)
  if THREAD_ID=0 then {
    ASYNC_SEND_FINISH
    ASYNC_RECV_FINISH
    UPDATE(top and bottom red points)
  }
  thread_barrier() //All threads must have finished red-update
  if THREAD_ID=0 then {
    ASYNC_SEND(top and bottom red rows)
    ASYNC_RECV(top and bottom neighbor red rows)
  }
  UPDATE(black points)
  if THREAD_ID=0 then {
    ASYNC_SEND_FINISH
    ASYNC_RECV_FINISH
    UPDATE(top and bottom black points)
  }
  thread_sum(change) //Collect the change from all threads
  if THREAD_ID=0 then MPI_SUM(change) //Find the global change
  thread_update(change) //Tell all threads the global change
} while ( change>epsilon )
```

Figure 8: CLUMPS version of Red-Black Successive Over-Relaxation

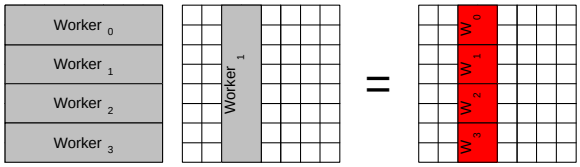


Figure 9: Matrix tiling

continues until the change in the system is below a given threshold.

The CLUMPS version divides the system into a set of static blocks which are divided amongst the nodes, each node subsequently divides its dataset between the CPUs on the node. Since LAM-MPI is not thread-safe one thread is responsible for communication and updating the upper-most and lower-most data-rows in the dataset, in addition to a portion of the dataset similar to the other threads, this thread also represents the node in the global reduction to find the total change in the system, see pseudo-code in figure 8. An iteration then includes two asynchronous send operations and two asynchronous receives as well as one synchronous global reduction operation.

4.4 Matrix Multiplication

Matrix multiplication is frequently used in scientific applications. Although several concurrent and parallel algorithms exist that requires extensive communication during the calculation, an alternative approach is to coarsely distribute the matrixes amongst the nodes and broadcast one matrix to all nodes one block at a time as shown figure 9. This approach is

```
in parallel C = 0
for i = 0;N-1{
  broadcast A*i within rows
  broadcast Bi* within columns
  in parallel C = C + (A*i)(Bi*)
}
```

Figure 10: PBLAS Matrix multiplication

similar to the one used in PBLAS [11]. The generic solution in [11] is based on the following matrix decomposition:

$$
\begin{aligned}
C = & \begin{pmatrix} A_{00} \\ A_{10} \\ \vdots \\ A_{(N-1)0} \end{pmatrix} \begin{pmatrix} B_{00} B_{01} \ldots B_{0(N-1)} \end{pmatrix} \\
& + \begin{pmatrix} A_{01} \\ A_{11} \\ \vdots \\ A_{(N-1)1} \end{pmatrix} \begin{pmatrix} B_{10} B_{11} \ldots B_{1(N-1)} \end{pmatrix} \\
& + \quad \ldots \\
& + \begin{pmatrix} A_{0(N-1)} \\ A_{1(N-1)} \\ \vdots \\ A_{(N-1)(N-1)} \end{pmatrix} \begin{pmatrix} B_{(N-1)0} B_{(N-1)1} \ldots B_{(N-1)(N-1)} \end{pmatrix}
\end{aligned}
\tag{1}
$$

We have chosen a less generic, but more efficient algorithm where we maintain both A and B in distributed state and thus only need to broadcast one matrix amongst the other processes. In addition the matrix is not broadcast column by column, but the entire block at once. Each node uses an efficient sub-blocked matrix-multiplication to take advantage of cache-memory. The MPI solution results in one synchronous broadcast per node in the overall execution. The broadcasts are rather large however and will stress the MPI layer significantly.

4.5 WATOR

The WAter TORus world, WATOR, is a classic discrete event simulation[12], and while it provides valuable information in itself we chose to introduce it here for reasons similar to the Monte Carlo π example, namely that it is simple and easy to understand, while still being typical for the class of applications that it represents. Discrete event simulations are widely used to model everything from digital systems to financial forecasts, logistics and traffic simulations.

WATOR is the simulation of a very special world, first of all the planet is not a sphere as most planets we know, rather the planet is shaped as a doughnut, or a torus, which greatly simplifies mapping the world into discrete blocks. As the whole surface of WATOR is covered with water there are only two types of life, which we are interested in, fish and sharks.

Fish are simple organisms that move around at random, and at some point when a fish comes of age is will have two children and die itself. A fish can move into any of its neighboring eight squares, given that the square is empty, if all eight neighboring squares are occupied

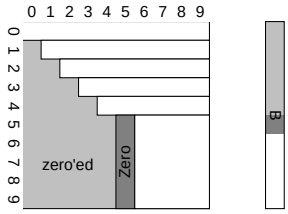


Figure 11: Gaussian Elimination

the fish remains in place. At any discrete time step a fish becomes one time-unit older, and once it has come of age it will split into two fish, one of which will go to a neighbor cell while the other stays in the cell where it was born, both new fish is age zero.

Sharks are similar simple creatures, however sharks also need to eat fish in order to survive. At each time-step a shark moves to a random neighboring cell which holds a fish, if there are no fish which neighbors the shark it moves to a random neighboring cell and increases its hunger index, if the hunger index reaches a starvation limit the shark dies, when the shark eats it resets the hunger index to zero. Similar to fish, sharks will at some point get old enough to breed and once this age is reached the shark is replaces by two new sharks, both with age and hunger index zero.

Thus the simulation of one time-step consists of two steps, first all fish are moved, then the sharks, each step issues four asynchronous MPI operations, two send and tworeceive. The randomness of the application allows us to ignore further synchronization issues.

4.6 Gaussian Elimination

Gaussian elimination seek to solve large systems of linear equations, by performing a LU (Lower Upper) restructuring of the system matrix and back-substitution of the parameters that were used for achieving the LU version of the system-matrix. Figure 11 show how each line is transformed into its LU shape by zeroing the left-most entry and dividing all lower rows by the row that was finished. There is a twist however, since the limited numerical representation in the processor can destroy the data if the divisor is close to zero. Because of this we need to perform a partial pivot on the remaining rows for each iteration such that the largest possible divisor is used.

To perform the LU decomposition in parallel we basically have two choices, either divide the matrix by rows or by columns. If we divide the system by columns then one processor alone can decide the pivot row and broadcast the pivot to all processors. After this an all-to-all communication phase is needed to create a copy of the active row at all nodes. If the matrix is distributed row-wise then an actual election of the best pivot value is needed. After the election the process that won the election broadcasts its row.

We chose the row-wise solution in order to reduce the large messages to a one-to-all broadcast. Including the partial pivot this means that each iteration use two synchronous group operations, one election of the best divisor and a broadcast of the 'winning' row.

5 Performance

To compare the performance of 2, 4, and 8 way SMPs, we have run the application from section 4 on three different clusters that all have a total of 32 CPUs, distributed over six-

	Dual	**Quad**	**Oct**
CPUs per node	2	4	8
Bus width	8B	8B	8B
Mem. Busses	1	1	2
CPU type	PIII	PPro	PPro
2' level cache	512KB	512KB	1MB
CPU speed	450MHz	166MHz	200MHz
Memory per node	256MB	128MB	2048MB
Bus Speed	100MHz	66MHz	66MHz
Eff. Bandwidth	0.89B/cycle	0.80B/cycle	See figure 12
Interconnect	Fast Ether	Fast Ether	100VG

Table 3: Architectural specifications of the three clusters used for the experiments

teen, eight and four nodes respectively. 100Mb networks interconnect the nodes in all three clusters.

Since our primary concern in this work is scalability we present performance as speedup, which will be presented as CPU utilization. The use of speedup as a measure of success is heavily debated but while the information that can be extracted from speedup numbers may be limited, it fits our purpose well. First of all, since the cluster-nodes differ in more ways than simply SMP-size other measures than relative scalability are unachievable. Secondly, since scalability is the primary topic of investigation in our work, achieved speedup is the parameter we are interested in.

5.1 Experiment Design

All experiments reported on in this paper were done using three clusters. The first cluster, *Dual*, is comprised of sixteen "no name" PCs, all equipped with two 450MHz Pentium III processors and the Intel BX chipset. Each PC has 256MB main memory, and a single 33MHz, 32-bit PCI bus. The second level cache-size is 512KB per processor.

The second cluster, *Quad*, is comprised of four HP LX-Pro Net-servers, each with four 166MHz Pentium Pro processors. Each server has 128MB main memory, and dual peer 33MHz, 32 bit PCI buses. The level 2 cache size is 512KB per processor. These machines belong to the first generation of Intel's SHV machines.

The third cluster, *Oct*, is comprised of four HP Netserver LXr Pro 8 servers, that all have eight 200MHz Pentium Pro processors. Each server has 2GB RAM and dual peer 33MHz, 32 bit PCI buses. The 2. level cache size is 1MB.

The *Dual* and *Quad* clusters used the Trendnet TE100-PCIA (DEC[4] Tulip 21143 chip set) 100 Mb/s network interface cards (NIC) connected to a switch. The network interface cards (NIC) were on PCI bus no. 0 on the *Quad* cluster. The *Oct* cluster is interconnected with a HP 100VG 100Mb/s network, also on PCI bus no. 0. The 100VG network is connected via a 100VG hub. The complete specifications for all three clusters are summarized in table 3.

The operating system on all clusters is Linux v. 2.2.14, and LAM-MPI 6.5 is used as the MPI layer. All the applications were compiled with gcc 2.96.2 and optimization-flag '-O3'. Time is measured with the Linux `gettimeofday` system call.

In order to test the performance under varying problem sizes, the applications have been run with three different datasets; tiny, medium and large. The datasets corresponds to 10, 100

[4]Now Intel, but most literature references work with the original DEC chipset

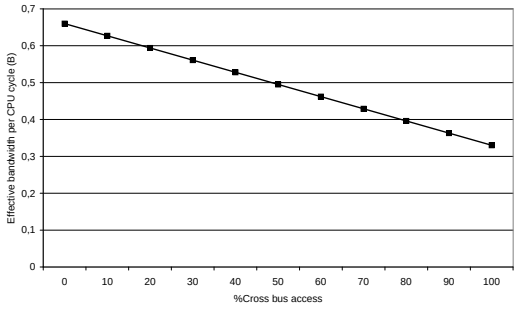


Figure 12: Effective available bandwidth per CPU per cycle in an eight CPU machine as a function of cross-bus accesses

	MC-π	**Mat. Mul.**	**SOR**	**TSP**	**WATOR**	**Gauss**
Tiny	25E6 darts	640x640	1000	14 cities	250x250, 100 itt	900
Medium	25E7 darts	1536x1536	3000	16 cities	800x800, 100 itt	1900
Large	25E8 darts	3072x3072	5000	17 cities	2600x2600, 100 itt	4000

Table 4: Applications and problem sizes used to measure the performance of 2-4-8 way SMPs

and 1000[5] seconds sequential runtime on the fastest CPUs, the 450MHz PIII processors in the dual-CPU cluster. Table 4 sums up the applications and the dataset parameters that have been chosen for each.

5.2 Monte Carlo π

The performance of the Monte Carlo application in figure 13 does not provide much information to us. All systems achieve close to perfect speedup even with the tiny problem-size. This was expected and the result validates that all the processors do in fact provide identical performance. The conclusion one may draw from the experiment is that if an application is embarrassingly parallel, there is no reason to invest in expensive SMP hardware; this conclusion is hardly surprising.

In order to achieve the expected result however, we had to replace the `libc` random function by a custom pseudo-random generator, since the use of the library version eliminated all parallelism within one node. In fact even a single threaded version is significantly slower than a standard sequential version, due to mutual-exclusion protection of the random function.

5.3 TSP

The TSP application was rather hard to fit into the timing parameters and required manual layout of the sequence of cities in the list to fit the desired runtime. While this dictates the execution time of the application it does not influence the parallelism in the execution.

Since TSP is both threaded and tasked, there are two features in the implementation that work against each other; on one hand more CPUs mean that the bound variable on any node is updated faster and thus reduces the work that need to be executed on that node. However,

[5]The large problem for SOR is not really 1000 seconds, but the largest we can do in 128MB

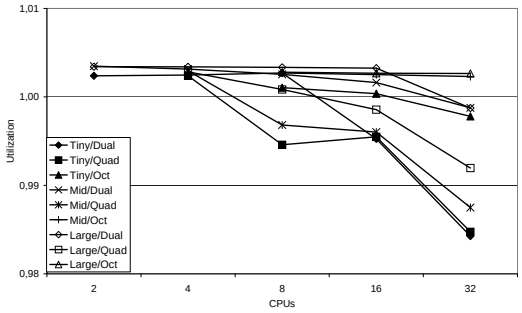


Figure 13: Performance of the Embarrassingly Parallel Application Monte Carlo π

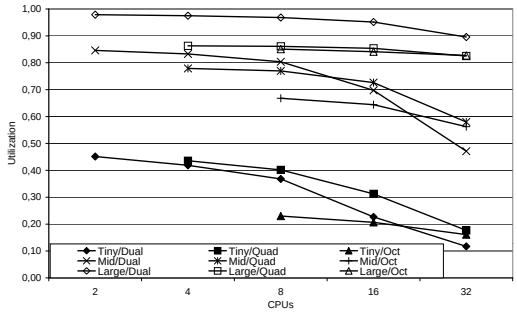


Figure 14: Performance of the Traveling Salesman Problem

more CPUs also increase the probability for contention on access to the MPI layer. The impact of the two elements is clearly seen in figure 14. The tiny portion benefits from faster updates of the bound variable, e.g. more CPUs per node, while the larger problems benefit more from lower contention on the network interface and favor the dual-processor cluster over the four- and eight-way systems.

5.4 SOR

The four-way servers are quite old and as a result have very little memory, which limits the size of the SOR application to a maximum of 5000x5000, floats in the large dataset. This results in a sequential execution time of only 250 seconds on the dual nodes, far from the target of 1000 seconds. Thus, this is not really a huge data-set per se.

SOR performance is quite similar on all three architectures as shown in figure 15. The dual-processor cluster has similar performance to the others up until 8 nodes while at 16 nodes it is 15% slower with the tiny test-set and 5% slower on the largest problem. The difference in effective bandwidth does not shine through because SOR is so regular that out-of-order execution, memory perfecting and cache locality is quite effective at hiding the memory latency. As figure 4 show the number of nodes is a significant parameter in the time a global reduction takes and this becomes obvious as the dual cluster falls behind the other two. The cross bus exchange of data in the eight-way nodes is visible too. The result is that the eight-way cluster is slightly slower than the four-way although the measurements in table

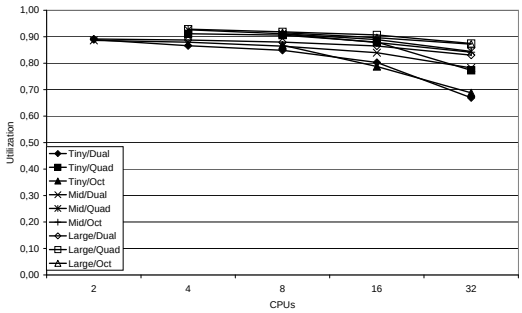


Figure 15: Performance of Successive Over Relaxation

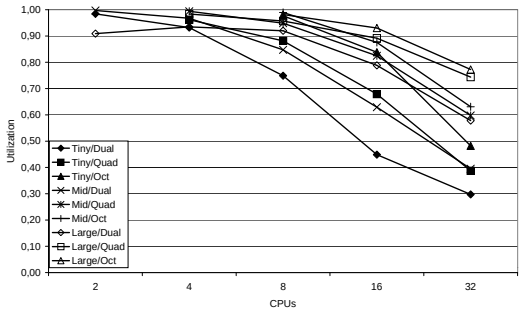


Figure 16: Performance of Matrix Multiplication

2 would indicate that it should be vice versa.

5.5 Matrix Multiplication

Since the matrix-multiplication algorithm we use is based on large broadcasts, it is not surprising that we see a clear advantage of larger node-sizes, that is, fewer nodes in the system. The pattern is clear over all problem-sizes and numbers of CPUs in the cluster. The result is that using 32 CPUs on the huge problem-set, as shown in figure 16, the eight-way CLUMPS is 34% faster than the two-way system.

The advantage of eight-way nodes over four-way nodes is not as large as table 2 would indicate. This is because once the data-block is received on the node, it still need to be communicated across the memory-bus connection, in effect making the cross bus rate close to 50%, which corresponds to an effective bandwidth of 0.5B/cycle. The success of the cache-efficient block-multiplication algorithm is evident in that the higher effective bandwidth of the two-way system does not outweigh the higher cost of broadcasting with more nodes.

5.6 WATOR

The WATOR application is the application that provides the highest level of asynchrony, since this application only uses non-blocking communication. The consequence of the asynchrony is that with the medium and large datasets the communication cost can be hidden quite well

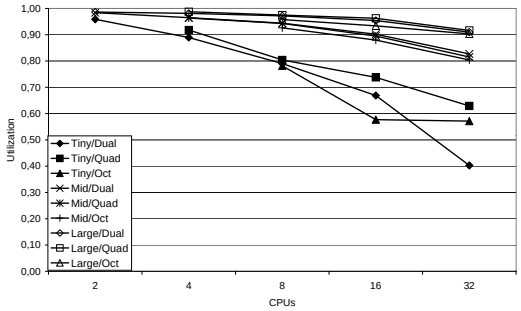


Figure 17: Performance of the WAter TORus simulation

and we achieve close to linear speedup and similar performance for all three systems. The small problem-set is not enough to hide the latency when we use more than a few nodes.

The performance of WATOR on our CLUMPS in figure 17 includes an interesting point on the choice of nodes size, since the four-way system consistently outperforms the eight-way CLUMP. While the difference is small, ranging from 10% on the small data-set to only 1% on the large set, it is consistent and except on the large data-set, it is also statistically significant. This behavior is the result of data-pushing vs. -pulling. Between nodes data is exchanged via MPI, which is slow but executes asynchronously, i.e. when the thread needs the data, it is likely to be ready. Within nodes the exchange scenario is vice versa; here the threads wait for the neighboring thread to finish, and only after the data is ready can the receiver start reading the data. In the eight-way system this means crossing the inter bus connection for two of the processors. While the waiting period is likely to be zero,[6] the data transfer still cannot begin before the data is needed.

A similar pattern is also seen in the SOR application, which is similar to the WATOR access pattern. However the global reduction in the SOR code favors the eight-way system and returns in less clear results.

5.7 Gausssian Elimination

Overall the Gaussian elimination provides much worse speedup than the other applications. This is no surprise since the parallel version is based on two synchronous MPI operations per iteration. It is interesting to see though that with 32 CPUs and solving the largest system, the higher available bandwidth provides better performance for the dual-processor based CLUMPS, which is 11% faster than the eight-way system and 33% faster then the four-way. Since there is little cross-bus communication the eight-way system does not suffer from lowered bandwidth and instead benefits from fewer nodes when performing the rather costly synchronous operations.

The implementation of the Gaussian elimination is simple and the performance could be improved by adding an extra thread, which performs the MPI operations while the other threads perform the calculations. This kind of hand-coded asynchrony would only increase the advantage of the two-processor CLUMPS since this is the architecture that suffers the most under the high cost of group operations.

[6] Almost certainly in fact, however the race-condition is handled in the code

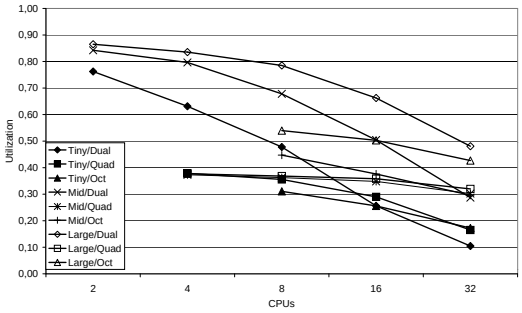


Figure 18: Performance of Gaussian Elimination

	Favors small nodes	**Favors large nodes**	**Indifferent**
MC-π			X
Mat. Mul.		X	
SOR		X	
TSP	X		
WATOR			X
Gauss	X		

Table 5: Summary of the applications and their favored node-size with 32 CPUs and the large data-set

5.8 Performance Summary

There is no one truth as to whether CLUMPS should use small or large nodes, as table 5 shows. It is clear, however, that there are applications that clearly favor larger nodes, such as the chosen matrix multiplication algorithm, and some that favor smaller nodes, most significantly the Gaussian elimination. Since the matrix multiplication is the strongest argument for large nodes, and given that the chosen algorithm is quite naive, the arguments for large nodes are quite weak.

6 Related Work

CLUMP architectures have been around for a while, but few in-depth investigations have been published.

In [13] Takahashi et al. design and implement a special CLUMP version of MPI, MPICH-PM/CLUMP. The MPI version uses Myrinet [2] and implements a full zero-copy protocol. The work compares the performance of the NAS benchmark suite on a uni-processor and a dual-processor cluster and concludes that the CLUMP machine is as much as 30% slower than, and never better than, the uni-processor version. Since the goal of this work is to build an efficient SMP MPI implementation, the dual-cluster is treated as uni-processors, e.g. the applications them-selves are not threaded.

Another angle is found in [14], where the authors investigate the performance of sparse matrix systems on shared memory, distributed memory and hybrid architectures. Here the authors find a small advantage of mixing shared memory and message passing programming.

More recently [15] compares Discrete Element Modelling code on a CLUMP using MPI

and OpenMP and concludes that MPI is favorable to OpenMP on a CLUMPS, but not on a single SMP node.

7 Conclusions

Comparing CLUMPS architectures is not straightforward, since we cannot compare systems that differ only in the number of CPUs per node. However, it is still evident that increased node-size comes at a price, and that this price is closer to 500% than to 50% extra cost per CPU, when comparing dual and eight-way nodes.

Initial micro-benchmarks show a huge difference between synchronous and asynchronous point-to-point operations. More surprisingly, a significant dependence on the number of nodes in the system is seen in group-operations such as broadcasts. This is interesting given that the nodes are interconnected by a media that supports broadcasts.

Our experiments show that the benefit of higher order nodes depends heavily on the application nature; embarrassingly parallel applications like the Monte Carlo π and highly asynchronous applications like WATOR shows no performance difference at all, while applications that depend on synchronous MPI operations such as SOR and Matrix Multiplication show an advantage of more CPUs per node, i.e. fewer nodes in the system. A global optimization problem, TSP, shows an expected advantage of larger node-size with small problems since these are not bandwidth dependent and more threads may lower the total work that has been performed. With large problem-sets the dual-processor system outperforms the four- and eight-way systems because of the lower contention on the network interface. The most significant advantage of smaller node-size is in applications that require more processor-memory bandwidth, such as Gaussian elimination, where we see as much as a 33% advantage of two-way nodes over four-way, and 11% over the eight-way system, with large problems. This is in spite of the fact that Gaussian elimination is based purely on synchronous MPI operations. Small equation-systems show an advantage of fewer nodes, with a 65% performance advantage of the eight-way system over the cluster based on dual-processors.

If we focus on the large problem sets, which after all are the main target for high-performance clusters, the results show two applications that favor large nodes, two that favors small nodes and two that are neutral towards node-size. Considering the price-premium on larger CLUMPS the straightforward conclusion is that larger nodes are not worthwhile, though there are advantages to be had on certain applications and for small problem-sets.

Acknowledgements

The authors would like to thank Dr. Joan Boyar for valuable input for the final version of this paper.

References

[1] Genetic-Programming.com. 1,000-pentium beowulf-style cluster computer for genetic programming. *http://www.genetic-programming.com/machine1000.html*, 1999.

[2] N. Boden, D. Cohen, R. Felderman, A. Kulawik, C. Seitz, J. Seizovic, and W. Su. Myrinet: A gigabit-per-second local area network. *IEEE Micro*, 15(1):29–38, Feb 1995.

[3] K. Alnaes, E.H. Kristiansen, D.B. Gustavson, and D.V. James. Scalable coherent interface. In *CompEuro 90*, 1990.

[4] Giganet. Giganet clan.

[5] G. Ethernliance. Gigabit ethernet: Accelerating the standard for speed, 1999.

[6] David W. Walker. The design of a standard message passing interface for distributed memory concurrent computers. *Parallel Computing*, 20(4):657–673, March 1994.

[7] Steven S. Lumetta, Alan M. Mainwaring, and David E. Culler. Multi-protocol active messages on a cluster of smp's. In *Proceedings of the 1997 ACM/IEEE SC97 Conference*, pages 15–21, San Jose California, USA., November 1997. ACM Press and IEEE.

[8] A. D. Birrel and B. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, pages 39–59, Feb 1984.

[9] Multiprocessor specification version 1.4. Technical report, Intel Corp, 1997.

[10] G. Burns, R. Daoud, , and J. Vaigl. LAM: An Open Cluster Environment for MPI. www.lam-mpi.org, 1994.

[11] Almadena Chtchelkanova, John Gunnels, Greg Morrow, James Overfelt, and Robert A. van de Geijn. Parallel implementation of blas: General techniques for level 3 blas. Technical report, The University of Texas at Austin Austin, Texas 78712, 1995.

[12] A.K. Dewdney. Computer recreations. *Scientific American*, 250:22–34, 1984.

[13] Toshiyuki Takahashi, Francis O'Carroll, Hiroshi Tezuka, Atsushi Hori, Shinji Sumimoto, Horoshi Harada, Yutaka Ishikawa, and Peter H. Beckman. Implementation and evaluation of MPI on an SMP cluster. In *IPPS/SPDP Workshops*, pages 1178–1192, 1999.

[14] D. O'Hallaron. Spark98: Sparse matrix kernels for shared memory and message passing systems, 1997.

[15] D. S. Henty. Performance of hybrid message-passing and shared-memory parallelism for discrete element modeling. In *Supercomputing*, 2000.

A.4 Extending the Applicability of software DSM by adding user redefinable memory semantics

EXTENDING THE APPLICABILITY OF SOFTWARE DSM BY ADDING USER REDEFINABLE MEMORY SEMANTICS

B VINTER[†,‡], O J ANSHUS[‡],T LARSEN[‡], J M BJØRNDALEN[‡]
Department of Mathematics and Computer Science[†,‡],
University of Southern Denmark

Department of Computer Science[‡],
University of Tromsø

This work seeks to extend the applicability of distributed shared memory (DSM) systems by suggesting and testing an extended functionality "User Redefinable Memory Semantics" (URMS) that may be added to most DSM systems. We show how URMS is included with one DSM system, PastSet, and sketch the applicability and ease of use of URMS for a set of current applications. Finally, a micro-benchmark of URMS on PastSet, show performance gains of 79 times over a naive DSM model, 48 times over a tuple based DSM model and 83% over MPI.

1 Introduction

Software distributed shared memory (SW-DSM) systems have existed for about 15 years, but have yet to achieve widespread application by programmers outside of the DSM research community. One reason contributing to this may be that DSM systems are usually demonstrated and tested only for a limited set of high-performance computing test applications; typically a subset of the SPLASH[1] benchmark suite. These test applications provide a thorough test of specific performance limits of DSM systems, but fails to demonstrate the applicability of DSM outside high-performance computing.

This work proposes an extended DSM functionality, User Redefinable Memory Semantics (URMS), to simplify the use of DSM by programmers. We demonstrate how URMS is added to one SW-DSM system, PastSet, as well as the performance impact of URMS on PastSet, and show how URMS may be added to other types of SW-DSM.

2 User Redefinable Memory Semantics

To address some of the performance problems for DSM systems, we propose the concept of "User Redefinable Memory Semantics" (URMS). The principle behind URMS is to offer users the opportunity to redefine the semantics of any or all memory operations for memory areas that are specified by the user. The redefined semantics are specified by providing code that should be executed instead of the memory

operation. The specification of the redefinition may also include initialization code that is applied once to initialize the specified memory area. In effect, the redefined memory operation semantics will be applied for memory operations on the specified areas only.

For example, a memory location may be redefined to accumulate a global sum of partial sums that are produced by independent processes. For that location, the STORE operation would be redefined according to code that stores the aggregate sum, and keeps track of a complition criterion that may be realized as an access count, process list, or by other means. The LOAD operation is redefined to return the aggregate sum only after the reduction has completed, in effect blocking until the termination criterion used for the STORE operation is satisfied. This approach makes it very simple for programmers to overlap communication and calculation if there is any work that may be done between the partial sum is ready and the global sum is needed. For the variations of this example that are to follow, we will consider only the limited case where a given number of contributions define the completion of an operation.

Basically URMS may provide a way to introduce well known message-passing techniques to DSM systems, while preserving the illusion that the programmer is using a shared memory computer. We hope that URMS based systems will be simpler to program and at the same time provide superior performance because the URMS memory cells may be handled in an optimized way by the runtime environment.

2.1 Adding URMS functionality to DSM systems

In this section we sketch how URMS functionality may be added to different variations of distributed shared memory: Region Based DSM, Shared Virtual Memory based systems, Object based DSM, and Structured DSM systems.

Region Based DSM systems provide differing memory models, e.g. whether a region constitutes a single variable, a fixed size block, or a variable size block. Further, the transparency of the systems varies from full transparency to no transparency. We will illustrate how URMS may be implemented for a variable region size system with little transparency, such as CRL[2]. C Region Library, CRL, is based on program information to maintain the distributed shared memory. Programmers explicitly associate a region with a block of memory, and state when this memory is used with the instructions, `rgn_start_read`, `rgn_end_read`, `rgn_start_write` and `rgn_end_write`. Before regions can be addressed they must be created, and it is straightforward to add URMS functionality at this stage. Instead of a `rgn_create(size)` a programmer could use a `rgn_create(size, operation, parameters)`., e.g. `rgn_create(sizeof(double), REDUCE_SUM, WORKERS)` would create

a new region which is a global sum from WORKERS processes. When a process issues a `rgn_start_write` on the created region, the value that is written is automatically added to a global sum, and a `rgn_start_read` will simply block until all WORKERS processes has issued their `rgn_end_write` on the region. Further, the global sum can be distributed to all participants before they issue `rgn_start_read` operations, which will remove the overhead on fetching the data.

Shared Virtual Memory (SVM) systems, such as Shrimp-SVM[3], base the DSM system on the page size of the native architecture, and use the paging mechanism to trap memory operations that cannot be serviced, e.g. a read to an address that is not present in local memory, or a write to a memory cell that is either replicated on other nodes or not present. The handling of URMS may be easily added to the exception handling of SVM page faults. For a global reduction operation, a SVM system can incorporate a specific data type for reduction variables. STORE to variables of this type causes exceptions that the runtime system may capture and add to the reduction. Loads from reduction variables are stalled if the reduction is not yet finished; otherwise the result is available and the application may continue un-delayed. If the SVM system supports shared writes by multiple processes, the URMS functionality may easily be added to the write assembly process.

Object Based DSM, like Orca[4] is perhaps the least obvious candidate for URMS, since similar functionality may be achieved on user level, via a carefully considered library. Adding URMS to the runtime system is extremely easy however, and may result in improved performance compared to a user level implementation. Since most object based DSM systems are closely integrated with the programming language, it is straightforward to add keywords to identify URMS data: e.g. double sum = new GLOBAL_SUM(WORKERS), will associate the variable sum with a global sum reduction among WORKERS. Rather than maintaining coherent replicas of sum, writes to sum are identified at compile time and converted into a global reduction participation call. During execution, reads of sum may be blocked until the reduction has completed.

Structured DSM cover a set of slightly similar DSM models that model memory in some structured way, such as Linda[5]. Linda models memory as an associatively addressed tuple space. Processes can add, read or retract process tuples and data tuples from tuple space. Adding URMS to a tuple space model is simple and probably represents the system with least intrusion to the given DSM model. A generic tuple is made up of a flag followed by a set of data entities, e.g. ("P", double). The model can easily be extended by a set of flags, which relates to memory semantics. In this manner the global reduction can be handled by issuing out("GLOBAL_SUM", "group-flag", int WORKERS, double data), which is captured by the runtime system. Only as WORKERS tuples matching the group-flag are placed in tuple space, will

the runtime system make available WORKERS tuples of the template ("group-flag", double data) where the data is the resulting sum.

3 Applications

Identifying useful URMS functionalities will be an application driven investigation; we have identified several potentially useful functions. One class can be taken directly from the reductions that are supported by MPI, such as global-sum, -product, -min and -max.

The appearance of very high performance WAN's have spawned a large interest in meta- computing, i.e. the idea of a large computational grid[6]. A common feature in the grid-middleware projects is the ability to read a file from a remote server. However, scientific datasets are often very large, from multiple terabytes to pentabytes of data. Most often the user does not wish, nor possess the computing power, to process all the data. Instead, the application work only on data that fit a predefined criterion, thus transferring the complete dataset over a WAN, only to filter the data locally. If we replace the standard grid-model with DSM, the natural solution is to spawn a thread that runs on the data-server, which then filters the data at the source and only transfers the specific data the user is actually going to use. However, while any meta-computing middleware requires the servers to place some trust in verified clients, the full flexibility that comes with allowing remote clients to start threads, is unlikely to be accepted by those that hold the servers. Alternatively a DSM system, which supports URMS, may provide a solution that is just as simple to use for the programmer while preserving the control at the server-side. Data may be mapped in the users memory as available, and then accessed using the underlying DSM system. The filtering of data may be done by a URMS function that processes a logic-based query, which the user specifies. This way the server maintains full control over both the instructions that are executed and the compute-time a user may consume.

In general URMS may prove to be a convenient abstraction to eliminate explicit communication in applications that use streams of data, such as streaming video, net-chat applications etc. URMS can eliminate stream operations by basically replacing sockets with URMS memory cells, thus reading from a stream becomes reading from an address and vice versa for writing. In many ways this translates into the difference between accessing port-mapped and memory mapped devices in a computer system, anybody that have written operating systems knows that memory mapped devices are far easier to program than port-mapped devices.

This approach could ease programming in many distributed fields and given a sensible naming convention within the DSM system, could hide most of the distribution aspects altogether. Servers for mobile agents can provide a single address where incoming agents are copied to for migration, which reduces the migration problem

to a memory move.

URMS may also allow large server applications, such as web servers or database servers to execute on DSM based clusters, by modeling request queues as URMS memory cells, e.g. adding a request to the queue is done by writing the request to an address. The URMS functionality can do simple load-balancing, or it may keep a history of the requests passed to the individual nodes and attempt to achieve a better utilization of cached data.

4 Adding URMS to a DSM system

To demonstrate our idea we have added URMS to our own structured DSM system, PastSet[7]. PastSet is similar to Linda in its perception of memory, but has a distinctly different execution model. Tuples can be stored, read, and manipulated in PastSet memory. Tuples are generated dynamically based on tuple templates that may also be generated dynamically. The collection of all tuples in PastSet based on a unique template is denoted an element of PastSet. Elements are unique, each corresponding to a unique template. A move operator (`Mv`) writes tuples to PastSet memory and an observe operation (`Ob`) specifies a template and reads a matching tuple.

URMS functionality has been added to PastSet, we call the URMS framework X-functions, which are pairs of functions, one associated with writing and one with reading. In this way tuples may be manipulated as they are stored and read. In the current implementation only one X-function pair can be associated with each element at any time.

We have implemented X-functions that do generic pattern matching as well as global operations as used in MPI[8]. A global reduction works as follows, the first time the element is accessed the desired global reduction, e.g. sum, and the number of participants in the reduction is specified. When the reduction takes place all processes will simply write(`Mv`) their tuple to the element and read(`Ob`) the resulting sum, the reads automatically block until all participants have written their contribution to the global sum. We are experimenting with more advanced X-functions, including an X-function that compresses bitmaps on write and decompresses on read.

4.1 Micro benchmark

To illustrate that one may achieve performance improvement of URMS over conventional DSM solutions we show the time taken for a global reduction of an integer for a set of DSM approaches. To further emphasize on scalability we show the progressive performance on one through 32 participants. Our test bench is a cluster of eight four-way Pentium Pro nodes interconnected by a 100 Mb/sec switched FastEther network. We choose a global reduction over a full application, even though the usability

```
Ob(semaphore);
Ob(counter); counter++;
Ob(value); value+=local;
if(counter==workers){
  result=value;
  Mv(result);
  counter=0;
  Mv(counter);
  value=0;
  Mv(value);
  Mv(semaphore);
} else {
  Mv(value);
  Mv(counter);
  Mv(semaphore);
  Ob(result);
}
```

```
Ob(counter, value);
counter++;
value+=local;
if(counter==workers){
  result=value;
  Mv(result);
  counter=0;
  value=0;
  Mv(counter, value);
} else {
  Mv(counter, value);
  Ob(result);
}
```

```
Mv(local);
Ob(result);
```

Figure 1. Global reduction using std. memory model, tuples and URMS.

of global reductions is limited to dusty-deck type applications, which is not necessarily the most likely use of URMS. However the global reduction is simple, widely known and allows us to perform simple performance comparisons. The global reduction is implemented in three versions, one that treats memory in a conventional way, one that utilizes the tuple nature of PastSet and finally one that use an URMS implementation of global reductions, all three reduction-codes are listed in figure 1.

The first implementation models a standard memory. Because PastSet is built on blocking tuple operations there are no other synchronization mechanisms. Instead memory operations are used as replacements for semaphores and signals. Thus in the code a tuple called semaphore is used for achieving mutual exclusion, and the signal to show that the result is ready is replaced by a blocking read of the result. The second version is the natural way to perform a global reduction with a tuple based memory model. The reduction is performed with a tuple containing the partial sum and a counter. This tuple migrates between all processes, the last process to add its partial sum writes a global result, which all others read. The final version is based on a URMS function, which performs the reduction. The URMS function simply treats writes to this tuple as partial sums and reads as reading the global sum, thus all reads are blocked until the global sum is ready.

The three versions have been tested on one through 32 CPUs, measurements are taken as average over 1000 reductions and each experiment is repeated five times for each number of processes, the resulting numbers are quite stable with a standard derivation less than 1% at 32 processors. Figure 2 shows the latency of a global reduction. In addition to the three memory versions we have added the time used by LAM-MPI to perform `MPI_Allreduce`.

The improvement that is gained with the use of URMS is significant. With 32

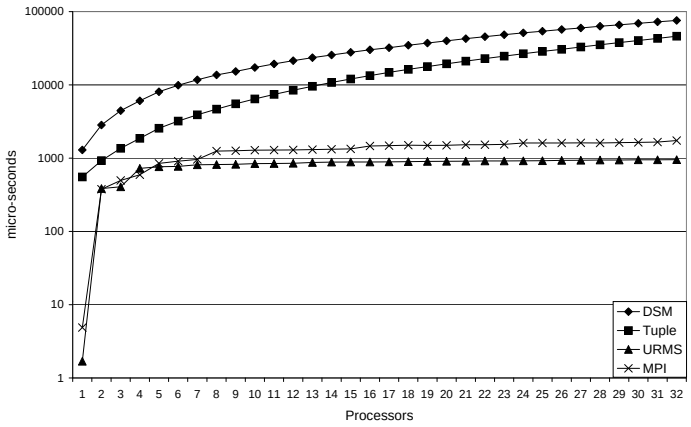


Figure 2. Global reduction performance using the algorithms in figure 1.

processors the URMS solution is 79 times faster than the standard memory model and 48 times faster than the more natural tuple version. The MPI version is 83% slower than the DSM version using URMS, since the global reduction is a generic MPI operation this is a significant proof-of-concept for URMS.

5 Conclusions

In this work we have introduced the concept of User Redefinable Memory Semantics, URMS. We believe that URMS is easy to use for programmers, and solves efficiently some operations that have been documented to be costly to perform. We have shown the benefit of adding URMS to a structured DSM system, PastSet, and have argued why other DSM models may have even greater advantages from URMS. We are working to identify which functionalities are useful in URMS systems. We wish to test the URMS concept on other types of DSM, particularly region based and page based systems, in order to test potential performance benefits with other DSM models than the structured DSM. Using a micro-benchmark that performs a global reduction of an integer value, we have shown that there may be as big a performance advantage of URMS over conventional memory as 79 times and 48 times over the natural PastSet approach. The fact that a DSM system using URMS is 83% faster than LAM-MPI, on a global reduction micro-benchmark, indicates that future work on URMS could be very rewarding.

References

1. Jaswinder Pal Singh, Wolf-Dietrich Weber, and Anoop Gupta. SPLASH: Stanford parallel applications for shared-memory. *Computer Architecture News, 20(1):2–12, March 1992.*
2. *Kirk L. Johnson, M. Frans Kaashoek, and Debroah A. Wallach. Crl: High-performance all-software distributed shared memory. In* Proceedings of the 15th SOSP, *pages 213–228, December 1995.*
3. *R. Samanta, A. Bilas, L. Iftode, and J. P. Singh. Home-based svm protocols for smp clusters: Design and performance. In* Proc. of the 4th IEEE Symp. on High-Performance Computer Architecture, *February 1998.*
4. *H. E. Bal and A. S. Tanenbaum. Orca: A language for distributed object-based programming.* SIGPLAN Notices, 25(5):17–24, May 1990.
5. N. Carriero and D. Gelernter. Linda in context. *Commun. ACM, 32(4):444–458, April 1989.*
6. *Ian Foster and Carl Kesselman. Computational grids. In Ian Foster and Carl Kesselman, editors,* The Grid: Blueprint for a New Computing Infrastructure, *pages 15–51. Morgan Kaufmann, San Francisco, CA, 1999.*
7. *Brian Vinter, Otto J. Anshus, and Tore Larsen. Pastset - a distributed structured shared memory system. In* Proc. of High Performance Computers and Networking, *Amsterdam, April 1999.*
8. *David W. Walker. The design of a standard message passing interface for distributed memory concurrent computers.* Parallel Computing, 20(4):657–673, March 1994.

A.5 PATHS - Integrating the Principles of Method-Combination and Remote Procedure Calls for Run-Time Configuration and Tuning of High-Performance Distributed Applications

PATHS - Integrating the Principles of Method-Combination and Remote Procedure Calls for Run-Time Configuration and Tuning of High-Performance Distributed Applications

John Markus Bjørndalen[1], Otto Anshus[1], Tore Larsen [1], Brian Vinter[2]

Department of Computer Science [1]
University of Tromsø

Department of Mathematics and Computer Science [2]
University of Southern Denmark

Abstract

A "path" based on the idea of method-combination and remote procedure calls to provide run-time configurable networks of computational communication paths between threads and data in distributed, high performance applications is proposed. An initial design is implemented, tested and analyzed.

We use a "wrapper" to provide a level of indirection to the actual run-time location of data by forwarding function calls to servers holding the target data. A wrapper specify where data is located, how to get there, and which protocols to use. Wrappers are also used to add or modify methods accessing data. Wrappers are specified dynamically. A "path" is comprised of one or more wrappers. Sections of a path can be shared among two or more paths. Establishing a path is a two-phase process of specifying the path, and then (recursively) setting up the path based on the specification.

A test system using the proposed architecture is implemented, demonstrated and performance measured using two benchmarks on three different clusters. We show that the proposed architecture can be used for mapping and improving the performance of applications on different topologies including a wide-area multi cluster configuration, and how wrappers can be used to distribute computational load off of heavily loaded target servers. We also show how thread distributions can be changed at run-time without altering application code.

We believe that the proposed semi-manual approach will prove useful in debugging and coarse-tuning distributed high-performance applications, and that it will provide valuable insights for developing later, more automated, middleware systems.

1 Introduction

A key challenge when running distributed high performance applications is to maintain thread-to-host mappings that achieve high performance or efficient execution. Attacking this challenge requires balancing the potentially conflicting goals of distributing threads for improved load balancing and for reduced communication overhead.

In reality, high scalability cannot be achieved unless the system is fine-tuned to balance computation, communication, and synchronization requirements. Unfortunately, high performance is often achieved

only after rigorous manual fine-tuning to obtain an efficient mapping of threads to hosts.

Efficient thread-to-host mappings may be achieved by directives in the application source-code, reflecting the topology of the host architecture in the application source-code. Alternatively, mappings may be obtained by communication libraries or by middleware. Static or dynamic mappings may be used depending on application dynamics, the homogeneity of the underlying architecture, or the regularity of the interconnection topology. Dynamic mappings accommodate changing needs over the application lifetime by use of costly thread migrations. In this case, the original placement problem is transformed into a problem of determining where and when any thread should be migrated.

One alternative to compile-time static placement and runtime dynamic placement is to implement a static placement on an irregular architecture by deciding on a mapping at load-time. A "manual" approach to the load-time solution is to allow the application programmer to instruct the middleware as to the distribution and communication patterns, specified either explicitly by the programmer or chosen from a library of algorithms. We have combined the static load-time and the dynamic run-time approaches. We allow for run-time placement, but we do this typically at the startup of the application by using a configuration map loaded with the application. However, the application can change this mapping at will if it so wishes.

This paper describes our initial work on a middleware extension inspired by method combination[8] and remote procedure calls[3] which allows the communication topology to be directed by specifying meta-code and meta-data, without introducing any modifications to the application code. The extended middleware also allows computations to be placed along the access paths to data. For now, we assume that the application under study is alone in using the underlying architecture; there are no other applications competing for resources. The goal of the mapping then, is to achieve high performance for one single application running alone.

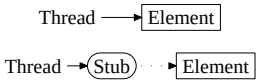
We show how one may experiment with different mappings of threads and their intercommunication, and demonstrates that this can identify flexible location policies which are independent of the application code and still supply effective placements.

Section 2 describes how wrappers are used and combined to specify and identify access paths. Section 3 describes our experiments using three different clusters located at the University of Tromsø, Norway and the University of Southern Denmark, Odense. Section 4 presents and analyzes the experiment results, section 5 presents related work, while section 6 presents our conclutions.

2 The configurable path framework

Our research platform uses the PastSet[1][13], a structured distributed shared memory system in the tradition of Linda[4]. A PastSet Element is a tuple space with tuples of the same or equivalent types, and is globally addressable with a name and the type of the tuples residing within it. PastSet also supports X-functions[14], which can be used to modify the behavior of the PastSet operations (to implement functionality such as, but not limited to, global reductions and caches).

A common way to implement remote access to shared objects such as a PastSet Element is to give the accessing thread a stub which forwards the operations to the remote server. The stubs interface is identical to the interface of the acessed element, providing transparent access to both local and remote elements.



A stub does not necessarily have to be limited to implement remote access though. It can just as easily be used to modify the semantics of an elements operations by implementing one of the PastSet X-functions (such as the global reduction sum). If stubs can be combined, we can easily set up a remote element which is used to compute global sums as follows:

We call the combination of the stubs a *path* to the remote element. The combination of all existing paths to an element forms a tree with threads as the leaves and the element as the root.

As long as the application only uses the reference to the topmost stub, it can be mapped to another cluster configuration without changing the application code. Fitting and optimizing the application to a particular configuration can instead be done by changing the path-building metadata and code.

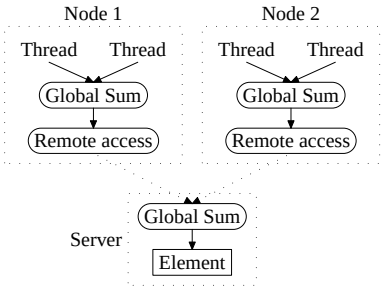


Figure 1: Threads in two nodes accessing a shared global sum element. Each node computes a partial sum before forwarding it to the global sum wrapper in the server.

2.1 Building and specifying paths

Setting up access from a thread to a PastSet element involves the following two stages:

1. Specify the the path. This involves examining information about the cluster topologies, the location of threads in the cluster and where the target element is located.

2. Build the path from the description. This involves creating and binding the wrappers with parameters specified in the path description.

To allow configurability of the wrappers, we include parameters for each wrapper in the path description. Some of these parameters are common for all wrapper types (such as whether the wrapper need to use thread synchronization mechanisms), or type specific (such as the protocol to use, remote address and service requirements in a remote access wrapper). Parameters not specified are assigned default values.

An example path description used by one of the nodes in Figure 1 is included in Figure 2. `build_path` builds the given path and returns a reference to the toplevel wrapper in the path.

Each thread creates (or is given) its own path description and calls `build_path` to get its own reference to the path. `Build_path` takes care of merging paths when the path descriptions allow for sharing parts of the path.

2.2 Current implementations

The current implementations use Common Lisp and Python for management of paths, while the PastSet applications and wrappers are implemented in C.

This allows us the flexibility of high level dynamic languages for experimenting with path building code, while keeping the high-level languages out of the loop when benchmarking the different configurations.

```
path = make_path(stage("reduce-sum", num_threads=2),
                 stage("remote", proto=TCP, host="p0"),
                 stage("reduce-sum", num_threads=2),
                 stage("core", name="PI-SUM1"))
elm = build_path(path)
```

Figure 2: Example path description

The work reported in this paper is based on experiments with the Python framework, which allows the path framework to be provided and extended either through embedded Python (by overloading two default functions for acquiring and releasing a path to an element) or, as we did, by handing path references to C algorithms written as Python extension modules. The latter method allows different Python scripts to experiment with path building and thread spawning using the same compiled C code for all experiments.

The wrappers can also be used directly from the high-level languages allowing, for instance, Python scripts direct access to tuples and elements.

Simple profiling of PastSet operations is provided with two trace wrappers. The "timestamp" trace wrapper simply forwards the operation to the next stage in the path and uses the Pentium timestamp counter to timestamp the start and completion time for each operation invoked. The timestamp data is logged to an array in memory and written to disk when the trace wrapper is deleted (reference counting is used to determine when wrappers should be deleted). The overhead of this wrapper is around 100-120 clock cycles.

The "operation" trace wrapper is an extension of the timestamp trace wrapper which additionally logs the contents of the tuples provided to or returned from the PastSet operations.

Any number of trace wrappers can be inserted anywhere in the path trees.

3 Experiments

To show how the framework can be used for mapping and optimizing an application to different topologies, we devised experiments to map two benchmarks to different path trees and two different thread allocation policies. We used these experiments to examine some choices which can be made when mapping an application to a cluster or multi-cluster environment.

We believe that changing the mapping will produce performance benefits because of the different emphasis put upon locality, load balancing and communication. Also, the potential mismatches between the collective data access patterns by all the threads, and each processor's individual data cache will be influenced by different mappings. Of course, the application will play a role in how successful a mapping is. For instance, frequent use of synchronization using locks, and especially global locks, will play a role in the resulting performance and the effect of a mapping.

Two basic benchmark codes were used:

- The **Global Sum benchmark** (GSum), which measures the average execution time of a global sum operation. The number of values to sum is equal to the number of threads used in the experiment.
- **Monte Carlo Pi** (MCPi), which computes an approximation of Pi by randomly throwing a number of darts and counting those hitting inside a circle.

  A total number of N darts are thrown by the threads, splitting the darts

evenly between the threads.

The time of throwing the N darts and running a global sum with the results is measured. N was 10 million for the cluster tests, and 100 million for multi-cluster tests. The problem is large enough that the communication latency should be masked by the time spent in the computation.

Figure 3 shows pseudocode for the benchmarks. The *TS()* macro samples the pentium timestamp counter, and stores the timestamp in an array. *gettimeofday()* samples the real-time clock on the host computer with microsecond resolution. The gsum benchmark was run with "iters" set to 1000. For both tests, the average of 5 benchmark runs are plotted in the graphs.

Based on the two benchmarks, we deviced the following experiments:

- Scaling on one node. Measure the execution time of a global reduction when we vary the number of threads from 1 to 16 on a single node.
- Thread placement and topology optimization. Two different thread distribution algorithms are used to assign threads to nodes in the clusters.

  The path trees were also varied to experiment with computing partial sums within partitions of the clusters to reduce the work and communication on the node hosting the target element.
- Monte Carlo Pi in cluster and multicluster configurations. Verify that the application can be mapped and scaled to the three clusters and when using the three clusters together in a multicluster configuration.
- Multicluster global sum. Measure the execution time of global sum using all three clusters with 3 to 96 threads. Threads are assigned evenly among the clusters.

The benchmark code was unchanged during the experiments, we only changed parameters and metadata for the Python framework code used to map threads and set up the paths.

Available for the experiments were 3 clusters with 32 processors in each, organized as follows:

- 2W cluster - 16 * 2-Way (Dual) Pentium III 450 MHz, 256MB RAM, Location: Odense, Denmark.
- 4W cluster - 8 * 4-Way (Quad) Pentium Pro 166 MHz, 128MB RAM, location: Tromsø, Norway
- 8W cluster - 4 * 8-Way Pentium Pro 200 Mhz, 1GB RAM, location: Tromsø, Norway

In addition, the root node for the multicluster experiments was a dual Pentium II 300 MHz machine with 256MB RAM located in Tromsø. One experiment was also run on a 650 Mhz Pentium III notebook (Dell Latitude CPx, 256MB RAM) to get results for a single-processor node.

For the current experiments, we only used TCP/IP over 100MBit ethernet for intra-cluster communication. The 4W cluster was connected to the root node through a HP 100 VG anylan switch, while the 8W cluster was connected to the root node using the departments local area network. The intra-cluster for the 8W cluster was a switch connected to the departments LAN.

The connection between Tromsø and Odense was the departments internet backbone.

4 Results

Figure 4 show the difference in execution time of the two different thread distribution algorithms. The *even distribution* algorithm distributes threads evenly among the nodes in the cluster. It starts with the first node and adds one thread to each node before it

```
barrier_sync();
gettimeofday();
TS(0);
for (i = 1; i= < iters; i++) {
   sum = gsum(i);
   TS(i);
}
gettimeofday();
```

(a) gsum

```
barrier_sync();
gettimeofday();
TS(0);
n_inside = mcpi(to_throw);
total = gsum(n_inside);
TS(1);
gettimeofday();
```

(b) mcpi

Figure 3: Pseudocode for the Global Sum and Monte Carlo Pi benchmarks. Only one of the threads runs the timestamp code. The others run the same code without the timestamp code in it.

goes back to the first node again. The *bucket* algorithm fills up one node with threads (number of threads equal to the number of CPUs in the node) before proceeding to the next.

As expected, once we reach 32 threads and the two distribution algorithms generate the same number of threads on all nodes, we end up with the same execution time with both algorithms.

For the 4-way and 2-way cluster, the *bucket* distribution algorithm performs better in the range between 1 to 32 threads. This is because we only need to pay the cost of bringing in a new node when the bucket algorithm has filled up the last node.

The 8-way cluster shows a different pattern though. After 4 threads, the bucket algorithm performs worse than the even distribution algorithm. The reason for this can be found in figure 5, in which the "Non-partitioned" graphs show the execution time of global sums with 1-16 threads on single nodes.

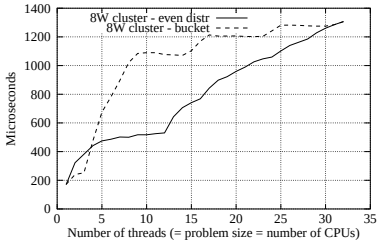
As the number of threads increase, we observe a sudden jump in execution time around 3-5 threads for the different SMP nodes. For the 8-way nodes, this execution time quickly grows over 800 microseconds when the number of threads equal the number of CPUs in the node. This shows that the internal synchronization cost for the global sum is higher than the node-to-node communication costs.

The curious jump in latency between 12 and 13 nodes for the *even distribution* algorithm on the 8-way nodes can also be explained from figure 5. At 12 threads, we have 3 threads running on each node. When we add one more thread, one of the nodes will increase to 4 threads, which corresponds to the point in the figure 5 where we get a sudden jump for the 8-way nodes. This jump is reflected in the cluster graphs since the execution time of the global sum is dictated by the slowest node.

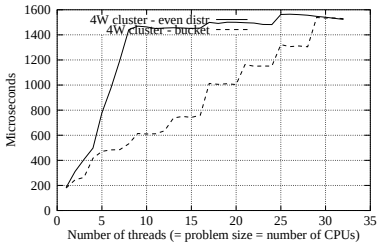
Since all the multiprocessor nodes show a distinct increase in latency once the node holds more that 3-4 threads, a natural assumption is that using partial sums might improve the execution time. The "partitioned" graphs in figure 5 shows an experiment where we limit the number of threads per sum wrapper to 4 by arranging the threads and wrappers in a hiearchial sum (see figure 6).

The graph shows that by limiting the number of threads to the range where the wrapper has the best performance, and at the same time increasing the potential parallelism, the execution time can be improved by roughly a factor two. The extra overhead of contributing through two layers of sums is less than the overhead reduced by partitioning the problem.

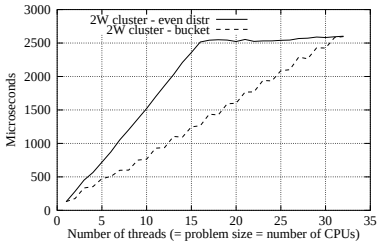
The above results suggest that partition-



(a) 8-Way cluster

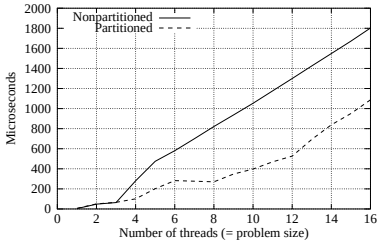


(b) 4-Way cluster

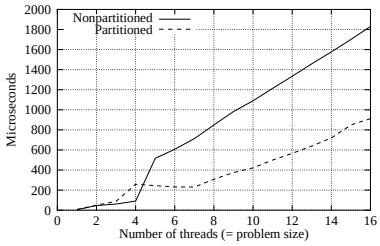


(c) 2-Way cluster

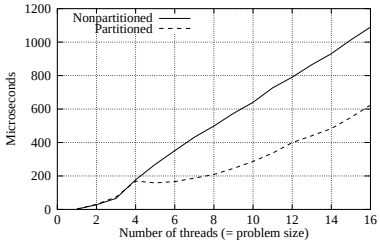
Figure 4: Execution time of global sum in each cluster using two different thread allocation algorithms



(a) Single 8-way node



(b) Single 4-way node



(c) Single 2-way node

Figure 5: Partitioning on single nodes. Single process, increasing the number of threads from 1 to 16. "Partitioned" uses a maximum of 4 threads per global sum, organizing the sum wrappers in a hierarchial partial sum tree.

ing the the clusters such that groups of nodes within the cluster contribute to a partial sum before the partial sums are added in a root node might improve the latency of the cluster, not only because of the higher level of parallelism in the cluster, but also due to a better resource usage in in the wrappers.

Figures 7 and 8 show experiments where we partitioned the path trees for the *even distribution* and *bucket* thread distribution algorithms. The path tree was first partitioned such that no sum wrapper had more than 4 contributing threads. The sum wrappers for the partitions were placed on nodes such that no node had more than one of the partial sum wrappers. This increased the network traffic from 16 to 20 roundtrip

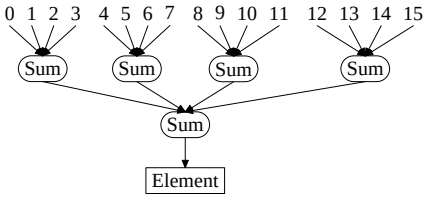
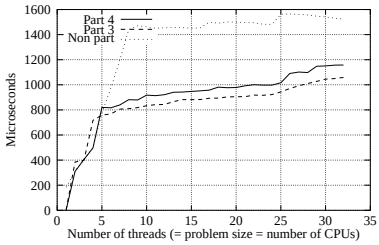
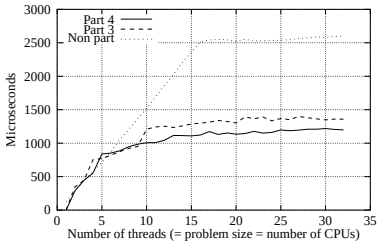


Figure 6: Hierarchial global reduction sum tree. The numbers represent threads. The upper layer of sum wrappers computes partial sums used in the lowermost sum wrapper.



(a) 4-Way cluster

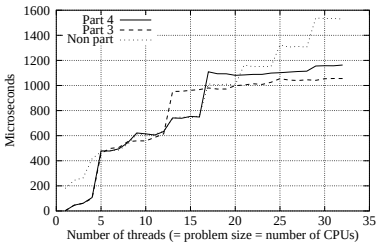


(b) 2-Way cluster

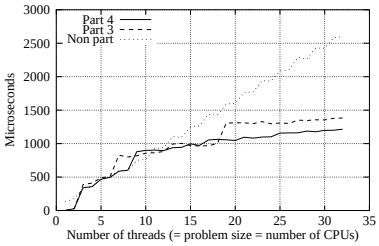
Figure 7: Cluster partition tests - even distribution

messages per sum in the 2-way cluster, and from 8 to 10 in the 4-way cluster.

Once the 4-split tests were made, we spent anoter 10-15 minutes making a map which reduced the number of threads per wrapper to 3. This added another level in the sum hiearchy and increased the number of roundtrip messages to 23 per sum for



(a) 4-Way cluster



(b) 2-Way cluster

Figure 8: Cluster partition tests - bucket

the 2-way cluster. The 4-way cluster didn't need another level, but increased the number of roundtrip messages to 11.

No tests for the 8-way cluster were made since the mechanism for partitioning the leaf threads within a node are not ready yet.

Both the 4-split and 3-split graphs show an improved operation execution time compared to the non-split graphs. For the *even distribution* graphs, we get a brakeoff at the point where the whole partial sum tree has been expanded, and only the number of threads at the toplevel is increased.

4.1 Multicluster results

Figure 9 shows the minimum, maximum and average operation execution time of a global reduction in a multi-cluster environment, going from 3 to 96 threads, at each step adding one thread to each cluster.

The figure show a signifiant variance in latency, ranging from 34 to 53 milliseconds,

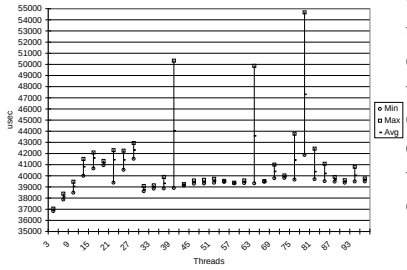


Figure 9: Multicluster global reduction

which is due to variation in the network latency between the Odense and Tromsø cluster.

The Path-framework allowed an easy hierarchical mapping of the threads in the global sum path trees, thus each reduction only generates one roundtrip message (contribute sum and retrieve result) between the Odense and Tromsø clusters, independent of the number of threads in the system. The intra-cluster latencies of the sums are so small, that they are not visible due to the high variation in latency in the Odense-Tromsø connection.

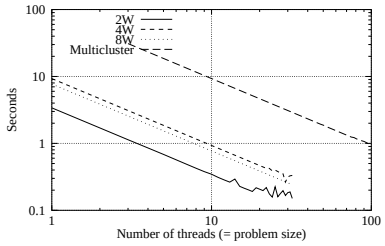


Figure 10: Monte Carlo Pi in clusters (10M darts) and multicluster (100M darts).

Figure 10 shows Monte Carlo Pi executed on the three clusters and on the multicluster. We observe the expected linear scaling for all instances. The multicluster problem is 10 times bigger than the one run on the individual clusters, and the work is divided evenly amongst the threads in the multicluster. Because the work is distributed evenly performance of the multicluster is dictated by the slowest CPUs in the system, which is the ones in the 4W cluster, as a result perfect speedup must be defined as the multicluster setup being 10 times slower than the 4W cluster, the graph clearly show this to be true.

5 Related work

Accurate and efficient performance prediction of existing distributed and parallel applications on target configurations with potentially thousands of processors is hard. Analytical solutions are difficult to develop, and many complex systems can be intractable. Simulation is a widely used tool, but its major limitation is its, often extremely, long execution time for large-scale systems. A number of simulators have been developed, including Parallel Proteus[9], LAPSE[5], SimOS[11], and Wisconsin Wind Tunnel[10]. These simulators typically are themselves parallel and use direct execution of portions of the code to reduce the cost. The slowdowns range from 2 to 100. Few simulators simulate both computation and I/O operations. In contrast to simulators, our approach execute the actual application code several times, each time with a different mapping. Of course, running an application, say, 10 times before deciding on a configuration to use, will give a slowdown of 10. However, the flexibility and simplicity is high.

In [7] it is shown that the three parallel computation models BSP, E-BSP and BPRAM in several situations do not precisely predict the actual runtime behaviour of an algorithm implementation. They report performance deviations between 25-200%. This is explained by the different approaches to communication and routing used by the models. Caching effects are also possible causes. Also, the efficiency of an implementation derived from the three

models did not match the performance possible by using hand tuned implementations. These results can be used to make a case for a system like ours where the programmer can try a few configurations and select the one giving the best performance. This can prove to be much simpler than hand coding an algorithm to utilize the hardware platform. The resulting performance will most likely not be optimal, but it can be better than not doing anything.

In [6] both processor and memory load balancing are used to support low contention and good scaling to hundreds of processors. Gang-scheduling is used to avoid wasting cycles spinning for a lock held by a descheduled process (actually, a virtual CPU). In contrast, our system is much simpler and provides for much less or no automatic support at the present time.

In [12] it is shown that there is a communication and load bralance trade-off when partitioning and scheduling sparse matrix factorization on distributed memory systems. Block based methods result in lower communication costs and worse load balancing, whereas a "round robin"-based scheme where all threads are distributed over the processors gives better load balance but higher communication costs.

In [16] an approach to load balancing for general-purpose simulations is reported in with little modification is needed to the user's code. Their approach uses runtime measurements and demonstrates better load-balancing than approaches without such measurements. Three different load-balancing mapping algorithms are used. This approach is similar to ours in that little modification of the user's code is needed. As they do, we also use different mappings and leave it to the application to control them. Our approach differ in that we can both try different mappings and add arbitrary code along the access path to data. Also, we differ in that we do a prerun of a few mappings, and then we choose a single one and we let the application use the selected mapping without incurring further overhead. Of course, we take all the overhead when choosing a mapping. For clusters where they can be dedicated to applications running often, this configuration hunting overhead will be amortizised over time.

In [2] three categories of useful tools were found when tuning the performance of NOW-Sort, a parallel disk-to-disk sorting algorithm on a cluster system: tools that help set expectations and configure the application to different hardware parameters, visualization tools that animate performance counters, and search tools that track down performance anomalies.

We believe that our system can, by simple means presently controlled by the programmer, improve performance by finding a configuration where the resource usage better avoids hot spots, bottlenecks, and expensive waiting times for processor, memory, cache, and I/O by compromising between load sharing and communication. The flexibility of using maps, paths and wrappers also make it possible to monitor the application and provide data for visualization of both behaviour and performance. At the present time we have not investigated approaches to sharing clusters among several concurrent computations.

6 Conclusion

Fine-tuning the performance of high-performance distributed applications through analytical means or simulation is hard, requiring detailed insights into the tradeoffs and effects of caching, synchronization, locality, load balancing, and communication demands.

We have proposed an approach and developed a middleware extention where different mappings of an applications communication and computations can quickly be tried out without changing the application code.

Experiments showed how we used this system to discover some of the factors con-

tributing negatively to the application performance, and then remapped the application to avoid configurations where components in the application did not scale well. We also showed how the application could be remapped to a multicluster environment without changing the application code.

The results from this work was used in [14] to benchmark PastSet using the path framework against MPI[15] (LAM-MPI), where we showed that PastSet was 83% faster than LAM-MPI on global reductions.

We believe the framework can be useful both for developing analytical models by providing information on factors relevant for analysis, and for tuning of an application where a fine-grained analysis can be difficult to attain.

By analyzing the performance results using different mappings, we have also exposed some bugs in the implementation of the application. Our approach can be useful both when debugging an application as well as finding configurations that offers improved performance.

References

[1] ANSHUS, O. J., AND LARSEN, T. Macroscope: The abstractions of a distributed operating system. *Norsk Informatikk Konferanse* (October 1992).

[2] ARPACI-DUSSEAU, A. C., ARPACI-DUSSEAU, R. H., CULLER, D. E., HELLERSTEIN, J. M., AND PATTERSON, D. A. Searching for the sorting record: Experiences in tuning NOW-Sort. *Proceedings of the SIGMETRICS symposium on Parallel and distributed tools (SPDT 98), USA* (1998), pp. 124–133.

[3] BIRRELL, A. D., AND NELSON, B. J. Implementing remote procedure calls. In *Proceedings of the ninth ACM Symposium on Operating Systems Principles* (1983).

[4] CARRIERO, N., AND GELERNTER, D. Linda in context. *Commun. ACM 32*, 4 (April 1989), 444–458.

[5] DICKENS, P., HEIDELBERGER, P., AND NICOL, D. Parallel direct execution simulation of message-passing parallel programs. *IEEE Transactions on Parallel and Distributed System* (1996).

[6] GOVIL, K., TEODOSIU, D., AND YONGQIANG HUANG AND, M. R. Cellular disco: resource management using virtual clusters on shared-memory multiprocessors. *ACM Symposium on Operating Systems Principles (SOSP'99), published in Operating Systems Review 34(5)* (December 1999), pp 154–169.

[7] JUURLINK, B. H., AND WIJSHOFF, H. A. A quantitative comparison of parallel computation models. *ACM Transactions on Computer Systems Vol. 16*, No. 3 (August 1998), pp. 271–318.

[8] KICZALES, G., DES RIVIERES, J., AND BOBROW, D. G. *The Art of the Metaobject Protocol*. MIT Press, 1991.

[9] LUO, Y. Mpi performance study on the sgi origin 2000. *Pacific Rim Conference on Communications, Computers and Signal Processing* (1997), pp 269–272.

[10] REINHARDT, S., HILL, M. D., LARUS, J., LEBECK, A., J.C, LEWIS, AND WOOD, D. The wisconsin wind tunnel: Virtual prototyping of parallel computers. *Proceedings of the 1993 ACM SIGMETRICS Conference* (May 1993).

[11] ROSENBLUM, M., BUGNION, E., DEVINE, S., AND HERROD, S. Using the simos machine simulator to study complex computer systems. *ACM*

Trans. On Modeling and Computer Simulation Vol. 7, No. 1 (January 1997), pp. 78–103.

[12] VENUGOPAL, S., AND NAIK, V. K. Effects of partitioning and scheduling sparse matrix factorization on communication and load balance. *Proceedings of the 1991 conference on Supercomputing* (1991), pp. 866–875.

[13] VINTER, B. *PastSet a Structured Distributed Shared Memory System*. PhD thesis, Tromsø University, 1999.

[14] VINTER, B., ANSHUS, O. J., LARSEN, T., AND BJØRNDALEN, J. M. Extending the applicability of software dsm by adding user redefinable memory semantics. *Parallel Computing (ParCo) 2001, Naples, Italy* (September 2001).

[15] WALKER, D. W. The design of a standard message-passing interface for distributed memory concurrent computers. In *Parallel Computing, Vol. 20*. April 1994, pp. 657–673.

[16] WILSON, L. F., AND NICOL, D. M. Experiments in automated load balancing. *Proceedings of the 10th Workshop on Parallel and Distributed Simulation (PADS '96)* (1996).

A.6 Scalable Processing and Communication Performance in a Multi-Media Related Context

Scalable Processing and Communication Performance in a Multi-Media Related Context

John Markus Bjørndalen[1], Otto J. Anshus[1]
Tore Larsen[1], Lars Ailo Bongo[1], Brian Vinter[2]

[1)] Department of Computer Science
University of Tromsø

[2)] Department of Mathematics and Computer Science
University of Southern Denmark

Abstract

The PATHS system for configuring an application on one or multiple clusters in a GRID is described and then used on three applications to demonstrate scalability with regards to processing and communication.

The PATHS system use a "wrapper" to provide a level of indirection to the actual run-time location of data. A wrapper specify where data is located, how to get there, and which protocols to use. Wrappers are also used to add or modify methods accessing data. Wrappers are specified dynamically. A "path" is comprised of one or more wrappers. Sections of a path can be shared among two or more paths.

The PATHS system is used to configure a global sum, a wind tunnel, and a video distribution application with the purpose of scaling processing performance when the number of processors increase, and scaling data distribution performance when the number of clients increase.

The performance measurements show that the PATHS system can be used to both scale the processing and communication performance.

1 Introduction

A key challenge when running distributed high performance applications is to establish mappings of processes to hosts that achieve high performance or efficient execution. Attacking this challenge requires balancing the conflicting goals of distributing threads for improved load balancing, while reducing communication and synchronization overheads. High performance and good scalability with respect to processing and communication typically requires manual fine-tuning of the mapping in order to balance the factors significantly influencing the performance.

Mappings may be specified by directives in the application source-code, or determined by communication libraries, middleware or the operating system. For the purpose of this paper, and due to space restrictions, we will focus on static mappings given to the application by the programmer.

In [2] three categories of useful tools were found when tuning the performance of NOW-Sort, a parallel disk-to-disk sorting algorithm on a cluster system: tools that help set expectations and configure the application to different hardware parameters, visualization tools that animate performance counters, and search tools that track down performance anomalies. We are developing similar tools.

This paper describes some components on our work on a middleware extension inspired by method combination[10] and remote procedure calls[3] which allows the communication topology to be directed by specifying meta-code and meta-data, without introducing any modifications to the application code. The extended middleware also allows computations to be specified and executed along the access paths to data. For now, we assume that the application under study runs alone on all hosts in the system; i.e. there are no other applications competing for host resources. The goal of the mapping is to achieve high performance for a particular application having exclusive access to all resources.

We provide tools for specifying and experimenting with load-time mappings, and demonstrate how one through a few experiments may identify flexible location policies achieving high performance and good scalability.

Section 2 describes how wrappers are used and combined to specify and identify access paths. Section 3 present the applications we have used to experiment with, section

4 describes our experiments using three different clusters located at the University of Tromsø, Norway and the University of Southern Denmark, Odense. Section 5 presents related work, and section 6 presents our conclusions.

2 PATHS: Configurable Orchestration and Mapping

Our research platform is PastSet[1][15], a structured distributed shared memory system in the tradition of Linda[7]. A PastSet *element* is a sequence of tuples that are of the same or equivalent type. Tuples can be read from and written to the element using the *move* and *observe* operations. Each element is globally accessible by specifying the element's name.

PATHS[4], is an extension of PastSet that allows for mapping of processes to hosts at load time, selection of physical communication paths to each element, and distribution of computations along the path. Figure 1 shows an example where a path is created between a thread and an element. A thread only references the toplevel stage in the path, and invokes operations through that stage.

Paths can be joined (forming a tree structure) to amortize communication overhead. As an example, figure 2 shows partial sums being computed in each node before being forwarded to the server containing the element.

A thread may use multiple paths to the same element, each of which can be specified and built dynamically. Each stage in the path is implemented using a *Wrapper* of a given type.

Since only leaf wrappers are referenced in the application source code, applications can be mapped onto arbitrary cluster configurations without changing source code or recompiling. Fitting and optimizing applications to any particular configuration is instead done by changing path-specifying meta-data and code.

The wrappers are partly inspired by the PastSet X-functions[16], which are operation modifiers specified by the programmer and associated with specific elements to modify the semantics of the elements operations.

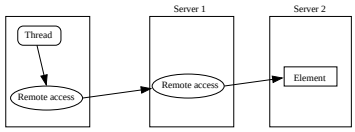


Figure 1. A path between a thread and an element

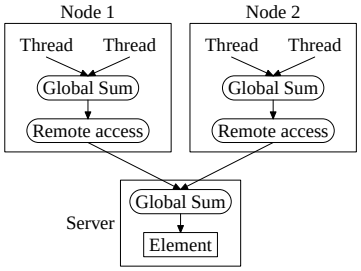


Figure 2. Four threads on two hosts accessing a shared global sum element on a separate server. Each host computes a partial sum that is forwarded to the global-sum wrapper on the server.

2.1 Building and specifying paths

After deciding on a mapping of processes onto hosts, setting up access from one thread to a PastSet element involves the following two stages:

Specify the path. This involves examining information such as process mappings, cluster topologies, and location of target elements.

Build the path from the description. This involves creating and binding wrappers with parameters specified in the path description.

Each wrapper in the path description is parameterized. Some parameters are common among all wrappers (such as whether the wrapper need to use thread synchronization mechanisms), whereas others are provided only for specific wrapper types (i.e. what protocol to use, server address, and service requirements for remote access wrappers).

An example path description used by one of the nodes in Figure 2 is included in Figure 3.

Each thread creates (or is given) its path description, before calling `build_path` which returns reference to the top-level wrapper in the path. `Build_path` checks path descriptions, and merges paths when it is feasible to share portions of the paths.

Profiling is provided via trace wrappers that log the start and completion times of operations that are invoked through each wrapper. The trace wrappers may also be used to store information about operation parameters, and tuples provided and returned in the operation. Any number of trace wrappers may be inserted anywhere in the path trees.

```
path = make_path(stage("reduce-sum", num_threads=2),
                 stage("remote", proto=TCP, host="p0"),
                 stage("reduce-sum", num_threads=2),
                 stage("core", name="PI-SUM1"))
elm = build_path(path)
```

Figure 3. Example path specifictation and building in Python

Traces stored with the corresponding path specifications are later read by specialized tools to examine the performance aspects of an application.

3 Applications

The applicability of PATHS is tested using three parallel applications, Global sum, Wind tunnel, and Video.

Global sum computes a global sum that is aggregated from parallel threads providing partial sums. The algorithm works essentially as the MPI *Allreduce* function. Global Sum is used in experimenting with the effects of thread mappings, communication parameters, and hierarchal reduction of the latencies of global reductions. In Section 4.2 we report our findings from experimenting with a hierarchal global reduction to reduce latencies. Earlier experiments using Global sum can be found in [4].

The **Wind tunnel** application is a Lattice Gas Automaton doing particle simulation. It uses eight matrices in which particles are shifted around to simulate the flow of air. The parallel version splits each matrix into slices, which are then assigned to threads. When running, each thread exchanges the border entries of its slices with threads computing on neighboring slices. We demonstrate how we can map the application to different cluster configurations and change the mapping to improve the scaling and performance of the system.

The **Video** application demonstrates parallel distribution of one video feed. A feeder application captures images from a frame grabber, converts each image into a jpeg, and stores each jpeg as a tuple in a PastSet element. Each video client uses the *last-observe* wrapper to retrieve the latest available new jpeg. If no jpeg exists that is newer than the ones already observed, last-observe blocks until one arrives. A client program decompresses the images and displays them in a TKinter window.

Organizing the PastSet servers and paths hierarchically, we demonstrate that no jpeg image needs to be transmitted more than once down any wire, that clients which cannot consume images at full speed only retrieves the latest available image (and skip older images), and that faster clients pre-fetch images for the slower clients; essentially turning the last-observe wrapper into a cache.

4 Experiments

Experiments were done using Global sum and Wind tunnel, to study the ability of PATHS to improve processing performance by re-configuring applications at load time. Additional experiments were done using the Video application to study the ability of PATHS to provide scalable distribution of one video-feed.

Wrappers were used to extract performance data which were visualized using our prototype performance data tools[5].

We are now experimenting with the Video Distribution system using the Wind Tunnel application output with the purpose of prototyping a multimedia application scaling well both with regards to computation and data distribution.

4.1 Hardware Platform

The hardware platform consists of three geographically dispersed clusters, each with 32 processors:

- 2W: 16*2-Way Pentium III 450 MHz, 256MB RAM (Odense, Denmark).
- 4W: 8*4-Way Pentium Pro 166 MHz, 128MB RAM, (Tromsø, Norway).
- 8W: 4*8-Way Pentium Pro 200 MHz, 2GB RAM, (Tromsø, Norway).

All clusters run intra-cluster communication using TCP/IP over 100 Mbps Ethernet. 4W has an additional internal 100 Mbps 100 VG-AnyLAN connection that is used for some experiments. Communication between 4W and 8W used a 100 Mbps 100 VG AnyLan connection. All communication between the cluster in Odense (2W) and the two clusters in Tromsø (4W and 8W), uses institutional resources at the respective sites, each country's national research and educational backbone, and the Nordic interconnection of national research networks (NORDUnet). Using the PastSet system, the latency between two threads on different nodes using the 100 Mbps networks is typically around 100-250 microseconds depending on the systems and protocols used. Using the network between Tromsø and Odense, the latency is typically around 40 milliseconds.

4.2 Global Sum Experiments

In the Global sum experiment we measured the average execution time of a global sum computation. The number of threads applied is equal to the number of values to be added. The Global sum experiment is run on 2W only.

```
barrier_sync();
gettimeofday();
TS(0);
for (i = 1; i= < iters; i++)
{
     sum = gsum(i);
     TS(i);
}
gettimeofday();
```

Figure 4. Pseudocode for the Global Sum benchmark. Only one thread runs the timestamp code, all other threads run the same code with the timestamp code removed.

Figure 4 shows pseudocode for the Global sum benchmark. The *TS()* macro samples the Pentium Pro timestamp counter, and stores the timestamp in an array. *Gettimeofday()* samples the real-time clock on the host computer with microsecond resolution. The gsum benchmark was run with an "iters" of 1.000. For each test, the average execution time of five runs is plotted on the graphs.

In one experiment we measured the performance using the *even distribution* algorithm. Using this algorithm, threads are distributed as evenly as possible among all hosts in the cluster; i.e. the number of threads on any two hosts differs by at most one. One single thread computes the aggregate of the partial sums computed by every other thread. More elaborate algorithms are investigated in[4].

In [4] we observed that increasing the number of threads per host beyond three or four would reduce the computational efficiency of each host. This observation suggests that using four threads per sum wrapper and arranging the threads and wrappers hierarchically as shown in 5 gives an organization that maps easily onto the cluster, while limiting the number of threads per wrapper to four. Compared to a flat organization, the hierarchical organization offers potential performance improvements due to a higher degree of parallelism.

Figure 6 shows a plot of the execution times using the hierarchical and even organizations as the number of threads increases from one up to the number of CPUs in the cluster. The number of values to be added is increased linearly for each thread added.

Using hierarchical organization, we measure double or

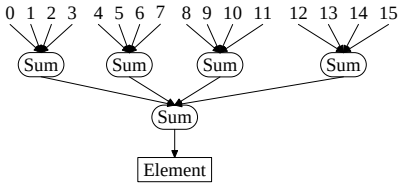
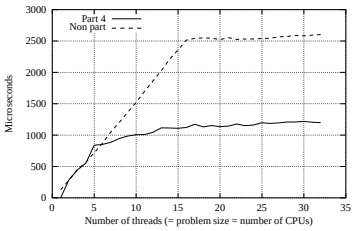


Figure 5. Hierarchial global reduction sum tree. The numbers represent threads. The upper layer of sum wrappers computes partial sums used in the lowermost sum wrapper.



(a) 2-Way cluster

Figure 6. Global sum, cluster partition, even distribution

better performance for twelve or more threads, compared to the even distribution algorithm. Observations from [4] were applied in [16] when comparing the performance characteristics of PastSet using the path framework with MPI[17] (LAM-MPI). Results demonstrated that PastSet was 1.83 times faster than LAM-MPI on global reductions. We are currently experimenting with a PATHS-inspired configuration mechanism for LAM-MPI, and preliminary results show that we are able to significantly improve some group operations.

4.3 Wind Tunnel Experiments

We used the Wind Tunnel benchmark to evaluate whether PATHS could be used effectively to experiment with different mappings and identify mappings that scaled well.

Using an even distribution of threads to CPUs, we experienced linear speedups for this application on every cluster by itself, and when combining both clusters in Tromsø (4W

and 8W). Combining the Odense cluster (2W) with any of the Tromsø clusters gave us less than linear speedups.

From an initial even mapping of one thread per 80th Mhz, we used PATHS to quickly set up several variant mappings that we experimented with. For these experiments, each thread carries the same effective workload regardless of mapping.

- Increasing the relative workload at 2W by evenly increasing the number of threads from 12 to 14, reduced the performance.
- Decreasing the relative workload at 2W by evenly decreasing the number of threads from 12 to 10, had no performance effect.
- Decreasing the relative workload at 2W further, by evenly decreasing the number of threads to eight reduced the performance.
- Reducing the load on the single node on 2W that handled all communication external to the cluster, slightly improved the performance.
- Reducing the number of sequential remote operations by moving data to 2W had no performance effect.
- Increasing the problem size to get a higher processing to communication ratio, resulted in improved performance. Effectively scaling our experiments along this approach was limited by the memory size (128 MB) of each host in 4W.

Using a prototype PATHS performance data visualization tool revealed that after some time, the progression of every thread in the clusters was effectively reigned in by the progression of the 2W inter cluster communication node. This phenomenon will be investigated further.

4.4 Video Distribution System

The Video Distribution experiment was designed to investigate if PastSet and PATHS could be applied in quickly developing a video broadcast system where each client that requests to receive the broadcast, receives its feed at a rate it can process, and without loading the server and network by delivering frames that would otherwise be skipped by the client.

The video broadcast system was configured to have one dedicated root node on each cluster. (see Figure 8). Every participating client thread has a path that goes through a host-local PastSet server, on through the root node in the cluster, and finally to the broadcast server that holds the original video data.

The last-observe wrapper is used to cache frames pulled down from a server further up in the hiearchy. A new frame

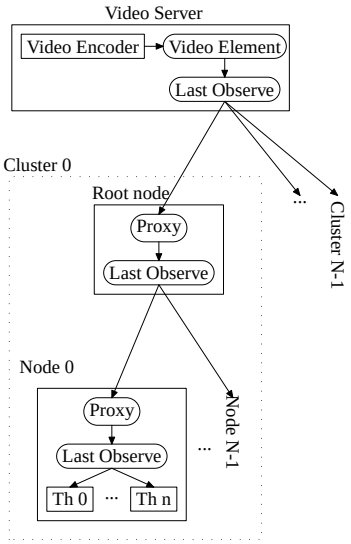


Figure 8. Hierarchial Video Application

is only requested from a server further up when the last-observe wrapper does not have a frame as new as, or newer than the one being requested by a client further down. Only the latest available frame will be returned by the wrapper. Older frames are dropped.

To test the scalability of this design, we started with one client per processor in the clusters and dynamically increased the number of clients while monitoring the frame rate at the server process and at every client process. The server process load was also monitored. The broadcast server was located in Tromsø, but external to 4W and 8W. The stream of jpegs consisted of 320x240 pixel images, delivered at 12.5 Hz. The client processes were evenly mapped across 2W, 4W, and 8W.

At 960 clients there was no noticeable degradation on the server or any client. Every client, including the 320 at 2W in Odense, ran at the full frame rate at 12.5 Hz.

At 2016 client processes, there was still no noticeable degradation of the server or of the processes on 2W in Odense. At this load however, the clients running on the clusters in Tromsø (next room to the video server) did exhibit slight degradation, dropping the frame rate to about 12 Hz on 4W, and about 10-11 Hz on 8W.

We observe that of the two clusters in Tromsø, processes at 4W (166MHz four-way nodes) perform better than processes at 8W (200MHz eight-way nodes) when the number

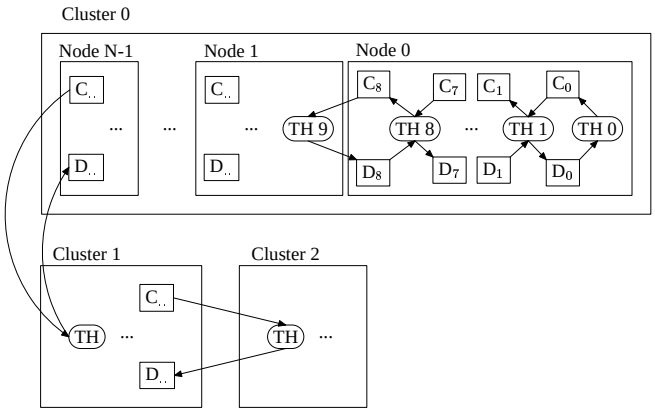


Figure 7. Multi-cluster Wind tunnel Experiment

of client processes is increased beyond a threshold level. We suspect this indicates processor-memory bus contention on the eight-way systems due to the high communication load of this benchmark. Processes at the faster 2W cluster perform best, and are not hindered by the slow Tromsø-Odense connection.

5 Related work

Accurate and efficient performance prediction of existing distributed and parallel applications on target configurations with potentially thousands of processors is hard. A number of simulators have been developed, including Parallel Proteus[11], LAPSE[8], SimOS[13], and Wisconsin Wind Tunnel[12]. The slowdowns range from 2 to 100. Few simulators simulate both computation and I/O operations. In contrast to simulators, our approach execute the actual application code several times, each time with a different mapping.

In [9] both processor and memory load balancing are used to support low contention and good scaling to hundreds of processors. Gang-scheduling is used to avoid wasting cycles spinning for a lock held by a descheduled process. In contrast, our system is much simpler and provides for much less or no automatic support at the present time.

In [14] it is shown that there is a communication and load balance trade-off when partitioning and scheduling sparse matrix factorization on distributed memory systems. Block based methods result in lower communication costs and worse load balancing, whereas a "round robin"-based scheme where all threads are distributed over the processors gives better load balance but higher communication costs.

In [18] an approach to load balancing for general-purpose simulations is reported in which little modification is needed to the user's code. Their approach uses run-time measurements and demonstrates better load-balancing than approaches without such measurements. This approach is similar to ours in that little modification of the user's code is needed. As they do, we also use different mappings and leave it to the application to control them. Our approach differ in that we can both try different mappings and add arbitrary code along the access path to data. Also, we differ in that we do a prerun of a few mappings, and then we choose a single one and we let the application use the selected mapping without incurring further overhead. Of course, we take all the overhead when choosing a mapping. For clusters where they can be dedicated to applications running often, this configuration hunting overhead will be amortizised over time.

In [6] it is shown that dramatic reductions in the bandwidth demand on the underlying server operating system can be gained via application-level data caching. This is in accordance with our work.

6 Conclusion

Fine-tuning the performance of high-performance distributed applications through analytical means or simulation is hard, requiring detailed insights into complicated factors including the tradeoffs and effects of caching, synchronization, locality, load balancing, communication demands, and how network protocols and synchronization mechanisms

have been implemented. Using the PATHS system different mappings of an applications communication and computations can quickly be tried out without changing the application code.

Through several experiments we discovered some of the factors contributing negatively to the performance of a set of applications, and then we remapped the applications to find configurations with better performance.

We believe that our system can improve performance by finding a configuration where the resource usage better avoids hot spots, bottlenecks, and expensive waiting times for processor, memory, cache, and I/O by trading between load sharing and communication. The flexibility of using maps, paths and wrappers also make it possible to monitor the application and provide data for visualization of both behaviour and performance.

References

[1] ANSHUS, O. J., AND LARSEN, T. Macroscope: The abstractions of a distributed operating system. *Norsk Informatikk Konferanse* (October 1992).

[2] ARPACI-DUSSEAU, A. C., ARPACI-DUSSEAU, R. H., CULLER, D. E., HELLERSTEIN, J. M., AND PATTERSON, D. A. Searching for the sorting record: Experiences in tuning NOW-Sort. *Proceedings of the SIGMETRICS symposium on Parallel and distributed tools (SPDT 98), USA* (1998), pp. 124–133.

[3] BIRRELL, A. D., AND NELSON, B. J. Implementing remote procedure calls. In *Proceedings of the ninth ACM Symposium on Operating Systems Principles* (1983).

[4] 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 application. *Norsk Informatikk Konferanse* (November 2001), 164–175.

[5] BONGO, L. A. Steps: A performance monitoring and visualization tool for multicluster parallel programs, June 2002. Large term project, Department of Computer Science, University of Tromsø.

[6] BRADSHAW, M. K., WANG, B., SEN, S., GAO, L., KUROSE, J., SHENOY, P., AND TOWSLEY, D. Periodic broadcast and patching services - implementation, measurement, and analysis in an internet streaming video testbed*. ACM MM'01, Ottawa, Canada.

[7] CARRIERO, N., AND GELERNTER, D. Linda in context. *Commun. ACM 32*, 4 (April 1989), 444–458.

[8] DICKENS, P., HEIDELBERGER, P., AND NICOL, D. Parallel direct execution simulation of message-passing parallel programs. *IEEE Transactions on Parallel and Distributed System* (1996).

[9] GOVIL, K., TEODOSIU, D., AND YONGQIANG HUANG AND, M. R. Cellular disco: resource management using virtual clusters on shared-memory multiprocessors. *ACM Symposium on Operating Systems Principles (SOSP'99), published in Operating Systems Review 34(5)* (December 1999), pp 154–169.

[10] KICZALES, G., DES RIVIERES, J., AND BOBROW, D. G. *The Art of the Metaobject Protocol*. MIT Press, 1991.

[11] LUO, Y. Mpi performance study on the sgi origin 2000. *Pacific Rim Conference on Communications, Computers and Signal Processing* (1997), pp 269–272.

[12] REINHARDT, S., HILL, M. D., LARUS, J., LEBECK, A., J.C, LEWIS, AND WOOD, D. The wisconsin wind tunnel: Virtual prototyping of parallel computers. *Proceedings of the 1993 ACM SIGMETRICS Conference* (May 1993).

[13] ROSENBLUM, M., BUGNION, E., DEVINE, S., AND HERROD, S. Using the simos machine simulator to study complex computer systems. *ACM Trans. On Modeling and Computer Simulation Vol. 7*, No. 1 (January 1997), pp. 78–103.

[14] VENUGOPAL, S., AND NAIK, V. K. Effects of partitioning and scheduling sparse matrix factorization on communication and load balance. *Proceedings of the 1991 conference on Supercomputing* (1991), pp. 866–875.

[15] VINTER, B. *PastSet a Structured Distributed Shared Memory System*. PhD thesis, Tromsø University, 1999.

[16] VINTER, B., ANSHUS, O. J., LARSEN, T., AND BJØRNDALEN, J. M. Extending the applicability of software dsm by adding user redefinable memory semantics. *Parallel Computing (ParCo) 2001, Naples, Italy* (September 2001).

[17] WALKER, D. W. The design of a standard message-passing interface for distributed memory concurrent computers. In *Parallel Computing, Vol. 20*. April 1994, pp. 657–673.

[18] WILSON, L. F., AND NICOL, D. M. Experiments in automated load balancing. *Proceedings of the 10th Workshop on Parallel and Distributed Simulation (PADS '96)* (1996).

A.7 Configurable Collective Communication in LAM-MPI

Communicating Process Architectures – 2002
James Pascoe, Peter Welch, Roger Loader and Vaidy Sunderam (Eds.)
IOS Press, 2002

Configurable collective communication in LAM-MPI

John Markus Bjørndalen[1], Otto J. Anshus[1]
Brian Vinter[2], Tore Larsen[1]

[1)] *Department of Computer Science, University of Tromsø*
[2)] *Department of Mathematics and Computer Science, University of Southern Denmark*

Abstract.

In an earlier paper, we observed that PastSet (our experimental tuple space system) was 1.83 times faster on global reductions than LAM-MPI. Our hypothesis was that this was due to the better resource usage of the PATHS framework (an extension to PastSet that supports orchestration and configuration) due to a mapping of the communication and operations which matched the computing resources and cluster topology better.

This paper reports on an experiment to verify this, and represents an ongoing work to add some of the same configurability of PastSet and PATHS to MPI.

We show that by adding run-time configurable collective communication, we can reduce the latencies without recompiling the application source code. For the same cluster where we experienced the faster PastSet, we show that Allreduce with our configuration mechanism is 1.79 times faster than the original LAM-MPI Allreduce.

We also experiment with the configuration mechanism on 3 different cluster platforms with 2-, 4-, and 8-way nodes. For the cluster of 8-way nodes, we show an improvement by a factor of 1.98 for Allreduce.

1 Introduction

For efficient support of synchronization and communication in parallel systems, these systems require fast collective communication support from the underlying communication subsystem as, for example, is defined by the Message Passing Interface (MPI) Standard [1]. Among the set of collective communication operations broadcast is fundamental and is used in several other operations such as barrier synchronization and reduction [2]. Thus, it is advantageous to reduce the latency of broadcast operations on these systems.

In our work with the PATHS[3] configuration and orchestration system, we have experimented with microbenchmarks and applications to study the effects of configurable communication.

In one of the experiments[4], we used the configuration mechanisms to reduce the execution times of collective communication operations in PastSet. To get a baseline, we compared our reduction operation with the equivalent operation in MPI (Allreduce).

By trying a few configurations, we found that we could improve our Tuple Space system to be 1.83 times faster than LAM-MPI[5][6]. Our hypothesis was that this advantage came from a better usage of resources in the cluster rather than a more efficient implementation.

If anything, LAM-MPI should be faster than PastSet since PastSet stores the results of each global sum computation in a tuple space inducing more overhead than simply computing and distributing the sum.

This paper reports on an experiment where we have added configurable communication to the Broadcast and Reduce operations in LAM-MPI (both of which are used by Allreduce) to validate or falsify our hypothesis.

The paper is organized as follows: Section 2 summarizes the main features of the PastSet and PATHS system. Section 3 describes the Allreduce, Reduce and Broadcast operations in LAM-MPI. Section 4 describes the configuration mechanism that was added to LAM-MPI for the experiments reported on in this paper. Section 5 describes the experiments and results, section 6 presents related work, and section 7 concludes the paper.

2 PATHS: Configurable Orchestration and Mapping

Our research platform is PastSet[7][8], a structured distributed shared memory system in the tradition of Linda[9]. PastSet is a set of Elements, where each Element is an ordered collection of tuples. All tuples in an Element follow the same template.

The PATHS[3] system is an extension of PastSet that allows for mappings of processes to hosts at load time, selection of physical communication paths to each element, and distribution of communications along the path. PATHS also implements the X-functions[4], which are PastSet operation modifiers.

A path specification for a single thread needing access to a given element is represented by a list of stages. Each stage is implemented using a wrapper object hiding the rest of the path after that stage. The stage specification includes parameters used for initialisation of the wrapper.

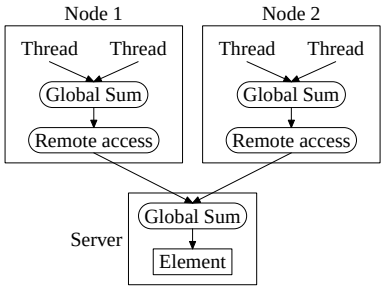


Figure 1: Four threads on two hosts accessing shared element on a separate server. Each host computes a partial sum that is forwarded to the global-sum wrapper on the server. The final result is stored in the element.

Paths can be shared whenever path descriptions match and point to the same element (see figure 1). This can be used to implement functionality such as, for instance, caches, reductions and broadcasts.

The collection of all paths in a system pointing to a given element forms a tree. The leaf nodes in the tree are the application threads, while the root is the element.

Figure 1 shows a global reduction tree. By modifying the tree and the parameters to the wrappers in the tree, we can specify and experiment directly with factors such as which processes participate in a given partial sum, how many partial sum wrappers to use, where each sum wrapper is located, protocols and service requirements for remote operations and where the root element is located. Thus, we can control and experiment with tradeoffs between placement of computation, communication, and data location.

Applications tend to use multiple trees, either because the application uses multiple elements, or because each thread might use multiple paths to the same element.

To get access to an element, the application programmer can either choose to use lower-level functions to specify paths before handing it over to a path builder, or use a higher level function which retrieves a path specification to a named element and then builds the specified path. The application programmer then gets a reference to the topmost wrapper in the path.

The path specification can either be retrieved from a combined path specification and name server, or be created with a high-level language library loaded at application load-time[1]

Since the application program invokes all operations through its reference to the topmost wrapper, the application can be mapped to different cluster topologies simply by doing one of the following:

- Updating a map description used by the high-level library.
- Specifying a different high-level library that generates path-specifications. This library may be written by the user.
- Update the path mappings in the name server.

Profiling is provided via trace wrappers that log the start and completion time of operations that are invoked through it. Any number of trace wrappers can inserted anywhere in the path.

Specialized tools to examine the performance aspects of the application can later read trace data stored with the path specifications from a system. We are currently experimenting with different visualizations and analyses of this data to support optimization of a given application.

The combination of trace data, a specification of communication paths, and computations along the path has been useful in understanding performance aspects and tuning benchmarks and applications that we have run in cluster and multicluster environments.

3 LAM-MPI implementation of Allreduce

LAM-MPI (Local Area Multicomputer) is an open source implementation of MPI available from [5]. It was chosen over MPICH[10] for our work in [4] since it had lower latency with less variance than MPICH for the benchmarks we used in our clusters.

The MPI Allreduce operation combines values from all processes and distribute the result back to all processes. LAM-MPI implements Allreduce by first calling Reduce, collecting the result in the root process, then calling Broadcast, distributing the result from the root process. For all our experiments, the root process is the process with rank 0 (hereafter called process 0).

The Reduce and Broadcast algorithms use a linear scheme (every process communicates directly with process 0) up to and including 4 processes. From there on they use a scheme that organizes the processes into a logarithmic spanning tree.

The shape of this tree is fixed, and doesn't change to reflect the topology of the computing system or cluster. Figure 2 shows the reduction trees used in LAM-MPI for 32 processes in a cluster. We observe that broadcast and reduction trees are different.

By default, LAM-MPI evenly distributes processes onto nodes. When we combine this mapping for 32 processes with the reduction tree, we can see in Figure 3 that a lot of messages are sent across nodes in the system. The broadcast operation has a better mapping for this cluster though.

[1] Currently, a Python module is loaded for this purpose.

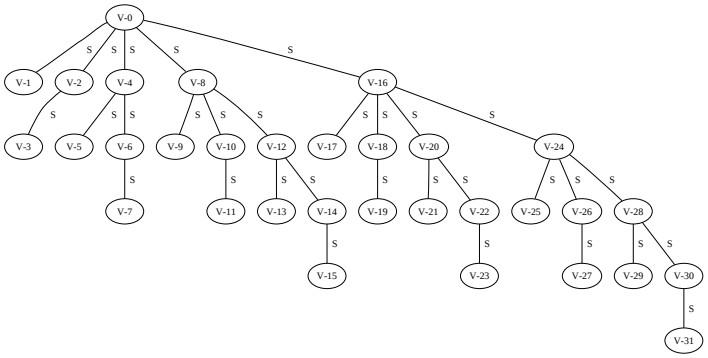


Figure 2: Log-reduce tree for 32 processes. The arcs represent communication between two nodes. Partial sums are computed at a node in the tree before passing the result further up in the tree.

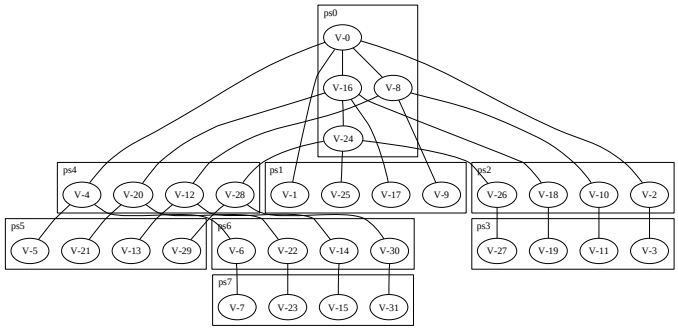


Figure 3: Log-reduce tree for 32 processes mapped onto 8 nodes.

4 Adding configuration to LAM-MPI

To minimize the necessary changes to LAM-MPI for this experiment, we didn't add a full PATHS system at this point. Instead, a mechanism was added that allowed for scripting the way LAM-MPI communicates during the broadcast and reduce operations.

There were two main reasons for this. Firstly, our hypothesis was that PastSet with PATHS allowed us to map the communication and computation better to the resources and cluster topology. For global reduction and broadcast, LAM-MPI already computes partial sums at internal nodes in the trees. This means that experimenting with different reduction and broadcast trees should give us much of the effect that we observed with PATHS in [4] and [3].

Secondly, we wanted to limit the influence that our system would have on the performance aspects of LAM-MPI such that any observable changes in performance would come from modifying the reduce and broadcast trees.

Apart from this, the amount of code changed and added was minimal, which reduced the chances of introducing errors into the experiments.

When reading the LAM-MPI source code, we noticed that the reduce operation was, for

any process in the reduction tree, essentially a sequence of N receives from the N children directly below it in the tree, and one send to the process above it. For broadcast, the reverse was true; one receive followed by N sends.

Using a different reduction or broadcast tree would then simply be a matter of examining, for each process, which processes are directly above and below it in the tree and construct a new sequence of send and receive commands.

To implement this, we added new reduce and broadcast functions which used the rank of the process and size of the system to look up the sequence of sends and receives to be executed (including which processes to send and receive from). This is implemented by retrieving and executing a script with send and receive commands.

As an example, when using a scripted reduce operation with a mapping identical to the original LAM-MPI reduction tree, the process with rank 12 (see figure 2) would look up and execute a script with the following commands:

- Receive (and combine result) from rank 13
- Receive (and combine result) from rank 14
- Send result to 8

The new scripted functions are used instead of the original logarithmic Reduce and Broadcast operations in LAM-MPI. No change was necessary to the Allreduce function since it is implemented using the reduce and broadcast operations.

The changes to the LAM-MPI code was thus limited to 3 code lines, replacing the calls to the logarithmic reduction and broadcast functions as well as adding and a call in MPI_Init to load the scripts.

Remapping the application to another cluster configuration, or simply trying new mappings for optimization purposes, now consists of specifying new communication trees and generating the scripts. A Python program generates these scripts as Lisp symbolic expressions.

5 Experiments

```
t1 = get_usecs();
MPI_Allreduce(&hit, &ghit, 1, MPI_INT, MPI_SUM, MPI_COMM_WORLD);
t1 = get_usecs();
for (i = 0; i < ITERS; i++) {
   MPI_Allreduce(&i, &ghit, 1, MPI_INT, MPI_SUM, MPI_COMM_WORLD);
   if (ghit != (i * size))
      printf("oops at %d. %d != %d\n", i, ghit, (i * size));
}
t2 = get_usecs();
MPI_Allreduce(&hit, &ghit, 1, MPI_INT, MPI_SUM, MPI_COMM_WORLD);
```

Figure 4: Global reduction benchmark

Figure 4 shows the code run in the experiments. The code measures the average execution time of 1000 Allreduce operations. The average of 5 runs is then plotted. To make sure that the correct sum is computed, the code also checks the result on each iteration.

For each experiment, the number of processes was varied from 1 to 32. LAM-MPI used the default placement of processes on nodes, which evenly spread the processes over the nodes.

The hardware platforms consists of three clusters, each with 32 processors:

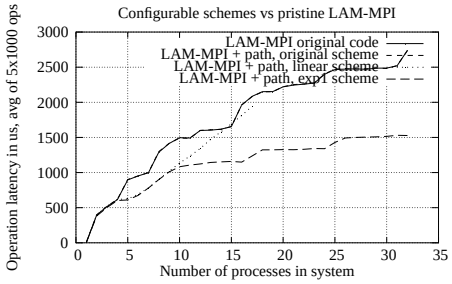


Figure 5: Allreduce, 4-way cluster

- 2W: 16*2-Way Pentium III 450 MHz, 256MB RAM
- 4W: 8*4-Way Pentium Pro 166 MHz, 128MB RAM
- 8W: 4*8-Way Pentium Pro 200 MHz, 2GB RAM

All clusters used 100MBit Ethernet for intra-cluster communication.

5.1 4-way cluster

Figure 5 shows experiments run on the 4-way cluster. The following experiments were run:

Original LAM-MPI The experiment was run using LAM-MPI 6.5.6.

Modified LAM-MPI, original scheme The experiment was run using a modified LAM-MPI, but using the same broadcast and reduce trees for communication as the original version used.

This graph completely overlaps the graph from the original LAM-MPI code, showing us that the scripting functionality is able to replicate the performance aspects of the original LAM-MPI for this experiment.

Modified LAM-MPI, linear scheme This experiment was run using a linear scheme where all processes report directly to the process with rank 0, and rank 0 sends the results directly back to each client.

This experiment crashed at 18 processes due to TCP connection failures in LAM-MPI. We haven't examined what caused this, but it is likely to be a limitation in the implementation.

Modified LAM-MPI, hierarchal scheme 1 Rank 0 is always placed on node 0, so all other processes on node 0 contribute directly to rank 0. On the other nodes, a local reduction is done within each node before the nodes partial sum is sent to process 0.

The modified LAM-MPI with the hierarchal scheme is 1.79 times faster than the original LAM-MPI. This is close to the factor of 1.83 reported in [4], which was based on measurements on the same cluster. It is possible that further experiments with configurations would have brought us closer to the original speed improvement.

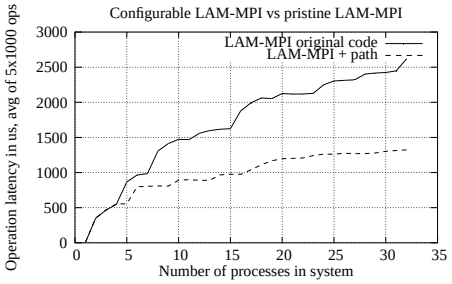


Figure 6: Allreduce, 8-way cluster

5.2 8-way cluster

For the 8-way cluster, we expected the best configuration to involve using an internal root process on each node and limit the inter-node communications to one send (of the local sum) and one receive (of the global result) resembling the best configuration for the 4-way cluster. This turned out not to be true.

Due to the 8 processes on each node, the local root process on each node would get 7 children in the sum tree (with the exception of the root processes in the root node). The extra latency added per child of the local root processes was enough to increase the total latency for 32 processes to 1444 microseconds.

Splitting the processes to allow further subgrouping within each node introduced another layer in the sum tree. This extra layer brought the total latency for 32 processes to 1534 microseconds.

The fastest solution found so far involves partitioning the processes on each node into two groups of 4 processes. Each group computes a partial sum for that group, and forwards it to directly to the root process. This doubled the communication over the network, but the total execution time was shorter (1322 microseconds).

The fastest configuration is 1.98 times faster than the original LAM-MPI Allreduce operation. In figure 6, the fastest configuration and the original LAM-MPI Allreduce are plotted for 1-32 processes.

5.3 2-way cluster

On the 2-way cluster, after trying a few configurations, we ended up with a configuration that was 1.52 times faster than the original LAM-MPI code. This was not as good as for the other clusters. We expected this since a reduction in this cluster would need more external messages.

Based on the experiments, we observed three main factors that influenced the scaling of the application:

- The number of children for a process in the operation tree. More children adds communication load for the node hosting this process.
- The depth of a branch in the operation tree. That is, the number of nodes a value has to be added through before reaching the root node.
- Whether a message was sent over the network or internally on a node.

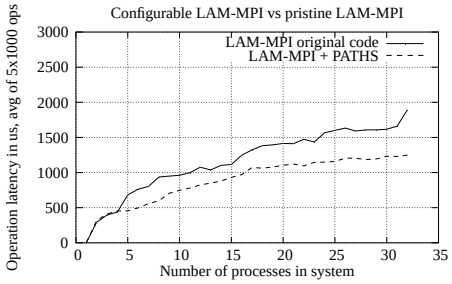


Figure 7: Allreduce, 2-way cluster.

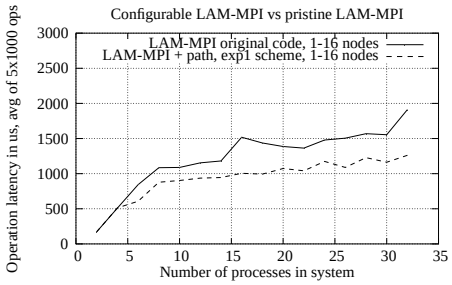


Figure 8: Allreduce, 2-way cluster, scaling the number of nodes from 1 to 16. Each node added adds two new processes in the experiment.

The depth factor seemed to introduce a higher latency than adding a few more children to a process. On the other hand, adding another layer in the tree would also allow us to use partial sums. We have not arrived at any model which allows us predict which tree organization would in practice lead to the shortest execution time of the Allreduce operation.

One difference between the PastSet/PATHS implementation and LAM-MPI is that PastSet/PATHS process incoming partial sums by order of arrival, while LAM-MPI process them in the order they appear in the sequence specified either by the original LAM-MPI code, or the script. This might influence the overhead when receiving messages from a larger number of children. We are investigating this further.

5.4 Scaling up the number of nodes on the 2-way cluster

We also scaled the 2-way cluster along another axis: the number of available nodes for the application.

The motivation for this is that clusters do not necessarily have a power of two nodes, such as the clusters we used in the previous sections[2].

A given algorithm for building an operation tree might result in a good mapping for one cluster configuration, but result in a less efficient operation on another cluster configuration.

In this experiment, we scaled the number of nodes from 1 to 16. The number of processes was scaled with the number of nodes, adding two processes for each node added to the

[2] either because of lack of funding, broken nodes, or nodes allocated for other purposes

experiment.

Figure 8 shows an experiment where we compare the original LAM-MPI mappings with a setting where we reconfigure the best algorithm from section 5.3 to fit with a different sized cluster. We generally outperform the original LAM-MPI algorithm for all cluster sizes larger than 2 in this experiment. At 16 nodes and 32 processes, we end up with the same cluster configuration and the same latencies as in the previous section.

6 Related work

Jacunski et al.[11] shows that selection of the best performing algorithm for all-to-all broadcast on clusters of workstations based on commodity switch-based networks is a function of both network characteristics as well as the message length and number of participating nodes in the all-to-all broadcast operation. In contrast our work used clusters with multiprocessors, we did performance measurements on the reduce operation as well as on the broadcast, and we documented the effect of the actual system at run time including the workload, communication performance of the hosts.

Bernashci et al.[12] study the performance of the MPI broadcast operation on large shared memory systems using a-trees.

Kielmann et al.[13] show how the performance for collective operations like broadcast depend upon cluster topology, latency, and bandwidth. They develop a library of collective communication operations optimized for wide area systems where a majority of the speedup comes from limiting the number messages passing over wide-area links.

Husbands et al.[14] optimizes the performance of MPI_Bcast in clusters of SMP nodes by using a two-phase scheme; first messages are sent to each SMP node, then messages are distributed within the SMP nodes. Sistare et al.[15] uses a similar scheme, but focus on improving the performance of collective operations on large-scale SMP's.

Tang et al.[16] also use a two-phase scheme for a multithreaded implementation for MPI. They separate the implementation of the point-to-point and collective operations to allow for optimizations of the collective operations, which would otherwise be difficult.

In contrast to the above works, this paper is not focused on a particular optimization of the spanning trees. Instead, we focus on making the shape of the spanning trees configurable to allow easy experimentation on various cluster topologies and applications.

Vadhiyar et al.[17] shows an automatic approach to selecting buffer sizes and algorithms to optimize the collective operation trees by conducting a series of experiments.

7 Conclusions

We have observed that the broadcast and reduction trees in LAM-MPI are different, and do not necessarily take into account the actual topology of the cluster.

By introducing configurable broadcast and reduction trees, we have shown a simple way of mapping the reduction and broadcast trees to the actual clusters in use. This gave us a performance improvement up to a factor of 1.98.

For the cluster where we observed the performance difference of a factor 1.83 between PastSet/PATHS and LAM-MPI, we arrived at a reduction and broadcast tree that gave us an improvement of 1.79 for Allreduce over the original LAM-MPI implementation. This supports our hypothesis that the majority of the performance-difference between LAM-MPI and PastSet/PATHS was a better mapping to the resources and topology in the cluster.

We have also observed that the assumption that doing a reduction internally in each node before sending a message on the network did not lead to the best performance on our cluster of 8-way nodes. Instead, increasing the number of messages on the network to reduce the

depth of the reduction and broadcast as well as the number of direct children for each internal node proved to be a better strategy.

The reason for this may be found by studying three different factors that add to the cost of the global reduction and broadcast trees:

- The number of children directly below an internal node in the spanning tree.
- The depth of the spanning tree.
- Whether an arc between a child and a parent in the spanning tree is a message on the network or internally in the node.

We suspect that these factors are not increasing linearly, and that the cost of, for instance, adding another child to an internal node in the spanning tree depends on factors such as the order of the sequence of receive commands as well as contention on the network layer in the host computer.

8 Acknowledgements

Thanks to Ole Martin Bjørndalen for reading the paper and suggesting improvements.

References

[1] Mpi: A message-passing interface standard. *Message Passing Interface Forum*, Mar 1994.

[2] D.K. Panda. Issues in designing efficient and practical algorithms for collective communication in wormhole-routed systems. *Proc. ICPP Workshop Challenges for Parallel processing*, pages 8–15, 1995.

[3] John Markus Bjørndalen, Otto Anshus, Tore Larsen, and Brian Vinter. Paths - integrating the principles of method-combination and remote procedure calls for run-time configuration and tuning of high-performance distributed application. *Norsk Informatikk Konferanse*, pages 164–175, November 2001.

[4] Brian Vinter, Otto J. Anshus, Tore Larsen, and John Markus Bjørndalen. Extending the applicability of software dsm by adding user redefinable memory semantics. *Parallel Computing (ParCo) 2001, Naples, Italy*, September 2001.

[5] http://www.lam-mpi.org/.

[6] Jeffrey M. Squyres, Andrew Lumsdaine, William L. George, John G. Hagedorn, and Judith E. Devaney. The interoperable message passing interface (IMPI) extensions to LAM/MPI. In *Proceedings, MPIDC'2000*, March 2000.

[7] O. J. Anshus and Tore Larsen. Macroscope: The abstractions of a distributed operating system. *Norsk Informatikk Konferanse*, October 1992.

[8] Brian Vinter. *PastSet a Structured Distributed Shared Memory System*. PhD thesis, Tromsø University, 1999.

[9] N. Carriero and D. Gelernter. Linda in context. *Commun. ACM*, 32(4):444–458, April 1989.

[10] William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. A high-performance, portable implementation of the mpi message passing interface standard. *Parallel Computing, Volume 22, Issue 6*, September 1996.

[11] Matt Jacunski, P. Sadayappan, and D.K. Panda. All-to-all broadcast on switch-based clusters of workstations. *13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing*, 12 - 16 April 1999. San Juan, Puerto Rico.

[12] Massimo Bernaschi and Giorgia Richelli. Mpi collective communication operations on large shared memory systems. *Proceedings of the Ninth Euromicro Workshop on Parallel and Distributed Processing (EUROPDP.01)*, 2001.

[13] Thilo Kielmann, Rutger F. H. Hofman, Henri E. Bal, Aske Plaat, and Raoul A. F. Bhoedjang. Magpie: Mpi's collective communication operations for clustered wide area systems. *Proceedings of the seventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, 1999. Atlanta, Georgia, United States.

[14] Parry Husbands and James C. Hoe. Mpi-start: delivering network performance to numerical applications. *Proceedings of the 1998 ACM/IEEE conference on Supercomputing*, 1998. San Jose, CA.

[15] Steve Sistare, Rolf vandeVaart, and Eugene Loh. Optimization of mpi collectives on clusters of large-scale smp's. *Proceedings of the 1999 conference on Supercomputing*, 1999. Portland, Oregon, United States.

[16] Hong Tang and Tao Yang. Optimizing threaded mpi execution on smp clusters. *Proceedings of the 15th international conference on Supercomputing*, 2001. Sorrento, Italy.

[17] Sathish S. Vadhiyar, Graham E. Fagg, and Jack Dongarra. Automatically tuned collective communications. *Proceedings of the 2000 conference on Supercomputing*, 2000. Dallas, Texas, United States.

144

A.8 The Performance of Configurable Collective Communication for LAM-MPI in Clusters and Multi-Clusters

The Performance of Configurable Collective Communication for LAM-MPI in Clusters and Multi-Clusters

John Markus Bjørndalen[1], Otto J. Anshus[1]
Brian Vinter[2], Tore Larsen[1]

[1)] *Department of Computer Science, University of Tromsø*
[2)] *Department of Mathematics and Computer Science, University of Southern Denmark*

Abstract

Using a cluster of eight four-way computers, PastSet, an experimental tuple space based shared memory system, has been measured to be 1.83 times faster on global reduction than using the Allreduce operation of LAM-MPI. Our hypothesis is that this is due to PastSet using a better mapping of processes to computers resulting in less messages and more use of local processor cycles to compute partial sums. PastSet achieved this by utilizing the PATHS system for configuring multi-cluster applications. This paper reports on an experiment to verify the hypothesis by using PATHS on LAM-MPI to see if we can get better performance, and to identify the contributing factors.

By adding configurability to Allreduce by using PATHS, we achieved a performance gain of 1.52, 1.79, and 1.98 on respectively two, four and eight-way clusters. We conclude that the LAM-MPI algorithm for mapping processes to computers when using Allreduce was the reason for its poor performance relative to the implementation using PastSet and PATHS.

We then did a set of experiments to examine whether we could improve the performance of the Allreduce operation when using two clusters interconnected by a WAN link with 30-50ms roundtrip latency. The experiments showed that even a bad mapping of Allreduce which resulted in multiple messages being sent across the WAN did not add significant performance penalty to the Allreduce operation for packet sizes up to 4KB. We believe this is due to multiple small messages concurrently in transit on the WAN.

1 Introduction

For efficient support of synchronization and communication in parallel systems, these systems require fast collective communication support from the underlying communication subsystem as, for example, is defined by the Message Passing Interface (MPI) Standard [1]. Among the set of collective communication operations broadcast is fundamental and is used in several other operations such as barrier synchronization and reduction [12]. Thus, it is advantageous to reduce the latency of broadcast operations on these systems.

In our work with the PATHS[5] configuration and orchestration system, we have experimented with micro-benchmarks and applications to study the effects of configurable communication.

In one of the experiments[18], we used

the configuration mechanisms to reduce the execution times of collective communication operations in PastSet. To get a baseline, we compared our reduction operation with the equivalent operation in MPI (Allreduce).

By trying a few configurations, we found that we could improve our Tuple Space system to be 1.83 times faster than LAM-MPI (Local Area for Multicomputer MPI) [11][14]. Our hypothesis was that this advantage came from a better usage of resources in the cluster rather than a more efficient implementation.

If anything, LAM-MPI should be faster than PastSet since PastSet stores the results of each global sum computation in a tuple space inducing more overhead than simply computing and distributing the sum.

This paper reports on an experiment where we have added configurable communication to the Broadcast and Reduce operations in LAM-MPI (both of which are used by Allreduce) to validate or falsify our hypothesis. In [4] we report on complimentary experiments where we also varied the number of computers per experiment.

The paper is organized as follows: Section 2 summarizes the main features of the PastSet and PATHS system. Section 3 describes the Allreduce, Reduce and Broadcast operations in LAM-MPI. Section 4 describes the configuration mechanism that was added to LAM-MPI for the experiments reported on in this paper. Section 5 describes the experiments and results, section 7 presents related work, section 6 presents our multicluster experiments, and section 8 concludes the paper.

2 PATHS: Configurable Orchestration and Mapping

Our research platform is PastSet[2][17], a structured distributed shared memory system in the tradition of Linda[6]. PastSet is a set of Elements, where each Element is an ordered collection of tuples. All tuples in an Element follow the same template.

The PATHS[5] system is an extension of PastSet that allows for mappings of processes to hosts at load time, selection of physical communication paths to each element, and distribution of communications along the path. PATHS also implements the X-functions[18], which are PastSet operation modifiers.

A path specification for a single thread needing access to a given element is represented by a list of stages. Each stage is implemented using a wrapper object hiding the rest of the path after that stage. The stage specification includes parameters used for initialisation of the wrapper.

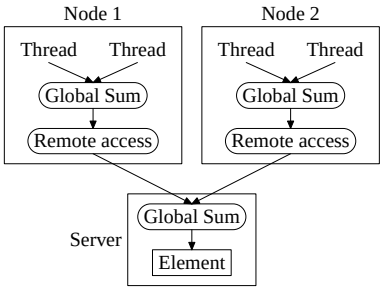


Figure 1. Four threads on two hosts accessing shared element on a separate server. Each host computes a partial sum that is forwarded to the global-sum wrapper on the server. The final result is stored in the element.

Paths can be shared whenever path descriptions match and point to the same element (see figure 1). This can be used to implement functionality such as, for instance, caches, reductions and broadcasts.

The collection of all paths in a system pointing to a given element forms a tree.

The leaf nodes in the tree are the application threads, while the root is the element.

Figure 1 shows a global reduction tree. By modifying the tree and the parameters to the wrappers in the tree, we can specify and experiment directly with factors such as which processes participate in a given partial sum, how many partial sum wrappers to use, where each sum wrapper is located, protocols and service requirements for remote operations and where the root element is located. Thus, we can control and experiment with tradeoffs between placement of computation, communication, and data location.

Applications tend to use multiple trees, either because the application uses multiple elements, or because each thread might use multiple paths to the same element.

To get access to an element, the application programmer can either choose to use lower-level functions to specify paths before handing it over to a path builder, or use a higher level function which retrieves a path specification to a named element and then builds the specified path. The application programmer then gets a reference to the topmost wrapper in the path.

The path specification can either be retrieved from a combined path specification and name server, or be created with a high-level language library loaded at application load-time[1]

Since the application program invokes all operations through its reference to the topmost wrapper, the application can be mapped to different cluster topologies simply by doing one of the following:

- Updating a map description used by the high-level library.
- Specifying a different high-level library that generates path-specifications. This library may be written by the user.
- Update the path mappings in the name server.

Profiling is provided via trace wrappers that log the start and completion time of operations that are invoked through it. Any number of trace wrappers can inserted anywhere in the path.

Specialized tools to examine the performance aspects of the application can later read trace data stored with the path specifications from a system. We are currently experimenting with different visualizations and analyses of this data to support optimization of a given application.

The combination of trace data, a specification of communication paths, and computations along the path has been useful in understanding performance aspects and tuning benchmarks and applications that we have run in cluster and multi-cluster environments.

3 LAM-MPI implementation of Allreduce

LAM-MPI is an open source implementation of MPI available from [11]. It was chosen over MPICH [7] for our work in [18] since it had lower latency with less variance than MPICH for the benchmarks we used in our clusters.

The MPI Allreduce operation combines values from all processes and distribute the result back to all processes. LAM-MPI implements Allreduce by first calling Reduce, collecting the result in the root process, then calling Broadcast, distributing the result from the root process. For all our experiments, the root process is the process with rank 0 (hereafter called process 0).

The Reduce and Broadcast algorithms use a linear scheme (every process communicates directly with process 0) up to and including 4 processes. From there on they use a scheme that organizes the processes into a logarithmic spanning tree.

[1] Currently, a Python module is loaded for this purpose.

The shape of this tree is fixed, and doesn't change to reflect the topology of the computing system or cluster. Figure 2 shows the reduction trees used in LAM-MPI for 32 processes in a cluster. We observe that broadcast and reduction trees are different.

By default, LAM-MPI evenly distributes processes onto nodes. When we combine this mapping for 32 processes with the reduction tree, we can see in Figure 3 that a lot of messages are sent across nodes in the system. The broadcast operation has a better mapping for this cluster though.

4 Adding configuration to LAM-MPI

To minimize the necessary changes to LAM-MPI for this experiment, we didn't add a full PATHS system at this point. Instead, a mechanism was added that allowed for scripting the way LAM-MPI communicates during the broadcast and reduce operations.

There were two main reasons for this. Firstly, our hypothesis was that PastSet with PATHS allowed us to map the communication and computation better to the resources and cluster topology. For global reduction and broadcast, LAM-MPI already computes partial sums at internal nodes in the trees. This means that experimenting with different reduction and broadcast trees should give us much of the effect that we observed with PATHS in [18] and [5].

Secondly, we wanted to limit the influence that our system would have on the performance aspects of LAM-MPI such that any observable changes in performance would come from modifying the reduce and broadcast trees.

Apart from this, the amount of code changed and added was minimal, which reduced the chances of introducing errors into the experiments.

When reading the LAM-MPI source code, we noticed that the reduce operation was, for any process in the reduction tree, essentially a sequence of N receives from the N children directly below it in the tree, and one send to the process above it. For broadcast, the reverse was true; one receive followed by N sends.

Using a different reduction or broadcast tree would then simply be a matter of examining, for each process, which processes are directly above and below it in the tree and construct a new sequence of send and receive commands.

To implement this, we added new reduce and broadcast functions which used the rank of the process and size of the system to look up the sequence of sends and receives to be executed (including which processes to send and receive from). This is implemented by retrieving and executing a script with send and receive commands.

As an example, when using a scripted reduce operation with a mapping identical to the original LAM-MPI reduction tree, the process with rank 12 (see figure 2) would look up and execute a script with the following commands:

- Receive (and combine result) from rank 13
- Receive (and combine result) from rank 14
- Send result to 8

The new scripted functions are used instead of the original logarithmic Reduce and Broadcast operations in LAM-MPI. No change was necessary to the Allreduce function since it is implemented using the Reduce and Broadcast operations.

The changes to the LAM-MPI code was thus limited to 3 code lines, replacing the calls to the logarithmic reduction and broadcast functions as well as adding and a call in MPI_Init to load the scripts.

Remapping the application to another cluster configuration, or simply trying new mappings for optimization purposes, now consists of specifying new communication trees and generating the scripts. A Python program generates these scripts as Lisp symbolic expressions.

5 Experiments

Figure 4 shows the code run in the experiments. The code measures the average execution time of 1000 Allreduce operations. The average of 5 runs is then plotted. To make sure that the correct sum is computed, the code also checks the result on each iteration.

For each experiment, the number of processes was varied from 1 to 32. LAM-MPI used the default placement of processes on nodes, which evenly spread the processes over the nodes.

The hardware platforms consists of three clusters, each with 32 processors:

- 2W: 16*2-Way Pentium III 450 MHz, 256MB RAM
- 4W: 8*4-Way Pentium Pro 166 MHz, 128MB RAM
- 8W: 4*8-Way Pentium Pro 200 MHz, 2GB RAM

All clusters used 100MBit Ethernet for intra-cluster communication.

5.1 4-way cluster

Figure 5 shows experiments run on the 4-way cluster. The following experiments were run:

Original LAM-MPI The experiment was run using LAM-MPI 6.5.6.

Modified LAM-MPI, original scheme The experiment was run using a modified LAM-MPI, but using the same broadcast and reduce trees for communication as the original version used.

This graph completely overlaps the graph from the original LAM-MPI code, showing us that the scripting functionality is able to replicate the performance aspects of the original LAM-MPI for this experiment.

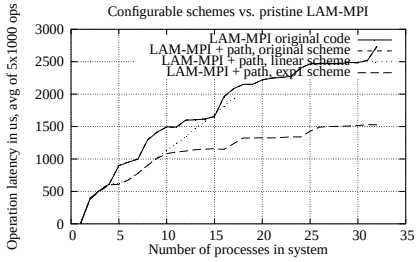


Figure 5. Allreduce, 4-way cluster

Modified LAM-MPI, linear scheme This experiment was run using a linear scheme where all processes report directly to the process with rank 0, and rank 0 sends the results directly back to each client.

This experiment crashed at 18 processes due to TCP connection failures in LAM-MPI. We haven't examined what caused this, but it is likely to be a limitation in the implementation.

Modified LAM-MPI, hierarchal scheme 1 Rank 0 is always placed on node 0, so all other processes on node 0 contribute directly to rank 0. On the other nodes, a local reduction is done within each node before the nodes partial sum is sent to process 0.

The modified LAM-MPI with the hierarchal scheme is 1.79 times faster than the original LAM-MPI. This is close to the factor of 1.83 reported in [18], which was based on measurements on the same cluster. It is possible that further experiments

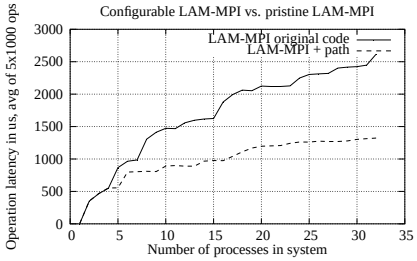


Figure 6. Allreduce, 8-way cluster

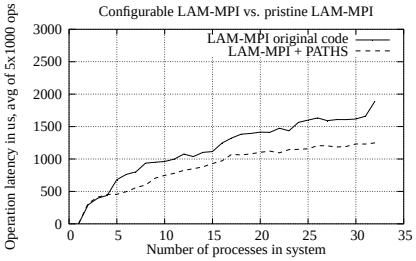


Figure 7. Allreduce, 2-way cluster.

with configurations would have brought us closer to the original speed improvement.

5.2 8-way cluster

For the 8-way cluster, we expected the best configuration to involve using an internal root process on each node and limit the inter-node communications to one send (of the local sum) and one receive (of the global result) resembling the best configuration for the 4-way cluster. This turned out not to be true.

Due to the 8 processes on each node, the local root process on each node would get 7 children in the sum tree (with the exception of the root processes in the root node). The extra latency added per child of the local root processes was enough to increase the total latency for 32 processes to 1444 microseconds.

Splitting the processes to allow further subgrouping within each node introduced another layer in the sum tree. This extra layer brought the total latency for 32 processes to 1534 microseconds.

The fastest solution found so far involves partitioning the processes on each node into two groups of 4 processes. Each group computes a partial sum for that group, and forwards it to directly to the root process. This doubled the communication over the network, but the total execution time was shorter (1322 microseconds).

The fastest configuration is 1.98 times faster than the original LAM-MPI Allreduce operation. In figure 6, the fastest configuration and the original LAM-MPI Allreduce are plotted for 1-32 processes.

5.3 2-way cluster

On the 2-way cluster, after trying a few configurations, we ended up with a configuration that was 1.52 times faster than the original LAM-MPI code. This was not as good as for the other clusters. We expected this since a reduction in this cluster would need more external messages.

Based on the experiments, we observed three main factors that influenced the scaling of the application:

- The number of children for a process in the operation tree. More children adds communication load for the node hosting this process.
- The depth of a branch in the operation tree. That is, the number of nodes a value has to be added through before reaching the root node.
- Whether a message was sent over the network or internally on a node.

The depth factor seemed to introduce a higher latency than adding a few more children to a process. On the other hand, adding another layer in the tree would also allow us to use partial sums. We have

not arrived at any model which allows us predict which tree organization would in practice lead to the shortest execution time of the Allreduce operation.

One difference between the PastSet/PATHS implementation and LAM-MPI is that PastSet/PATHS process incoming partial sums by order of arrival, while LAM-MPI process them in the order they appear in the sequence specified either by the original LAM-MPI code, or the script. This might influence the overhead when receiving messages from a larger number of children. We are investigating this further.

6 Multi-cluster experiments

In [5] we also ran multi-cluster global reductions using PastSet and PATHS, and [10] shows an approach where they reduce the number of messages over WAN connections to reduce the latency of collective operations over wide-area links.

To study the effect of choosing various configurations on multi-clusters, we added an experiment where we installed an IP tunnel between the 4W cluster in Tromsø and the 2W cluster in Denmark and ran the Allreduce benchmark using those clusters as one larger cluster.

Unfortunately, during the experiments two things happened: two of the nodes in Denmark went down, and the 2W cluster was allocated to other purposes after we finished the measurements with the unmodified LAM-MPI Allreduce, so we were unable to run the benchmark using configurable Allreduce on the multi-cluster system.

However, the performance measurements using the unmodified Allreduce did produce interesting results. An experiment with 32 processes both in Denmark and Tromsø documented that the reduce phase of the Allreduce operation had a very good mapping for minimizing messages over the WAN link: only one message was sent from the Tromsø cluster to the root process in the Denmark cluster. However, in the broadcast phase each process in Tromsø was sent a message from a process in Denmark.

We expected this configuration of Allreduce to perform gradually worse when we scaled up the number of processes in Tromsø from 1 to 32. This did not turn out to be the case. With 32 processes in Denmark, the difference between the Allreduce latency of using 1 and 32 processes in Tromsø was small enough to be masked by the noise in latency.

We suspected that the reason for this was that the numerous routers and links down to Denmark allowed multiple small messages to propagate towards Tromsø concurrently.

This suggests that for the cluster sizes that we have available, a configurable mechanism that reduces the number of *small* messages on a long WAN link does not significantly reduce the latency. However, for larger packet sizes, the bandwidth of the link is important, and a reduction in the number of messages should matter more.

To study this, the benchmark was extended to include not only scaling from 1 to 64 processes, but also to calling Allreduce with value vectors from 4 byte (1 integer) to 4MB.

Since two computers in Denmark had gone down, we ran the benchmark with 28 processors (14 nodes) in Denmark, and 32 processors in Tromsø reducing the total cluster size to 60 processors. This produced a similar mapping to the original: when running with 60 processes, all Tromsø nodes would receive a message directly from Denmark during broadcast, and only two messages would be sent down to Denmark on reduce.

We also ran the benchmark with 61 processes. This resulted in a configuration where the process with rank 60 (the last process added when scaling the system to

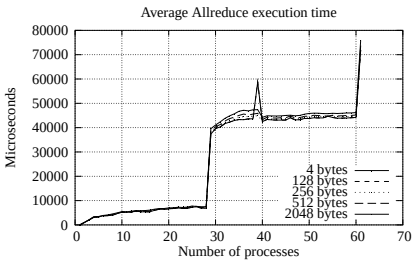


Figure 8. Allreduce performance using two clusters.

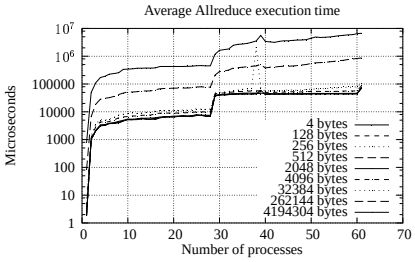


Figure 9. Allreduce performance using two clusters. Note logarithmic scale of y-axis.

61 processes) is located in Denmark but only communicates directly with nodes in Tromsø for Reduce and Broadcast. This forces the nodes in Tromsø to wait for process 60 before one of the two messages can be sent down to Denmark again, and the corresponding broadcast message for 60 has to propagate through Tromsø before reaching it. Figure 8 shows that this configuration nearly doubles the latency of Allreduce compared to running Allreduce with 60 processes. Clearly, configurations can be encountered where it pays off to control who communicates with who even for small messages.

Figure 8 also shows us that sending many messages (there is one message per process in the figure) for vectors up to 2KB does not add much performance

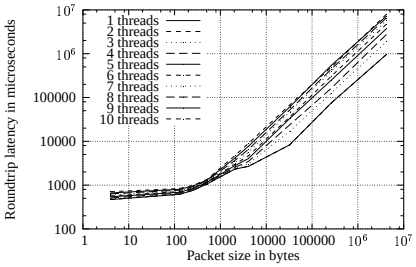


Figure 10. Roundtrip TCP/IP latency for one of N threads. 4W node to 8W node (LAN)

penalty to Allreduce compared to sending only a few messages. The figure shows a spike around 38 processes, which comes from a temporary network routing problem somewhere between Denmark and Tromsø when the experiment was performed.

Figure 9 shows that the performance of Allreduce rapidly deteriorates when the size and number of packets increase. The graph uses a logarithmic scale for the y-axis.

To examine this effect further, we devised another experiment. We had one thread measuring the latency of roundtrip messages over TCP/IP. To simulate the bad mapping of the Broadcast operation, a number of threads were added sending roundtrip messages concurrently with the first thread.

The hypotheses were that adding the extra threads would impact the first thread the least using the WAN to Denmark, and that the background threads would influence the first thread more for small messages when using the LAN.

Figure 10 shows that communicating between two cluster nodes, using only a single router between them, the background communication would influence the roundtrip latency of the benchmark thread even at the smallest packet sizes. As we add background communication,

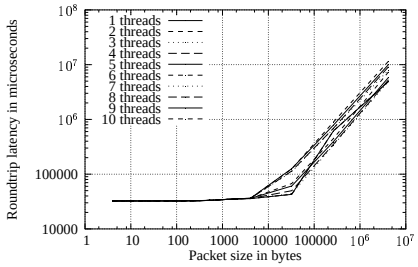


Figure 11. Roundtrip TCP/IP latency for one of N threads. Gateway node in Denmark to 4W node (WAN)

the latency increases for all packet sizes.

The cluster in Denmark was busy at the time of the experiment, so we had to use the clusters gateway node when running the WAN experiment. The gateway was where we originally installed the IP tunnel for the LAM-MPI Allreduce benchmark, so we still used the tunnel for the experiment.

Figure 11 shows that for messages up to 4KB, we can hardly see any difference in the performance of Allreduce as we add background communication. Only with larger packet sizes, where we assume that the bandwidth plays a larger role, is the performance influenced by the background communication.

7 Related work

Jacunski et al.[9] shows that selection of the best performing algorithm for all-to-all broadcast on clusters of workstations based on commodity switch-based networks is a function of both network characteristics as well as the message length and number of participating nodes in the all-to-all broadcast operation. In contrast our work used clusters with multiprocessors, we did performance measurements on the reduce operation as well as on the broadcast, and we documented the effect of the actual system at run time including the workload, communication performance of the hosts.

Bernashci et al.[3] study the performance of the MPI broadcast operation on large shared memory systems using atrees.

Kielmann et al.[10] show how the performance for collective operations like broadcast depend upon cluster topology, latency, and bandwidth. They develop a library of collective communication operations optimized for wide area systems where a majority of the speedup comes from limiting the number messages passing over wide-area links.

Husbands et al.[8] optimizes the performance of MPI_Bcast in clusters of SMP nodes by using a two-phase scheme; first messages are sent to each SMP node, then messages are distributed within the SMP nodes. Sistare et al.[13] uses a similar scheme, but focus on improving the performance of collective operations on large-scale SMP's.

Tang et al.[15] also use a two-phase scheme for a multithreaded implementation for MPI. They separate the implementation of the point-to-point and collective operations to allow for optimizations of the collective operations, which would otherwise be difficult.

In contrast to the above works, this paper is not focused on a particular optimization of the spanning trees. Instead, we focus on making the shape of the spanning trees configurable to allow easy experimentation on various cluster topologies and applications.

Vadhiyar et al.[16] shows an automatic approach to selecting buffer sizes and algorithms to optimize the collective operation trees by conducting a series of experiments.

8 Conclusions

We have observed that the broadcast and reduction trees in LAM-MPI are dif-

ferent, and do not necessarily take into account the actual topology of the cluster.

By introducing configurable broadcast and reduction trees, we have shown a simple way of mapping the reduction and broadcast trees to the actual clusters in use. This gave us a performance improvement up to a factor of 1.98.

For the cluster where we observed the performance difference of a factor 1.83 between PastSet/PATHS and LAM-MPI, we arrived at a reduction and broadcast tree that gave us an improvement of 1.79 for Allreduce over the original LAM-MPI implementation. This supports our hypothesis that the majority of the performance-difference between LAM-MPI and PastSet/PATHS was a better mapping to the resources and topology in the cluster.

We have also observed that the assumption that doing a reduction internally in each node before sending a message on the network did not lead to the best performance on our cluster of 8-way nodes. Instead, increasing the number of messages on the network to reduce the depth of the reduction and broadcast as well as the number of direct children for each internal node proved to be a better strategy.

The reason for this may be found by studying three different factors that add to the cost of the global reduction and broadcast trees:

- The number of children directly below an internal node in the spanning tree.
- The depth of the spanning tree.
- Whether an arc between a child and a parent in the spanning tree is a message on the network or internally in the node.

We suspect that these factors are not increasing linearly, and that the cost of, for instance, adding another child to an internal node in the spanning tree depends on factors such as the order of the sequence of receive commands as well as contention on the network layer in the host computer.

For the multi-cluster experiments, we observed that configurations which sent more messages than necessary over the WAN link did not perform as bad as we had expected. For message sizes up to 4KB, the extra messages did not add a noticeable operation time to the Allreduce operation. We believe that multiple messages concurrently in transit through the routers along the WAN link masks some of the overhead of the extra messages send by LAM-MPI.

For larger messages, we see that the bandwidth of the WAN link is starting to penalize the bad configurations.

9 Acknowledgements

Thanks to Ole Martin Bjørndalen for reading the paper and suggesting improvements.

References

[1] Mpi: A message-passing interface standard. *Message Passing Interface Forum* (Mar 1994).

[2] ANSHUS, O. J., AND LARSEN, T. Macroscope: The abstractions of a distributed operating system. *Norsk Informatikk Konferanse* (October 1992).

[3] BERNASCHI, M., AND RICHELLI, G. Mpi collective communication operations on large shared memory systems. *Proceedings of the Ninth Euromicro Workshop on Parallel and Distributed Processing (EUROPDP.01)* (2001).

[4] BJØRNDALEN, J. M., ANSHUS, O., VINTER, B., AND LARSEN, T. Configurable collective communication

in lam-mpi. *Proceedings of Communicating Process Architectures 2002, Reading, UK* (September 2002).

[5] BJØRNDALEN, J. M., ANSHUS, O., LARSEN, T., AND VINTER, B. Paths - integrating the principles of method-combination and remote procedure calls for runtime configuration and tuning of high-performance distributed application. *Norsk Informatikk Konferanse* (November 2001), 164–175.

[6] CARRIERO, N., AND GELERNTER, D. Linda in context. *Commun. ACM 32*, 4 (April 1989), 444–458.

[7] GROPP, W., LUSK, E., DOSS, N., AND SKJELLUM, A. A high-performance, portable implementation of the mpi message passing interface standard. *Parallel Computing, Volume 22, Issue 6* (September 1996).

[8] HUSBANDS, P., AND HOE, J. C. Mpi-start: delivering network performance to numerical applications. *Proceedings of the 1998 ACM/IEEE conference on Supercomputing* (1998). San Jose, CA.

[9] JACUNSKI, M., SADAYAPPAN, P., AND PANDA, D. All-to-all broadcast on switch-based clusters of workstations. *13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing* (12 - 16 April 1999). San Juan, Puerto Rico.

[10] 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. *Proceedings of the seventh ACM SIGPLAN symposium on Principles and practice of parallel programming* (1999). Atlanta, Georgia, United States.

[11] http://www.lam-mpi.org/.

[12] PANDA, D. Issues in designing efficient and practical algorithms for collective communication in wormhole-routed systems. *Proc. ICPP Workshop Challenges for Parallel processing* (1995), 8–15.

[13] SISTARE, S., VANDEVAART, R., AND LOH, E. Optimization of mpi collectives on clusters of large-scale smp's. *Proceedings of the 1999 conference on Supercomputing* (1999). Portland, Oregon, United States.

[14] SQUYRES, J. M., LUMSDAINE, A., GEORGE, W. L., HAGEDORN, J. G., AND DEVANEY, J. E. The interoperable message passing interface (IMPI) extensions to LAM/MPI. In *Proceedings, MPIDC'2000* (March 2000).

[15] TANG, H., AND YANG, T. Optimizing threaded mpi execution on smp clusters. *Proceedings of the 15th international conference on Supercomputing* (2001). Sorrento, Italy.

[16] VADHIYAR, S. S., FAGG, G. E., AND DONGARRA, J. Automatically tuned collective communications. *Proceedings of the 2000 conference on Supercomputing* (2000). Dallas, Texas, United States.

[17] VINTER, B. *PastSet a Structured Distributed Shared Memory System*. PhD thesis, Tromsø University, 1999.

[18] VINTER, B., ANSHUS, O. J., LARSEN, T., AND BJØRNDALEN, J. M. Extending the applicability of software dsm by adding user redefinable memory semantics. *Parallel Computing (ParCo) 2001, Naples, Italy* (September 2001).

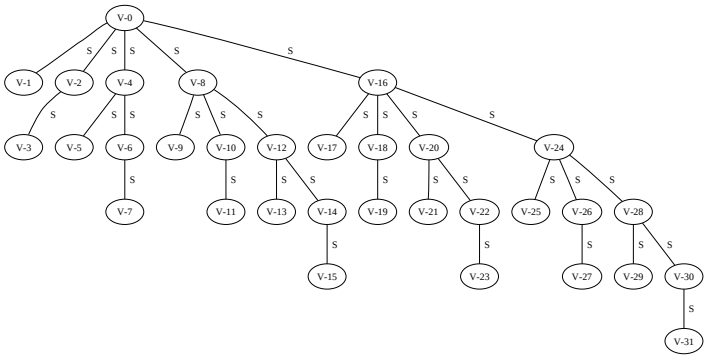


Figure 2. Log-reduce tree for 32 processes. The arcs represent communication between two nodes. Partial sums are computed at a node in the tree before passing the result further up in the tree.

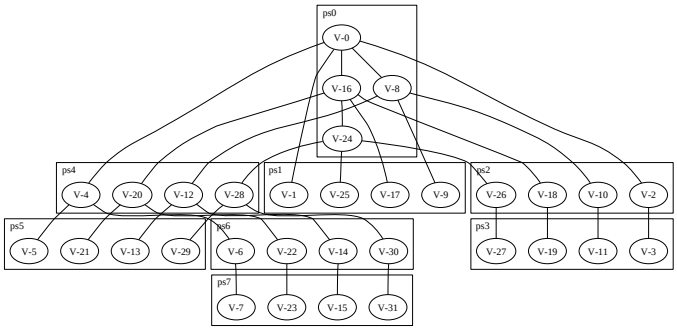


Figure 3. Log-reduce tree for 32 processes mapped onto 8 nodes.

```
t1 = get_usecs();
MPI_Allreduce(&hit, &ghit, 1, MPI_INT, MPI_SUM, MPI_COMM_WORLD);
t1 = get_usecs();
for (i = 0; i < ITERS; i++) {
   MPI_Allreduce(&i, &ghit, 1, MPI_INT, MPI_SUM, MPI_COMM_WORLD);
   if (ghit != (i * size))
      printf("oops at %d. %d != %d\n", i, ghit, (i * size));
}
t2 = get_usecs();
MPI_Allreduce(&hit, &ghit, 1, MPI_INT, MPI_SUM, MPI_COMM_WORLD);
```

Figure 4. Global reduction benchmark

A.9 The latency of user-to-user, kernel-to-kernel and interrupt-to-interrupt level communication

The latency of user-to-user, kernel-to-kernel and interrupt-to-interrupt level communication

John Markus Bjørndalen[1], Otto J. Anshus[1], Brian Vinter[2], Tore Larsen[1]

[1)] Department of Computer Science
University of Tromsø

[2)] Department of Mathematics and Computer Science
University of Southern Denmark

Abstract

Communication performance depends on the bit rate of the networks physical channel, the type and number of routers traversed, and on the computers ability to process the incoming bits. The latter depends on factors such as the protocol stack, bus architecture and the operating system.

To see how these factors impact the roundtrip latency, we experiment with four different protocols, an optimization of one of the protocols, and with moving the communicating endpoints from user-level processes into the kernel, and then further on into the interrupt handler of two communicating computers.

We then add a processor-bound workload to the receiving computer to simulate a server which also do computations in addition to network I/O. We report how this workload affects the roundtrip latency of the different protocols and endpoints, and also how the different benchmarks influence the execution time of the workload.

The largest reduction in latency can be achieved by using less complex protocols specialized for local networks. A less complex protocol also has less variation in the roundtrip latency when the workload is running.

For small messages, interrupt-level communication and the TCP/IP based communication both increase the workload running time the least. However, the interrupt-level communication will transmit 5.3 times more messages.

1 Introduction

The importance of distributed computing is increasing as clusters of workstations or PCs becomes more and more important. Intranets are becoming important mechanisms to manage information flow in organizations. We are seeing an increasing number of services that need to perform significant computations, and the importance and role of application and compute facilities are becoming larger.

Processor, memory and bus performance, and more and more communication performance are important factors in determining the time an application needs to complete.

The communication performance de-

pends on the bit rate of the networks physical channel and the type and number of routers traversed, and it depends on the hosts ability to process the incoming bits. The latter depends on factors like the protocol stack, the bus architecture, and the operating system. When a message arrives or departs, many activities take place including data copying, context switching, buffer management, and interrupt handling.

In particular, low latency and low perturbation of the workload is important for scientific computations, which typically combine high computational load with a need for tight, low-latency communication.

To learn more about where the overheads are located and how large they are, we have done performance measurements to find out how much the latency of sending a message between two hosts using blocking message passing improves when increasingly less complex communication protocols are used from the user level. To establish a lower bound on the achievable latency, we also measured the latency when sending a message between two operating system kernels without going to user level. We did this in two steps, first we measured the latency when running the message passing threads in the kernel, and then by running a roundtrip benchmark directly from the interrupt handler for the network card in use. The ratios between the user level protocols and the kernel and interrupt level communication latencies will provide us with data to evaluate if there are benefits from moving, say, a distributed shared memory server closer to the network card by locating it at kernel or even interrupt level. This is work in progress.

There is overhead in allocating, copying and queueing buffers for later processing by a layer further up. If the requested service (say, send/receive operation, or read/write to distributed shared memory) takes less time than the overhead it could be better to execute the operation directly from the interrupt handler. In an earlier paper[2], we compared the performance of the read and write operations of the PastSet distributed shared memory [1][6] when using TCP/IP and two implementations of VIA (Virtual Interface Architecture). One of the VIA implementations had direct hardware support from the Gigabit network card, and the other was a software implementation, M-VIA [5], using a 100Mbit network. The results showed that both VIA implementations were significantly faster than TCP/IP. This was as expected. However, M-VIA using a 100Mbit network for small messages had comparable performance to the hardware supported VIA using a Gigabit network when blocking message passing was used between two user level processes.

We believe that the reason for this is that the majority of the overhead in both the hardware supported VIA implementation and M-VIA comes from interrupt handling and scheduling the user-level threads.

Furthermore, we observed that some of the factors contributing to the low latency of the M-VIA implementation were:

- Modified versions of the network device drivers were used which provided hooks for the M-VIA subsystem
- A localnet/Ethernet-optimized protocol was used
- Checksums were computed directly on the network interface card
- A low-overhead API with statically allocated buffers in user space was used
- Fast kernel traps were used instead of using the ordinary system call or device call mechanisms

One of the more important factors was that the implementation made use of a low-overhead Ethernet-optimized protocol. As such, it avoided a lot of the processing that internet-protocols such as TCP/IP do. However, M-VIA was also able to queue packets directly to the network interface card, and provide hooks which allows the interrupt handler to dispatch incoming VIA packets to the M-VIA system instead of sending them to the ordinary Ethernet layer in the kernel.

2 The Low-Latency Protocol

M-VIA uses a special kernel trap mechanism to speed up kernel access. A simpler (but slightly less efficient) way to provide a quick access to a device driver is to use the Linux /proc filesystem, which is a mechanism for exporting interfaces to device drivers (or any other Linux kernel module).

The /proc filesystem is typically used to export interface files which can be used for tasks such as inspection of devices (by reading files) and setting of options (by writing to specific files). To provide such interfaces, a device driver exports a file in a subdirectory of /proc and binds functions of its own choice to the read and write operations of those files.

To implement the Low Latency Protocol (LLP), we use the /proc interface to provide four different interfaces (described below) for sending and receiving Ethernet frames with a protocol ID which separate them from other protocols transmitted on the cable. The interfaces were exported by an extension to the device driver for the 100Mbit Ethernet cards we used (TrendNet NICs with DEC Tulip based chipsets).

The LLP protocol uses raw Ethernet frames with only a few addressing headers added.

Two of the interfaces were used to control the kernel-based and interrupt handler based benchmarks.

The four interfaces are:

basic This interface allows a user level process to read and write raw Ethernet frames directly to and from the network card using only a simple packet processing layer in the driver.

When a user writes a packet to the interface file, an sk_buf (a network buffer) is allocated, the packet is copied into the buffer and the buffer is queued directly with the device driver.

We used a unique ethernet protocol ID, which allowed us to recognize our own protocol directly in the device drivers interrupt handler. This was used to dispatch incoming packets to the LLP subsystem in the interrupt handler instead of going through the Ethernet packet processing in the Linux kernel.

Incoming packets are written to a buffer in the LLP subsystem. If a user-level process is blocked waiting for a packet, the process is unblocked.

basic static-skb Allocating and freeing network buffers introduces overhead in the protocol. Instead of dynamically allocating buffers for each packet, we statically allocate an sk_buf when the interface file is opened.

This buffer is reused every time the client wants to send a packet.

The interface is identical to the one above, and uses the same read routine.

kernel-based roundtrip To measure the roundtrip latency between two kernel level processes, we provide an interface file where the write operation starts a benchmark loop in the kernel.

An iteration count is provided when writing to the file, and timing data is

written back to the client by modifying the write buffer.

The remote end of the benchmark starts by invoking a read operation on the corresponding interface file on that host.

The benchmark loop is run by the Linux user level processes which invoked the write and read operations. Thus, the benchmark is not run by spawning Linux kernel threads.

interupt-handler roundtrip When a packet arrives from the network for the kernel level server, the interrupt handler in the device driver still needs to queue up (or register) the packet and wake up the server thread in the benchmark.

This involves some synchronization between the interrupt handler and kernel process, as well as scheduling and running the server thread.

To avoid this, we allowed the interrupt handler to process the packet directly and queue up a reply packet for transmission with the NIC. This allows us to keep the bechmark entirely in the interrupt handlers.

To start the interrupt handler benchmark, a write operation is invoked on the corresponding interface file. An iteration count and a timestamp is stored in the packet, and the packet is immediately queued with the device driver.

The iteration count is decreased every time the packet passes trough an interrupt handler. A second timestamp is stored in the packet when the iteration count reaches 0, and the packet is returned up to the thread which invoked the write operation.

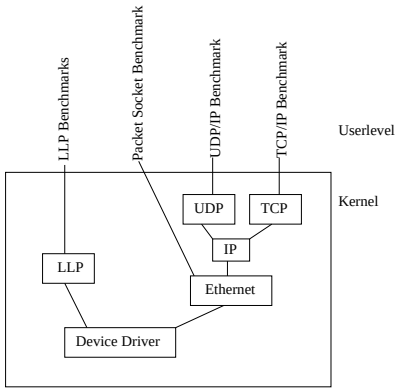


Figure 1. Protocols used in benchmarks, and location of protocol implementations.

3 Experiments

We have measured the round-trip latency of communication between two user-level processes, each on a different computer, using four different protocols. For the LLP protocol, we also measured the round-trip latency between two threads located in the operating system kernel, each on a different computer. Finally, we measured the round-trip latency of communication between two interrupt handler device drivers, each on a different computer.

The experiments measure the time it takes sending a message from one computer to another, sending a reply back, and receiving it on the original computer. We call this the round-trip latency.

For each experiment, we measure the time it took to send and receive 1500 round-trip messages, and compute the average round-trip latency. We then repeat each experiment five times, and compute the minimum, maximum, and average of the averages.

The following experiments were run:

- TCP/IP protocol, user-level to user-

level

- UDP/IP protocol, user-level to user-level
- Packet socket, transmitting raw Ethernet frames, user-level to user-level
- LLP, using the /proc filesystem, user-level to user-level
- LLP, using the /proc filesystem, statically allocated network buffers, userspace to userspace
- LLP, kernel-level to kernel-level
- LLP, interrupt handler to interrupt handler

The whole set of experiments were repeated with a work load on the receiving computer. This scenario emulates a typical client/server model where a client requests a service from a server with has other work loads than servicing remote requests. We measured both the round-trip latencies and the running time of the work load in each case. As a baseline we measured the running time of the work load without any communication.

As the work load we used a tiled matrix multiplication with good cache utilization.

The experiments were run on two HP Netserver LX Pros (4-Way 166 MHz Pentium Pros with 128 MB RAM), with Trendnet TE100-PCIA 100MBit network cards (DEC Tulip chipsets) connected to a hub. The computers were run with uniprocessor Linux kernels to simplify the experiments.

All of the protocols in the experiments are implemented at kernel level. Figure 1 shows the protocols and some of the subsystems used in the various experiments. The TCP/IP and UDP/IP benchmarks use blocking send and receive on standard sockets. The packet socket benchmark uses the PF_PACKET socket type to send and receive raw Ethernet packets on the network. Even if this saves us from processing the packet through the IP and UDP/TCP layers, the Ethernet frames are still processed by a packet processing layer in the kernel. This introduces extra processing overhead which, for instance, the M-VIA[5] implementation of the VIA communication API avoids.

To avoid perturbations with IP (and other protocols), we used Ethernet protocol ID 0x6008, which is, as far as we know, not allocated. This means that we will only receive traffic meant for this benchmark on the sockets.

4 Results

4.1 Roundtrip latency

Benchmark	min	max	avg
TCP/IP, userlevel	187	189	188
UDP/IP, userlevel	141	142	141
Packet socket, userlevel	114	115	115
LLP userlevel	87	88	88
LLP userlevel static buf	82	83	82
LLP kernel level	67	67	67
LLP interrupt handler	54	54	54

Table 1. Roundtrip latency in microseconds, 32 byte message, without workload

In table 1, we show the roundtrip latency for each benchmark with a message size of 32 bytes. The computers have no workload apart from the benchmarks. We have added a divider in the table to make it easier to see where we introduce kernel level benchmarks.

Most of the latency reduction (106 microseconds) comes from choosing a simpler protocol. Moving the benchmark into the interrupt handlers only reduce the latency by another 28 microseconds compared to the best user-level protocol.

Benchmark	min	max	avg
TCP/IP, userlevel	231	441	286
UDP/IP, userlevel	145	291	202
Packet socket, userlevel	118	178	140
LLP userlevel	90	130	106
LLP userlevel static buf	84	131	102
LLP kernel level	68	69	68
LLP interrupt handler	54	54	54

Table 2. Roundtrip latency in microseconds, 32 byte message, with workload

Table 2 shows the roundtrip latency for the benchmarks when we add a matrix multiplication workload to the receiving computer.

The less complex user-level protocols are less influenced by the workload than the more complex protocols. The kernel-level benchmark is hardly influenced by the workload, while the interrupt-level benchmark is not influenced by the workload at all.

This is interesting, since the kernel-level benchmark is implemented with user-level threads that enter the kernel and run the benchmark code. We assume that the kernel-level benchmark is scheduled the same way as any other userlevel process.

Benchmark	min	max	avg
TCP/IP, userlevel	384	385	384
UDP/IP, userlevel	339	340	339
Packet socket, userlevel	311	312	311
LLP userlevel	283	286	284
LLP userlevel static buf	278	278	278
LLP kernel level	251	253	252
LLP interrupt handler	215	215	215

Table 3. Roundtrip latency in microseconds, 1024 byte message, without workload

Table 3 shows the benchmarks with no workload on any of the hosts, and with the message size increased to 1KB.

Compared to the latencies in table 1, the increase in average latency is 196 microseconds for all user-level protocols, except from the UDP benchmark which has an increase of 198 microseconds.

The amount of processing per packet for any one of the protocols should be the same both for 32-byte and a 1024-byte message since both packet sizes fit within an Ethernet frame. Thus, the extra overhead of sending a 1024-byte message should depend on the extra time spent in the network cable, in the network adapter, time spent being copied between the network adapter to the host memory as well as time spent copying the packet within the host memory.

Observing that the added overhead for a 1KB packet compared to a 32 byte packet is nearly constant for all user-level protocols suggest that the attention paid in the TCP/IP and UDP/IP stacks to avoid extra copying of data packets has paid off and brought the number of copies down the same number as the other userlevel protocols.

For the kernel-level benchmark, the difference between the latency for 32-byte and 1024-byte payloads is further reduced to 185 microseconds.

A roundtrip message between two processes at user-level is copied 4 times between user-level and kernel-level per roundtrip. We have measured that copying a 1KB buffer takes about 2.6 microseconds on the computers we used. This explains most of the extra overhead for the user-level protocols compared to the kernel-level protocol.

For the interrupt-handler roundtrip messages, the difference between 32-byte and 1KB packets is even less at 161 microseconds. Apart from the copying done by the Ethernet card over the PCI bus to and from memory, the interrupt handler benchmark does not copy the contents of the message. Instead, it modifies the headers of the incoming message and inserts the outgoing

message directly in the output queue for the network card.

Benchmark	min	max	avg
TCP/IP, userlevel	387	467	427
UDP/IP, userlevel	423	921	598
Packet socket, userlevel	316	396	346
LLP userlevel	287	288	287
LLP userlevel static buf	281	287	282
LLP kernel level	254	255	255
LLP interrupt handler	215	215	215

Table 4. Roundtrip latency in microseconds, 1024 byte payload, with workload

Table 4 shows the roundtrip latency of 1KB messages when the receiving computer runs the the matrix multiplication workload.

As in table 2, we see that the less complex protocols are less influenced by the workload than the more complex protocols.

4.2 Implications of the roundtrip benchmarks on the workload

When no benchmarks are run, the average execution time for multiplying two 512 by 512 matrices is 13.4 seconds. In tables 5 and 6, we show the impact on the execution time of the matrix multiplication when running the benchmarks. We also show the average number of roundtrip messages per second while the benchmarks execute.

Table 5 shows that the benchmarks increase the execution time of the matrix multiplication by 1.6 to 2.3 times.

The benchmark which influences the matrix multiplication the most is the packet socket benchmark. We currently do not have an explanation why this benchmark performs as bad as it does, but observe that the benchmark sends more than twice as many roundtrip messages than the TCP/IP benchmark. A likely place to look for an explanation is the amount of code executed by the protocol combined with the time that the matrix multiplication is allowed to compute before being interrupted by another packet from the client.

The two benchmarks which influence the matrix multiplication the least are the TCP/IP benchmark and the interrupt handler benchmark.

We believe that one of the reasons the workload is not influenced more by the TCP/IP benchmark is that the higher overhead in the TCP/IP protocols means that it takes longer for the client host to send the next message to the host with the workload. The matrix multiplication application is thus allowed to compute more before being interrupted by another request from the client end.

Another observation is that there is a factor 5.3 difference in the number of roundtrip messages between the two best benchmarks in this experiment, implying that a server in the kernel could support a much higher network load without disturbing the workload on a host more than a TCP/IP server which serves a smaller load.

Table 6 shows us a similar pattern to table 5. The packet socket benchmark still comes out worst when comparing the influence on the workload execution time, while the least disturbing benchmarks are the TCP/IP and interrupt handler benchmarks.

5 Related Works

In [4] it was documented that CPU performance is not the only factor affecting network throughput. Even though we have not used different computers, we have also shown that the network latency is dependent upon several other factors than CPU performance, and we have detailed some of these.

In [3] it is shown how the cost of user-level communication can be reduced by reducing the cost of servicing interrupts,

Benchmark	min	max	avg	messages/s
TCP/IP, userlevel	19	23	21	3494
UDP/IP, userlevel	23	26	24	4950
Packet socket, userlevel	25	37	30	7120
LLP userlevel	23	27	26	9406
LLP userlevel static buf	22	27	24	9801
LLP kernel level	23	24	24	14621
LLP interrupt handler	21	21	21	18501

Table 5. Impact of the benchmarks on the work load execution time, 32 bytes message

Benchmark	min	max	avg	messages/s
TCP/IP, userlevel	16	16	16	2341
UDP/IP, userlevel	16	16	16	1671
Packet socket, userlevel	20	20	20	2893
LLP userlevel	17	18	18	3482
LLP userlevel static buf	18	18	18	3540
LLP kernel level	16	16	16	3923
LLP interrupt handler	15	15	15	4642

Table 6. Impact of roundtrip benchmarks on matrix multiplication execution time, 1024 bytes message

and by controlling when the system uses interrupts and when it uses polling. They showed that blocking communication was about an order of magnitude more expensive than spinning between two computers in the Shrimp multicomputer.

Active messages [7] invoke a receiver-side handler whenever a message arrives. Control information at the head of each message species the address of a user-level routine that is responsible for extracting the message from the network. This approach often requires servicing an interrupt for each message received.

6 Conclusion

We have observed that the largest impact on roundtrip latency is the complexity and implementation of the protocol used, not whether the endpoints of the communication are in userspace or in kernel space. Reducing the overhead by choosing simpler protocols also resulted in less variation in the roundtrip latency.

This suggests that a significant reduction in communication latency can be achieved without modifications to the operating system kernel. However, to achieve the best possible latency, a combination of interrupt level handling of the communication with an efficient activation of user-level processes are necessary. The interrupt level latency effectively gives a practical lower bound on how low the latency can be.

Moving data between kernel and user-level by copying also introduces extra overhead which is visible in the roundtrip latency. This suggests that we can benefit from moving communication handling to kernel space if the data can be stored in the kernel[1]. This can be of use for distributed

[1]The extra copying could also be avoided if we used shared buffers between kernel and userspace, as VIA does

shared memory systems using servers on each computer.

The interrupt-level benchmark which had the lowest latency, was least influenced by the workload (not visible in the tables). This benchmark also disturbed the workload the least. Again, this suggests that there may be performance benefits from integrating, say, a distributed shared memory server with the interrupt handler.

7 Acknowledgements

We thank Lars Ailo Bongo for reading the paper and suggesting improvements.

References

[1] ANSHUS, O. J., AND LARSEN, T. Macroscope: The abstractions of a distributed operating system. *Norsk Informatikk Konferanse* (October 1992).

[2] BJØRNDALEN, J. M., ANSHUS, O., VINTER, B., AND LARSEN, T. Comparing the performance of the pastset distributed shared memory system using tcp/ip and m-via. In *Proceedings of WSDSM'00, Santa Fe, New Mexico* (May 2000).

[3] DAMIANAKIS, S. N., CHEN, Y., AND FELTEN, E. Reducing waiting costs in user-level communication. In *11th International Parallel Processing Symposium (IPPS '97)* (April 1997).

[4] MALY, K. J., GUPTA, A. K., AND MYNAM, S. Btu: A host communication benchmark. Computer, pp. 66-74.

[5] http://www.nersc.gov/research/ftg/via/.

[6] VINTER, B. *PastSet a Structured Distributed Shared Memory System*. PhD thesis, Tromsø University, 1999.

[7] VON EICKEN, T., CULLER, D. E., GOLDSTEIN, S. C., AND SCHAUSER, K. E. Active messages: A mechanism for integrated communication and computation. In Proceedings of 19th International Symposium on Computer Architecture, pages 256-266, May 1992.

A.10 Cluster Monitoring with Steps: Making the Application Behaviour Visible

Cluster Monitoring with Steps: Making the Application Behaviour Visible

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

larsab@stud.cs.uit.no, johnm@cs.uit.no, otto@cs.uit.no

Department of Computer Science,
University of Tromsø

Abstract

We introduce Steps*, an application behaviour monitoring environment for clusters of computers. Steps is designed to aid the user in instrumenting, visualize, and analyse the post mortem behaviour of parallel applications. Steps is based on combining logged data from low overhead event based* data logging wrappers *with a configuration map of the application specifying in detail how threads access data. Steps visulize an applications behavior in various ways, and this together with the knowledge an application programmer has of an application, are used to identify interesting behaviour characteristics. The user can then reconfigure the application and try again with the purpose of changing the behavior to get better performance. We have implemented a Wind Tunnel application and a barrier benchmark, and used Steps on them to validate our design by showing were to log data, what to visualize, and how the applications behaviour are reflected in the visualizations.*

1 Introduction

Writing distributed and parallel applications for clusters of scalar uni- and multiprocessors built from commodity hardware and software can be difficult. To use the available resources efficiently to get good scaling is even more difficult. A tool that can monitor the behaviour of an application or a system, aid in analyzing the results, and present them in a useful manner has been found to be useful [2]. In this paper we report on such a tool, Steps, and on our experience building and using it.

Steps support a post-mortem analysis and visualization of communication events recorded using the PATHS[4] cluster application middleware enhancement. We combine the event information recorded with Steps with PATHS meta-data on the actual configuration of an application in a distributed system (in our case this is a distributed system of three clusters). Steps then visualizes the behaviour in various graphical ways. We also show how the recorded data can be used to do a post-mortem global clock synchronization of communication events.

We report on our experiences building an analysis and visualization tool using a high-level lan-

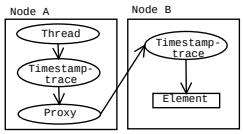


Figure 1. A simple path for a thread accessing an element on another node. Two trace wrappers are added to collect performance information.

guage and utilizing existing tools. We demonstrate how Steps can be used to find interesting communication patterns indicating potential performance problems in a simple two dimensional wind-tunnel simulator. We also show how we can extend Steps to find performance problems for applications with barrier synchronizations.

2 Research Platform

2.1 PastSet

PastSet [1][21] is a structured distributed shared memory system in the tradition of Linda [6]. A PastSet system consists of a number of servers hosting PastSet *elements*. An element is a named sequence of tuples of the same type. Tuples can be read from and written to the element using the *move*, *observe*, and *mob* (move-observe) operations. A process typically use more than one element for communication and synchronization with other processes. Elements can physically be located on different computers according to the distribution algorithm in use by PastSet.

2.2 PATHS

PATHS [3], is an extension of PastSet that enables mapping of threads to hosts at load time, selection of physical communication paths to each element, and distribution of computations along the path.

In figure 1 a path from a thread on node A is created to an element in node B. Each stage in the path is implemented using a *Wrapper* of a given type. A wrapper exports the same interface as a PastSet element, and is typically used to run code before and after forwarding the PastSet operation to the next stage in the path.

A path is specified by listing the stages from the thread to the PastSet element, specifying the wrapper type used at each stage as well as parameters used to initialize the wrapper.

The path in figure 1 consists in addition to the element, a proxy wrapper, which is used to access wrappers on remote nodes by specifying parameters such as the remote nodes name and protocols to use, and two trace wrappers. The path specification is given in figure 2. Default parameters are not necessary to include in the path specification.

A thread may use multiple paths to the same element, each of which can be specified and built dynamically. Paths can also be joined (forming a tree structure) to amortize communication overhead. It is also possible to use PATHS to create, for instance, tree barriers.

Applications can be mapped onto arbitrary cluster configurations by changing the path specification meta-data as long as only leaf wrappers are referenced in the application source code.

2.3 Using PATHS with PastSet applications

Applications using PATHS and PastSet often have the following parts:

- Multi-threaded application code written in C. The threads use PastSet operations to communicate with other threads in a location and access transparent manner.
- A meta-data file containing the path description and thread to host mapping. The meta-data is typically specified using a high-level language such as Python, and may include meta-code to do run-time decisions on path specifications.

```
make_path(stage('trace-timestamp', log_file='log1'),
          stage('proxy', host='B'),
          stage('trace-timestamp', log_file='log2'),
          stage('element'))
```

Figure 2. Path specification code for the path in figure 1.

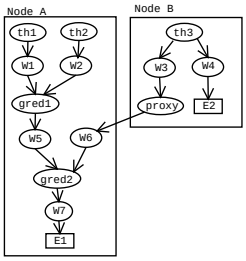


Figure 3. A more advanced path with tracing.

Path specifications may also be retrieved from path specification servers.

- A script to start the servers and processes on the different nodes.

Interfaces are also provided to allow applications written in high level languages to use PATHS and PastSet.

2.4 Instrumentation

The application is instrumented using PATHS trace wrappers. Any number of trace wrappers can be added anywhere in the path tree. The trace wrappers log the start and completion times of all PastSet operations that are invoked through the wrapper. Other parameters that may be useful in analyzing the traces and for debugging are also recorded.

We use the high-resolution Pentium timestamp counter (TSC) to get the timestamps. TSC records the number of elapsed processor cycles since the processor was booted. Reading this value has very little overhead [16]. We have measured the overhead of the timestamp trace wrapper to be less than one microsecond on the platform used [3].

New trace wrapper types can easily be added if additional information is needed.

Presently, the data is recorded in main memory and written to a log-file on disk when the program releases the path or finishes.

Each path is usually instrumented by adding trace wrappers before and after each (non-trace) wrapper. Figure 3 shows how paths are usually instrumented by adding trace wrappers before and after each non-trace wrapper. For operation on element E1 from th1, a total of 6 timestamps are collected at various levels.

We instrument more than we may need, since it is hard to know exactly what and how much information is needed. Thus the overhead is usually larger than strictly necessary.

3 Steps Overview

3.1 Log-file analysis

The application is analyzed by combining the data recorded using the trace wrappers with the same meta-data that was used for setting up the paths. The path description gives us a specification of the causality between trace wrapper events along a given path, while the Pentium time stamp counter used in the wrappers gives us a temporal ordering of the communication events internally in each node.

Below are some examples of what kind of information we can extract with the PATHS mapping shown in figure 3. As described in section

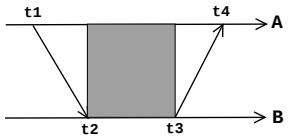


Figure 4. Clock synchronization.

2.4, the information logged at each timestamp wrapper is the start and completion time of each operation, and the operation type.

The data from the bottom-level wrappers (W7) can be used to calculate the number of unobserved tuples in an element (E1), which can be used to determine when a thread is blocked waiting for a tuple to arrive at the element.

For hierarchical global reductions and barriers[1], we combine the data in W1, W2, W5 and W6 in figure 3 to establish when the values for each partial sum arrives. This gives us the order of arrivals as shown in section 4.5.

We can calculate the overhead of a remote operation by subtracting the operation time recorded in W6 from the operation time recorded in W3. We can also use this information to do a post-mortem synchronization of the clocks in A and B (as described in section 3.2). By knowing the tuple sizes, we can also compute the effective bandwidth of the TCP connection.

It is also possible to calculate for a given time which threads are waiting for which other threads by examining, for instance, which threads are currently waiting for an observe operation to finish.

The meta-data has some limits. We cannot link recorded events to a specific location in the source code. Nor can we link recorded events to abstractions such as the different stages in the algorithm. Currently, the user must write a component to do this binding.

3.2 Global clock

In this section we will describe our post-mortem clock synchronization of the Pentium time stamp counters (TSC). It has no additional runtime overhead or perturbation since only existing data is used for the synchronization.

The approach is inspired by Lamports logical clocks [10], and by the approach described in [11] where they used an algorithm similar to Christian's algorithm [8] to synchronize the clocks on a Myrinet-based cluster.

The design is based on the following assumptions:

- The CPU clocks have high oscillator stability, hence the clock drift for a short period of time is roughly constant [11] [16].
- On a multiprocessor the TSC's are synchronized.
- The data is intended to be used for high-level visualization.

To synchronize the TSC for two nodes A and B, we use the following observation. If node A accesses an element on B, then we have a path with a wrapper ("proxy") transparently doing routing from A to B. If trace wrappers are added as shown in figure 1, then we have a wrapper W_1 before the proxy on node A, and a wrapper W_2 on node B after the proxy. When a thread on A does an operation on the element four timestamps will be recorded as shown in figure 4: start time of the operation in wrapper W_1 (t_1), start time of the operation in wrapper W_2 (t_2), completion time of the operation in wrapper W_2 (t_3), and the completion time of the operation in wrapper W1 (t_4). This information can be used to calculate the offset of the TSC's on the two nodes. Our current implementation uses a very simple scheme where the offset is the average of several offsets calculated using: $o = t_1 - (t_2 + c)$, where $c = ((t_4 - t_1) - (t_3 - t_2))/2$.

[1] Barriers are implemented using global reduction.

To synchronize the TSC's on all nodes, we create a graph that shows which nodes communicate with which other nodes. Then we used breadth first search to create a minimum spanning tree (MST) starting from a node selected as the reference node. For each pair of neighbors the difference between the TSC's is calculated (as described above), before the offsets are used to get an offset relative to the reference node. This offset is used to adjust the recorded timestamps.

The MST can be compared to a set of NTP [14] servers where the reference node is the primary server, and its children are secondary servers, and so on. The reference node should be choosen such that most of the nodes are as close to the 'primary server' as possible (i.e. the MST is as low as possible). Currently, the user uses his/ her knowledge of the application and topology to choose a good reference node.

The simple algorithm used to synchronize two nodes has been sufficient for our purposes, but could be replaced by an algorithm that uses data filtering to get a more accurate offset (e.g. the one used by NTP servers [14]). Also the accuracy of the synchronization is affected by operating system noice [16].

We can find if the global clocks on node A and B are synchronized with an offset less than the one-way latency, by calculating for each remote operation if $gt_1 > gt_2$ or $gt_3 > gt_4$, where gt_i are the four timestamps in figure 4 adjusted to the global clock.

The logged data from the barrier wait application is well suited to be used as input to the global clock algorithm, since the communication consist of mob operations (send and receive of a fixed size tuple). This implies that an equal amount of data is sent each way, giving a more accurate communication time. Also, all nodes are directly connected to the reference node (where the element is located) and there is little load on the computers. The windtunnel application on the other hand is worse. The communication consists of only moves and observes using large tuples (meaning that on each operation much data is sent one way, and no data sent the other way). Also there is no central node that can be used as a reference node (we get a MST with only one node per level).

We used performance data from the barrier wait and windtunnel applications to test the global clock synchronization. A 5 minute period was considered. The TSC's were synchronized once, after 2 1/2 minutes. All recorded timestamps were adjustet to the global clock and checked as described above.

In table 1 results using data from the barrier wait application and the windtunnel application are shown. The barrier wait was only run on one cluster, while the windtunnel was run both on one and on three clusters. The table shows the percentages of events that were correctly synchronized (i.e. where the offset was less than the one-way latency), and the range, median, mean and standard deviation of the misses (i.e. where the offset is larger than the one-way latency).

In table 1 we see that the number of correct synchronizations is dependent on the one-way latency. In the barrier wait application, small amounts of data is sent, hence the one-way latency is small (around 200 μs). In the windtunnel the one-way latency is much larger (from 1 ms to 200 ms).

We can also see that, as expected, the miss numbers are much better for the barrier wait than for the windtunnel. For the barrier wait we get a mean miss time of 750 μs (plus the one-way latency), and for the single cluster 8 ms, but the median is actually better (so is the correct rate). For the multi-cluster windtunnel the average miss time is increased to 45 ms, the median is 6 ms and the maximum miss time is 0.8 seconds. The increase was caused by problems synchronizing three pair of nodes.

We also did an experiment where only the communication events of a single step in the windtunnel and barrier wait computations was considered. The results are shown in table 2.

We can see that the number of correct synchro-

Offset	Barrier wait	Windtunnel	Windtunnel (multi-cluster)
Correct	0%	44%	18 %
Minimum	12 μs	0 μs	0 μs
Maximum	2644 μs	52140 μs	820677 μs
Mean	750 μs	8100 μs	45168 μs
Median	500 μs	332 μs	6136 μs
Stddev	710 μs	16024 μs	114842 μs

Table 1. Accuracy of computed clock offsets (one-way latency not added). Data for all communication events over a 5 minute period is used.

Offset	Barrier wait	Windtunnel	Windtunnel (multi-cluster)
Correct	50 %	50 %	47 %
Minimum	0 μs	0 μs	0 μs
Maximum	98 μs	4701 μs	345584 μs
Mean	55 μs	1933 μs	20667 μs
Median	13 μs	1600 μs	4182 μs
Stddev	43 μs	1275 μs	61975 μs

Table 2. Accuracy of computed clock offsets (one-way latency not added). Only data for the second step is used.

nizations increase, and the miss times are lower. Especially good are the numbers for the barrier wait application, and the numbers for the single cluster windtunnel are also acceptable. For the multi-cluster windtunnel the average and the maximum are still to high, but the mean is acceptable. Still the problems are caused by the three pairs described above.

For comparison, using NTP under optimal conditions (e.g. 100Mb/s LAN access to a primary server) the offset can be bound to the order of 1 millisecond, but can be far worse [20]. Using the Global Positioning System (GPS) based timing the offset can be improved to around 10 μs [16]. And by using special hardware such as the 'DAG3.2e' series measurements cards the offset can be reduced to 100 ns [12].

3.3 Visualization

For this paper we have visualized communication events. The visualization is focused on helping the programmer solve the problems encountered when balancing the workload and reducing time spent waiting at synchronization points.

The user using the visualizations must have knowledge about the communication patterns of the application. In our case, the user should understand the paths in use by the application.

The tools are not intended to visualize every aspect of the parallel program, to do that other tools must also be used. We envision the visualization tools to be used to get an idea of what to change in the path description to get better performance. We have found it useful to just discover that there is a performance problem even without actually understanding why as long as we can get better performance by trying a few changes to the paths in use.

We wanted a visualization tool supporting a high level view giving an overview of the logged data, while allowing the user to zoom in on interesting patterns. We use a Gantt chart showing communication and computation times for each thread. This allows the user to easily spot interesting synchronization patterns, potentially indicating performance problems like bottlenecks and unbalanced load sharing. A similar chart is used to show the time spent on doing an operation, and the time between each execution of the operation.

We use line charts, bar charts and tables with statistics to show data for high-level objects such as threads, elements, hierarchical barriers and hierarchical global sums. For each high-level object several graphs are drawn, e.g. for a thread, graphs for all PastSet operations on different element are drawn. The user can select which graphs to draw, and what type of operations to draw (e.g. only show observe operations on element A).

The user can point to a bar to get information about the thread, elapsed time and the current step. By clicking on a step in one bar the corresponding steps in the other bars are highlighted, helping the user see how far in the computation each thread has gotten. Also the user can zoom in on interesting areas, or select an area to get more information about it.

Steps also visualizes the path specifications themselves. This is useful when developing and debugging a path specification. It can also be useful to see the path specification when analyzing the performance.

3.4 Implementation

Steps is implemented using Python and Tkinter. Graphviz [9] is used to draw the path specification graphs, Blt.Graph [5] is used to draw the line and bar charts, statistics are calculated using the Python stats module [19]. Using a high level language and existing tools allowed us to easily develop a prototype.

Steps is implemented such that it is easy for application programmers to write modules for other types of visualizations. The programmer can use the analysis modules, chart modules and other useful libraries in Steps to analyze and visualize the data.

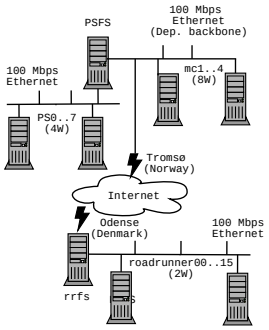


Figure 5. Overview of hardware platform.

4 Experiences Using Steps

We used a wind-tunnel simulator as a case study for how Steps can be used to find performance problems in real applications. Additionally, a simple application was devised to show how Steps can visualize barrier arrival times.

4.1 Hardware Platform

Figure 5 shows the clusters used in the experiment. The hardware platform consists of three clusters, each with 32 processors:

- Two-way cluster (2W): 16 * 2-way Pentium III 450 MHz, 256 MB RAM (Odense, Denmark)
- Four-way cluster (4W): 8 * 4-way Pentium Pro 166 MHz, 128 MB RAM (Tromsø, Norway)
- Eight-way cluster (8W): 4 * 8-way Pentium Pro 200 MHz, 2GB RAM (Tromsø, Norway)

All clusters use TCP/IP over a 100 Mbps Ethernet for intra-cluster communication. For communication between 4W and 8W we use the departments 100Mbit local area network. Communication between Tromsø and Odense is the departments Internet backbone (100 Mbps Ethernet). To access the 4W cluster communication must go through a 2-way Pentium II 300 MHz machine with 256 MB RAM. And to access the 2W cluster communication must go through a 2-way Pentium III 800 MHz machine with 256 MB RAM.

4.2 Windtunnel simulator

The windtunnel application models airflow represented as particles in a two-dimensional windtunnel. The flow is modeled as particles rather than using gas dynamics. This model is both simple and efficient, and is to some degree easy to parallelize. The idea is to model the two-dimensional space using a set of eight large matrices, representing the physical model, immovable particles, and particles flowing in different directions.

The space is divided along the X-axis into equal sized parts. There is one thread per part, pseudo code a the worker thread is shown in figure 6. Since the spaces are equal all threads do the same amount of work. This makes it easier to analyze and visualize the data. The load is distributed among the nodes in the clusters by assigning more threads to faster nodes. Each worker thread communicates with two neighbor threads. All threads communicate with each other using PastSet even when they are in the same process. This increases the number of events that can be monitored using PATHS.

The result of the simulation is a bitmap that shows the distribution of particles in the windtunnel. Each worker thread generates part of the resulting bitmap and sends it to a thread that merges and displays the bitmaps.

All worker threads are written in C using Pthreads, and they all run the same code. All threads get at initialization time information about the parts they work on, and top-level handles to paths. To simplify the discussion result bitmaps were not created.

We added trace wrappers to log all significant events (all calls to read or write to a PastSet ele-

```
barrier_sync();
TS(0);
for (i = 0; i < steps; i++)
    calculate particle movement
    do collision check
    send particles leaving my space to neighbors
    receive particles entering my space from neighbors
barrier_sync();
TS(1);
```

Figure 6. Pseudo code for a windtunnel worker thread.

ment). Therefore we log more data than needed, but we get flexibility to choose which data to be used for the post-mortem analysis and visualization.

4.3 Single cluster windtunnel experiment

We used the 4W cluster to document how the windtunnel scaled on one cluster. In summary, it scaled linearly from one CPU to 32 CPU's. We used Steps to examine the performance data for two configurations, one with one thread per CPU and one with four threads per CPU. The time taken to compute 500 steps for a problem size of 5312 x 2656 points was measured. In all experiments the load was evenly distributed among the nodes. We observed that when 200 steps were executed, having 4 threads per CPU was about 5% faster than having 1 thread per CPU. When the number of steps was increased to 500, they were equally fast. We detail these results in the following.

We also did experiments without tracing to determine the overhead of doing tracing. Without tracing the execution varied from being 1% slower to 1% faster. The implication is that the tracing overhead is not significant.

Figure 7 shows the computation (light gray) and communication (black) times for the configuration with one thread per CPU. There is one horizontal bar for each thread, that shows the communication and computation for each step. On the x-axis elapsed time is shown. Several thick black bars can be seen. By using the statistics table we can see that communication times varies from the average couple of milliseconds up to 50 milliseconds. By zooming in and counting the number of red lines on a node at a specific time we can see that we are not able to overlap communication with computation. Using this knowledge we created a new configuration with four threads per CPU.

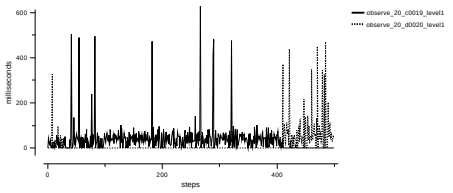


Figure 10. Observe operation times for worker thread 20.

Figure 8 shows the communication and computation times for the configuration with four threads per CPU. As expected the amount of black is larger since the relative amount of time spent on communication is larger since each thread has less work and spends more time blocked waiting for the others.

We can see in figure 8 thick black bars starting

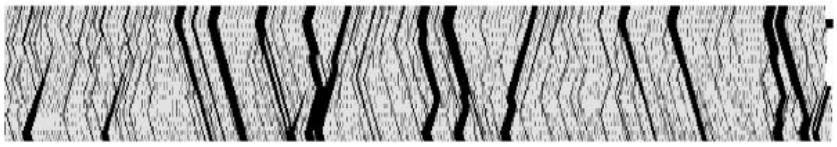


Figure 7. Communication (black) and computation (ligh gray) times for the 32 threads in the 'one thread per CPU' configuration (one cluster, 8 nodes, 4 CPU's per node).

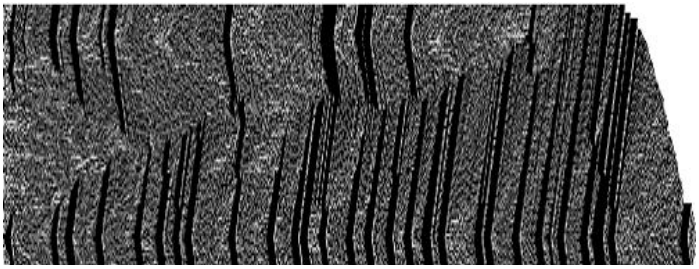


Figure 8. Communication and computation times for the 96 threads in the 'four threads per CPU' configuration (one cluster, 8 nodes, 4 CPU's per node).

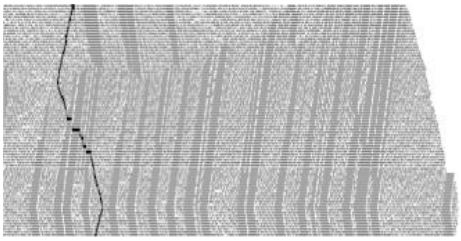


Figure 9. Part of figure 8 with with 250th step highlighted.

at the lower threads going upwards. By using the step information displayed when pointing at a bar, we can see that most threads are one step ahead of the thread below (neighbors can only be one step apart).

By looking at the completion times (where the bars end) we can see that the threads higher up finishes earlier than the threads further down. By

highlighting some steps we found that after 100 steps the threads had gotten roughly equally far, after 200 steps we could see a pattern looking like a “wavefront” emerging, and after 400 steps this pattern was clearly visible. The wavefront can be seen in Figure 9, where the 250th step is highlighted. Comparing figure 8 with figure 9 we can see that that the wavefront only affects threads affected by the thick black bars.

In figure 10 we can see that when the wavefront hits worker thread 20 (WT20) at about the 400th step, it shifts from spending most of its time waiting for data from the thread above (C, solid line), to waiting for data from the thread below (D, dotted line) (both elements and threads are on the same node). By highlighting the 400th step in the communication-computation view, we can see that this is where the wavefront hits worker thread 20. Also the time per step is slightly increased after the 400th step (not shown).

We have shown that by providing a high-level view it is possible to compare communication times versus computation times, find load unbalance, and find synchronization patterns that can indicate performance problems.

4.4 Multi cluster Windtunnel experiments

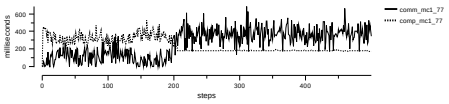


Figure 13. Communication (whole line) vs. computation (dotted line) times per step for worker thread 77. Wavefront hits at about the 200th step.

We used Steps to analyze the performance data collected for a simulation run on all three clusters for 500 steps, with problem size of 29680 x 14840 points. The amount of RAM on the 4W cluster did not support a larger problem size without demand paging. On each node in the 2W, 4W, 8W clusters we had 12, 8, and 19 threads respectively. When using the 4W and 8W clusters we measured linear speed up. Combining all three clusters gave just 1.54 times better performance than using the 2W and 4W clusters. Using the processor frequency of the computers as a guideline for expected performance, the three clusters should have been about 2.22 times faster than using just the 2W and 4W clusters.

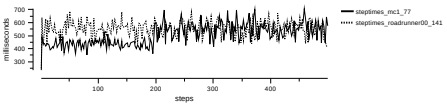


Figure 14. Time per step for worker threads 77 (whole) and 141 (dotted). Wavefront hits worker thread 77 at about the 200th step. The average of two samples is plotted.

Figure 11 shows the computation and communication times for the initial configuration. We can see blackness ’spreading’ from a thread in the middle (WT141) meaning more and more waiting for communication events to finish. WT141 is a thread on a node in Odense that does two moves and two observes to elements in Tromsø, i.e. it sends and receives data using the slow Internet between Tromsø and Odense. This results in a state where most other threads end up waiting for data from it (actually, each thread wait for a thread, that waits for a thread, ..., that waits for WT141).

In figure 13 we can see that the communication time per step (solid line) increases when the wavefront hits worker thread 77 located at one of the 4W nodes. In figure 14 we can also see that the time per step (solid line) increases and becomes roughly the same as for WT141.

Using this insight we created a couple of new configuration where we changed the load on the clusters, the load on the Odense-Tromsø nodes,

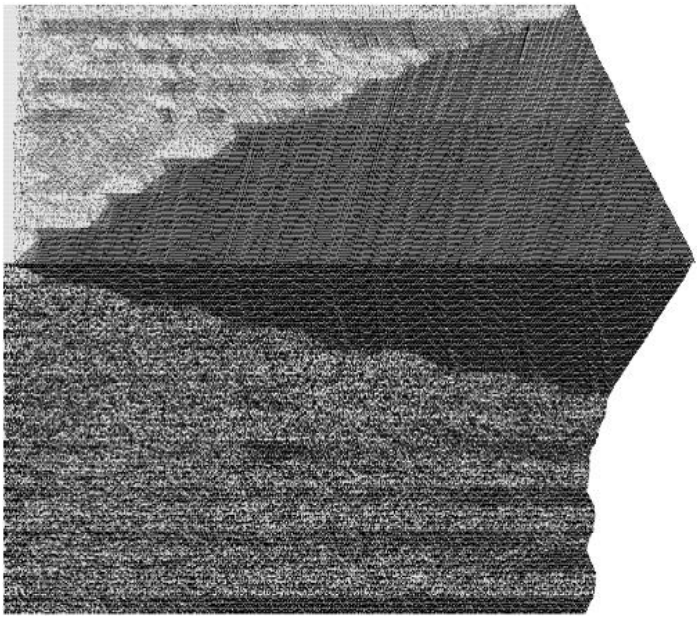


Figure 11. Communication and computation times for the 332 threads in the 'multi-cluster' configuration (3 clusters, 28 nodes)

and the data placement. But with no results, since the problem was due to limited bandwidth.

4.5 Barrier arrival times benchmark

In this section we will demonstrate how we used Steps to examine barrier arrival times using a Gantt chart. In the experiment there were 32 threads, one thread waited for a random time (0 to 8 seconds), the others waited for shorter random time (0 to 2 seconds). The hierarchical barrier had two levels where each node had a local barrier reporting to a single root barrier. The experiments were run on 8-way cluster.

In figure 15 one horizontal bar per thread, shows the barrier wait times (black) and the times between each synchronization (light gray). We can see that the amount of black is much smaller in one of the bars than on the other bars. Also in that bar the wait time often is so small that it is not visible. Combined with our knowledge about the application we can conclude that 31 of the threads often wait for the last thread to arrive at the barrier.

It is also possible to search for the thread that most often contributes latest to a global sum or arrives latest at a barrier. Using this method we can (for this artificial case) see that at the global barrier the signal from the local barrier containing the odd thread arrives latest in 99% of the cases. And at the local barrier on the node with the odd thread, the odd thread arrives latest in 62% of the

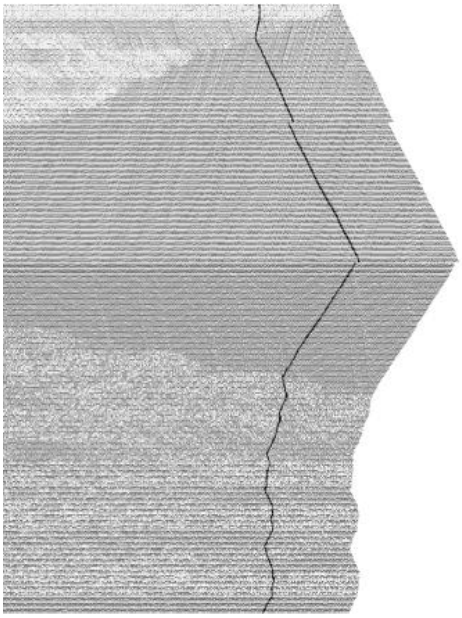


Figure 12. Figure 11 with step 427 highlighted.

Figure 15. Barrier wait time (black) and the time between each synchronization (light gray) for the 32 threads in the barrier wait application.

synchronizations.

4.6 Visualizing windtunnel path specifications

In this section experiences in visualizing the communication paths for the windtunnel are described. The path map used was for a configuration with a total of 30 nodes, 96 processors, 332 threads, 1321 elements and (about) 2750 wrappers.

Figure 16 illustrates some of the problems encountered. When all details are displayed the graph has about 3000 nodes and 3000 edges. With

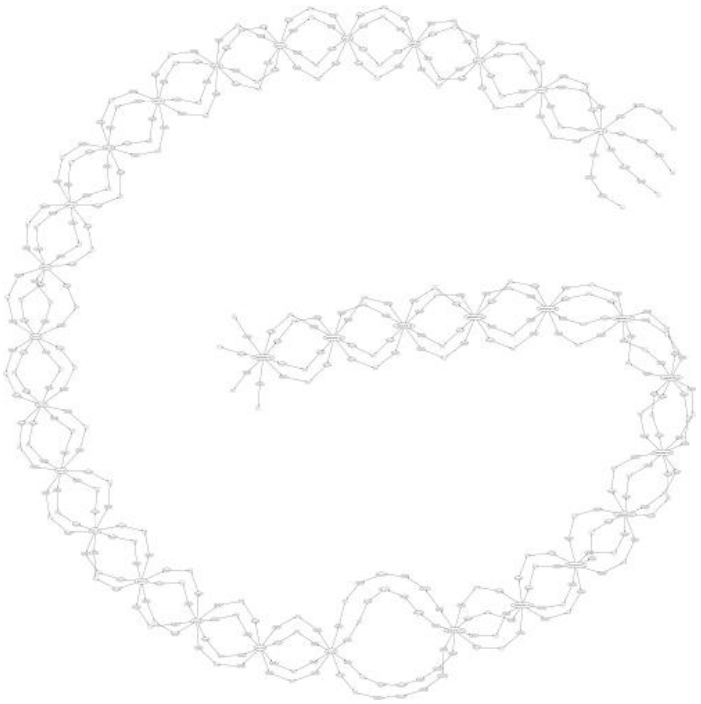


Figure 16. Path specification graph for the multi-cluster windtunnel application (only wrappers for two nodes are shown).

a typical 21 inch 1280*1024 display, the user can only see a subset of the path graph at a time. To examine different parts of the graph, the graph must be zoomed or the window must be scrolled. The small display makes it difficult to get an overview of the graph, and to find specific wrappers in the graph. A really large (several meters) high resolution (6000*4000 or better) display would have been interesting to use, but at the present time we do not have such a tiled display wall available.

Another problem was the time required to create the maps. We used Graphviz to calculate the node and edge placement on a Pentium III 600 MHz with 384 MB RAM. It took about 15 seconds to generate a graph with two nodes. A graph with 8 nodes took over 65 minutes, and we were not able to create a graph with 28 nodes without crashing.

4.7 Observations

By instrumenting liberally, several hundred megabytes of data is easily accumulated. It can be time consuming to move the data to a central node for analysis. We are working on moving

only the data needed. We would also like to get the flexibility offered by SQL when analyzing the data. But we found that using a database resulted in bad performance when accessing and moving data. This was probably caused by transaction support and the schema used in the database we had available.

When analyzing and visualization the performance, we found it useful to be able to select in a hierarchical manner the nodes, threads and (data) elements to visualize. This made it easier to get an overview and understand what the data represented. Before opening the visualization window the set of threads, elements, barrier-levels and wrappers to show must be selected. Afterwards the user can interactively select the graphs to be drawn.

We also found the ability to extend the functionality of the views useful. For example when drawing line charts, our data tended to differ by several orders of magnitude. We solved this problem simply by writing a function that allowed the user to plot the average of several points. We could easily add new functionality since the visualization modules were implemented in a high-level language.

It was difficult to name specifics parts of hierarchical global sums-, and tree barriers. The problem being that the trees are unbalanced, and can be very wide, this makes it difficult to find a simple and useful naming approach. Presently we just give the user a dialog box where he/ she could specify the barrier name, barrier level, nodes to consider, and nodes not to consider. A graphical interface would have made this easier.

Also it is difficult to give short and meaningful names to the graphs. We have used long descriptive names that contains the data type (e.g. move operations), thread name, element name, and level.

The Gantt chart module implemented in Python did not scale very well. It was too slow to be useful when visualizing the data from 300 threads. We believe the problem was our prototype implementation.

5 Related work

Analysis and visualization of shared virtual memory is described in [11] where monitoring code is embedded in programmable network interfaces to collect network-level data. This data is tied to higher level software events. In [15] monitoring is used to allow shared memory programs to tune their memory performance. And in [23] the underlying system's cache coherence protocol is used to detect data sharing patterns that indicate potential performance bottlenecks in shared memory parallel programs. As our applications use structured shared memory, the significant events are the latency of read and write operations. In addition, we have explicit meta-data available about the communication paths, making it easier to tie trace data to higher-level software events.

Paradyn [13]is a performance measurement tool for heterogeneous parallel and distributed programs. It instruments, analyzes and visualizes the performance at runtime. The search for bottlenecks is automated. Users can extend Paradyn by adding instrumentation code and visualization modules. Pablo [17] is a performance instrumentation and analysis toolkit for parallel systems. Pablo allows the user to do extensible performance analysis by using a coarse-grained graphical data flow programming model.

The Network Weather Service (NWS) [22] is a monitoring system for grid-style computing. NWS uses active probing to measure among other things the available network bandwidth and TCP latency. This information is used to forecast future network performance. We can measure network connections using both TCP and other protocols, and do this without probing. However, we have no support for forecasting.

Prism [18] is a debugger for multi-process MPI programs that supports performance analysis and (application data) visualization. They do post-

mortem clock synchronization similar to ours, but collect the data by running a separate MPI job. As perturbation is an issue, the offsets are recalculated every 3 minutes interpolating values between intervals. In contrast we can calculate the intervals at the same rate as messages are sent.

6 Conclusions and Future work

In this paper we present our experiences building Steps, a tool to analyze and visualize the performance of parallel applications.

We collect performance data, with low overhead, using PATHS. We use the communication path specifications to tie low-level data to high-level abstractions, such as threads, TCP connections, global sums and barriers. This allows us to examine, say, barrier synchronization in great detail. We also do a post-mortem synchronization of the Pentium timestamp counters on the nodes using the gathered data.

Steps is implemented in a high-level language using existing Python modules and tools. Combined with the flexibility offered by the large amounts of data collected, we can easily experiment with different views and functionality. However, we are still investigating how to get the visualizations to scale.

We have shown that even a simple well understood application can have unexpected behaviour, and that Steps can be used to identify some of these problems.

We are currently working on collecting, analyzing and visualizing the data at run-time. An important part of this work is data reduction and data collection using PATHS. We are also investigating other ways we can use the path specifications with the gathered data to find additional performance indicators. Work should also be done to improve the visualization, especially providing views at several abstraction levels.

We are currently using PATHS and Steps to analyze the performance of a real scientific application, the ELCIRC [7] river simulation.

7 Acknowledgements

The authors wish to thank Brian Vinter for interesting discussions.

References

[1] ANSHUS, O. J., AND LARSEN, T. Macroscope: The abstractions of a distributed operating system. *Norsk Informatikk Konferanse* (October 1992).

[2] ARPACI-DUSSEAU, A. C., ARPACI-DUSSEAU, R. H., CULLER, D. E., HELLERSTEIN, J. M., AND PATTERSON, D. A. Searching for the sorting record: Experiences in tuning NOW-Sort. *Proceedings of the SIGMETRICS symposium on Parallel and distributed tools (SPDT 98), USA* (1998), pp. 124–133.

[3] 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 application. *Norsk Informatikk Konferanse* (November 2001), 164–175.

[4] BJØRNDALEN, J. M., ANSHUS, O., VINTER, B., AND LARSEN, T. Comparing the performance of the pastset distributed shared memory system using tcp/ip and m-via. In *Proceedings of WSDSM'00, Santa Fe, New Mexico* (May 2000).

[5] http://www.ifi.uio.no/ hpl/Pmw.Blt/doc/.

[6] CARRIERO, N., AND GELERNTER, D. Linda in context. *Commun. ACM 32*, 4 (April 1989), 444–458.

[7] http://www.ccalmr.ogi.edu/CORIE/.

[8] CRISTIAN, F. Probabilistic clock synchronization. *Distributed Computing 3* (1989), 146–158.

[9] http://www.research.att.com/sw/tools/graphviz/.

[10] LAMPORT, L. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM* (1978).

[11] LIAO, C., JIANG, D., IFTODE, L., MARTONOSI, M., AND CLARK, D. W. Monitoring shared virtual memory performance on a myrinet-based pc cluster. In *ICS 1998* (1998).

[12] MICHEEL, J., GRAHAM, I., AND DONNELLY, S. Precision timestamping of network packets. In *Proceedings of the SIGCOMM IMW* (2001).

[13] 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 tools. *IEEE Computer* (1995).

[14] MILLS, D. L. Improved algorithms for synchronizing computer network clocks. *IEEE Transactions on Networks* (1995).

[15] NIKOLOPOULOS, D. S., PAPATHEODOROU, T. S., POLYCHRONOPOULOS, C. D., LABARTA, J., AND AYGUADÉ, E. A case for user-level dynamic page migration. In *ICS 2000* (2000).

[16] PÁZTOR, A., AND VEITCH, D. Pc based precision timing without gps. *ACM SIGMETRICS 2002* (2002).

[17] 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. *IEEE Proc. Scalable Parallel Libraries Conf.* (1993).

[18] SISTARE, S., DORENKAMP, E., NEVIN, N., AND LOH, E. Mpi support in the prism programming environment.

[19] http://www.nmr.mgh.harvard.edu/Neural_Systems_Group

[20] UIJTERWAAL, H., AND KOLKMAN, O. Internet delay measurments using test traffic: Design note. *Tech. Report RIPE-158, RIPE, NCC* (June 1997).

[21] VINTER, B. *PastSet a Structured Distributed Shared Memory System*. PhD thesis, Tromsø University, 1999.

[22] WOLSKI, R., SPRING, N., AND HAYES, J. The network weather service: A distributed resource performance forecasting service for metacomputing. *Journal of Future Generation Computing Systems* (1999).

[23] XU, Z., LARUS, J. R., AND MILLER, B. P. Shared-memory performance profiling. In *ACM PPoPP' 97* (1997).

Appendix B

PATHS Implementation

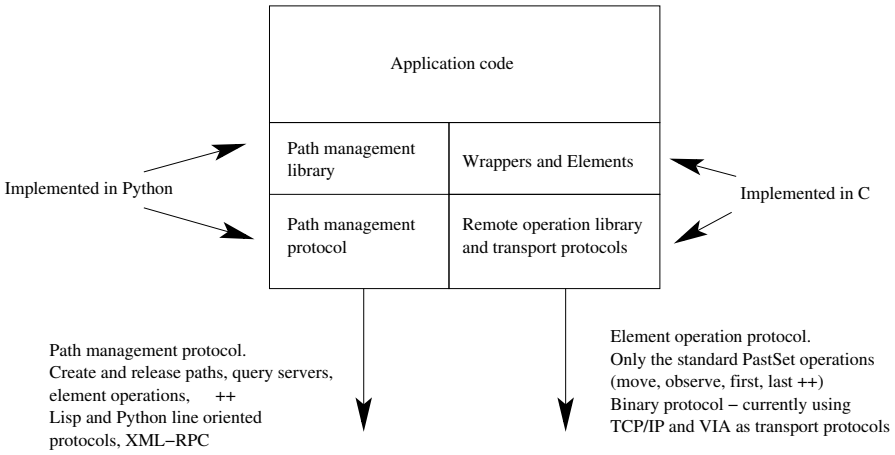


Figure B.1: Organization of a PATHS enabled client application

This appendix provides some additional information about the current PastSet/PATHS implementation.

B.1 Implementation

A PastSet client with PATHS follows the architecture in figure B.1.

The path management library is implemented in Python. This was partly done to ease prototyping. Using Python also provided a simple option for allowing the users to provide their own path specification modules: a Python module is loaded at initialization time. Which module to load can be specified by the user, allowing the user to override the default behaviour.

The Wrappers, Elements and the RPC library are implemented in C. This provides efficient execution of operations along a path, as Python is only invoked when managing a path, and not when invoking PastSet operations on an existing path.

One of the reasons for implementing the Wrappers and Elements in C was to allow paths to be specified that can enter the operating system kernel. C is also relatively easy to integrate with other programming languages.

PastSet Elements are currently hosted at user-space within the PastSet server or client application.

Path management and PastSet functions are available both from C and Python, and created paths can be handed freely between the languages. This allows the user to write applications using various combinations of the languages.

The path management protocol used is either a simple line-based text protocol (using Lisp or Python syntax), or XML-RPC. One of the reasons for adding XML-RPC was to open up for implementing tools and application programs as Javascript or Java Applets running in a web browser. XML-RPC is also supported by a large number of programming languages [59], allowing for simple interfacing with those languages.

B.1.1 PastSet servers

The PastSet/PATHS library includes the service routines used to implement PastSet servers. Any PastSet application can act as a PastSet server by calling a function which spawns a service thread and initializes the server functionality.

There can only be one PastSet server on each host, however, as addressing of individual servers within one host has not been implemented. Thus, either one of the client processes acts as a server for the other processes, or a dedicated server process is started.

B.2 Size of implementation

The latest distribution consists of 7334 lines of Python and C source code, including comments and white space. This includes source code for the PastSet servers, 9 wrapper classes, two different Element classes, path management server and client library for the XML-RPC and Python line-based protocols, and some simple regression testing code. Some experimental code not necessary for the distribution is also included.

The wrapper implementations range from about 100 to 300 lines depending on the functionality implemented and the amount of comments.