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 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.

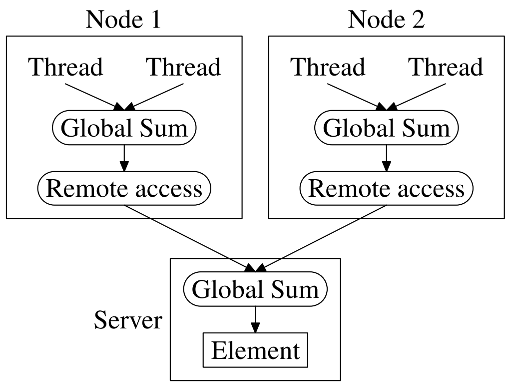

Figure 1: Four threads on two hosts accessing shared element on a separate server. Each host computes a partial sum that is forwarded to the global-sum wrapper on the server. The final result is stored in the element.

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.

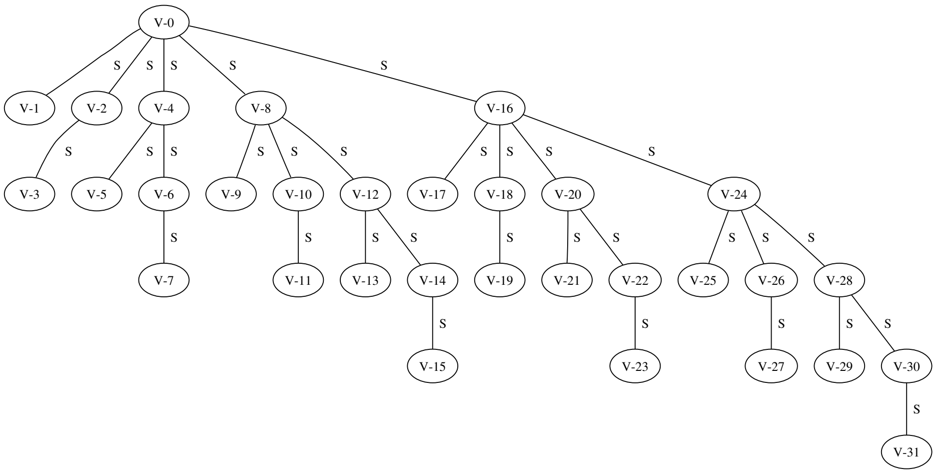

Figure 2: Log-reduce tree for 32 processes. The arcs represent communication between two nodes. Partial sums are computed at a node in the tree before passing the result further up in the tree.

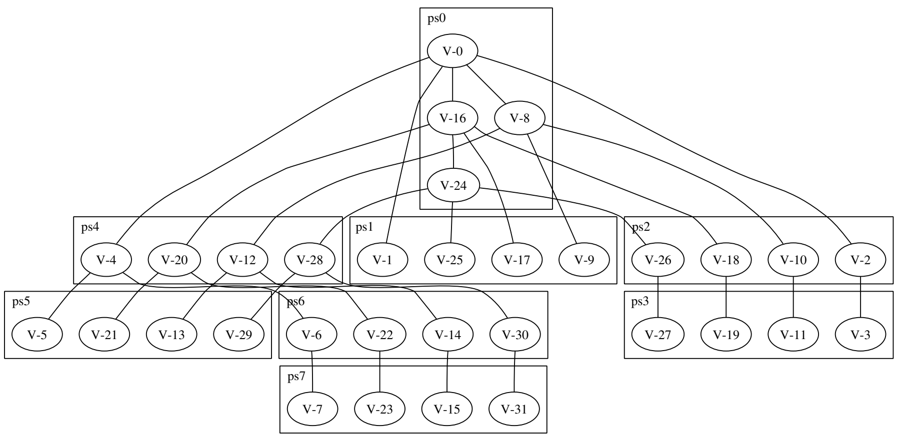

Figure 3: Log-reduce tree for 32 processes mapped onto 8 nodes.

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:

- 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.

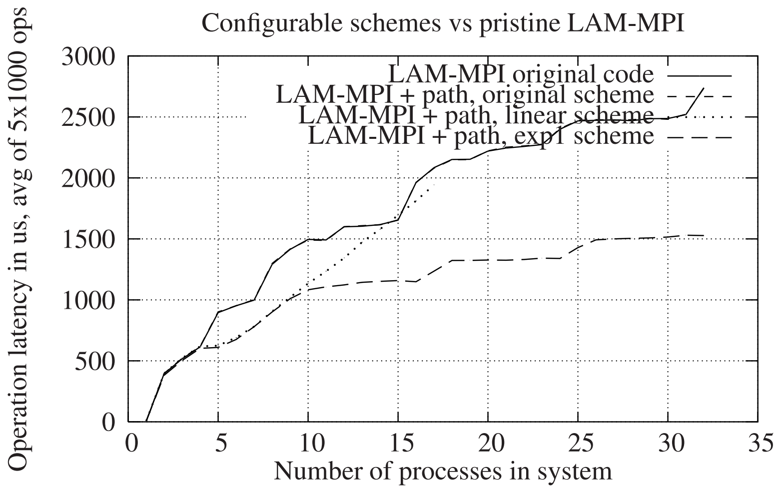

Figure 5: `Allreduce`, 4-way cluster

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.

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.

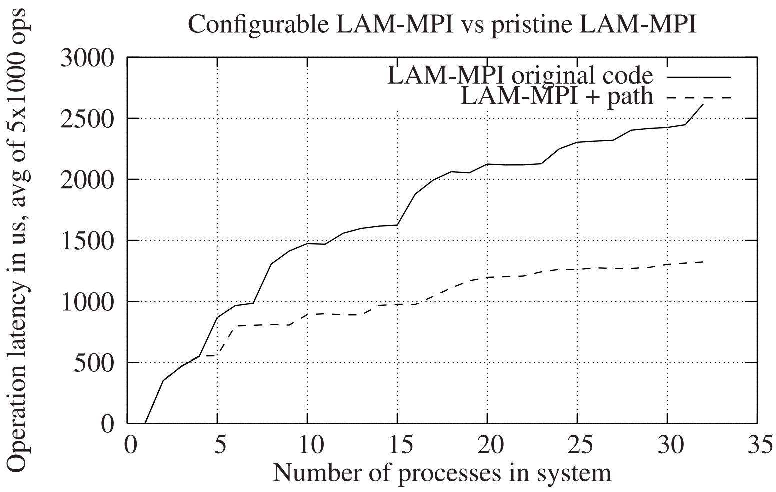

Figure 6: `Allreduce`, 8-way cluster

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.

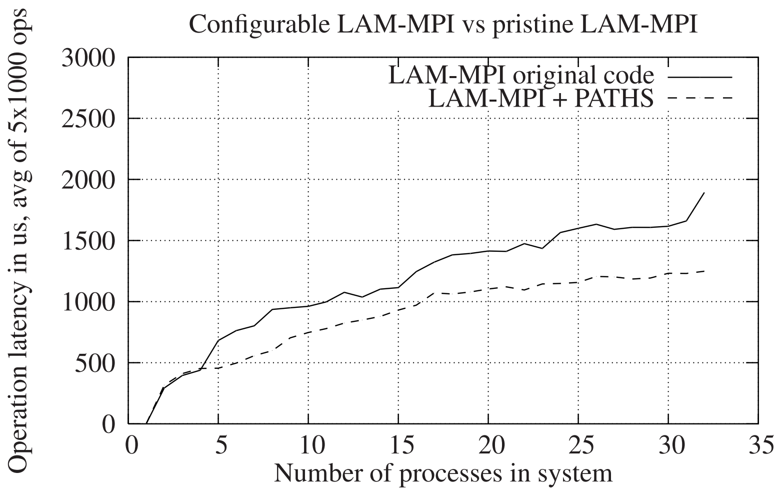

Figure 7: `Allreduce`, 2-way cluster.

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.

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 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

[2]either because of lack of funding, broken nodes, or nodes allocated for other purposes

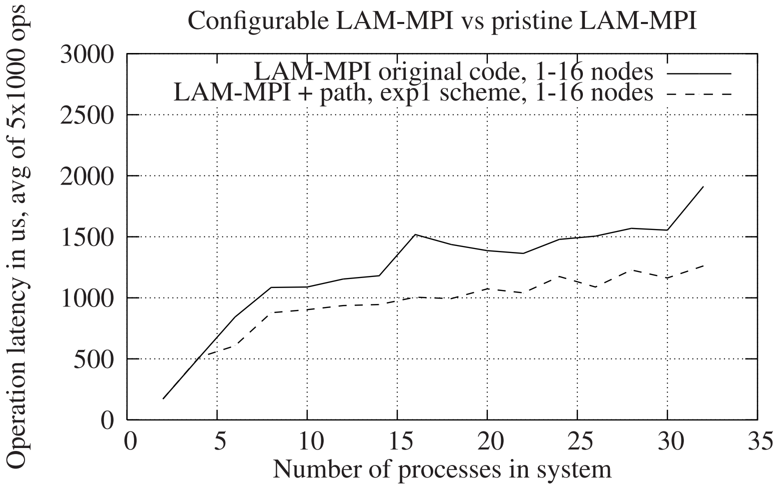

Figure 8: `Allreduce`, 2-way cluster, scaling the number of nodes from 1 to 16. Each node added adds two new processes in the experiment.

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.