Evaluating the Performance of the Allreduce Collective Operation on Clusters: Approach and Results

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

Abstract

The performance of the collective operations provided by a communication library is important for many applications run on clusters. The communication structure of collective operations can be organized as a tree. Performance can be improved by configuring and mapping the tree to the clusters in use. We describe and demonstrate an approach for evaluating the performance of different configurations and mappings of allreduce run on clusters of different size, consisting of single-CPU hosts, and SMPs with a different number of CPUs. A breakdown of the cost of allreduce using the best configuration on different clusters is provided. For all, the broadcast part is more expensive than the reduce part. Inter-host communication contributes more to the time per allreduce than the synchronization in the allreduce components. For the small messages sizes used (4 and 256 bytes), the time spent computing the partial reductions is insignificant. Reconfiguring hierarchy aware trees improved performance up to a factor of 1.49, by avoiding scalability problems of the components on SMPs, and by finding the right balance between available concurrency, load on 'root' hosts and the number of network links in a tree. Extending a tree by adding more threads, or by combining two trees does not have a negative influence on the performance of a configuration, but increasing message size does.

1 Introduction

Clusters are becoming an increasingly important platform for scientific computing. A large portion of the programs are written using a communication library such as MPI [18] or PVM [22]. Both provide collective operations, such as broadcast, reduce and allreduce, to simplify the development of parallel applications. Of the eight scalable scientific applications investigated in [29], most would benefit from improvements to MPI's collective operations. Also, the importance of collective operations is likely to grow, if the high performance overhead due to lack of scalability is reduced [28].

The communication structure of a collective operation can be organized as a tree, with threads as leafs. Communication proceeds along the arcs of the tree and a partial operation is done in each non-leaf node. In earlier work [2], we improved the performance of a collective operation up to a factor of two, by configuring the tree shape, and by modifying the mapping of the tree to the clusters in use.

In this paper we evaluate how, why, and by how much performance can be improved by reconfiguring the communication structure and mapping of collective operations to different clusters. We break down the cost of collective operations to guide future optimizations. Also, we evaluate the effect of extending trees by adding more threads and clusters, and by increasing message size.

We use the PATHS system [4] to specify the communication structure and mapping of collective operations. We limit our study to the allreduce operation. In allreduce, each thread has data that is reduced using an (associative) operation, followed by a broadcast of the reduced value. The implementation of the tree components and the communication protocols are not changed.

We provide an approach for analyzing the performance of different configurations, and we demonstrate the approach by analyzing different allreduce configurations on a blade cluster with ten uni-processor blades, a cluster of thirty two-way hosts, a cluster of eight four-way hosts, and a cluster of four eight-way hosts.

Reconfiguring hierarchy aware trees improves performance by a factor up to 1.49, by avoiding scalability problems of the components on SMPs, and by finding the right balance between available concurrency, load on root hosts, and the number of network links in a tree. A tree can be extended by adding more threads, and by using two single-cluster trees to create a multi-cluster tree without decreasing performance. However, increasing the message size may require a reconfiguration.

A breakdown of the cost of allreduce using the best configuration, shows that the broadcast part is more expensive than the reduce part. Inter-host communication contributes more to the time per allreduce than the synchronization in the allreduce components. For the small messages sizes used (4 and 256 bytes), the time spent computing the partial reductions is insignificant.

The rest of this paper proceeds as follows. Related work is discussed in section 2. Collective operation implementation, and our system for configuring them are described in section 3. Our monitoring and analysis approach are described in section 4. Section 5 describes the methodology for the experiments that are used to demonstrate the analysis approach in sections 6 and 7. Finally, in section 8 we conclude and outline future work.

2 Related Work

The MPI standard [18] includes the topology mechanism for remapping the ranks of processes according to a logical arrangement of communication specified as a graph. This may be used by the run-time system to aid in mapping the processes onto hardware, and as an advice for optimizing collective operations. However, current MPI implementations seems to make little use of this mechanism [26]. We also believe that using an implicit mechanism such as changing ranks to influence the mapping and structure of the collective operation tree is inadequate. Hence, the communication system should allow the communication structure and behavior to be inspected and explicitly mapped to the available resources.

There are several performance analysis tools for MPI programs [17]. Generally these tools monitor and analyze how the application use the communication system, and not what happens inside the communication system.

Many research projects have optimized MPI collective operations [8, 11, 12, 15, 21, 23, 27]. Our work is complimentary in that it reports detailed on where the time is spent inside an allreduce operation.

Vadhiyar et al. [27] describes an approach where experiments are automatically conducted on a system to find the best algorithm for creating the communication structure for a given collective operation. In our approach a new communication structure is created, based on an analysis of the performance of previous configurations, allowing for arbitrary communication structures that may not easily be created by an algorithm.

Kielmann et al. [12] shows an approach where they reduce the number of messages over WAN connections to reduce the latency of collective operations over wide-area links. However, we observed that for small messages, and with our cluster sizes, the number of roundtrip messages used to implement a collective operation was more important than the number of messages crossing the WAN link in the same direction [3].

In CC-MPI [11] the compiler can determine the communication requirements of an application, allowing special features, such as native broadcast or multicast, of switched Ethernet to be used in the implementation of collective operations. The allreduce operation that is optimized in this paper is not optimized in CC-MPI. Also, they find that for (MPI) broadcast with small message sizes a tree based unicast, similar to ours, has better performance than using reliable multicast.

In [21], Sistare et al. presents a two level collective operation tree for SMPs. The subtree on each host use the high-backplane bandwidth and shared-memory capabilities of SMPs. When small messages are used in the reduce operation, a spanning tree similar to ours is used. Similar hierarchy aware collective operations are also used in TMPI [23], and in MPI-StarT [8]. Tipparaju et al. [25] describes an allreduce implementation for SMPs that is more efficient than point-to-point communication.

LAM/MPI version 7.0 [14], also supports SMP aware collective operations.

Different implementations of locks and barriers on SMPs are examined in [16], and [13]. Our implementation is at a higher level. It use the Pthread library for implementing synchronization, and we do not consider architecture specific optimizations. We also do not consider how to optimize point-to-point communication using different network protocols or interconnects such as Myrinet and SCI.

A theoretical study, using the LogP model, is presented in [10]. However, mathematical models based on only a few network parameters does not take into account the overlap and variation in the communication that occurs in collective communications [27]. In [1] and [15] the collective communication structure is based on the measured communication time between two hosts. Our results shows that it is difficult to base the analysis only on communication time, since other factors such as the synchronization primitives in the implementation influence the performance of the collective operation, and also the communication time.

Factors not studied by us, but found to influence the performance of collective operations include: (a) group management, resource management and bookkeeping overheads [11], (b) life-span [11], and number of a network connections [23], (c) buffer size for collective communication [27], and (d) rank order with regard to the given topology [26].

Also, we do not evaluate the, potentially significant, influence of application load imbalance on collective operations [13]. Daemons and other applications running on a cluster can also reduce the performance of allreduce [19, 9]. We believe being able to analyze the performance of collective operations, inside the communication system, becomes even more important if the solutions to these problems are implemented.

3 Configuring Collective Operations

A communication library usually provide an API that the application programmer use to invoke collective and other operations. The implementations usually do not provide the user with the ability to inspect and adapt the implementation of the operations to the specifics of an application and the clusters used by focusing on parameters such as the number of threads, communication patterns and where threads and data are located.

Often a spanning tree is used to describe and implement the communication structure of collective operations (figure 1). For the actual communication between two nodes in the tree, point-to-point communication is used, usually implemented using TCP/IP. For some collective operations, like a reduce, the non-leaf nodes in the tree does some operation on the data received from its parents, before sending the result further down the tree. Synchronization is required in the non-leaf nodes. Exactly how this is done is dependent on the architecture of the run-time system (single-threaded, multi-threaded, event-based, etc).

Essential for the performance of a collective operation is the shape of the tree, and the mapping of the tree to the clusters in use [2]. To implement allreduce, LAM-MPI first use a reduce tree, and then a broadcast tree. Both use a linear scheme[1] up to and including four threads. For more than four threads, a logarithmic spanning tree is used, as shown in figure 1. The configuration in figure 1 is probably suboptimal, since many messages are sent across hosts using a (potentially) slow network, and because processing in the hosts is not overlapping as much as it could.

3.1 Configuring Collective Communication

To expose, and allow the collective operation tree to be better mapped for the clusters in use we have developed the PATHS system [4]. For communication, the parallel applications use PastSet [30], a structured distributed shared memory in the tradition of Linda [7]. Two threads communicate, in an access and location transparent manner, by writing and reading tuples from/to PastSet *elements*. The programming model used can be compared to message passing.

PATHS allows us to add *wrapper* code to be run before accessing the PastSet element. Also, the properties and mapping of all wrappers on a *communication path* from a thread to a PastSet element can be specified. The wrapper code can be used to implement collective operations. Figure 2 shows pseudo code for an allreduce wrapper doing global sum (gred).

[1]Every thread communicates directly with thread 0.

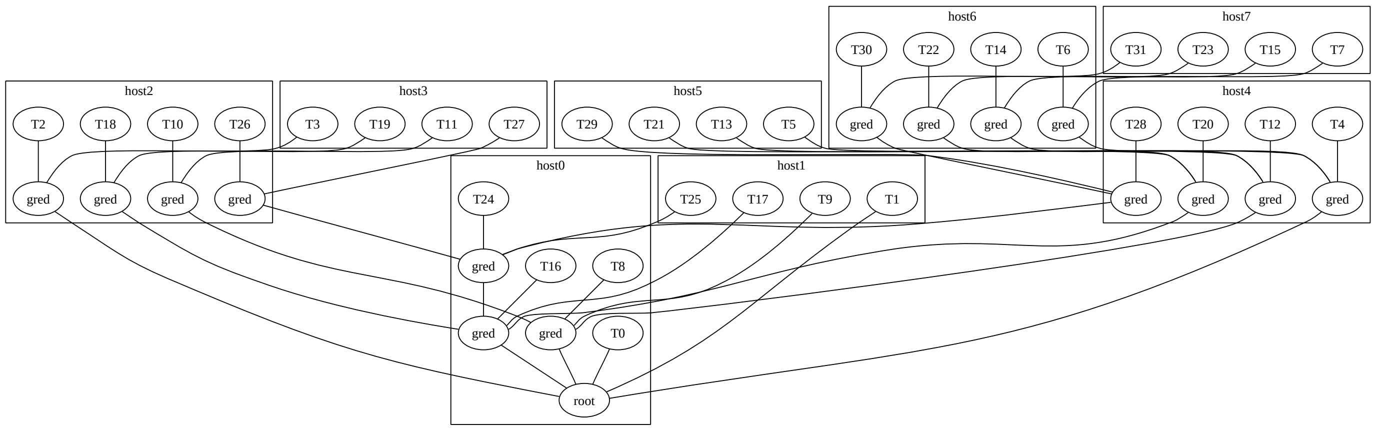

Figure 1. Allreduce tree for 32 threads mapped onto 8 hosts. The arcs represent communication. Partial sums are computed at non-leaf nodes in the tree before passing the result further down in the tree.

```
acquire_mutex();
do_sum();
if not last thread
  condition_wait();
if last thread
  condition_broadcast();
release_mutex();
return global_sum;
```

Figure 2. Pseudo code for gred wrapper.

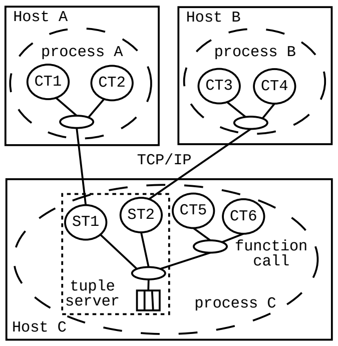

Figure 3. An application with 6 computational threads (CT) and 2 TCP/IP service threads (ST) using a collective operation tree implemented using wrappers (small ovals). The result is stored in a PastSet element.

Figure 3 shows the PATHS/PastSet runtime system. Applications are usually written as multi-threaded processes, which are mapped to the host such that there is only one process per host. In each process there is also a PastSet tuple server, hosting PastSet elements, and service threads that provide communication for remote clients. Each communication link is explicitly defined, and has its own TCP/IP connection. The client side stub is implemented by a *proxy* wrapper.

When a thread invokes an operation using a given path, the wrappers (on the path) are run in the context of the initiating thread until a wrapper on another host is invoked. This wrapper (and the other wrappers on that host) are run in the context of the threads serving the given connection.

The runtime system is written in C. The code for specifying and setting up the paths is written in Python.

Figure 4, shows how the allreduce operation can be implemented using PATHS. The participating threads are the leaf nodes in the tree, and the root is a PastSet element (CoreElm). The threads send data down the path by invoking the wrappers. All but the latest arrival are blocked (on a Pthread synchronization variable) after doing the reduce operation in the allreduce wrappers gred1–gred4. The latest arrival continues down the path. The final reduced tuple is stored in the PastSet element, before it is broadcasted by return-

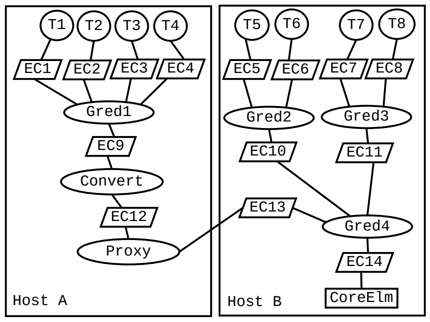

Figure 4. An allreduce tree used by threads T1–T8 instrumented with event collectors (EC1–EC14).

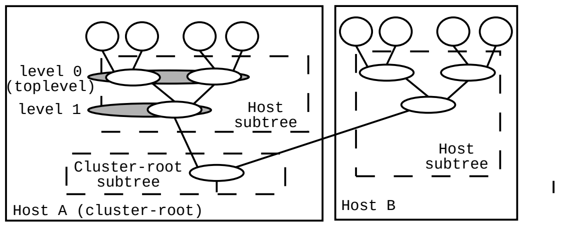

Figure 5. A collective operation tree used by four threads (circles), implemented using wrappers (ovals).

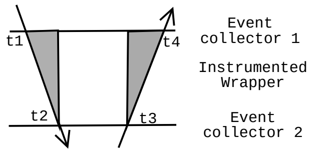

Figure 6. Timestamps recorded by two event collectors for an instrumented wrapper.

ing the tuple up the tree. When an allreduce wrapper receives the result tuple, all blocked contributors are awakened, and each return with a copy of the tuple. The reason for having three partial allreduce wrappers (gred1–gred3) is to improve scalability.

All path specifications, including collective operation path specifications, are stored in a *pathmap*. By reconfiguring this map, the shape of the collective tree, the parameters of the nodes in the tree, and the mapping of tree nodes to cluster hosts can be changed without modifying the application.

An initial pathmap is generated by a Python script that takes as input three mappings: (1) An *application mapping* describing which threads access which elements (including elements used for collective operations). (2) A *cluster mapping* describing the topology and the hosts of each cluster. (3) An *application to cluster mapping* describing the mapping of threads and elements to the hosts.

For the initial pathmap the collective operation trees are created in a hierarchical manner by first creating a *host subtree* for each host, and one *cluster-root subtree* on one of the hosts. Then the trees are connected to the root subtree. Proxy wrappers are used to bind together the trees over network links. A subtree can have multiple levels of wrappers doing partial allreduce, or only one level with one wrapper. The result is a tree as shown in figure 5.

4 Performance Analysis Approach

We use the EventSpace [6] system to monitor the communication behavior of collective operation trees. The wrappers in a collective operation tree are instrumented by inserting *event collectors*, implemented as PATHS wrappers, before and after each wrapper. In figure 4, an allreduce tree used by threads T1–T8 is instrumented, by event collectors EC1–EC14. Each event collector records data about communication events. The recorded data is stored in memory and written to trace files when the paths are released.

For each instrumented wrapper four timestamps (t_1, t_2, t_3 and t_4) are recorded, using the high-resolution Pentium timestamp counter[2] (figure 6).

The overhead of a single event collector is measured to be $0.5\,\mu s$–$1.7\,\mu s$ on a 200 MHz Pentium Pro [5]. Compared to the hundreds of μs per collective operation, the overhead is low. Hence, we assume the timestamp represent the time when the instrumented wrapper was entered and exited.

The performance analysis is done post-mortem. We

[2]The PastSet operation type and arguments are also recorded but are not used in this paper.

do not adjust the recorded timestamps for clock drift, since we have not seen any indications of clock drift during the analysis (most experiments are run for less than one minute). Also, we do not synchronize the Pentium timestamp counters on the hosts, as it is difficult to achieve the needed accuracy (tens of μs) without using special hardware such as GPS or special measurement cards [20].

For each instrumented wrapper we use the four timestamps collected for the wrapper to calculate:

Wrapper latency $(t_4 - t_1) - (t_3 - t_2)$, the total time spent in the wrapper.

Down latency $t_2 - t_1$, the time spent in the wrapper when moving down the path.

Up latency $t_4 - t_3$, the time spent in the wrapper when moving up the path.

Since the timestamp counters on two hosts are not synchronized the down, and up latency are each set to be *wrapper latency / 2*, for wrappers with event collectors on two hosts, such as a proxy. For the bottommost wrapper only wrapper latency is calculated, since there is no event collector below the wrapper.

Allreduce wrappers, or gred for short, have multiple children that contribute with a value to be reduced. The contributor can be a thread or data from another gred wrapper (e.g. in figure 4, threads T5 and T6 contributes to gred2, while gred1, gred2, and gred3 contributes to gred4). There is one event collector on the path to the parent that collects timestamps t_2 and t_3, while the paths from the P parents each have an event collector collecting timestamps $t_{1,i}$, and $t_{4,i}$.

We define the *down latency* for a gred wrapper to be $t_2 - t_{1,l}$, the down latency for the last arrival l. While the *up latency* is $t_{4,f} - t_3$, the up latency for the first departurer f. In addition, for each of the P contributors the following are calculated:

Arrival order distribution For a given number of collective operations, the number of times the contributor arrived at the gred wrapper as the first, second, and so on.

Departure order distribution For a given number of collective operations, the number of times the contributor departed at the gred wrapper as the first, second, and so on.

Arrival wait time $t_{1,l} - t_{1,i}$. The amount of time the contributor i had to wait for the last contributor l to arrive. The wait time is a function of the arrival order.

Departure wait time $t_{4,i} - t_{4,f}$. Elapsed time since the first contributor f departed from the gred wrapper, until contributor i departed. The wait time depends on the departure order.

Wrapper latency $(t_{4,i} - t_{1,i}) - (t_3 - t_2)$, the total time spent in the gred wrapper for contributor i.

For a collective operation tree, we calculate similar metrics as for a single gred-wrapper. The timestamps are collected by event collectors above the toplevel gred wrappers, and below the root-wrapper. For example, for the tree in figure 4, $t_{1,i}$ and $t_{4,i}$ are collected by EC1–EC8, while t_2 and t_3 are collected by EC14. As discussed above, we do not have accurate enough clock synchronization to calculate up and down latencies for a tree spanning over multiple hosts, so we do this by summing the latencies of all wrappers in the path. A similar analysis can also be done for a subtree, by selecting the event collectors above the toplevel of the subtree, and below the subtree root.

For the analysis, we often divide the path from a thread to the PastSet element into a down and up path (each has the same wrappers). For each thread we calculate the time spent in different *stages* when moving up and down the path. The stages for the down path are: down latencies, arrival wait times (for gred wrappers), and wrapper latency for the core stage (the bottommost wrapper). The up path stages are: up latencies, and departure wait times.

To calculate the time a thread spent in a specific part of the tree, we add together the time per stage for all stages in that part of the tree. Usually the mean time per stage is used, and arrival wait times are not added, since these reflect the time faster threads must wait for slower threads, thus canceling the difference between fast and slow threads. Also, arrival wait time can only be improved by improving the applications load balance, or by improving another part of the tree.

The time spent in various stages of the tree can also be used for a hotspot analysis.

5 Experiment Methodology

5.1 Hardware

The hardware platform comprise four clusters:

8W: Four eight-CPU Pentium Pro 200 MHz, 2 GB RAM.

4W: Eight four-CPU Pentium Pro 166 MHz, 128 MB RAM.

NOW: 30 dual-CPU Pentium II 300 MHz, 256 MB RAM.

Blade: 10 single-CPU Mobile Pentium III 900 MHz, 1024 MB RAM.

The clusters use TCP/IP over a 100 Mbps Ethernet for intra-cluster communication. The intra-network for 8W, 4W and Blade has no other traffic. The NOW[3] hosts are connected through the departments 100 Mbps Ethernet. There were no other users on the NOW. The clusters are connected through the departments 100 Mbps Ethernet. Communication to and from the 4W cluster goes through a two-way Pentium II 300 MHz with 256 MB RAM, while the 8W hosts are directly accessible. There is no background workload on the cluster hosts. However, there is other traffic on the departments network.

The operating system on all cluster is Linux, version 2.2.14 on 4W and 8W, and 2.4.20 on Blade and NOW. The compilator on 4W and 8W was gcc 2.96.2, and gcc 3.2.2 on Blade and NOW. For all clusters the optimization-flag ’-02’ was used. On all TCP/IP connections the Nagel algorithm is disabled to ensure that even small data packets are sent immediately.

5.2 Allreduce Benchmark

The allreduce benchmark, Gsum, measures the time it takes T threads to do N allreduce operations. The allreduce computes a global sum. The number of values to sum is equal to the number of threads, T. Pseudo-code is shown in figure 7. The threads alternate between using two identical allreduce trees to avoid two allreduce calls to interfere with each other.

[3]NOW is the departments undergraduate laboratory.

```
// synchronizes all threads
barrier();
start_clock();
for (i = 0; i < N; i++)
  allreduce(tree1);
  allreduce(tree2);
stop_clock();
```

Figure 7. Pseudo code for Gsum benchmark.

Only 4 and 256 byte messages are used, since the collective operations used by most scientific applications have small message sizes (less than 256 bytes), and the message size does not change neither with the number of threads or with the problem size [28]. Unless otherwise noted, 4 byte messages are used.

The Gsum benchmark was run for 20 000 iterations on the 4W cluster, and for 25 000 iterations on the other clusters. The Gsum execution time has a small standard deviation (less than 1 %). The slowdown due to data collection is small (from no slowdown up to 2 %). For the performance analysis all samples except the first 10 are used (these can be several order of magnitudes larger than the other samples). We only analyze the performance of one of the trees, since we cannot see any difference in the behavior of the two trees.

6 Reconfiguration Experiments and Cost Breakdown Results

In this section we analyze how reconfiguring communication structure can improve performance, and we use the fastest allreduce configuration to provide a cost breakdown.

6.1 8W Cluster

In the initial configuration for the 8W cluster, *nary8W*, a hierarchy aware n-ary tree is used (figure 8). The tree has one thread per CPU. A reduce is done on each 8W host, before sending the partial results to the cluster-root wrapper, located on one of the 8W hosts. The broadcast use the same wrappers.

Adding an additional level (figure 9) to the host subtrees gives a speedup of 1.17 (when comparing two

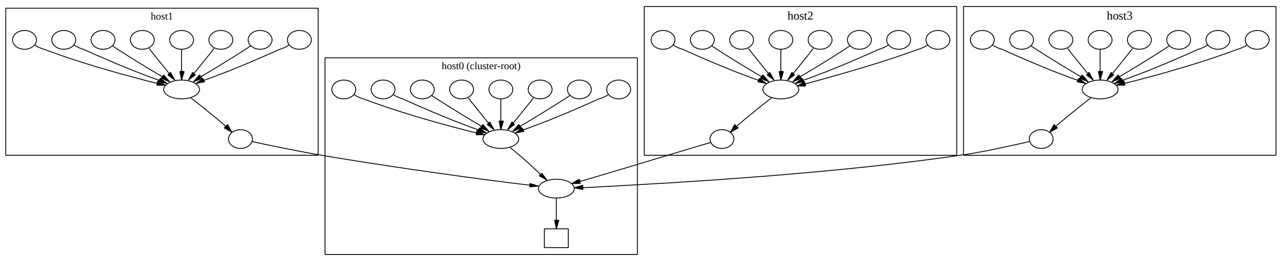

Figure 8. Initial 8W Gsum configuration (n-ary hierarchy aware tree).

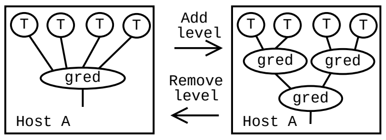

Figure 9. Add a level to a subtree to increase concurrency, or remove a level to reduce latency.

non-instrumented configurations). Adding another level to the host subtrees, gives yet another speedup of 1.17. Adding a level to the cluster-root gives a further speedup of 1.08, resulting in a total accumulated speedup of 1.49. Based on the tree shape, the configurations are called respectively, *quad8W*, *binaryquad8W* and *binary8W*. *Binaryquad8W* have binary host subtrees, and a quad cluster-root subtree.

Adding an additional level introduces a small increase in the down-latency for the entire subtree (up to $13\,\mu s$, including latencies introduced by the additional event collectors). However, as shown in figure 10, the up-latency ($x = 0$) decreases when additional levels are added to the host subtrees. The departure wait times are also decreased, resulting in improved concurrency during the broadcast (slope of the curve, flatter is better).

The host subtree on the cluster-root host has higher up-latency than the other host subtrees, due to the additional load introduced by the cluster-root subtree. Compared to *binaryquad8W*, the additional level added to the cluster-root subtree in *binary8W*, results in better performance for the host subtree (figure 10), even if the host subtrees have similar shape.

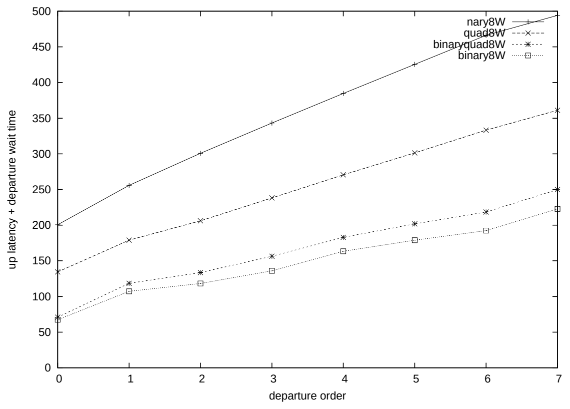

Figure 10. Up-latency and departure wait times for 8W cluster-root host subtree using various configurations.

The different reconfigurations did not change the amount of time spent in the proxy stages or the core stage.

The departure order of a gred wrapper is dependent on the arrival order (table 1). The last arrival almost always departs first. The first arrival mostly departs as the second, and the second arrival mostly departs as the third. However, for the third to seventh arrival there is some variation. This indicates that some form of FIFO queue is used by the synchronization variables that are used to implement the gred wrapper.

In earlier work [4], we improved the performance of a configuration similar to *nary8W* by moving the root wrapper in each host subtree to the cluster-root hosts (figure 11). However, adding additional network links did not improve (or decrease) the performance of the *binary8W* configuration.

	1st dep	2nd dep	3rd dep	4th dep	5th dep	6th dep	7th dep	8th dep
1st arrival	0.0%	99.9%	0.1%	0.0%	0.0%	0.0%	0.0%	0.0%
2nd arrival	0.0%	0.1%	99.8%	0.1%	0.0%	0.0%	0.0%	0.0%
3rd arrival	0.0%	0.0%	0.2%	96.5%	3.2%	0.1%	0.0%	0.0%
4th arrival	0.0%	0.0%	0.0%	3.3%	95.9%	0.7&	0.1%	0.0%
5th arrival	0.0%	0.0%	0.0%	0.0%	0.9%	95.0%	4.1%	0.0%
6th arrival	0.0%	0.0%	0.0%	0.0%	0.0%	4.2&	90.9%	5.0%
7th arrival	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	5.0%	95.0%
8th arrival	100.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%

Table 1. For a gred wrapper, how many times the first arrival departed as the first, second and so on.

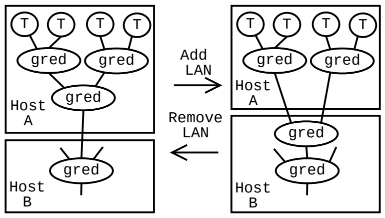

Figure 11. Add additional network link to reduce contention on shared resources and synchronization mechanisms, or remove network link to reduce number of messages sent over a network.

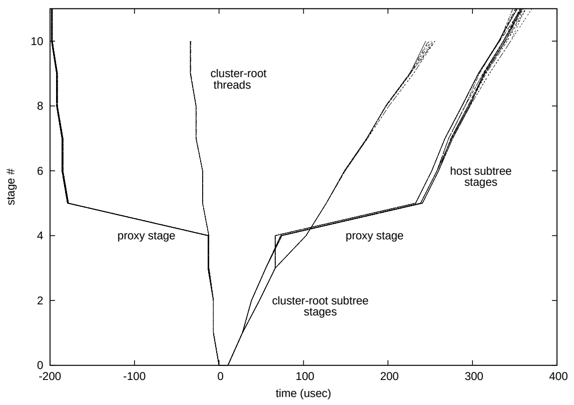

Figure 12. Timemap for all threads in the *binary8W* configuration.

6.1.1 Cost Breakdown

To find the overall behavior of the *binary8W* configuration we use a *timemap visualization*[4]. The timemap in figure 12 shows the mean time spent (x-axis) in each stage of the path (y-axis) when moving down and up the path. The arrival wait times are not shown. $X = 0$ is when the threads enter the core wrapper (the bottommost wrapper).

The tree has a regular shape, and the threads have similar behavior. The threads can be divided into two classes, those on the cluster-root host, and those not (these have an additional proxy wrapper). The variation in the cluster root up-path is due to the arrival-departure order dependency described earlier.

Statistics for the slowest thread in the *binary8W* are shown in table 2. The departure wait time is dependent on the departure order, hence the large standard deviation (the distribution is a combination of several distributions). The large standard deviation for the up, and down latencies for gred wrappers are probably caused by queuing in the synchronization variables that are used in the implementation. However, we believe the mean time per stage can be used to get an overview of the cost of each stage.

Table 3 summarizes the mean time spent in different parts of the allreduce tree for the fastest (0) and slowest thread (17). Both spend more time in the up-path (broadcast) than on the down path (reduce). More time is spent in the host subtree than in the cluster-root subtree, since there are more CPUs per host than hosts in the cluster. The fastest thread has a slower host

[4]The timemap is inspired by the lifeline visualizations used in the NetLogger Visualization [24].

	mean	median	stdev	min	max	10-perc	90-perc
Host level 0 down latency	6	5	18	2	692	4	7
Host level 0 up latency	25	25	7	14	152	16	34
Host level 0 departure wait	25	33	22	0	160	0	53
Cluster-root level 1 down latency	7	4	10	2	73	4	7
Cluster-root level 1 up latency	17	17	1	15	65	16	18
Cluster-root level 1 departure wait	11	0	15	0	80	0	31
Proxy wrapper latency	333	331	24	268	1367	308	357
Core wrapper latency	11	10	3	8	64	8	14
Time per allreduce	680	678	124	388	2509	537	807

Table 2. Time in μs spent in various stages in the *binary8W* configuration for thread 17.

Part	Fastest thread (0)	Slowest thread (17)
Down-path, with arrival wait	409	320
Down-path, no arrival wait	45	212
Up-path	233	360
Host subtree, no arrival wait	156	149
Cluster-root subtree, no arrival wait	104	77
Proxy stage	0	332
Core	11	11
Total, no arrival wait	278	572
Total time with arrival wait	642	680

Table 3. Mean time in μs spent in various parts of the *binary8W* configuration, for a thread on the cluster-root host (0), and a thread on another host (17).

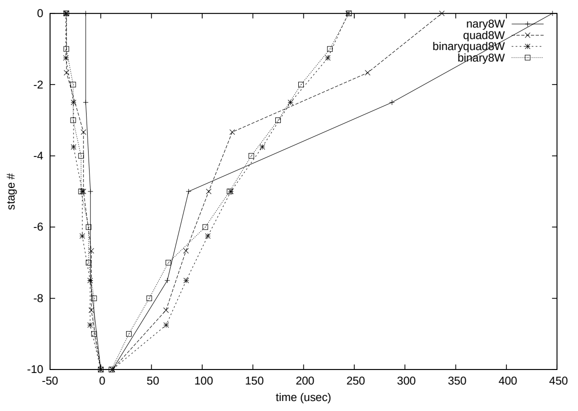

Figure 13. Timemap without arrival wait times, for thread 0 using various 8W configurations.

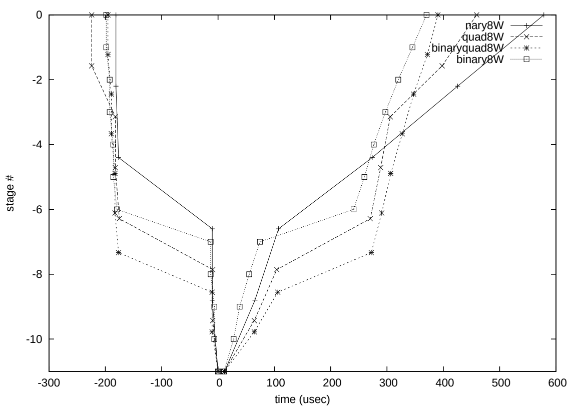

Figure 14. Timemap without arrival wait times, for thread 17 using various 8W configurations.

subtree due to the additional load introduced by the cluster-root subtree. It also spends more time in the cluster-root subtree due to the arrival-departure order dependency (it mostly arrives first, and hence mostly departs last). The slowest threads are slower due to the proxy stages, which dominates the time per collective operation. For both threads the time spent storing the tuple in the PastSet element (core) is insignificant.

Figures 13 and 14 shows the timemap for the fastest (0) and slowest thread (17). Since paths in two configurations can have unequal length, the y-coordinates are scaled such that both have the same y_0 and y_{max}. The largest improvements are on the up-path due to the improvements described above.

6.2 4W Cluster

The initial configuration, *nary4W*, has a hierarchy aware n-ary tree similar to the initial 8W configuration (figure 8). A binary tree configuration, *binary4W*, where a level is added to all host subtrees and two levels are added to the cluster-root subtree resulted in a speedup of 1.07. As for the 8W cluster, the performance is improved due to improvements in the up-latency and improved concurrency of gred wrappers.

Adding additional LAN links (figure 11), resulted in a slowdown of 1.49, due to increased load on the cluster-root host.

In the *binary4W* configuration the cluster root has three levels. The *split4W* configuration is created by

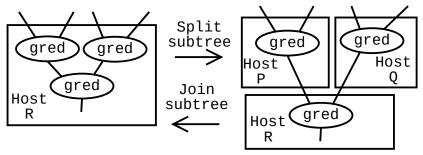

Figure 15. Increase parallelism and reduce load on root host by moving toplevel wrappers to other hosts, or move toplevel wrappers to root host to reduce number of messages sent over network.

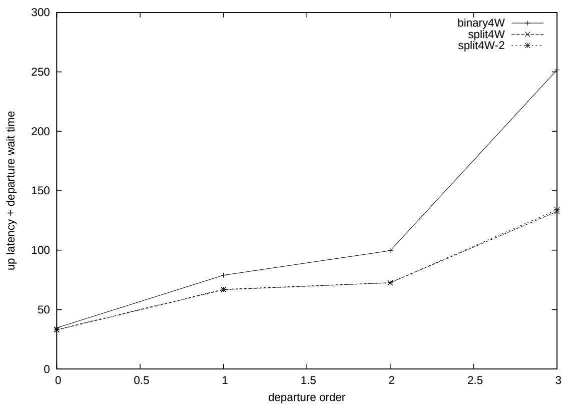

Figure 16. Up-latency and serialization introduced by the two bottom levels of the 4W cluster-root subtree.

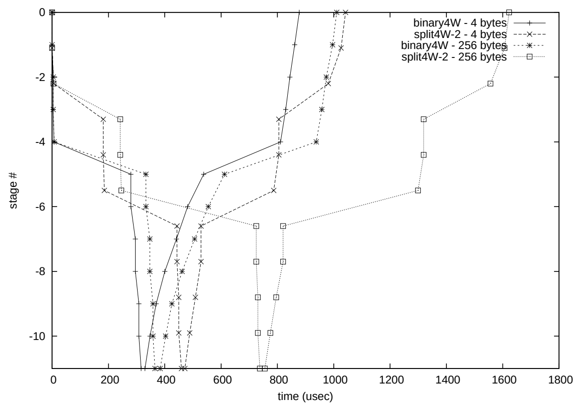

Figure 17. Timemap without arrival wait times, for thread 9 using various 4W configurations.

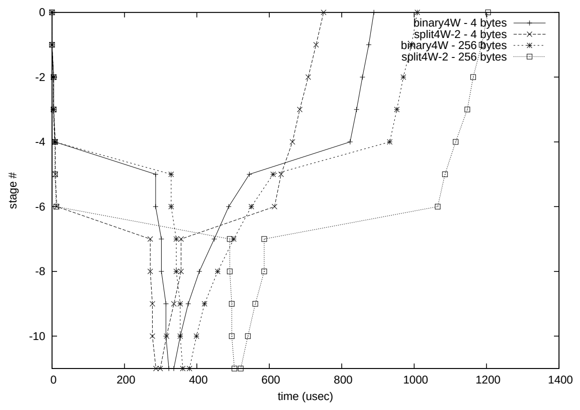

Figure 18. Timemap without arrival wait times, for thread 13 using various 4W configurations.

moving three of the four toplevel gred wrappers to other hosts (figure 15). The fourth is not moved since it has the cluster-root host subtree as one of the contributors. A speedup of 1.10 is achieved, due to performance improvements in the two bottom levels in the cluster-root subtree (figure 16), and in the toplevel gred wrappers moved to other hosts. In the new configuration, 24 threads get an additional proxy wrapper, but the time spent in the proxy wrappers is only increased by 1.55. The host subtree on the hosts where the gred wrappers where moved to also got worse performance. This shows how a trade-off between the number of network links, and the load on the cluster-root host can improve performance even for a small cluster with 8 hosts.

For the four host without any part of the cluster-root subtree, the initial *nary4W* configuration has better performance than *split4W*. A new configuration, *split4W-2*, where the host subtrees on these four hosts have similar shape and performance as in *nary4W*, gives a speedup of 1.05.

The 12 slowest threads in *split4W-2* are slower than the 12 slowest threads in *binary4W*, as shown by the '4 bytes' graphs in figure 17. However, the threads with one proxy wrapper are faster (figure 18). Overall, the *split4W-2* configuration is faster since the variation in time per stage leads to the slowest thread on the average not being slowest all the time. Thus it is difficult to determine which configuration is fastest by only considering the performance of the slowest thread.

6.2.1 Cost Breakdown

Figure 19 shows the timemap for all threads in the *split4W-2* configuration. The irregular shape of the tree can be seen in the figure. As for the 8W configuration, most of the time is spent in the up-path for all threads. However, more time is spent in the cluster-root subtree since there are more hosts than CPUs per host. As for the 8W configuration the proxy stages dominate the time per allreduce. The fastest thread use 339 μs on the average per allreduce (without arrival wait times), while the slowest use 1042 μs (the difference being the proxy stages). The difference between the fastest and slowest thread with one and two proxy wrappers is respectively 70 μs and 147 μs. The difference for both, is the time spent in the proxy stages to and from the cluster-root host stages.

6.3 NOW

In the initial NOW configuration a quad tree is used. Splitting the cluster-root subtree, by moving seven of the eight toplevel wrappers to other hosts (the cluster-root host subtree contributes to the eight), gives a speedup of 2.44. Using a binary tree gives a speedup of 1.37. Splitting the cluster-root subtree further, gives a slowdown of 1.11.

An overview of the behavior of the binary configuration is shown in figure 20. The tree has an irregular shape, and the threads also have irregular behavior.

Part	Zero (0)	One (56)	One (4)	Two (11)	Two (36)
Down-path, with arrival wait	674	628	575	534	506
Down-path, no arrival wait	27	204	199	351	369
Up-path	368	415	483	520	566
Host subtree, no arrival wait	21	11	15	13	20
Cluster-root subtree, no arrival wait	368	232	319	196	217
Proxy stages	0	370	342	656	694
Core	6	6	6	6	6
Total, no arrival wait	395	619	682	871	935
Total, with arrival wait	1042	1043	1058	1054	1072

Table 4. Average time in μs spent in each stage of the binary configuration for threads with zero, one, and two proxy wrappers on their path. The slowest and fastest threads with one and two proxies are shown.

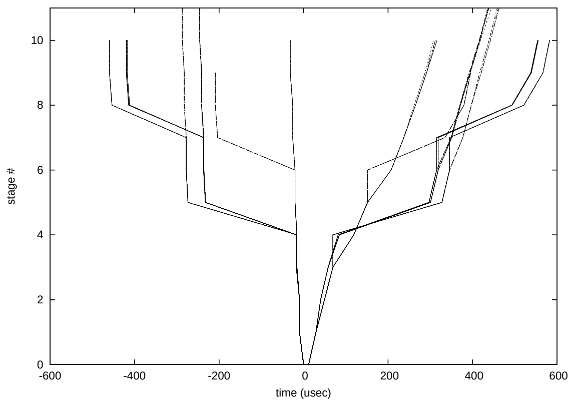

Figure 19. Timemap without arrival wait time for all threads in the *split4W-2* configuration.

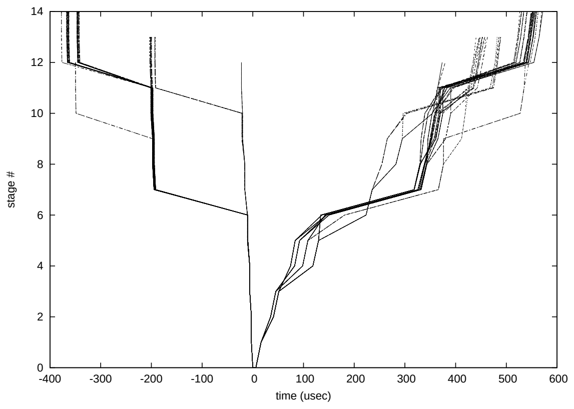

Figure 20. Timemap without arrival wait times, for all threads in the binary NOW configuration.

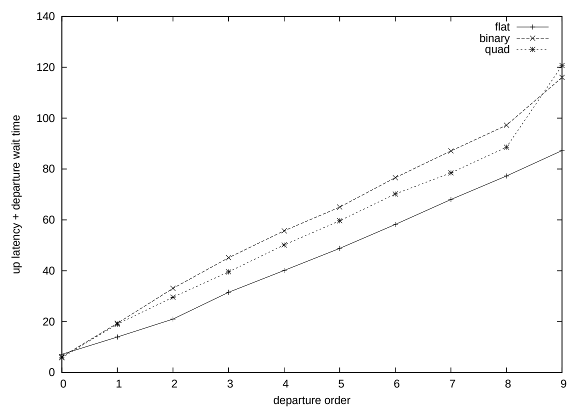

Figure 21. Up latency + departure wait time for Blade cluster-root subtree.

The fastest threads are the two without proxy wrappers. All threads with one proxy are faster than threads with two proxies. The slowest thread with one proxy is 61 μs faster than the fastest thread with two proxy wrappers.

Table 4 shows the amount of time spent in various parts of the binary tree for the fastest thread, and the slowest and fastest threads with one and two proxies. The up-path is slower than the down-path. Most of the time is spent in the proxy stages, and more time is spent in the cluster-root subtree than in the host subtree. The fastest threads spend more time in the cluster-root subtree due to the arrival-departure order dependency (it is also visible in the up-path in figure 20).

6.4 Blade Cluster

For the Blade cluster the best configuration has one gred wrapper which all ten threads contribute to. Adding additional levels does not improve performance, since the host has only one CPU and hence operations cannot be done in parallel (figure 21).

The Blade cluster-root subtree has more contributors than the 4W cluster-subtree. However, splitting the Blade cluster-root subtree among four hosts does not improve performance (as it does for the 4W subtree). This is probably caused by the faster CPU on the Blade hosts.

As for the other configuration less time is spent on the down path, and most of the time is spent in the proxy stages.

7 Sensibility Analysis Experiments

In this section we examine if a configuration can be extended by adding more threads, or by combining two configurations. We also examine how an increase in message size influence the performance.

7.1 Increasing Number of Threads on 8W Cluster

To evaluate how the increase in number of threads influence the performance of a binary tree (*binary8W*), and quad tree configuration (*quad8W*), we use the cluster with largest SMPs (8W). Table 5 shows that the binary tree is always faster, since the height of the tree increases logarithmically and the gred wrappers have scalability problems. The increase in threads, did not decrease the performance of the cluster-root subtree.

7.2 Combining 4W and 8W Clusters

To examine if two single cluster configurations can be combined in a multi-cluster configuration without decreasing performance, we combine the *split4W* and *binary8W* configurations. A root wrapper is located on the 4W cluster-root host. The cluster-root subtrees contribute to this wrapper. Communication between the 4W and 8W cluster must go through a gateway host.

The performance of the 8W tree, and the performance of all 4W and 8W host subtrees has not changed. The 4W tree is around 300 μs faster in the multi-cluster configuration than in the single cluster configuration, due to reduced load caused by the 4W threads having to wait for the slower 8W threads (the 24 slowest threads are on the 8W cluster).

For the slowest thread, the proxy stages contribute on the average with 930 of the 1193 μs per allreduce. 598 μs are used for overlaying through the gateway host. The slowest 4W threads are only around 150 μs faster than the slowest 8W threads.

A configuration with one of the 8W hosts chosen as root resulted in a slowdown of 1.46. The reason for this is that the slowest 4W threads are even slower than the slowest 8W threads in the initial configuration.

Threads	Levels		Execution time		Speedup
	binary	quad	binary	quad	
32	3	6	40.5 sec	32.0 sec	1.26
64	3	7	60.9 sec	45.8 sec	1.33
128	4	8	107.4 sec	78.0 sec	1.38

Table 5. Comparing two configurations when increasing the number of threads per CPU. Gsum is run for 25 000 iterations.

7.3 Increasing Message Size

To examine how a small increase in message size affects the performance of different configurations, we use the cluster with slowest CPUs and the largest user-to-user level communication latency (4W). The tuple size is increased from 4 bytes to 256 bytes (64×4 byte integers).

The additional 'reduce' work is not shown in the down-latency of the gred wrappers since the down-latency is dominated by synchronization time. The only wrappers affected by the increase are the proxy wrappers (for the NOW cluster, a similar increase in tuple size also only affects the proxy wrappers).

Figures 17 and 18 shows the timemap for two threads using 4 and 256 byte tuples. The *binary4W* configuration is 1.23 faster than the *split4W-2* (it is 1.12 slower with 4 byte tuples), since splitting the cluster root increases the time spent in the proxy stages more than it improves the cluster-root subtree performance.

8 Conclusion and Future Work

An allreduce collective operation can be organized as a tree, that describes which threads communicate with which other threads, and where data should be reduced and broadcasted. We have documented how reconfiguring the shape and mapping of the collective operation tree can improve performance. Also a system and approach for monitoring, analyzing and visualizing the performance of such a tree is described. We analyze the communication behavior of threads using a given configuration and compare the communication behavior of a thread using different configurations.

An allreduce micro-benchmark was run on a blade cluster with ten uni-processor blades, a cluster of thirty 2-way hosts, a cluster of eight 4-way hosts, and a cluster of four 8-way hosts.

Even if all our initial configurations were hierarchy aware, reconfiguration improved performance by 1.18, 1.37, 1.20, 1.49 for the blade, 2-way, 4-way and 8-way cluster respectively.

For the 8-way and 2-way SMPs, a binary tree has best performance. However for the 4-way SMP a quad tree may have better performance depending on the load on the host. For single-CPU hosts a flat tree has best performance.

We improved performance by finding the right balance between available concurrency, load on root hosts, and the number of network links in a tree. However, finding the right balance can be difficult since the user-to-user-level network latency is dependent on factors such as the CPU speed, and load on hosts.

For the 2-way and 4-way clusters the best configuration has an irregular shape, complicating the analysis due to rather large variations in the time spent in various parts of the tree.

Our results shows that performance is not negatively effected when two cluster configurations are combined in a multi-cluster configuration, or when the number of threads is increased. But a reconfiguration may be necessary even for a small increase in message size (from 4 to 256 bytes).

A breakdown of the cost of an allreduce shows that for all clusters more time is spent on the broadcast part than the reduce. Also more time is spent on the network than in the partial allreduce components, and the time to do the reduce is not visible.

As future work we intend to study the performance of the allreduce operation using more complex benchmarks, for example to evaluate how reconfiguration can be used to improve the performance of applications with load balance problems. Also, other non-synchronizing, collective operations such as reduce should be analyzed. Finally, we continue working towards our long-term goal, a communication system where collective communication is analyzed, and adapted at run-time.

Acknowledgments

Thanks to Jon Ivar Kristiansen and Torfinn Holand for assisting us with the NOW experiments, Brian Vinter for discussions, and Espen Johnsen for reading the paper and suggesting improvements.

References

[1] BERNASCHI, M., AND IANNELLO, G. Collective communication operations: Experimental results vs.theory. *Concurrency: Practice and Experience 10*, 5 (1998), 359–386.

[2] 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).

[3] 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).

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

[5] BONGO, L. A. EventScope: Configurable Online Monitoring of Parallel and Distributed Applications. Master's thesis, Department of Computer Science, University of Tromsø, 2002.

[6] BONGO, L. A., ANSHUS, O., AND BJØRNDALEN, J. M. EventSpace - Exposing and observing communication behavior of parallel cluster applications. In *Euro-Par* (2003), vol. 2790 of *Lecture Notes in Computer Science*, Springer.

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

[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] JONES, T., TUEL, W., BRENNER, L., FIER, J., CAFFREY, P., DAWSON, S., NEELY, R., BLACKMORE, R., MASKELL, B., TOMLINSON, P., AND ROBERTS, M. Improving the scalability of parallel jobs by adding parallel awareness to the operating system. In *Proceedings of the 2003 ACM/IEEE conference on Supercomputing* (November 2003), ACM/IEEE.

[10] KARP, R. M., SAHAY, A., SANTOS, E. E., AND SCHAUSER, K. E. Optimal broadcast and summation in the LogP model. In *ACM Symposium on Parallel Algorithms and Architectures* (1993), pp. 142–153.

[11] KARWANDE, A., YUAN, X., AND LOWENTHAL, D. K. CC-MPI: a compiled communication capable MPI prototype for Ethernet switched clusters. In *Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming* (2003), ACM Press, pp. 95–106.

[12] 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.

[13] KUMAR, S., JIANG, D., CHANDRA, R., AND SINGH, J. P. Evaluating synchronization on

shared address space multiprocessors: methodology and performance. In *Proceedings of the 1999 ACM SIGMETRICS international conference on Measurement and modeling of computer systems* (1999), ACM Press, pp. 23–34.

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

[15] LOWEKAMP, B., AND BEGUELIN, A. ECO: Efficient collective operations for communication on heterogeneous networks. In *International Parallel Processing Symposium* (1996), pp. 399–405.

[16] MELLOR-CRUMMEY, J. M., AND SCOTT, M. L. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems (TOCS) 9*, 1 (1991), 21–65.

[17] MOORE, S., D.CRONK, LONDON, K., AND J.DONGARRA. Review of performance analysis tools for MPI parallel programs. In *8th European PVM/MPI Users' Group Meeting, Lecture Notes in Computer Science 2131* (2001), Springer Verlag.

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

[19] PETRINI, F., KERBYSON, D. J., AND PAKIN, S. The case of the missing supercomputer performance: Achieving optimal performance on the 8,192 processors of ASCI Q. In *Proceedings of the 2003 ACM/IEEE conference on Supercomputing* (November 2003), ACM/IEEE.

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

[21] 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.

[22] SUNDERAM, V. S. PVM: a framework for parallel distributed computing. *Concurrency: Practice and Experience 2*, 4 (1990), 315–339.

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

[24] TIERNEY, B., JOHNSTON, W. E., CROWLEY, B., HOO, G., BROOKS, C., AND GUNTER, D. The NetLogger methodology for high performance distributed systems performance analysis. In *Proc. 7th IEEE Symp. On High Performance Distributed Computing* (1998), pp. 260–267.

[25] TIPPARAJU, V., NIEPLOCHA, J., AND PANDA, D. Fast collective operations using shared and remote memory access protocols on clusters. In *17th Intl. Parallel and Distributed Processing Symp.* (May 2003).

[26] TRÄFF, J. L. Implementing the MPI process topology mechanism. In *Proceedings of the 2002 ACM/IEEE conference on Supercomputing* (2002), IEEE Computer Society Press, pp. 1–14.

[27] VADHIYAR, S. S., FAGG, G. E., AND DONGARRA, J. Automatically tuned collective communications. In *Proceedings of the 2000 ACM/IEEE conference on Supercomputing* (2000).

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

[29] VETTER, J. S., AND YOO, A. An empirical performance evaluation of scalable scientific applications. In *Proceedings of the 2002 ACM/IEEE conference on Supercomputing* (November 2002), ACM/IEEE.

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