Collective Communication Performance Analysis Within the Communication System

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

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

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

1 Introduction

Clusters are becoming an increasingly important platform for scientific computing. Many parallel applications run on clusters use a communication library, such as MPI [9], which provides collective operations to simplify the development of parallel applications. Of the eight scalable scientific applications investigated in [16], most would benefit from improvements to MPI's collective operations.

The communication structure of a collective operation can be organized as a spanning tree, with threads as leafs. Communication proceeds along the arcs of the tree and a partial operation is done in each non-leaf node. Essential for the performance of a collective operation is the shape of the tree, and the mapping of the tree to the clusters in use [7, 12–14].

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

Performance monitoring tools for MPI programs [8] generally treat the communication system as a black box and collect data at the MPI profiling layer (a layer between the application and the communication system). To understand why a specific tree and mapping have better performance than others it is necessary to collect data for analysis inside the communication system. We describe our experiences about what type of data is needed, how to do the analysis, and what the challenges are for collective communication performance analysis.

M. Danelutto, D. Laforenza, M. Vanneschi (Eds.): Euro-Par 2004, LNCS 3149, pp. 163–172, 2004.

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

For a given tree configuration and mapping, our analysis approach and visualizations allows us to find performance problems within the communication system, and to compare the performance of several configurations. This allows us to do a more fine grained optimization of the configuration than approaches that only use the time per collective operation (as in [14]).

In addition to remapping trees, collective operation performance can be improved by taking advantage of architecture specific optimizations [12, 13], or by using a lower-level network protocol [6, 13]. However, the advantage of these optimizations depends on the message size. For example, for small message sizes, as used by most collective operations, point-to-point based communication was faster than Ethernet based broadcast in [6], and a shared memory buffer implementation for SMPs in [12]. Our approach allows comparing advantages of changing the communication protocol, or synchronization primitives for different message sizes.

On SMPs, collective communication performance can also significantly be reduced, by interference caused by system daemons [10]. Reducing the interference will make the performance analysis within the communication system even more important.

Mathematical models can be used to analyze the performance of different spanning trees (as in [1]), but these do not take into account the overlap and variation in the communication that occurs in collective operations [14].

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

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

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

2 Reconfigurable Collective Operations

We use the PATHS system [2] to experiment with different collective operation spanning tree configurations and mappings of the tree to the clusters in use. In

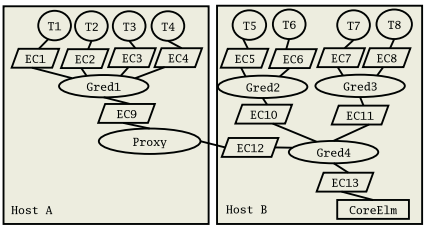


Fig. 1. An allreduce tree used by threads T1–T8 instrumented with event collectors (EC1–EC13). The result is stored in a PastSet element (CoreElm).

allreduce, each thread has data that is reduced using an associative operation, followed by a broadcast of the reduced value. The semantics differs from MPI in that the reduced value is stored in the PastSet structured shared memory [17].

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

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

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

3 Monitoring and Analysis

To collect performance data we use the EventSpace system [3]. The paths in a collective operation tree are instrumented by inserting *event collectors*, implemented as PATHS wrappers, before and after each wrapper. In figure 1, an allreduce tree used by threads T1–T8 is instrumented, by event collectors EC1–EC13. For each allreduce operation, each event collector records a timestamp

when moving down, and up the path. The timestamps are stored in memory and written to trace files when the paths are released. In this paper analysis is done post-mortem.

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

There are three types of wrappers in an allreduce spanning tree: gred (partial allreduce), proxy (network) and core (PastSet). For core wrappers the two timestamps collected by the event collector above it (EC13 in figure 1) are used to calculate the *store latency*; the time to store the result in PastSet. For proxy, we calculate the two-way TCP/IP latency by $(t_4 - t_1) - (t_3 - t_2)$, where t_1 (down) and t_4 (up) are collected by the event collector above the proxy in a path, and t_2 (down) and t_3 (up) are collected by the event collector below. To achieve the needed clock synchronization accuracy for calculating one-way latencies (tens of μs) special hardware is needed [11].

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

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

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

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

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

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

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

4 Experiments

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

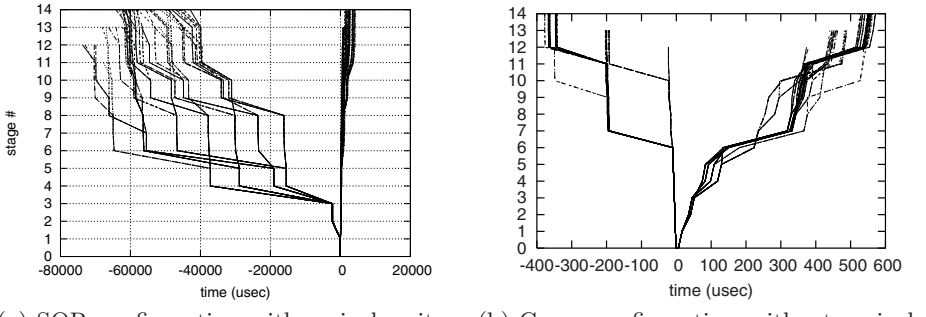
We use a microbenchmark, Gsum, which measures the time it takes T threads to do N allreduce operations. The allreduce computes a global sum. The number of values to sum is equal to the number of threads, T. Threads alternate between using two identical allreduce trees to avoid two allreduce calls to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

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

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

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

The large standard deviation for gred up, and down latency is caused by queuing in the synchronization variables, while the departure wait time distribution is a combination of several distributions since the wait time depends on the departure order (which, in our implementation, depends on the arrival order). We expect most implementation to have similar large variations for these stages. For Gsum, arrival wait time variation is caused by variations on the up-path, while for SOR and other applications there are additional noise sources such as system daemons [10].

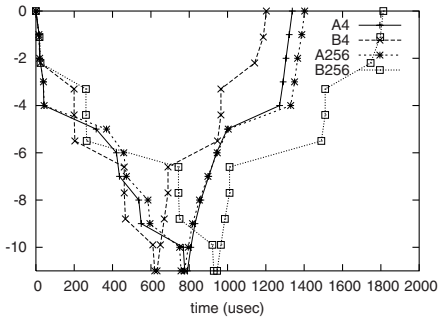


(a) SOR configuration with arrival wait times.

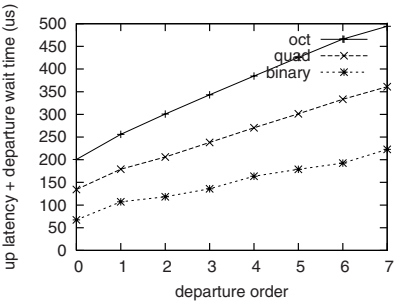
(b) Gsum configuration without arrival wait times.

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

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



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



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

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

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

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

For the mostly used allreduce message sizes (below 256 bytes [15]), a cost breakdown shows that broadcast is more expensive than reduce. The up, and down gred-latency are only a few μs, hence the time to do the reduce operation is insignificant. For SMPs the departure wait time can be large, but for the best configuration the TCP/IP stages dominate the execution time. Also, the time spent storing the result in PastSet is insignificant. For some single-CPU cluster configurations we have a few outliers in the proxy stage that can significantly reduce performance. For one Blade Gsum configuration they caused a slowdown of 54.2.

When tuning the performance of a configuration it is important to find the right balance between load on *root* hosts and number of network links. Load on root hosts can be reduced by moving gred wrappers to other hosts. This will also improve potential parallelism, but it will introduce another TCP/IP connection

to some threads paths. For SMPs, the performance of a host subtree can be improved by adding or removing an additional level in the spanning tree.

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

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

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

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

5 Discussion

The timestamps collected by EventSpace allows us to analyze the performance of spanning trees and their mapping to the clusters in use. However, information from within the operating system is needed to understand why a synchronization operation, or a TCP/IP connection is slow. Hence, information from within the operating system should be collected and used in the analysis.

For ease of prototyping we used our own parallel programming system (PATHS). Our analysis approach should be applicable for MPI runtime systems provided that we can collect timestamps that can be correlated to a given MPI collective operation call. Once monitoring tools using the proposed MPI PERUSE interface[5] are available these may be used to collect the data necessary for our analysis. However, other runtime systems will have performance problems not treated by this work, for example with regards to buffering.

The calculated values used in the analysis have large variation, and outliers can have a significant effect on performance. A complete trace of all messages sent within a given time period provides enough samples for statistical analysis, and can be used to detect any periodical performances faults (e.g. caused by system daemons [10]). EventSpace allows to collect such traces with a small overhead and memory usage (1 MB of memory can store 29.127 EventSpace events).

We only studied one application; SOR. Since MPI defines the semantics of collective operations the communication pattern of SOR is general and frequent. Since SOR has a load imbalance a better application for the study would have been one of the applications described in [10]. Also, only the allreduce operation has been analyzed. We believe the analysis will be similar for other collective operations with small message size such as reduce and barrier. Message arrival order, synchronization points, and network latencies are also important for the performance of operations with larger messages, such as alltoall, allgather, and the MPI-IO collective operations. For MPI-IO we can wrap the I/O operations using PATHS wrappers.

6 Conclusion and Future Work

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

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

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

References

1. Bernaschi, M., and Iannello, G. Collective communication operations: Experimental results vs.theory. *Concurrency: Practice and Experience 10*, 5 (1998).
2. Bjørndalen, J. M. *Improving the Speedup of Parallel and Distributed Applications on Clusters and Multi-Clusters*. PhD thesis, Tromsø University, 2003.
3. Bongo, L. A., Anshus, O., and Bjørndalen, J. M. EventSpace - Exposing and observing communication behavior of parallel cluster applications. In *Euro-Par* (2003), vol. 2790 of *Lecture Notes in Computer Science*, Springer, pp. 47–56.

4. Bongo, L. A., Anshus, O., and Bjørndalen, J. M. Evaluating the performance of the allreduce collective operation on clusters: Approach and results, 2004. Technical Report 2004-48. Dep.of Computer Science, University of Tromsø.
5. Jones, T. Personal communication. 2003.
6. Karwande, A., Yuan, X., and Lowenthal, D. K. CC-MPI: a compiled communication capable MPI prototype for Ethernet switched clusters. In *Proc. of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming* (2003), ACM Press, pp. 95–106.
7. Kielmann, T., Hofman, R. F. H., Bal, H. E., Plaat, A., and Bhoedjang, R. A. F. Magpie: Mpi's collective communication operations for clustered wide area systems. In *Proceedings of the seventh ACM SIGPLAN symposium on Principles and practice of parallel programming* (1999), ACM Press, pp. 131–140.
8. 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.
9. MPI: A Message-Passing Interface Standard. *Message Passing Interface Forum* (Mar. 1994).
10. Petrini, F., Kerbyson, D. J., and Pakin, S. The case of the missing supercomputer performance: Achieving optimal performance on the 8,192 processors of ASCI Q. In *Proc. of the 2003 ACM/IEEE conference on Supercomputing* (2003).
11. Pasztor, 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.
12. Sistare, S., vandeVaart, R., and Loh, E. Optimization of mpi collectives on clusters of large-scale smp's. In *Proceedings of the 1999 ACM/IEEE conference on Supercomputing* (1999), ACM Press.
13. 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).
14. Vadhiyar, S. S., Fagg, G. E., and Dongarra, J. Automatically tuned collective communications. In *Proceedings of the 2000 ACM/IEEE conference on Supercomputing* (2000).
15. 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).
16. Vetter, J. S., and Yoo, A. An empirical performance evaluation of scalable scientific applications. In *Proceedings of the 2002 ACM/IEEE conference on Supercomputing* (2002), IEEE Computer Society Press.
17. Vinter, B. *PastSet a Structured Distributed Shared Memory System.* PhD thesis, Tromsø University, 1999.