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

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

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

I. Introduction

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

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

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

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

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

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

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

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

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

II. Related Work

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

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

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

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

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

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

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

III. Systems

A. PATHS

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

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

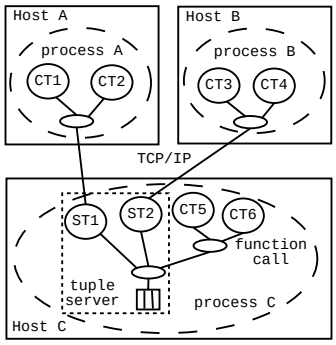

Fig. 1. An application with six computational threads (CT) and two TCP/IP service threads (ST) using a collective operation tree implemented using allreduce wrappers (small ovals). Results are stored in a PastSet element.

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

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

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

B. EventSpace

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

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

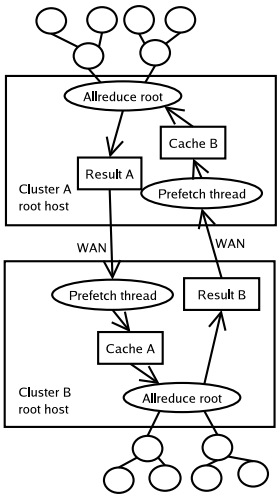

Fig. 2. Conditional-allreduce implementation for two clusters.

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

IV. Conditional-allreduce

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

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

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

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

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

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

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

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

V. Methdology

A. Noise-allreduce Microbenchmark

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

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

B. Input Data

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

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

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

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

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

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

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

C. Clusters

The hardware platform comprises six clusters:

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

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

[2]The surface temperature of a blue star.

[3]Zero Kelvin, or absolute zero.

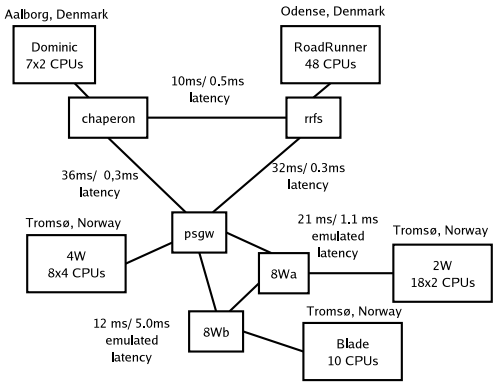

Fig. 3. Clusters, gateway hosts and WAN link emulator hosts of the multi-cluster used in the experiments. For each WAN link the average and standard deviation of the two-way TCP/IP latency is given.

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

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

D. Wide-area Network Emulator

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

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

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

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

VI. Experiments

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

A. Sequential Allreduce

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

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

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

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

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

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

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

B. MagPIe Allreduce

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

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

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

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

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

[5]MagPIe is implemented on top of MPICH.

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

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

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

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

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

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

C. Conditional-allreduce

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

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

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

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

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

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

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

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

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

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

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

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

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

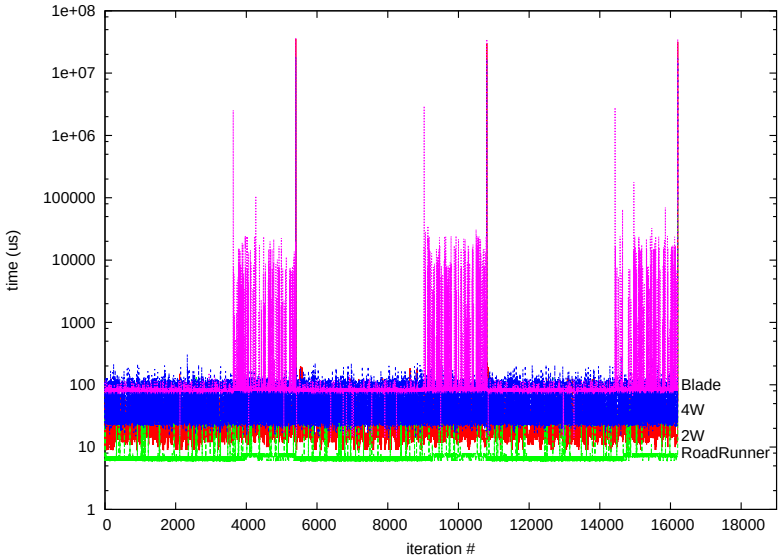

Fig. 4. For each cluster root, the time to determine whether the condition is true using the *frosty* data set. For clarity the graph for Dominic is not shown (it is similar to the 4W graph).

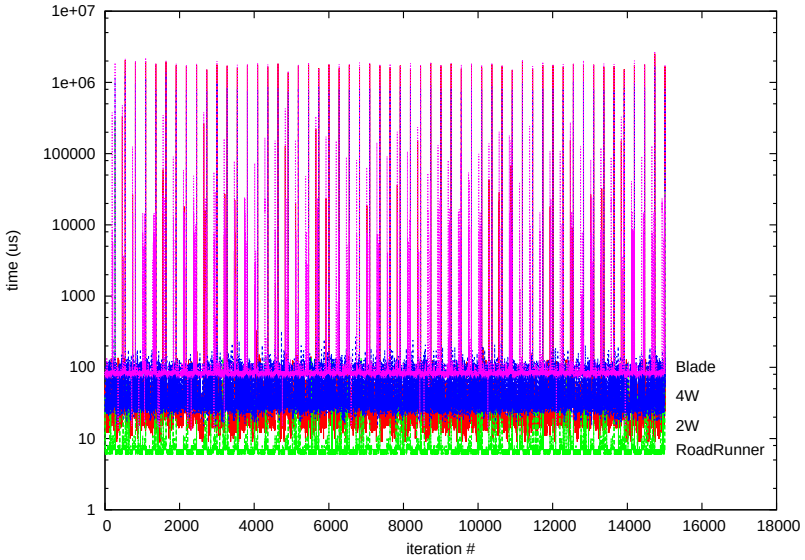

Fig. 5. For each cluster root, the time to determine whether the condition is true using the *random* data set. For clarity the graph for Dominic is not shown.

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

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

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

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

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

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

VII. Conclusion and Future Work

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

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

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

We proposed hiding some of the WAN latency in system noise, which delays the arrival of threads at synchronizing collective operations. However, our results demonstrate that the system noise in our multi-cluster is too low to allow a significant part of the WAN latency to be hidden.

For our setup, a wide area optimal allreduce algorithm did not perform significantly better than a sequential allreduce spanning tree. This is due to the multi-cluster topology, workload tuning problems on one cluster, and competition for resources between the communication system and the workload on another cluster.

Using conditional-allreduce, WAN latency was avoided for most operations since these require values from only one cluster. For the remaining operations, only values from a few clusters were needed, and these where often pre-fetched. There was no difference in performance when using a data set from an iterative converging algorithm that converged after 5403 iterations, or a data set from another algorithm which converges after 273 iterations. Conditional-allreduce only introduced overhead on the cluster with the slowest hosts.

Applications using conditional-allreduce can be run on a grid without performance degradation, provided that the point-to-point and other collective operations can tolerate the WAN latency and bandwidth problems. For many applications asynchronous point-to-point communication can be used [2]. We will as future work evaluate algorithms and communication systems for Grids using other types of collective operations with larger messages, such as all-to-all. We believe pre-fetching and replication may improve the performance of these operations. An open question is whether and how the semantics of these operations can be relaxed, or if other programming models may be required for applications using these operations.

ACKNOWLEDGMENT

Thanks to Brian Vinter for providing us access to the clusters in Denmark, and helpful discussions about the experiments. Also thanks to Josva Kleist and Gerd Behrmann for allowing us to use the cluster in Aalborg.

REFERENCES

[1] I. Foster and C. Kesselman, Eds., *The Grid 2: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 2003.

[2] A. Plaat, H. E. Bal, R. F. Hofman, and T. Kielmann, "Sensitivity of parallel applications to large differences in bandwidth and latency in two-layer interconnects," *Future Generation Computer Systems*, vol. 17, no. 6, pp. 769–782, 2001.

[3] "MPI: A Message-Passing Interface Standard," *Message Passing Interface Forum*, Mar. 1994.

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

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

[6] T. Jones, W. Tuel, L. Brenner, J. Fier, P. Caffrey, S. Dawson, R. Neely, R. Blackmore, B. Maskell, P. Tomlinson, and M. Roberts, "Improving the scalability of parallel jobs by adding parallel awareness to the operating system," in *Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, 2003.

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

[8] S. Sistare, R. vandeVaart, and E. Loh, "Optimization of mpi collectives on clusters of large-scale smp's," in *Proceedings of the 1999 ACM/IEEE conference on Supercomputing*. ACM Press, 1999.

[9] H. Tang and T. Yang, "Optimizing threaded mpi execution on smp clusters," in *Proceedings of the 15th international conference on Supercomputing*, 2001.

[10] P. Husbands and J. C. Hoe, "Mpi-start: delivering network performance to numerical applications," in *Proceedings of the 1998 ACM/IEEE conference on Supercomputing*. IEEE Computer Society, 1998, pp. 1–15.

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

[12] J. Liu, B. Chandrasekaran, J. Wu, W. Jiang, S. Kini, W. Yu, D. Buntinas, P. Wyckoff, and D. K. Panda, "Performance comparison of MPI implementations over InfiniBand, Myrinet and Quadrics," in *Proc. of the 2003 ACM/IEEE conference on Supercomputing*, 2003.

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

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

[15] S. Kumar, D. Jiang, R. Chandra, and J. P. Singh, "Evaluating synchronization on shared address space multiprocessors: methodology and performance," in *Proceedings of the 1999 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*. ACM Press, 1999, pp. 23–34.

[16] S. V. Adve and K. Gharachorloo, "Shared memory consistency models: A tutorial," *IEEE Computer*, vol. 29, no. 12, pp. 66–76, 1996.

[17] R. V. Renesse, K. P. Birman, and W. Vogels, "Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining," *ACM Transactions on Computer Systems (TOCS)*, vol. 21, no. 2, pp. 164–206, 2003.

[18] J. M. Bjørndalen, "Improving the speedup of parallel and distributed applications on clusters and multi-clusters," Ph.D. dissertation, Tromsø University, 2003.

[19] B. Vinter, "PastSet a Structured Distributed Shared Memory System," Ph.D. dissertation, Tromsø University, 1999.

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

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

[22] T. Kielmann, H. E. Bal, J. Maassen, R. van Nieuwpoort, L. Eyraud, R. Hofman, and K. Verstoep, "Programming environments for high-performance grid computing: the albatross project," *Future Generation Computer Systems*, vol. 18, no. 8, pp. 1113–1125, 2002.

[23] W. Gropp, E. Lusk, N. Doss, and A. Skjellum, "A high-performance, portable implementation of the MPI message passing interface standard," *Parallel Computing*, vol. 22, no. 6, pp. 789–828, September 1996.

[24] L. A. Bongo, O. Anshus, and J. M. Bjørndalen, "Collective communication performance analysis within the communication system," 2004, to appear in Proceedings of Euro-Par 2004.