Low Overhead High Performance Runtime Monitoring of Collective Communication

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

Scalability of parallel applications run on clusters and multi-clusters is often limited by communication performance. Message tracing can provide data for understanding bottlenecks, and for performance tuning. However, it requires collecting, storing, analyzing, and transferring potentially gigabytes of data. We have designed the EventSpace system for low overhead and high performance runtime collective communication trace analysis. EventSpace separates the perturbation and performance requirements of data collection, analysis, gathering and visualization. Data collection overhead is low since the minimum amount of data is recorded and stored temporarily in main memory. The recorded data is either discarded or analyzed on demand using available cluster resources. Analysis is distributed for high performance, and coscheduled with the computation and communication system threads for low perturbation. Gathering of analyzed data is done using extensible collective communication operations, which can be tuned to trade off between performance and monitoring overhead. EventSpace was used to do run-time monitoring and analysis of collective communication micro-benchmarks run on clusters, multi-clusters, and multi-clusters with emulated WAN links. Performance data was collected, analyzed and gathered with 0–3% monitoring overhead.

1 Introduction

In Grids rapid changes will be the norm. Hence, it is necessary for applications and the underlying systems to adapt, at run-time, to changes in the availability and performance of resources. An important part of the adaptation will be to reconfigure the point-to-point and collective communication structures used by parallel applications.

On large clusters, a much less dynamic environment than a Grid, communication system performance is important. Of eight scalable scientific application studied in [30], most would benefit from improvements to collective operations, and four would benefit from improvements in point-to-point communication performance. Improved communication performance is essential if Grids are to be used as a high performance computing platform.

Collective operation performance has been shown to improve by using better mappings of computation and data to the clusters in use [16, 24, 26, 27]. In earlier work, we have shown how to tune the mapping based on a performance analysis within the communication system [9]. We found that a global view of the system was needed to detect hotspots and simplify the hotspot analysis. Also, traces of all messages sent in a collective operation spanning tree were needed to understand some performance problems (as the problems described in [21]). Thus, we need to collect, store, analyze, gather, and visualize a large amount of performance data.

Monitoring tools need to collect data with minimal perturbation of the monitored application. For runtime analysis the performance data must be analyzed and often gathered to a single front-end host for use before the data becomes irrelevant. We have built the EventSpace system [8] for low overhead and high performance runtime collective communication trace analysis.

EventSpace is evaluated on clusters, multi-clusters, and multi-cluster with emulated WAN links. We demonstrate how data gathering performance can be tuned to either provide high performance or low perturbation. Our results show that performance data can be collected with less than 1% overhead. The data can be analyzed and gathered with 0–3% overhead, since collective communication intensive applications have low CPU utilization, and since analysis threads can be coscheduled with application and communication system threads.

2 Related Work

Generally performance monitoring tools for MPI programs [19] treats the communication system as a black box

and collect data at a layer between the application and the communication system (the MPI profiling layer). To understand why a specific collective operation spanning tree and mapping have better performance than others it is necessary to collect data for analysis inside the communication system, as EventSpace does.

MRNet [23] is the system most similar to EventSpace. Both use collective operations spanning trees to build scalable multi-cast/reduction overlay networks used by performance monitoring tools. MRNet shares the flexible organization and extensibility of EventSpace. In MRNet, communication is only between compute hosts and the front-end host, while EventSpace allows arbitrary communication structures resulting in more flexible and efficient analysis. EventSpace is also more tightly integrated with the underlying communication system, allowing the monitor activity to be coscheduled with the application. Our evaluation differs in that we use EventSpace for a different problem domain than used in [23], and we examine the performance of more complex spanning tree topologies than the balanced trees used in [23]. Another data aggregation tool for Grids is Yggdrasil [4].

PHOTON [28] allows monitoring point-to-point operations used by MPI applications run on large clusters. EventSpace is designed for collective operations, but share the same goals as PHOTON in reducing the monitoring overhead, perturbation and storage requirements of post-mortem trace analysis tools. PHOTON appends information to messages, which requires modifications to the MPI runtime system. This information is sampled and statistics are computed at runtime. Our experience in collective operation analysis [9] is that statistical profiling does not provide the necessary level of detail to understand all performance problems. Hence message tracing is necessary.

NetLogger [25] provides end-to-end application and system level monitoring of high performance distributed systems. It can provide similar performance data as EventSpace does. However, our focus is on how to aggregate and analyze the communication performance of collective operations. This requires monitoring more hosts than the single path usually monitored by NetLogger.

Data stream management systems (for an overview of DSMSs see [3]) have been used to implement network monitors [12]. DSMSs provide a relational/ query interface for the performance analyst. Such an interface could be useful for specifying EventSpace scopes as SQL queries. However, to achieve the desired performance and perturbation, it is still necessary to map, configure and tune the query plan to the clusters in use; as shown in this paper.

Astrolabe [22] is a system for collecting, aggregating and updating large scale system state. Astrolabe is targeted for widely distributed applications and the primary design goal was scalability. EventSpace uses some of the Astrolabe techniques for improving scalability such as hierarchies and aggregation. Other aggregation and filtering systems for Internet are publish-subscribe systems [10], and Grid monitoring and discovery services such as Remos [13]. The filtering and aggregation functions in EventSpace are more specialized towards performance analysis. Also, since Astrolabe and publish-subscribe systems are targeted at widely distributed applications run on the Internet, low latency aggregation is not important.

Cluster monitoring tools such as Ganglia [18], and Grid monitoring tools such as the Network Weather Service [32], does not support the high sample rate necessary for collective operation analysis.

To reduce monitoring overhead, EventSpace coschedule execution of monitoring threads with application and communication system threads. Coscheduling has traditionally been used to schedule communicating processes [1]. Our design is similar to [11], where coscheduling is used to boost the priority of communication threads doing collective communication to improve application performance. However, we do not modify kernel code since coscheduling can be added to the communication system.

Many research projects have optimized MPI collective operations. Some of the approaches used are: (i) using knowledge about the topology hierarchy, going from multi-cluster [16] to individual clusters of SMPs [24, 17] and uni-processors. (ii) taking advantage of architecture specific optimizations [24, 26], (iii) using a lower-level network protocol [14, 26], and (iv) automatically trying different algorithms and buffer sizes [27].

3 Performance Analysis and Optimization

Applications monitored by EventSpace use the PATHS communication system [5], which is an extension to the PastSet structured shared memory system [31]. Threads communicate by reading and writing tuples to shared memory buffers.

The purpose of the analysis is to detect performance problems in a spanning tree and understand how the tree can be reconfigured to improve performance. We briefly describe the metrics computed for the allreduce operation. Other synchronizing collective operations will have similar metrics. For a more detailed description see [9].

Central to the analysis are communication *paths* through the communication system starting from a thread and ending in a PastSet buffer. Each path consists of several *wrappers*; each wrapper has code that is run before and after calling the next wrapper in the path. Wrappers are used to implement communication between hosts and for instrumentation. Also, some wrappers join paths used to implement collective operation spanning trees, and handle the necessary synchronizations. The spanning tree is configured by

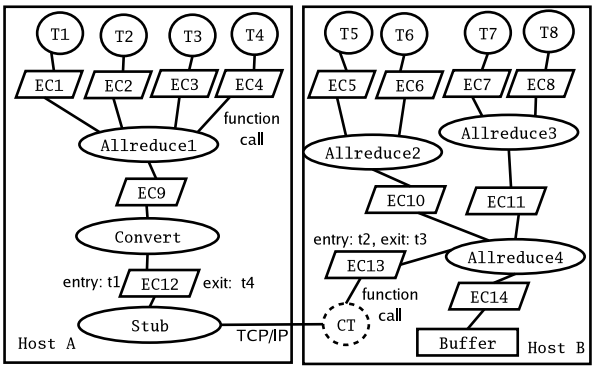

Figure 1. PATHS allreduce spanning tree.

specifying properties of the wrappers and the mapping of wrappers to cluster hosts [5, 9].

In summary, we do for the performance analysis the following steps: (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.

Figure 1 shows an allreduce spanning tree used by threads T1–T8 instrumented with *event collectors* (EC1–EC14). These collect entry and exit timestamps for each wrapper. The reduced value is stored is a PastSet buffer. CT is a communication thread serving one TCP/IP connection.

For inter-host communication we calculate the two-way TCP/IP latency by $(t_4 - t_1) - (t_3 - t_2)$, where t_1 and t_4 are collected by the event collector before the stub in a path (EC12), and t_2 and t_3 are collected by the first event collector called by the communication thread (EC13).

Allreduce wrappers are called by multiple threads each contributing with a value to be reduced. There is one event collector after the allreduce wrapper, that collects timestamps t_2 and t_3, while the paths from each contributor i have an event collector collecting timestamps $t_{1,i}$ and $t_{4,i}$. For each contributor three latencies are calculated: *down latency* $t_2 - t_{1,i}$, *up latency* $t_{4,i} - t_3$, and *total latency* $(t_{4,i} - t_{1,i}) - (t_3 - t2)$.

Also calculated for each contributor are the *arrival order distribution* and the *departure order distribution*; the number of times the contributor arrived, and departed, at the allreduce wrapper as the first, second, and so on. In addition we calculate: *arrival wait time* $t_{1,l} - t_{1,i}$; how long 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 allreduce wrapper, until contributor i departed.

4 EventSpace

The architecture of the EventSpace system is given in figure 2. An application is instrumented by inserting *event collectors* into its communication paths. Each event collector record data about communication operations into a *trace tuple* and stores it in an *event space* consisting of PastSet bounded buffers. Different *views* of the communication behavior can be provided by extracting and combining trace tuples provided by different event collectors. Consumers use an *event scope*, an aggregation/gather network, to do this.

4.1 Design

Runtime monitoring tools need to provide the data necessary for analysis at high performance and without perturbing the monitored application. We describe the design choices made in EventSpace to achieve these goals.

Configurability and extensibility. Being a research tool, EventSpace is designed to be extensible and flexible in order to experiment with different approaches for tuning the trade-off between monitoring performance and perturbation. It is also possible to extend EventSpace by adding other event collectors, and event scopes.

Separation of functional concerns. The tasks of collecting, storing, analyzing, gathering and presenting data are clearly separated in order to allow each part to be implemented and tuned separately. Data is collected by communication system wrappers, and stored using the PastSet structured shared memory system. EventSpace provides mechanisms for distributed analysis and fast collective operations for gathering data from compute hosts to a front-end host, which is responsible for presentation or further analysis of the data.

Low overhead data collection. We expect the number of trace tuple writes to be much larger than the number of reads; hence an event collector only record the minimal information for each communication operation and stores it in binary format in memory using native byte ordering. For heterogeneous environments, the tuple content can be parsed to a common format when it is read. Due to separation of concerns all communication paths are instrumented, and data is recorded for each operation, since event collectors do not know what data monitors need and when they need it.

Temporal trace storage. The challenge for large scale message tracing is the amount of data produced [28]. EventSpace provides temporal storage requiring only a few megabytes of memory (each trace tuple is 28 bytes allowing about 37 450 tuples to be stored in one megabyte of memory). The event scopes used by monitors need to have sufficient performance to read the trace tuples before they

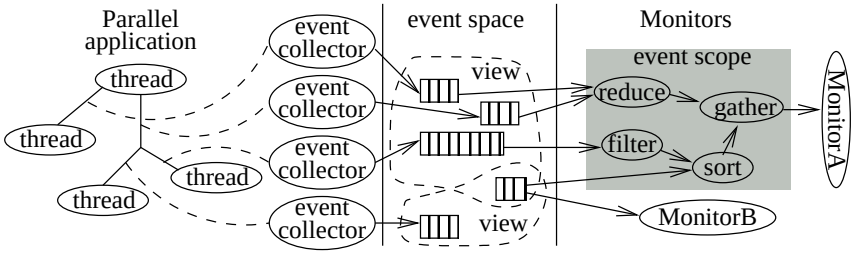

Figure 2. EventSpace architecture.

are discarded. Presently, the amount of tracing can not be dynamically be adjusted as in other monitoring systems (for example NetLogger [25]).

Distributed data analysis. Monitors use event scopes to analyze and gather data from compute hosts. The performance and perturbation of an event scope can be tuned by configuring the collective communication structures used by the event scope, and the mapping of these to the clusters. Data can be reduced or filtered close to the source, to avoid sending all data over a shared resource such as Ethernet, or a slow Internet link. Also some data preprocessing can be done on the compute clusters, thereby reducing the load on the front-end host.

Monitors using distributed analysis can be implemented either as a process on a front-end using an event scope or as a distributed application with several analysis threads. Each analysis thread can read and analyze trace tuples, and stores the result in a PastSet buffer. The results can then be gathered to a front-end for presentation.

Coscheduling. During a synchronizing collective operation all threads on a host must wait for data from other hosts. During the wait-time it is possible to run analysis threads if they are coscheduled with computation, and PATHS/PastSet communication threads. Coscheduling is possible since computation threads are blocked inside the communication system during collective operations and analysis threads also use the communication system for reading trace tuples. Hence, the release order of the different threads can be controlled by releasing all communication threads before computation threads, and finally any blocked analysis threads. No changes to the operating system scheduler are required.

On demand data gathering. Analyzing and gathering performance data comes at a cost. *Computation* is needed for the analysis, *communication* for moving data between hosts, and *storage* for intermediate results. Often these activities use the same resources as the monitored application. Pulling is used by monitors such that shared resources are not used until the data is needed.

Separation of performance concerns. Different parts of the monitoring system have different performance requirements. Event collectors run at the rate the application uses a collective operation. Some analysis threads must also run at this rate, but some lag is allowed due to the trace buffers. With distributed analysis, it is not necessary to gather all intermediate results; hence the gather rate can be lower than the event collecting rate. Further performance relaxation is allowed for presentation to users. The separation of performance concerns also makes it easier to tradeoff between monitoring performance and perturbation.

4.2 Implementation

Event Space. An event space is implemented using PastSet buffers. Each trace buffer can have a different size and lifetime. The oldest tuple is automatically discarded when the number of tuples is above a specified threshold.

Event Collectors. An event collector writes a trace tuple to a trace buffer using the blocking PastSet write operation. During the write, the traced communication operation is blocked. As a result it is important to keep the introduced overhead low. The write consist of a mutex lock, a memory copy of 28 bytes, and a mutex unblock (a read is similar). The recorded information is: event collector identifier, PastSet operation type, tuple sequence number, return value, and the start and completion timestamps.

Event scopes. An event scope for a specific monitor is implemented as a spanning tree with PATHS wrappers for: (i) storage, (ii) data manipulation including aggregation, filtering and conversion, (iii) data gathering and scattering, and (iv) inter-host communication. Storage wrappers provide access to PastSet buffers, while inter-host communication wrappers allow setting properties of TCP/IP connections such as socket buffer size. Only the data manipulation wrappers are aware of tuple format and content.

Gather wrappers read tuples from several PastSet buffers, concatenate these and returns one large tuple. Scatter divides and writes a tuple into several PastSet buffers. The gathering and scattering is done in the context of the calling thread. It is also possible to specify that a given number of helper threads should be started for the wrapper. The helper threads allow parallel reads and writes on remote PastSet buffers.

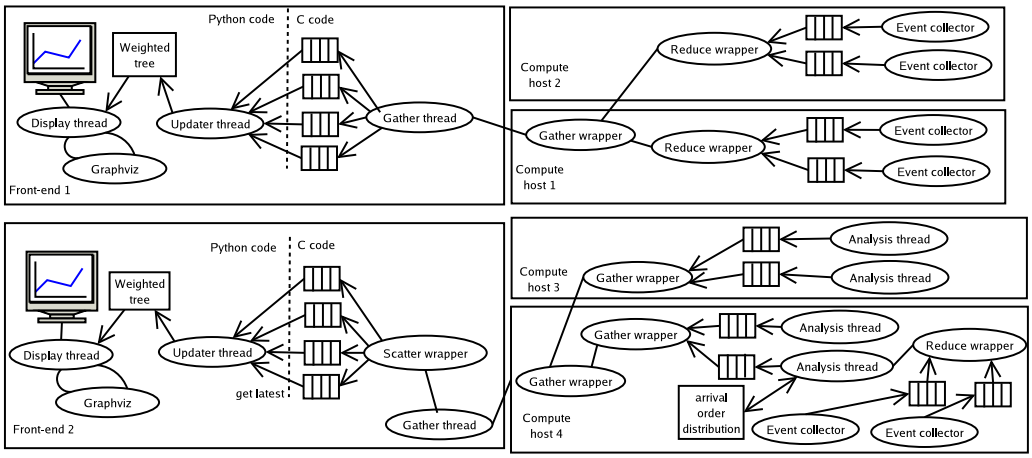

Figure 3. Load-balance monitor with a single event scope (top), and with distributed analysis (bottom).

4.3 Monitors

Load balance monitor. The load balance monitor is used to find load balance problems, which can be caused by workload imbalance, differences in point-to-point communication latency, or the mapping of a spanning tree to clusters. Two implementations are used. The first has a single event scope (figure 3). A gather thread uses the event scope to pull trace tuples produced by the event collectors on each compute host. A reduce wrapper is used to find the tuple with the largest down timestamp. All reduced tuples are then gathered to the front-end where they are scattered to PastSet buffers (one per allreduce wrapper). The tuples contain the number of last arrivals for each participant, and are read by a thread which applies updates to a weighted tree with the number of last arrivals for each participant. This tree is used to generate visualizations.

Distributed analysis reduces communication cost by increasing computation cost, but also complicates the monitor (figure 3). Each host has one analysis thread that counts the last arrivals for each participant by reading and reducing trace tuples as described above. After each read an *intermediate result* tuple is written to a PastSet buffer, containing the number of last arrivals for each participant. The gather thread gathers all intermediate result tuples from the compute hosts and scatters these to the local PastSet buffers. In the visualization we are only interested in the newest state of the system. Hence, not all intermediate result tuples need to be gathered since the arrival order state is maintained by the analysis threads.

Statistics monitor. The statistics monitor (statsm) is used to find paths with similar behavior and to detect hotspots. Computation is offloaded from the front-end by having on each compute host one or more analysis threads computing all statistics for the spanning tree wrappers on the host (figure 4). Our analysis assumes that all trace tuples are read before being discarded.

For each PATHS wrapper, statsm computes mean, minimum, maximum, standard deviation and median (using the sliding window median implementations from NWS [32] with window size set to 100) for the up, down and total latencies. For each wrapper, the results are stored in three 24 byte result tuples and written to three PastSet buffers. In addition, for allreduce wrappers similar results tuples are written for each arrival and departure order wait time. Also, for allreduce wrappers per thread arrival and departure wait time means are computed and stored in a PastSet buffer.

Two gather threads are used. The first gathers all up and down latencies in addition to the arrival and departure wait times. The second gathers per thread statistics (these are not always needed). Results are stored in two buffers at the front-end. These are used by an updater thread that maintains an analysis tree structure with statistics for each wrapper. The analysis tree is used by visualization threads.

5 Methodology

Two micro-benchmarks are monitored. In *Gsum* threads alternate between using two identical allreduce trees to compute a global sum. Gsum is run for 20 000 iterations using 8 byte messages (most scientific applications use small messages in allreduce [29]). *Compute-gsum* alternates between computing (integer sort) and calling allreduce. The benchmark can easily be perturbed since delaying one thread causes all others to wait for it [21]. Compute-gsum is run for either 10 000 or 20 000 iterations, and is tuned to spend 50% of its execution time computing and 50% in allreduce. Both have one computation thread per CPU. Each experiment is repeated at least three times and

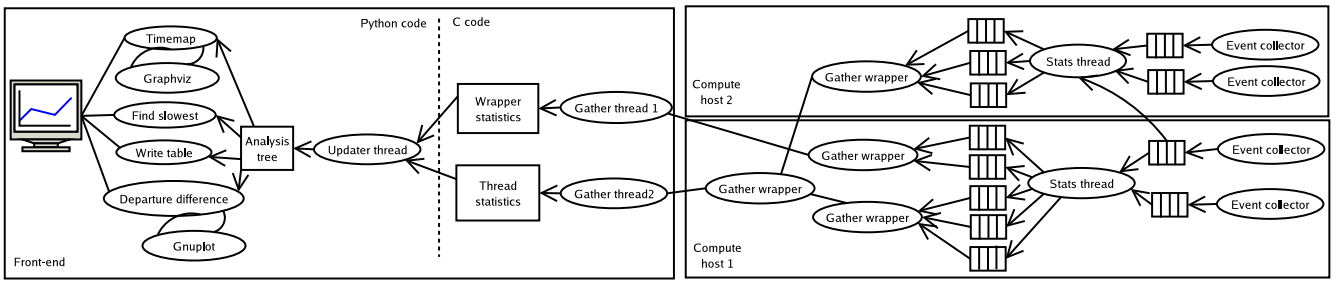

Figure 4. Statistics monitor threads and gather tree.

execution time averages are used to compute monitor overhead. Standard deviation is low (less than 1% of mean). To ensure fairness and experiment repeatability, all event scopes were set up and analysis threads were started before the monitored application.

Four clusters are used: *Copper* 18 dual-CPU Pentium II 300 MHz, 256 MB RAM, *Lead* 10 single-CPU Mobile Pentium III 900 MHz, 1024 MB RAM, *Tin* 51 single-CPU Pentium 4 Hyper-threaded 3.2 GHz, 2 GB RAM, *Iron* 39 singe-CPU Pentium 4 Hyper-threaded 3.2 GHz, 2 GB RAM with EM64T extension.

Copper and Lead share a two-way Pentium II 300 MHz with 256 MB RAM which is used as a gateway and which all communication to/from the cluster goes through. For Tin and Iron one host similar to the compute-hosts is used as gateway. A host outside the clusters, a Pentium 4 1.8 GHz with 2 GB RAM, is used as monitor front-end.

The Tin and Iron clusters have Gigabit Ethernet, while Copper, Lead, and all inter-cluster communication use 100 Mbit Ethernet. The operating system on all clusters is Linux, with the LinuxThreads Pthread library. Iron runs 32-bit code. Hyper-threading was enabled for Tin and Iron. On all TCP/IP connections the Nagle algorithm was disabled and default socket sizes were used.

We emulate WAN links between our clusters using the Longcut WAN emulator [7]. The design of Longcut is similar to the Panda WAN emulator [15]. Tin and Iron are each split into three sub-clusters. For each sub-cluster we select one host to act as a gateway. All communication to the sub-cluster is routed through its gateway, which adds delays to the routed messages to simulate the higher latency and lower bandwidth of a WAN TCP/IP connection. The emulator is implemented using PATHS wrappers.

To calculate the delay added to a message of a given size, we use a latency and bandwidth trace collected by running an instrumented communication intensive application on hosts in Tromsø, Trondheim, Odense and Aalborg. The largest latency is between Tromsø and Aalborg, and is about 36 milliseconds ([7] has additional details about the topology). The sub-clusters are assigned to these sites with two sub-clusters in Tromsø and Odense.

For each cluster we choose an allreduce spanning tree with, to our knowledge, the best performance. For Tin, Iron and Copper this is a hierarchy aware (as in [24, 17]), 8-way spanning tree, while for Lead it is a flat tree. For the LAN multi-clusters the cluster spanning trees are connected by adding an inter-cluster allreduce. For WAN multi-clusters the inter-cluster allreduce is replaced by an all-to-all for improved performance (as in MagPIe [16]). The average time per allreduce for the different topologies is about 0.5 ms for Tin with 32 hosts, 0.6 ms for Tin with 49 hosts, 1 ms for a LAN multi-clusters and 65 ms for a WAN multi-cluster (both multi-clusters with 43 Tin hosts and 39 Iron hosts).

6 Experiments

6.1 Data Collection

The overhead added to a PastSet operation by a single event collector is low (1.1 μs on a 3.2 GHz Pentium 4), compared to the hundreds of microseconds per collective operation. Thus, for the gsum and gsum-noise experiments presented below, the overhead due to event collectors range from 0–2%.

The storage requirement for temporal traces is small. For our 8-way allreduce, the hosts with most event collectors (9) stores 252 bytes per call. We use one megabyte memory for trace tuples and one megabyte for intermediate results. Thus, trace buffer size is set to 3750 tuples, and the intermediate result buffers have size set to 5000 tuples.

6.2 Event Scope

To experiment with the performance, perturbation and tuning of an event scope, we instrumented both allreduce trees used by gsum with event collectors, but only monitored one. The allreduce tree for 49 *Tin* hosts has 241 event collectors, but only data from 57 are needed to compute the arrival order at each allreduce wrapper. These are on 8 hosts, and due to the reduce wrapper only 28 bytes need to be gathered from each host. For a single cluster,

Event Scope	Overhead
Event collectors	none–1%
32 Tins, sequential	tuples discarded
32 Tins, parallel	0.4%
LAN multi-cluster, seq.	tuples discarded
LAN multi-cluster, par.	none
WAN multi-cluster, seq.	1%

Table 1. Load balance monitor with single event scope.

Event Scope	Overhead	Gather rate
49 Tins, sequential (gsum)	2%	51%
49 Tins, parallel (gsum)	2%	99%
49 Tins, sequential	1%	65%
49 Tins, parallel	1%	99%
LAN multi-cluster, seq.	none	45%
LAN multi-cluster, par.	3%	100%
WAN multi-cluster, seq.	1%	94%
WAN multi-cluster, par.	3%	100%

Table 2. Load balance monitor with distributed analysis.

the event scope has only one gather wrapper which is run on the cluster gateway. For multi-clusters the event scopes have a gather wrapper on each cluster gateway and a gather wrapper on the monitor front-end gathering from these.

Gsum. Adding event collectors to a 49 *Tin* spanning tree does not introduce a measurable overhead (monitored mean is within one standard deviation of un-monitored mean). Neither does the load balance monitor. To ensure that all trace tuples are read before being discarded, helper threads must be added to the gather wrappers such that data is gathered in parallel. LAN and WAN multi-clusters have similar results.

Compute-gsum. The largest monitoring overhead was for a multi-cluster with emulated WAN links with 49 *Tin*, 18 *Copper* and 10 *Lead* hosts (table 1). However, the overhead is caused by the WAN emulator becoming inaccurate when there are many emulated connections. As for gsum, sequential gathering has often not sufficient performance.

Scalability. For the event scope achieving sufficient performance is harder than keeping the overhead low. The event scope need to be hierarchy aware and do all intra-host reduces before inter-host gathers, and intra-cluster gathers before inter-cluster gathers. Further reconfiguration by for example moving gather wrappers to unused cluster hosts does not improve performance. Also, for the cluster sizes we had available a flat gather tree had sufficient performance. For larger clusters additional levels may be necessary.

Increasing the number of hosts by connecting clusters with LANs or WANs often lowers the performance requirements for the monitor, since the performance of the monitored operation decreases. Also, the event scopes used by monitors such as load balance scale better than allreduce trees, since data is not needed from all hosts.

The higher WAN latency is usually tolerated since the monitored operation is latency bound, and the messages sent by the event scope are small (a few hundred bytes) making them also latency bound. We believe most WAN links have enough bandwidth for concurrent transfers of application and monitor data.

The monitoring scales well with number of monitored spanning trees. Monitoring both spanning trees in gsum and gsum-compute does not increase monitoring overhead or reduce monitoring performance. Similarly modifying gsum to use four spanning trees and monitoring all trees did not increase overhead or reduce performance.

6.3 Distributed Analysis

Load balance monitor. Distributed analysis uses more resources than the single event scope. For each host with allreduce wrappers, 352 bytes are gathered (compared to 224). Also, there is additional computation cost for running the analysis threads, and storage must be allocated for intermediate results. Using distributed analysis increases monitoring overhead from none to about 2% for gsum on a single cluster (table 2). For compute-gsum the monitoring overhead has not changed.

Monitoring cost can be reduced since it is not necessary to gather all intermediate results to the front-end. Hence, the overhead on a LAN multi-cluster can be reduced from 3% to none, by removing the helper threads in all gather wrappers (parallel vs. sequential in table 2). The performance difference between sequential and parallel gather is smallest for the WAN multi-cluster, and largest for the LAN multi-cluster.

6.3.1 Statistics monitor

Gsum. The statistics monitor is a computation and communication intensive monitor; the analysis threads read data from all trace buffers on the host. Some are also read twice; when computing statistics for the wrapper before and after the associated event collector. Also, to compute TCP/IP latencies a trace tuple must be read from another host.

Initially we have one analysis thread per host. Running distributed analysis on a 32 *Tin* host spanning tree, has 9% monitoring overhead. We tried different approaches for reducing the overhead. Removing all statistics computation (but still reading trace tuples) did not reduce the overhead,

Event Scope	Overhead	Wrapper	Thread
Event collectors	none–1%	-	-
Analysis threads	5–9%	-	-
with coscheduling 1	3%	-	-
with coscheduling 2	1%	-	-
32 Tins, sequential	2%	50%	69%
32 Tins, parallel	2%	77%	99%
LAN multi, seq.	see text	43%	68%
LAN multi, par.	+1%	100%	100%
WAN multi, seq.	none	100%	100%

Table 3. Statsm overhead and gather rates.

showing that the slowdown is not caused by computation. Similarly, removing the read and computation of statistics for allreduce wrappers did not reduce the overhead. Thus the problem was not caused by synchronization in the many buffer reads. Removing statistics computation for TCP/IP connections reduced the overhead to 4%, showing that the slowdown was caused by reads on trace buffers on other hosts.

For TCP/IP connections we can choose whether statistics should be computed at the source or destination (the direction of a path is from the thread to a PastSet buffer). Moving the computation from the source to destination host reduced the overhead to 5%. However, the analysis thread was not able to read all trace tuples before they were discarded (since it reads from 8 hosts sequentially). Running two analysis threads on each host allowed reading all tuples, but increased the overhead to 6%.

Finally, we used two coscheduling strategies: (i) analysis threads are blocked until all participating threads have contributed and a message is sent to the next-level host, and (ii) analysis threads are blocked until all participating threads are unblocked. The first strategy tries to do the analysis while the host is idle waiting for the broadcasted reduced value. The second makes sure the broadcast is done before unblocking analysis threads. The first strategy reduced the overhead to 3%, while the second reduced it to 1%. For the remaining experiments the second coscheduling strategy is used.

Adding gathering increased the overhead to 2%. There was no difference in overhead when gather wrappers had helper threads, but with the latter more intermediate results could be gathered (table 3).

The allreduce spanning trees for a LAN multi-cluster with 43 *Tin* hosts and 39 *Iron* hosts had about 20% slower inter-cluster communication than expected. We were not able to reconfigure or remap the spanning tree to remove the problem. However, when data is gathered from the cluster, allreduce operation time *decreases* with up to 18%. Thus we cannot measure the gather overhead for the multi-cluster topology. But we can compare the performance of a gather tree with sequential and parallel gathering. The latter improved wrapper-, and per thread statistics gather rate, but increased monitoring overhead with 1% (table 3).

The larger latency of emulated WAN links hides the performance problem described above. With WAN links, analysis threads introduce a 1% overhead, but data gathering can be done without helper threads, without increasing the overhead, and with sufficient performance to gather all intermediate results.

Compute-gsum. For compute-gsum the execution time variation is larger than for gsum (about 2% of mean), hence we could not see any monitoring overhead. Also, the gather rate is better. Both are probably due to less communication, since compute-gsum has one less allreduce per iteration.

Scalability. Analysis thread performance is independent of cluster size, since each only monitors a subtree. However, the subtree is dependent on the spanning tree shape.

Gather scalability depends on how analysis threads are mapped to the cluster. For example in our initial configuration all hosts had analysis threads which produced intermediate results that had to be gathered, while the final configuration only had analysis threads on the hosts with allreduce wrappers.

Data gathering for multi-cluster with WAN links has better performance, relative to allreduce performance, than for a single-cluster. This could be due to the small cluster sizes used. The largest cluster had only 12 hosts, requiring only 4400 bytes to be sent over a WAN link. For larger clusters the message size would increase, probably decreasing the gather rate.

Monitoring both 32 *Tin* host allreduce spanning trees in gsum, increased the analysis thread overhead to 5%. We were not able to reconfigure the event scope or coschedule the monitoring to reduce it. The overhead is caused by increased communication activity in the monitor. Adding data gathering does not increase the overhead. Neither does increasing the number of allreduce trees to four, since the communication frequency does not increase neither for the benchmark nor the analysis threads. We have similar results for LAN multi-clusters. However, with emulated WAN links monitoring both allreduce trees does not increase the overhead, since the time between each allreduce operation call is larger (due to WAN latency), hence monitoring activity can be scheduled to run during the WAN communication part of the allreduce operation.

We also modified compute-gsum to alternate between using two and four different spanning trees. Monitoring overhead did not increase, since the number of computations, number of allreduce calls, and allreduce call frequency did not change (we reduced the size of all trace and intermediate PastSet buffers to reflect the fewer allreduce calls per spanning tree).

7 Discussion

The low monitoring overhead and high performance of EventSpace suggest that runtime analysis can be incorporated into a communication system for automatically tuning collective operation performance. In earlier work we have shown how our performance analysis approach can be used to improve allreduce performance up to 49% [9].

It is probably easier to reduce monitoring overhead and improve monitoring performance for real applications than the micro-benchmarks we used, which were designed to stress the monitoring system. We believe the benchmarks are representative for the type of applications interesting to monitor with EventSpace, but real applications will have a more complex interaction between computation, communication and I/O providing further challenges for the analysis and tuning of collective operations.

For the load balance monitor we achieved the same performance and scalability when using an aggregation network than with distributed analysis. Due to the increased complexity of distributed analysis aggregation networks should be used. However, for monitors such as the statistics monitor aggregation networks do not have the necessary performance. Event scope performance was tuned by allocating more resources to the collective operations used to implement them. Changing the spanning tree shape or mapping to clusters did not improve performance.

All our clusters use Ethernet for communication. Faster interconnects, such as Myrinet [6], will improve the performance of collective operations. Thus, application with high enough communication ratio to be interesting to monitor with EventSpace will have a higher communication frequency. This requires the analysis computation to be done in a shorter time, but the event scopes will benefit from the improved communication performance.

Even when using Ethernet, communication latency can be improved by using a lower level protocol than TCP. But, we believe it is easier to add distributed analysis than to implement an event scope with a non-reliable lower level protocol.

We have not measured, or focused, on the time to setup and initialize the event scopes (as in [23, 4]). Currently it can take seconds due to the implementation using Python and XML-RPC. A significant performance improvement is possible by using a more efficient implementation.

Coscheduling the computation threads, communication system threads and the analysis threads did reduce perturbation for one benchmark. We believe further reduction could be achieved by priority scheduling all inter-host communication such that the applications messages always had higher priority than EventSpace messages. This would require a reimplementation of the PATHS/PastSet communication system.

8 Conclusions

We have described the EventSpace system for runtime performance monitoring of collective operations within the communication system. EventSpace allows high-performance message tracing without a large perturbation of the monitored application. By combining distributed analysis with fast collective operations to gather and analyze performance data, temporal storage for only a few megabytes of data is required. Separation of performance concerns allows us to tune the different parts of the system to achieve the required monitoring overhead and performance. Close integration with the communication system allows to coschedule analysis activity with the computation and communication of the monitored application.

We evaluated different monitors for collective operation performance analysis. Our findings were as follows: (i) monitor overhead was low, from none to maximum 3%, (ii) for many monitors it is harder to get sufficient performance than low perturbation, (iii) coscheduling allowed to reduce monitoring overhead from 9% to 1% for one benchmark, (iii) the monitoring has good scalability both with regards to the number of cluster hosts, number of clusters, and number of monitored spanning trees, (iv) high performance monitoring of a WAN multi-cluster is often easier than a single cluster, and (v) performance tuning should be done by allocating more threads to a monitor rather than reconfiguring its communication structure.

9 Future Work

Our long term goal is to build automatically reconfigurable collective operations. We will build and evaluate such a system based on the data provided by the monitoring tools in this paper.

Presently we are porting the NAS parallel benchmarks [20] to PATHS/PastSet to be able to use our tools. EventSpace may also be used to monitor other type of communication systems, for example to optimize global work scheduling in distributed work queues [2]. For data Grid applications large data sets are accessed. For such applications communication performance is important, making them interesting to monitor with EventSpace.

Also, important for the usability of EventSpace are graphical tools to simplify the building and tuning of event scopes.

References

[1] A. C. Arpaci-Dusseau. Implicit coscheduling: coordinated scheduling with implicit information in distributed systems. *ACM Transactions on Computer Systems*, 19(3):283–331, 2001.

[2] R. H. Arpaci-Dusseau. Run-time adaptation in River. *ACM Transactions on Computer Systems*, 21(1):36–86, 2003.

[3] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *Proceedings of the twenty-first Symposium on Principles of database systems*, pages 1–16. ACM Press, 2002.

[4] S. M. Balle, J. Bishop, D. LaFrance-Linden, and H. Rifkin. Ygdrasil: Aggregator network toolkit for the grid. In *Proceedings of PARA'04 - Workshop on State-of-the-Art in Scientific Computing*, volume To appear of *Lecture Notes in Computer Science*. Springer, June 2004.

[5] J. M. Bjørndalen. *Improving the Speedup of Parallel and Distributed Applications on Clusters and Multi-Clusters*. PhD thesis, Department of Computer Science, University of Tromsø, 2003.

[6] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W.-K. Su. Myrinet: A gigabit-per-second local area network. *IEEE Micro*, 15(1):29–36, 1995.

[7] L. A. Bongo. The Longcut wide area network emulator: Design and evaluation. Technical Report 2005-53, Dep.of Computer Science, University of Tromsø, 2005.

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

[9] L. A. Bongo, O. Anshus, and J. M. Bjørndalen. Collective communication performance analysis within the communication system. In *Euro-Par*, volume 3149 of *Lecture Notes in Computer Science*, pages 163–172. Springer, August 2004.

[10] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf. Design and evaluation of a wide-area event notification service. *ACM Trans. Comput. Syst.*, 19(3):332–383, 2001.

[11] G. S. Choi, J.-H. Kim, D. Ersoz, A. B. Yoo, and C. R. Das. Coscheduling in clusters: Is it a viable alternative? In *Proceedings of the 2004 ACM/IEEE conference on Supercomputing*. IEEE Computer Society Press, 2004.

[12] C. Cranor, T. Johnson, O. Spataschek, and V. Shkapenyuk. Gigascope: a stream database for network applications. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 647–651. ACM Press, 2003.

[13] P. Dinda, T. Gross, R. Karrer, B. Lowekamp, N. Miller, P. Steenkiste, and D. Sutherland. The architecture of the Remos system. In *Proc. 10th IEEE Symp. on High Performance Distributed Computing*, 2001.

[14] 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*, pages 95–106. ACM Press, 2003.

[15] 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*, 18(8):1113–1125, 2002.

[16] 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*, pages 131–140. ACM Press, 1999.

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

[18] M. Massie, B. Chun, and D. E. Culler. The Ganglia distributed monitoring system: Design, implementation, and experience. *Parallel Computing*, 30, 2004.

[19] S. Moore, D.Cronk, K. London, 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*. Springer Verlag, 2001.

[20] NASA. NAS Parallel Benchmarks.

[21] 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.

[22] 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)*, 21(2):164–206, 2003.

[23] P. C. Roth, D. C. Arnold, and B. P. Miller. MRNet: A software-based multicast/reduction network for scalable tools. In *Proceedings of the 2003 ACM/IEEE conference on Supercomputing*. IEEE Computer Society Press, 2003.

[24] 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.

[25] B. Tierney, W. E. Johnston, B. Crowley, G. Hoo, C. Brooks, and D. Gunter. The NetLogger methodology for high performance distributed systems performance analysis. In *Proc. 7th IEEE Symp. On High Performance Distributed Computing*, pages 260–267, 1998.

[26] 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.

[27] S. S. Vadhiyar, G. E. Fagg, and J. Dongarra. Automatically tuned collective communications. In *Proceedings of the 2000 ACM/IEEE conference on Supercomputing*, 2000.

[28] J. Vetter. Dynamic statistical profiling of communication activity in distributed applications. In *Proceedings of the 2002 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 240–250. ACM Press, 2002.

[29] 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.

[30] 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.

[31] B. Vinter. *PastSet a Structured Distributed Shared Memory System*. PhD thesis, Department of Computer Science, University of Tromsø, 1999.

[32] R. Wolski, N. T. Spring, and J. Hayes. The network weather service: a distributed resource performance forecasting service for metacomputing. *Future Generation Computer Systems*, 15(5–6), 1999.