EventSpace - Exposing and Observing Communication Behavior of Parallel Cluster Applications

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

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

Abstract. This paper describes the motivation, design and performance of EventSpace, a configurable data collecting, management and observation system used for monitoring low-level synchronization and communication behavior of parallel applications on clusters and multi-clusters. *Event collectors* detect events, create *virtual events* by recording timestamped data about the events, and then store the virtual events to a *virtual event space. Event scopes* provide different *views* of the application, by combining and pre-processing the extracted virtual events. Online monitors are implemented as consumers using one or more event scopes. Event collectors, event scopes, and the virtual event space can be configured and mapped to the available resources to improve monitoring performance or reduce perturbation. Experiments demonstrate that a wind-tunnel application instrumented with event collectors, has insignificant slowdown due to data collection, and that monitors can reconfigure event scopes to trade-off between monitoring performance and perturbation.

1 Introduction

As the complexity and problem size of parallel applications and the number of nodes in clusters increase, communication performance becomes increasingly important. Of eight scalable scientific applications investigated in [12], most would benefit from improvements to MPI's collective operations, and half would benefit from improvements in point-to-point message overhead and reduced latency.

The performance of collective operations has been shown to improve by a factor of 1.98 by using better mappings of computation and data to the clusters [2]. Point-to-point communication performance can also be improved by tuning configurations. In order to tune the performance of collective and point-to-point communication, fine-grained information about the applications communication events is needed to compute how the application behaves.

[0] This research was supported in part by the Norwegian Science Foundation project "NOTUR", sub-project "Emerging Technologies - Cluster"

In this paper we describe EventSpace, an approach and a system for online monitoring the communication behavior of parallel applications. It is the first step toward a system that can dynamically reconfigure an applications communication structure and behavior.

For low-level performance analysis [3, 11] and prediction [14, 5], large amounts of data may be needed. For some purposes the data must be consumed at a high rate [9]. When the data is used to improve the performance of an application at run-time, low perturbation is important [8, 10]. To meet the needs of different monitors the system should be flexible, and extensible. Also the sample rate, latency, perturbation, and resource usage should be configurable [8, 5, 11]. Finally, the complexity of specifying the properties of such a system must be handled.

The approach is to have a *virtual event space*, that contains traces of an applications communication (including communication used for synchronization). *Event scopes* are used by consumers to extract and combine virtual events from the virtual event space, providing different *views* of an applications behavior. The output from event scopes can be used in applications and tools for adaptive applications, resource performance predictors, and run-time performance analysis and visualization tools. When triggered by communication events, *event collectors* create a virtual event, and store it in the virtual event space. A virtual event comprises timestamped data about the event.

The EventSpace system is designed to scale with regards to the number of nodes monitored, the amount of data collected, the data observing rate, and the introduced perturbation. Complexity is handled by separating instrumentation, configuration, data collection, data storage, and data observing.

The prototype implementation of the EventSpace system is based on the PATHS [1] system. PATHS allows configuring and mapping of an applications *communication paths* to the available resources. PATHS use the concept of *wrappers* to add code along the communication paths, allowing for various kinds of processing of the data along the paths. PATHS use the PastSet [13] distributed shared memory system. In PastSet *tuples* are read from and written to named *elements*.

In EventSpace, an application is instrumented when the configurable communication paths are specified. Event collectors are implemented as PATHS wrappers integrated in the communication system. They are triggered by PastSet operations invoked through the wrapper. The virtual event space is implemented by using PastSet. There is one *trace element* per event collector.

Event scopes are implemented using scalable hierarchical *gather trees*, used to gather data from a set of trace elements. PATHS wrappers to filter, sort, convert, or reduce data can be added to the tree as needed. To trade-off between performance and perturbation the tree can be mapped to the available resources, and wrapper properties can be set.

This paper proceeds as follows. In section 2 we discuss related work. The architecture and implementation of EventSpace are described in sections 3 and 4. Monitors using EventSpace are described in section 5. The performance of

EventSpace is evaluated in section 6. In section 7 we draw conclusions and outline future work.

2 Related Work

There are several performance analysis tools for parallel programs [6]. Generally such tools provide coarse grained analysis with focus on processor utilization [11]. EventSpace supplements these tools by providing detailed information about the internal behavior of the communication system.

NetLogger [11] provides detailed end-to-end application and system level monitoring of high performance distributed systems. Analysis is based on lifelines describing the temporal trace of an object through the distributed system. EventSpace provides data for similar paths inside the communication system. However, data is also provided for joined and forked paths forming trees, used to implement collective operations and barriers.

There are several network performance monitoring tools [5, 7]. While these often monitor low level network data, EventSpace is used by monitors monitoring paths that are used to implement point-to-point and collective communication operations. Such a path may in addition to a TCP connection, have code to process the data, synchronization code, and buffering.

JAMM [10] is a monitoring sensor management system for Grid environments. Focus is on automating the execution of monitoring sensors and the collection of data. In JAMM sensors generate events that can be collected, filtered and summarized by consumers using event gateways. In EventSpace, events are generated by sensors integrated in the communication system, and consumers collect, filter, and preprocess data using event scopes. Event scopes are implemented by extending the PATHS system, allowing the data flow to be configured at a fine granularity.

The Network Weather Service [14] is a system service producing short-term forecast for a set of network and computational resources. In EventSpace monitoring is per application, hence data is only collected for the resources used by the application simplifying the resource usage control.

EventSpace, and many other monitoring systems [5, 8, 11, 14] separate producers and consumers, and allow intermediates to filter, broadcast, cache, and forward data. Most monitoring systems also have a degree of configurability [5, 8, 11, 14]. In EventSpace focus is on allowing all parts of the monitoring to be configured and tuned at a fine granularity.

3 EventSpace Architecture

The architecture of the EventSpace system is given in figure 1. An application is instrumented by inserting *event collectors* into its communication paths. Each event collector record data about communication events, creates a virtual event based on the data, and stores it in a virtual event space. Different *views* of the

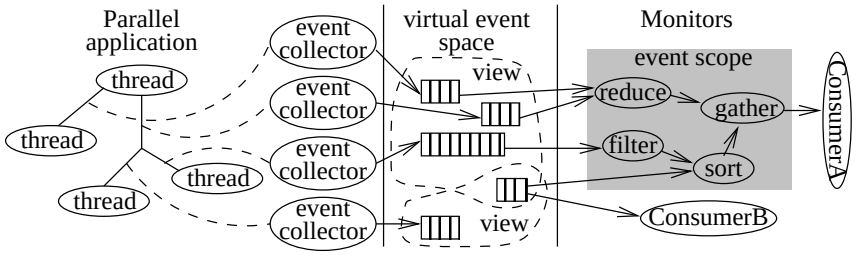

Fig. 1. EventSpace overview.

communication behavior can be provided by extracting and combining virtual events provided by different event collectors. Consumers use an *event scope* to do this.

An event collector record operation type, operation parameters, and start and completion times[1] of all operations sent through it. Typically, several event collectors are placed on a path to collect data at multiple points.

EventSpace is designed to let event collectors create and store events, with low overhead introduced to the monitored communication operations. Shared resources used to extract and combine virtual events are not used until the data is actually needed by consumers. We call this *lazy event processing*. By using lazy processing we can, without heavy performance penalties, collect more data than may actually be needed. This is important because we do not know the actual needs of the consumers, and we expect the number of writes to be much larger than the number of reads [9].

EventSpace is designed to be extensible and flexible. The event collectors and event scopes can be configured and tuned to trade off between introduced perturbation and data gathering performance. It is also possible to extend EventSpace by adding other event collectors, and event scopes.

The communication paths used by event collectors and event scopes can also be instrumented and monitored. Consequently, a consumer can monitor its own, another consumer's, or an event collectors performance and overhead.

4 EventSpace Implementation

The implementation of EventSpace is built on top of PATHS and PastSet. Presently, the monitored applications must also use PATHS and PastSet.

PastSet is a structured distributed shared memory system in the tradition of Linda [4]. A PastSet system comprises a number of user-level *tuple servers* hosting PastSet *elements*. An element is a sequence of tuples of the same type. Tuples can be read from and written to the element using blocking operations.

[1] Using the Pentium timestamp counter.

PATHS supports mapping of threads to processes, processes to hosts, specifying and setting up physical communication paths to individual PastSet elements, and insertion of code in the communication paths. This code is executed every time the path is used.

A path is specified by listing the stages from a thread to a PastSet element. At each stage the wrapper type and parameters used to initialize an actual instance of the wrapper are specified. A wrapper is typically used to run code before and after forwarding the PastSet operation to the next stage in the path.

Paths can be joined or forked forming a tree structure. This supports implementation of collective operations, barriers and EventSpace gather trees.

The threads, elements, and all communication paths used used by an application are specified in a *pathmap*. It also contains information about which paths are instrumented, and the properties of event collectors and trace elements. The path specifications are generated by path generate functions. As input to these functions three maps are used: (1) An application map describing which threads access which elements. (2) A cluster map, describing the topology and the nodes on each cluster. (3) An application to cluster map, that describes the mapping of threads and elements to the nodes.

Threads can use elements in an access and location transparent manner, allowing the communication to be mapped onto arbitrary cluster configurations simply by reconfiguring the pathmap. Presently, run-time reconfiguration of the pathmap is not implemented.

4.1 Event Collectors

When triggered, an event collector creates a virtual event in the form of a *trace tuple*, and writes it to a *trace element*. A virtual event space is implemented by a number of trace elements in PastSet. Each trace element can have a different size, lifetime, and be stored in servers locally or remotely from where the communication event took place.

The trace tuple is written to a trace element using a blocking PastSet operation. As a result it is important to keep the introduced overhead low. If the trace element is located in the same process as the event collector (a local server), the write only involves a memory copy and some synchronization code. To keep the introduced overhead low, trace tuples are usually stored locally.

Tuples can be removed either by explicit calls, or automatically discarded when the number of tuples is above a specified threshold (specified on a per element basis). Presently, for persistent storage some kind of *archive consumers* are needed.

The recorded data is stored in a 36 byte tuple. Since write performance is important, tuples are stored in binary format, using native byte ordering. For heterogeneous environments, the tuple content can be parsed to a common format when it is observed.

By using PATHS to specify the path from an event collector wrapper to a trace element, we can specify the location of the trace element, its size, and its properties.

Presently, the amount of tracing cannot dynamically be adjusted as in [8, 11]. However, when a consumer decides to start monitoring a part of the system, a backlog of collected events can be examined.

4.2 Event Scopes

An event scope is used to gather and combine virtual events providing a specific *view* of an applications communication behavior. It can also do some preprocessing on the virtual events. An event scope is implemented using a configurable *gather tree*. The tree is built using PATHS wrappers. Even if a tree can involve many trace elements with wrappers added at arbitrary places, the complexity of building and configuring it is reduced due to the hierarchical nature of views, resulting in a regular structure of the tree. For example, the Heartbeat view in figure 2a, comprises two node views, each comprising two thread views. The tree can be built hierarchically by creating similar sub-trees for each sub-view. Any number of event scopes can be dynamically built by different consumers using the applications pathmap as input.

The desired performance and perturbation of a gather tree are achieved by mapping the tree to available resources and setting properties of the wrappers. 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 node where the consumer is running. Since data is gathered on-demand, shared resources such as CPU, memory and networks are only used when needed.

5 Monitors - Consumers of Virtual Events

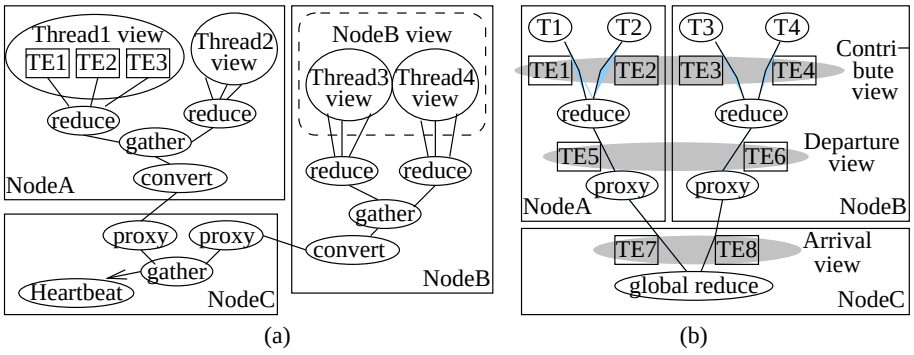

Fig. 2. (a) Heartbeat gather tree. (b) Data collected for a reduce operation tree.

In this section we describe several monitors that use event scopes to extract and process virtual events.

5.1 Heartbeat Monitor

The Heartbeat monitor has the task of establishing whether a thread has reached a stable state where no further progress can take place or not. The monitor uses an event scope as shown in figure 2a, to find the last virtual event for each thread of the application. The event scope extract the latest virtual event for every communication path used by a thread, and then selects the last of these events ("reduce"). The output ("gather") from the event scope is one single timestamp for each thread representing the last event happening in each thread.

After a set time the monitor use the event scope to reduce and gather new timestamps. These are compared with the previous values. If no change is detected the thread has had no progress.

5.2 Get Event Monitor

GetEventMon just consumes virtual events without further processing of the data. In [3] the data collected by GetEventMon is used for performance analysis and visualization of the behavior of an application.

GetEventMon uses an event scope with a gather tree similar to Heartbeat's, except that all reduce wrappers are replaced with gather wrappers.

5.3 Collective Operation Monitor

ColOpMon monitors the performance of MPI type collective operation, implemented and instrumented using PATHS. As these operations use broadcast, gather, scatter, and reduce trees, we collect data about the activity in the trees for later extraction from the virtual event space, and analysis by ColOpMon.

ColOpMon is a multi-threaded distributed application, with its own pathmap. It has threads monitoring the performance of each internal tree node, and one thread monitoring the contribution times of threads participating in the collective operation.

In figure 2b the different views for (a part of) a reduce operation tree are shown, where four threads, T1 - T4 on nodes A and B, do a partial reduce on each node before the global reduce on node C. The path is instrumented using event collectors that store events in trace elements TE1 - TE8. An internal node monitor thread gathers, does clock synchronization, and analyzes events from the departure and arrival views. A contribution time monitor gathers and analyzes events from the contribute view.

6 Experiments

To demonstrate the feasibility of the approach and the performance of the EventSpace prototype, we monitor a wind-tunnel application.

The hardware platform comprise two clusters:

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

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

The clusters use TCP/IP over a 100 Mbps Ethernet for intra and inter-cluster communication. Communication to and from the 4W cluster goes through a two-way Pentium II 300 MHz with 256 MB RAM, while the 8W nodes are directly accessible.

The wind-tunnel is a Lattice Gas Automaton doing particle simulation. We use eight $2N \times N$ matrices. Each matrix is split into 140 slices, which are assigned to 140 threads. Each thread uses 12 PastSet elements to exchange the border rows of its slices with threads computing on neighboring slices. We use three different problem sizes: *large*, *medium* and *small*.

Large is the largest problem size that does not trigger paging on the 4W nodes, while *medium* and *small* are 1/2 and 1/8 of the size of *large*[2].

For the large problem size, border rows are exchanged approximately every 300 ms (and thus virtual events are produced at about 3.3 Hz). For *medium* and *small*, rows are exchanged every 70 ms and every 5 ms respectively. The discard threshold is set to 200 tuples for all 2240 trace elements. Less than 0.5 MB of memory on a 4W node is used for trace tuples.

6.1 Event Collecting Overhead

The overhead introduced to the communication path by a single event collector wrapper is measured by adding event collector wrappers before and after it. The average overhead, calculated using recorded timestamps in these wrappers, is 1.1 μs on a 1.8 GHz Pentium 4 and 6.1 μs on a 200 MHz 8W node. This is comparable to overheads reported for similar systems [8, 11].

For *large* and *medium*, the slowdown due to data collection is insignificant. For *small*, the slowdown is 1.03.

Further experiments are needed to determine if the insignificant and small slowdowns are due to unused cluster resources, and the effect of the overhead introduced by event collectors on other applications.

6.2 Event Scope Perturbation and Performance

In this section we document how reconfiguring an event scope can be used to trade-off between the rate at which virtual events can be observed for a given view, and the perturbation of monitored application. We measure how many times a monitor can pull, using an event scope, one tuple from all trace elements in a view during a run of the wind-tunnel. All monitors are run on a computer outside the cluster[3], if not otherwise stated.

GetEventMon consumes events from a single trace element, a thread view with 14 elements, and a node view with 266 elements (19 thread views) with no

[2] As the size of the matrices, N, increase the computation increase by $O(N^2)$, and the communication by N. For *large*, N is 7680.

[3] A 1.8 GHz Pentium 4 with 2 GB RAM.

slowdown for *large*, and 1.03 for *small*. Events are consumed at 2500 Hz, 1500 Hz, and 200 Hz respectively.

When consuming events from a multi-cluster view with 1948 trace elements (140 thread views), the wind-tunnel has no slowdown for *large*. However when using the small problem size, the event scope introduces a slowdown of 1.36. Using a different gather tree shape increases the observe rate from 58 Hz to 106 Hz, and the slowdown to 1.50. Reconfiguring the event scope to use less threads, reduces the slowdown to 1.07, and the observe rate to 25 Hz.

When consuming from another multi-cluster view for *large*, with only two trace elements per thread, there is no difference in sample rates and slowdown when consuming events sequentially and concurrently. However, when more processing are added to the communication paths the concurrent version is faster, due to better overlap of communication and computation.

The event scopes used by the collective operation monitor, ColOpMon, results in a slowdown of 1.17. We discovered that an event scope actually perturbs the wind-tunnel more than the computation intensive internal node monitoring threads running on the clusters. By reconfiguring the event scope to use less resources the slowdown is reduced to 1.08 (the observe rates are also decreased by about 50%).

When running four monitors concurrently, the slowdown is about the same as the largest slowdown caused by a single monitor (ColOpMon). For the consumers running on the computer outside the cluster, observe rates are reduced by 10-50%. For the ColOpMon internal monitor threads running on the clusters, observe rates are unchanged.

7 Conclusions and Future Work

The contributions of this work are two-fold: (i) we describe the architecture and design of a tunable, and configurable framework for low-level communication monitoring, and (ii) we demonstrate its feasibility and performance.

The approach is to have event collectors integrated in the communication system, that when triggered by communication events, create a virtual event that contains timestamped information about the event. The virtual events are then stored in a virtual event space from where they can be extracted by consumers using different event scopes.

Low-level communication monitoring is implemented by adding event collection code to communication paths. The data is stored in a structured shared memory. The data is extracted, combined and pre-processed using configurable communication paths. The architecture and its implementation, allows consumers, producers, and data management issues to be clearly separated. This makes handling the complexity simpler.

We have described several monitors using EventSpace, and given initial performance results. The results show that a wind-tunnel application instrumented with event collectors, has insignificant slowdown due to data collection. We have

also documented how event scopes can be reconfigured to trade-off between monitoring performance and perturbation.

Currently, we are using other types of applications and monitors to evaluate the performance and configurability of the system. We are also using EventSpace to analyze and predict the performance of MPI type collective operations, with the purpose of dynamically adapting these for better performance.

References

1. Bjørndalen, J. M., Anshus, O., Larsen, T., and Vinter, B. Paths - integrating the principles of method-combination and remote procedure calls for run-time configuration and tuning of high-performance distributed application. *Norsk Informatikk Konferanse* (2001).
2. Bjørndalen, J. M., Anshus, O., Vinter, B., and Larsen, T. Configurable collective communication in LAM-MPI. *Proceedings of Communicating Process Architectures 2002, Reading, UK* (2002).
3. Bongo, L. A., Anshus, O., and Bjørndalen, J. M. Using a virtual event space to understand parallel application communication behavior, 2003. Technical Report 2003-44, Department of Computer Science, University of Tromsø.
4. Carriero, N., and Gelernter, D. Linda in context. *Commun. ACM 32*, 4 (April 1989).
5. Dinda, P., Gross, T., Karrer, R., Lowekamp, B., Miller, N., Steenkiste, P., and Sutherland, D. The architecture of the Remos system. In *Proc. 10th IEEE Symp. on High Performance Distributed Computing* (2001).
6. Moore, S., D.Cronk, London, K., and J.Dongarra. Review of performance analysis tools for MPI parallel programs. In *8th European PVM/MPI Users' Group Meeting, Lecture Notes in Computer Science 2131* (2001), Y. Cotronis and J. Dongarra, Eds., Springer Verlag.
7. http://www.caida.org/tools/taxonomy/.
8. Ribler, R. L., Vetter, J. S., Simitci, H., and Reed, D. A. Autopilot: Adaptive control of distributed applications. In *Proc. of the 7th IEEE International Symposium on High Performance Distributed Computing* (1998).
9. Tierney, B., Aydt, R., Gunter, D., Smith, W., Taylor, V., Wolski, R., and Swany, M. A grid monitoring architecture. *Tech. Rep. GWD-PERF-16-2, Global Grid Forum, January 2002.* (2002).
10. Tierney, B., Crowley, B., Gunter, D., Holding, M., Lee, J., and Thompson, M. A monitoring sensor management system for Grid environments. In *Proc. 9th IEEE Symp. On High Performance Distributed Computing* (2000).
11. Tierney, B., Johnston, W. E., Crowley, B., Hoo, G., Brooks, C., and Gunter, D. The NetLogger methodology for high performance distributed systems performance analysis. In *Proc. 7th IEEE Symp. On High Performance Distributed Computing* (1998).
12. Vetter, J. S., and Yoo, A. An empirical performance evaluation of scalable scientific applications. In *Proceedings of the 2002 ACM/IEEE conference on Supercomputing* (2002).
13. Vinter, B. *PastSet a Structured Distributed Shared Memory System.* PhD thesis, University of Tromsø, 1999.
14. Wolski, R., Spring, N. T., and Hayes, J. The network weather service: a distributed resource performance forecasting service for metacomputing. *Future Generation Computer Systems 15*, 5–6 (1999).