# Using a Virtual Event Space to Understand Parallel Application Communication Behavior

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

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

**Abstract.** We have developed EventSpace, a configurable data collecting, management and observation system for monitoring low-level synchronization and communication events with the purpose of understanding the behavior of parallel applications on clusters and multi-clusters. Applications are instrumented by adding data collecting code in the form of *event collectors* to an applications *communication paths*. When triggered these create and store *virtual events* to a *virtual event space*. Based on the meta-data describing the communication paths, virtual events can be combined to provide different *views* of the applications communication behavior. We used the data collected by EventSpace to do a post-mortem analysis of a wind-tunnel application, a river simulator, global clock synchronization, and a hierarchical barrier benchmark. The views allowed us to detect anomalous communication behavior, detect load balance problems, analyze hierarchical barriers, synchronize the Pentium timestamp counters on the cluster nodes, and analyze the accuracy of the synchronization.

## 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 [16], 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.

However, having sufficient understanding of the communication behavior when reconfiguring the communication is difficult. A tool is needed that can monitor the communication behavior of the application, aid in analyzing the results, and present them in a useful manner. In this paper, we describe how the EventSpace [3] monitoring approach can be used for understanding the low level communication and synchronization behavior of parallel applications.

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

In EventSpace, *event collectors* are triggered by communication events. An event collector creates a virtual event, and stores it in a *virtual event space*. A virtual event comprises an identifier, timestamps, and other contextual data about the communication event. *Event scopes* are used to observe virtual events in the virtual event space. Event scopes can combine virtual events in various ways, providing different *views* of an applications behavior.

The prototype implementation of the EventSpace system is based on the PATHS [1] system. PATHS allows for configuring and mapping the *communication paths* of an application to the resources of the system used to execute the application. 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 [17] distributed shared memory system. In PastSet *tuples* are read from and written to named *elements*.

This paper makes the following two contributions: (i) we describe the architecture and design of a a tunable, and configurable framework for low-level communication monitoring, and (ii) we show how it can be used for analyzing different parallel applications.

This paper proceeds as follows. In section 2 we relate EventSpace to other monitoring systems. The EventSpace monitoring approach and system are described in sections 3, and 4. We explore how the data collected by EventSpace can be used for analysis in section 5. Finally, in section 6 we draw conclusions and outline future work.

## 2 Related Work

There are several performance analysis tools for message passing parallel programs [9]. Generally such tools provide coarse grained analysis with focus on processor utilization [14]. EventSpace supplements these tools by providing data for analysis and visualization of low-level behavior of the communication system. We expect EventSpace to be used together with other tools. For instance to understand why collective operations or synchronization operations have poor performance, once other tools have identified it as a problem.

NetLogger [14] 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. Using EventSpace, similar paths inside the communication system can be analyzed. However, the paths are joined and forked, forming trees used to implement collective operations and barriers. Hence the paths are more complex, and the analysis and visualization might involve several threads, and several concurrent events.

There are several network performance monitoring tools [10]. 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.

In EventSpace an application is instrumented by adding data logging code to its communication paths. A similar approach is used by a firmware based distributed shared virtual memory (DSVM) system monitor for the SHRIMP multicomputer [7]. Here monitoring code is embedded in programmable network interfaces to collect network-level data. The data is tied to higher level software events. In [6] a tool that measures the performance of an application and the DSVM system is described. Monitoring structured shared memory systems is different, since communication is explicit. Hence, it is easier to tie trace data to higher-level software events. Also we have explicit meta-data available about the communication paths, showing the data flow between threads.

Prism [12] is a debugger for multi-process MPI programs that supports performance analysis and (application data) visualization. They do post-mortem clock synchronization similar to ours, but collect the data by running a separate MPI job. Our approach uses already collected data, and hence has no additional data collection perturbation.

## 3 EventSpace Approach

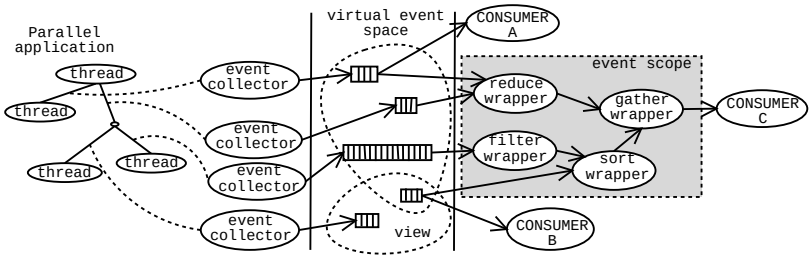


**Fig. 1.** EventSpace overview.

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

Each event collector record operation type, operation parameters, and start and completion times of all operations invoked 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 [13].

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.

## 4 EventSpace Monitoring

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 according to parameters given at call time. The read and write operations blocks the caller until they have finished.

PATHS supports mapping of threads to hosts, specifying and setting up physical communication paths to individual PastSet elements, and insertion of code in the communication paths.

A path is specified by the stages needed to bind a thread to a PastSet element. Each stage is specified by a *wrapper type*, and parameters used to initialize an actual instance of the wrapper. A wrapper comprises code and data to handle the data flow at a given stage. The wrappers in a path are executed every time the path is used.

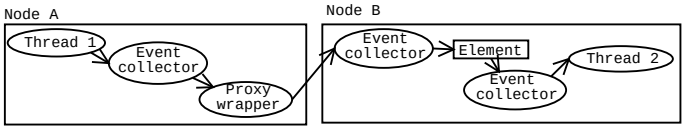


**Fig. 2.** Two threads communicating using a PastSet element.

In figure 2, the path from *thread 1* to the element consists of a a proxy wrapper and two event collector wrappers. The proxy wrapper is used to access wrappers on remote nodes by specifying parameters such as the remote nodes name and protocols to use. The event collector wrappers are described below.

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

Threads can use elements in an access and location transparent manner, allowing the communication to be mapped onto arbitrary cluster configurations simply by changing the path specifications.

### 4.1 Specifying Communication Paths

Path specifications are generated by path generate functions. Input to these functions are three mappings: (1) An *application mapping* describing which threads access which elements. (2) A *cluster mapping* describing the topology and the nodes of each cluster. (3) An *application to cluster mapping* describing the mapping of threads and elements to the nodes.

The set of all paths in an application define a *path map*. Based on the path map, PATHS do the actual set up of the paths. The path map is also used by various tools, including tools for analysis and visualization.

### 4.2 Instrumenting Communication Paths

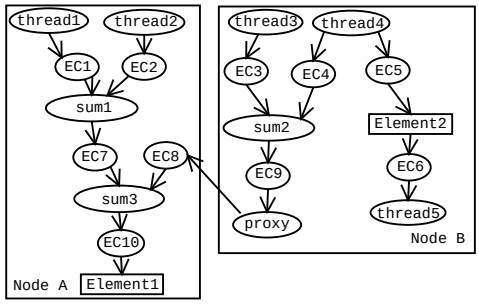


**Fig. 3.** Instrumentation of a global sum reduction tree: Paths are instrumented with event collector wrappers (EC1 - EC10). The global sum is stored in element 1. Thread 5 does not participate in the global sum.

The application is instrumented by adding event collector wrappers to the communication paths. Event collectors can be added anywhere in the path. The lower cost of using lazy event processing approach, typically makes it feasible to add an event collector before and after every other wrapper, as shown in figure 3. The figure shows how a global sum reduction tree is instrumented with event collectors. An event collector is placed before and after each wrapper doing a partial sum.

The event collectors can produce large amounts of data. For a multi-cluster wind-tunnel (described in section 5.1) with 332 threads, we collected over 165 MB of data in five minutes.

### 4.3 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. If the trace element is located on a local server, the write only involves a memory copy and some synchronization code[1].

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.

As an operation passes down and up the path it can pass through several wrappers. Timestamps are recorded on both passes, but only one tuple is created, and written on the up pass. For example in figure 3, 6 timestamps are recorded for operations on element 1 from thread 1. The timestamps are recorded using the high-resolution Pentium timestamp counter.

We have measured the overhead of an event collector to be between 0.5 $\mu$s to 6.1 $\mu$s depending on the architecture it is run on [3]. This is comparable to systems such as Autopilot [11], NetLogger [14], and SHRIMP [7].

### 4.4 Event Scopes

An event scope is used to gather and combine virtual events providing a specific *view* of an applications communication behavior (as described in section 4.5). It can also do some pre-processing on the virtual events. An event scope is implemented using a *gather tree*. The tree is built using PATHS wrappers. The desired performance and perturbation of a gather tree are achieved by mapping the tree to available resources and setting properties of the wrappers.

In this paper we focus on what kind of views can be provided, rather than how the view data can be gathered at run-time. This, and the performance and perturbation of different event scopes are described in [3].

### 4.5 Views

During analysis, several *views* of the communication behavior of the application are used. A view can be used for inspection of the behavior of high-level abstractions such as threads, or to look at the individual phases of collective operations

[1] A local server is in the same process as the event collector.

such as a global sum reduction. Views can be hierarchical, for example, a view can comprise the views of all threads on a node.

To establish a view, the path map is used to correlate data from a group of trace elements to higher-level abstractions. The path map gives a specification of the causality between trace wrapper events along a given path, while the time stamps gives a temporal ordering of the communication events per node.

Below are some examples on how the trace wrappers in figure 3 can provide different views. Examples on how views are used in communication behavior analysis are given in section 5.

By combining data from all top-level event collectors (EC1 - EC6), we get a *communication oriented* view. It provides information about *where* and *when* each thread is using the communication system.

The data from EC5 and EC6, provides information on how thread 4 and thread 5 use element 2. It can be used to determine the order of reads and writes to the element.

The overhead of a remote operation can be calculated by subtracting the operation time recorded in EC8 from the operation time recorded in EC9. The same information can also be used to synchronize the clocks in A and B, as described in section 5.4.

For the global sum in figure 3 event collectors EC1-EC4 provides a *thread wait* view, that can be used to calculate how long each thread wait for the operation to complete.

Event collectors EC7 and EC9 provides a *signal departure* view[2] for partial summation no. 3. Event collectors EC7 and EC8 provides a *signal arrival* view for partial summation no. 3. EC7, EC8 and EC10 provides a *performance oriented* view of the partial summation no. 3.

It is also possible to have more complex views, such as which threads are waiting for which other threads at a given time, as shown in section 5.2.

### 4.6 Visualization

To visualize the different views, we use timeline graphs, line charts, bar charts, and tables with statistics. A connection graph is used to visualize the path specifications themselves. The visualizations of the different views can interactively be controlled, and the functionality can be extended. Some examples on the visualizations, and how they are used are given in section 5.

### 4.7 Analysis and Visualization Tools

We use several simple tools for analysis and visualization. Presently, the tools read virtual events from files. We are working on an implementation where the tools use event scopes directly.

[2] This requires an accurate global clock synchronization.

The tools are implemented using Python[3] and Tkinter[4]. Graphviz[5] is used to draw the path specification graphs, Blt.Graph[6] is used to draw the line and bar charts, statistics are calculated using the Python stats module[7]. Using a high level language and existing tools allowed us to easily develop a prototype.

## 5 Experiments

In this section we provide examples on how the data in a virtual event space can be used for analyzing the communication behavior of parallel applications, and how the approach can be used for clock synchronization.

The hardware platform consists of three clusters, each with 32 processors:

- Two-way cluster (2W): 16 * 2-way Pentium III 450 MHz, 256 MB RAM (Odense, Denmark). Accessed through a 2-way Pentium III 800 MHz with 256 MB RAM.
- Four-way cluster (4W): 8 * 4-way Pentium Pro 166 MHz, 128 MB RAM (Tromsø, Norway). Accessed through a 2-way Pentium II 300 MHz with 256 MB RAM.
- Eight-way cluster (8W): 4 * 8-way Pentium Pro 200 MHz, 2GB RAM (Tromsø, Norway). Accessed directly.

All clusters use TCP/IP over a 100 Mbps Ethernet for intra-cluster communication. For communication between 4W and 8W we use the departments 100Mbit local area network. Communication between Tromsø and Odense is the departments Internet backbone.

### 5.1 Wind-tunnel

In this section we analyze the communication behavior of a wind-tunnel simulator, a Lattice Gas Automaton doing particle simulation. We use eight matrices. Each matrix is split into slices, which are then assigned to threads. Each thread does an equal amount of work. PastSet elements are used to exchange border entries of a threads slices with threads computing on neighboring slices. Bulk synchronous communication is used. First the entries used by the thread calculating on the slice above are written, then the entries used by the thread below. Finally, entries from the threads above and below are read.

The wind-tunnel had linear scalability when run on the 4W cluster, and the perturbation due to monitoring could not be measured. We observed that when 200 steps were executed, having 4 threads per CPU[8] was about 5% faster than

[3] http://www.python.org
[4] http://www.python.org/topics/tkinter/
[5] http://www.research.att.com/sw/tools/graphviz/
[6] http://www.ifi.uio.no/ hpl/Pmw.Blt/doc/
[7] http://www.nmr.mgh.harvard.edu/Neural_Systems_Group/gary/python.html
[8] Each thread did an equal amount of work

having 1 thread per CPU (the same problem size was used for both configurations). When the number of steps was increased to 500, they were equally fast.

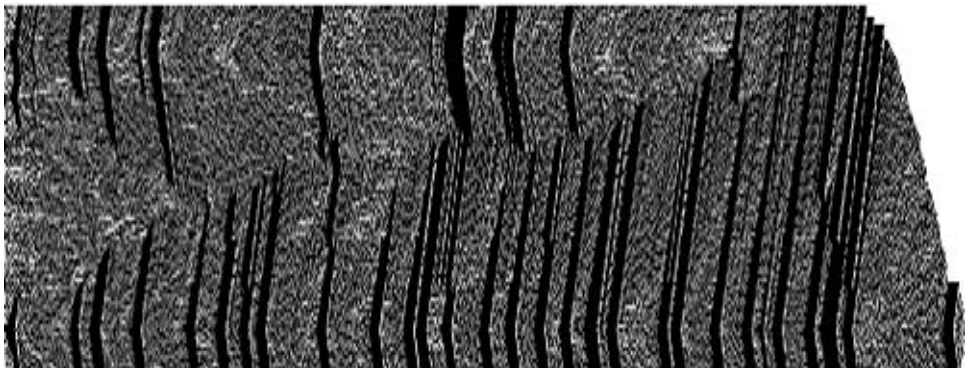

**Fig. 4.** Communication and computation times for the 'four threads per CPU' configuration (in total 96 threads).

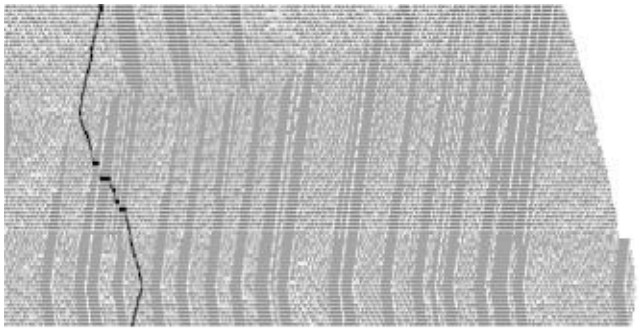

**Fig. 5.** Part of figure 4 with with 250th step highlighted.

In figure 4, there is one horizontal bar for each thread, that shows when it was using the communication system (black) and when it is was computing (light gray). On the horizontal-axis elapsed time is shown. We can see thick black stripes starting at the lower threads going upward. By using the step information displayed when pointing at a bar, we can see that most threads are one step ahead of the thread below (neighbors can only be one step apart).

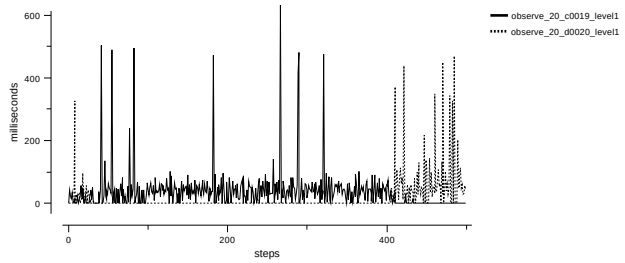


**Fig. 6.** Read operation times for worker thread 20, on elements from thread above (solid), and from the thread below (dotted).

By looking at the completion times (where the bars end) we can see a *wavefront*, where the threads higher up finishes earlier than the threads further down. By highlighting some steps we found that after 100 steps the threads had gotten roughly equally far, after 200 steps we could see a wavefront shape starting to emerge, and after 400 steps it was clearly visible. An emerging wavefront is shown in figure 5, where the 250th step is highlighted. In the background the same black stripes as seen in figure 4 are shown. The threads above the strip, are not affected by the wavefront.

Figure 6 shows how thread 20 changes, at around the 400th step, from spending most of its time waiting for data from the thread above (solid line), to waiting for data from the thread below (dotted line). By highlighting the 400th step in the communication-computation view, we can see that this is where the wavefront hits worker thread 20. Also the time per step is slightly increased after the 400th step.

To conclude, the four thread per CPU configuration slows down as the computing proceeds, due to the communication behavior of the wind-tunnel application.

### 5.2 Barrier Benchmark

In this section we analyze the behavior of hierarchical barriers, using a barrier benchmark with 32 threads. Each thread has a loop were it waits for some random time, before doing a barrier synchronization operation. One thread, Q, waits for a random time (0 to 8 seconds), the others waits for shorter random time (0 to 2 seconds). The hierarchical barrier has two levels where each node has a local barrier reporting to a single root barrier. The experiment was run on the 8W cluster.

By analyzing the virtual events, we found that on the node where Q was run it arrived latest in 62% of the barrier synchronizations. However, the signal from that local barrier to the global barrier arrived latest in 99% of the cases.

### 5.3 ELCIRC River Simulator

In this section we show how a load balance problem in a real scientific application, the ELCIRC river simulator [5], can be detected by analyzing the communication behavior of all threads.

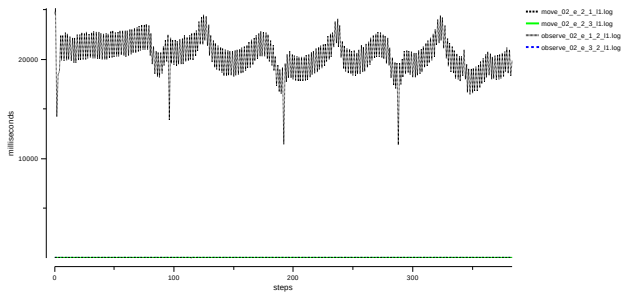


**Fig. 7.** ELCIRC: Write and read times for thread p02.

**Fig. 8.** ELCIRC: Computation (light gray) and communication (black) for each thread. Step 160 is highlighted.

In figure 8 communication (black) and computation (light gray) times are shown for four computation threads (we use a four thread version for clarity, similar results are found for a 32 thread version). It shows that the topmost thread (p01) spends almost no time communicating, while the others spend a significant amount of time communicating. In figure 7, the read and write times for thread p02 are shown. It shows that p02 spends most of its time waiting for data from p01. A read on data from p03 is almost as fast as a write (i.e. reads has almost no blocking time). Similar graphs can be shown for other threads. Thus all threads spends a significant amount of time waiting for data from p01, indicating a load balance problem. A search (similar to the barrier wait) can be used to find which thread a specific thread spends most time waiting for.

### 5.4 Global Clock

The pathmap can be used together with the virtual events to synchronize the Pentium time stamp counters (TSC).

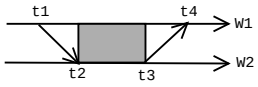


**Fig. 9.** Clock synchronization.

To synchronize the TSC for two nodes A and B, we use the following observation. If we have a path as show in figure 2, then we have a wrapper $W_1$ before the proxy on node A, and a wrapper $W_2$ on node B after the proxy. When a thread on A does an operation on the element four timestamps will be recorded as shown in figure 9: start time of the operation in wrapper $W_1$ ($t_1$), start time of the operation in wrapper $W_2$ ($t_2$), completion time of the operation in wrapper $W_2$ ($t_3$), and the completion time of the operation in wrapper $W_1$ ($t_4$). This information can be used to find an approximation of the offset between the TSC's on the two nodes. Our current implementation uses a simple scheme where B's offset relative to A is the average of several offsets calculated using: $o = t_2 - (t_1 + c)$, where $c = ((t_4 - t_1) - (t_3 - t_2))/2$. We assume that the communication time moving down, and up the path are equal.

To synchronize multiple clocks, we use the pathmap to create a graph that shows which nodes communicate with which other nodes. Then we use breadth first search to create a minimum spanning tree (MST)[9] starting from a node selected as the reference node (it should be be chosen such that the MST has a minimal height). For each pair of neighbors the difference between the TSC's is calculated (as described above), before the offsets are used to get an offset relative to the reference node. This offset is used to adjust the recorded timestamps.

### 5.5 Global Clock Accuracy

We can determine if the Pentium timestamp counters (TSC) on node A and B are synchronized with an offset less than the one-way latency, by asserting for each remote operation if $gt_1 > gt_2$ or $gt_3 > gt_4$, where $gt_i$ are the four timestamps in figure 9 adjusted to the global clock.

**Table 1.** Accuracy of computed clock offsets (one-way latency not added).

| Offset | Barrier wait | Wind-tunnel | Wind-tunnel (multi-cluster) |
|---|---|---|---|
| Correct | 50 % | 50 % | 47 % |
| Mean | 55 $\mu$s | 1933 $\mu$s | 20667 $\mu$s |
| Median | 13 $\mu$s | 1600 $\mu$s | 4182 $\mu$s |
| Stddev | 43 $\mu$s | 1275 $\mu$s | 61975 $\mu$s |

[9] The MST can be compared to a set of NTP [8] servers where the reference node is the primary server, and its children are secondary servers, and so on.

Performance data from the barrier benchmark and wind-tunnel applications were used to do a post-mortem analysis of the global clock synchronization accuracy. All timestamps belonging to a single step were adjusted to the global clock and checked as described above. The results are shown in table 1.

For all experiments 50% were within the tolerance offset. For the barrier benchmark the synchronization miss times are lower than for the wind-tunnel. The barrier benchmark is better suited since the communication consists of sending and receiving equal amount of data for each operation. Also all nodes are directly connected to the reference node. In the wind-tunnel application much data is sent one way, and no data sent the other way, and the spanning tree has only one node per level. When the wind-tunnel is run on all three clusters, the miss times are larger.

For comparison, using NTP under optimal conditions (e.g. 100Mb/s LAN access to a primary server) the offset can be bound to the order of 1 millisecond, but can be far worse [15].

## 6 Conclusions and Future Work

This paper describe the EventSpace monitoring approach that allows the low-level communication behavior of parallel applications to be monitored. By combining the collected data, high-level global views can be calculated.

In EventSpace, event collectors are integrated in the communication paths. When triggered by communication events, they create a virtual event that contains timestamps and other information about the event. The virtual events are then stored in a virtual events space from where they can be extracted by consumers using event scopes.

We have shown how the data in the virtual event space can be used for post-mortem analysis of a wind-tunnel application, a river simulator, global clock synchronization, and a hierarchical barrier benchmark. We have also shown how different communication behavior views are visualized using simple charts. We were able to detect anomalous communication behavior, detect load balance problems, analyze hierarchical barriers, synchronize the Pentium timestamp counters, and analyze the accuracy of the synchronization.

Further research is needed to find other useful views provided by the data in the event space. Presently, we are investigating how to use the data to analyze the performance of collective operations.

## Acknowledgments

We wish to thank Tore Larsen for discussions and being vital in getting the 4W and 8W clusters to Tromsø, Jonathan Walpole and Antonio Baptista for making ELCIRC available to us, and Brian Vinter for making the 2W cluster available.

## 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* (November 2001), 164–175.
2. Bjørndalen, J. M., Anshus, O., Vinter, B., and Larsen, T. Configurable collective communication in LAM-MPI. *Proceedings of Communicating Process Architectures 2002, Reading, UK* (September 2002).
3. Bongo, L. A., Anshus, O. J., and Bjørndalen, J. M. EventSpace - Exposing and observing communication behavior of parallel cluster applications. To appear in Proceedings of Euro-Par 2003.
4. Carriero, N., and Gelernter, D. Linda in context. *Commun. ACM 32*, 4 (April 1989), 444–458.
5. http://www.ccalmr.ogi.edu/CORIE/.
6. Kim, S. W., Ohly, P., Kuhn, R. H., and Mokhov, D. A performance tool for distributed virtual shared-memory systems. In *4th IASTED Int. Conf. Parallel and Distributed Computing and Systems* (2002), Acta Press.
7. Liao, C., Jiang, D., Iftode, L., Martonosi, M., and Clark, D. W. Monitoring shared virtual memory performance on a myrinet-based PC cluster. In *International Conference on Supercomputing* (1998), pp. 251–258.
8. Mills, D. L. Improved algorithms for synchronizing computer network clocks. *IEEE Transactions on Networks* (1995).
9. Moore, S., D.Cronk, London, K., and J.Dongarra. Review of performance analysis tools for MPI parallel programs. In *8th European PVM/MPI Users' Group Meeting, Lecture Notes in Computer Science 2131* (2001), Springer Verlag.
10. http://www.caida.org/tools/taxonomy/.
11. 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), pp. 172–179.
12. Sistare, S., Dorenkamp, E., Nevin, N., and Loh, E. MPI support in the Prism programming environment. In *13th ACM International Conference on Supercomputing* (1999).
13. 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).
14. Tierney, B., Johnston, W. E., Crowley, B., Hoo, G., Brooks, C., and Gunter, D. The NetLogger methodology for high performance distributed systems performance analysis. In *Proc. 7th IEEE Symp. On High Performance Distributed Computing* (1998), pp. 260–267.
15. Uijterwaal, H., and Kolkman, O. Internet delay measurments using test traffic: Design note. *Tech. Report RIPE-158, RIPE, NCC* (June 1997).
16. Vetter, J. S., and Yoo, A. An empirical performance evaluation of scalable scientific applications. In *Proceedings of the 2002 ACM/IEEE conference on Supercomputing* (November 2002).
17. Vinter, B. *PastSet a Structured Distributed Shared Memory System.* PhD thesis, University of Tromsø, 1999.