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

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

{larsab, otto, johnm}@cs.uit.no

Department of Computer Science, University of Tromsø

Abstract

For scientific application run on clusters, communication performance becomes increasingly important when the number of cluster nodes increase. To understand the communication behavior, we have developed EventSpace, a configurable data collecting, management and observation system for monitoring low-level synchronization and communication events. 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 collective operation. The views allowed us to detect anomalous communication behavior, detect load balance problems, find hotspots in a collective communication structure, 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 [15], most would benefit from improvements to MPI's collective operations [9], and half would benefit from improvements in point-to-point message overhead and reduced latency.

In earlier work [1] we improved the performance of collective operations up to a factor of two, by reconfiguring the communication structure, and by using better mappings of computation and data to the clusters. Point-to-point communication performance can also be tuned to improve performance, for example by changing protocols.

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 the applications communication behavior. Also needed are tools that can aid in analyzing

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

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.

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 [2] 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 [16] structured 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 communication behavior of 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 [8]. Generally such tools provide coarse grained analysis with focus on processor utilization [13]. Also, most tools does not monitor what happens inside the communication system. We expect EventSpace to supplement these tools, for instance to understand why collective operations or synchronization operations have poor performance, once other tools have identified it as a problem.

NetLogger [13] 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 [6]. Here monitoring code is embedded in programmable network interfaces to collect network-

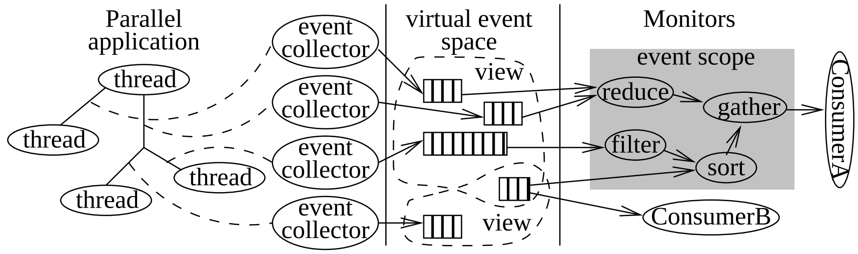

Figure 1: EventSpace overview.

level data. The data is tied to higher level software events. In [5] 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

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.

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 for communication and synchronization.

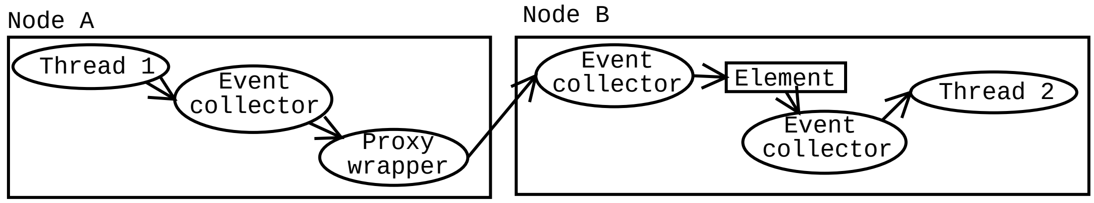

Figure 2: Two threads communicating using a PastSet element.

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 are read from and written to a element using blocking operations.

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

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.

In figure 2, the path from *thread 1* to the element consists of 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.

In the application source code only leaf wrappers are referenced, allowing threads to communicate in access and location transparent manner. This also allows communication to be mapped onto arbitrary cluster configurations simply by reconfiguring the path specifications.

Specifying and Instrumenting Communication Paths

Path specifications are generated by path generator 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 *pathmap*. Based on the pathmap, PATHS do the actual set up of the paths. The pathmap is also used by various tools, including tools for analysis and visualization.

An application is instrumented by inserting event collector wrappers to the communication paths (figure 3). This is done by scripts that manipulate the paths in a pathmap (application source code is not changed). Event collectors can be added anywhere on any path. However, the lower cost of using the lazy event processing approach, typically makes it feasible to add an event collector before and after every other wrapper.

The event collectors can produce large amounts of data. For a multi-cluster wind-tunnel (described in section 5) with 332 threads, we collected over 165 MB of data in five minutes.

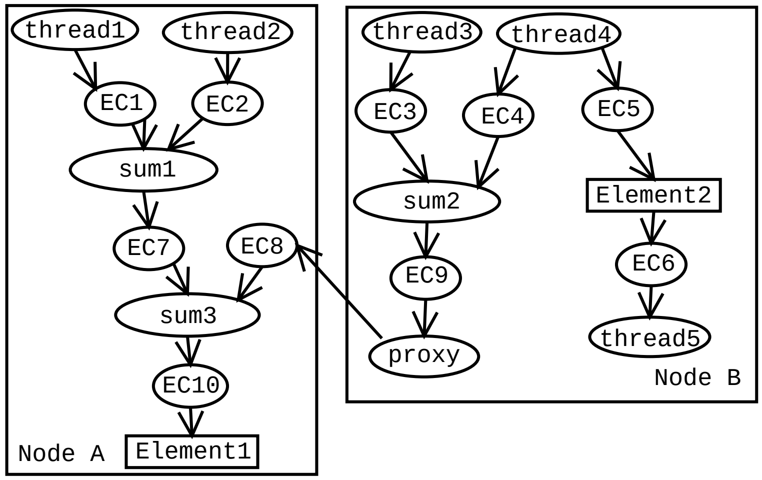

Figure 3: Instrumentation of a global sum tree: Paths are instrumented with event collector wrappers (EC1–EC10). The global sum is stored in element 1. Thread 5 does not participate in the global sum.

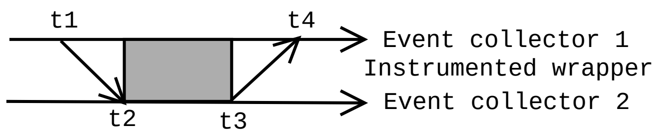

Figure 4: Timestamps recorded by two event collectors for an instrumented wrapper.

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* using a blocking PastSet operation. 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. 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). If the trace element is located on a local server, the write only involves a memory copy and some synchronization code[1].

As an operation passes down and up the path it can pass through several wrappers (figure 4). For each operation, each event collector record the start time of the operation on the down-pass, and the stop time on the up-pass. The timestamps are recorded using the high-resolution Pentium timestamp counter. For each operation, only one 36 byte tuple is created. It is written on the up pass. This tuple also includes the PastSet operation arguments.

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 during the lazy event processing.

We have measured the overhead of an event collector to be between 0.5 μs to 6.1 μs depending on the architecture it is run on [3]. This is comparable to systems such as Autopilot [11], NetLogger [13], and SHRIMP [6]. A write to a local PastSet element

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

may take a few microseconds, a remote write may take hundreds of microseconds, and a (blocking) read or a collective operation may take thousands of microseconds.

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

To establish a view, the pathmap is used to correlate data from a group of event collectors to higher-level abstractions. The pathmap gives a specification of the causality between virtual events along a given path, while the timestamps gives a temporal ordering of the communication events per node.

Below are some examples on how the event collectors in figure 3 can provide data for different views. Examples on how views are used in communication behavior analysis are given in section 5.

By subtracting the stop timestamp from the start timestamp for all top-level event collectors (EC1–EC6), we get a computation-communication view, that provides information about when, and for how long each thread use the communication system.

The order of the timestamps and the associated PastSet operation arguments recorded by EC5 and EC6, can be used to determine the order of reads and writes to *element 2*.

The wrapper latency of a wrapper can be calculated by $(t_4 - t_1) - (t_3 - t_2)$ (figure 4). While $t_2 - t_1$, and $t_4 - t_3$ gives the time spent in a wrapper on respectively the down and up pass.

We get an application load balance view by calculating how long each thread wait for the global sum to complete. This is done by subtracting the stop timestamp from the start timestamp from the virtual events provided by EC1–EC4.

For *sum 1*, we get the arrival order distribution by using the start timestamps collected by EC1 and EC2 to calculate how many times *thread 1* arrived before *thread 2*. The difference between the timestamps is the time the first arrival had to wait for the second. Similarly, the departure order distribution and departure wait time can be calculated using the stop timestamps in EC1 and EC2.

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

Event Scopes

An event scope is used to gather and combine the virtual events in a view. 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.

A library is used to get references to the trace elements associated with a view. New views can be created either by combining existing views (e.g. in a hierarchical manner) or by extending the library.

Since views often are hierarchical, the gather tree is built in a hierarchical manner. The simplest tree only gathers data, and all analysis is done in code written by the performance analyst. By inserting special analysis wrappers to the tree, such as sort or filter, part of the analysis can be done in the gather tree. Presently, creating and configuring the tree must be done by either writing or using an existing script. We intend to build a graphical tool for this task.

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].

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.

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.

The tools are implemented using Python[2] and Tkinter[3]. Graphviz[4] is used to draw the path specification graphs, Blt.Graph[5] is used to draw the line and bar charts, statistics are calculated using the Python stats module[6]. 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, 2 GB 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 100 Mbps local area network. Communication between Tromsø and Odense is the departments Internet backbone.

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

[2]http://www.python.org
[3]http://www.python.org/topics/tkinter/
[4]http://www.research.att.com/sw/tools/graphviz/
[5]http://www.ifi .uio.no/ hpl/Pmw.Blt/doc/
[6]http://www.nmr.mgh.harvard.edu/Neural_Systems_Group/gary/python.html

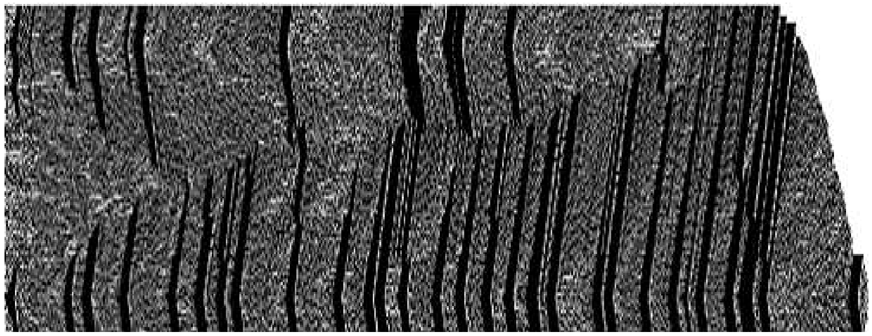

Figure 5: Communication and computation times for the ’four threads per CPU’ configuration (in total 96 threads).

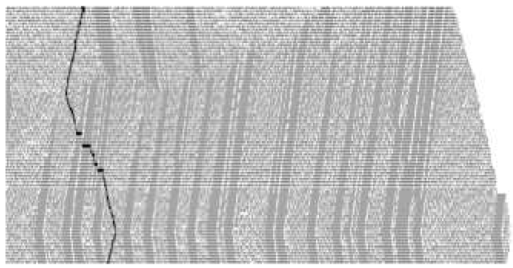

Figure 6: Part of figure 5 with with 250th step highlighted.

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 there was no perturbation due to monitoring. We observed that when 200 steps were executed, having 4 threads per CPU[7] was about 5% faster than 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.

In figure 5, 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) (i.e. a communication-computation view as described in section4). 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).

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 6, where the 250th step is highlighted. In the background the same black stripes as seen in figure 5 are shown. The threads above

[7]Each thread did an equal amount of work

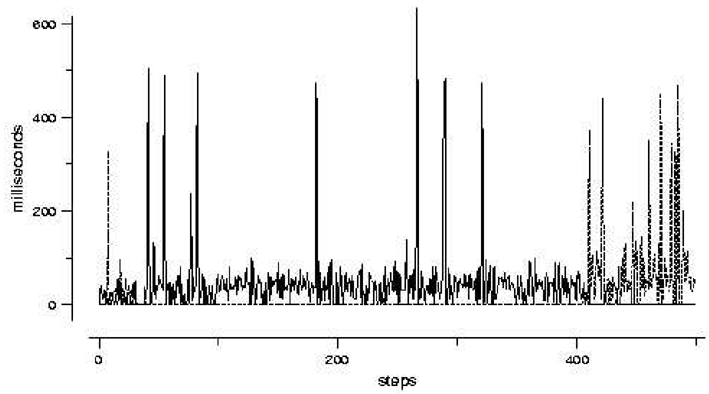

Figure 7: Read operation times for worker thread 20, on elements from thread above (solid), and from the thread below (dotted).

the strip, are not affected by the wavefront.

Figure 7 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.

Global Sum Benchmark

In this section we use the arrival order distribution (described in section 4) to find the slowest threads participating in a global sum collective operation run on the 4W and 8W cluster. The communication structure is organized as a binary tree (as in figure 3).

For the bottommost sum wrapper one of the participant always arrive first (in the 25000 operations examined). We exclude the faster branch, and follow the slower branch to find the next sum wrapper. For this wrapper, the faster participant arrive first in 67% of the operations. For the next sum wrapper on the slowest branch, there is an even arrival order distribution. Thus, the slowest threads are leafs on the subtree with root in the sum wrapper.

A breakdown of the cost of a global sum for the slowest threads, shows that inter-host communication dominates the time for a global sum. The breakdown is computed by summing the time spent in each wrapper, and then subtracting the arrival wait times for all sum wrappers.

ELCIRC River Simulator

In this section we show how a load balance problem in a real scientific application, the ELCIRC river simulator[8], can be detected by analyzing the communication behavior of all threads.

ELCIRC was run on the 8W cluster. Figure 8 shows communication (black) and computation (light gray) times for four computation threads (we use a four thread version for clarity, similar results are found for a 32 thread version). The topmost thread (p01)

[8] http://www.ccalmr.ogi.edu/CORIE/

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

spends almost no time communicating, while the others spend a significant amount of time communicating.

By examining the read and write order for all elements used by thread p02, we find that p02 wait longest for data from p01. Also, the wait time dominates the total communication time for p02. The other threads also spend most of their time blocked waiting for data from p01, indicating a load balance problem.

Global Clock

The pathmap can be used together with the virtual events to synchronize the Pentium time stamp counters (TSC) in a cluster. 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 an event collector before the proxy on node A, and an event collector 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 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 when 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.

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 4 adjusted to the global clock.

Performance data from the global sum 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 global sum benchmark the synchronization miss times are lower than for the wind-tunnel. The global sum benchmark is better suited since the communication consists of sending and receiving

[9]The MST can be compared to a hierarchy of NTP [7] servers where the reference node is the primary server, and its children are secondary servers, and so on.

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

Experiment	Correct	Mean	Median	Stdev
Global sum benchmark	50%	55 μs	13 μs	43 μs
Wind-tunnel	50%	1933 μs	1600 μs	1275 μs
Wind-tunnel multi-cluster	47%	20667 μs	4182 μs	61975 μs

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. 100 Mbps LAN access to a primary server) the offset can be bound to the order of 1 millisecond, but can be far worse [14].

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 collective operation. 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, detect hotspots in collective communication structure, 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, and a better approach for defining views.

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., VINTER, B., AND LARSEN, T. Configurable Collective Communication in LAM-MPI. *Proceedings of Communicating Process Architectures 2002, Reading, UK* (September 2002).

[2] 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. In *Norsk Informatikk Konferanse* (Nov. 2001), pp. 164–175.

[3] BONGO, L. A., ANSHUS, O., AND BJØRNDALEN, J. M. EventSpace - Exposing and observing communication behavior of parallel cluster applications. In *Euro-Par* (2003), vol. 2790 of *Lecture Notes in Computer Science*, Springer.

[4] CARRIERO, N., AND GELERNTER, D. Linda in Context. *Commun. ACM 32*, 4 (Apr. 1989), pp. 444–458.

[5] 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.

[6] 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.

[7] MILLS, D. L. Improved algorithms for synchronizing computer network clocks. *IEEE Transactions on Networks* (1995).

[8] MOORE, S., D.CRONK, LONDON, K., AND J.DONGARRA. Review of performance analysis tools for MPI parallel programs. In *8th European PVM/MPI Users' Group Meeting, Lecture Notes in Computer Science 2131* (2001), Springer Verlag.

[9] MPI: A Message-Passing Interface Standard. *Message Passing Interface Forum* (Mar. 1994).

[10] 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., 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.

[14] UIJTERWAAL, H., AND KOLKMAN, O. Internet delay measurments using test traffic: Design note. *Tech. Report RIPE-158, RIPE, NCC* (June 1997).

[15] 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), ACM/IEEE.

[16] VINTER, B. *PastSet a Structured Distributed Shared Memory System*. PhD thesis, Tromsø University, 1999.