Real time response to streaming data on Linux clusters

Beth Plale[1], George Turner[2], and Akshay Sharma[1]
[1] *Computer Science Department, Indiana University*
[2] *University Information Technology Services, Indiana University*

Computer Science Department Technical Report TR-569*
November, 2002

Abstract

Beowulf-style cluster computers which have traditionally served batch jobs via resource managers such as Portable Batch System (PBS) are now under pressure from the scientific community to support real time response to data stream applications. That is, applications supporting events streamed from remote sensors or instruments at high rates. Cluster management must be agile and timely in its response to resource demands prompted by the unpredictable occurrence of real-world events.

Streaming applications differ in fundamental ways from interactive applications. While solutions exist to support interactive applications, the solutions are not well suited to streaming applications. In this paper we experimentally evaluate two approaches to supporting streaming applications on Linux clusters. Both are coexistence schemes, that is, streaming applications share nodes with traditional batch jobs on nodes scheduled by a batch job manager. One approach works with the batch job manager (*i.e.,* PBS) to reallocate computational resources to meet the needs of real time data analysis. The second approach works outside of PBS utilizing Linux real time scheduling features. We also introduce a software architecture for streaming applications that we developed. Its purpose is a general reusable software structure for IU scientists requiring streaming solutions for acquisition of resources necessary for real time stream analysis.

*Supported in part by the National Science Foundation under Grant No. 0116050

1 Introduction

Beowulf-style clusters have traditionally used resource managers such as PBS Pro [11] to optimize the utilization of a cluster's resources, to ensure fairness among users, and to maximize cluster throughput. Today there is a growing need for these clusters to be responsive to non-deterministic event driven data streams occurring in applications we refer to as streaming applications [10, 8, 1, 2]. *Streaming applications* differ from event-driven (*i.e.*, interactive) applications in their high event generation rates, and near absence of human interaction. Event-driven applications respond to frequent user input, and are evaluated in terms of human response times. Streaming applications, on the other hand, respond to timestamped events flowing from sensors, scientific instruments, or to a lesser degree users, and are evaluated based on their responsiveness to remote complex, scientific events.

Administrators of large-scale production clusters have workable techniques to support interactive jobs. One site administrator we spoke with approaches the problem by partitioning off a small fraction of nodes that are dedicated to interactive use. A second we spoke to makes a set of nodes available to PBS for 12 hours out of the 24 hour day. The remaining 12 (daylight) hours are for interactive use. The latter approach has the advantage over the former in that by leveraging the known diurnal pattern of human usage [4], the cluster can provide higher batch production capacity during times when interactivity is likely to be low anyway.

Streaming applications interact with, and must be responsive to, the needs of a remote entity. This is no different from an interactive application. But whereas for an interactive application the remote entity is a human, for the streaming application the remote entity is a complex event such as a thunderstorm, or flood. As natural events have shown no propensity toward honoring human slumber times by occurring during waking hours, accommodating streaming applications using a strategy based on the diurnal pattern of human usage is ineffectual. For the small number of streaming applications we envision having to support, the first approach of dedicated nodes will likely result in unacceptably high levels of idle resources. Our work explores other approaches to managing streaming jobs in a batch scheduled production cluster.

We base our solution on the idea of coexistence as a means to support streaming applications in batch clusters. In a *coexistence* scheme, streaming jobs coexist with batch jobs on production cluster nodes that are schedulable for batch jobs. That is, the streaming jobs use the same set of nodes as the batch jobs. The approach offers the benefit that no nodes need be permanently or cyclically removed from the scheduling pool as is the case in the interactive schemes. A coexistence scheme, however, has potential drawbacks for both streaming jobs and batch jobs. Response latency is a key metric for evaluating performance of a streaming application. *Response latency* is the time between when a complex event occurs and when

a response is initiated at the remote site. Response latency will suffer if initiating a response requires lengthy setup time for amassing necessary resources. The second drawback is a potential loss of throughput for both the streaming applications and the batch job manager when both must coexist on cluster nodes.

This paper examines two approaches to supporting streaming applications on a Linux production-oriented batch cluster. Both are coexistence schemes, that is, they allow the streaming application to use nodes that are controlled by a resource manager such as PBS. Our first approach is to work within the PBS domain to allocate resources and provide job startup services in response to the detection of developing real-world phenomenon. The second approach explores a 'bullying' relationship with PBS in which the real time scheduling features of the Linux kernel are utilized to preempt resources already allocated by PBS. The first contributions of the paper is an experimental evaluation of response response latency. We are currently evaluating the long-term implications of coexistence on both streaming and batch jobs. The second contribution is a software architecture for streaming applications. Its purpose is a general reusable software structure for our scientists in need of streaming solutions.

2 Motivation

An atmospheric scientist at Indiana University (IU) studies atmosphere-surface exchange of trace chemicals such as CO2, water, and energy from a variety of ecosystems. These atmospheric flux measurements generate large streams of data; sensors operating at rates up to 10 Hz can generate data at rates of hundreds of MB/day/sensor. The remote sensors use wireless 802.11b to communicate to a central computer remotely located within the Morgan Monroe State Forest north of IU. The remote computer is connected to the IU campus network via a radio link having a bandwidth of roughly 4.3 Mbits/second. Under normal operation the data from the remote sensors are collected at the remote site and streamed to IU at a nominal rate of 1-3 Hz. At IU the event data are analyzed for particular meteorological conditions. An appropriate response to the detection of a meteorological condition is to increase the event rate to 10Hz and to enlist additional additional CPU cycles to handle high fidelity analysis for the duration of the meteorological condition.

This work is being carried out on the prototype AVIDD (Analyzing and Visualizing Instrument Driven Data) cluster called proto-AVIDD located at the Bloomington campus of Indiana University. The prototype cluster consists of five dual Pentium-IV Xeon nodes, two Itaniuim dual processor IA64 nodes, and a RAID-5 file system NFS served internally to the cluster. Proto-AVIDD runs RedHat 7.2 with cluster management tools provided by Oscar 1.2 [5]. Proto-AVIDD is a testbed cluster used to study the issues of

interactive and stream data analysis in a production-level Linux cluster.

The production AVIDD cluster is a geographically distributed data analysis cluster sited at three of the IU campuses: Bloomington (IU), Indianapolis (IUPUI), and Gary (IUN). Due to ongoing vendor negotiations, an explicit description of the AVIDD cluster is not possible at this time; however, it is expect to be in the range of $0.5 - 1$ TFOPS aggregate, 6 TBytes of distributed cluster storage and will leverage IU's existing HPSS based massive data storage system. Local intra-cluster networking will be provided by a Myrinet fabric with the inter-cluster networking utilizing Indiana's 1 Tb/s I-Light infrastructure. AVIDD is expected to be operational late Fall, 2002.

3 Approach

Streaming applications encompass a broad range of functionality including performance monitoring [7, 6], remote visualization [3], and stream filtering [9]. The streaming applications we target in this work differ from these other types of streaming applications because, unlike the others, they meet the following three criteria. That is, the

- Response is resource intense – requires a significant allocation of resources such as CPU cycles, network bandwidth, and I/O,
- Occurrence of complex conditions (e.g., tornado) is sporadic and unpredictable, and
- Duration of complex condition is limited.

It is these conditions that make a coexistence scheme both possible and necessary. For instance, a complex event that is predictable could be scheduled using the batch management system. A complex event that occurs frequently and/or lasts for days or weeks would consume too many resources, causing a significant negative impact on batch management system throughput.

The software architecture we developed to support streaming applications on an HPC cluster is shown in Figure 1. It consists of four components: a client at the remote location, shown in the lower left, that streams events from the remote sensors or instruments; a server resident on the cluster that looks for interesting conditions, an event cache to save incoming events, and worker tasks that are kicked off in response to detection of a complex, scientific event.

The usage scenario is as follows. Under normal, non-eventful conditions, the remote client pushes low resolution data to the server at a low frequency. The server, resident on a dedicated communications node within the AVIDD cluster, monitors the stream using an application-specific algorithm to determine when an interesting condition is developing. All events it receives are written to the event cache. When an interesting condition occurs, a meteorological event for instance, the event detection server responds with two actions. The first is to send a message to the remote client. The directive is

application specific, but generally involves commanding it take to reconfigure its sensor system for higher fidelity data acquisition. This is shown in the figure as Step 1. The second is to enlist additional resources within the cluster to process the higher fidelity streams. This is Step 2 in the figure. Once a worker task becomes active, it receives the data stream from the remote source. Parallelization of computation among the worker tasks and of the data stream is application dependent so is not discussed here. One could envision parallelization of the 2D grid by a spatial decomposition; streaming could be multicast.

The event cache exists as a file managed by NFS and resident on a storage server with fast, uniform connectivity to the cluster nodes. The event cache provides historical trace for the worker nodes should they need access to events that arrive before the worker is running. The file implementation achieves the real time sharing, though we are currently optimizing this part of the system for more efficient, synchronized file access.

We explore two variations of coexistence whereupon streaming applications and batch jobs can coexist on PBS scheduled nodes of the AVIDD cluster. The key metrics in evaluating different approaches are minimized latency, and acceptable levels of degradation for batch job throughput. This paper presents a quantitative evaluation of response latency for the coexistence approaches. The long running tests required to assess acceptable levels of degradation are part of our ongoing work. Our first coexistence scheme works within the PBS domain to allocate resources and provide job startup services in response to the detection of developing real-world phenomenon. The second approach explores a 'bullying' relationship with PBS in which the real time scheduling features of the Linux kernel are utilized to preempt resources already allocated by PBS. Both approaches will clearly have a strong negative impact on throughput of batch jobs if the worker tasks are long running. But recall that a defining characteristic of the streaming applications we support is the infrequency of interesting conditions, and the finite, limited duration over which the condition persists. Longer conditions, such as hurricanes or flood, would have to be dealt with as a special case. The approaches are discussed in detail in the following subsections.

3.1 Startup Within Context of PBS

Resource managers such as PBS have a coherent global view of a cluster's state. They also have the tools necessary to stop, start, and signal jobs running under their control. It would appear to be a natural choice to use a resource manager's preexisting infrastructure to implement cluster resource reallocation. This approach works as follows for the architecture shown in Figure 1; the event detection server runs outside of PBS control on a dedicated communications node of the AVIDD cluster. When a meteorological condition is detected, the server submits a job to PBS using either the Qsub

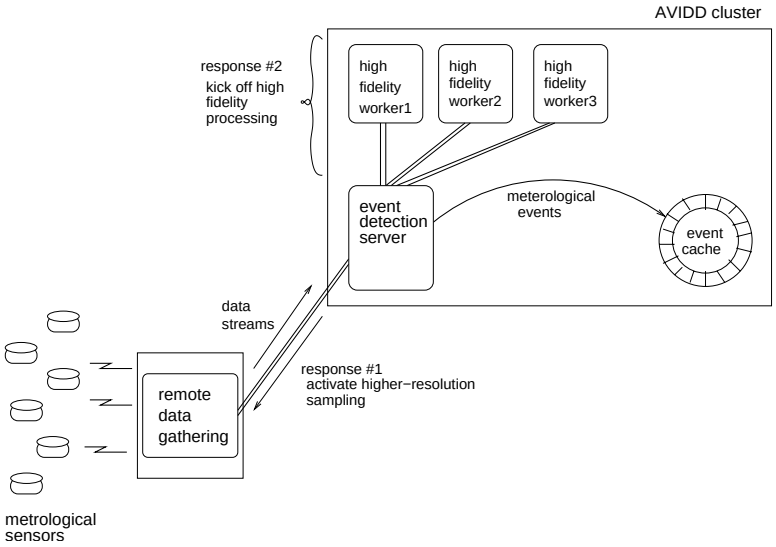

Figure 1: Data stream processing response to application event of interest.

command line call or the pbs_submit library call. Results for the former is given in Section 4.

The job description passed to PBS includes a count of the number of worker tasks to spawn. The job is submitted to a queue having a high priority (of 150). The workers retrieve connection information about the remote client from the startup script. PBS preemption is enabled so that when a worker is scheduled to a machine, it preempts existing jobs on the machine for the duration of the worker task.

3.2 Linux Real-time Scheduling Startup

The second coexistence scheme bypasses PBS, and on PBS scheduled nodes starts the worker process as real time tasks using Linux real time scheduling features. Scheduling the worker tasks as real time processes ensures that the kernel scheduler will schedule the tasks on CPUs as soon as they are computable. Resources already in use by an existing PBS batch job would only be consumed as needed. For example, memory previously used by the PBS job would be swapped out only as needed. This works as follows for our architecture: as in the case described above, the event detection server resides on a dedicated communications node. When a meteorological condition occurs, the server forks worker tasks using rsh or ssh (both are evaluated in Section 4), passing connection information about the remote client in the command line.

The worker thread issues a call to a resident root owned process that issues a *sched_setscheduler* command to elevate the priority of the validated caller and changes the scheduling policy to SCHED_FIFO. Linux's real time support is intended to serve tasks that should not be blocked by lower-priority processes; require short response time, and require a response time with minimum variance. The demands imposed upon the worker task of serving the incoming event stream while simultaneously executing computationally intense functions over the events make it a suitable candidate for real time support. Linux real time support is not intended for hard real time applications because the kernel is non preemptive. As in the PBS approach described above, the viability of the approach in a coexistence setting depends upon the short-lived behavior of the worker processes.

4 Experimental Evaluation

We undertook an experimental evaluation of response latency for the streaming application illustrated in Figure 1 under several conditions for both coexistence schemes. Response latency is the time from when a complex event occurs until a response is generated at the monitoring entity. [1] The first scheme, in which we work within the PBS context to allocate resources and provide job startup services, is referred to in this section as 'PBS'. The second, in which the real time scheduling features of the Linux kernel are utilized to preempt resources, is referred to as 'Linux real time'.

The environment in which we conducted the experiments is depicted in Figure 2. The event detection server and worker processes run on the five IA32 nodes of the proto-AVIDD cluster. Proto-AVIDD is located in a building on the Indiana University campus that houses campus-wide resources managed by UITS, University Information Technology Services. The event cache resides as a file managed by NFS and resident on a storage server. The experiment was conducted for three remote clients, one located at the Morgan Monroe State Forest (MMSF), connected to IU by a radio link (the bandwidth from the remote host to the Avidd cluster is roughly 4.3 Mbits/sec as measured by Iperf 1.6.2.) The other is located in the Computer Science Dept at IU. The latter's connection to the Avidd cluster is measured at roughly 94.1 Mbits/sec. The final has gigabit Ethernet WAN connectivity to the AVIDD communication node.

We measured startup latency under the two coexistence schemes discussed earlier, 'PBS' and 'Linux real time'. Within each scheme we tested variations. A list of tested combinations is as follows:

- PBS/Qsub: submitting via PBS using Qsub command line interface, no running jobs

[1] Since a complex event may only be detectable by a global view over distributed sources, we say a complex event occurs when it is first detected at the event detection server.

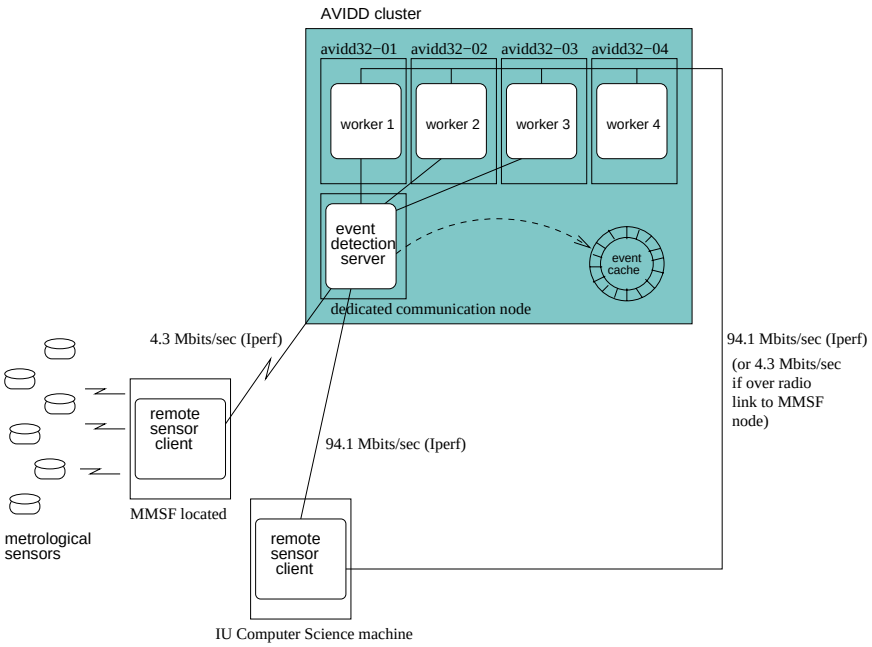

Figure 2: Experiment setup.

- PBS/Qsub/preempt: Qsub interface, running PBS jobs exist on worker processors that are preempted by PBS
- Linux real time/rsh: Linux real time scheduling and rsh
- Linux real time/ssh: Linux real time scheduling and ssh

In the two PBS cases, the event detection server works with PBS, and using a high priority PBS queue to start up the worker processes. The second two cases use either RSH or SSH for startup by the event detection server of the worker processes. In all four cases, the protocol used between the remote client and the Avidd cluster is SSH.

Startup latency is a broad measure. To better understand where the

Timestamp	*Definition*
t_0	condition first detected at server, determined from low rate data
t_1	server receives first 'high rate' event from remote source
t_2	worker 'up and ready', that is, ready to open socket connection back to remote client.
t_3	worker receives first 'high rate' data event.

Table 1: Time instances of interest.

Interval	*Definition*
$\{t_1 - t_0\}$	interval between when condition detected, and server receives first high rate event.
$\{t_2 - t_0\}$	startup time for set of workers
$\{t_3 - t_2\}$	worker feedback loop: interval between when worker ready to receive high rate event, and when first high rate event received

Table 2: Intervals of interest.

time is spent, we break latency down into component time intervals that are defined in Tables 1 and 2. The timing results for these individual intervals under the four cases described above appear in Tables 3 and 4.

Interval	*PBS/Qsub LAN (sec)*	*PBS/Qsub remote (MMSF) (sec)*
$\{t_1 - t_0\}$	0.008414 ± 0.001685	0.022217 ± 0.004000
$\{t_2 - t_0\}$	1.264183 ± 0.463738	1.265159 ± 0.490130
$\{t_3 - t_2\}$	0.003163 ± 0.006258	0.088769 ± 0.182738

Table 3: PBS cases: latency to 'turn up' stream $\{t_1 - t_0\}$, latency to worker startup $\{t_2 - t_0\}$, and worker startup delay $\{t_3 - t_2\}$.

Interval	*ssh MAN (CS dept) (sec)*	*rsh MAN (CS dept) (sec)*
$\{t_1 - t_0\}$	0.053161 ± 0.001464	0.056945 ± 0.000361
$\{t_2 - t_0\}$	0.757201 ± 0.017996	0.0573517 ± 0.000181
$\{t_3 - t_2\}$	0.012534 ± 0.006820	0.013329 ± 0.006804

Table 4: Linux real time cases: latency to 'turn up' stream $\{t_1 - t_0\}$, latency to worker startup $\{t_2 - t_0\}$, and worker startup delay $\{t_3 - t_2\}$.

Table 3 shows the PBS case. The case of 'PBS/Qsub LAN' has the remote client resident on a machine with a LAN gigabit connection to the Avidd cluster. 'PBS/Qsub MMSF' is the slow 4.3 Mbits/sec radio link. It can be seen that the low bandwidth penalizes the MMSF during the first and last intervals where the high fidelity stream must be turned on, and where the worker must delay in waiting for its first high fidelity event. Worker startup, the second interval, is the time it takes to start a worker through PBS channels on an unloaded node. The time is roughly 100 milliseconds longer if two PBS jobs exist on each of the four dual-processor nodes that had to be preempted. As can be seen by in comparing Tables 3 and 4, worker startup dominates total response latency. It can also be seen that PBS is costlier than either ssh or rsh. Table 4 shows results for ssh (on the left) and rsh (on the right) cases. In both cases, the remote client is hosted on the

remote machine having a relatively fast (94.1 Mbits/sec) MAN connection to the Avidd cluster.

As can be seen by comparing the LAN and MAN numbers, available bandwidth obviously matters. Startup time is noticeably less in the rsh/ssh case. The advantage of rsh over ssh during worker startup, $t_2 - t_0$, is due to the absence of key generation that ssh must undertake.

	High rate event first detected at server $\{t_1 - t_0\}$ *(sec)*	*Response initiated by worker* $\{t_3 - t_0\}$ *(sec)*
PBS LAN (local)	0.008414	1.267346
PBS remote (MMSF)	0.022217	1.353928
rsh MAN	0.056945	0.070681
ssh MAN	0.053161	0.769735

Table 5: Response latencies for condition detection, and initiation of response

The experiment is summarized in Table 5. The values in the first column of numbers give an indication of the amount of time the event server must wait after detecting a complex condition before it receives the first high rate event. If a severe thunderstorm is detected in the Morgan Monroe State Forest, it will take anywhere between 6 ms and 56 ms before the server receives the first high resolution event on which it can crunch. The second column of numbers give an indication of the amount of time that passes between when a complex condition is detected and when a worker can begin the computationally intense processing on the high fidelity stream. Our results show that a response can begin as early as 70 ms after detection when rsh is employed for remote start up of the worker processes.

Rsh/ssh are shown to be superior in reducing startup latency for data driven applications. But startup latency is only one of the relevant metrics by which performance is measured. In ongoing work below, we discuss additional metrics.

5 Conclusion

We have addressed the problem of supporting streaming applications on production clusters. We identified characteristics of the streaming applications that make traditional approaches for supporting interactive applications not applicable. We identified two schemes for supporting streaming applications based on coexistence, that is, the ability of streaming tasks and batch jobs to share the same set of nodes. We identified a set of restrictions on the type of streaming applications we support to make coexistence feasible.

We undertook the experimental evaluation of a key metric for streaming applications: response latency. That is, the time between when a complex

event occurs and when it is detected and responded to at the cluster. We evaluated response latency with multiple variations within each of the two coexistence schemes. Current work includes evaluating a second important metric, that is, the long running impact to both batch job throughput and streaming application performance. Finally we are exploring scalability, particularly in number of workers. We suspect response latency numbers are sensitive to scalability. For instance, the PBS/Qsub/preempt time will likely grow exponentially with number of nodes and number of existing jobs.

6 Acknowledgments

We would like to take the opportunity to thank Randy Bramley for his helpful comments on an earlier version of this paper.

References

[1] Shivnath Babu and Jennifer Widom. Continuous queries over data streams. In *International Conference on Management of Data (SIGMOD)*, 2001.

[2] Fabián E. Bustamante and Karsten Schwan. Active I/O streams for heterogeneous high performance computing. In *Parallel Computing (ParCo) 99*, Delft, The Netherlands, August 1999.

[3] Ian Foster, Joseph Insley, Gregor von Laszewski, Carl Kesselman, and Marcus Thiebaux. Distance visualization: Data exploration on the grid. *Computer*, 32(12):36–43, December 1999.

[4] Steven Gribble and Eric Brewer. System design issues for internet middleware services: deductions from a large client trace. In *USENIX Symposium on Internet Technologies and Systems*, 1997.

[5] Open Cluster Group. OSCAR: Open source cluster application resources. 2002.

[6] Weiming Gu, Greg Eisenhauer, Karsten Schwan, and Jeffrey Vetter. Falcon: On-line monitoring for steering parallel programs. *Concurrency: Practice and Experience*, 10(9):699–736, Aug. 1998.

[7] Dan Gunter, Brian Tierney, Brian Crowley, Keith Jackson, Jason Lee, and Martin Stoufer. Dynamic monitoring of high-performance distributed applications. In *Proc. 11th IEEE Intl. High Performance Distributed Computing (HPDC)*, Los Alamitos, CA, August 2002. IEEE Computer Society.

[8] Sam Madden and Michael J. Franklin. Fjording the stream: An architecture for queries over streaming sensor data. In *International Conference on Data Engineering ICDE*, 2002.

[9] Beth Plale. Leveraging run time knowledge about event rates to improve memory utilization in wide area data stream filtering. In *Proc. 11th IEEE Intl. High Performance Distributed Computing (HPDC)*, Los Alamitos, CA, August 2002. IEEE Computer Society.

[10] Beth Plale and Karsten Schwan. dQUOB: Managing large data flows using dynamic embedded queries. In *Proc. 9th IEEE Intl. High Performance Distributed Computing (HPDC)*, Los Alamitos, CA, August 2000. IEEE Computer Society.

[11] PBS Pro. Portable Batch System. http://www.pbspro.com/, 2002.