Scalable Analysis of Distributed Workflow Traces

Daniel K. Gunter, Brian L. Tierney and Stephen J. Bailey
Lawrence Berkeley National Laboratory
Berkeley, CA, USA

Abstract

Large-scale workflows *are becoming increasingly important in both the scientific research and business domains. Science and commerce have both experienced an explosion in the sheer amount of data that must be analyzed. An important tool for analyzing these huge data sets is a compute "cluster" of hundreds or thousands of machines. However, debugging and tuning clusters requires specialized tools. Current cluster performance tools are more oriented towards tightly coupled parallel applications. We describe how the NetLogger Toolkit methodology is more appropriate for this class of cluster computing, and describe our new automatic workflow anomaly detection component. We also describe how this methodology is being used in the Nearby Supernova Factory (SNfactory) project at Lawrence Berkeley National Laboratory.*

Keywords: cluster performance, troubleshooting, parallel debugging [1]

1 Introduction

An important class of parallel processing jobs on clusters today are *workflow*-based applications that process large amounts of data in parallel (e.g.: searching for supernovae or Higgs particles). In this context we define *workflow* as the processing steps required to analyze a unit of data. In these types of applications, each node of the compute cluster or farm is autonomous, with no communication or synchronization between nodes. Often this type of computing is I/O or database bound, not CPU bound. This means that the performance analysis requires system-wide analysis of competition for resources such as disk arrays and database tables, which is very different from traditional parallel processing analysis of CPU usage and explicitly synchronized communications.

In this paper, we classify cluster applications as "tightly coupled", "loosely coupled" and "uncoupled". Tightly coupled applications have a large amount of communication between nodes, using specialized interfaces such as the Message Passing Interface (MPI). Loosely coupled jobs have occasional synchronization points, but are largely independent. Uncoupled jobs have no communication or synchronization points. There are a number of performance analysis tools which focus on tightly coupled applications. In this paper we focus on uncoupled and loosely coupled applications.

For example, consider an astrophysics application where telescopic images are stored on a large RAID-based file server connected to each cluster node. An application that is searching for supernovae starts with a new image, does a database query to locate all related data from previous surveys of that part of the sky, then loads and processes those images. The performance bottleneck might be the database query, reading from the RAID disk, or the image processing. There are many possible ways to tune this application: should the application copy the images to local disk, or perform NFS reads directly from the RAID array? What is the optimal number of nodes to use? Does more effort need to be put into optimizing the database query?

The traditional focus of parallel performance analysis tools such as such as TAU [13], Paraver [11], FPMPI [4] or the Intel Trace Collector [8] has been on *profiling* and *tracing* the code sections of a single (parallel) application. Little support exists for combining the performance characteristics of multiple application components or distributed resources, such as databases and wide-area networks. But in a loosely coupled workflow, the bottleneck often lies in these interactions rather than in the core algorithms of the application code itself. Therefore, we must use experimental techniques to understand the performance of the entire system, including dynamic probes of all the major systems used by the workflow.

Performance tools have distinct classes of consumers. Application developers need to verify that their code is as efficient as possible, and to pipeline activities whenever possible. For example, to keep the pipeline full, one may be

[1] Published in the Proceedings of the 2005 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'05), Las Vegas, USA

able to load the next set of images while processing the current set. Performance analysis tools can be used to help determine if and where the pipeline can be optimized. These tools must be simple and intuitive; unless compelled by circumstances, most programmers are reluctant to invest much time and effort to learn new performance tools.

Cluster administrators need performance tools to determine when performance problems are due to overloaded resources, and when they are due to poorly designed applications. Most cluster administrators have experienced the problem of a user complaining about a performance problem such as access to the shared cluster disk is too slow, but monitoring of the disk server shows no reason for the performance problems experienced by the user. A "finger-pointing tool" is needed to determine conclusively if the problem is in the disk server, compute node, or the application code.

We have developed NetLogger methodology and toolkit to help both application developers and cluster administrators understand the performance of their respective systems [14]. NetLogger was originally developed to analyze distributed applications. However we have found it to be extremely useful to help understand loosely coupled cluster applications as well.

Previous papers on NetLogger have focused on overhead issues and dynamic activation [6], and use in a grid environment [7]. In this paper we focus on how NetLogger can be used in a cluster environment with loosely coupled parallel jobs. In particular we will show how the NetLogger "lifeline" provides an intuitive visualization for this class of application, and how the combination of application and system monitoring is critical for understanding the bottlenecks. We will also demonstrate the importance of automatic anomaly detection tools for scaling the analysis and visualization techniques to the large volume of workflow event traces generated by cluster applications.

2 Sample Cosmology Application

An example of an uncoupled cluster application is the Nearby Supernova Factory (SNfactory) project at Lawrence Berkeley National Laboratory (http://snfactory.lbl.gov/) [1], whose mission is to find and analyze nearby "Type Ia" supernovae. Type Ia supernovae are important celestial bodies because they are used as "standard candles" for gauging the expansion of the universe.

Supernovae are found by comparing recently acquired telescope images with older reference images. If there is a source of light in the new image that did not exist in the old image, it could be a supernova. Subtracting the new image from the reference image identifies new light sources. This process is quite delicate: aligning the images, matching the point-spread functions, and matching the photometry and bias all require precise calibration.

Around 25,000 new images are captured each day, and the goal is to complete all processing before the next day's images arrive. Image data is copied in "real time" from the Palomar observatory to a mass storage system at NERSC [10]. Then the image data is copied to a large shared disk array on a 344-node cluster called the Parallel Distributed Systems Facility (PDSF) [12]. Each image is 8 MB (uncompressed), and the processing of each image requires between 5 and 25 reference images, for a total disk space requirement of about .5 TB each day.

For each new image, a database query is performed to locate all related reference images. These reference images are then fetched from the mass storage system if they are not currently on the cluster file server. Then the images are processed and resulting new images are copied to the mass storage system, to be used again around a year later when the same part of the sky will be imaged again.

In this paper we describe how NetLogger is being used to analyze and troubleshoot SNfactory processing on the PDSF cluster.

3 NetLogger Overview

For over 10 years we have been developing a toolkit for instrumenting distributed applications called NetLogger [14]. Using NetLogger, distributed application components are modified to produce timestamped traces of "interesting" events at all the critical points of the distributed system. Events from each component are correlated, allowing one to characterize the performance of all aspects of the system and network in detail.

All tools in the NetLogger Toolkit share a common monitoring event format, and assume the existence of accurate and synchronized system clocks. The NetLogger Toolkit itself consists of four components: an API and library of functions to simplify the generation of application-level event logs, a service to collect and merge monitoring from multiple remote sources, a monitoring event archive system, and a tool for visualization and analysis of the log files. In order to instrument an application to produce event logs, the application developer inserts calls to the NetLogger API at all the critical points in the code, then links the application with the NetLogger library.

The NetLogger Toolkit also includes a data analysis component. One of the major contributions of NetLogger was the concept of linking a set of events representing a workflow together and representing them visually as a "lifeline", as shown in Figure 1. Visualizing event traces in this manner makes it easy to see where the most time is being spent, and when a workflow does not complete. The NetLogger Visualization tool, *nlv*, provides an interactive graphical representation of system-level and application-

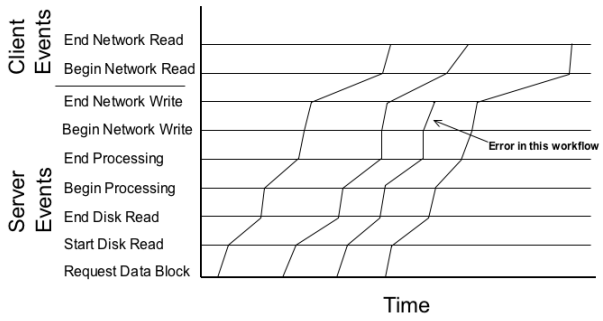

Figure 1. NetLogger Lifelines.

level events. NetLogger's ability to correlate detailed application instrumentation data with host and network monitoring data has proven to be a useful tuning and debugging technique for distributed application developers.

4 Related Work

As mentioned above, performance analysis tools such as TAU and Paraver are designed to analyze a tightly coupled parallel applications, and are not as relevant for loosely coupled workflows. On the other hand, a number of other projects that started out as mainly for tightly coupled applications have extended or adapted their tracing facilities to work in a more distributed setting. These include SvPablo [3], Prophesy [15], and Paradyn [9].

SvPablo is a graphical environment for instrumenting application source code and browsing dynamic performance data. SvPablo supports performance data capture, analysis, and visualization for application written is a variety of languages and executing on both sequential and parallel systems. SvPablo relies on a single user interface for performance instrumentation and visualization. During the execution of the instrumented code, the SvPablo library captures data and computes performance metrics based on the execution of each instrumented component. To capture dynamic performance data, calls to the SvPablo performance instrumentation library are generated. During execution of the instrumented code, the SvPablo library maintains statistics on the execution of each instrumented event on each processor and maps these statistics to constructs in the original source code.

The Prophesy data collection component performs automatic instrumentation of codes at the level of basic blocks, procedures, or functions. The default mode consists of instrumenting the entire code at the level of basic loops and procedures. A user can specify that the code be instrumented at a finer granularity than that of loops or identify the particular events to be instrumented. The resulting performance data is automatically placed in the performance database and is used by the data analysis component to produce an analytical performance model, at the granularity specified by the user.

Both SvPablo and Prophesy instrument source code automatically, this approach provides maximum portability of source code, but requires tools for each source language. These tools also provide a way to capture summary performance data (rather than event logs) from grid applications to be used for data analysis.

Paradyn is a performance measurement tool for parallel and distributed programs. It can provide precise performance data down to the procedure and statement level. Paradyn is primarily designed for identifying bottlenecks in parallel programs written using message passing or *pthreads*. Paradyn is based on a dynamic notion of performance instrumentation and measurement. Unmodified executable files are placed into execution and then performance instrumentation is inserted into the application program and modified during execution. Paradyn controls its instrumentation overhead by monitoring the cost of its data collection, limiting its instrumentation to a (user controllable) threshold. The instrumentation in Paradyn can be configured to accept new operating system, hardware, and application specific performance data. Paradyn is designed to gather and present performance data in terms of high-level parallel languages (such as data parallel Fortran) and can measure programs on massively parallel computers, workstation clusters, and heterogeneous combinations of these systems.

Our toolkit, NetLogger, is quite different from these tools. NetLogger does not perform automatic instrumentation like the above tools, and it has fewer pre-configured analysis components. Paradyn, SvPablo, and Prophesy all focus on timings for individual code sections and comparing these timings across nodes and runs. In contrast, NetLogger's concept of a "lifeline" is particularly well-suited to user-defined sequences of actions, like workflows. It should not be surprising, then, that NetLogger is the only one of these tools to provide workflow analysis and workflow anomaly detection. NetLogger is also the only of these tools that facilitates the collection of logged events from multiple locations.

5 Using NetLogger in a Cluster Environment

Until recently, NetLogger has mostly been used in a distributed computing environment where there are tens of systems involved in a particular distributed application. In this paper we explore how NetLogger can be used with hundreds of systems in a large cluster or farm.

5.1 Collecting NetLogger Events

Even though clusters have shared disk space, which typically uses the Network File System (NFS), logging in parallel to NFS from multiple independent jobs results in a profusion of log files, and is not practical or efficient. The alternative of logging to local disk and collecting the log files by hand is equally impractical. To solve this problem, NetLogger makes it easy log across the network (TCP or UDP) to a collection daemon, called *netlogd*. To reduce perturbation from fine-grained instrumentation, NetLogger events can also be written to local disk, and then forwarded by the a forwarding agent called *nlforward*.

NetLogger also includes a "debug level" mechanism that makes it easy to change what instrumentation or debugging data is being collected. The user communicates the new debug level by modifying a simple text file, whose location is determined by an environment variable. The NetLogger library automatically checks this file periodically and updates its debug level accordingly. For example, if a server process is running slower than normal, one can increase the debug level without having to restart the server.

5.2 Scalable Troubleshooting Tools

Once the basic framework for collecting logs is in place, the primary challenge for application instrumentation on large clusters is sifting through the volume of data that even a modest amount of instrumentation can generate. For example, if a 24 hour application run produces 50MB of application and host monitoring data per node, then while a 32-node cluster might be manageable (50 MB x 32 nodes = 1.6 GB), when scaled to a 512-node cluster the amount of data starts to become quite unwieldy (50 x 512 = 25.6 GB).

Our approach to this is twofold. First, we make it easier to selectively delete and analyze the raw data using a configurable tool called *nldemux* that can split and "roll over" the input data according to the contents of the event records themselves. The second part of the approach is an analysis tool called *nlfindmissing* that filters out everything except the anomalous workflows from the input data, greatly reducing data volume. These are both described in more detail below.

The results of *nlfindmissing* can then be viewed graphically, as shown in the result section below, or they can be passed to a report generation program. Reports can be periodically generated and emailed to the scientist monitoring the jobs.

5.3 NetLogger File Management

As described above, NetLogger applications send instrumentation data to *netlogd*, the NetLogger collection daemon. In normal use, *netlogd* then writes the events to a disk file.

For large clusters, the size of the output file of netlogd quickly becomes hard to manage. To address this we added the ability to filter incoming events into different files based on any NetLogger field. For example, if each instrumentation event contained a workflow ID, then a separate file per workflow could be generated. Another example might be to create a new file for each day, or hour of the day, based on the "DATE" field. A third example would be to generate a separate file per cluster node. This mechanism is very flexible, and any portion of any field can be used. For example, if one assigned hierarchal workflow IDs to the job, such as "A.B.C.D", then one could bin the instrumentation data into files where the workflow ID started with "A.B".

In addition, files or entire directories of files can be periodically "rolled over". This mechanism can be combined with the output splitting function described above. For example, system monitoring information can be split into a directory, further split by the monitored host name, and then the whole directory can be rolled over once every 24 hours.

Files and directories can even be rolled over remotely. If a special field, called `NL.ACTION`, is present in a NetLogger event record, then the *nldemux* program will either flush its open files to disk, close/rollover open files, or close and re-read its configuration file, depending on the value of the field. Only demultiplexed streams which "match" this event record, using the same criteria as for any other event record, are affected. This RPC-like feature has proven invaluable in isolating the results from a short test run without perturbing the overall system.

Storing all this monitoring data directly into a relational database is another possible solution to this problem. The NetLogger archiver, *netarchd*, provides this functionality. However this violates the "low entry barrier" principle for performance monitoring tools. In other words, in our experience developers are not likely to use a tool that requires installing and configuring (and maintaining) a database, so this alternate solution was found.

5.4 NetLogger Automatic Anomalous Workflow Detection Tool

Running a large number of workflows on a large cluster will generate far too much monitoring data to be able to use the standard NetLogger lifeline visualization techniques to spot anomalies. For even a small set of nodes, these plots can be very difficult to read, as shown in Figure 2.[2] (See section 6 for a detailed explanation of this figure).

To address this problem, we designed and developed a new NetLogger automatic anomaly detection tool, called

[2]This figure is much more readable in color. If possible, try to find a color printer or view this on your screen.

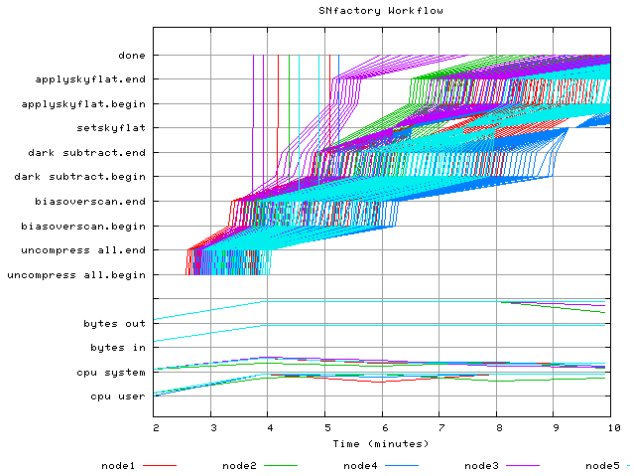

Figure 2. SNfactory Job Workflows on 5 Nodes

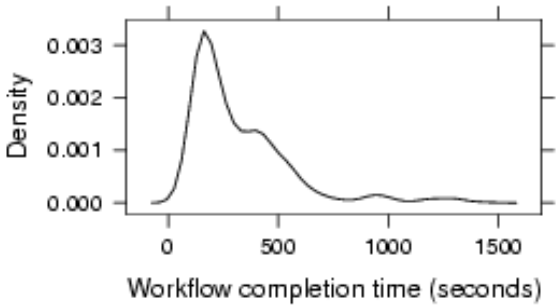

Figure 3. Distribution of SNFactory Lifeline Completion Times

nlfindmissing. The basic idea is to identify lifelines that are "missing" events. Using a configuration file, one defines the events that make up an important linear sequence within the workflow, as a lifeline. The tool then extracts lifelines from the raw data, and any time a workflow is missing an event, it outputs that lifeline.

To be scalable and intuitive, the tool needed to resolve two issues. The first is how, given an open-ended dataset that is too large to fit in memory, to determine when to give up waiting for a lifeline to complete. The second is how to output the right amount of context for the anomalous workflow with nearby "normal" workflows.

The first problem boils down to calculating a timeout for a given grouping of lifelines. Our approach was to approximate the density function of the lifeline latencies by maintaining a histogram with a relatively large, e.g. O(1000), bins. Then the timeout becomes a user-selected section of the tail of that histogram, e.g. the 99-th percentile. This works well, runs in a fixed memory footprint, is computationally cheap, and does not rely on any assumptions about the distribution of the data. Some additional parameters, such as a minimum and maximum timeout value, and how many lifelines to use as a "baseline" for dynamic calculations, make the method more robust to messy real-world data.

We also implemented alternative approach, which may be applicable to some data sets. Instead of a histogram, we calculate a running mean and standard deviation for completed lifelines, and dynamically update the timeout value to some user-specified multiple, 3 by default, of standard deviations above the mean. With proper algorithms, as described in [2], the computation itself is fast and accurate. But this approach has the major weakness that it assumes a statistically "normal" distribution for its input data, in this case lifeline (workflow) latencies. For the SNFactory data, and probably most real-world data, the distribution is not at all normal. For example, Figure 3 shows a sample distribution (density plot) of lifeline completion times for SNFactory workflows, which has little resemblance to the bell-curve of a normal distribution.

Once the timeout is available, it is straightforward to identify unfinished lifelines, and mark them as anomalies. The same basic method is used to identify extreme values for host monitoring data such as CPU load or available memory. The only difference is that a value contained in the event, instead of the time between events, is used as input to the computation. Also, timeouts are only concerned with defining upper limits, whereas system parameters may be interesting when values are high, low, or both.

Just examining the lifelines of anomalous workflows without some context of "normal" behaviour is not very useful, so *nlfindmissing* also outputs all other events that occur on that host during the lifeline, along with some lifelines on either side of the anomalous one.

If a high proportion of the workflows are anomalous, then providing context with normal lifelines is not very helpful, and could lead to no data reduction at all. Therefore *nlfindmissing* has on option to just flag the anomalous events.

Sample results from *nlfindmissing* are shown in the results section below.

6 Results

In this section we describe how NetLogger is being used by the Nearby Supernova Factory (SNfactory) project at Lawrence Berkeley National Laboratory for both fault detection and performance tuning.

SNfactory jobs are submitted to PDSF cluster at NERSC,

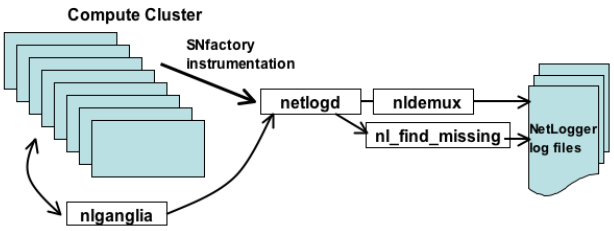

Figure 4. NetLogger Deployment for SNfactory.

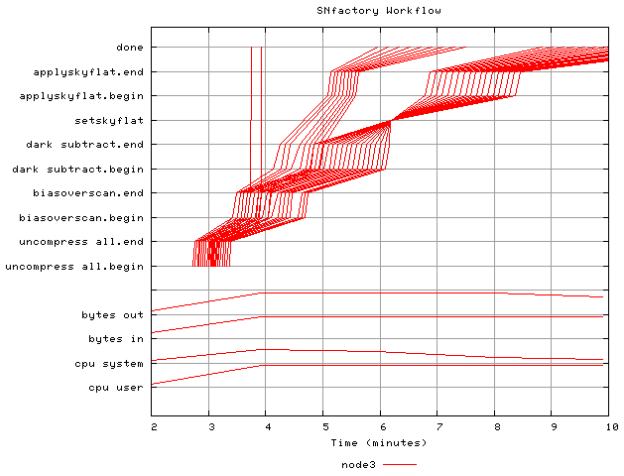

Figure 5. SNfactory Workflow shown with NetLogger Lifelines

and typically run on 40 nodes. Figure 4 shows how NetLogger is deployed for SNfactory analysis. SNfactory jobs were instrumented using NetLogger at each step in the workflow. On average, SNfactory jobs produce about one event per second on each node, for a total of roughly up to 1.1 million events per day. NetLogger instrumentation data is all sent to a single *netlogd* process, which is running on one of the cluster "front-end" nodes. Ganglia host monitoring [5] data is also collected using the NetLogger ganglia collector (*nlganglia*), which periodically polls the ganglia server (*gmetad*) for current host monitoring data from all cluster nodes, and forwards the data to *netlogd*. The *nldemux* tool is then used to group monitoring data into manageable pieces. The ganglia data is placed in its own directory, and data from each node is written to a separate file; the entire directory is rolled over once per day. The workflows are placed in files named for the observation date at the telescope, which is information carried in each event record.

Figure 5 shows a typical workflow for the SNfactory application on a single cluster node. CPU and network data from Ganglia is shown at the bottom. The SNfactory application processes a group of images together, starting with uncompressing the images, and then doing image calibration and subtraction. The next step is to generate a *skyflat* image, which is a calibration image that is formed from a median combine of several of other images. The *skyflat* is used to correct other images to adjust for the sky brightness on a given night, which can vary due to humidity, cloud cover, and so on. The skyflat calibration image is then applied to all images within the job.

This figure actually demonstrates a bug in the SNFactory processing that went undetected for several months before NetLogger analysis. Under some conditions it was determined erroneously that the skyflat calibration was not necessary. All lifelines except the two nearly vertical ones near the beginning should have converged at the *setskyflat* event.

The two nearly vertical lines near the beginning are, in fact, correct. Part of the processing is to generate a "dark image" that requires no further processing after *biasoverscan.end*, but is used in other calculations.

The first problem the SNFactory scientists asked us to solve was to figure out why some of their workflows where failing without any error messages as to the cause. Even when error messages were generated, the SNfactory application produced thousands of log files, and it was very difficult to locate the log messages related to failed workflows. NetLogger was very useful for easily characterizing where the failures were occurring so they would know where to focus debugging efforts. The new NetLogger anomaly detection tool was also applied to this problem.

One complication for SNFactory workflows was that the "dark" calibration images, as explained above, do not execute the latter portion of the workflow. Because the anomaly detection algorithm is based on whether a workflow executes its last event, this caused all the "darks" to get flagged as anomalies. This problem was avoided by the introduction of an artifical event that all workflows could write as they came to the end of their normal processing. This event appears with the keyword "done".

Figure 2 actually contains three workflows that did not complete, but this is not at all obvious in the graph. In comparison, Figure 6 shows clearly how the anomaly detection tool is able to highlight the incompleted lines. It also illustrates the large data reduction with a moderate (6%) rate of anomalies.

Note that highlighting the anomaly results in the NetLogger Visualization tool *nlv* was trivial because the anomaly detection tool adds an attribute to the NetLogger event indicating whether or not the record is part of an anomalous workflow. The *nlv* tool can color the lifeline based on this attribute.

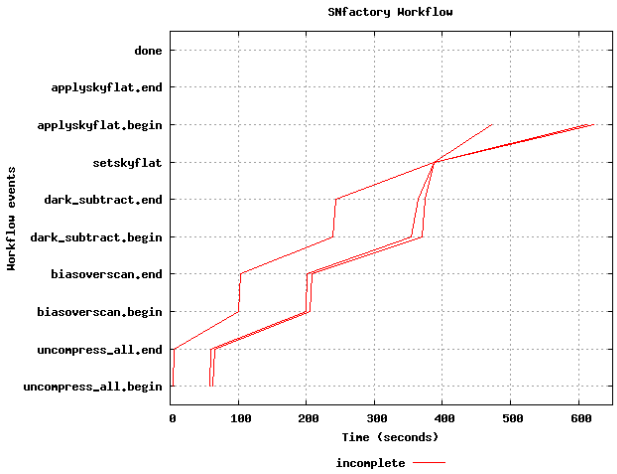

Figure 6. Anomaly Detection Results.

7 Conclusion

We have shown how the NetLogger "lifeline" methodology can be very useful for analyzing and troubleshoot workflow-based cluster applications. For example, the NetLogger Toolkit analysis of the Nearby Supernovae Factory workflow has already discovered some important bugs. The results from analyzing this application have also demonstrated the importance of the anomaly detection tool, both for reducing the volume of the data and for helping the visualization tools to scale to large numbers of workflows.

In future work, we would like to enhance the types of anomalies that can be automatically recognized, for example anomalies that combine the behavior of more than one lifeline. We would also like to explore database integration.

But in all our future enhancements, we will continue the basic philosophy of simplicity first, in order to keep the "entry barrier" for new applications as low as possible. When dealing with large shared resources such as clusters, it is important to be able to deploy a useful subset of the tools with no special privileges or resource-consuming processes. This philosophy is essential to NetLogger's usefulness in grid environments, but in our experience is also quite appropriate to large cluster environments.

Acknowledgment

This work was supported by the Director, Office of Science. Office of Advanced Scientific Computing Research. Mathematical, Information, and Computational Sciences Division under U.S. Department of Energy Contract No. DE-AC03-76SF00098. This is report no. LBNL-57060.

References

[1] G. Aldering and all. Overview of the nearby supernova factory. In *Proceedings of SPIE, Volume 4836, pp. 61-72*, 2002.

[2] T. F. Chan and J. G. Lewis. Computing Standard Deviations: Accuracy. *Communications of the ACM*, 22(9):526–531, 1979.

[3] L. deRose and D. Reed. Svpablo: A multi-language architecture-independent performance analysis system. In *International Conference on Parallel Processing. 1999. Wakamatsu, Japan*, 1999.

[4] fpmpi. `http://www-unix.mcs.anl.gov/fpmpi/WWW/`.

[5] Ganglia. `http://ganglia.sourceforge.net/`.

[6] D. Gunter, B. Tierney, K. Jackson, J. Lee, and M. Stoufer. Dynamic monitoring of high-performance distributed applications. In *11th IEEE Symposium on High Performance Distributed Computing. 2002.*, 2002.

[7] D. Gunter, B. Tierney, C. E. Tull, and V. Virmani. On-demand grid application tuning and debugging with the netlogger activation service. In *4th International Workshop on Grid Computing (Grid2003)*, 2003.

[8] Intel Trace Collector. `http://www.intel.com/software/products/cluster/tcollector/`.

[9] B. P. Miller, M. D. Callaghan, J. M. Cargille, J. K. Hollingsworth, R. B. Irvin, K. L. Karavanic, K. Kunchithapadam, and T. Newhall. The paradyn parallel performance measurement tool. *IEEE Computer*, 28(11):37–46, 1995.

[10] NERSC. `http://www.nersc.gov/`.

[11] Paraver. `http://www.cepba.upc.edu/paraver/overview.htm`.

[12] PDSF. `http://www.nersc.gov/nusers/resources/PDSF/`.

[13] S. Shende, A. D. Malony, J. Cuny, K. Lindlan, P. Beckman, and S. Karmesin. Portable profiling and tracing for parallel scientific applications using c++. In *Proceedings of SPDT'98: ACM SIGMETRICS Symposium on Parallel and Distributed Tools, pp. 134-145, Aug. 1998*, 1998.

[14] B. Tierney, W. Johnston, B. Crowley, G. Hoo, C. Brooks, and D. Gunter. The netlogger methodology for high performance distributed systems performance analysis. In *Proceedings of the Seventh IEEE International Symposium on High Performance Distributed Computing (HPDC 7)*, pages 260–267, 1998.

[15] X. Wu and V. Taylor. Design and development of prophesy performance database for distributed scientific applications. In *Proc. the 10th SIAM Conference on Parallel Processing for Scientific Computing. 2001. Virginia*, 2001.