Application Monitoring

Robert A. Ballance[*], Sandia National Laboratories
John T. Daly[†], Los Alamos National Laboratory
Sarah E. Michalak[‡], Los Alamos National Laboratory

SAND Number: 2008-2931C
May 6, 2008

Abstract

Application monitoring is required to determine the true performance of an application while it is running. Two case studies will be used to illustrate different aspects of the problem. First, we present a light-weight monitoring system for first-order determination of whether a job is making progress. However, full application monitoring requires deeper characterization and accurate measurement of the phases of an application's processing. Using the data gathered through application monitoring we can also derive information about the underlying system reliability. This paper will show how maximum likelihood estimates (MLEs) can be used to derive information about platform reliability based on application monitoring data.

1 What is Application Monitoring?

When applications are not making progress the system is not productive. Despite the truistic nature of this observation, the HPC community has yet to implement a standard application monitoring solution to answer the fundamental question: are applications making real progress? This is not the same as asking whether a system is being utilized or checking to see that the applications are "running." Those are system monitoring questions to be addressed by system tools, such as Cray's HSS software, and system data.

Whether or not an application is actually making progress — meaning that it is performing useful computational work — is an application monitoring question that has traditionally been left up to the users and code developers to address. However, the results of the W76 LEP calculations run by Daly on Red Storm, Purple, and BG/L in tandem (i.e., 40 million PE hours of computation completed in 16 weeks) at a cost of only 0.25 FTEs of user effort suggest that even a rudimentary application monitoring capability can have tremendous pay-offs for user productivity [3]. This connection arose after the fact, as Daly was discussing the ad-hoc scripts that he had developed for monitoring his own application. Daly's scripts have since been adapted locally at Sandia for one or two key applications, and have also informed the design of the *Jobmonitor* tool described below. Accurate application monitoring is also key to effective system administration, since system administrators will have better insights into the workload, and can react more quickly when problems arise.

In this paper we present an overview of work-in-progress at Sandia and Los Alamos. The text of the paper revolves around two case studies of how to approach application monitoring. Think of these case studies as design sketches of where this research is heading, rather than finished pieces. Early review and good feedback will help us to refine the work and hopefully to produce tools of general use to the Cray and HPC communities.

2 Sketch: A Simple Progress Monitor

Users appear to have many ways to check whether their application is, in fact, progressing. At first glimpse, the number of checks is as varied as the number of applications times the number of users.

[*]Sandia National Laboratories, `raballa@sandia.gov`.
[†]Los Alamos National Laboratories, `jtd@lanl.gov`.
[‡]Los Alamos National Laboratories, `michalak@lanl.gov`.

However, it seems that the actual test cases can be reduced to a handful. Most users tail a log file, looking for timestep markers, or they look to see if particular files are being updated. Knowing which files to check, and how to check them is daunting, but we can usefully restrict our attention to batch-scheduled jobs. In this case, it is feasible to ask the user to add information to their batch script to provide the necessary hints.

Our approach is guided by two assumptions, and one key observation:

1. Job progress monitoring is interesting only in the presence of long-running jobs. For short jobs, the key metric is queue throughput. Throughout the DOE Tri-Labs, capability users run jobs whose total duration is measured by days and weeks, with multiple intervening restarts. In such cases, it is important to know whether a job on the mesh is in fact doing any work. Section 3 addresses some of the issues concerning whether the work being accomplished is "useful."

   This assumption loosens the real-time constraints on monitoring, but may not affect the scalability issues. As we consider a centralized monitoring database spanning multiple platforms, we may need to deal with many jobs, only a few of which need progress checks at any given time.

2. Progress monitoring is simply a way to provide centralized and informed observations. The monitoring process is based on hints provided by users — which files to check, how frequently to check, what to check for, what constitutes a hang — and so the appropriate response to failure is to ignore or question the hint, rather than initiate a possibly hasty or improper corrective action.

   This assumption relieves the need for 100% correctness on each and every check, as well as the need for correctness on the part of the user.

A third observation quickly becomes apparent:

3. *There are as many or more ways for a job to cease progressing, or for a monitoring tool to fail, than there are simple, straightforward cases.*

   The complexities introduced by building a resilient monitoring application will be exposed in the following discussions.

With these observations in hand, we have begun to deploy a generic job monitoring facility built around periodic user-specified checks of user-specified log files.

The only other general issue was to create the appropriate structures for the monitoring, but this was reduced to a "simple matter of programming" (SMOP) by adopting an application structure based on managing a simple SQL database.

2.1 Overview and Approach

The *Jobmonitor* tool is a monitoring application developed on the LAMPs (Linux, Apache, Mysql, Perl) platform. Figure 1 illustrates the key components. The arrows in the diagram indicate primary information flows.

The "A" component of the LAMPs base is deemphasized; the Web server is currently used only to display the current job status. The choice of LAMPs was deliberate; this tool is being evaluated at Sandia (and perhaps the Tri-Labs) for roll-out across all of the HPC platforms. The portability and ubiquity of MySQL and Perl make this possible.

Adopting a database-based application schema also simplified many portions of the implementation, since the existence of clients and servers for MySQL could be presupposed. This reduced the "hard" implementation requirements to developing a resilient system that would be secure in a multi-client environment.

There is actually a suite of programs involved, most of which are database clients. Several tools, which do not appear in Figure 1 are for the administrators of the system to use when validating configurations or cleaning up the databases. Table 1 lists the top-level executables.

Table 1: Jobmonitor Executables

monitor	Registration script
job_status	CLI status check
jobmonitor.cg	WWW status page
monitor_cron.sh	cron wrapper
update_monitored_jobs	Checking script
clean_jobmon_db	DB management
print_config	Installation diagnostic
check_config	Installation diagnostic

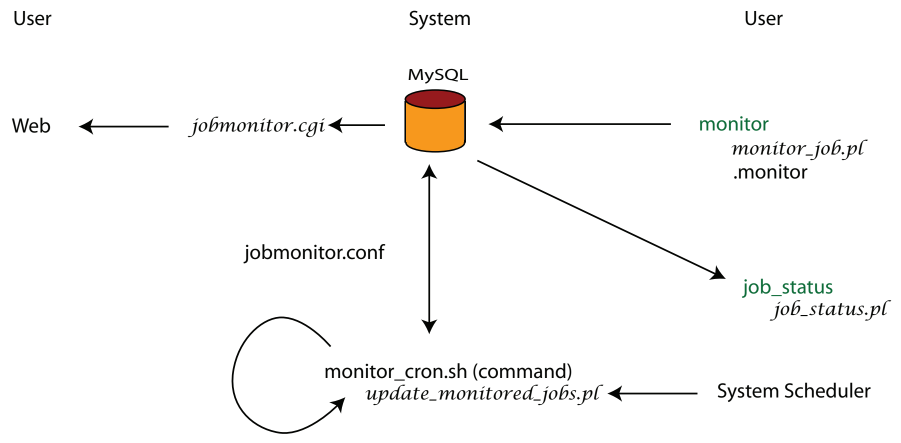


Figure 1: Jobmonitor Application Architecture

The system itself builds on widely available PERL packages, including DBI, DBD:mysql, DBI::Class, AppConfig, Mail::Mailer, Text::Template and others.

2.2 A Tour of Jobmonitor

By design, jobs are monitored only when a user requests monitoring. Once a job is registered with the *Jobmonitor* application, a slow polling loop (run from *cron*) updates the job status according to the user's hints along with the known state of the job. The time frame for checking is measured in minutes, not seconds. Each job is in a purported known state; but determining the state of the machine is a bit more involved than first anticipated.

For background, let us follow a single job through the system.

2.2.1 Registering the Job

A user "registers" a job by invoking the *monitor* command from within a batch job. On the XT systems at Sandia, this portion of the job script typically looks like

```
module load jobmonitor
monitor ...
```

The arguments to the *monitor* command specify the particular progress check to be made, the timing between checks, and whether the user is to be informed if the job hangs. So, for example, to monitor a log file for any file modifications, the user might specify

```
monitor --check=mtime --filename=run.log
```

In this case, the working directory of the job script will be used (by default) for the location of the output file. On each check, the file `run.log`'s modification time will be compared against the previous value, and if it has increased, the job will be marked as progressing.

The only required arguments to the monitor command are the check, any arguments required by the check, and the file to monitor. System-configured options provide all other default values. However, as users become more sophisticated, and jobs are more complicated, other choices become relevant. Suppose, that the job should only be checked once an hour after it gets through initialization, but that initialization takes roughly 45 minutes. Additions to the command line can express such requirements:

```
monitor --check=mtime --filename=run.log
             --lag=45m --freq=1h
```

Since the *monitor* registration script runs as the user, in a batch shell, all normal shell operations and environment variables can be used.

```
monitor --filename=$PBS_JOBID.log
     --check=mtime   --lag=45m --freq=1h
```

The full set of command line options is shown in Table 2.

Commonly reused arguments, such as the email notification settings, can also be specified in a *.monitor* file that can be located either in the current working directory of the batch script, or the user's home directory.

Table 2: Arguments to *monitor*

Name	**Purpose**
check	Test to run
filename	File to monitor
directory	Path to file
frequency	Time between checks
failafter	See below
regex	A regular expression
description	Job description
decreasing	Direction of change, when appropriate
lag	Minimum time before first check
email	Email user on hang?
email_addr	Email addresses to include

The current set of implemented checks appear in Table 3. The "grep" check requires a PERL-style regular expression that will isolate a single numeric value from a line in the file. The test finds the last occurrence of the regular expression, and then uses the value found to compare against the previous check. For example, if the output log regularly emits a line of the form

Table 3: Standard Checks

Name	**Checks**	**Arguments**
mtime	file modification time	
atime	file access time	
size	file access time	
grep	match a regex	regex

```
Timestep N at time XX:YY:ZZ
```

a valid set of arguments would be

```
 monitor --filename=run.log --check=grep
      --regex=".*Timestep (\d+)"
```

The implementation assumes that the file itself is growing, so the comparison scan can begin in the file at the point of the previous match.

Both "size" and "grep" allow the computed values to be decreasing.

As we were writing this paper, Daly suggested that the system should be able to count the number of files whose names match a regular expression. The modular design of the system will make this check straightforward to add.

2.2.2 Job States

Jobs move through a series of states as they are monitored. It is useful to divide the states into three groups: states that reflect only queuing status, states that represent current running status, and "holding" states that represent some failure condition in the monitoring. Once a job is moved to a holding state, no further checks on the job itself are made, but the system will transition the job to EXITED once it is no longer running. Table 4 lists the states; the leftmost column indicates (Q,R,H) for queuing, running, or holding states.

Table 4: Job States

Q	QUEUED	Job is in queue
Q	DEQUEUED	Job has left queue without a status check
Q	EXITED	Job was running, but is no longer running
R	INITIAL	Job Running, no data yet
R	OK	Normal progress found
R	STALLED	No progress lately
H	PROBABLY HUNG	
H	CHECK TIMEOUT	
H	FILE SYSTEM TIMEOUT	
H	CONFIGURATION ERROR	
H	STATUS UNAVAILABLE	
H	INVALID CHECK MODULE	

Jobmonitor has the ability to write the current queue status of jobs into its database. This feature came at the suggestion of an interested user who regularly manages flocks of jobs across several systems. Queue monitoring is regarded as experimental, since the functionality replicates that of the web-based queue management tools that are now widely deployed. The queue-related states only indicate whether a job is in the queue, or has left the queue. If the queue-tracking feature is disabled, then neither the QUEUED nor DEQUEUED states will be activated. It is possible for a job to start running, and to stop running before the monitoring system ever notices. In this case, there will be no record of it attempted run, except for the transition from QUEUED to DEQUEUED.

Once a job begins running, it enters the running states via the INITIAL state which indicates that the job has started running, but no status data has yet accrued. If everything goes well, the sequence of job-state transitions is given by the following regular expression:

$$\text{QUEUED}\cdot\text{INITIAL}\cdot \quad (1)$$
$$(\text{OK} \mid \text{STALLED})^* \cdot \text{EXITED} \quad (2)$$

More importantly, we are interested in the sequence

$$\text{QUEUED} \cdot \text{INITIAL}\cdot \quad (3)$$
$$(\text{OK} \mid \text{STALLED})^* \cdot \text{OK}^+ \cdot \quad (4)$$
$$\text{STALLED}^N \cdot \text{PROBABLY HUNG} \cdot \text{EXITED} \quad (5)$$

In the above regular expressions, "·" denotes sequence, | denotes choice, the superscript "*" denotes zero or more repetitions, "+" denote one or more repetitions, and the superscript "n" indicates precisely N repetition. Note that in both lines (2) and (4) we allow transitions to and from the STALLED state, since one or more checks might fail to indicate progress, even when the job is progressing. What the system is really looking for is a sequence of N checks, all of which result in "no progress". At that point, the job is moved into the PROBABLY HUNG holding state, and no further checks are made. The value of N is specified by the user via the `failafter` argument to *monitor*; the system default is 4.

Figure 2 shows a simplified version of the entire state machine. Explicit transitions into the holding states are not shown in the figure.

2.2.3 Updating status

Once a set of jobs are registered, a Unix *cron* job regularly checks the database for jobs that are ready to be updated. At each pass through the cron job, the list of jobs queued or running in the scheduler is compared against the contents of the job monitor database. If a job is running, is due to be checked, and is not in a holding state, then the status of the job is updated. As discussed, the holding states are present to describe outcomes in which further checking, aside from job exit, is unwarranted.

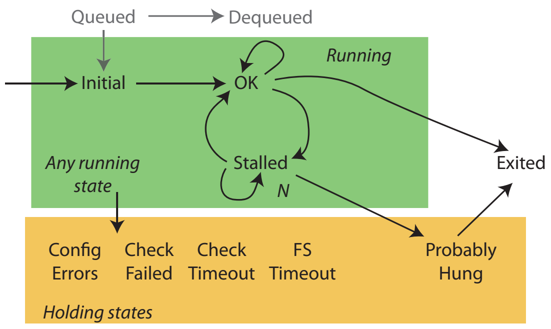


Figure 2: Simplified State Machine

Update checking proceeds in 2 stages and several steps. Stage 1 performs the basic job bookkeeping.

1. Contact the scheduler to list queued and running jobs. This interface is modularized, and has been tested with both PBS Pro and Moab.
2. Contact the database to collect running jobs.
3. Move any jobs that are no longer queued and that are not running into the DEQUEUED state.
4. Move any jobs that have been queued and are no longer running to the EXITED state. This includes jobs that are active, or are in one of the holding states.

   This point ends the basic bookkeeping; it runs quickly, but the duration is linear in both the number of jobs in the database, and the number of jobs in the queues.
5. Update the jobs that are ready to be updated.
6. Clean up and complete.

Of course, many things can go wrong:

- The user could provide the wrong arguments, such as misspelling the log file name, or an invalid check module.
- The user could provide improper hints — such as checking every 10 minutes on a job that only timesteps every 2 hours.
- The actual checking operations could time out, either due to load or due to a file system outage.

- Checking could run so long that another cron job starts, creating race conditions on the states of jobs

- A file system could be flaky, causing checks to timeout or fail.

Without delving further into details, we will leave you with the thought that these conditions, and others, have been characterized and handled. Two examples might help: the *cron* job itself has a deadline for completion that is set as part of system configuration. For example, if it runs every 10 minutes, then it might have a deadline of 8 minutes. Each check operation also has a timeout-value that sets the maximum time allotted to a check. Different timeouts can be defined for each class of checks, so that simple date checks can have faster thresholds than a check that reads the file. The update task, then, uses the deadlines and the thresholds to perform as many checks as possible. If it doesn't get around to a job, the next check will pick it up. Since we don't require a hard timing constraint for the checks, this implementation is robust.

What happens if a file system dies? In this case, all of the checks for that file system will probably fail, even though other jobs might be progressing. The *Jobmonitor* tool has built-in a set of configurable "short-circuit" tests that allows a site to specify the actions to be taken when individual tests time out. This makes it possible to specify a rule like "If any test on the `/scratch1` file system times-out, quit running tests on `/scratch1`, and treat them as File System Timeout."

2.3 Checking the status

At any time, the user can check the current (and historical) state of the job. Two checking mechanisms are provided: a Web-based interface and a command-line tool. By design, users can access only the status of their own jobs, while "administrators" can see all jobs. The term "administrator" is quoted since there is no assumption that users with administrative rights on a platform have the same rights in *Jobmonitor*; the configuration file for *Jobmonitor* includes the list of user-ids granted the ability to see all jobs. As a design extension, we are considering ways to enable groups of users to view jobs belonging to any member within the group.

2.4 Alerts

What happens when a job enters the Probably Hung state? Right now, the only actions available are to log the transition to a file, or to email interested parties such as system administrators or the owner of the job. Other alerts can easily be added to the system, and the *monitor* command itself allows the user to specify additional email addresses. In the long run, we'd like to develop dashboard-style agents that could run on a user's desktop and show the status of her jobs.

2.5 Privileges & Permissions

Finally, a word about permissions. As noted, the system is written primarily in Perl. The JobMonitor database in SQL maintains two sets of permissions: read-only rights for users so that job status can be queried, and higher privileges for processes that can update the database fields. Credentials for the database access reside in configuration files. These credentials are orthogonal to the status-viewing privileges implemented by the *job_status* and *jobmonitor.cgi* commands.

However, two activities require some significant revision of Unix permissions: 1. job registration, and 2. status checking. On job registration, the *monitor* command has to be run by the user, but the database rights should allow update access. In this case, the monitor command is split into two portions: the user-level script that can be run by any user, and a lower-level script that can be run only by the `jobmonitor` user-id, and that has access to the administrative tools.

At job registration, the effective user and group of the process that is running the registration command is collected as an attribute of the job. On job status update, the actual checks that examine a user's files are run by a setuid process that immediately lowers its permissions to that of the effective user and group that was gathered at registration.

All this means that there are three sets of rights to administer: the rights for the command (normally owned by `jobmonitor.jobmonitor`), the database privileges on MySQL, and the viewing privileges for displaying job information. Administrative tools have been written to assist in the process of setting and verifying the configurations.

2.6 Development Status

The system is now installed and in beta-test at Sandia. Development is largely complete, but the roll-out to users has been delayed due to higher-priority demands on the support staff. We are also considering an Open-Source licensing and distribution option.

3 Using Application Monitoring Data to Measure Useful Work

In this section we will look at ways to use data available from application monitoring to quantify the impact of system reliability on user productivity as measured by application throughput. Application throughput is defined as the amount of computational work completed on the system in a given amount of wall clock time and entails components of performance and reliability (i.e., time lost to defensive I/O and rework for a calculation that has been interrupted and must be restarted). The contribution of system reliability to application throughput may be hard to measure at the system level because it is sometimes difficult to discern the impact of a system event on the running jobs. However, the application knows when it is interrupted or unable to make progress, so application monitoring may be used to understand the impact of system reliability on the ability of the application to make progress.

A useful metric for measuring application throughput is "runtime efficiency" [5] which is the ratio of the amount of time an application is doing productive work to total runtime, which includes defensive I/O, restart, and rework. Using the model derived by Daly [7] and validated on large application runs on Red Storm [6], it has been shown that the application runtime efficiency can be accurately characterized by the following three quantities:

1. δ – the time for an application to create a checkpoint
2. R – the time for an application to restart after an interrupt
3. $1/\lambda$ – the expected time to application interrupt

How application monitoring can be used to help minimize the restart time, R, has already been examined [10]. In this section, we consider how data collected from application monitoring combined with a simple system model based on a reliability block diagram [7] may be used to better estimate the application interrupt rate, which is a measure of the reliability of the system.

This means that in addition to the value added by application monitoring to the user for keeping applications running, it also provides a light weight and relatively easy-to-implement alternative to system monitoring for gathering information about the reliability of a system. Although application monitoring data is not intended to replace system monitoring data, we believe that it can complement that data in interesting and insightful ways by providing the application's perspective on system reliability.

One way to think of system reliability from the application's perspective is by considering that every job running on a system has a probability of being interrupted and exiting with a fatal error that is, at the very least, dependent upon the number of nodes allocated to that job and the amount of time that it runs. If we know the application interrupt rate then we can predict the amount of time that we expect the application to run before encountering an interrupt [9]. Conversely, if we know the amount of time that the application runs between interrupts and the number of nodes it is using then we can estimate the application interrupt rate, which in turn gives us an estimate of system reliability from the application's perspective.

One may estimate system reliability as follows. Assume that for the jth application run on a system we have the following information available to us from application monitoring:

1. k_j – number of nodes allocated to the application
2. Δt_j – time that the application spent running
3. m_j – number of interrupts that occurred during the run

If we further assume that the inter-arrival times for interrupts are exponentially distributed, then interrupts arrive according to a Poisson process, which has been demonstrated to be a not unreasonable assumption [7]. The Poisson probability mass function (PMF) associated with the random variable M describing the number of interrupts that occur during time interval Δt, given a k-node application interrupt rate of λ_k interrupts per unit time, is given by:

$$P(M = m) = \frac{e^{-\lambda_k \Delta t}(\lambda_k \Delta t)^m}{m!} \quad .$$

If we assume that the jobs running on the system experience interrupts independently, then the likelihood function associated with any particular sequence of jobs $j = 1, ..., n$ is the product of the PMFs associated with the n jobs:

$$L(\lambda_{\{i|i\in\{k_j\}\}}) = \prod_{j=1}^{n} \frac{e^{-\lambda_{k_j}\Delta t_j}(\lambda_{k_j}\Delta t_j)^{m_j}}{m_j!} \ .$$

The maximum likelihood estimates (MLEs) of the k-node application interrupt rates that are estimable are obtained by maximizing the above function with respect to the $\lambda_i's$. It is equivalent to maximize the log of the likelihood function or the log-likelihood:

$$\ell(\lambda_{\{i|i\in\{k_j\}\}}) = \sum_{j=1}^{n} \left(m_j \ln(\lambda_{k_j}\Delta t_j) - \ln(m_j!) - \lambda_{k_j}\Delta t_j\right)$$

with respect to the $\lambda_i's$. Since the logarithm transforms the product into a summation, this simplifies the numerical procedure for large numbers of jobs involving small individual failure rates by reducing the amount of required numerical precision. Using the results of [9], we parameterize the k-node application interrupt rate in terms of M_1, the application mean time to fatal error (MTTFE) of a single node job, and M_N, the MTTFE for a full system job:

$$\lambda_k(M_1, M_N) = M_1^{-1}\frac{N-k}{N-1} + M_N^{-1}\frac{k-1}{N-1} \ .$$

The MLEs of M_1 and M_N are computed by setting the partial derivatives of the log-likelihood function with respect to M_1 and M_N equal to zero and solving the resulting equation:

$$\frac{\partial\Lambda(M_1, M_N)}{\partial M_1} = \frac{\partial\Lambda(M_1, M_N)}{\partial M_N} = 0 \ .$$

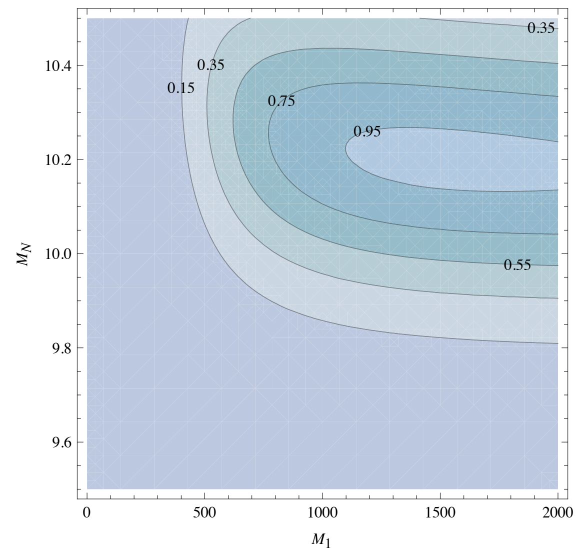
Although we could have calculated the MLEs of the k-node application interrupt rates that are estimable, by using the preceding parameterization we used an approximation based on two parameters, M_1 and M_N, that allows us to estimate the failure rate for any value of k, including values of k for which we have no application monitoring data. Thus, we do not have to measure the failure rate of specific job sizes to characterize the system, but instead we can use the job sizes available as part of the normal system workload to estimate the failure rates for job sizes that may not have been run as part of that workload. Figure 3 provides the likelihood as a function of M_1 and M_n for simulated job event data. To get this simulated data, $M_1 = 1000$ and $M_n = 10$ were used to calculate values of λ_k. The number of nodes for each simulated job was picked from a uniform distribution between 1 and 1024, and the job duration was picked from a normal distribution with mean 500 and standard deviation 200 that was truncated to be greater than 0.1. The number of interrupts each simulated job experienced was then generated from the corresponding Poisson distribution.

The failure rate of the application must be estimated in order to compute runtime efficiency, which is in turn a contributor to the application throughput for the system [7]. We have outlined a method for estimating application failure rates based on data that has been gathered from application monitoring. Application monitoring data can provide insight into other metrics such as performance, utilization and availability that are also required to quantify throughput.

Application performance could be approximated by measuring the frequency or amount of output generated, for instance. Machine learning techniques [2] might be employed in conjunction with such feedback to determine when applications are running abnormally fast or slow. Utilization is relatively simple to extract from the given application data assuming that one knows the size of the system, in terms of number of compute nodes, and the time period over which the data were collected. Availability is more subtle. We know that system availability is bounded by the utilization and 100%. To narrow it down further, one must determine what fraction of the unutilized time results from downtime, as opposed to shortage of work, using the application data collected by application monitoring. These are subjects for future research.

4 The Application Monitoring Project at DOE

In 2008, the DOE TriPOD software effort chartered a working group to review Application Monitoring issues, and to develop the requirements for a common monitoring system [8]. The largest hurdles to implementing application monitoring for production computing across the tri-lab may not be technical but logistical. The level of inter-lab and intra-lab coordination required among the user, developer, and HPC communities makes TriPOD is an ideal vehicle



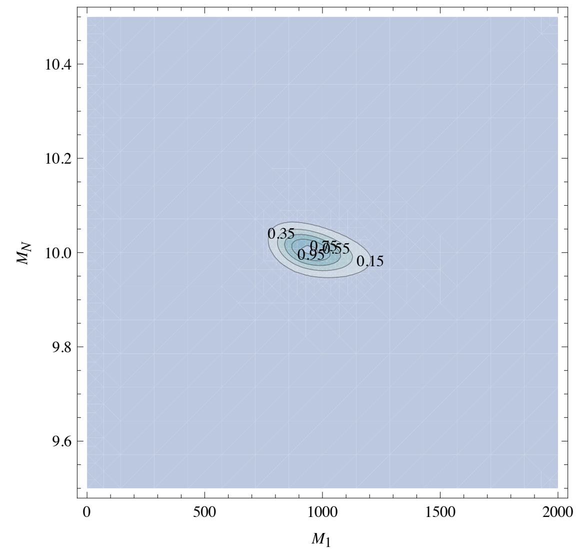


Figure 3: Two examples using simulated job data to estimate application MTTFEs for a single-node application, M_1, and for an application that uses the full system, M_N. Contours represent the scaled likelihood function with 1.0 being the maximum. The figure on top shows the likelihood function for 100 jobs run on a 1024 node platform, while the figure on the bottom presents the likelihood function for 5000 jobs run on the same platform. This demonstrates how additional job data can be used to refine the estimate of the application MTTFE.

for implementing an extensible application monitoring framework, tools, and interfaces to the applications that would complement system monitoring efforts already in progress at each of the labs. We emphatically do not wish to create a set of tools that are of no value for production work. To avoid this pitfall, we are forming a team of users and developers, each one of whom is well respected in their community, and with whom we will collaborate closely as we design and implement a suite of tools with hooks and interfaces that conform to the requirements of the tri-lab developer and user communities.

Building on application monitoring prototypes by and the experiences of Daly [4] and Ballance [1], we will work with tri-lab users and developers from B-Division (LLNL), the Crestone project (LANL), and the Sierra project (SNL) to design and implement system tools that will facilitate automated monitoring of production applications. We will be focused on answering basic questions about a user job, such as the following:

- Is the job making progress?
- At what rate is it making progress?
- How frequently is it interrupted?
- What are the causes and symptoms of the interrupts?
- Should the system intervene (e.g., to kill or restart the job)?
- Should the system operators or the user be notified?
- How much time and storage is spent preparing for restarts?

These questions need to be answered in terms of the implementation challenges posed and the potential productivity impacts to both parties, i.e. users/developers and HPC. We propose to develop an application monitoring strategy and a list of implementation priorities. Further, we expect to design, code, test, and implement a basic set of monitoring tools, along with their system and application interfaces. The anticipated result will be a functioning, extensible tool that can serve as a framework for future application monitoring functionality.

This work addresses several areas of particular concerns:

Application Performance Application monitoring and application performance are two sides of the same coin called application throughput. Daly [7] and others [11] have suggested that as many as two-thirds of the system errors that prevent application progress are not currently tracked by system monitoring. An effective implementation of application monitoring could conceivably provide data that might be used to improve end-to-end application throughput by a factor of three or more. **You might want to justify the factor of 3 that is used here.**

Standardization Both the prototype application monitoring tools developed by Daly [4] and Ballance [1] employ common user interfaces across all of the compute platforms at each of the labs. We intend to maintain this same level of standardization in the design and implementation of any production level tools that we develop. Part of the work being proposed includes leveraging these two developments to create a common information infrastructure.

Shared Environment By creating a standard user interface for application monitoring we are creating a monitoring environment that can be closely coupled to the system software stack at the level of the job scheduler and resource manager. Ballance[1] has developed a beta-level prototype that uses a shared relational database to track application progress checks and to distribute application status via command line, email, and web interfaces. Other distribution tools such as RSS feeds can be easily incorporated.

Cost Reduction Daly demonstrated the feasibility of performing 2.5 million PE hours of computational work, not just accumulated runtime, per week over a 16 week period at a user cost of 0.25 FTEs by the use of an effective application monitoring toolset [3]. If one assumes that the combined capability and capacity programmatic resources of the tri-labs is $100k$ Purple equivalent PEs then the application throughput for the entire complex can in principle be maintained at a cost in user time of approximately 1.5-2.0 FTEs.

System Resilience Runtime efficiency and operational utilization are necessary to compute the productive work rate and application throughput. Resilience is a measure of the ability of an application to sustain a throughput of productive work in the face of system hardware and software failures. Application monitoring, or the ability to determine the extent to which an application is making progress, fills a fundamental gap in the ability to quantify the resilience of a system by providing data to accurately measure runtime efficiency and operational utilization.

User Support Support requirements provided by the users represented on the application monitoring team will be integrated into the design and implementation of the production toolset.

5 Acknowledgments

The authors have had many fruitful discussions with coworkers. While any errors or omissions are entirely our own, we especially wish to acknowledge Michael R. Collette (LLNL), Application Developer; Kim Cupps (LLNL), System Monitoring, Michael W. Glass (SNL), Application Developer; Lori A. Pritchett-Sheats (LANL), Application User; Rand Rheinheimer (LANL), System Monitoring; Thomas E. Spelce (LLNL), Applications Support; and James L. Tomkins (SNL), System Architecture.

6 Bibliography

References

[1] Robert A. Ballance. Job monitor: a platform-independent tool for application progress monitoring. in preparation.

[2] M. Chen, A.X. Zheng, J. Lloyd, M. I. Jordan, and E. Brewer. Failure diagnosis using decision trees. In *International Conference on Autonomic Computing (ICAC-04)*, 2004.

[3] J. T. Daly. The capability workload: keeping big systems busy. Los Alamos Technical Report LA-UR-07-2950, Los Alamos National Laboratory, 2007.

[4] J. T. Daly. Facilitating high-throughput asc calculations. *ADTSC Nuclear Weapons Highlights 2007*, pages 204–205, 2007.

[5] J. T. Daly and S. E. Michalak. Proposed metrics for lanl hpc. Los Alamos Technical Report LA-UR-06-8512, Los Alamos National Laboratory, 2006.

[6] John T. Daly. Cis external review: A customer perspective on red storm. Los Alamos Technical Report LA-UR-06-3802, Los Alamos National Laboratory, 2006.

[7] John T. Daly. Methodology and metrics for quantifying application throughput. *Proceedings of the Nuclear Explosives Code Developers Conference*, 2006.

[8] John T. Daly, Robert A. Ballance, Thomas E. Spelce, Michael R. Collette, Lori A. Pritchett-Sheats (tentative), Michael W. Glass John T. Daly, Robert A. Ballance, Thomas E. Spelce, Michael R. Collette, Lori A. Pritchett-Sheats (tentative), and Michael W. Glass. Application monitoring fy08 tripod proposal. Proposal submitted to ASC TriPOD management team.

[9] J.T. Daly, L.A. Pritchett-Sheats, and S.E. Michalak. Application mttfe vs. platform mtbf: A fresh perspective on system reliability and application throughput for computations at scale. In *Proceedings of the 2008 Workshop on Resiliency in HPC*, 8th IEEE International Symposium on Cluster Computing and the Grid, 2008.

[10] W. M. Jones, J.T. Daly, and N.A. DeBardeleben. Application resilience: Making progress in spite of failure. In *Proceedings of the 2008 Workshop on Resiliency in HPC*, 8th IEEE International Symposium on Cluster Computing and the Grid, 2008.

[11] D. A. Reed, C. Lu, and C. L. Mendes. Reliability challenges in large systems. *Future Generation Computer Systems 22*, pages 293–302, 2006.