

Automated Testing of Supercomputers

Argonne Leadership Computing Facility

About Argonne National Laboratory

Argonne is a U.S. Department of Energy laboratory managed by UChicago Argonne, LLC under contract DE-AC02-06CH11357. The Laboratory's main facility is outside Chicago, at 9700 South Cass Avenue, Argonne, Illinois 60439. For information about Argonne and its pioneering science and technology programs, see www.anl.gov.

Availability of This Report

This report is available, at no cost, at <http://www.osti.gov/bridge>. It is also available on paper to the U.S. Department of Energy and its contractors, for a processing fee, from:

U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831-0062
phone (865) 576-8401
fax (865) 576-5728
reports@adonis.osti.gov

Disclaimer

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor UChicago Argonne, LLC, nor any of their employees or officers, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of document authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof, Argonne National Laboratory, or UChicago Argonne, LLC.

Automated Testing of Supercomputers

by
Eric Pershey
Argonne Leadership Computing Facility, Argonne National Laboratory

August 20, 2013

AUTOMATED TESTING OF SUPERCOMPUTERS

by

Eric Pershey

August 20, 2013

CONTENTS

1	THE NEED FOR AUTOMATED TESTING	1
2	CREATING THE TEST HARNESS.....	2
2.1	Harness Requirements	2
2.2	Harness Components	4
2.2.1	The Client	4
2.2.2	The Server.....	5
2.2.3	The State Machine	6
2.2.4	The Tests.....	7
2.2.5	Reporting	8
2.2.6	Enhancements	9
2.2.7	Additional Information	9
3	SUMMARY.....	11
	APPENDIX.....	12

1 THE NEED FOR AUTOMATED TESTING

In this new era of scientific supercomputing, supercomputers are becoming more complex. To ensure that supercomputers are running with their comprehensive capabilities, at their maximum performances, and that they are working properly, there is a need for automated testing.

Automated testing allows for more thorough testing than can be done manually (by any individual). This is accomplished by tirelessly and programmatically pushing the supercomputer's systems to their limits. If done correctly, this leads to a more stable and reliable supercomputer that is running at its maximum performance.

Argonne National Laboratory was set to receive the latest and most advanced supercomputer, Mira, an IBM Blue Gene Q, at the Argonne Leadership Computing Facility (ALCF) sometime in 2012. To enable the acceptance of the supercomputer from IBM, and to ensure that it was running at its stated specifications, there was an abundant need for automated testing. This need for automated testing was the inspiration for the creation of the test harness. The test harness was designed to encompass the functional, performance, and stability testing of the supercomputer Mira.

2 CREATING THE TEST HARNESS

While creating the test harness, ALCF staff determined that there were a few prevalent items that had to be addressed, including:

1. Each test's source, configuration, and output data must be isolated in a well-defined structure on the machine.
2. All information is to be collected and stored.
3. The harness must be able to stop, start, and be fault tolerant.
4. Frequency of tests must be configurable.
5. System failures must not cause the harness to fail.
6. Utilization must be above 90%.

The evolving needs of the user also must be taken into consideration. As time elapses, the specifications of the test harness need to be refined and transformed to meet those needs. To account for the refinements, I needed to construct the harness so that in the event of change, it would be easily adapted to the testing requirements and capabilities.

Using these guidelines, I examined a test harness from Oak Ridge National Laboratory (ORNL), which was given to ALCF by Ricky Kendall and Arnold Tharrington. After much analysis of the source code, I determined that the test harness would not meet our requirements. Their harness did not collect all the necessary information, it was not fault tolerant, and it was too serial; thus, it was not usable. Using this knowledge, the test harness that I planned to create had to be capable of running many tests concurrently, and it had to record all of the data produced by each of them. In addition to this, the test harness had to be fault tolerant, so if there was a system failure or if the harness was to fail, then it could be restarted from where it left off. Therefore, I developed a harness to meet these criteria.

2.1 HARNESS REQUIREMENTS

The requirements were not very strict nor hard to address. Below I will shed some light on these issues.

1. To isolate the information for each test, a directory would be created for each run of each test. This was quite simple but if you dig deeper you will realize that some of these tests create quite a bit of output. Our final size of the acceptance run was 327TB.

2. All information must be collected and stored. I chose to store the source for each test in SVN. This allowed for greater control of each test. The output was stored on disk due to its size — 327TB — since the only other alternative for storage would be on tape. Now the specifics of each run needed to be put somewhere easy to get to, that would allow us to easily figure out how each test went. After much thinking, I picked a database. Now this may hurt fault tolerance because it adds another point of failure, but the way I addressed that is via a client/server configuration. The harness client doesn't talk directly to the database; thus, it can keep running even if the database goes down. Yes, it's true it cannot get any more test information or even store it, but it can buffer it.
3. To allow for great control of the harness, I used a client/server model. The state of each test was stored directly in a database on the server side and the client pushed and pulled all the information to the server. If the web server had an issue, there were handlers in the client that allowed for retry and notification. The tests, when submitted, were not directly tied to the harness client so it could be shut down and restarted at almost any time. There was a danger during a compile that if the compile script was not cleanly written, that breaking in the middle of it could break the build. This ultimately would show up anyway in the test interface and it could easily be handled. Later on, I will also talk about the build controller that could be easily implemented. This aside, the harness proved very fault tolerant and drove acceptance to completion.
4. Setting the frequency of tests was somewhat manual. It was done through the web interface. The problem with setting tests to run within a certain frequency is that it can really hurt utilization so we opted for the harness to pack the machine as best as it could. There were certain tests that were only run a few times because of the nature of the tests. Personnel had to be standing by in case of failure and to assist with diagnostics. This was all done through the web interface.
5. System failures were not much of an issue but there were handlers in certain places that allowed for recovery from such failures as the database, the web server, and the file system. These all were recoverable.
6. Utilization is the amount of the available machine used at a given time. The best way we found to make this work was to submit large jobs first and pepper the machine with smaller jobs. We had about 100 tests and this allowed for a large diversity of sizes that packed the machine well. The 100 tests were cycled about once every 8 hours. Another thing that helped is that we ran many full machine jobs for a long time. This would accomplish two things; prove that it could withstand high usage and prove that the machine was stable.

2.2 HARNESS COMPONENTS

Now that the requirements have been explained and some base information has been given, we will dive into the specifics about the pieces of the harness that made this happen. Here are the pieces that will be described in detail: the client, the server, the state machine, the tests, reporting, enhancements, and additional information.

2.2.1 The Client

The harness client is the interface between the web services and scheduler. This splits up the system so that any back-end can be used and it can be much easier to change how it works. Its job is to create a good file-system structure, check out, build, submit, and finally, complete the tests. This is where the actual processing of the state machine works.

The harness pulls all active tests through a web query and then starts processing each one. For each test, it pulls its current state and attempts to progress to the next. There is much room for improvement here for parallel processing tests, but for our needs, this worked. There is more information about this in the enhancements section.

The file-system structure that the harness maintains allows test owners to very easily find and diagnose problems if one of their tests fails. The job of the harness is to make fresh checkouts and builds for each run and the harness maintains separate directories for each test run (status). This allows the owner to ship the entire directory to the responsible party to find a fix for that specific test. There are three different directories under each test status; output, status and src.

- output: this is the directory that the test is submitted from and where the output will land.
- status: this is where specific scheduler information is replicated for debugging.
- src: this is where the entire checkout of the test goes.

The purpose of each of these folders is to separate output, source, and debugging information for easier processing. This also forces the person who automates their tests to really understand how their tests work and where data is going to and coming from. See the Appendix for an annotated screen capture of the file system structure.

The checkouts are simple SVN info and SVN checkouts of each test. The first thing it does before the checkout is an SVN info of the location in the repository. This marks the last modification time of the repository so that if it needs to be referenced, it can be. Then the checkout is run and it puts the source into the source directory. Some debugging information such as the SVN info is sent to the test's log file so that you can use the web page to find out

problems before you even get into the system. The stderr and stdout from the checkout is placed in the output directory.

The build is done inside the source directory and if it completes successfully, you can move to the next step. If the build fails, you can check some of the output from the test's log in the web page. The stderr and stdout from the build are placed in the output directory.

The submit is done inside the output_directory as many tests dump output directly from where they are submitted. This leads well into the way the directory structure is setup. Since each run of each test gets its own folder, there will be no collisions. An auxiliary submit script is also created so that if you need to, you can manually submit the test again. This script contains the environment used during submission, which can be very useful for debugging.

The complete state will be reached if it detects that the scheduler has completed the test. This stage will then upload some of the test's stderr and stdout to the web interface and into the database. This data has proved to be invaluable in debugging tests. It generally can answer 99% of all questions about the test. We currently grab 8K from the head and tail of each of the outputs.

The theme is to get as much information as possible to the person running the tests so that they can stay in the web page. This allows for much more efficient debugging and testing. In the end, the harness has most of the work, but it remains very stable due to its multicomponent build, its state machine, and its isolation from the state of each test. You can shut down and bring up the client at almost any time and there will be minor disruption. There are certain things you may break — especially during a build — but if the build was done right, rerunning it should allow it to continue without issue.

2.2.2 The Server

The server is built using Django on top of a webserver, which allows quick database design, easy client/server data exchange, and a great interface for easily processing data and generating graphs. Most of the harness user-facing work will be done using the harness web pages. The server provided a small set of interfaces to talk to the client. This, being loosely coupled, allowed for easier testing and faster development. Test entry, lookup, and graphs can all be generated from here as all of the data needed lies inside the database. There were also many other web pages that were created to help assist in debugging the harness, such as: job lookup, usage graphs, and availability graphs. They don't use their data from the harness, but from other systems we have. The harness web page is just an easy place to join their data. Many of the web pages can be seen in the Appendix.

Many of those debugging tools have turned into their own projects. We had a requirement to meet 90% usage on the machine. I needed a way to determine what usage was so I created a system to load up the jobs into a database and then I can process that data to generate graphs. These graphs are now part of our annual reporting.

A great advantage to using the client/server model is that this allowed the whole system to be loosely coupled. I created decorators around key functions in the harness client that accessed the website and allowed them to retry their connection in the case of the website going down. These decorators could also directly email a distribution group to notify the responsible party of an error condition. These were meant to recover so no action was needed unless the machine the client was running on went down.

2.2.3 The State Machine

Given the requirement for parallel execution, I needed to keep track of the state of each test. I created a generic state machine that would allow for tracking the state. It was generically created to allow for later change which saved much time. Once the state machine was in place, I could walk through each test and progress each to the next state. The states are: waiting, checked out, built, submitted, completed, and deleted. The transitions are: checked out, build, submit, complete, reset, hard reset, and database insert. This diagram evolved into its current state as the system was being built (see the states diagram in the Appendix). There were also triggers that allowed control of the state machine. These could be used to pause, reset or auto-rerun a test. It would be good to note that this state machine and diagram were designed before I created one piece of code. This partial plan allowed for the isolation and direction of each task to be completed.

Here are the descriptions for each state.

- waiting: the test has been entered in the system and is waiting to proceed. It will remain here until the trigger action “paused” is cleared.
- checked out: the test has been successfully checked out from our version control system (SVN in our case) and is ready to build.
- built: the test has been successfully built and it is ready to be submitted.
- submitted: the test has been successfully submitted to the scheduler.
- complete: the test has completed. A state machine trigger was added later that upon entry of this state, it would check if it should rerun the machine, and if it was supposed to do that, it would enter waiting automatically and the state machine could clear all actions.

Here are the descriptions for each transition.

- checked out: this function checks out the source code.
- build: this function builds the source code.

- submit: this function submits the binary to the scheduler.
- complete: this function cleans up, collects information on the test and closes out the test (status).
- reset: resets the state machine of the test back to waiting.
- hard reset: resets the state machine of the test back to waiting, it short circuits some checks. If the scheduler did something unexpected with a test job and didn't return correctly, this was needed to fix the test.
- database insert: this is how the test gets put in the system. Our way was through a web interface.

Each transition was given a function to run — to move from state to state. For each test, progress was run that would invoke the correct function to attempt to progress to the next state. The output of each function was in a lookup table so the entire flow of the state machine could be easily identified. Since each transition is isolated, they can easily be tested individually. This made it much easier when it came time to write each transition. This also made it much easier to change each transition. Later in the enhancements section, you can see why.

2.2.4 The Tests

The tests would consist of a set of predefined tests that were identified at the conception of the machine. Now given that, we just need to get a return true script working. This keeps our number of variables to test down. One of the main themes you will notice is that each new addition was isolated and then added. This allowed for easy testing and turnaround. Now to get the system fault tolerant, a data store of some kind must be used to store the state of each test. Using a web framework called Django, I was able to mock up the database that was to keep state of the tests. This would change over time but the structure would remain the same throughout its development. There are three tiers of a test.

- Test Group: This would contain data pertaining to all tests that would be grouped together. If you had a test called MPI PI, this would contain all MPI PI tests of various sizes or iterations.
- Test: This would contain specific information about a single test that could be submitted. It would have very specific build information including, size, wall time, and submitting information.
- Test Status: This would be specific information about a single run of a test. This is also where state is held.

Later I found that keeping a join table instead of test status→test→test group, where → is a foreign key to the next, was much more efficient for queries. You can check this out in the Appendix. The functional harness shows this.

The definition of each test is done directly through a web page. This adds and updates information in the database that maintains state. Once the data is in and correct, the test is ready to go. Getting these tests in here correctly can be very difficult because of the inherent difficulty in automating these processes. We spent much time getting each test to build automatically, breaking apart the dependency of each test, and then finally getting each test to check for performance and correctness. The harness uses the exit code of each scheduler job to determine if the test passed or failed. The harness also provided a number of environment variables to help the test owner run his test according to the harness.

Once the tests are in, you can just release the trigger on a test and allow the harness to process each test. The harness client will start pulling each of the tests to run and processing each of their state machines.

Once the tests are finished running, you will need to find out all the jobs that succeeded and failed. The harness website has all of the information you need to find all the tests. If you look up the jobs in the harness you can see the output and possibly debug the problem without going to the system. In the case of real correctness, performance, or stability problems, you can easily get information on where the test is on the system and get the information to the right person extremely fast. You must not forget to check tests that succeed. It is very possible that one of your scripts may not be correct and return the wrong code; thus, making you think that a test passed when actually it failed.

2.2.5 Reporting

Reporting was not very well defined except for a few key numbers, but this is where much time was spent. The reports were split in two ways: debugging and graphs. Almost all the information needed to debug jobs was in the database. It was easy to develop views that exposed this information to be looked at. This greatly helped for both types of reports. Some of the information in the database was the actual test/job output. The problem with putting job output in a database is that it can be in the gigabytes. I tackled this by sending the first and last 8k of the job output into the database. With this extra bit of information, it helped solve almost 90% of all problems with jobs, which saved time because the test owner didn't have to login anywhere, traverse a file system, and open up files. More debugging information was needed on the test/job, so I pulled in the job information and then correlated it with the harness data. This allowed for greater debugging capability and in the long run, it allowed us to build usage graphs because we had all the job information. Basically, in the end, having a central server that could talk to databases allowed us to create other connections to other databases without creating a single complex tightly-coupled piece of code, allowing for the easy generation of graphs.

2.2.6 Enhancements

There are a few enhancements that I would implement if I were to refactor the source of the harness. Many of them are not that far off.

- Implement a better database structure using join tables. This would allow for much faster queries.
- Worker queues with workers for state transitions. Some of this is already complete and it would increase the speed of the harness by the number of workers.
- Command line control of tests. This would have allowed control of the tests without having to VPN into our system.
- Implement the harness as a service rather than as a standalone tool. This would increase reliability.
- An interface directly to the harness client for checking status. This would allow the harness administrator to check on the message queue, what it's doing, and allow for more than one person to watch its status.

2.2.7 Additional Information

The first time testing on a live machine was interesting because it really gave us an idea of how it was going to work. Basically, the harness not only tests the supercomputer itself, but the scheduler, file system, and more. Many bugs in the scheduler were found using the harness and the reliability was increased because of it. The sheer volume at which it can run tests can greatly outrun any human; and thus, it can find problems that a human may not easily hit.

The first machine it was run on was our Blue Gene P 1024 node machine, Surveyor. I was able to run simple MPI PI tests and ensure that everything was working correctly. After using Surveyor we started to use it on the 40960 node Intrepid machine with great success. It was so successful to test on that we started to use it to test Intrepid for various problems. In the end I used the harness to test all of our Blue Gene P and Q machines successfully.

It is important to note some of the other neat things that the harness exposes. The harness tracks much information on each test, and because of its speed, it can run through all the tests quickly — quick enough that many runs of each test can be done during a day. If you start plotting trend-lines for each runtime, you can see when there are file system slowdowns and basically find out how the machine is acting from day to day. There have been instances when a test has passed and the runtime was far shorter, thus leading us to look at a problem with a test's script that lead us to think it passed when it did not. Even trivial tests such as simple MPI PI or a random matrix multiply have proved to be invaluable in testing just simple booting of partitions

and job turnaround time. These tests may not be to test the compute power and reliability of each node, but just to make sure that the machine is working as it is supposed to work.

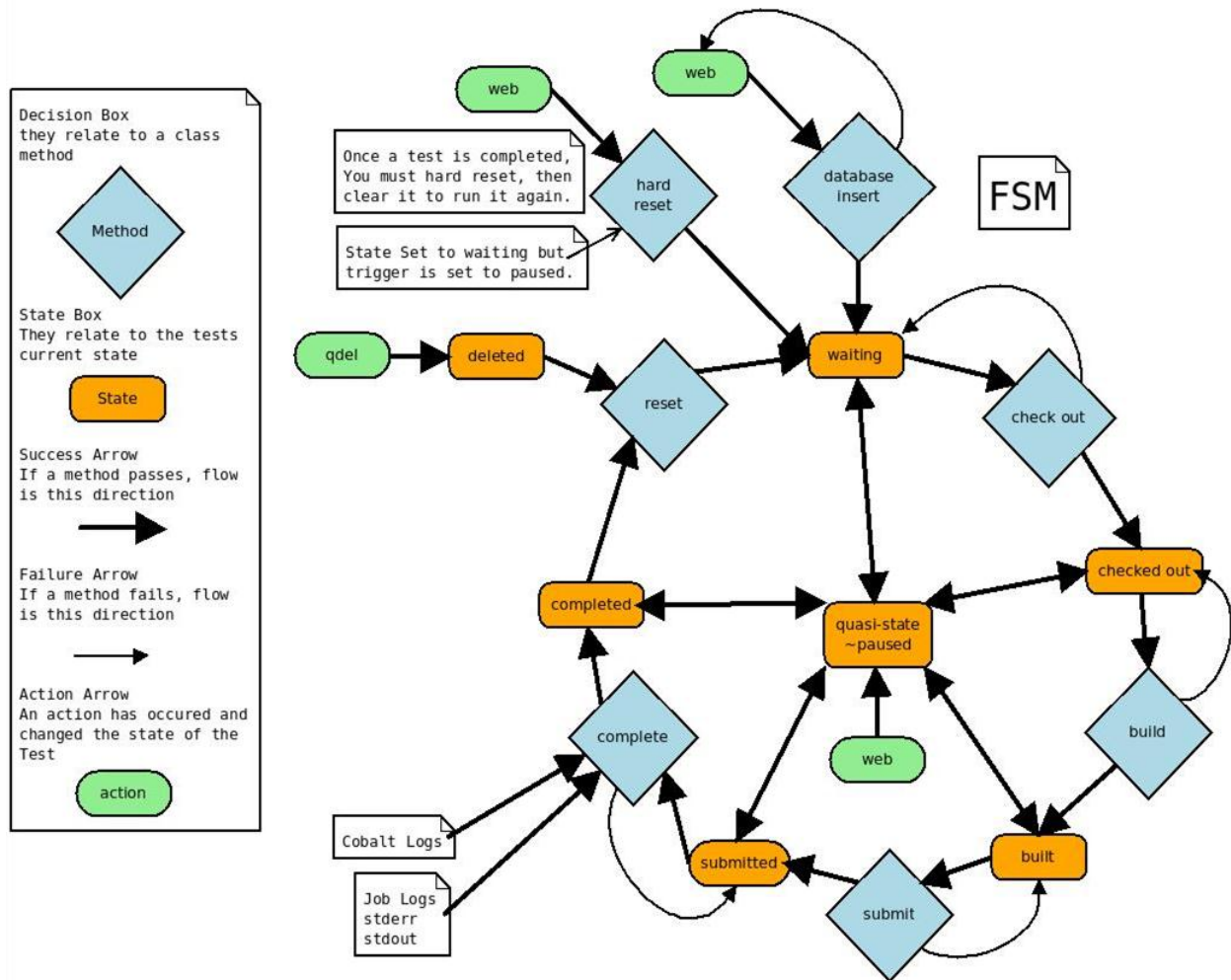
3 SUMMARY

In the end, you are not just testing the compute ability and power of a supercomputer. You are testing many, if not all of the components, that bring the system together. You must not test once, you must test many times. The environment of a new machine is very dynamic and you must be ready to act.

By using an automated framework to test your machine you can save much time and find many more problems much faster than if the testing is done manually. If you do it right, this testing framework can also be used at every maintenance cycle or during some scheduled period to test for regression and other problems with the click of a button. This harness has saved a great amount of time, lowered stress, and reduced the overall problem of testing the machine.

APPENDIX

State Machine



File System

The image shows a terminal window with a file system tree and several annotations. The terminal output is as follows:

```

18:27:12 pershey@vestalac1: /gpfs/vesta_scratch/projects/Stability_Harness/harness_workspace/NEK-medium-32rpn_121/838_7155
File Edit View Terminal Help
18:26:23 pershey@vestalac1: /gpfs/vesta_scratch/projects/Stability_Harness/harness_workspace$ ls
GPAW_117 gtc_112 input lib mpi_pi_107 NAMO_109 NEK-medium-32rpn_121 NEK-medium-64rpn_122 test
18:26:25 pershey@vestalac1: /gpfs/vesta_scratch/projects/Stability_Harness/harness_workspace$ ls lib
ase-r2047 ESSL5.1.1-20120305 esll_fftw fftw3 HPM old python-2.6.6-ckn-gcc status.txt
18:26:31 pershey@vestalac1: /gpfs/vesta_scratch/projects/Stability_Harness/harness_workspace$ ls input
FLASH GFOL GFMC gpaw-benchmarks gpaw-setups-0.6.6300 namd namd-input namd_test100M namd_test1M namd_test_20M nek-input
18:26:35 pershey@vestalac1: /gpfs/vesta_scratch/projects/Stability_Harness/harness_workspace$ cd NEK-medium-32rpn_121/
18:26:57 pershey@vestalac1: /gpfs/vesta_scratch/projects/Stability_Harness/harness_workspace/NEK-medium-32rpn_121$ ls
831_7267 838_7155 845_7265
18:26:58 pershey@vestalac1: /gpfs/vesta_scratch/projects/Stability_Harness/harness_workspace/NEK-medium-32rpn_121$ cd 838_7155/
18:27:04 pershey@vestalac1: /gpfs/vesta_scratch/projects/Stability_Harness/harness_workspace/NEK-medium-32rpn_121/838_7155$ ls
output src status

```

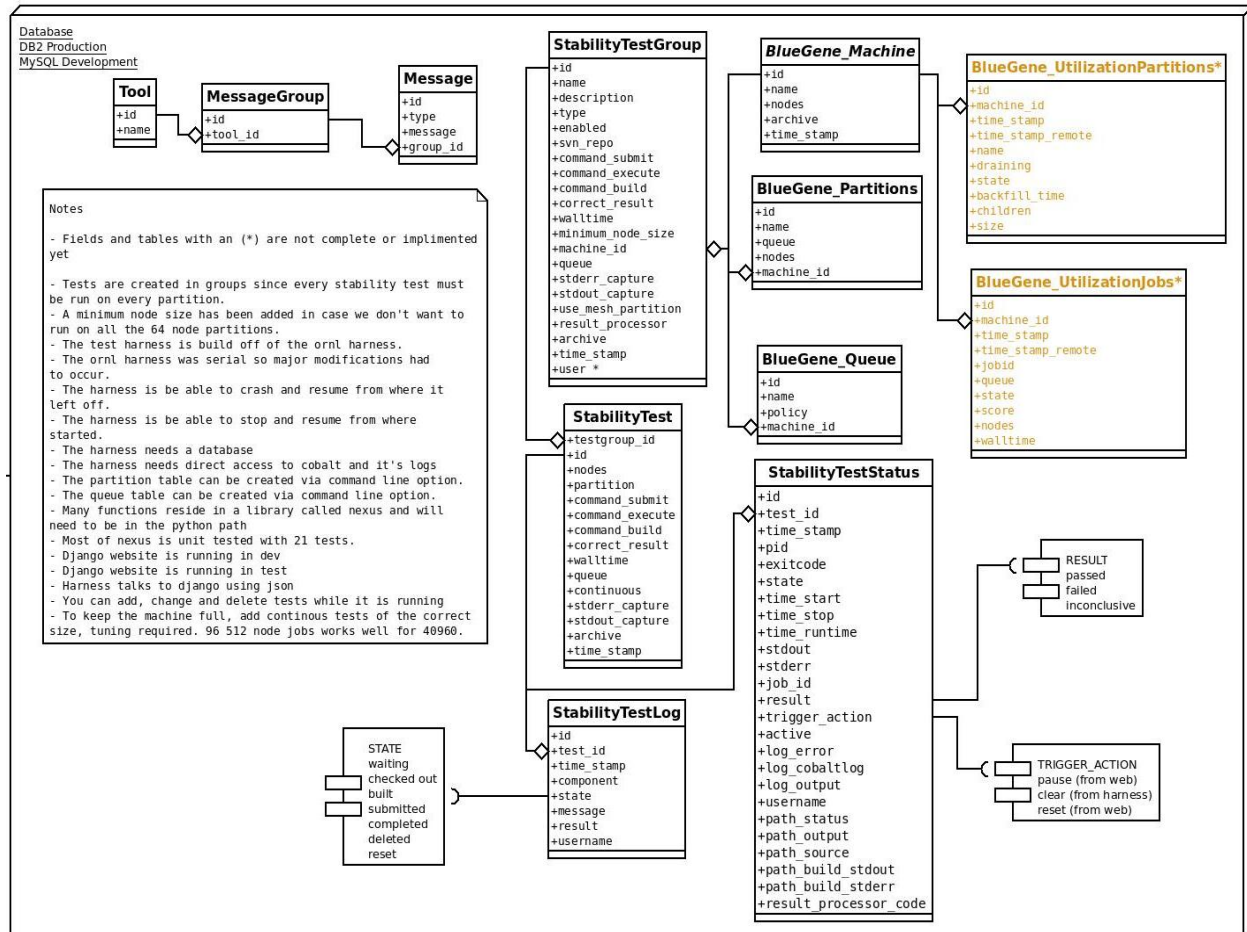
Annotations (green text and arrows):

- Test Group Name**: Points to the `test` directory in the root of the workspace.
- Test Status ID**: Points to the `status` file in the `838_7155` directory. Description: "one exists for every single run of a test it contains the specific jobid and location of paths".
- Test ID**: Points to the `838_7155` directory. Description: "Is basically a template that will be run # of nodes, command line, etc...".
- Test Group ID**: Points to the `838_7155` directory.
- \$status_directory**: Points to the `status` file. Description: "location cobalt logs are moved to".
- \$source_directory**: Points to the `src` directory. Description: "Where the source is stored location of build scripts location of submit scripts".
- \$output_directory**: Points to the `output` directory. Description: "Build Logs go here. jobs Submitted from here".
- \$base_directory**: Points to the `src` directory. Description: "use to reference such things as \$base_directory/input/gpaw-benchmarks \$base_directory/lib/charm-mpi".

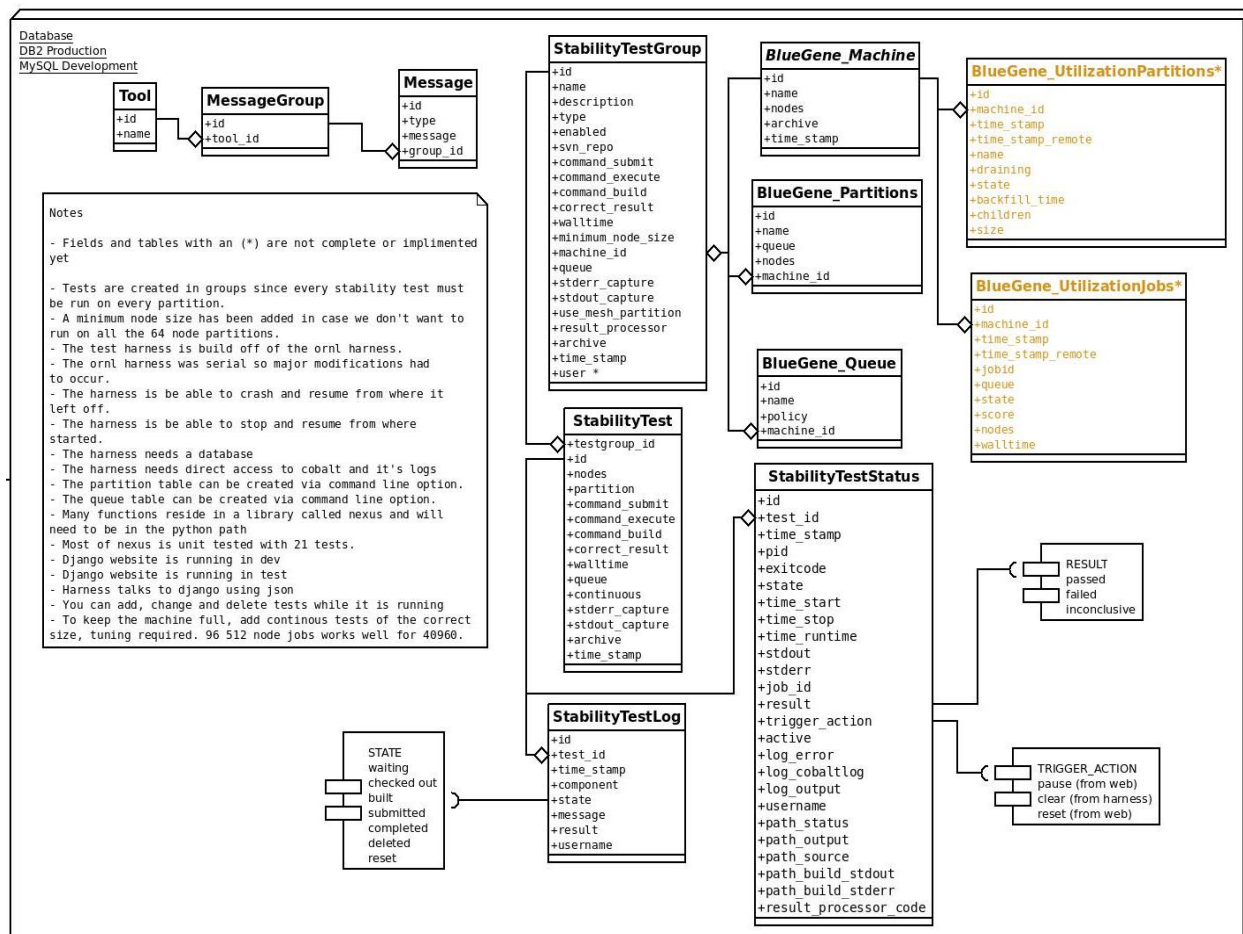
Additional notes:

- "I run manual checkouts of tests from in here. I set the variables directly and run the scripts" (points to the `test` directory).

Stability Harness Diagram



Functional Harness Diagram



Test Group

[Mira Home](#)
[Welcome pershey](#)
[Admin Tools](#)

Harness

[Home/Summary](#)
[Live](#)
[Test Groups](#)
[Add a Test Group](#)
[GOW apps](#)

[Top](#)
[Reload](#)
[Group](#)
[Tests](#)
[Bottom](#)

Harness Group View

name	Mira
nodes	49152
cores	16
username	0

Machine:Mira Group:debug-mpi_pi:174
Start 2012-12-17 09:36:30 UTC
End 2012-12-17 21:36:30 UTC

Group

name	debug-mpi_pi
description	Simple fast tests of a job
type	Functional
enabled	1
queue	testing
svn_repo	https://svn.xicf.aci.gov/repos/mira_atg/tests_stability/mpi_pi/
command_submit	harness.gow -o \$nodes -t \$walltime -q \$queue -A \$project \$cmd.execute
command_execute	\$source_directory/mpi
correct_result	1.14159265
command_build	make \$source_directory/mpi
walltime	10
stderr_capture	1
stdout_capture	1
use_peak_partition	0
result_processor	/bin/true
time_stamp	2012-10-17 08:58:41.574000
username	2
project	Acceptance

[View Test Group Live](#)
[Edit Test Group](#)
[Results Test Group \(history\)](#)

Controls

Generate one test per partition.

Generate one Test

Set all tests Continuous

Set all tests Non-Continuous

Inactivate Tests

Activate Tests

Clear all Actions

Reset all Tests

Fix all Tests, this will clear the log

Delete all Tests

Fill the machine with continuous node tests of size.

Tests, Count:1

This box is scrollable, use the arrow keys left or right. You may also use the scrollbar at the bottom.

Trigger	Links	Current Result	Current Submit JobID	Last Result	Last Submit JobID	Last ExitCode	nickname	testgroup	nodes	queue	command	command execute	command build
<div>Play/Pause</div> <div>Reset</div> <div>This will reset the test. You do not build or execute it.</div>	test link				test	0	debug-mpi_pi	debug-mpi_pi	312	testing	harness.gow -o \$nodes -t \$walltime -q \$queue -A \$project \$cmd.execute	\$source_directory/mpi	make \$source_directory/mpi

Test View

Mira Home

Welcome pershey

Admin Tools

Harness

Home/Summary

Test Groups

Add a Test Group

SW apps

Top

Reload

Group

Test

Controls

Status

Log

Bottom

Harness Test View

View Test Group Live

Edit Test Group

View Test Group

Test

Machine:Mira

Test:debug-mpi_pi::1165

Start 2012-12-17 09:37:16 UTC

End 2012-12-17 21:37:16 UTC

total runtime(minutes):0.000

Runtime (Minutes)

1.0

0.8

0.6

0.4

0.2

0.0

2012-12-17 09

2012-12-17 10

2012-12-17 11

2012-12-17 12

2012-12-17 13

2012-12-17 14

2012-12-17 15

2012-12-17 16

2012-12-17 17

2012-12-17 18

2012-12-17 19

2012-12-17 20

2012-12-17 21

2012-12-17 22

Edit Test

nickname	debug-mpi_pi
testgroup	debug-mpi_pi
nodes	512
queue	testing
command_submit	Sharness_runb -n \$nodes -t \$walltime -q \$queue -A \$project \$cmd_execute
command_execute	\$source_directory/mpi
command_build	make \$source_directory/mpi
correct_result	3.14159265
walltime	10
active	1
continuous	0
stderr_capture	1
stdout_capture	1
auto_pass	0
id	1165
project	Acceptance
partition	
Result Current	<div></div>
Result Last	<div>returned 0</div>

Controls

Play/Pause

Pause or Play the test.

Reset

This will reset the test. Use to fix a build or incorrect checkout.

Pause

Stop processing the test through the state machine.

Clear

This will clear the trigger.

Toggle Active

Activate the test. This will cause the harness to start processing it.

Duplicate

Create an empty non-active version of the test.

Delete

Delete the test.

Status

This box is scrollable, use the arrow keys left or right. You may also use the scrollbar at the bottom.

id	time stamp	username	state	result	trigger action	pid	exitcode	time start	time stop	time runtime	stdout	stderr
<div>View</div> 26741	2012-10-29 19:06:06.307470	None	waiting	<div></div>	pause	None	None	None	None	None	None	None
<div>View</div> 26714	2012-10-29 17:13:15.692716	pershey	completed	<div>returned 0</div>	None	None	0	2012-10-29 19:07:22	2012-10-29 19:07:58	136	this is node 435 of 512 total this is node 149 of 512 total this is node 130 of 512 total this is node 233 of 512 total this is node 467 of 512 total this is node 3 ...MESSAGE TRUNCATED... 061386699133272 node 140: local pi value is 3.0061381263375469	2012-10-29 19:07:15.727 (I [0x40001398ad]) ibm_runjob.AbstractOptions properties file /bgape/local/etc/bg-proper 2012-10-29 19:07:15.728 (I [0x40001398ad]) ibm_runjob.AbstractOptions open file deacr ...MESSAGE TRUNCATED...

Automated Testing of Supercomputers

Eric Pershey

17

.65/

[Home/Summary](#)
[Test Groups](#)
[Add a Test Group](#)
[SOW apps](#)

[Top](#)
[Reload](#)
[Group](#)
[Test](#)
[Controls](#)
[Status](#)
[Log](#)
[Bottom](#)

time stamp *	username *	component *	message *	result *	next state *
2012-10-29 19:06:06.306812	perahay	harness-complete	Job is complete.	passed	completed
2012-10-29 19:00:52.241918	perahay	harness-submit	<pre> cSubmitted test CMD:/usr/bin/gsub -n 512 -t 10 -q testing -A Acceptance /gpfs/mira-fs0/projects/Acceptance/harness_workspace/debug-mpi_pi_174/1165_26714/src/mpi PSE CMD:/gpfs/mira-fs0/projects/Acceptance/harness_workspace/debug-mpi_pi_174/1165_26714/output STDOUT:37133 STDERR: FOOT CMD:/gpfs/mira-fs0/projects/Acceptance/harness_workspace/debug-mpi_pi_174/1165_26714/output RETURN:0 </pre>	complete	submitted
2012-10-29 19:00:52.241918	perahay	harness-submit	<pre> Submitting test /usr/bin/gsub -n 512 -t 10 -q testing -A Acceptance /gpfs/mira-fs0/projects/Acceptance/harness_workspace/debug-mpi_pi_174/1165_26714/src/mpi CHKmake /gpfs/mira-fs0/projects/Acceptance/harness_workspace/debug-mpi_pi_174/1165_26714/src/mpi PSE CMD:/gpfs/mira-fs0/projects/Acceptance/harness_workspace/debug-mpi_pi_174/1165_26714/src STDOUT:mpicc /gpfs/mira-fs0/projects/Acceptance/harness_workspace/debug-mpi_pi_174/1165_26714/src/mpi.c -o /gpfs/mira-fs0/projects/Acceptance/harness_workspace/debug-mpi_pi_174/1165_26714/src/mpi STDERR: FOOT CMD:/gpfs/mira-fs0/projects/Acceptance/harness_workspace/debug-mpi_pi_174/1165_26714/src RETURN:0 </pre>	started	submitted
2012-10-29 19:00:52.241918	perahay	harness-build	<pre> CHKmake /gpfs/mira-fs0/projects/Acceptance/harness_workspace/debug-mpi_pi_174/1165_26714/src/mpi PSE CMD:/gpfs/mira-fs0/projects/Acceptance/harness_workspace/debug-mpi_pi_174/1165_26714/src STDOUT:mpicc /gpfs/mira-fs0/projects/Acceptance/harness_workspace/debug-mpi_pi_174/1165_26714/src/mpi STDERR: FOOT CMD:/gpfs/mira-fs0/projects/Acceptance/harness_workspace/debug-mpi_pi_174/1165_26714/src RETURN:0 </pre>	complete	built
2012-10-29 19:00:52.241918	perahay	harness-build	Built test	complete	built
2012-10-29 19:00:52.241918	perahay	harness-build	<pre> Building test: make /gpfs/mira-fs0/projects/Acceptance/harness_workspace/debug-mpi_pi_174/1165_26714/src/mpi </pre>	started	built
2012-10-29 19:00:52.241918	perahay	harness-check_out_test	<pre> Check out test, Output: svn co https://svn.alcf.anl.gov/repos/mira_atp/tests_stability/mpi_pi/ /gpfs/mira-fs0/projects/Acceptance/harness_workspace/debug-mpi_pi_174/1165_26714/src returned 0 STDOUT Can be found at /gpfs/mira-fs0/projects/Acceptance/harness_workspace/debug-mpi_pi_174/1165_26714/output/svn_stdout.txt STDERR Can be found at /gpfs/mira-fs0/projects/Acceptance/harness_workspace/debug-mpi_pi_174/1165_26714/output/svn_stderr.txt Check out complete </pre>	complete	checked out
2012-10-29 19:00:52.241918	perahay	harness-check_out_test	<pre> Check out info test, Output: svn info https://svn.alcf.anl.gov/repos/mira_atp/tests_stability/mpi_pi/ returned 0 STDOUT:Path: mpi_pi URL: https://svn.alcf.anl.gov/repos/mira_atp/tests_stability/mpi_pi Repository Root: https://svn.alcf.anl.gov/repos/mira_atp Repository UUID: 0be2aab-22c4-48ed-aec0-9af098782705 Revision: 808 Node Kind: directory Last Changed Author: perahay Last Changed Rev: 2 Last Changed Date: 2011-04-13 16:04:40 +0000 (Fri, 13 Apr 2011) </pre>	complete	checked out

Test Status View

Mira HomeWelcome persheyAdmin Tools

Harness

[Home/Summary](#)
[Test Groups](#)
[Add a Test Group](#)
[jovl apps](#)

[Top](#)
[Reload](#)
[Bottom](#)

Harness Test Status View

Group

[View Test Group Live](#)
[View Test Group](#)
[Edit Test Group](#)

Test

nickname	debug-mpi_pi
testgroup	debug-mpi_pi
nodea	512
queue	testing
command_submit	harness_qsub -n 3nodes -t 3walltime -q Square -A \$project_ford_execute
command_execute	source_directory/mpi
command_build	make source_directory/mpi
correct_result	3.14159265
walltime	10
active	1
continuous	0
stderr_capture	1
stdout_capture	1
auto_pass	0
id	1145
project	Acceptance
partition	
Result Current	<div>Passed</div>
Result Last	<div>returned 0</div>

[View Test](#)
[Edit Test](#)

Test Status

Confirm Pass: this will set user_comments and user_confirm.

Confirm Pass the test.

Pass

Confirm Fail: this will set user_comments and user_confirm.

Confirm Fail the test.

Fail

Clear the Confirmation:

Clear

key	value
status	
test_cache	debug-mpi_pi-
active	0
exitcode	0
id	26714
job_id	37133
log_mballing	/opta/mira-fa0/projects/Acceptance/harness_workspace/debug-mpi_pi_174/1145_26714/status/37133-mballing
log_stderr	/opta/mira-fa0/projects/Acceptance/harness_workspace/debug-mpi_pi_174/1145_26714/status/37133-stderr
log_output	/opta/mira-fa0/projects/Acceptance/harness_workspace/debug-mpi_pi_174/1145_26714/status/37133-output
path_build_stderr	/opta/mira-fa0/projects/Acceptance/harness_workspace/debug-mpi_pi_174/1145_26714/output/build_stderr.txt
path_build_stdout	/opta/mira-fa0/projects/Acceptance/harness_workspace/debug-mpi_pi_174/1145_26714/output/build_stdout.txt
path_output	/opta/mira-fa0/projects/Acceptance/harness_workspace/debug-mpi_pi_174/1145_26714/output
path_stderr	/opta/mira-fa0/projects/Acceptance/harness_workspace/debug-mpi_pi_174/1145_26714/err
path_status	/opta/mira-fa0/projects/Acceptance/harness_workspace/debug-mpi_pi_174/1145_26714/status
pid	None

Automated Testing of Supercomputers

Eric Pershey

19

Test Status View (Continued)

Home/Summary Test Groups Add a Test Group SQL stats Top Reload Bottom	node 418: local pi value is 0.0061328234952204 node 253: local pi value is 0.0061359718522820 node 142: local pi value is 0.0061380884118816 node 48: local pi value is 0.0061398799140130 node 509: local pi value is 0.0061310860474444 node 414: local pi value is 0.0061328998489388 node 91: local pi value is 0.006139604096036 node 377: local pi value is 0.0061336060504477 node 426: local pi value is 0.0061326707833311 node 223: local pi value is 0.0061365440087095 node 22: local pi value is 0.0061403752899372 node 146: local pi value is 0.0061380121594305 node 503: local pi value is 0.0061312006280611 node 412: local pi value is 0.0061329380252414 node 193: local pi value is 0.0061371160812514 node 449: local pi value is 0.0061322317035900 node 274: local pi value is 0.0061355712929038 node 127: local pi value is 0.0061383743452640 node 166: local pi value is 0.0061376308747663 node 50: local pi value is 0.0061398418055513 node 327: local pi value is 0.0061345601748278 node 302: local pi value is 0.0061350371498766 node 251: local pi value is 0.0061360099986524 node 505: local pi value is 0.0061311624348925 node 410: local pi value is 0.0061329762011729 node 237: local pi value is 0.0061362770128120 node 38: local pi value is 0.0061400704507052 node 42: local pi value is 0.0061399942371517 node 333: local pi value is 0.0061344456921686 node 294: local pi value is 0.0061351897696144 node 211: local pi value is 0.0061367728477960 node 345: local pi value is 0.0061342167168114 node 386: local pi value is 0.0061334342833977 node 87: local pi value is 0.0061391367314961 node 20: local pi value is 0.0061404133931555 node 154: local pi value is 0.0061378596500461 node 383: local pi value is 0.0061334915399167 node 388: local pi value is 0.0061333781119209 node 65: local pi value is 0.0061395537811555 node 379: local pi value is 0.0061335676806421 node 282: local pi value is 0.0061354186880553 node 207: local pi value is 0.0061368491245082 node 62: local pi value is 0.0061396131469191 node 24: local pi value is 0.0061403371863443 node 469: local pi value is 0.0061318498552608 node 436: local pi value is 0.0061324798851226 node 67: local pi value is 0.0061395178685120 node 475: local pi value is 0.0061317352935348 node 402: local pi value is 0.0061331289011874 node 71: local pi value is 0.0061394416441022 node 164: local pi value is 0.0061376690049131 node 162: local pi value is 0.0061377071346865 node 357: local pi value is 0.0061339877280724 node 310: local pi value is 0.0061348845241851 node 89: local pi value is 0.0061390986157368 node 323: local pi value is 0.0061346364947413 node 266: local pi value is 0.0061357238917945 node 213: local pi value is 0.0061367347088805 node 60: local pi value is 0.0061396512376270 node 144: local pi value is 0.0061380502858428 node 453: local pi value is 0.0061321553368897 node 316: local pi value is 0.0061347700510100 node 97: local pi value is 0.0061389461489597 node 459: local pi value is 0.0061320407840589 node 306: local pi value is 0.0061349608377750 node 101: local pi value is 0.0061388699133272 node 140: local pi value is 0.0061381265375469 node 58: local pi value is 0.0061396893679605 node 420: local pi value is 0.0061327853178046 pi is approximately 3.1415926535917800, Error is 0.000000000019869																					
	<table> <tr><td>test_id</td><td>1163</td></tr> <tr><td>time_runtime</td><td>158</td></tr> <tr><td>time_stamp</td><td>2012-10-29 17:15:15.602716</td></tr> <tr><td>time_start</td><td>2012-10-29 19:00:22</td></tr> <tr><td>time_stop</td><td>2012-10-29 19:07:18</td></tr> <tr><td>trigger_action</td><td>None</td></tr> <tr><td>user_action</td><td>None</td></tr> <tr><td>user_comments</td><td>None</td></tr> <tr><td>user_confirmed</td><td>-1</td></tr> <tr><td>user_threshold</td><td>None</td></tr> <tr><td>username</td><td>pershey</td></tr> </table>	test_id	1163	time_runtime	158	time_stamp	2012-10-29 17:15:15.602716	time_start	2012-10-29 19:00:22	time_stop	2012-10-29 19:07:18	trigger_action	None	user_action	None	user_comments	None	user_confirmed	-1	user_threshold	None	username
test_id	1163																					
time_runtime	158																					
time_stamp	2012-10-29 17:15:15.602716																					
time_start	2012-10-29 19:00:22																					
time_stop	2012-10-29 19:07:18																					
trigger_action	None																					
user_action	None																					
user_comments	None																					
user_confirmed	-1																					
user_threshold	None																					
username	pershey																					



Argonne Leadership Computing Facility

Argonne National Laboratory

9700 South Cass Avenue, Bldg. 240

Argonne, IL 60439-4847

www.anl.gov



Argonne National Laboratory is a U.S. Department of Energy
laboratory managed by UChicago Argonne, LLC