CUG 2024 Slurm Rollout
Minimizing Downtime for Node Upgrades on Shasta systems and beyond

Practicalities of maintaining and improving a production HPC cluster

Introduction

Motivation
The motivation for pioneering this method of HPC system upgrades was to 1) minimize utilization waste while also 2) reducing risk of unexpected prolonged downtimes occurring from a derailed upgrade process, 3) provide users with uninterrupted access to the machine and 4) possibly eliminate the need for a Test and Development System (TDS). We consider the impact to the users in addition to the availability of the machine itself. It would be proper to describe a typical system upgrade process along with a few hazards and horror stories. We will also share our observances of the behavior of the user community throughout this process.

Summary of Previous Serial Upgrade Process
A typical upgrade process today for LANL proceeds like this. At a predetermined time, the cluster is 'taken down'. This includes preventing users from submitting new workload. Current workload is halted. (Alternatively, the system can be drained using a reservation, but in either case, productive time is lost either via drain time or terminated jobs not being fruitful.) And in the situations where major programming environment changes or the like will occur with the upgrade, all submitted workload is canceled. This in essence begins the “dark side of the moon” countdown clock. System admins work furiously to apply the system updates, upgrades and image modifications. Once that has been completed, the lengthy boot sequence is commenced and fingers are crossed in hopes for a smooth bring up of the system. If things go well, the system will be up after a few hours and functional testing can begin. This generally takes a few hours even when automated. If things do not go smoothly, the system can be held overnight and into the next day given that this process barely fits into a full day's work.

After functional testing is complete, performance testing begins. If everything checks out, the system is returned to the users and they begin recompiling their codes and validating their builds. The admins hold their breath and brace for a possible influx of user issues.

The days following the upgrade are a tentative time where changes and fixes are addressed in a heightened state of urgency. In the near worst case scenario the entire upgrade is found to have major flaws and the system has to be taken down again in the manner described above and the previous incarnation of the system restored. Then the

arduous task of figuring out what went wrong and what can be done to remedy or mitigate the issues. This can potentially cost weeks of full system down time.

Summary of Parallel Scale Over Process

In our new procedure, we look to minimize utilization loss and prolonged outage risk. We mitigate these by differentiating between function testing and scale testing in a parallelized environment with phased steps in scale. We begin by utilizing a user access node (aka login node, user node) hereinafter referred to as UAN. This UAN is updated to the new environment configuration without the users being able to access it initially. A compute node is segregated from the general pool where user workflows are continuing on as usual. This compute node is tagged with a Slurm feature that will not allow it to be selected by the users. (This also assumes that the job submission script is automatically tagging all currently submitted workload with a feature constraint that ties them to the old environment.) Now we have a UAN and compute node where functional testing can be carried out with a minimal number of resources all the users carrying on their normal workloads. This allows ample time to functionality of the new environments over the period of days if necessary.

Once satisfied that the new environment is functionally stable, more compute nodes can be rebooted into the new environment and be flagged with the new environment feature in Slurm. Once adequate performance testing has been completed, users will be allowed to log into the new UAN and begin compiling, validating and submitting their workload in the new environment in parallel with their old environment workload being allowed to work through the system under the old environment in which it was submitted.

At this point, most of the old UANs can be rebooted into the new UAN environment. As the old workload completes and phases out (possibly by the administration disallowing new submissions from the old UANs.) More and more compute nodes can be transitioned into the new environment. Once all the old workflow has completed, all of the UANs and compute nodes can be transitioned and the update process can be considered complete.

IF problems arose during the functional phase, they can be addressed without any impact to the users. (Save the lack of use of one UAN and one compute node.) If problems are encountered during scale testing, the majority of computes can simply be rebooted back into the old environment until a solution is discovered. The same holds true in the case of a full rollback since you still have the old environment available to users in parallel.

Fundamentals

How do you upgrade and maintain a production cluster resource without actually making risky changes? That's a core question for many supercomputing centers. It comes down to scarce resources — it's hard to spend budget on machines on which to test changes before putting them into production. The closer the test machine comes to being a real test case for its production instance, the harder that gets. Weather forecast systems are an example, where having twin systems and upgrading and testing them each in turn is justified for uptime criticality. That's the ideal — being able to fully check out a configuration before committing to it.

More commonly, a Testing and Development System (TDS) will be purchased along with the main system. A TDS represents a minimal subset of all of the functional components of the larger system, and can shake out many issues with a configuration or software change before it hits production. Clearly there are issues that can't be brought to light, however — the one thing a TDS can't simulate is behavior at scale. The classic workflow is therefore to make changes on the TDS and think really hard about issues or bottlenecks that might arise when jobs scale up. The system is then taken from the users, upgraded, tested for functionality, and returned to the users if all goes well. If issues arise, the changes are backed out, the system is tested again, and is returned to the users. This can take hours when all goes well, and days under more challenging conditions. Backing out changes is not always a perfect reversion, either.

At Los Alamos National Laboratory (LANL), we have been pushing toward a different ideal — to make changes at a small scale and roll them out to the cluster progressively, moving nodes from the previous "green" state to the new "blue" state as they prove out at multiple scales. (We will use "green" as the old state and "blue" as the new by convention in this paper, but the order is not important and will reverse with every other system change.) This is definitely not a new idea, but the implementation is nontrivial. Several issues crop up immediately. Any change will potentially affect user output. User codes may need recompilation. The workload manager (WLM — Slurm, in this case) will therefore have to track which parts of the machine are in which state — something for which it was not designed. If poorly done, there is a potential for significant idle time as the switch occurs, but there's also the potential for disruption of the WLM or other components if updating is too aggressive.

Here's how LANL addresses these issues, from a practical point of view.

Practical Operations

Changes to the system can encompass the compute nodes (which may not be a homogeneous set), interactive front-end (FE) nodes, management nodes, and service nodes (such as Lnet routers and filesystem gateways). Each of these classes has different user impact, and the end metrics for our system change can be summarized as: was the change successful, and how much science time did our users lose. Especially with more modern systems, management plane changes can often be upgraded with minimal impact. This leaves us free to focus on user-facing systems — the FE and compute nodes, and the service nodes upon which they depend.

Overview

The key parts of a clean system rollout with happy users are

- Time for users to compile and test code against new software and configs if desired
- Time for user jobs to run to completion in the old environment
- Maximized nodes available at any given time to prevent workload backup
- Clean separation between old and new
- Few to no “gotchas” where users might wind up with unexpected results

To make this happen, the machinery has to be right and human error has to be minimized. We’ll first discuss the machinery that makes this happen, then outline procedures.

Underlying machinery

The key to all of this is the workload management system and the node orchestration working in harmony. As nodes boot or are configured into the new “blue” state, they need to be recognized by WLM and automatically associated with one or more updated FE nodes. Once this occurs, users on the new FE node will use the “blue” side of the cluster without any friction or special knowledge, while users on the “green” side remain undisturbed and productive.

To make this distinction in Slurm, there are a variety of ways to distinguish node states, some of which are better adapted than others. They are all defined in the slurm.conf file statically, but some can also be added dynamically via scontrol.

We could, for example, define a new node partition called “Blue” and start moving nodes over to it. Downsides include complexity (partitions are already used for other distinctions, and this would add another dimension) and a lack of dynamic update mechanism. There are also accounting issues with moving nodes between partitions.

Another Slurm tool is GRES (Generic RESources), which is focused on scheduling specific node features. There’s no fundamental reason it would not be useful for node

tagging, but its stated function doesn't overlap our desired function as well as Slurm node_features.

A Slurm node can be tagged with a set of node_features using the AvailableFeatures flag. All or a subset of those features can then be set as ActiveFeatures depending on node state. It's even possible to define helper plugins in Slurm to change boot parameters, kernels, and even images. While we don't yet need that level of sophistication, those tools would be very useful for sites that are not using a management plane that provides inbuilt boot orchestration like the Cray CSM Boot Orchestration Services/Boot Scripting Service (BOS/BSS).

Node updates and node boot states are central to this rollout concept. Other job schedulers will offer different methods, but we are working with Slurm, which will inform the next few paragraphs. With that in mind, the Slurm concept of Features seems exactly suited to our tagging needs. Nodes will be tagged with the AvailableFeatures green and blue, and the feature list will be modified based on which set the nodes should inhabit. Nodes will lose both ActiveFeature tags when they the rollout starts, and are retagged with the correct set when they are booted, configured, and ready to accept jobs in the new set.

This on-the-fly tagging is ephemeral, which presents some risk. If the process is interrupted, the state of the system may not be as clear. If for whatever reason the scheduler restarts in the midst of the rollout, there's no record of what nodes were in which state. Making the changes as synchronous as possible with the local config file just requires rewriting the nodes specification of the slurm.conf file. To make this easier to track and version (and to make the changes more distinguishable from the more global configuration changes and partition modifications), the node definitions will be done in a new nodes.conf file which will be included into the slurm.conf file. Once a node's ActiveFeatures have been updated, it is very fast to expose the features of all of the nodes in JSON format, parse them into a new nodes.conf file, and create a new version of that file. Old copies can be kept at very low impact, for easy error tracking and reversion to a known good state.

If the Slurm cluster is running configless (where the compute nodes configuration files are updated on the fly by the slurmctld), there is no need to propagate the changes in nodes.conf out to the compute and FE nodes. If, however, the slurm.conf is not being propagated automatically, it either needs to be in a shared space visible to all of the nodes and the scheduler, or it needs to be propagated by some other means on every sync, and the slurmctld and each node's slurmd restarted. This is more demanding, and may affect the sync cycle time.

Node configuration will be designated from the rollout tooling to a subset of nodes depending on the phase of rollout. Depending on whether the nodes will need a reboot into a new image or a reconfiguration, our rollout tool will call boot orchestration or configuration tools on a per-node basis as nodes are queued to roll over to the new state.

Node configurations and image assignments are specific to the management plane of the system. For the purposes of this paper, we will be implementing with the aforementioned BOS/BSS mechanisms, as well as the Cray CSM Configuration Framework Service, which applies Ansible configurations to sets of nodes. If you are adapting this framework to a different system, you may need to build or plug into different tools that reboot nodes into a new image and/or apply configurations via tools like Ansible or Puppet.

The final component we need is the coordination between the scheduler (Slurm for us, as mentioned above) and the power controls in case of node reboots. In Slurm, nodes can be given a reboot ASAP command — as soon as the current job is completed (or immediately, if the node is unused). Some boot orchestration systems (BOS/BSS, for example), will not recognize an externally initiated node reboot as a trigger for a new image to be applied. In CSM 1.4 and above, BOS v2 works with Slurm (using SlurmctldParameters and RebootProgram) to inform BOS of a Slurm-initiated reboot, allowing the new image path to be fed to BSS.

Procedures

For initial changes and testing, we want a minimal set of nodes and on FE set aside for the new “blue” nodeset. This set will get new images/configs, a new DNS designation for the FE node (something like “cluster-fe-next”), a new Message of the Day (MOTD), and new “blue” labels. Some scheduler installations may also need to propagate a new configuration file to all of the nodes in the cluster, to make them aware of the new configuration (mentioned above). The Slurm submission command alias is also set to select the “blue” feature. Boot orchestration or configuration management is given new image or config tags.

Once the initial set is ready, its key function is to serve as a testbed. It may get multiple refreshes before the rollout can take place, but its basic function is to allow admins to verify the config changes, then pass the nodes to the software and environments team, who can verify that the system meets user needs. The new FE node can also serve as a build environment for advanced or priority users who want to do any necessary code rebuilds.

With validation in hand, the next step is to start checking application scaling. The number of FEs remains constant, for now, but the set of compute nodes expands to a reasonable testing number. Here reasonable depends on system scale, application scaling details, and user toleration of nodes removed from production.
Because we're now starting to test at scale, this is a great point to discuss automated testing frameworks. LANL uses a locally-developed Pavilion2 framework, and there is steady movement toward automated testing. Having automatic testing in place for this phase and subsequent pre-release phases is a goal for the automation that we hope to eventually achieve. In any case, successful testing with representative codes and with substantial MPI that encompasses all available nodes in the new "blue" set is the gating condition.

Once testing has given us the confidence we need, we can start rollout to the remaining majority of the cluster. At this point, we need to flip the majority of the FE nodes to the new image and flip the DNS entry of the first "blue" FE from cluster-fe-next to cluster-fe, and leave one FE as cluster-fe-previous, still in the previous config. MOTDs change on all FEs to indicate whether they are the past "green" configuration or the now current "blue" configuration, and any Slurm submission aliases or Lua scripts are set to submit to "blue". At this point, new jobs and new users will default to the new "blue" state.

This is also the point at which we trigger BOS and Slurm to work together. Nodes that complete jobs are designated for an ASAP reboot, and the reboot command triggers BOS to pick an image. Simultaneously, we flip the default BOS image to the new "blue" image if this is a configuration change with a reboot, so that the Slurm reboots pick up the new nodes. The nodes are also tagged with the "blue" feature — they won't get new jobs until after the reboot, so the tag is safe.
If the change is just a configuration, the nodes will simply be set to the drain state. As they get into the drained state, they can be selected for configuration, tagged "blue", and resumed.

As the cluster completes its "green" jobs, the last of the old configuration peters out and the cluster has been brought into a new state. The final FE is brought over to the new image (or left aside for future testing cycles) and the system change is complete.
If possible, however, it is useful to reserve some large set of nodes for a larger scale test during the rollout. This lets us detect issues that only arise at full scale before users hit them too much. One advantage to this approach is that the same procedure can allow us to roll back to "green" with the same mechanism. If there is an issue and the decision is made to roll back, we set the FE nodes back to "green" state, set the boot orchestration default or configuration back to the previous, set the already-configured

or rebooted nodes to configure or reboot ASAP, and the cluster immediately starts reverting. Retaining the small development pool allows debugging work to continue.

Caveats and Limitations

Clearly this rollout technique can't work in some circumstances. Major upgrades of Slurm, for example, might benefit from some full-system downtime, where jobs are drained and the database has time to upgrade. It is possible to push on with such upgrades without cluster interruption, but it may not be desirable.

More subtle issues may arise from the rolling upgrade of service nodes like Lnet routers and filesystem projection nodes. It is not straightforward to point a set of nodes at a particular service node and switch them during production without damaging existing mounts. Making arrangements for a doubled number of service nodes to allow them to be used by turns is expensive, but would allow for more reliable change testing.

There will also be times where the management plane needs major upgrade work that impacts user job operation or makes system stability unpredictable. If compute or service nodes require re-imaging during the upgrade, it is likely that the mechanisms the rollout requires could be in a degraded or down state. This increases the risk to user jobs, which a given site may find unacceptable.

There are few circumstances where a user job interruption is absolutely necessary, but the situations above may either increase risk past acceptable levels or require inordinate effort to avoid an outage.

Analysis

System Downtime

In terms of node hours (NHrs) spent unavailable, our baseline for an upgrade outage is the entire number of nodes times the number of hours spent on the outage. For example, a 1,000 node cluster that is down for an 8 hour upgrade, the node hours down would be 8,000 NHrs. On an extreme scale system like Trinity which had nearly 20,000 nodes, the same outage would be 160,000 NHrs.

If for some reason that upgrade ran long and caused the machine to be held overnight and then was released again after 4 more hours were spent on completing the upgrade (a total of 24+4 hours). The outage would end up costing 560,000 node hours. This is the equivalent of 4 years wasted for a 16-node departmental cluster.

Using our alternative method, requiring 1 node costs only 168 NHrs/week. Scale testing using 16 nodes costs 2,688 NHrs per week. The break-even point for scale testing at 100 nodes on a 1,000 node cluster (10% of the whole) would be 80 hours, or 3.33 days at 24 hrs/day.

Scale testing is certainly one area where lax attention to detail could end up costing more than a full system outage. However, there should be more than adequate time for system testing at reasonable scale.
One thing that should be noted, this system testing only includes administrative system tests. User testing is not included as it would be carried out in either case and will be addressed later in the 'Interrupted Science' section below.

Prolonged Downtime Risk
It could be said that the majority of prolonged downtimes occur in situations where administrative actions were neglected, overlooked, or just did not go as planned. When these arise during a full system outage, the remedy is usually modifying a script or some process and starting the entire, or at least a significant part of the upgrade over again. With admin work windows routinely approaching a full day's effort, unless the error was found early in the upgrade AND fixed without delay, the risk of pushing the outage overnight is significant.
If the problem cannot be addressed in short order, a roll-back could be required until a solution is found. If the problem was encountered early in the upgrade outage, the time lost would be the cost of the reboot into the new environment, scramble to decision and then time to reboot back to the old environment. 100% of this time is a complete waste of utilization. But further than that, it makes the group responsible for upgrades look bad. This is especially acute in the eyes of the users who had their submitted workload cancelled during the onset of the upgrade outage and now must figure out where their progress was halted and resubmit new workload to continue until the next upgrade outage.

Human Performance Under Pressure
At institutions where extreme scale machines exist, a single person rarely makes important and costly decisions alone. Heroic efforts have saved the day on numerous occasions, but are not reliable. Sometimes the stars align against the valiant system administrator where no amount of pre-examination or testing could have revealed the derailing problem. And other times, when a solution can be determined quickly, its implementation will be one that is a function of time and scale.
There is also a word of caution that needs to be mentioned that sometimes a quick solution is not necessarily a correct solution, and sometimes can be wrong with dire consequences. Having time to think through the implications of a course of action to address a problem is important — one might not be thorough enough in a short evaluation time. In fact, even with many eyes on the problem, “gotcha” cases may be overlooked.
Sharply reducing the risk of error under pressure should be considered a significant accomplishment, especially as the scale of modern supercomputers increases.

Interrupted Science

This area in particular may be one of the most impactful changes regarding this method, especially in terms of user experience. Scientists' time is also a very valuable and costly resource. In the prior upgrade full system outage method, users will have a day, or potentially more where they are unable to interact with the machine, and at the same time, none of their workload will be progressing. Furthermore, once the system once again becomes available, then and only then will they be able to start recompiling their codes and testing them. This can take days or even weeks worth of effort before they begin making progress in terms of science.

In our proposed method, the user can begin compiling and testing once the new environment UANs and compute node are made available. This happens in parallel to the continuation of their previously submitted workloads. A full scale system typically is not needed for users to begin their rebuild and test processes. The small number of nodes brought up in the new environment can provide them the opportunity to do so before the majority of compute nodes are scaled over to the new environment.

Cost Savings

With the new method of managing upgrades, the argument could be made to eliminate the TDS system all together. Since the TDS system is generally not a full scale testbed, it is relegated to the testing of functional things. This is also the case for the scale over method. However, when the upgrades are finished, the TDS hardware either sits idle or is used in some other experimentation. While this experimentation can also be seen as a benefit in certain situations, it might be the case that the TDS resources could be better utilized as part of the proper cluster rather than sitting idle. TDS systems are not cheap as they require a similar amount of infrastructure overhead but with a sparse compute node count.

Conclusion

System uptime, resiliency, performance, and reliability are all at stake whenever a modification takes place. Thanks to existing work on submission-time node reconfiguration, Shasta boot orchestration features, Slurm reboot functionality, and new work on retooling Slurm configuration files, LANL is implementing tools to roll out system changes live to active users without downtime. With rollout and rollback features, users can start using updated nodes as soon as they clear and reboot, but the risk of a broken system is mitigated — scaling back a faulty change even faster than scaling it out.

Further work on this tooling over the next year will integrate with the LANL shasta_wrapper toolkit and take many of the manual operations out of this process.