Optimisation of PBS Hooks on the Cray XC40

Sam Clarke
Met Office
Fitzroy Road
Exeter, UK
Email: sam.clarke@metoffice.gov.uk

***Abstract*—The Met Office, the UK's national weather agency, is both a global forecasting centre with a requirement to produce regular, timely weather forecasts, and a major centre for climate and weather science research. Each of these groups require access to a large supercomputing facility which is highly available, reliable, and which provides good turnaround to facilitate scientific development work.**

This paper describes some of the steps taken to address performance and queuing concerns at an HPC facility that typically runs 300,000 jobs every day, many of which require good turnaround. We detail our experiences with the performance of PBS under increasing levels of load and the impact of these on the organisation. We discuss some of the optimisation techniques we have developed to improve the performance of some of our locally written hook scripts and describe how we measured the impact of these changes on the organisation.

***Keywords*-Scheduling; PBS Pro**

I. Introduction

The Met Office is a global weather forecasting centre with a requirement to generate regular and timely weather forecasts. It is also a major centre for scientific research, with specialisations ranging from weather and ocean forecasting to climate research, including everything from fundamental atmospheric physics to ecosystem modelling to observation processing and data assimilation techniques.

All of these areas of research exhibit a high demand for supercomputer time and the Met Office has three Cray XC40 systems which it uses to satisfy the requirements of both operational weather forecasting and cutting-edge scientific research. Given the demand for supercomputer time in the organisation, it is important to ensure that the resources are used efficiently and that they are available to users in a prompt fashion — two requirements that are often in tension with each other.

II. About the Met Office Workload

The Met Office workload exhibits a number of characteristics which differentiate it from more typical scientific computing loads but which tend to be common to modern meteorological and climate applications. These include the time-critical nature of many of the jobs; the homogeneous nature of much of the work; and the high volume and short duration of many of the jobs.

All operational weather forecasts have time-critical components. They are required to start as late as possible in the day in order to ensure that the best quality of observations are available, but they must also start early enough to ensure that the processed output is available at a specific time. There should be as little variation between start and end times as possible between daily cycles, and run times should be as consistent and reliable as possible.

Each forecast suite is made up of a series of different tasks. Each task takes the form of a separate batch job with its own resource requirements. Some of these tasks are able to run alongside one another; others have simple dependencies on other tasks with other jobs in the suite; while some tasks have complex dependencies and cannot begin until another task has passed a specific point but without needing to wait for a task to finish.

In order to ensure timely performance of these jobs, it is necessary to use PBS reservations. These block out resources on the system, ensuring they are available to a given forecast run at the required time, effectively eliminating long queuing delays. Reservations are necessarily less efficient than the standard backfill scheduler because the nature of each forecast suite is such that some tasks, such as observation processing and data synchronisation, require far fewer resources than compute-intensive tasks such as variational assimilation or atmosphere modelling, meaning that compute resources are sometimes left idle.

Less obviously, much of the research work run on the system also has a time-dependent component.

Weather science research into next generation forecasting systems typically takes the form of a suite of very short jobs, some of which contain large parallel models while others are composed of wholly serial work. These jobs exhibit complex dependencies, where poor turnaround for one job in a cycle can result in extensive delays to the entire suite. And where a suite is being used to trial a new configuration prior to implementation in an operational forecast, an accumulation of these delays can slow down the pace of adoption of new features for end customers.

It is also important to ensure that routine model development — either speculative model changes or basic scientific research — does not spend excessive time queuing for resources. While this work is treated as lower priority,

poor turnaround of jobs may have a substantial impact on the amount of useful work individual scientists are able to carry out, resulting in general dissatisfaction with the supercomputer service.

A. The Rose Framework

The Met Office workload also demonstrates a high degree of homogeneity, with much of the work consisting of variations on a core set of tasks. For example, both operational forecasting work and next-generation weather modelling involve a combination of closely coupled and dependent jobs. Similarly, the climate studies which make up much of the rest of the workload typically involve long-scale coupled ocean-atmosphere model combinations which require large amounts of data to be transferred to archived storage by dependent jobs. Given this similarity, it makes sense to run the workload under a common framework which provides common tools and supports the sharing of jobs between users.

Almost all the work on the Met Office supercomputers runs under the Rose framework[1]. This allows tasks to be grouped together into suites and held in a database which facilitates sharing. It provides an abstract interface to the batch system which allows tasks to be submitted for execution on a supercomputer, and provides a series of tools which manage the execution of a job — monitoring it and, optionally, automatically resubmitting it in the event of a failure — as well as automatic job output processing.

Rose uses the cylc workflow engine[2] to manage dependencies between tasks in a suite. This makes it possible for users to build networks of tasks which result in different jobs being submitted depending on the status and output of a previous job. This behaviour means Rose favours breaking up suites of tasks in to large numbers of small jobs. It also means that much of the workload may be latent: when a job finishes, it may trigger Rose to submit a number of dependent jobs.

As a consequence of these traits, the Met Office systems execute an extremely large number of jobs while also having a relatively small number of jobs queued at any one time. For example, each of the systems typically runs 100,000 jobs a day with queue depths of around 300–400 being normal.

The jobs typically have a short dwell-time in the system, as measured by walltime reported by PBS. The average duration of a job on the large research system is around 280 seconds. The majority of the workload is limited by the queue configurations to a maximum wallclock of three hours, after which it must resubmit and restart from a model checkpoint; a very small number of collaboration jobs on the research system have a high restart overhead and allowed to run for up to 24-hours between resubmissions.

B. Hook Architecture in PBS

In addition to the overhead imposed by running a large number of jobs, each of the systems also runs a number of PBS hook scripts at different points in each job's life-cycle. These can be grouped into two main categories: submit hooks, which run when a job first arrives in the system; and execution hooks, which can be triggered before and after the execution of each job.

The submit hooks run on the PBS server and provide a way to impose local site policies, to perform job validation, and to set default resources. For example, the Met Office uses a hook script to impose some standard settings, such as umask values, with a second script to set compute node resources, and a third to confirm that a user is entitled to use the specified project. The hooks are grouped into separate scripts in order to allow new policies to be imposed and to allow other rules to be selectively deactivated.

The submit hooks do not pass control back to the user's `qsub` command until the entire set of hooks has completed. This allows a script to reject a job which fails to validate in a fashion that allows an indicative error message to be passed back to the job's owner.

The execution hooks run on a node where the job has been scheduled for execution. On the Cray XC40, this is a node with a `vntype` of either `cray_login` or, in the case of non-compute work, a node with a type of `cray_serial`[1]. A number of different life-cycle points are supported, allowing arbitrary commands to be triggered both before the start of the user's job script and after its completion. This makes it possible for a site to run number commands to customise the state of a particular node or set of compute nodes ahead of each job and to return the nodes to a pristine state at the end of each job. For example, the Met Office uses this facility to implement a soft partitioning scheme[3].

Unlike submit hooks, the time required to execute the execution hooks is counted again the job's wallclock time limit. This ensures that jobs can be guaranteed by a specific time, making it possible for the scheduler to run in backfill mode. However this also means that, where a user has requested a very short wallclock time limit, it is possible for the execution hooks to use so much time that the user script does not have time to complete.

In order to prevent problems with slow hooks, PBS supports the ability to set a time-out limit on each hook. This is set by the daemon responsible for running the hook and is implemented using an `alarm()` system call and, unless explicitly overridden, the time-out defaults to 30 seconds. Whenever a time-out occurs, the PBS daemon responsible for executing the hook logs a message which details both the name of the hook script and the hook point it was run from.

[1]Cray typically refer to `cray_login` nodes as MOM nodes and `cray_serial` nodes as MAMU nodes

III. Hook Performance Problems

As the workload on the systems increased, the number of hook time-out messages in the PBS MOM daemon logs began to increase. Initially, most of these errors occurred during the job epilogue phase and appeared to be harmless – at worst, they resulted in a failure to append job summary information to the user's output. However, as the complexity of the hook infrastructure increased — especially, following the introduction of both Altair's cgroups hook and our own trustzone facility — we came to realise that hook performance problems and time-outs were having a significant impact on our workload.

The first problem we noticed was that, where one of our site epilogue scripts would fail with time-out, other clean-up actions on the node would also fail. For example, where our trustzone hook had timed out waiting for an action to complete on a compute node, we noticed that the job summary also failed to be appended to the job. To our surprise, we also noticed that Altair's cgroup clean-up hook — which, unlike our site hooks was installed using Altair's internal `pbshook` facility — also failed to run.

The second problem we noticed was that user output was intermittently slow to stage out, with delays of several minutes observed in some cases. These delays had a double impact: firstly, by causing a second time-out to occur in the Rose framework, making it appear as though the job had failed; and secondly, by adding a number of minutes to the execution times of what where expected to be very short development and operational jobs.

The Rose problem was found to be caused by a secondary time-out in the framework. This occurred because, as the last action in every job, the framework sent a message to the Rose server indicating that the user script had completed and that the server should start polling for the job output. Control was then handed back to the PBS daemons, allowing the job output to be spooled to its output location on the Cray, but a time-out in one of the hooks run by the daemon was sufficient to exceed the Rose polling time-out, causing the remote server to abandon its attempts to stage the output.

IV. Performance Analysis

As part of a previous effort to characterise the performance of our hook scripts, we had created a python decorator which could be wrapped around the function of each script. Initially created to capture tracebacks and write them to the PBS daemon logs, this also reported a short message at the end of each hook indicating its total duration. These completion messages provided a good starting point for evaluating hook performance, although we found it necessary to add the trigger point to the log message to differentiate scripts which were run both as job events and as periodic hooks.

An examination of the performance data from the decorator showed a slight increase in some of the hooks, but on average this was not sufficient to cause the hook to experience a time-out. Rather, the the data showed a substantial increase in hook time-out events clustered in time, but spread across a number of different scripts and across a range of nodes.

The PBS daemon logs also revealed another puzzle. We expected to see hook time-outs in the cases where users had reported problems with their output not being staged back to Rose. We did not see this.

Rather, we saw evidence of the job running normally up to the point of termination. We then saw a message from the decorator indicating that it had started to run another hook. We subsequently observed some indications of activity in the log, but nothing associated with any of the jobs in the system — neither the job running the hook nor the job being terminated. Eventually, we saw a time-out for the original hook event followed by a flurry of activity in the logs, including various stage-out tasks associated with the spooling back of job output and the reaping of child processes associated with the batch jobs.

A process listing of the system while this type of hiatus was occurring showed a python process associated with the on-going hook and various zombie processes associated with other jobs which were in the process of being terminated and existing batch jobs continued to run as normal. This appeared to rule out a problem on the executing host as the underlying cause.

A. Characterisation of the Problem

A closer inspection of the hooks which experienced time-outs revealed that failures were far more common in scripts which requested configuration data from the PBS server.

For example, the trustzone partitioning hook needs to know whether a job is running in a serial queue in order to determine which provide to apply when it begins executing. This is achieved by calling a locally written python library function which: accepts the PBS job as an argument; extracts the name of the queue or, where the queue attribute is not set, queries the PBS server for the name of the default queue; and queries the PBS server using the queue name obtained in the previous step. The queue attributes obtained from the server are returned to the caller, allowing the original hook to make its decision about whether to apply serial or compute configuration changes.

This pattern proved to be typical of the hooks which experienced problems. A library routine would be called which, hidden from the top-level script, would query the PBS server — sometimes, as in the case of the queue query function, multiple times — with one of these calls blocking until the script hit its time-out.

In combination with the clustering of time-outs at particular times but across multiple nodes in the systems, we were able to conclude that the problem was caused by the responsiveness of the PBS server. It seemed likely that the

problem was caused by the high volume of jobs in the system bombarding the server with a large number of trivial requests for attributes.

V. Hook Optimisations

Having identified what we believed to be the underlying cause of the problem, we began to replace a number of the functions used by our hook scripts with versions intended to remove much of the load from the PBS server. We were able to do this relatively quickly because almost all of the most common server queries were contained within library functions, making it possible to install a new version with a simple restart of the PBS daemons to force a reload.

A. Implementation of Caching

Our first action was to create a pair of new low-level libraries functions called `readCachedVariable()` and `writeCachedVariable()`, both of which accept the name of a PBS server attribute and, optionally, a path to a cache directory and an integer indicating the time-to-live in seconds of the cache.

When an attempt is made to query a server variable, the higher level routine calls `readCachedVariable()` which attempts to obtain the value from a file stored on the node-local `/var` file system. If the file cannot be found or if the age of the file exceeds the time-to-live, an exception is raised, otherwise the query is satisfied using the contents of the file. Where the attribute is expected to contain multiple values, the function can be supplied with an optional argument which causes it to automatically convert the stored string into a python list.

In the event of a problem with the cached file, a custom exception is raised, allowing the caller to catch and handle it. This allows the caller to query the PBS server for the attribute using the appropriate calls. This cannot be done by the underlying `writeCachedVariable()` routine because the exact resource call varies depending on the attribute being queried — for example, querying the attributes of a queue is different to querying those of the server. However, once the target attributes have been obtained, they can be written to the cache with the `writeCachedVariable()` routine.

This approach yields particular dividends when dealing with common tasks, such as determining whether a particular queue is for serial or compute node work. The process of querying the server for the attributes of the queue only needs to occur once on every node, once in every time-to-live period, and the subsequent processing only needs to take place once before the values can be cached. This eliminates both a PBS server call and much of the subsequent processing needed to generate an answer to the calling routine's question.

B. Time-to-Live Problems

Care must be taken when selecting a time-to-live, because although many server attributes are static and change rarely, some change more rapidly and this may result in unusual errors until the situation is correctly understood as a divergence between the actual configuration and the cached settings.

For example, when we implemented caching for queue settings, everything worked as expected until we experienced a problem with short-notice reservations.

Under normal circumstances, all operational reservations are created at least six hours in advance of their start time. This was found to give the system sufficient lead time to ensure the resources were available to forecast jobs without placing additional load on the scheduler due to the need to place work around reservations with start times that are more than 24 hours away. This lead time was also sufficient to ensure that the cache was populated with information about the reservation, given that the initial time-to-live on the cache was set at 1800 seconds.

However, when a small reservation was created with only a few minutes notice to allow Cray to run diagnostics on a suspect node, all attempts to submit jobs to the reservation failed, with the error message indicating that the destination queue did not exist. The problem was traced to a difference between the cache — which lacked an entry for the reservation — and the actual configuration of the server.

The problem was eventually resolved by adding an explicit rule for reservations to the library functions which handle queue attributes. By adding a check to determine whether the given queue name matches the pattern of a reservation, we were able to add a block of code to the function which explicitly bypasses the cache where the check is true. This reduces the efficiency of the function slightly, but gives greater reliability — something that is particularly important in the case of reservation handling, because these are used extensively by operational forecast work.

C. Cache Location Problems

A second problem encountered during implementation was restricted to submit hooks and concerned the use of cache files for queries which worked successfully when run in the hooks on the executing nodes, but which returned inconsistent information when run on the PBS server.

The hook used to implement the trustzone method of soft partitioning runs as both a submit hook and an execjob hook. During the submit phase, it determines the zone of particular job using the system where the `qsub` command has been run by querying the PBS server for an attribute associated with current node. The same method is used when the hook is run at the start and end of each job, using the executing host as a target rather than the host where the job was submitted.

All the zone queries were updated to use the caching method. This worked well during the job prologue and epilogue phases, where the cache information was saved in a file in the `/var/spool/PBS` directory which was unique to each executing node. However during the submit phase, the same method was used to write zone information to the `/var/spool/PBS` directory on the SDB — a location that was globally visible across the entire PBS complex. This manifested itself as an alternating problem with incorrectly identified zones, with the cached copy being updated with the zone of the first job to submit after the time-to-live had expired, causing some jobs to be incorrectly rejected.

Once the problem was identified, some of the caching library routines were updated to make them safe for use on the SDB. This essentially involved bypassing caching for calls which accepted the executing host as an argument.

D. Caching Using Environment Variables

In addition to caching server information in files, an attempt was made to reduce the number of server queries in the execution hooks by moving values specified as job attributes into environments. This was achieved by modifying the submit hooks, which already have access to the job and its attributes as part of the submit event, and copying their values into specific environment variables which can then be passed through to subsequent events.

For example, one commonly used hook creates a memory-resident file system as the job starts and removes it when the job exits. The user requests the file system by adding a resource to their job at submit time which specifies the amount of space required. This attribute is picked up by one of the submit hooks and copied into an environment variable.

When the job is dispatched to the executing node, the prologue hook runs ahead of the user's job script and checks for the presence of the environment variable. If it exists, the prologue hook creates a file system of the specified size, mounts it, sets the user's `TMPDIR` environment variable to point to it and completes. Similarly, at job shut-down, the same hook runs during the epilogue phase and, if the environment variable is set, unmounts the memory-resident file system.

This change, while small scale, eliminates two calls to the PBS server for every job that runs. And although the change makes use of an environment variable, the value can be used to communicate safely between hook scripts because both the prologue and epilogue hooks are run by the PBS MOM daemon on the node — the process that is also the parent of the user's job script — which prevents change from the user's environment from being propagated back and picked up by the hook script.

VI. Removal of Estimated Start Times

In addition to making changes to the hook architecture to minimise the load on the server caused by job execution, we also chose to disable the process of job start time estimations. These are generated by a secondary process which runs alongside the main PBS server and scheduler daemons to generate exact start times and placement information by duplicating the steps of the scheduling algorithm for every queued job.

In order to generate start times and placements, the estimator task needs to query the server for a complete listing of both all the jobs in the system and all the nodes in the inventory. This places a substantial load on the server every time the task runs. And because the estimator needs to determine the start time of each job, it has to carry out the computationally expensive exercise of calendaring every job, leading to cycle times that are heavily extended relative to the core PBS scheduler.

We found that, on our larger systems, the estimator often took in excess of 15 minutes to complete; in comparison, the job scheduler was able to dispatch the queued work in around 60 seconds. This meant that the information produced by the estimator was often some minutes out of date, which, given that much of our workload consists of small, short jobs with a low dwell time in the system, meant that estimated times were only of use with a minority of the workload — usually larger, longer jobs assigned to a lower priority queue. This made it possible to disable estimated start times with almost no impact on end users.

VII. Performance Assessment

Following the implementation of the hook changes to enable caching, we saw a clear improvement in the performance times reported by the decorator. These improvements were reflected across a number of hooks, with the degree of speed-up largely dependent on the number of PBS server queries issued by each hook.

The trustzone hook, which makes a number of server queries, showed markedly better performance in both its submit and job execution phases. The elapsed time of the submit hook dropped by an order of magnitude. This did not have a substantial impact on overall performance — the initial, unoptimised performance of the hook averaged around 0.01 seconds — but it demonstrated the effectiveness of the approach.

The job execution portion of the trustzone hook showed a three-fold improvement on the `cray_login` nodes, with much of the remaining time attributable to the need to run remote commands on the compute nodes. On the `cray_serial` nodes, the trustzone hook execution times dropped from around one second to 0.03 of a second on average.

The memory resident file system hook clearly shows the benefits of using environment variables to hold persistent hook data, rather than job attributes. Prior to optimisation, the hook took a third of a second to execute on average. Following the optimisation, elapsed times dropped to close

to zero — effectively just the cost of a python call to check whether an environment variable was set.

In addition to improving the performance of individual hooks, the optimisations also resulted in a substantial decrease in the number of hook time-outs across the system, with the largest of the three machines showing the greatest improvement. Following the change, the daily total of hook time-outs across the nodes dropped from 1–2 failures an hour to 1–2 failures per day.

Combined, the improvements to the performances of individual hooks and the reduction in the number of time-outs resulted in an overall improvement in the rate at which PBS was able to process completed jobs. This has resulted in a 10 second reduction in the average time required for a job to shut down — that is, to move from the termination of the user's job script, through the spooling back of the job output, to the point where all the processes have terminated and, in the case of a parallel job, the ALPS resources have been released.

This improvement is sufficient to allow the majority of jobs to complete within the window of opportunity available to Rose to stage the output to a remote server. This has resulted in a perceived improvement to the reliability of the system and to the turnaround of user development work, largely because short jobs now complete more quickly and output is available immediately from within the Rose user interface.

VIII. Discoveries

In the process of investigating the performance of hook scripts on the system and, especially, while investigating the problems caused by hook time-out events, we made a number of useful discoveries about the way PBS hooks are implemented and how hook errors are handled.

A. Hook Ordering Matters

PBS hooks are executed in sequence based on their order attribute, which can be set with `qmgr` and defaults to one if not specified. Hooks can only have a single order value, which means that if the same hook is triggered from a number of different events and there are many scripts attached to each event, it is difficult to force the hook to execute early in one sequence and late in another.

In general, where a hook needs to be executed at different points in event sequences, it is best to create a new hook and use it to install the same script as the original event. This increases the complexity of managing a hook, because it is possible for the different events to be running different versions of the script, but provides greater flexibility and reliability when ordering is important.

B. Time-outs Are Fatal

When PBS encounters a time-out while running a sequence of hooks, it stops and does not execute any remaining scripts associated with the event. This means that hooks which are important to the health of the system should be assigned an order that ensures they are run ahead of less essential tasks. Ideally, important hooks should also include a periodic mode which can be used to catch situations where the clean-up phase of an epilogue has failed to run.

C. PBS Hooks Are Not Special

Some hooks, such as Altair's cgroup hook and their `PBS_alps_inventory_check`, are treated differently to site scripts and have different priority ranges. They cannot be printed with `qmgr`, although they can be displayed using the command "`print pbshook`". This gives the impression that PBS has a set of internal system hooks which exist separately from any local site hooks, but this is not the case: all the hooks execute from the same set of events and use the same ordering.

In combination with the previous two points, this means that a time-out in a low-ordered site hook can cause a higher-ordered PBS script to fail. This has potential consequences for the maintainability of the system where the PBS scripts are used to perform important system tasks.

D. Hooks and Daemons are Synchronous

While investigating problems with jobs taking a long time to stage their output, we noticed that the problems often affected a large number of jobs on a single node, even when the job was not running a hook and did not experience a hook time-out.

An examination of the open source version of PBS[4] revealed the source of the problem: with the exception of periodic hooks, the flow of execution in a PBS MOM daemon is synchronous. This means that if a MOM daemon has triggered a hook script, it has to wait for the hook to complete before handling the next event. Where PBS only has to manage a small number of jobs on a particular host or where the jobs have a long runtime and are not time sensitive, the impact of this is limited.

In the case of the Met Office, where many of the jobs have very short run times and where serial and login nodes can be responsible for managing tens of jobs at any one time, the impact of a PBS daemon stall due to a slow hook can have a disproportionate impact by causing processing to stop for as long as it takes for the hook to time-out. This was found to be the root cause of the problem which prevented user output from being copied back to the Rose server.

IX. Conclusions

While the core of PBS Pro scales well and is capable of running many thousands of jobs per hour, it is possible for poorly optimised site hook scripts to place a great deal of stress on the server as the job load increases.

This load on the server can result in decreased throughput, as individual hook scripts take longer than expected to

complete. In the worst case, due to the architecture of the Cray XC40 and the synchronous nature of the PBS MOM daemons, it is possible for the slow performance of the server to cause a script to time-out, and it is possible for each time-out to result in a delay to other jobs running on the node where the time-out has occurred.

We have shown how a python decorator can be used to log useful performance information about each invocation a hook script. We have shown how this information was used to guide the process of finding instances of poorly performing hook scripts and how judicious use of caching can be used to improve the performance of both individual hooks and the entirety of the PBS complex.

We have detailed some of the difficulties we encountered with our optimisations, and described some of the features we encountered in PBS Pro's hook architecture as we worked to identify the root causes of some of the problems we observed — where we found the PBS Pro open source project an invaluable guide.

REFERENCES

[1] Rose - a framework for running meteorological suites. [Online]. Available: https://www.metoffice.gov.uk/research/modelling-systems/rose

[2] Cylc - a workflow engine. [Online]. Available: https://cylc.github.io/

[3] S. Clarke, "Trust separation on the cray xc40 using pbs pro," *Concurrency and Computation: Practice and Experience*, vol. 30, no. 1, 2018.

[4] PBS Professional Open Source Project. [Online]. Available: https://www.pbspro.org/