Preparing Slurm for use on the Cray XC30

Stephen Trofinoff and Colin McMurtrie
CSCS – Swiss National Supercomputing Centre
Lugano, Switzerland
Email: {trofinoff; colin}@cscs.ch

***Abstract*—In this paper we describe the technical details associated with the preparation of Slurm for use on the 12 cabinet XC30 system installed at the Swiss National Supercomputing Centre (CSCS). The system comprises internal and external login nodes and a new ALPS/BASIL version so a number of technical challenges needed to be overcome in order to have Slurm working on the system. Thanks to a Cray-supplied emulator of the system interface, work was possible ahead of delivery and this eased the installation when the system arrived. However some problems were encountered and their identification and resolution is described in detail. We also provide detail of the work done to improve the Slurm task affinity bindings on a general-purpose Linux cluster so that they, as closely as possible, match the Cray bindings, thereby providing our users with some degree of consistency in application behaviour between these systems.**

***Keywords*-Slurm; ALPS/BASIL 1.3; SPANK plugin; Task Affinity; esLogin; nppcu; QUERY(SUMMARY)**

I. Introduction

Slurm [1], a robust open-source resource manager, is growing widely popular in the HPC sector to the extent that some 30% of the Top500 systems now run Slurm (including 5 of the top 15 systems). CSCS has a long history in Slurm development, having written the very first port to Cray systems back in 2010, for a new XE6 delivered to the Centre in the the middle of that year [2]. Slurm is now used site-wide at CSCS, across a wide selection of systems including the flagship XE6 system and the production MeteoCH machines. The release of the Cray XC30 and CSCS' intention to purchase such a system raised the opportunity for further work on Slurm in order to port it to this new architecture. First, there was the basic question of if and how Slurm would work on the XC30. Then there was the question of whether Slurm could make use of any new features provided by Crays low-level system software. Finally, we examined ways to improve the robustness and functionality of Slurm, including on some of the smaller non-Cray clusters.

II. Port of SLURM to the XC30

A. Basic BASIL v1.2 Interface

On Cray systems Slurm, as with all third-party resource managers, is currently layered on top of Cray's own primitive resource management software, namely the Application Level Placement Scheduler (ALPS). Communication between ALPS and any third-party resource manager is handled via the Batch Application Scheduler Interface Layer (BASIL) which provides an XML-based interface for such. In the past Slurm was written for use with earlier versions of BASIL (e.g. version 1.2) on CSCS' various Cray systems such as the XT5, XE6, XK6 and XK7. Beginning with the XC30, BASIL had been enhanced to version 1.3. Fortunately, due to the continued backwards compatibility of BASIL, the existing v2.5 Slurm code base needed only a very minor update (by the addition of the new version number of ALPS/BASIL) in order to work.

At this point, Slurm was essentially working on the new 2256-node XC30 system. However a peculiar issue became exposed, namely that jobs were somehow limited to a size of 2047 nodes. The problem was discovered when running a test job in which over 2100 nodes was allocated. Strangely it was observed that from within the allocation a simple aprun using only a few processes worked. However, when attempting to execute aprun using the full allocation, ALPS reported that the request had exceeded the allocation. Checking the Slurm job allocation and the ALPS reservation identified that ALPS reported it had only a small subset of the nodes originally requested while Slurm was erroneously reporting that it had all of them. Further investigation revealed that the number of nodes in the ALPS reservation was exactly N mod 2047, where N is the number of nodes requested. It should be noted here that each node on the system in question contains 32 processors. Knowing that and the limit at which the problem occurs, it was observed that this problem was appearing at a power-of-two boundary, notably 65,536. This, of course, happens to be the limit on a 16-bit integer.

Given that ALPS/BASIL was able to allocate more than 2047 nodes per job when issuing the XML request directly (without the resource manager), it was suspected that Slurm had passed some faulty information in its request. Indeed, our experience has shown that the first place to start when debugging problems of this type on Cray systems is to analyse the coherency of the XML passing between Slurm and `apbasil`. In the past this was a tedious process so we developed a convenient debugging feature, included in all versions since v2.4.0-pre4, which prints all outgoing XML messages from Slurm. Debugging can be turned on by setting the environment variable `XML_LOG` to yes. With this set, the output is written to the `slurmctld.log` but this can be changed by using the XML_LOG_LOC variable to specify a different file.

Hence, with XML debugging turned on, it was easy to confirm our suspicions that Slurm was indeed passing erroneous XML data to apbasil. For example, for a job request of 2050 nodes, Slurm would specify the complete list of nodes that it wanted correctly, but would state the number of nodes to be only three.

Investigation of the code revealed that the value used by Slurm for the count of the number of nodes was based upon a value representing the sum of the processors on each of the requested nodes and that this variable was declared as a 16-bit type. Therefore, the problem can be more correctly described as being a function of the number of requested processors and, hence, if each node contained only 16 processors, the node limit would have been reached at a higher node count and conversely if each node contained 48 processors it would have been reached at a lower node count. Apparently this was a throwback to when the Cray systems using Slurm were smaller and this variable had been sufficient to hold the maximum number of processors available on those systems. With the simple change to a larger type, the problem was resolved and this patch is included in Slurm from v2.5.5.

B. New BASIL v1.3 Interface

With Slurm v2.5 successfully working on the XC30 using the pre-existing BASIL 1.2 protocol, the next phase of the work was to modify Slurm to utilize the new BASIL 1.3 protocol instead. This work involved essentially two types of modifications. The first was to simply modify the parts of the existing Slurm code-base necessary to communicate with BASIL using the 1.3 protocol. The second, and more involved set of modifications, is to enhance Slurm to take advantage of any new BASIL features that could be of use. This means first identifying which features are relevant to Slurm.

The first major change to the BASIL interface, which impacts the first phase of this work, was the restructuring of the XML element hierarchy on the QUERY(INVENTORY) response [3]. With version 1.3, two new components were inserted, namely sockets and compute units. In turn each of these has two element types, namely a basic element type and then an array of that type: Socket, SocketArray, ComputeUnit, and ComputeUnitArray. Essentially, these changes provide extra, finer-grained detail on the hardware layout. The ComputeUnit logically consists of processors and there can be one or more ComputeUnits per segment. Therefore, the ComputeUnitArray is now the direct child of a segment and the ProcessorArray element has been moved down the hierarchy to be a direct child of the ComputeUnit.

The introduction of ComputeUnits to the hierarchy also came with the addition of a new attribute to the RESERVE method's ReserveParam, nppcu, which represents the *number of processors per compute unit*. The intuitive meaning of this new attribute is that the selected compute units must have at least this number of processors. A value of 0 means to ignore this and thus allow any compute unit to be eligible, which is the default if no value is specified.

In the initial modification of Slurm for BASIL 1.3, this new attribute was added but was always assigned the default value of 0. Thus, Slurm had it as a placeholder for future use and in fact we implemented it as part of the second phase of work on BASIL 1.3 support (more on this below).

Furthermore, sockets consist of segments and there can be one or more sockets per node. Therefore, the SegmentArray has been moved down from the Node element to the Socket element. The SocketArray is now the child element of the Node element. The addition of Socket elements also comes with the placement of the architecture and clock_mhz attributes on this level. The values provided here apply to all Processor elements contained therein (thereby reducing the message size somewhat). According to BASIL 1.3 documentation [3], a consequence of there now being multiple segment arrays per node is that with reservation requests we must compute the segment's absolute (per node) id, as it otherwise would not necessarily be unique. This change, however, does not affect Slurm as it never specifies segments in its reservation requests.

C. New BASIL v1.3 Features

With the basic BASIL 1.3 support completed, it was then time to explore the second phase, namely enhancing Slurm to take advantage of new BASIL v1.3 features. There appeared to be two further major changes to the BASIL interface.

1) Introduction of nppcu*:* One of these features, as previously described, was the introduction of the nppcu (i.e. the number of processors per compute unit) attribute in the QUERY(RESERVE) method. The basic idea is that through the use of this option, the user, via the resource management system, can specify how many of the processors on each core that ALPS should consider when scheduling jobs. For reasons of backwards compatibility, Cray made the default 0 which is a flag stating to use all processors. This option corresponds to the -j0 option of aprun. Synchronisation of the behaviour of Slurm and ALPS with respect to this new feature was specifically seen as advantageous for the wider user community and hence we were requested to implement the new functionality.

A number of changes were made to fully realize this feature. First, the do_basil_reserve function in basil_interface.c (select/cray plugin) had to be modified to check whether the slurm.conf option, CR_ONE_TASK_PER_CORE was used. If it were, we would now set the value of nppcu in the QUERY(RESERVE) request to 1. In addition, do_basil_reserve was modified to check to see whether the user had used the job submission option ntasks-per-core. If this value was used, it was to take precedence over the potential use of CR_ONE_TASK_PER_CORE in the slurm.conf. These first two changes were relatively simple and straightforward, the addition of a couple new variables and if-statements along with the passing of the value down the function call tree to the appropriate functions and making minor adjustments to those functions to use this new value.

This was all fine for merely setting the XML tag of nppcu=... to some specific value; however, this then had ramifications and thus started a process of finding and fixing collateral issues. The first such issue was that Slurm must pass the mppwidth as part of the reservation request to ALPS/BASIL. This value, corresponds to the number of processors in the overall allocation. Since Slurm computes this value by simply going through each node of the allocation and summing up the number of CPU's it has and now that the number of processors available per node had been reduced, two things had to change. First, in the loop that performs the summation, a few lines of code had to be added to adjust the value reported for the number of processors available on the node to reflect what was "visible" due to use of the the nppcu option. Second, the spot in the Slurm code where the number of nodes, itself, is determined had to be changed. In other words, both the correct number of processors per node and the correct number of nodes, itself, both had to be computed. If either of these items were not corrected, one or more values would be incorrect. Either the total width would be wrong and/or the number of nodes. One such case would arise if the user had specified some number of tasks, via the --ntasks option. Slurm would compute the number of nodes based upon this and its knowledge of the node layout. However, in this case it would think it had more processors and would thus need less nodes; thereby, creating a shorter list of nodes. Yet, the nppcu option would now notify ALPS of the correct view of the nodes and hence the two would not match.

Another problem that arose due to the use of the nppcu value came to be known internally as the *Malformed Job Problem*. This was the case where the user had submitted a job that was requesting a number of processors per node somewhere between the actual absolute maximum of the node and what was actually visible due to the use of nppcu. As its value wasn't above the real maximum of the node, it wasn't considered an error by Slurm but because it was greater than the value that could be seen (due to the nppcu value), this job couldn't run. When Slurm scheduled the job to run and therefore submitted to ALPS, ALPS would correctly report an error and then Slurm would get caught in a perpetual cycle of trying to periodically schedule the job. This in turn would cause the backfill scheduler to grind to a halt and no jobs would be run.

The solution to this problem was to find the spots in the code where Slurm had performed the basic "runnable" tests and again, just as before, adjust some of the values being used according to the nppcu value. It involved tracking down in the code not only where Slurm computes what it believes is the number of nodes required by the job but also adding various utility functions to return the correct value to use for several related items such as the number of hardware threads or the number of tasks per core, etc.

The _job_test and _job_test_topo functions are two of the key spots in Slurm where it computes some of these values. These functions are specific to the node selection plugin being used. As the Slurm plugin for Cray systems is layered on top of the select/linear plugin, it was the version of these functions in that plugin that required modification. Here, wherever there were calls to _get_total_cpus, these got expanded to include the use of three other adjustment functions in order to correctly compute the value required. Still, other changes were made. For instance, the function slurm_get_avail_procs was completely rewritten as it was found to apparently contain almost completely unused code. What code was used did not compute what was needed and thus it was greatly simplified and new computations were added to it.

Yet another side-effect of the implementation of nppcu, was that squeue would not report the correct value for the number of nodes for pending jobs. Given that job priority favours large jobs, there was some concern that, possibly, this could mean that the job's priority could be affected. However, simple tests quickly proved that this was not the case and rather it was just a "cosmetic" bug. The specific symptom of the problem was that for pending jobs that were using an nppcu value of 1 (on our system where there are two processors per core), we would see exactly half the number of nodes expected. As could be deduced from the fact that this ratio is precisely the same as the number of processors visible versus the total number on the node, this was clearly a result of the new nppcu functionality.

When executed, the squeue command issues a request to the Slurm controller for information on the job or set of jobs that it was asked for. Amongst the information is the number of processors of the job and the number of allocated nodes. As no nodes are allocated yet for a pending job, Slurm estimates what this should be based upon the number of processors to be used (this value having already been set). It then performs some basic arithmetic based upon this value and the largest number of processors per node of any node in the system (which for our purposes was always the same). Obviously, this came down to the same problem as before where the total value of processors per node needed to be adjusted. Thus, another patch was made to fix this.

All the above functionality, including the various patches, will be in Slurm v2.6.

2) Investigation of the new QUERY(SUMMARY) *feature:* The second new feature in BASIL 1.3 we examined was the new QUERY(SUMMARY) request. The purpose of this new request is to try to reduce the amount of overhead incurred by constantly calling QUERY(INVENTORY) when only a few items within that query's response may have changed. As deadlines were rapidly approaching and there were several other significant work items to handle, only a cursory and very preliminary look was taken of this new feature, with mixed results.

3) Implementation Details: First and foremost it had to be decided how to implement the use of the new feature within Slurm. Cray's suggested use model is essentially to only perform a full inventory when necessary, that is, first perform the INVENTORY and thereafter only perform SUMMARY requests unless some non-scheduler changes have occurred. That is, there is a change in the difference between the changecount and schedchangecount attributes.

However, this immediately raises some problems for its use within the existing Slurm code base because when Slurm calls the QUERY(INVENTORY) method, it uses not only the node information (regarding which nodes are up and which are down) but also the job reservation information. With the latter information Slurm compares those jobs it has a record of to those reported by ALPS. If Slurm detects that there are ALPS job reservations for which there are no corresponding records in Slurm, these are termed *orphan* jobs and Slurm requests their release in order to free up their resources for legitimate jobs to use. Although not a frequent occurrence, orphan jobs do happen often enough to warrant some level of synchronization between Slurm and ALPS and, without this, there could certainly be situations where legitimate jobs are held up from being scheduled due to resources being erroneously marked as allocated by ALPS. Hence we felt that a good resource manager should be able to detect this situation and free the resources for reuse. At the same time, we still wanted to make a "quick try" of this new feature. Therefore, we opted for a hybrid approach where we would still occasionally call the INVENTORY method but would replace a subset of those calls with the SUMMARY method.

With the hybrid approach in mind, the first thing to do was to analyze the existing code to see what is calling the INVENTORY method and what sort of data Slurm is extracting from it. The function that issues the INVENTORY request in Slurm is get_full_inventory. The call tree, mapping out all of the different paths to this function, is a bit complex. However, tracing showed that the two most frequent paths to it were through the functions _attempt_backfill and schedule. The former is called during the process of scheduling a job and the latter is a periodic check that Slurm performs to keep in sync with the ALPS system.

It was concluded that the logical place then to try to retrofit the QUERY(SUMMARY) would be in the schedule call-path as Slurm should certainly continue to check the state of the ALPS reservations before actually trying to backfill jobs. Here even, choices exist as to how it should be done. For instance, one possibility was to call QUERY(SUMMARY) for x number of times and then follow it with the next query, namely the traditional INVENTORY, and then repeating x number of SUMMARY calls. In this way, Slurm could still check the state of the ALPS reservations versus that of its own but simply would not perform it as often, thereby saving time. In the end, for simplicity and since this was just a quick proof-of-concept, the call path (from schedule) was modified to always call the SUMMARY method and if, in the rare event as Cray described, it found there was a state change then it would call the INVENTORY method. This seemed to be both the simpler approach and whatever gains it would have should be more significant.

The slightly tricky part at this juncture was to have a way to distinguish between the different callers. As mentioned, the call tree is a bit convoluted and between these "callers" and the function of interest lay several layers of functions including one "indirect" function call via a plugin (Slurm plugins are implemented as calls to functions within dynamically loaded libraries). There are various ways of handling this and perhaps a preferred way. Indeed, if this were to be implemented in production and had adequate time for development and testing, it would probably be best to split the schedule function call-tree completely out into a separate tree so there would be no need to make any distinction at all. However, in the interests of expediency and simplicity, it was decided to retain the existing call-tree. In order to do this and in order to not have to modify numerous function interfaces, we used the crude and simple technique of having a global variable that each "caller" function would apply a particular mask to. When the target function would be reached, it would be able to check this variable to see which function (or functions) had called it. Thus if the schedule function was determined to be the caller, it would call the QUERY(SUMMARY) method instead of the QUERY(INVENTORY).

Another tricky point for integrating this new query with the existing Slurm code is the addition of a new XML element with a tag of *Accelerator* in the response to the QUERY(SUMMARY). A tag with an identical name already exists in the response to the QUERY(INVENTORY) method but at a different level of the XML tree. The current code is designed to perform a simple lookup of a tag in a table; thus, the first entry would be hit and not the second. Each of these table entries has amongst its data, a field called depth which corresponds to the depth, in the XML tree, at which the given tag is expected. The code then uses this to see if the tag has occurred at the correct level. If not, it the XML response is considered faulty and an error ensues. Thus, a basic hack had to be put into place to circumvent this test for this given tag. This included having a variable set (another static global) upon encountering of the `AcceleratorSummary` tag. This would indicate that upon encountering the next Accelerator tag, that it would be the second type of accelerator.

4) Testing: Once the code was modified to use the new query, we wanted to test it somehow. Again, due to time constraints and workload, we did not create a comprehensive nor thorough test suite. We simply timed how long each of the BASIL queries took to obtain a very rough idea of relative performance.

This implied adding timers to the code so that we could obtain the required information. In this respect, the structure of the existing code was very beneficial as there was a single spot where all communication to/from BASIL occurs. Hence, we could place timers in this location and receive timing information for all BASIL methods. Furthermore, as Slurm already takes timings for other purposes and in other parts of the code we were able to use this existing code for our purposes.

In Slurm, there are three preprocessor aliases defined that one uses for timings:

- `DEF_TIMER`
- `START_TIMER`
- `END_TIMER`

The first simply defines the appropriate timeval variables, which is a structure from the standard `sys/time.h` header, and a string. The `START_TIMER` just calls the standard `gettimeofday` function with the first variable. The `END_TIMER` does the same with second variable but also calls a Slurm-defined utility function that finds the difference of the two time variables and creates the appropriate string representing this. Thus, the usage model is simply to place the `DEF_TIMER` where needed and in this case we placed it at the top of the function handling all BASIL requests. Then we placed the `START_TIMER` before any of the work of the function was started and the `END_TIMER` at the end, thereby timing the entire process of writing out the request and receiving and parsing the response. The resultant timer string along with the query type was then written to the standard Slurm log-files for later retrieval and analysis.

The resultant data consisted of a set of times for each method which was then averaged to find the method's average performance. Had there been more time available it would have been beneficial to define more thorough test suites and to work out a few peculiarities in the gathering of the information, but even so we can obtain an idea of the relative performance of each method in a few different situations. Specifically, we used basic tests included submitting large numbers of jobs (from between ∼500 and ∼1500) of various sizes (i.e. numbers of nodes) and wall-clock times on both a small and large system. The small system consists of a mere eight nodes whereas the large system consists of 2256 nodes. A simple script was used to help generate a set of tests with sizes varying from a single node up through the full size of the system with varied job wall-clock times. The tests on the larger system took place while other users were trying to run various application-level tests as well. Hence we had to keep the wall-clock times of each job very short (less than a minute) so one caveat is that this may not have been the ideal way to test how backfilling in Slurm would change with the use of the `SUMMARY` method; that itself would be another future test to perform.

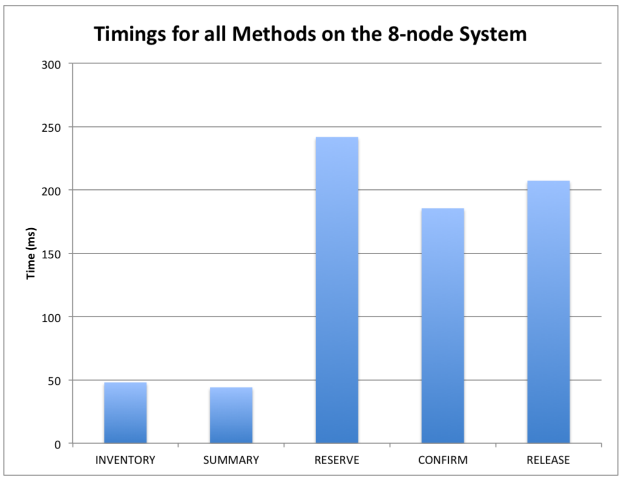

Figure 1. Method timings for 8-node TDS.

As can be seen from Figure 1, on the smaller system performance gains seemed relatively negligible. The `INVENTORY` method on average appears to take roughly 48 milliseconds whereas the `SUMMARY` took 44 milliseconds. Thus, the `SUMMARY` was taking just over 90% of the time that `INVENTORY` took. This is a respectable improvement but then something interesting was observed.

It appears that as the system settled down to a idle state

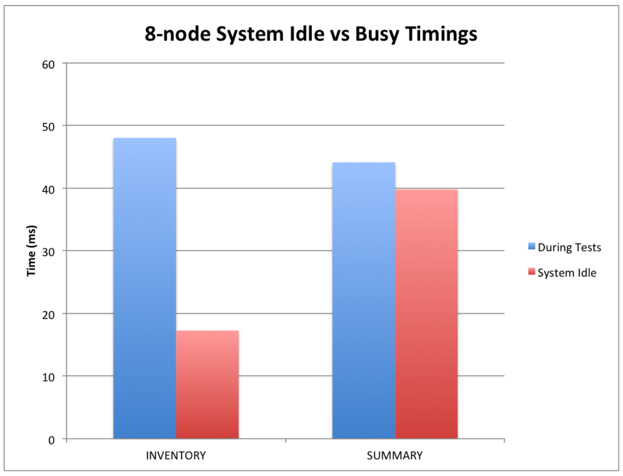

Figure 2. Idle vs Busy timings for the `SUMMARY` and `INVENTORY` methods on the 8-node TDS.

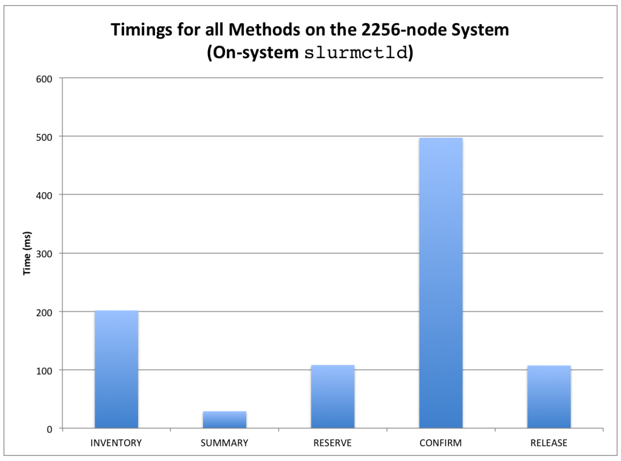

Figure 3. Method timings for the 2256-node system.

after the tests that the INVENTORY actually, on average, began to improve dramatically against the SUMMARY (see Figure 2). It is conceivable that this makes sense since, on a system as small as this, the INVENTORY response is not nearly as large to begin with. More importantly, regardless of the system size, if there are no jobs running, there will be no reservation information in the INVENTORY response and much less work for Slurm to do in the processing of this request. It should be noted that the RESERVE, CONFIRM and RELEASE methods ranged from 185 to 242 milliseconds and hence on a small system such as this, when the system is heavily used, the minor improvements gained by using the SUMMARY versus the INVENTORY will be overshadowed by the long times spent reserving, confirming and releasing jobs. More testing should be done to confirm this.

For the large system there was a significant difference in the length of time spent by the SUMMARY method as opposed to the INVENTORY (see Figure 3). Whereas the average INVENTORY run took over 201 milliseconds, the SUMMARY method on average was just under 29 milliseconds, an order of magnitude improvement! This is indeed a tremendous speed up but, as noted above, we are not receiving the reservation information and hence are not performing any sort of sync check between Slurm and ALPS jobs when issuing the SUMMARY RPC. It is also important to note that, as in the case of the smaller system, the RESERVE, CONFIRM and RELEASE methods still take far longer than the SUMMARY and although are not called as frequently (they are not called periodically but only when reserving or releasing a job) on a heavily job-laden production system, their number of invocations will be dramatically higher and hence consume a larger proportion of Slurm's run-time. Specifically the RESERVE and RELEASE methods each took roughly 108 and 107 milliseconds, i.e. over 3.5 times that of the SUMMARY so that it appears that, as on the small system, the INVENTORY does not appear to be the real bottleneck.

Furthermore, the times spent in the CONFIRM method were also significantly longer still. However, between a run where the Slurm code used only INVENTORY and another where both INVENTORY and SUMMARY were used there was a major discrepancy. In the former, CONFIRM took on average roughly 294 milliseconds but in the latter it took roughly 497 milliseconds. Therefore it would be useful to rerun these tests several more times to gather more data. Regardless, these results again illustrate that overall this method takes significantly longer than INVENTORY and hence will dominate it in terms of being a potential bottleneck.

Another test we conducted on the large system was to measure the times of the various methods when the Slurm controller was being run on an external login node of the system (see Figure 4). Implementation details of how we managed to run the slurmctld remotely can be found in the next section and involved some minor middleware between the remote slurmctld and apbasil. For now let us consider the results of these tests. As one may expect, the performance of all methods dramatically slowed, taking in the region of hundreds of milliseconds, most likely due to network latency and the added communication through the intermediary middleware.

Specifically the RESERVE and RELEASE took on average around 650 milliseconds and the INVENTORY roughly 785 milliseconds. The SUMMARY, with an average of roughly 592 milliseconds, was approximately 25% faster than INVENTORY. Still a significant improvement but far less than the order-of-magnitude improvement seen when the slurmctld was running on the main system.

In summary, the results of this exploratory work on the use of the SUMMARY method are somewhat mixed. On the one hand, it does demonstrate that the new method

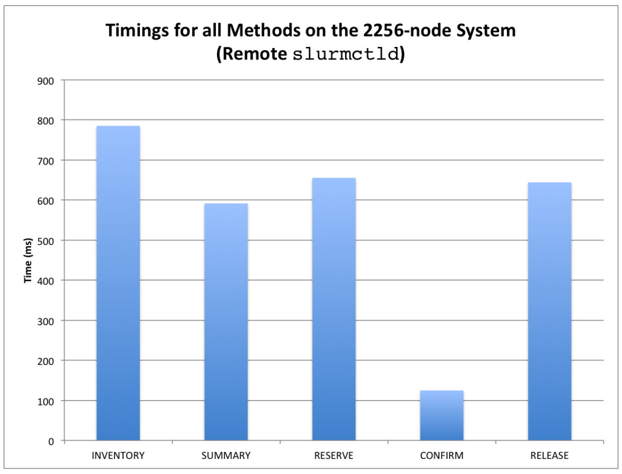

Figure 4. Method timings for the 2256-node system using a remote slurmctld.

provides significant improvements over `INVENTORY` for certain types of tests (albeit these tests were by no means exhaustive). There could be other metrics to use to compare these methods including the total number of jobs able to be backfilled in one version versus the other or their effect on the responsiveness of Slurm. Furthermore, the actual functionality of the `SUMMARY` method, as both stated in the documentation and observed in use, shows that it does not provide all, but only some, of the same information as `INVENTORY` in a compact form. Of course, as previously stated, this raises the question of whether Slurm really needs to check the full system `INVENTORY` as often as it does. However, that would be the subject of a different investigation and for now Slurm does need the additional information and so the new `SUMMARY` method has inherent limitations in its use. Perhaps it would be more useful if there were a corresponding *`QUERY(RESVSUMMARY)`* or similar method that would provide a compact summary of the job allocations. The original motivation for the new `SUMMARY` method, namely the burgeoning size of the `INVENTORY` as systems continue to grow in size, and the associated potential speedup mean that it does merit further testing.

III. Decoupling SLURM from the Main XC30 System

Although not necessary for Slurm to function and not necessarily particular to the XC30, one sub-project was to explore the use of running the Slurm controller (`slurmctld`) on an external login (esLogin) node of the system. This would serve to free up some resources on the service nodes of the system and more importantly, result in increasing the resiliency of the Slurm environment. That is, should the main system go down, Slurm's controller could remain up and present the user with a consistent experience. Although users would obviously not be able to run jobs, they would be at least able to submit them and see the state of the system through Slurm.

The current state of the this work is ongoing but significant progress has been made. The `slurmctld` can be started and will remain up both as a secondary and even as a primary controller on the external login node. It can schedule jobs and function just as if it were running on the primary system. However, more testing and possible enhancements need to be made for the case when the real system goes down.

In order to enable the `slurmctld` to execute remotely, various minor adjustments had to be made, from a system administration perspective, such as specifying fully qualified DNS names of the nodes that will be the front-ends. More importantly, any configuration option that involved a specific path needed to be checked to see if it would still make sense. This meant that the location of such things as the log file or the pid file could be different but more importantly that the state file directory must be mounted on both the external logins and the front-end nodes so that there would be a consistent state between the controller and the Slurm daemons.

Once all of these relatively minor tweaks were made the next major hurdle was to enable the Slurm controller to call the `apbasil` command on the main system, in order to communicate with ALPS. Since the command is not present on the external login nodes, we considered a number of solutions but in the end we opted to write a small, relatively simple C-program of a few hundred lines of code that acts as a conduit for communication between the remote Slurm controller and `apbasil` on the main system. Initially a script was used but there were difficulties in being able to read the resultant response and so a compiled program seemed the better approach. For some parts of the intermediary `apbasil`, we were even able to use pieces of existing Slurm code, notably the popen2 and supporting functions that Slurm uses to create its own pipe with `apbasil`.

Once finished, this `apbasil` was placed on the external login node and since Slurm has a configuration file called cray.conf, which is used to specify any settings for various Cray programs that may be in non-default locations (usually used for emulated systems but proves very useful in this case as well), we were able to specify the the location of the new `apbasil` command on the esLogin node. Thus, this intermediary is specified and Slurm communicates with it as if it were the normal `apbasil`. For its part, the intermediary `apbasil` simply reads in an XML request from the `slurmctld` and then opens an `ssh` connection, via a pipe, to the real system and runs the `apbasil` from the real system, in the process writing the XML request that it had received to the pipe. Then it reads the XML response from the pipe and writes it as output which is then read by the Slurm controller itself on its pipe.

Now that we have the remote functionality working, we can now experiment to see what, if anything, needs to be modified to handle the case when the main system goes down. Also, a similar work item is to enable salloc to always function from the external login nodes. This already appears to sometimes work so it may be more of a matter of setting up the environment in just the right way.

IV. New Task Affinity SPANK Module

In an attempt to prepare users for the XC30 system in the months leading up to its delivery, a small non-Cray Cluster was set up. Although it was not a Cray system and had a different interconnect (InfiniBand with a fully on-blocking fat tree topology rather than the Cray Aries with Dragonfly topology) the nodes had the same 2.60GHz Intel Xeon E5-2670 processors. As with the eventual real XC30, the test system's nodes contained a $2\times8\times2$ configuration where 2 is the number of sockets and 8 is the number of cores per socket and the final 2 is the number of hardware threads per core.

Being a non-Cray system there were two general differences with regard to jobs and scheduling that had to be addressed in order to make this system a viable *testbed* for the XC30. First, resources could be consumed on a sub-node basis and secondly, the order and nature in which software threads were bound was different from the XC30.

The limiting of one job per node was straightforwardly achieved by simply setting the `Shared` partition attribute to `EXCLUSIVE`. However, in order to get the task bindings to be set in a similar fashion to the Cray systems without necessitating that the user compute and specify a specific bit mask, it was decided that a new SPANK module would be needed. SPANK, or *Slurm Plugin Architecture for Node and Job (K)Control*, modules are yet another example of the extensibility of Slurm. This mechanism allows the Slurm administrator to stack one or more such modules and its architecture provides a number of function interfaces that the programmer/administrator can then provide an implementation for. These functions, if defined, are called at various points in the Slurm logic during such tasks as job launch, thereby allowing for modification of normal Slurm behaviour. A typical use is to add some additional launch options to a job which is precisely what our SPANK module would be used for.

The idea was that all the user would need to do is add an option to their `srun` command line requesting the special binding and Slurm would handle all of the details. The binding pattern itself was to be as follows:

- A unique fat mask is created for each task of each node of the job;
- Each fat mask will have one processor per software thread of the task;
- Each processor assigned to a given task will be "next" to each other (i.e. in numerical order);
- No more than one processor per core will be used; thereby effectively binding at the core level;
- Each task will be confined to one socket but there can be more than one task per socket if they all can completely fit;
- Any violation of this policy will cause the job to be rejected and if arbitrary and eclectic bindings are required such as having some tasks spread across sockets or others using multiple processors per core or over-committing certain resources then the user must specify this explicitly using the normal task/affinity masks.

As an example, given hardware that is $2 \times 8 \times 2$ (2 sockets $\times$ 8 cores-per-socket $\times$ 2 processors-per-core) and a job with, say, 4 tasks and 3 threads-per-task then the binding should be such that two tasks are placed on each socket and there will be two unused cores on each socket. In other words, binding always starts at the beginning of a socket, filling that socket in terms of whole tasks without splitting the task.

As a starting point for the work, the old *auto-affinity* SPANK module written at LLNL was tested to see what sort of binding it provided. Our testing showed that it did not use the same pattern as we wished to have and included several options that were unnecessary for our purposes. However, it contained the general framework of how to write this type of SPANK module. Therefore, we took this module and stripped out all the options except for `off`, `help` and `(v)erbose` and added `on`. Also we changed the default behaviour, slightly, so that now it is *off* unless the user specifies to use it. Then, finally most of the CPU mapping logic was rewritten to match our desired functionality.

Building and installing the SPANK module is straightforward. The developer need only compile against the `spank.h`, `slurm.h` and `slurm_error.h` header files provided by Slurm. Note that the Slurm documentation states that only `spank.h` is needed but because the original SPANK module used several Slurm types and various Slurm functions and we continue to use these, we also needed the other two header files. Once compiled, the `*.so` is placed in the `.../lib` directory of the Slurm installation and a `plugstack.conf` file goes into the same directory as the `slurm.conf` file. The contents of the `plugstack.conf` file is simply a list of the SPANK modules to be used, one per line, with each line stating whether the module is required or not[1].

From the user perspective, it is easy to use this default binding pattern. All that need be included on their job-step submission line (i.e. as part of the `srun` command line) is the option `--reduced-auto-affinity=on`. Note that here we see that this option appears to be merely another native Slurm option for `srun` but, in fact, is processed by the SPANK module. As an example, consider the following example batch job-file (called `job.sh`):

```
#!/usr/bin/ksh
#SBATCH -n 4
#SBATCH -N 1
#SBATCH --cpus-per-task=4
#SBATCH --time=00:05:00
#SBATCH --output=autoaff-%J.out
#SBATCH --error=autoaff-%J.err
#SBATCH --partition=normal

export MV2_ENABLE_AFFINITY=0
export OMP_NUM_THREADS=$SLURM_CPUS_PER_TASK

srun --reduced-auto-affinity=on,v \
--cpu_bind=no ./myapp.exe
```

The command line for this would appear as follows:

[1]e.g. "required /slurm/2.5.4-pam/lib/reduced-auto-affinity.so"

```
$ sbatch job.sh
```

The above example would produce a job of a single node with four tasks running on it. Given a 2×8×2 node layout and the fact that each of our tasks is to have four threads, we would have two tasks placed across the first socket and the next two on the second socket. Each software thread, of course, would be using only one processor per core.

This new task affinity SPANK plugin will likely be included as part of the Slurm 2.6 release.

V. Conclusion

All in all, our experience with preparing Slurm for use on the XC30 was a positive one. With the backwards compatibility of BASIL, the core Slurm code needed little modification to run on the new architecture. Through a variety of sub-projects we were able to provide additional functionality, such as the use of the new BASIL 1.3 feature `nppcu`. We were also able to lay the groundwork for decoupling the Slurm controller from the main system and provided XC30-like binding patterns, at relatively low-cost, to some of our users of general-purpose clusters. Finally, we had the opportunity to briefly explore the use of the new `QUERY(SUMMARY)` method and so familiarize ourselves with some potential difficulties and also benefits of its use.

Acknowledgment

Thanks to Moe Jette, Danny Auble and the rest of the team at SchedMD for their continued support and feedback on Slurm-related issues.

References

[1] [Online]. Available: http://www.schedmd.com/

[2] G. Renker, N. Stringfellow, K. Howard, S. Alam, and S. Trofinoff, “Deploying SLURM on XT, XE, and Future Cray Systems,” *Cray User Group Meeting*, 2011.

[3] B. Landsteiner and C. Albing, “Cray BASIL 1.3 Specification,” *Unpublished Cray Proprietary Information*, 2012.