Experiences Using the Portable Batch System on Large Origin Systems

Ed Hook (hook@nas.nasa.gov)
Computer Sciences Corporation
NASA Advanced Supercomputing Division
NASA Ames Research Center
Moffett Field CA 94035-1000

The NASA Advanced Supercomputing (NAS) Division operates a number of large SGI Origin systems, with all access to their resources controlled by the Portable Batch System (PBS). In this paper, I describe some of the things that NAS has found it necessary to do, in order to provide highly-reliable and robust service on these systems to our users. In particular, I focus on the use of cpusets to (in effect) provide "software partitioning" of these systems and address some of the issues raised by "coarse-mode addressing" on large Origins.

1. A Word About PBS

The Portable Batch System (PBS) is a resource-management system that is used to control the workload on all of the "compute engines" which NAS manages. It accomplishes its work by the cooperative functioning of three daemons:

pbs_server : the daemon that users interact with, it accepts jobs from users and manages their progress through the system

pbs_mom : the one that actually launches jobs into execution, because it has the system-specific knowledge about how that's done and also how to track the resources used by a running job

pbs_sched : the daemon that decides which jobs should be run at any given time, based on policy and the availability of resources requested.

One of PBS' best features is the fact that there is a separate scheduler and that it is fairly easy for a site to provide its own implementation of that scheduler. This makes it possible to implement policies that would be extremely difficult (if not impossible) to realize in the old queue-based approach that characterized (say) NQS.

Another vital feature is the fact that the PBS source code is available. (There are now two versions of PBS. Anyone can obtain the source code for the OpenPBS release, while it is necessary to negotiate a source- code license agreement in order to get the code for the PBSPro release.) Without access to the source, the work reported here would have been impossible.

2. Some History

A fact: a multithreaded job running on a large Origin system is vulnerable to a number of factors that tend to degrade its performance (and, as a side effect, lead to a great deal of

variation in its measured runtime). The most obvious problem is caused by the fact that (since it *is* a single-system-image) the kernel will move processes around within the machine in a manner that grows increasingly less deterministic as the system load goes up. But the memory allocated to those processes does *not* move, so jobs can see significant evidence of the "nonuniformity" in "NUMA". Because this is such a problem, NAS and SGI have devoted quite a lot of effort over the past few years to assuaging the difficulty.

The first approach involved assigning to each job a "nodemask", which told the kernel just which nodes we'd like it to use when allocating memory to the processes making up that job. Since the kernel did attempt to take memory placement into account when choosing which CPU to assign to a runnable process, the nodemask tended to confine processes and the memory they used to the specified set of CPUs ... so long as it worked. The problem with using nodemask was/is that it is purely advisory, a hint that the kernel is free to `i`gnore when honoring it proves to be impossible (or even merely inconvenient). So we would observe that everything was working just as we had hoped, until the system became full. At that point (the very point where we most needed the control that nodemasks tried to provide), all bets were off and, typically, things would proceed from bad to worse. And even when the system was not loaded down, a rogue job could spawn many more processes than the number of CPUs the user had requested, bogging down the system and roughing up the other users, and we had no effective way to prevent this. In theory, a job could also use more memory than was installed on the nodes it was allocated (that is, the nodemask approach provided no way to stop that happening), but PBS was able to detect this sort of misbehavior and kill off the offending job.

Despite the problems with nodemask, we made do with it until we got our 256-CPU Origin system (called steger). Very early in the lifetime of steger, the variations in runtime for any given job were found to be so extreme that NAS was embarrassed to charge users for running on it. In response, steger became (and remained) "free" for more than a year, while the systems folk addressed the problem. The solution that was eventually deployed has a number of pieces, but the most *important* piece involves the use of "cpusets". This was a concept that had been developed/implemented by SGI as part of their '*miser'* resource management system. Because *miser* didn't fit very well with the NAS infrastructure (in particular, getting it to play well with PBS did not appear to be something that anyone would want to undertake), we couldn't make use of *miser* cpusets directly, but they clearly had the properties that we wanted and needed. So NAS and SGI embarked upon a joint effort to implement cpusets outside of *miser,* initially using some new system calls to define an API that was later encapsulated in the library 'libcpuset.so'.

3. What's A Cpuset ?

A cpuset is a named set of CPUs which can have various properties. For an exhaustive account, "*man cpuset*" is recommended; here, I'll discuss the particular sorts of cpusets that are used at NAS.

For our purposes, a cpuset always consists of all of the CPUs in a collection of nodes (the allocation algorithm insures this) and a cpuset always has the following set of properties:

EXCLUSIVE:
Only processes attached to the cpuset can run there
MEMORY_LOCAL:
Memory for processes attached to the cpuset will be allocated locally unless there's no such memory available
MEMORY_EXCLUSIVE:
Memory for processes *not* attached to the cpuset will not be allocated locally, unless there's no memory available elsewhere
MEMORY_MANDATORY:
Memory for attached processes will be allocated locally - if no memory is available, what happens depends on the 'POLICY' specified
POLICY_KILL:
The kernel will free as much space as possible from its heaps, but will *not* page out user pages to the swap file. If the physical memory of the local nodes is exhausted, the process causing the problem will be killed

and one other that's (apparently) undocumented: CPU_EVENT_NOTIFY. This last causes the creator of the cpuset to be notified whenever POLICY_KILL forces a process to be terminated, the notification being by delivery of a SIGWINCH.

Taken all together, the above properties insure that all of the processes in a given session are isolated in their assigned cpuset. They can't get in the way of other sessions, nor can they be interfered with by those same other sessions. Since cpusets are disjoint from one another and so are the corresponding nodes, we get a "software partitioning" of the system into as many distinct pieces as there are running PBS jobs.

Actually, that last sentence is not quite accurate - there's another cpuset, named "boot", that is not associated with any PBS job. As part of system startup, this cpuset is established and the '*init'* process is moved into it. Since almost everything else is a child of '*init*', this means that virtually every process begins life running in the boot cpuset and root privilege is required to move off of the boot cpuset. This is the key to managing these systems under PBS. When *pbs_mom* is told to place a particular job into execution, that daemon (running as root in the boot cpuset) creates a cpuset for the job out of the CPUs populating some of the "free" nodes and then attaches the job's top-level process to that cpuset. Since all of the processes that constitute the job are children of that top-level process, this has the effect of placing the job into its very own dedicated cpuset. When the job finishes, the *pbs_mom* daemon tears down the cpuset, returning its pieces to the pool of available resources. In essence, that's how cpuset support is added to PBS.

Of course, there's more to it than that. For instance, there's a good deal of bookkeeping required to keep track of which nodes are available and the like. Beyond that, most of the additional code in *pbs_mom* is needed because tearing down a cpuset sometimes hits a snag or two. The difficulty is caused by the fact that the kernel will not disassemble a cpuset so long as there are active processes still attached to it. A typical scenario: an MPI application running on a

substantial number of CPUs encounters an error which causes a large number of its threads to dump core while running out of an NFS-mounted directory. MPI programs tend to produce humongous corefiles, there will be a lot of them and they are being written very slowly. It is very likely that the job will exceed its walltime limit and PBS will try to terminate it, eventually delivering a SIGKILL to all of its processes. At that point, *pbs_mom* (the daemon that's doing all of the heavy lifting here) will try to destroy the job's cpuset; the attempt will fail, because none of the core-dropping processes has seen the SIGKILL yet and so they remain active. To handle these sorts of complications, the daemon maintains a list of "stuck" cpusets and periodically retries tearing them down. (Most stuck cpusets are, in fact, reclaimed within a minute or two, 20 seconds or so being fairly typical. Of course, some are never reclaimed and it's necessary to reboot the system to get back the resources. This doesn't happen very often, however. And it happens a *lot* less now than it did in the early days on lomax.)

4. Bigger and Better: Coarse Mode Addressing

When NAS and SGI decided to build an even bigger Origin 2000 system, we took it as an opportunity to revisit the things that we had done to make the 256 CPU steger system usable. The major concern with the new 512 CPU system called lomax had to do with the effects of "coarse mode addressing" on job performance. Briefly, the original design of the Origin 2000 incorporated the assumption that the largest system that they'd ever build would have no more than 128 CPUs. As a consequence, the control messages passing over the internal network have only enough bits in their destination field to address 128 different targets. To deal with the problem caused by a system having more CPUs than can be addressed, the hardware implements a "coarse mode addressing" scheme. The system can be viewed as a collection of 128-CPU pieces (called "octants") that are cabled together. Within each of these pieces, internal control messages can be addressed to individual CPUs, but messages that have to travel between distinct pieces are delivered to all of the CPUs in the rack where the real intended recipient resides. This obviously leads to a noticeable increase in traffic over the internal network, which we would like to avoid as much as possible.

Clearly, there is no way to avoid the problem for a job using more than 128 CPUs, so we just fit those in wherever the nodes can be found. For smaller jobs, we run them within a single octant, which means that we have to be able to identify the various octants. This is not trivial. Each node has both a "compact node id" (its "node number") and a "nasid" (which is *usually*, but not *always*, identical to the node number). The hardware decides whether or not to use coarse mode addressing on the basis of the nasids of the nodes involved. For example, the 4 octants on lomax consist of the nodes with nasids 0-63, 64-127, 128-191 and 192-255 and we would want any "small" job to run entirely within one of those 4 pieces. This means that whatever software is handling this task has to be able to determine the mapping between nasids and node numbers. This mapping can only be reliably determined by walking the hardware graph to find the nodes and using a system-specific '*ioctl()*' to fetch each node's nasid. Once this mapping and its inverse are known, it is straightforward to translate between "logical" nodemasks (based on node numbers) and "physical" nodemasks (based on nasids); since cpusets are defined using a logical nodemask, this translation is a vital piece of our implementation.

5. Some Implementation Details

So who has to know about this translation ? It turns out that both *pbs_mom* and *pbs_sched* need to incorporate the code that determines the mapping, as well as sharing the code that determines just which nodes should be assigned to a given job. This unfortunate coupling of the two daemons seemed the natural approach, given the evolution of PBS on large Origins at NAS. Back when we relied on nodemasks alone, *pbs_sched* chose the nodemask for a job and passed it along to *pbs_mom*, whose only involvement was to pass it to the kernel as part of the startup code. When we switched to using cpusets, it was clear that *pbs_mom* ought to be determining which particular CPUs to allocate since she was creating the cpuset. So the nodemask code was removed from *pbs_sched* and installed in *pbs_mom* as the allocation algorithm (it fit in nicely, since we really schedule *nodes*, not CPUs). When we set out to redo things for lomax, it seemed that *pbs_mom* should still be the one that actually manages the cpusets, but *pbs_sched* should *also* know how to translate between node numbers and nasids and should know exactly which nodes would be assigned to a job if it were run. That's because there's no other reliable way to enforce the policy that small jobs must run within an octant. Just recently, we've reverted somewhat to the original design: now, the Scheduler actually specifies to *pbs_mom* just which cpuset should be assigned to a job, *pbs_mom* honors that request if possible (and it should always be possible, but, if it's not, she allocates what she can). This change will make it a lot easier to implement support for "advanced reservations" (a feature of PBSPro) and similar capabilities. Unfortunately, it doesn't do away with the coupling of the daemons (since, for example, a job that is run manually by an operator won't have the suggested nodemask, so *pbs_mom* still has to know the properties that we want cpusets to have).

Because they share the node-allocation code, *pbs_mom* and *pbs_sched* also share a couple of configuration parameters (unavoidable, as the algorithm uses those parameters to decide just how to assign nodes to a job). But the relationship of the two daemons is otherwise not exactly symmetrical. The scheduler relies on *pbs_mom* for information about what resources are available and makes no attempt to track that stuff itself. In particular, each time it runs, the scheduler asks *pbs_mom* for her idea of which nodes are available and for a list of any stuck cpusets. The response to the first request is a string giving the hexadecimal encoding of a 256-bit nodemask, a bit being set if and only if the corresponding ("logical") node is not yet assigned. This is converted internally to a bitstring, which is then translated into the corresponding "physical" nodemask bitstring. Once we have those bits in hand, it is a simple matter to implement our policy that all jobs asking for no more than 128 CPUs (64 nodes) must be run within a single octant. All that's needed is to test whether the job needs 64 nodes or less and, if so, to check whether the right number of '0' bits can be found in any of the 4 64-bit subtrings that correspond to the octants. If the job is big enough to avoid this policy, we just need to see whether there are enough nodes available at all, which means that users who don't care about getting the best possible timing results can sometimes improve their turnaround by asking for more CPUs than they really intend to use. In either case, if a job is selected to be run, the scheduler translates the physical nodemask to its logical equivalent, adds the corresponding bitstring to the properties of the job (so that *pbs_mom* can retrieve it to determine which cpuset to allocate) and updates its notion of which nodes are still free prior to moving on to consider the next queued job.

The use made by the scheduler of the list of stuck cpusets is not quite so straightforward. Because lomax is fairly heavily loaded with jobs of all sizes, it can be hard to find the resources requested by a given large job. So we implement the notion of "starvation" - if a job has been rotting in the queue for longer than a configurable period of time (currently set at 8 hours), it is deemed to be starved and the scheduler will resort to heroic measures to run any starved jobs. These heroic measures involve draining the system to free up the needed resources for the "most starved" job - given the list of currently running jobs, the scheduler can estimate how long it will be before enough of their resources will be freed up to allow the targetted job into execution. And, by tracking how the list of free nodes will evolve, the scheduler can also tell just which nodes will have to be assigned to the job when it *does* run. Knowing these two pieces of vital intelligence, the scheduler can "backfill" around the starved job: any job that (1) will complete before we can run the targetted job anyhow or (2) will be assigned a cpuset that won't interfere with the targetted job can be run right now. This works pretty well in practice, but the early implementation(s) would sometimes decide that we would *never* be able to run the starved job and so would give up the attempt. This was particularly maddening in cases where we *had* been draining the system for several hours and then walked away from the problem. A close examination of the situation turned up the fact that the anomaly was always caused by a stuck cpuset - if it involved CPUs that would be needed by the starved job, it was a real obstruction. After a few approaches that failed to work satisfactorily, we hit upon one that seems to work pretty well. Using the list of stuck cpusets that it gets from *pbs_mom*, the scheduler maintains a linked list of stuck cpusets that records for each of them just how long it has been stuck. The logic that handles starved jobs makes the assumption that a cpuset that has only been stuck for a configurable time interval (currently set at 600 seconds) will soon be reclaimed and adds the CPUs tied up in such cpusets back into its notion of what *will* be available in the near future. By doing this, we have almost entirely done away with the problem detailed above - the only time this approach doesn't work is when there is a truly wedged cpuset, one that can't be reclaimed without rebooting the system. (But *no* approach is particularly helpful in that case.)

6. Finally ...

The version of PBS that we are currently running on our various large Origin 2000 systems seems to work pretty well. And its implementation of support for cpusets seems to be a big part of the reason for its success.

Exactly the same software is also being used on our Origin 3000 system which is (at the moment) a 512 CPU single-sytem-image. The only real changes needed to move to the newer system were in the configuration files, because a node on the Origin 3000 has 4 CPUs (rather than 2, as on the Origin 2000). In the near future, this system will become a 1024-CPU system and I expect the software to handle that doubling in size virtually transparently.