The Usage of Cpusets on a 256 CPU Origin 3800

Jim Glidewell & Barry Sharp, Boeing Shared Services Group

Abstract

This paper reviews our experiences with cpusets on our 256 CPU Origin 3800 over the past year. It provides an overview of why we chose to use cpusets and how we used them, both under PBSPro and stand-alone. Particular attention is paid to our experiences with the use of the shared-cpuset feature of PBSPro, which was released with PBSPro 5.2.

The Need for cpusets – A Quick Review

The development of cpusets arose out of a need to address certain architectural issues that arise in the context of SGI's ccNUMA architecture. While ccNUMA offers a number of advantages, such as a large contiguous address space, efficient MPI execution when using SGI's MPI implementation, and relatively low cost compared to comparably sized "flat" memory systems, it has some significant drawbacks as well.

Among these drawback is the fact that good processor to memory latency requires that the memory be "nearby" to the CPU. Ideally, the bulk of the memory a given process needs to address will be located in the same "brick" in which the processor resides, failing that, every additional "hop" adds significant delay, resulting in slower processing. As these machines become larger, the problem is intensified, and focus on data locality becomes ever more critical.

Even minor oversubscription of the CPUs on a large system can have a disastrous effect on data locality and therefore application efficiency. And more troubling is the fact that such a situation is fairly difficult to detect, since the increase memory latency is usually "hidden" from administrators and only appears as additional CPU time for individual application runs.

The Cpuset solution

SGI's solution to this issue was to add the concept of cpusets to IRIX at version 6.5.8. Cpusets allow the system administrator to temporarily (or permanently) partition the system into multiple independent slices, assigning work to each slice. Cpusets thus force both running processes and their memory to be co-located in a small region of the system, while at the same time preventing other processes from interfering with them.

Cpusets and Miser had a somewhat problematic debut, and it appeared to us that there had not been wide adoption of cpusets at other sites. While the cpuset concept seemed sound, it appeared to us to be at best a necessary evil, which was to be avoided if at all possible. It seemed to add a fair amount of complexity and rigidity, which we weren't sure would be compatible with our fairly wide-open demand batch environment. And finally, it did not appear that cpusets would offer us a significant improvement on our 64 CPU Origin 3800.
But things were changing fast.

Cpusets Reconsidered

A number of factors caused us to take a second look at cpusets. The strongest single factor was the performance of the Aero application, Overflow. Our engineers were seeing huge variation in CPU for what were essentially identical runs. After receiving a test case from one of our users, we observed a variation of over 2 to 1 in the CPU time for an identical case. Given that this application consumed far more resources than any other on our system, we knew we needed to do *something*. These same users had experience running Overflow on the Origin at NAS under cpusets, and were thus able to tell us what the expected CPU time was in that environment.

Two other factors came into play in our decision to make a serious attempt to make use of cpusets on our site. One was the selection of PBSPro as our batch subsystem, which supported cpusets directly. The second was the fact that we were upgrading our 64 CPU system to 256 CPUs, which would aggravate the existing problems we were seeing with parallel applications.

Early Experiments with Cpusets

We chose an Overflow test case for our initial experiments with cpusets. This case took between 90,000 and 200,000 CPU seconds when run without cpusets. Our initial attempts to run the application on a newly created cpuset were disappointing, with little improvement in CPU time, with results ranging from 87,000 to 186,000 CPU seconds. After some discussion with our onsite SGI support personnel, we determined that the problem was probably due to the fact that we were creating cpusets “on top of” existing processes, and the displaced processes were still making heavy use of memory which resided inside the cpuset.

To verify this theory, we define a cpuset and then deliberately left it idle for nearly 24 hours, allowing any memory inside the cpuset to be freed as the associated processes terminated. After verifying as best we could that the memory was idle (using Performance Co-Pilot) we ran the test case several times. These runs offered consistent run times, all clustering around the 87,000 CPU-second target that we had set. We had proven two things to ourselves: that cpusets could make a real difference in the execution time for Overflow, and that in order to see the benefits of cpusets it was essential that the cpuset not be created on top of other work. This meant that cpusets needed to be either static, or managed in an environment which otherwise restricted process creation to specific nodes. The upshot was that for the most part, cpusets were an all or nothing proposition.

Interim Solutions

At this point, we were still unable to move to PBSPro managed cpusets, as we were still waiting for a resolution to the “single-CPU job” issue. Given the serious impact that the lack of cpusets was having on Overflow, we felt we needed to do *something* as an interim solution.

We started by simply defining a 64 CPU static cpuset “overa” which we allowed the Overflow users to use within their jobs. Since this cpuset was static, the issues of pre-existing processes on those nodes did not arise. But this solution but a fair amount of burden on the users to coordinate use of the cpuset, it did not work all that well from a human standpoint. In particular, there was no way for users to launch a number of jobs for weekend processing using this technique, since no one job could be assured that the cpuset was free. We needed a mechanism for communicating whether a cpuset was active or not. We therefore created the “idlecpuset” script, which would return either the name of an idle cpuset that the job could use, or null if no cpusets were available. The user job could then decide the appropriate action (generally, run outside of any cpuset). If a cpuset name was returned, the user would use the lines:

```
Export IDLENAME=`idlecpuset`
cpuset –q $IDLENAME –c user_command_line
```

to execute their main application on the cpuset.

The “idlecpuset” method allowed users to submit multiple jobs, and get the maximum use of the cpusets provided. It also allowed us to provide multiple cpusets, if the demand was there. And it was. We decided that we would keep all such cpusets the same size – 64 CPUs. To allow for jobs that did not use this mechanism to run, we did not want to create a number of static cpusets of this size. Instead, we leveraged off the fact that these users were all using the idlecpuset script to allow for a primitive client-server mechanism to create cpusets on demand.

When a user issued the idlecpuset command, if a cpuset was available, it would return the name off the cpuset immediately. If a cpuset was not immediately available, the idlecpuset script would post a request for a cpuset (via a flag file) then sleep for a few minutes. This
Delay would allow a separate createcpuset daemon to detect the request and create a cpuset if possible. When the idlecpuset script woke up after its sleep, the cpuset would be there and waiting. When a user job was through with a cpuset, the createcpuset daemon would detect the idle cpuset, and after a brief delay (30 minutes) would delete it. While this technique left us open to the possibility of cpusets being created on top of existing processes, it also ensured that we would not run in a mode where idle cpusets would occupy a large chunk of the machine, possibly forcing oversubscription on the remainder of the system. Although crude, this method was fairly well received by our users, with the understanding that it was only a temporary solution.

The “Flythru” cpuset

During this period, a new application was rehosted on our Origin 3800. Flythru is a system which allows access to Catia geometry data from Unix based workstations, primarily SGI and IBM. The two major functions of a Flythru server are file sharing (using NFS) and the “Update” process, which synchronizes the local database with the reference Catia database that resides on IBM mainframes.

What we learned, rather late in the migration, was that Update was implemented using nested shell scripts, and that it typically generated anywhere from 10,000 to over 100,000

processes per hour. Besides wreaking havoc with process accounting, this application was causing serious problems with our system as a whole, due to its constant “stirring of the pot” which caused processes to get bounced from one node to another.

While the ideal solution would be to get Update on to a separate system entirely (or at least a separate partition), we found that running all Update-related processes in a permanent 4–CPU cpuset has either mitigated or eliminated the effects of Flythru on our other production work.

While our previous focus had been to protect a specific application from then system as a whole, in this case we found that rather than protecting an application from the system, a cpuset was very useful in protecting the system from an application. Until we can find a better “home” for Update, or it is significantly reworked (rewritten in Perl, perhaps?), we will continue to use the flythru cpuset to control its impacts on the system as a whole.

Moving cpusets into the mainstream – PBSPro 5.2

We had deferred implementation of PBS managed cpusets until Veridian could provide us a solution to the “single CPU job” dilemma. Prior to version 5.2, PBSPro would always generate a separate cpuset for every job. Further, it would generate a cpuset with a minimum size of one node (4 CPUs on an Origin 3000). This meant that a single-CPU job would reserve 4X the number of CPUs it actually needed, and those CPUs would remain idle for the life of the job. While this might be acceptable at a site where single-CPU jobs were infrequent, this was not the case at our site.

During our initial evaluation of PBSPro, we cited this as a major issue and stated that we would need a solution before we could utilize cpusets at our site. Veridian committed to providing a solution to this problem within 6 months of our going production with PBSPro. They did it.

With the release of PBSPro 5.2, the concept of “shared cpusets” was introduced. At the site’s discretion, “small” jobs could be placed in a dynamically created cpuset that could be shared by up to three (on a 4 CPU per node Origin 3000) other jobs.

This resolved the issue of wasted resources when using cpusets in a job mix with a significant number of single-CPU jobs. We were ready to move forward.

Our initial configuration for running cpusets under PBSPro 5.2 was as follows:

A boot cpuset of 3 nodes (12 CPUs, 18GB memory)
The flythru cpuset - a single node (4 CPUs, 6GB memory)
The remaining 236 CPUs and 354GB of memory scheduled by PBSPro

In general, we took the default for most PBS cpuset-related parameters. We are running IRIX 6.5.12f.

A Rough Start

While our testing had turned up few problems with PBSPro 5.2, bringing it online during a period where our system utilization was quite high flush out quite a few problems immediately.

Some of the bugs or problems we have seen include:

- Failure to schedule more than 1/2 of the machine (workaround - schedule by ssinodes, rather than mem & cpu)
- Job aborts on shared nodes, due to “out of memory” conditions (workaround - change the cpuset creation flags from “POLICY_KILL” to “POLICY_PAGE”)
- Jobs stuck in “E” state (workaround - kill server with a “qterm -t quick” and restart server daemon)
- Failure to properly recover shared cpusets after restart of system or pbs_mom
- User concerns about reduced overall system throughput
- Excessive CPU consumption by pbs_mom
- Increased swapping

Many of these problems were greatly aggravated by the fact that we were experiencing multiple system crashes and hangs, due to a difficult to diagnose hardware problem. To their credit, (and our great relief) the support personnel at Veridian have been very helpful and supportive during this fairly rocky transition.

The Need for Visibility

Prior to PBSPro 5.2 and shared cpusets, there was a one-to-one correspondence between PBS jobs and their associated cpusets. With the advent of shared cpusets, this simplicity and clarity was gone. To address the need for better visibility, we have created a couple of tools to help us track jobs, cpusets, and their related resource usage. The first of these tools is cpurep, a Perl script that interrogates the cpuset and qstat commands and builds a report that ties the two sources of info together.

```
origin 11% cpurep

cpuset          cpus    procs   PBS_jobs
------          ----    -----   --------
boot            12      377
flythru         4       56
41013.or        64      66      41013(xxx7307)
40497.or        4       7       40523(yyy2754)
40550.or        4       12      40576(xxx7307)  41014(tecxxx1)
41015(tecxxx1)  41016(tecxxx1)
```

```
Job_id    user      Irix_jid              cpuset      cpus  shared?
------    ----      --------              ------      ----  -------
40523     yyy2754   0x2a1900000001120b    40497.or       4
40576     xxx7307   0x2a1900000001b712    40550.or       4     *
41013     xxx7307   0x2a19000000036cfe    41013.or      64
41014     tecxxx1   0x2a19000000036d53    40550.or       4     *
41015     tecxxx1   0x2a19000000036d54    40550.or       4     *
41016     tecxxx1   0x2a19000000036d55    40550.or       4     *

Assigned Nodes Map:

XXXX........X........XXXXXXXXXXXXXXXXX.........................

Total CPUs allocated = 88. Nodes allocated = 22.

Total CPUs free = 164. Nodes free = 41.
```

In addition to this tool, our performance specialist has created a tool called “pmgcpuset” which uses the PCP (Performance Co-Pilot) widgets to build a display which shows CPU and memory utilization for a specified cpuset.

These tools have been very handy in gaining an understanding of PBSPro’s strategies for scheduling shared cpusets, as well as providing us additional monitoring abilities. We are exploring other methods of acquiring and displaying cpuset-related information.

The “Packing” Problem

One issue that has become apparent since our implementation of PBSPro 5.2 is something I’ll refer to as the “packing” problem. Veridian’s implementation of shared cpusets always allocates a single node at a time – all shared cpusets are exactly one node in size. For our site, this can sometimes be a problem. We often have a number of single-CPU jobs which require more than “a CPU’s worth” of memory. In recent weeks, we have seen a large number of jobs that use just over 50% of a single node’s memory. As such, each of them ends up residing in a separate cpuset/node, leaving 3 CPUs and roughly 45% of the memory in that node sitting idle. If there are other jobs in the system which require small amounts of memory, those jobs will fit in along side the larger memory jobs, but this is often not the case on our system.

For this sort of job mix, it would be very nice if it was possible to allocate shared cpusets in bigger “chunks”, by creating shared cpusets which consist of multiple nodes. In the above example, a 2-node shared cpuset would allow three such jobs to be run in the same space previously occupied by two. We have already spoken with Veridian about the possibility of a site specified “shared cpuset allocation size” to allow sites more flexibility to customize the shared cpuset behavior to meet the needs of their site and its job mix.

User Response to Cpusets

The response of our users to the change in scheduling has been decidedly mixed. Our Overflow customers, who account for over half of our total system utilization, have been extremely happy with the results. They no longer are restricted to a single cpuset size (as they were with our interim solution) and now get the benefits of running in a cpuset for every one of their runs. For this user community, the move to cpusets has been a clear win.

Other users, some of whom were hit by some of the unexpected problems cited above are less enthusiastic about the change. In some cases, the implementation of cpusets has the direct result of their being able to run fewer jobs simultaneously. Whether this will be offset by an increase in performance for their individual jobs remains to be seen.

Most users appear to be oblivious to the change. For the most part, the move to cpusets under PBSPro has been transparent to end-users, as we had hoped it would be.

Cpusets – Worth the trouble?

We approached the cpuset feature reluctantly, unsure if the added complexity and risk was worth the benefits. But for our site, the implementation of cpusets has resolved a couple of truly vexing issues: poor performance of large parallel jobs (Overflow) and poor system performance due to a poorly behaving application. Despite the “teething pains” we encountered during our move to PBSPro 5.2, we are fairly confident that the combination of PBSPro and cpusets will provide a significant benefit to our user community.

We encourage other sites to explore the cpuset feature; particularly if they are experiencing issues similar to the ones discussed here.