Isolating Cluster Jobs for Performance and Predictability

Brooks Davis, Michael AuYeung
The Aerospace Corporation
El Segundo, CA
{brooks,mauyeung}@aero.org

Abstract

At The Aerospace Corporation, we run a large FreeBSD based computing cluster to support engineering applications. These applications come in all shapes, sizes, and qualities of implementation. To support them and our diverse userbase we have been searching for ways to isolate jobs from one another in ways that are more effective than Unix time sharing and more fine grained than allocating whole nodes to jobs. In this paper we discuss the problem space and our efforts so far. These efforts include implementation of partial file systems virtualization and CPU isolation using CPU sets.

Figure 1: Fellowship Circa February 2007

1 Introduction

The Aerospace Corporation operates a federally funded research and development center in support of national-security, civil and commercial space programs. Many of our 2400+ engineers use a variety of computing technologies to support their work. Engineering applications range from small models which are easily handled by desktops to parameter studies involving thousands of cpu hours and traditional, large scale parallel codes such as computational fluid dynamics and molecular modeling applications. Our primary resources used to support these large applications are computing clusters. Our current primary cluster, the Fellowship[Davis] cluster consists of 352 dual-processor nodes with a total of 1392 cores running FreeBSD. Two additional clusters, beginning at 150 dual-processor nodes each are being constructed to augment Fellowship.

As in any multiuser computing environment with limited resources, user competition for resources is a significant burden. Users want everything they need to do their job, right now. Unfortunately, other users may need those resources at the same time. Thus, systems to arbitrate this resource contention are necessary. On Fellowship we have deployed the Sun Grid Engine[SGE] scheduler which schedules batch jobs across the nodes.

In the next section we discuss the performance problems that can occur when sharing resources in a high performance computing cluster. We then discuss range of possibilities to address these problems. We then explain the solutions we are investigating and describe our experiments with them. We then conclude with a discussion of potential future work.

2 The Trouble With Sharing

In the early days of computing, time sharing schemes were devised so that users could share computers. Early systems were batch oriented and allowed only a single program at once. Later systems allowed multiple users to use the system at the same time. Today, virtually all operating systems (except for a few small real-time or embedded operating systems) support time sharing to allow users to run multiple programs at once. Depending on the nature of the load imposed by those programs, time sharing may help or hinder performance. For instance, if processes are

often blocked waiting for information from storage or the network, time sharing allows other processes to run when they can not. On the other hand, if processes are performing pure computation, the overhead of performing context switches between the processes will impose a performance penalty by consuming cycles the could otherwise have been used for useful computation. A common worst case scenario is running the system out of main memory and thus being forced to swap processes out. All systems resources can easily be consumed moving program state too and from disk.

In the area of high performance computing (HPC), time sharing is employed on individual systems by virtue of the fact that standard operating systems are generally used. Some some scheduling systems provide course-grained time sharing (typically on the order of hours or minutes), but fine-grained time sharing is generally considered too expensive due to the cost of maintaining synchronization between machines. The typical mode of operation is that users submit a job requesting to run one or more processes. The scheduler allocates space on one or more systems and typically provides some assistance in starting up those processes. The individual processes then run in a standard time sharing environment on each host until completion. If each process is assigned its own system, then application performance will be highly predicable. On the other hand, if resources on these systems are heavily contended, applications may have unpredictable run time. This is particularly problematic in parallel applications where work is divided between processes in a static manner. In that case, the process which ends up on the most overloaded system will hold up the whole computation. While such designs are generally considered a bad idea today, they remain common in practice.

This sort of unfairness creates difficulties for users, administrators, and scheduling systems. Users can not reliably predict when their job will finish which forces them to provide inaccurate estimates of job completion time. This in turn leads to reduced scheduler efficiency because schedulers can only make optimal decisions when they know when current processes will finish and how long queued jobs will run.

3 The Range of Possibilities

A number of possible techniques exist to isolate user jobs from each other. These options include classic time sharing based approaches, single application subclusters, and full or partial virtualization.

3.1 Time Sharing and Gang Scheduling

The historical solution to resource sharing– significantly predating[GE] Unix–is some form of time sharing. On a cluster this can either be done per-host or the cluster can be gang scheduled with the whole cluster dedicated to a particular job at a given time. Either method has the advantage that more jobs than can be simultaneously run can work toward partial results instead of one job blocking the next entirely. On the down side, switching contexts between jobs has significant costs.

Time sharing on individual hosts is a standard feature of all modern operating systems so it can be implemented with no administrator effort. However completion times in the face of unknown resource contention are highly unpredictable. With parallel jobs things are worse because in many algorithms, a single node can slow down the whole computation. If a lack of predictability of run times is acceptable, this approach is very effective at maximizing resource utilization.

Gang scheduling suffers less from contention and unpredictability since resources are typically wholly dedicated to a job during the (usually long) time slice. Context switching costs are much higher because complex state such as network connections may need to be created and destroyed. Also, memory resources used by gang scheduled processes are likely to be pushed out when they are not scheduled resulting in large swap in penalties when rescheduled. The largest draw back of gang scheduling is the general lack of available implementations. This lack seems to indicate that the HPC community is not sufficiently interested in the capability to deal with the complex issues involved in making it work for arbitrary applications.

3.2 Single Application (Sub-)Clusters

An approach at the opposite end of the spectrum from time sharing is single application clusters. In this approach, users are either given a private cluster for a period of time or are allocated clusters on demand using systems like EmuLab or Sun's Hedeby Project[Emulab, Hedeby]. Some labs with annual allocations for resources use the first variant.

This allows users complete control over how they run their jobs. They can let the systems run relatively idle for maximum predictability or heavily oversubscribe the system to get early results from many experiments. Since users are given their own systems, this allows more complete data security than a con-

ventional multi-user system. On the down side, users may oversubscribe node to the point of reducing overall throughput due to the cost of swapping. There is also no easy way to harvest under utilized resources unless users have a mix of job types. Additionally, this method only works well for relatively large jobs or projects.

3.3 Virtualization

One method of implementing per-application sub clusters is to use a virtualization system such as VMWare, Xen, or VirtualBox[VMWare, Xen, Virtualbox]. These systems can be deployed in ways that allow very rapid deployment of system images which means that virtual clusters can be deployed rapidly, potentially based on scheduler demand. Because multiple virtual images can be run on one machine at the same time, jobs can be given isolated environments with dependable performance while still allowing other uses such as cycle scavenging applications. Other benefits of virtualized systems include the ability to hide hardware details from applications and for applications to run in specially tailored operating environments. For example, running with an uncommon operating system version or configuration.

Virtualization does have a number of downsides which offset some of these benefits. Because hardware is hidden from the virtualized operating system, the OS can not take full advantage of it. A common example of this is high performance network devices. New network cards such as those from Neterion support allocating hardware queues to particular virtual images which can help in this area, but most current devices do not have such features. Additionally, there is some CPU, memory, and storage overhead from virtualization, CPU overhead is typically in the 5-20% range where as memory and storage overhead is often up to 100%. There may also be problems with licensing software in a virtualized environment, but this situation is improving over time.

3.4 Virtual Private Servers

The Internet hosting industry has developed an alternative to full virtualization which they use to provide virtual private servers. Their solution is to provide operating system enhancements which allows partitioning sets of processes off from the main system and other partitions. Implementations typically start with the file system virtualization provided by the `chroot()` system call and add new features to further isolate processes and prevent escape from the virtual root. The FreeBSD jail system is one such implementation. It restricts processes to a single IP address, reduces their ability to see processes and other objects outside a jail, and adds a set of restrictions on the root user within the jail. Another widely available implementation of this type of functionality is Solaris Zones which also includes CPU and memory use restrictions. Many hosting companies have developed their own versions in house.

The main advantage of these techniques is that that overhead is low compared to full virtualization. Some ISPs run hundreds or even thousands of virtual private servers on a single machine that could only handle tens of Xen or VMWare sessions. Additionally, the administrator or developer can choose how much they want to virtualize. For instance, multiple FreeBSD jails can share the same view of the file system. This allows processes to be separated and to run on different networks, but lets them share files.

The primary down side of this approach that since virtualization is incomplete, users have ability to affect each others processes to a greater extent than with full virtualization. An additional downside compared to full virtualization is that while virtual environments can differ, they are much more constrained in how they do so than with full virtualization. For example and FreeBSD jail can include a system image of an older version of FreeBSD (with some limitations) or from some Linux versions, but there is no support for Solaris or Windows images.

3.5 Resource Limits or Partitions

All modern Unix-like systems implement some sort of per-process resource limits. These generally include current memory use, total CPU time, number of open files, etc. The goal of these limits it to keep poorly designed programs or badly behaved users from exhausting system resources. Most cluster schedulers including Sun Grid Engine use these limits to partially enforce constraints on jobs. SGE can not rely on the limits because processes are allowed to create child process which are subject to the limits independently. As a result SGE tracks process resource use and keeps it's own list of total use for each job to enforce limits as needed.

Some operating systems provide the additional ability to limit resources for sets of processes. Irix has the concept of jobs which are subject to a collective resource limit[SGI-job]. Similarly, Solaris 9 has a concept of tasks[Sun]. A scheduler could take advantage of these limits to implement limits on job components. In general these limits are limits on total use over

time or on resource growth and as such do not protect against temporary exhaustion of resources such as CPU cycles or bandwidth to memory, disk, or network accessible resources. This means that they do not generally protect the predictability of application performance. Some exceptions such as per-process bandwidth limits have been demonstrated, but they are not widely deployed.

In addition to placing limits on resource use, some resources can be partitioned. For instance, most modern operating systems allow processes and their children to be tied to one or more CPUs. If this functionality is used to tie each job to a unique set of CPUs, the ability processes to interfere with each other will be significantly reduced. SGE once implemented such an allocation on some platforms, but the support was primarily implemented on Irix and no longer works. The biggest downside of this approach is that not all resources may be easily partitioned and often each resource much be partitioned though a different interface.

4 Our Experiments

With out diverse set of user applications, we need an approach to job isolation that works well for jobs using between one and several hundred processes with run times ranging from minutes to days and occasionally weeks. After examining of the pros and cons of the available techniques, we decided to investigate approaches based on resource partitioning and virtual private servers. We felt an approach which deployed application specific sub-clusters was an attractive approach from many perspectives, but it is too course grained without cheap, efficient virtualization. Our experience has shown that simply letting the OS try to handle the problem is fairly effective most of the time and thus we believe that appropriate partitions will solve many of our problems. On common issue is poorly designed jobs that consume more than their allotment of CPU cycles. Another problem is applications accidentally consuming all of a key resource such as local disk space. For example, an application was once filling `/tmp` on cluster nodes leading to an array of difficult to diagnose problems such as programs failing to perform necessary interprocess communications due to inability to create Unix domain sockets.

We have been experimenting in a few areas of resource partitioning. These include per-job memory based temporary files, binding of jobs to sets of CPUs, and partial file system virtualization using variant symbolic links. We have implemented these on our cluster using a wrapper script around the SGE shepherd program.

4.1 SGE Shepherd Wrapper

The SGE shepherd program (`sge_shepherd`) is executed by the SGE execution daemon (`sge_execd`) to manage and monitor the execution of the portions of an SGE job residing on a particular host. It handles both direct starting of single jobs and remote starting of components of parallel jobs. It has the task of setting resource limits on processes and tracking, monitoring, and reporting on their resource use through out the life of the job. We initially planned to modify the shepherd's job startup code to implement further resource restrictions, but fortunately discovered that SGE provides the ability to specify an alternate command that can act as a wrapper around the actually `sge_shepherd` program. We have implemented a modular wrapper system in ruby. Using a scripting language makes development simpler and allows us to easily leverage both library calls and command line tools.

The wrapper system allows wrapper modules to be registered and then have callbacks to set themselves up, to adjust the shepherd command line, and after the shepherd exits. This allows changes to be made such as mounting temporary file systems before execution and allows programs like `env`, `chroot`, or `jail` to wrap the `sge_shepherd` program and change it's environment directly.

4.2 Memory Backed Temporary Directories

The first wrapper we implemented extends the SGE feature of automatically created per-job temporary directories. By default, the SGE execution daemon creates a per-job sub-directory in an administrator defined directory and passes the path to the shepherd as part of its configuration. The shepherd then set the `TMPDIR` environmental variable so well designed applications place any temporary files in this directory. The directory is removed at the end of the job's execution so stray files left by crashes or inadequate cleanup algorithms are automatically removed. This feature is very helpful, but we found it does not go far enough in some cases. The problem is that all these temporary directories live on the same file system and thus a single job can run all jobs out of temporary space.

We have solved this problem with the `mdtmpdir` wrapper that mounts a memory backed file system (specifi-

cally a swap backed `md` device with UFS) over the top of the temporary directory. We have made the size a user requestable resource. We current set a hard limit on the total allocation size, but intend to replace that with an SGE load sensor based on swap levels in the future. The module registers a precmd hook which creates, formats, and mounts the file system and a postcmd hook which removes it. As a side benefit, many applications will see vastly improved temporary file performance due to the use of memory disks. We plan to deploy this hook on our cluster soon.

4.3 Variant Symbolic Links

While the memory `TMPDIR` wrapper solves a number of problems for us, it is not a complete solution to the problem of shared temporary space. It is not uncommon for applications, particularly locally written scripts or older code to hard code `/tmp` in paths. This can lead to running out of space in `/tmp` despite the existence of `TMPDIR`. Worse, some applications hard code full paths including `/tmp` which can result is corrupt results or bizarre failures if multiple application instances attempt to use the same paths. To work around this, we would like to virtualize `/tmp` for SGE jobs using variant symbolic links.

Variant symbolic links are a concept dating to at least Apollo Domain/OS[Wikipedia]. The idea is to make special symbolic links which contain variables that are expanded at run time and thus can point to different places for different processes. We have ported the variant implementation from DragonFlyBSD[DragonFly] to FreeBSD with significant modifications. These modifications include the ability to specify default values for links with the `%{VARIABLE:default-value}` syntax and a more secure precedence order between the variable scopes In our implementation, system wide variables override per-process one rather than the other way around. This means there is no risk of setuid binaries loading libraries from user controlled locations due to symlinks.

We intend to replace `/tmp` with a symlink with like `%{TMPDIR:/var/tmp}`. We will then set the per-process `TMPDIR` variable in the wrapper script so that it points to the memory based `TMPDIR`. That will result in all processes other than SGE jobs seeing `/tmp` and a symlink to `/var/tmp`, but SGE processes seeing it as a link to their own private directory. This configuration should fully resolve issues with contention over `/tmp` space (I/O bandwidth is another story).

4.4 CPU Set Allocator

With the introduction of CPU sets[FreeBSD-cpuset] and CPU affinity[FreeBSD-cpuaff] support in FreeBSD 7.1, it is possible to tie groups of processes to groups of CPUs. We have leveraged this support to allocate independent sets of CPUs to each job.

Our CPU set allocation algorithm is a relatively naive recursive one which does not take cache affinity into account. The algorithm first attempts to find smallest available range that the request will fit into. If that fails, it allocates the largest range and recursively requests an allocation for the remaining request size. We hope this results in minimal fragmentation over time, but we have not performed a detailed analysis.

We store CPU set allocations in `/var/run/sge_cpuset` and lock the file when allocating or deallocating ranges. The module registers a precmd hook which handles range allocation, a postcmd which deallocates ranges, and a cmdwrapper which invokes the `cpuset(1)` utility to restrict the `sge_shepherd` process and thus it's children to the allocated CPU range.

Since the point of the CPU set allocator is to isolator performance impacts, we conducted a experiment to verify this effect. Our test platform consisted of a system with dual Intel Xeon E5430 CPUs running at 2.66GHz for a total of 8 cores. The system has 16 GB of ram and 4 1TB SATA disks neither of which should not have a measurable impact on benchmark results. The system is running FreeBSD 7.1-PRERELEASE amd64 corresponding to subversion revision r182969. Sun Grid Engine 6.2 is installed. For our benchmark, we chose an implementation of the nqueens problem installed from the FreeBSD Ports Collection as `nqueens-1.0`. The executable is named `qn24b_base`. The program finds all possible layouts of N queens placed on an $N \times N$ chess board where no queen can capture another. Program performance is almost entirely dependent on integer CPU performance.

Our benchmark runs consisted of running one instance at a time of "`qn24b_base 18`" via SGE job submissions and using the reported elapsed time as our timing result. To provide competing load, we ran 0, 7, or 8 instances of "`qn24b_base 20`" in an SGE job which requested 7 of the 8 slots on the system. Each load process ran for more than an hour..

A summary of our results is presented in Table 1. The 7 load process case without CPU sets show the expected results that fully loading the system yields in small performance degradation and increase in variability. Likewise the 8 load process case yields a sig-

Table 1: Results of CPU set benchmark runs

	Without CPU sets			With CPU sets	
	Baseline	7 Load Procs	8 Load Procs	7 Load Procs	8 Load Procs
Runs	8	8	17	11	12
Average Run Time (sec)*	345.73	347.32	393.35	346.63	346.74
Standard Deviation	0.21	0.64	14.6	0.05	0.04
Difference from Baseline		0.59	46.63	†	†
Margin of Error		0.51	10.81	†	†
Percent Difference from Baseline		0.17%	13.45%	†	†

* Smaller numbers are better. † No difference at 95% confidence.

nificant performance degradation and corresponding increase in variability. With CPU sets enabled, no statistically valid difference can be demonstrated between the baseline and the 7 and 8 load process test cases and the variability of run times drops appreciably. These results demonstrate basic validity of our approach and confirm our hypothesis that CPU sets will benefit programs even when a node is not overloaded.

5 Future Work

A number areas of possible future work exist. Extending the current SGE shepherd wrapper framework to add more partitioning would be generally helpful. One we are particularly interested in is adding a wrapper to place jobs in individual chroots or jails, possibly with different user requested OS versions. This could enable a number of useful things including upgrading the kernel to take advantage to new performance features without exposing the users to disruptive user land changes. Users could also potentially request Linux jails for their jobs which would allow easier deployment of commercial applications. Potentially, tools such as DTrace could be used to analyze these jobs, something not possible on Linux due to licensing compatibility issues.

Other areas worth investigating are the creation of per-job VLANs for parallel jobs so jobs would be isolated from each other on the network and using the mandatory access control framework to better isolate jobs without the use of jails. Using features such as dummynet to allocate network bandwidth to jobs might also be interesting. For MPI traffic this would be fairly easy, but NFS would be more difficult due to the need to account for the actual process making the request. Similarly, it would be useful if disk bandwidth could be reserved similar to the guaranteed-rate I/O mechanisms in Irix[SGE-grio].

Also on the FreeBSD side, adding Irix-like job resource limits or Solaris like tasks would simplify tracking certain limits in SGE and could provide a useful place to hang per-job rate limits.

All in all, the use of resource partitioning and techniques from the virtual private server space is profitable space for further exploration in the quest to make clusters more useful and more manageable. We intend to continue exploring this area.

References

[Davis] Brooks Davis, Michael AuYeung, J. Matt Clark, Craig Lee, James Palko, Mark Thomas, *Reflections on Building a High-performance Computing Cluster Using FreeBSD*. Proceedings, AsiaBSDCon 2007.

[DragonFly] DragonFly BSD `http://www.dragonflybsd.org/about/history.shtml`

[FreeBSD-cpuaff] The FreeBSD Project, cpuset_getaffinity(2), *FreeBSD System Calls Manual*, `http://www.freebsd.org/cgi/man.cgi?query=cpuset_getaffinity&manpath=FreeBSD+8-current` March 29, 2008.

[FreeBSD-cpuset] The FreeBSD Project, cpuset(2), *FreeBSD System Calls Manual*, `http://www.freebsd.org/cgi/man.cgi?query=cpuset&manpath=FreeBSD+8-current` March 29, 2008.

[Emulab] White, Lepreau, Stoller, Ricci, Guruprasad, Newbold, Hibler, Barb, and Joglekar, *An Integrated Experimental Environment for Distributed Systems and Networks*, appeared at OSDI 2002, December 2002.

[GE] General Electric Computer Dept. Laboratory, *The Dartmouth Time-Sharing System – A Brief*

Description,
http://www.dtss.org/ge_dtss.php Sunnyvale California, 26 March 1965.

[Hedeby] The Hedeby Project
http://hedeby.sunsource.net/

[SGE] Sun Grid Engine Project
http://gridengine.sunsource.net/

[SGI-job] SGI, Inc. job_limits(5) *IRIX Admin: Resource Administration*
http://techpubs.sgi.com/library/tpl/cgi-bin/getdoc.cgi?cmd=getdoc&coll=0650&db=man&fname=5%20job_limits

[SGE-grio] SGI, Inc. grio(5) *IRIX Admin: Man Pages* http://techpubs.sgi.com/library/tpl/cgi-bin/getdoc.cgi?cmd=getdoc&coll=0650&db=man&fname=5%20grio

[Sun] Sun Microsystems. *System Administration Guide: Solaris Containers-Resource Management and Solaris Zones*. 2008.

[Wikipedia] Wikipedia contributors, "Symbolic link," *Wikipedia, The Free Encyclopedia*, http://en.wikipedia.org/w/index.php?title=Symbolic_link&oldid=234623190 (accessed September 19, 2008).

[Virtualbox] Sun xVM VirtualBox http://www.sun.com/software/products/virtualbox/

[VMWare] VMWare http://vmware.com/

[Xen] Xen Hypervisor http://xen.org/