New Issues and New Capabilities in HPC Scheduling with the Maui Scheduler

David B Jackson
Center for High Performance Computing,
University of Utah

I.Introduction

Much has changed in a few short years. Clusters have emerged from being a mere 'blip' on the radar screen of high performance computing to being a mainstream source of HPC cycles. These clusters offer enviable price/performance and are allowing almost ubiquitous access to HPC resources. To their advantage, in many regards, Linux clusters are quite similar to their Unix based MPP predecessors but they do possess a number of distinctions which provide unique opportunities for furthering HPC.

In most systems with some degree of complexity, the adage 'every cloud has a silver lining' may be rewritten as 'every silver lining brings at least some rain'. In HPC, every new advantage typically brings up a number of new issues which must be addressed. The Linux cluster is no exception. Its greatest strengths have already and will continue to move HPC in a slightly different direction than what has been the norm and this route change will require adjustments in the software infrastructure that is used to support these clusters.

Fortunately, open source code moves quickly, allowing new technologies to be adopted and new capabilities to be integrated. This paper will touch on some of the new realms of scheduler development required to meet the particular needs of Linux based clusters as well as the general HPC community. Specifically, it will focus on a few of the recent developments within the Maui Scheduler, describing the issues and resulting solutions.

II. HPC Scheduling Overview

Most HPC sites run with a batch system in place. The batch system allows a site to hide some of the complexity of the underlying architecture, enabling users to submit jobs to a site queue. The batch system is responsible for managing these jobs, locating and allocating the resources needed, staging data, and tracking job execution. These batch systems typically consist of two components: a resource manager and a scheduler. While the resource manager focuses on tracking and managing jobs and nodes, the scheduler is responsible for making decisions and directing the actions of the resource manager. It is the job of the scheduler to determine when, where, and how a job should run and then direct the resource manager appropriately.

In general, HPC schedulers have three primary objectives, specifically

- Provide traffic control
- Enforce mission policy
- Optimize resource usage

In a nutshell, the first objective prevents jobs from clobbering each other, using each other's resources, or otherwise interfering with one another. The second causes the scheduler to schedule jobs in accordance with potentially complicated sets of site mission policies. The third objective is for the scheduler to use all of the knowledge and freedom it has available to enforce objectives 1 and 2 in the most intelligent and optimal way, maximizing the return on investment of the underlying cluster resources. Typically, this involves maximizing system utilization, minimizing average job turnaround time, and maximizing fairness.

New sites often initiate a system with simple goals of maximizing system utilization. However, many quickly find that maximum system utilization is not the true end goal, but rather that greater value is realized through focused delivery of compute cycles to the right jobs so as to maximize the 'right' research and work. Maximizing research is not as easy as maximizing system utilization and often requires cycle distribution, quality of service, and allocation management.

III. Maui Overview

Maui[1] is an open source, integrated research and production scheduler now used on hundreds of HPC clusters and supercomputers worldwide. It has been used for years on high end SP2 and O2K supercomputers and has become widely accepted in the Linux cluster world. It was inspired by a need for a tool which enabled extensive site policy control, thoroughly integrated scheduling performance optimizations, and provided detailed information on how effectively the system was being managed at all levels. Maui extends the functionality of the underlying resource management system and currently interfaces with PBS, Loadleveler, and other systems. The resource manager continues to handle all job queuing and management, while Maui enables many new capabilities through simply directing the resource manager regarding when, where, and how to run and manipulate jobs.

Maui Scheduler’s design incorporates a number of key concepts described in minimal detail below.

Unified Queue

While Maui supports large numbers of queues or classes, its default configuration is that of a single global queue. It provides extensive throttling policies to impose desired resource limits on users, groups, and accounts of interest as well as elaborate job prioritization control to allow jobs to be ordered according to credentials, resources, history, desired service levels and other attributes.

Quality of Service

The quality of service (QOS) objects within Maui allow sites to grant access to special functions, resources, and levels of service to certain users. When submitting a job, users may select any QOS they have been granted access to, therefore taking advantage of these special privileges. Sites have the option of assigning a relative charge factor to each QOS allowing them to charge a higher rate for premium services or resource access.

Allocation Manager Interface

Maui provides interfaces to Qbank[2] and other allocation management systems, allowing sites elaborate control over the quantity of resources available to each user. These systems allow private and/or shared allocations to be created with independent expiration times and enable dynamic charge rates based on quantity and type of resources used, quality of service requested, time of day, and other factors. These interfaces are tightly coupled into Maui's scheduling algorithm allowing 'allocation reservations' to guarantee that users can never exceed their allocations. Qbank also allows multi-site allocation management, enabling sites to share, swap, forward, and otherwise manipulate allocations.

Fairshare

Fairshare[3] enables sites to control job decisions based on historical information. Sites can specify the timeframe, decay, and metric (ie, proc-seconds utilized, proc/memory seconds dedicated, etc) for determining fairshare usage. Sites can specify fairshare targets, ceilings, and floors on a per user, group, account, class, and QOS basis. They can also determine the fairshare enforcement policy, be it job priority adjustment, or job allow/reject.

Reservations

Reservations allow specific resources to be reserved for a particular timeframe. Access Control Lists can be enabled to grant access to these reserved resources to combinations of user, groups, accounts, classes, QOS's, and job types. These reservations can be created dynamically by administrators or configured to be automatically managed on a daily, weekly, or persistent basis by the scheduler. They can enable very powerful and flexible policies.

Simulation

Some of the greatest difficulties associated with managing a large HPC site is determining the answer to 'what if' questions: What happens if we adjust the constraints of this queue, or add these users, or temporarily remove these systems, or add more memory? Maui doubles as a simulator and will answer these 'what if' questions using your scheduling configuration, site workload, and system resources. The simulator can let you know the impact of many decisions on system utilization, average job turnaround, and distribution of cycles.

IV. Maui Development

Maui development has progressed rapidly in response to the needs of its user base. Linux clusters, in particular, have been both pushing the envelope and enabling use and extension of many latent scheduler features. This section will focus of some of the developments made within the last few months including managing heterogeneous resources, scalability enhancements, support for QOS based job preemption, and application level simulation.

Managing Heterogeneous Resources

The relatively low price of Linux clusters is, in large part, possible because of the commodity nature of the components. These commodity components must conform to protocols guaranteeing component interoperability. These protocols not only guarantee interoperability with current technologies, but also provide a great degree of backward compatibility. While this backward compatibility can be a significant advantage in terms of integrating current and previous hardware, this very integration can in fact decrease the overall job throughput of a system.

The problem arises with the use of parallel jobs. Many, even most, of these jobs are written with a very simple balanced task distribution model where it is assumed that each task has access to identical compute resources and is therefore capable of running an identical amount of work. Consequently, a parallel job's total workload is usually evenly divided amongst all job tasks. When identical resources are allocated, this approach works well. However, in a system with compute intensive jobs and a wide distribution of compute node processor speeds, this approach breaks down. An eight node job may allocate seven 1.2 GHz processors and one 300 MHz processor. If this job evenly distributes its workload, the seven tasks running on the 1.2 GHz processors will complete early leaving them allocated, but idle, for 75% of the life of the job. While this is a somewhat extreme example, it is indicative of what is actually seen.

Sites generally address this problem in one of three ways. First, they may choose to maintain separate systems associated with each compute node generation. This prevents the problem because each system is homogeneous. However, it also prevents the site from utilizing the aggregate power of the various systems and precludes load balancing across the systems.

A second approach is to integrate all resources into a single large system and allow users to specify a particular processor speed of interest. This works somewhat better in that users can determine the sensitivity of their application to processor speed variations and act accordingly. The problem is that most users are classified in the range of lazy to very, very lazy and will not specify processor speeds for their jobs even if doing so would be advantageous. Problems also exist for those users who do specify processor speeds. Since the user must specify his job constraints at job submission time, the user’s decision is locked in and may not be changed even if a better proc speed selection is available. For example, a user may request 800 MHz nodes, which may prevent the scheduler from utilizing idle 1 GHz nodes when no 800 MHz nodes are available.

The third common approach is to simply ignore the problem. Ignoring the problem is not as bad as it sounds, allowing the site to maximize scheduling efficiency while accepting the job efficiency losses. In some cases this results in higher overall utilization than could be accomplished if the system were partitioned.

Each of these approaches results in tradeoffs between scheduling efficiency and job efficiency. The product of these two values is termed 'true system efficiency' and is directly related to job throughput. Maximizing true system efficiency is the real goal and involves finding the right combination of scheduling and job efficiencies. In attempting to maximize this value, a new job constraint, called a 'node set', was created. Quite simply, this node set allows a user to request a set of nodes with a 'common' processor speed, not necessarily a 'particular' one. This approach is similar to approach two in that jobs may specifically request it, but does not result in significant scheduling performance losses because it does not force the user to over-specify his job constraints.

This capability was generalized to allow sites to mandate common sets for all jobs or allow common set selection on a job-by-job basis. It allows users to specify the attribute type that will be used to generate the feasible sets allowing attributes such as processor speed, network interface, or node feature. It also allows users to specify a subset of attribute values from which their job may be selected and specify whether or not the scheduler should select nodes which meet any of the attribute value criteria or which meet a common attribute value. For example, a user may request that a job run on any nodes with processor speed of 450, 550, or 600 MHz, or any combination of these speeds. Alternatively, the user may also request that his job run only on a set of resources with a common processor speed where only processor speeds of 450, 550, or 600 MHz may be considered.

Simulations and production use have shown that the use of processor speed based node sets can noticeably improve true system efficiency and job throughput. Research involving the CHPC 266 processor Icebox cluster demonstrated job throughput gains of over 10%[4]. Modification of the base algorithm has been investigated, allowing best effort node set scheduling (i.e., schedule on a 'pure' node set if possible, but on the 'best' available node set otherwise.) Development is under way to extend this capability to allow tolerances to be specified, (i.e., I'd like 450MHz +- 50MHz or I'd like to allocate nodes within a 10% processor speed range.)

The node set capability has definitely proven its value in terms of processor speed heterogeneity. Research is now under way to investigate its value in a heterogeneous network environment.

Scalability Enhancements

The price/performance advantages of Linux clusters have resulted in truly massive systems. The same amount of procurement money is translating into systems many times larger than previously possible. Consequently, multi-hundred and even multi-thousand processor systems are springing up around the world. Not only are the system sizes increasing, so to are the queue sizes. While scalability enhancements are not very exciting, they are mandatory to maintaining smooth operation on ever growing systems.

The current version of Maui, in its default configuration, supports over 4000 simultaneous jobs, 2000 nodes, and up to 500,000 processors. These limits represent significant improvements over those mandated only a year ago. In fact, just last year, these new limitations seemed adequately lofty so as to not constrain any site in the near term future. However, sites with nearly 20,000 simultaneously queued jobs are already in operation. Sites with 4000 compute nodes are already being planned.

Schedulers, by their nature perform best with global information and the easiest implementation of a globally aware scheduler is that of a non-distributed, centralized daemon. While, at some point, this centralized approach will break down, the centralized scheduler approach will most likely continue to function well for a number of years, surviving perhaps one more full order of magnitude increase in the number of jobs and nodes. However, to handle this scaling, continued enhancements need to be made in the following areas:

* Compressing resource and job data communicated between batch system components.
* Caching a greater portion of processed information.
* Improving techniques for 'best effort' scheduling.
* Minimizing 'in core' memory representations of jobs, nodes, and policies.
* Managing resources as dynamic groups rather than as individuals.

Each of these techniques is already being developed and to some extent utilized. Simultaneously, research is under way to enable distributed scheduler hierarchies and more generally, interfaces to metascheduling systems responsible for distributing workload out across the grid.

QOS Based Job Preemption

HPC workloads generally consist of a highly heterogeneous mix of jobs, with variations in terms of submitting user, job size, job duration, desired QOS, and other parameters. This workload heterogeneity results in two significant problems. The first involves maintaining maximum system utilization while still providing quick turnaround time for short running development jobs. The second arises with the need to provide improved service in terms of average queue time or expansion factor for certain high priority or special quality of service jobs.

Current solutions to both of these problems are very similar in that they require setting aside a subset of available resources for use by certain jobs, be they short development jobs, or special QOS jobs. To date, most sites have resorted to some form of logical partitioning, via queues, classes, or reservations to set aside these resources and have just accepted the resulting resource fragmentation and accompanying losses in system utilization. Maui's floating, time based reservations allowed improved resource utilization in these situations, but still suffered from the underlying requirement of dedicating idle resources for jobs 'which might show up'. The lack of information regarding jobs that have not yet been submitted prevents the scheduler from intelligently utilizing many of its available resources. The scheduler is essentially forced to be pessimistic, idling a block of resources 'just in case'.

Preemption provides a powerful tool that enables an alternative to reserving blocks of resources. Preemption comes in many flavors, including suspend-resume, checkpoint-restart, and kill-restart. All allow the scheduler the option of terminating an active job almost immediately and transferring its resources to another job. With the introduction of preemption, the scheduler can now be optimistic, starting all jobs it determines should run without concern for 'over the horizon' jobs. If a special QOS or short development job arrives, a low priority job may be preempted. If not, the job runs to completion.

While preemption may result in some wasted cycles, reasonable preemption algorithms can prevent these losses from exceeding the losses associated with reservations required to support special jobs turnaround time. Recent enhancements to Maui allow preemption to be used in 3 primary areas:

* QOS based preemption
* Quick turnaround based preemption
* Preemption based backfill

The first feature allows some QOS objects to be marked as a preemptor, others to be marked as a preemptee, and still others not marked at all. Simply put, this allows jobs with a preemptor QOS to preempt jobs with a preemptee QOS. This feature can be used for several purposes and in several configurations including the support of 'bottom feeder' jobs. These jobs typically consume massive amounts of CPU and will burn every cycle delivered to them. A site using such a system can maintain virtually 100% system utilization with these bottom feeder jobs using every cycle not consumed by a priority job. However, as in Condor[5] systems, with preemption in place, these jobs will always be automatically vacated and will not delay the start of any incoming priority job.

'Quick turnaround' based preemption allows a site to create a reservation that sets aside a set of resources for quick turnaround jobs. If the appropriate jobs are queued, Maui will run them on these reserved resources. However, if they are not, Maui will start other jobs on these resources. If a quick turnaround job shows up, Maui will preempt jobs using resources in the 'quick turnaround' reservation to allow these resources to be allocated by the appropriate job. Thus, Maui will optimistically attempt to use all idle resources and transfer these resources via preemption as needed, eliminating the wasting of cycles required to handle the 'just in case' scenario.

The final feature is preemption based backfill. In classic backfill, high priority jobs that cannot run are given reservations to guarantee start time and prevent starvation. However, wallclock limit accuracies often cause the reservations to be made for times much later than is actually possible resulting in system losses. These inaccuracies also result in the potential for backfill jobs to actually delay priority jobs. With preemption based backfill, no priority based reservations are made. Rather, backfill jobs are marked as preemptible and priority jobs are marked as preemptors. Thus, when resources do become available for a priority job to finally run, Maui allocates idle resources as available and preempts backfill resources as needed. Research to date has proven this approach to be very competitive with classic backfill and it is hoped that ongoing work will be able to improve the effectiveness of this class of algorithms even further.

Application Level Simulations

When planning for the future, information is critical. Maui's simulation capabilities were touched upon previously, and while this ability provides valuable and accurate information, its applicability is limited. Currently, job traces record all scheduler relevant aspects of a job including resources requested and resources used. Also, resource traces record all scheduling relevant aspects of the compute resources available on a system. During a simulation run, Maui simply replays the workload information, submitting jobs as they were actually submitted. It allocates the requested resources to the job according to the specified policies and consumes the resources as they were consumed when the job actually ran. While Maui is able to scale the duration of jobs according to the processor speed of the resources allocated, outside of this it simply duplicates job behavior identically.

The problem with this approach is that while it models behavior at the scheduler level very well, it fails to mimic the behavior within an application and within non-processor based constraining system resources. In particular, the current approach would fail to properly represent the effects of network contention within a site and its corresponding effects on job efficiency and run time. Issues like memory contention are not modeled, nor is the impact of a bottlenecking hierarchical storage system. Further, the impacts of malleable and dynamic jobs are only partially modeled under this approach.

What is needed is a more advanced simulation model. Recent changes to the Maui scheduler enable a new simulation model that can emulate application level behavior to a much higher degree. In the new model, additional information regarding application type is recorded. Associated with each application type is an application model. In its simplest incarnation, this model indicates to the scheduler what resources were consumed each iteration and determines when the job completes. When Maui starts a job in simulation mode, it calls an application 'initialize' routine. Then, from iteration to iteration, Maui calls the application 'update' routine, which adjusts job resource consumption, state, and behavior. This routine terminates the job at the appropriate time at which point Maui calls the associated 'finalize' routine.

While this model is very simple, it allows the creation of independent 'application simulation' libraries which can be distributed and shared amongst interested parties. While it is envisioned that these application simulation models will start off simply, they can quickly evolve to encompass more real-world application behavior. Also, with their simplicity, sets of these application emulation modules could quickly be created to focus research exclusively on one particular area. For example, emulating network contention would be a fairly straightforward process of representing inter-task communication, tracking total network utilization, and imposing the effects of network based job slowdowns. Dynamic jobs could scale themselves according to the number of tasks present and the amount of memory available to each task. The impact of a particular hierarchical storage system could be modeled by creating application models that only accurately reproduce the effects of data file migration at job start and completion.

This model is not the ultimate solution to HPC simulation and the work in this area is only beginning. Development of these application simulation libraries will be a community effort guided by experts in the various applicable fields. Still, while it may take years for these libraries to mature, this approach provides a significant leap beyond what is currently available. And while comprehensive simulators may be years off or may never develop at all, focused application simulators can provide answers to many existing questions with a level of accuracy and detail not possible through other existing methods.

Conclusions

HPC Linux clusters are in a fortunate position of inheriting many quality open source tools developed for use on traditional supercomputer systems. These tools have already integrated the lessons learned from these earlier systems and are adapting themselves to the particular needs of Linux clusters. These clusters are pushing HPC development and growth in new areas. While great progress has been made, the frantic pace of HPC cluster evolution will ensure that much opportunity will remain to make significant advances in HPC middleware and tools.

More information regarding the Maui Scheduler, its use, and ongoing research and development can be found at http://www.supercluster.org or by contacting the author at jacksond@supercluster.org.

Bibliography

[1] Maui Scheduler Homepage, http://www.supercluster.org/maui
[2] Qbank Homepage, http://www.emsl.pnl.gov:2080/docs/mscf/qbank-2.8
[3] D Jackson, Q Snell, "Core Algorithms of the Maui Scheduler", IEEE Sigmetrics 2001, Advanced Scheduling Workshop, Cambridge, Ma.
[4] D Jackson, B Haymore, et al, "Improving Cluster Utilization Through Node Set Based Allocation Policies", International Conference on Parallel Processing, 2001, Valencia, Spain.
[5] "Condor Project Homepage", http://www.cs.wisc.edu/condor.