Coasters: uniform resource provisioning and access for clouds and grids

Mihael Hategan
Computation Institute
University of Chicago
Chicago, IL USA
hategan@mcs.anl.gov

Justin Wozniak and Ketan Maheshwari
Mathematics and Computer Science Division
Argonne National Laboratory
Argonne, IL USA
{wozniak,ketan}@mcs.anl.gov

Abstract

In this paper we present the Coaster System. It is an automatically-deployed node provisioning (Pilot Job) system for grids, clouds, and ad-hoc desktop-computer networks supporting file staging, on-demand opportunistic multi-node allocation, remote logging, and remote monitoring.

The Coaster System has been previously [32] shown to work at scales of thousands of cores. It has been used since 2009 for applications in fields that include biochemistry, earth systems science, energy modeling, and neuroscience.

The system has been used successfully on the Open Science Grid, the TeraGrid [1], supercomputers (IBM Blue Gene/P [15], Cray XT and XE systems [5], and Sun Constellation [26]), a number of smaller clusters, and three cloud infrastructures (BioNimbus [2], Future Grid [20] and Amazon EC2 [16]).

1. Introduction

Grid computing brought about the globalization of high performance computing, characterized by the move from custom local clusters to supercomputing centers accessible openly by scientists through standardized interfaces [9, 28]. In recent years Cloud Computing has emerged with the purpose of allowing users to deploy custom environments, in particular custom operating systems through virtualization [20]. A parallel development is the creation of systems capable of expressing and executing complex process pipelines while opportunistically exploiting the new resources [7, 32]. These, we argue, are part of a trend aimed at commoditizing High Performance Computing with the purpose of providing easier and more flexible computing solutions for users. Nonetheless, shadows of the old systems remain. Grid sites necessarily employ Batch Queuing Systems, optimized for monolithic jobs, and shared filesystems which are required to enforce consistency of distributed and concurrent file access, but not without performance costs. On the other end, nodes on Computing Clouds do not, by default, offer much in the way of job management or file management. In this paper we analyze some of these issues and present a system aimed at taming the conflict between the desire for easy and efficient on-demand process pipeline execution and the current Grid and Cloud infrastructure.

In Section 2 we provide a little background into the origins of the system. Section 3 contains a brief architectural description. More detailed descriptions about the various components involved are provided in Section 4. We provide some performance measurements in Section 5. Related work is discussed in Section 7 and we conclude with possible paths for improvement in Section 8.

2. Motivation

Our initial motivation for the Coaster System (colloquially ”Coasters”) was prompted by the needs of the Swift system [32]. Swift is a scripting language oriented to scientific computing which can automatically parallelize the execution of scripts and distribute tasks to various resources. It is also a highly flexible, Turing complete language; as a consequence, static analysis cannot in general be used to determine whether a run will complete, how many jobs a run consists of, or what the exact dependencies between those jobs are. Apart from the trivial case when all the jobs in a run are independent, dependencies between jobs mean that a given resource on which a job executed can potentially be re-used for other jobs. Even in the independent job case, if fewer resources are available than what is needed to run all jobs in parallel, some or all resources will need to be re-used. On traditional cluster computers and clouds, and without additional middleware, re-use of resources suffers from non-proportionalities in job queuing times vs. job requested wall times.

This is especially obvious with small (less than a few minutes) wall time jobs. In other words, we observed that the queuing time for a 10 minute job tends to be smaller than ten times the queuing time for a one minute job. We believe that this is too high of a price to pay for the difference between knowing one's (future) workload with certainty and a high confidence about the shape of that workload.

The challenges faced when scheduling jobs on a busy system are illustrated in Figure 1. The graph shows the measured queuing times for jobs of various requested wall times. The results were obtained on TeraGrid's TACC Ranger system over a period of approximately 7 hours. This shows that increasing the wall time request does not correlate with increasing queuing time.

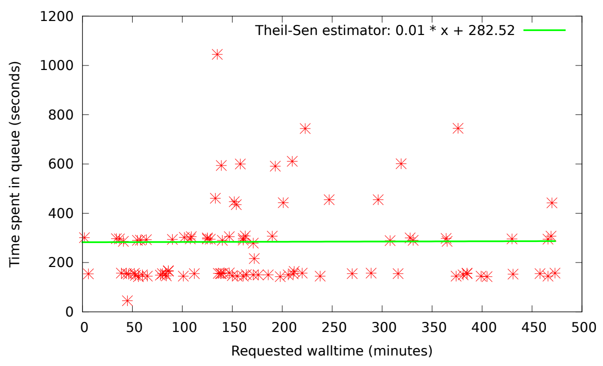

Figure 1. Queuing times for a single job as a function of walltime on Ranger

Although Swift [32] has been observed to benefit from employing Coasters, the two are distinct in scope and can be used independently. Specifically, Swift is used to describe the high level interconnects between application invocations, while the Coaster System is a job execution engine, with no knowledge of or ability to manage job dependencies.

3. Architecture

The Coaster System was designed to meet both usability and performance goals. In terms of usability, it was deemed necessary to not require users to log into remote systems and prepare services or configure compute nodes. After all, when plugging a desk lamp into a power outlet, one is not required to adjust knobs inside the power plant or make a separate contract with the electricity provider before turning the lamp on. In terms of performance, the system essentially enables a live connection between client and compute node. To take the lamp metaphor a bit further, one does not need to carry batteries from the power plant home in order to use the lamp, but a complex system of wires, transformers, breakers and switches takes care of the transport.

A typical use of Coasters is diagrammed in Figure 2. The *Coaster System* consists of a service (the *Coaster Service*), which typically runs on a cluster head node, a *Coaster Client* which provides an API used to establish a connection to and communicate with a service, and a *Coaster Worker* component which runs on compute nodes and executes the actual jobs. The *Coaster Client* - *Coaster Service* and *Coaster Service* - *Coaster Worker* communication is done through a custom protocol that allows re-using a single persistent connection between two given endpoints in order to reduce overheads typically associated with connection establishment. These overheads include the TCP handshake (requiring a full round-trip), TCP buffers, and the SSL session establishment (a processor intensive operation).

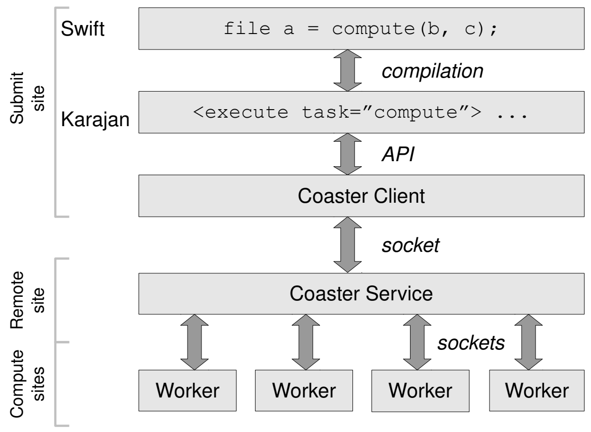

Figure 2. Coasters as used by Swift

The *Coaster Service* queues jobs and contains logic for allocating compute nodes based on the characteristics of the queued jobs. A number of providers available from the Java CoG Kit Abstraction Library [31] (e.g. GRAM [6], WS-GRAM, SSH [21], PBS [13], SGE [23], Cobalt [3], Condor [17]) can be used to interact with the queuing system, either directly or indirectly (as is the case of Globus GRAM). The *Coaster Service* can also act as a proxy to allow client file staging to happen even in cluster environments where a compute node cannot directly connect to hosts outside of the cluster.

4. Implementation

The bulk of the *Coaster System* is implemented in Java, which allows the *Coaster Service* to be deployed automatically on a various range of resources. The use of Perl for the worker script enables us to maintain the ability to perform automated deployment while still being able to run on clusters with limited environments, such as the Argonne Blue Gene/P.

4.1. Deployment

The *Coaster Service* is deployed automatically on target clusters. A bootstrap application is submitted to the target cluster using one of the available remote providers from Java CoG. In parallel, the *Coaster Client* starts a simple HTTP server. The bootstrap application then connects to the *Coaster Client* and downloads the main *Coaster Service* along with all its dependencies. The files that make up the *Coaster Service* are checksummed and cached by the bootstrap application. Subsequent deployments verify the checksums of the files and downloads are skipped for files that have not been modified.

We found that the automatic deployment provides some advantages. First, it facilitates a fluid distributed environment. Updates to the service can be deployed expediently, outside of the infrequent cycles characteristic of large grids. Second, it gives the users the freedom to use and modify the software as they see fit.

In an opinion article [25] Richard Stallman argued that *Software as a Service* implies a loss of freedom with respect to software, because a user of *SaaS* has no control over the software that implements those services. While a distinction between Grid Services and *SaaS* may exist, the reality of the lack of user control over the deployments of Grid Services remains, despite the fact that the software and/or protocols are open and/or open-sourced.

In addition to automatic deployment, the *Coaster Service* can also be started manually, as a stand-alone service.

4.2. Communication Library

The *Coaster System* is implemented on top of a simple messaging library. At the logical level, the messaging library provides communication channels between two endpoints. Channels carry messages which are logically grouped into conversations using tags. Conversations typically consist of commands and replies to commands, each of which can have multiple parts. Any number of concurrent conversations can be active at a time on a channel, which means that network latencies can be parallelized with minimal resource consumption. The messaging library does not specify a data format for the messages.

At the implementation level, channels can use plain TCP sockets, GSI-wrapped SSL sockets, UDP messages, and in-memory pipes (for channels whose endpoints live in the same JVM). For connection oriented protocols (TCP and GSI over TCP) a number of strategies can be used to establish and maintain connections.

4.3. Job Life-cycle

The *Coaster Client* provides an implementation of a Java CoG Job Submission Provider. From a user's perspective, this means that using Coasters can be nearly as simple as changing a string parameter from "GT2" to "coasters". When a job is submitted using the aforementioned provider, the coaster implementation first looks to see if a previously used *Coaster Service* exists for the target site. If not, it uses the mechanism described in Section 4.1 to start one. Once a *Coaster Service* is started on the target site, the job is submitted to it. The *Coaster Service* maintains an internal queue in which jobs are stored until they can be executed on one of the compute nodes. A periodic worker allocation mechanism, described in Section 4.5, determines the count, walltime, and partitioning of *Coaster Workers*, and subsequently submits them to the local resource manager. After *Coaster Workers* are started, a fast submission loop routes jobs to individual workers. Upon completion of a job by a *Coaster Worker*, a notification is sent to the *Coaster Service*, which in turn notifies the *Coaster Client* of the job status.

4.4. File Staging and Clean-up

Traditionally, the Swift runtime model required the existence of a shared file system as a temporary storage for application data between the client and the compute nodes. This requirement proved to be problematic on larger clusters, where shared file system load and contention caused unnecessary delays. Swift provides a programming model similar to a purely functional language, in which data is immutable. The general requirement placed upon shared file systems to enforce file level consistency in the face of distributed and concurrent reading and writing is both unnecessary for Swift's immutable data model and unavoidable in a typical cluster environment in which few universal high level assertions can be made on the data access patterns. In other words, one cannot temporarily disable the enforcement of file level consistency even when it is known that concurrent reads/writes will not occur. The issue becomes particularly problematic with petascale computers, such as the BG/P, where slowdowns can be significant even when a file is only read from a large number of compute nodes.

The *Coaster System* supports file staging directly to and from the compute node. A choice of mechanisms is available for reading and writing stage-in files and stage-out files respectively. These mechanisms, pictured in Figure 3, are as follows:

Proxy In proxy mode, files are accessed on the client side. The *Coaster Service*, typically living on a login node, acts as a proxy between the *Coaster Worker* and the *Coaster Client*. This allows staging to be done even if the compute nodes do not have direct access to the world outside of a cluster. The actual file data is sent using the coaster protocol.

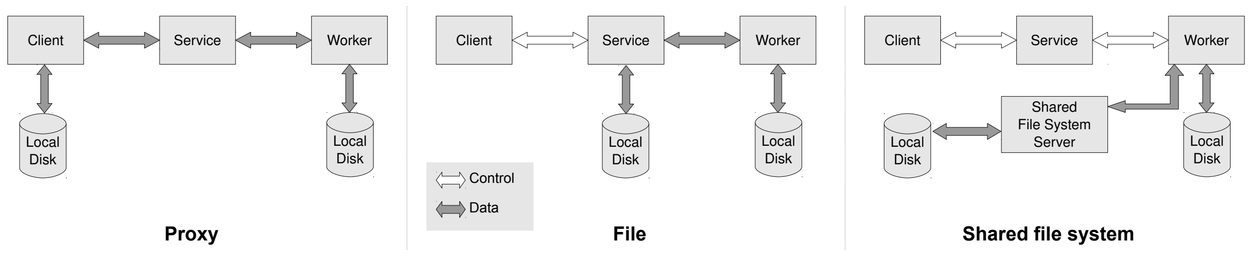

Figure 3. The different I/O modes

File In file mode, files are assumed to be accessible on the node on which the *Coaster Service* runs (typically a login node). The *Coaster Client* is not involved in the staging process. As in the proxy case, file data is sent using the coaster protocol.

SFS Short for "shared file system", it allows the *Coaster Worker* to directly stage the data from/to a file system that is accessible to the compute node. The file data is copied using standard operating system mechanisms.

In addition to file staging, cleanup directives are also supported. A flag, currently only accessible by modifying code, controls whether cleanups are synchronous or not. In synchronous mode, a job is marked as *completed* only after the cleanup completes. In asynchronous mode, a job is marked as *completed* before the cleanup is performed, which provides better performance when the job walltime is comparable to the cleanup time, but can interfere with job management systems that tightly control disk space usage.

4.5. Worker Allocation

In part, the worker allocation problem is roughly equivalent to a (2-dimensional) box-packing problem in which a multi-node *Worker Block* represents a box (with the number workers being one dimension and the walltime being the other). Further constrains are imposed by the architecture of the target cluster and by characteristics of the queuing system. Relevant such constraints were identified as follows:

- There may be a limited number of jobs that can be submitted to the queuing system. This means that the system must have the ability to start multiple workers in one job (i.e., a *Worker Block*)
- The compute nodes may have multiple cores or the cluster may require a certain number of nodes to be allocated at one time (*worker granularity*).
- In order to provide any benefits from avoiding queuing times, the walltimes of the worker jobs must be larger than the actual job walltimes when the actual job walltimes are comparable with the queuing times. The amount of times the worker walltime is larger than the actual job walltime is termed *overallocation*.

In addition, we identified a number of desirable characteristics:

- The *Worker Blocks* should be allocated conservatively in order to accommodate the possibility of future loads.
- If possible, the sizes (in CPU-hours) of the *Worker Blocks* should be spread out in order to both find free spots in the cluster queues (thus improving cluster utilization) and optimize the job execution process.

By prioritizing the above constraints, the following algorithm emerged (though no proof is provided and no claim is made that this is the only reasonable algorithm given the above constraints):

1. Iteratively process the queued jobs as follows:

   1 Sort the jobs according to their walltimes

   2 Determine the number of *Worker Blocks* to be used by multiplying a fraction parameter (named *allocationStepSize*) by the number of free *Worker Block* slots (determined by subtracting the number of currently active *Worker Blocks* from the site specific limit for the number of *Worker Blocks*).

   3 Compute the *overallocation* of each job. A number of choices may be available here. Given that the overallocation is more important for short jobs (with respect to typical queuing times) and almost irrelevant for long jobs, we chose an exponential decay model in which the user specifies the overallocation for 1 second jobs (*lowOverallocation*) as well as the overallocation for infinitely long jobs (*highOverallocation*). In-between, the overallocation is calculated as:

$$f(t) = (O_l - O_h)e^{-t\beta} + O_h$$

where O_l is the *lowOverallocation*, O_h is the *highOverallocation*, and β is a decay rate parameter (*overallocationDecayFactor*).

4 Sum up all the over-allocated walltimes to get the total CPU time required.

5 Partition the total CPU time required in a number of blocks equal to the number of *Worker Blocks* used in this round. The partitioning is done based on a *spread* parameter. A zero spread indicates that all blocks should have equal size, while a spread of 1 indicates that the block sizes should be linearly spread between the minimum possible size and the maximum possible size (subject to the constraint that the sum of the block sizes is equal to the total CPU time required).

6 Adjust the partitioned block size to fit the specified worker granularity as well as the overallocated job walltimes.

2. Repeat

The *Worker Block* allocation algorithm requires a traversal of the entire job queue: a O(n) operation in the number of queued jobs. It is consequently unsuitable to re-run it for every job that is added to the queue. In addition, the job walltimes are, as usual, approximate, making any exact determination of the *Worker Block* sizes difficult. Instead, the algorithm above tries to make a "reasonable" allocation.

After a *Worker Block* allocation is done and the workers are started, a fast dispatching algorithm distributes actual jobs to workers. The dispatching algorithm iterates through the available workers and for each worker selects the largest possible job that can fit in the remaining run-time for the worker. This solution is similar to the greedy solution to the unbounded knapsack problem. This choice works under the assumption that the entire workload consists of a sufficiently large number of jobs of varied sizes. However, as opposed to the standard knapsack problem, we have the choice of adjusting the knapsack sizes by terminating the *Worker Block* jobs early if it turns out that the workload is lower than expected.

5. Performance

The primary performance metric in the Coaster System is the ability of the system to run a large number of jobs without reducing utilization. In the following subsection we evaluate the system in various settings, with a focus on multi-user cluster environments.

5.1 Submission Rate

A raw submission rate of 400 jobs/second can be achieved even on modest networks, such as a DSL connection. Here, raw submission rate is defined as the time between right before the first job is submitted to the time right after the last job is acknowledged as being submitted by the server.

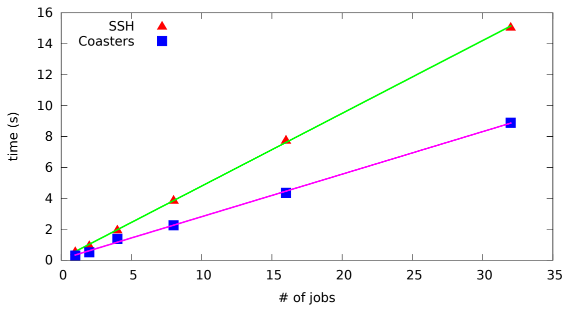

Figure 4. Comparison of sequential job run times between SSH and Coasters

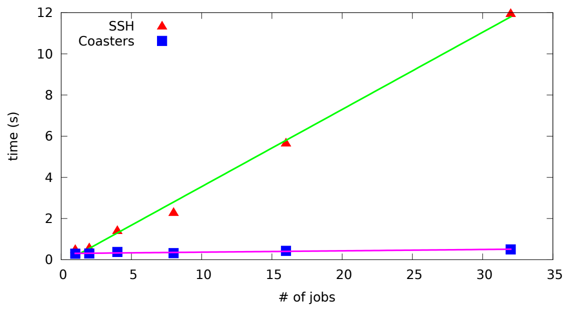

Figure 5. Comparison of parallel job run times between SSH and Coasters

5.2 Serial Job Performance

A comparison of job completion rates between SSH and Coasters without any queuing systems involved is shown in Figure 4 for sequential no-op jobs and in Figure 5 for parallel no-op jobs. The times, in both cases, were measured after an initial pre-load job was successfully completed. As such, they do not include the one-time SSH authentication overhead or, in the case of Coasters, the deployment and connection overheads.

5.3 Application

In this section, we describe an application *modFTDock*, ported to the Bionimbus cloud (*www.bionimbus.org*) using the Coasters System. Bionimbus is an open source cloud-based system for managing, analyzing and sharing genomic data that has been developed by the Institute for Genomics and Systems Biology (IGSB) at the University of Chicago. The model of computation on the Bionimbus cloud is based on Virtual Machines (VMs) provisioning. Users can create an "image" of a VM which can be instantiated. Images running the Linux OS with varying compute cores (1 to 4), memory (3-15GB) are available. The data storage capacities are fixed to 10GB.

ModFTDock is a protein-RNA docking application. The application is based on a molecular docking algorithm that can identify protein molecules having a high structural affinity towards a specific RNA molecule. A large repository of protein molecules are processed by the application in a series of independent (hence parallel) docking operations. The resulting jobs are short. The distribution of run times is shown in Figure 7. The distribution is close to a normal distribution with a peak of over 17,500 jobs at 15 seconds of run time. The application is constructed as a SwiftScript which integrates a large number of such invocations in a loosely coupled manner.

Owing to this pattern of computation, a characteristic large number of highly parallel tasks can be generated for execution. Coasters were used to rapidly dispatch hundreds of modFTDock tasks over the 20 nodes (20x4=80 cores) of the Bionimbus Cloud resulting in the execution of 100,000 tasks in one run. A manual Coaster Service was run on the Bionimbus Cloud head node. Coaster Workers were started on the VMs and instructed to connect back to the service via a reverse SSH tunnel in due to firewalls restricting the root-only accessible TCP ports on VMs. This setup allowed for 20 VMs to be up for the entire duration (4 hours) of the experiment. The plot in figure 6 illustrates a rapid ramping up of tasks and a sustained utilization of the 80 cores over a time period of 200 minutes. This resulted in an average job rate of 8.3 jobs per second.

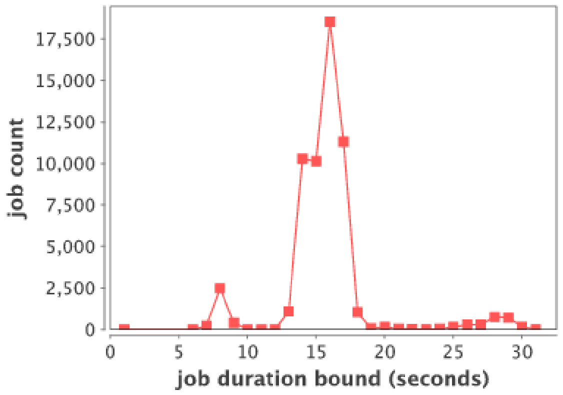

Figure 7. Histogram of job run times for ModFTDock on Bionimbus Cloud

6. Impact on Queuing Systems

An issue voiced on occasion by system administrators in regards to similar systems is that they circumvent the queuing system policies which are tailored for traditional jobs with predictable number of nodes and wall times. We believe that, in fact, our system combined with the Many Task Computing paradigm has the potential of improving overall cluster utilization.

If backfilling is used, the ability of a queuing system to utilize the unused "holes" in the schedule [19] depends on a diverse spread of the queued jobs. If queued jobs consist exclusively of large blocks on the processor/time space, such holes are unavoidable. We reason that the ability to divide one's workload into small pieces which can then be flexibly combined can provide such a diverse spread.

Actual run times are impossible to predict with certainty in the general case, in part due to the undecidability of the Halting Problem and in part due to variations in the load of various cluster subsystems which affect job run times. However, users need to specify an upper bound on the job run times. User-supplied wall times are constrained by the expectation that over-estimation will lead to increased queue wait times and that under-estimation will lead to jobs being killed. As mentioned in [29], seemingly conflicting results have been obtained on the issue of the impact of estimated job run time accuracy on cluster utilization. In some studies inaccuracies lead to better results ([19, 33]), while in others accurate predictions allow for better schedules ([14, 11]).

Under the assumpion that user predicted job run times are statistically proportional to the actual run time, a monolithic job will tend to have, on average, a larger inaccuracy than what is achievable when the monolithic job is broken down into shorter pieces. This is because when dynamically combining shorter jobs, the predicted run time inaccuracy is limited to the inaccuracy of a single short job. The Coaster System can essentially move jobs from one block to another in order to follow the requested wall times as closely as the short job advertised wall times permit. This, in turn, means that more accurate wall time estimations are possible with such a system. If a queuing system can benefit from more accurate run time estimations, the scheduling flexibility of the Coaster System can improve utilization. If, on the other hand, better results are obtained by precise overestimations of actual run time (as is suggested by [24]), then that can be easily derived from an accurate prediction using a multiplicative factor.

7. Related Work

The development of grid standards and wide area networks drove the creation of systems designed to enable the

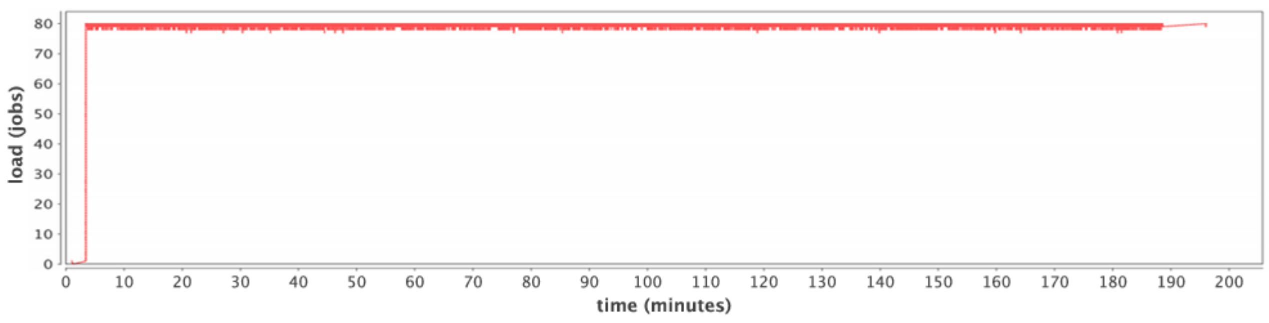

Figure 6. Plot showing a rapid ramp up and high sustained utilization of a Cloud resource through Coasters for modFTDock tasks (RNA-RNA docking)

effective use of these computational resources. However, enabling access to diverse, distant clusters does not necessarily address the ease of use and seamlessness implied by utility computing. Consequently, complex software systems were developed to make overall collections of computing resource useful, by presenting a reliable, elastic aggregate on a physical layer of varied and at times unreliable resources.

The Java Commodity Grid (CoG) Kit [31] is an API that provides a uniform interface to local and remote computational resources and file systems. It enables the development of portable Grid applications, but does not attempt to create a more reliable view of the underlying resources. A similar system, which is also an OGF standard, is SAGA [12].

Process pipelines are described and executed using "workflow systems". These systems typically aggregate multiple execution resources and allow jobs to be dynamically scheduled on them. Some workflow systems implement automated fault tolerance mechanisms (e.g., Swift [32]), some rely on the underlying infrastructure for fault tolerance (e.g., Chimera [8], Pegasus [7], DAGMan [28]), and others provide diverse user-specifieable strategies for dealing with faults (e.g., Karajan [4]).

Pilot jobs are a well-established mechanism for distributing computation to a set of remote compute sites. In this model, initial *pilot jobs* are distributed to the compute sites. Each of these acts a compute service, and is available to perform work for a centralized scheduler. Pilot job systems were initially constructed for performance reasons; work may be sent to a pilot job over a network much faster than through the traditional system scheduler.

One of the first implementations of the Pilot Job idea is the Condor [27] Glidein [10]. A Glidein is a daemon process that can automatically download Condor binaries to a compute node and advertise the node as a resource in a Condor pool. Glideins can be configured to automatically shut down after a certain idle time in order to avoid wasting resources.

The DIRAC system [30] was developed with the goal of handling large analysis tasks on data from the Large Hadron Collider. DIRAC uses the XMPP messaging protocol and has the ability to cache its various components when deployed on computing resources. It supports different mechanisms for data transfer reliability.

The Falkon [22] system shows that impressive performance can be achieved with a pilot job system in comparison to traditional execution mechanisms.

SAGA BigJob [18] is a Pilot Job system implemented as a SAGA [12] adaptor and on top of SAGA adaptors, not unlike Coasters and CoG providers.

8. Conclusion and Future Work

Pilot Job systems can dramatically improve the performance of process pipelines with small job run times. The Coaster System is one such implementation. We believe that it goes further than similar systems by providing a zero-install solution, advanced node management and I/O management, while achieving non-trivial performance and scalability.

However, considerable work remains to be done. The ability to easily run the Coaster Service on compute nodes instead of cluster head nodes is necessary in order to avoid overloading the cluster head nodes (and the subsequent angry emails from the resource providers). The I/O subsystem could employ caching to improve the performance of broadcast operations. Buffer copies when forwarding I/O data in proxy mode should also be reduced. And last, but not least, better diagnostic mechanisms should be provided for those times when things simply do not go according to plan.

References

[1] P. H. Beckman. Building the TeraGrid. *Philosophical Transactions of the Royal Society*, 363(1833), 2005.

[2] Bionimbus web site. http://www.bionimbus.org.

[3] Cobalt web site. http://trac.mcs.anl.gov/projects/cobalt.

[4] CoG workflow guide. http://wiki.cogkit.org/wiki/Java_CoG_Kit_Workflow_Guide.

[5] Cray XE web site. http://www.cray.com/Products/XE/CrayXE6System.aspx.

[6] K. Czajkowski, I. Foster, N. Karonis, C. Kesselman, S. Martin, W. Smith, and S. Tuecke. A resource management architecture for metacomputing systems. *Lecture Notes in Computer Science*, 1459, 1998.

[7] E. Deelman, G. Singh, M.-H. Su, J. Blythe, Y. Gila, C. Kesselman, G. Mehta, K. Vahi, G. B. Berriman, J. Good, A. Laity, J. C. Jacob, and D. S. Katz. Pegasus: A framework for mapping complex scientific workflows onto distributed systems. *Scientific Programming*, 13, 2005.

[8] I. Foster, J. Voeckler, M. Wilde, and Y. Zhao. Chimera: A virtual data system for representing, querying, and automating data derivation. In *Proc. Scientific and Statistical Database Management*, 2002.

[9] I. T. Foster. Globus Toolkit Version 4: Software for Service-Oriented Systems. *Journal of Computer Science and Technology*, 21(4):513–520, July 2006.

[10] J. Frey, T. Tannenbaum, M. Livny, I. Foster, and S. Tuecke. Condor-g: A computation management agent for multi-institutional grids. *Cluster Computing*, 5:237–246, July 2002.

[11] R. Gibbons. A historical application profiler for use by parallel schedulers. In *In Job Scheduling Strategies for Parallel Processing*, pages 58–77. Springer Verlag, 1997.

[12] T. Goodale, S. Jha, H. Kaiser, T. Kielmann, P. Kleijer, G. V. Laszewski, C. Lee, A. Merzky, H. Rajic, and J. Shalf. SAGA: A Simple API for Grid Applications. High-level application programming on the Grid. In *Computational Methods in Science and Technology*, page 2006.

[13] R. L. Henderson and D. Tweten. Portable batch system: Requirement specification. Technical report, NAS Systems Division, NASA Ames Research Center, 1998.

[14] S. hui Chiang, A. Arpaci-dusseau, and M. K. Vernon. The impact of more accurate requested runtimes on production job scheduling performance. In *In Job Scheduling Strategies for Parallel Processing*, pages 103–127. Springer Verlag, 2002.

[15] IBM Blue Gene team. Overview of the IBM Blue Gene/P project. *IBM J. Research and Development*, 52(1/2), 2008.

[16] A. Inc. *Amazon Elastic Compute Cloud (Amazon EC2)*. Amazon Inc., http://aws.amazon.com/ec2/#pricing, 2008.

[17] M. Litzkow, M. Livny, and M. Mutka. Condor - A hunter of idle workstations. In *Proc. International Conference of Distributed Computing Systems*, 1988.

[18] A. Luckow, L. Lacinski, and S. Jha. SAGA BigJob: An extensible and interoperable pilot-job abstraction for distributed applications and systems. In *Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, CCGRID '10, pages 135–144, Washington, DC, USA, 2010. IEEE Computer Society.

[19] A. W. Mu'alem and D. G. Feitelson. Utilization, predictability, workloads, and user runtime estimates in scheduling the ibm sp2 with backfilling. *IEEE Trans. Parallel Distrib. Syst.*, 12:529–543, June 2001.

[20] NSF. Award abstract #091081 - FutureGrid: An experimental, high-performance grid test-bed, 2009.

[21] OpenSSH web site. http://www.openssh.com.

[22] I. Raicu, Y. Zhao, C. Dumitrescu, I. Foster, and M. Wilde. Falkon: A Fast and Light-weight tasK executiON framework. In *Proc SC'07*, 2007.

[23] Sun Grid Engine web site. http://gridengine.sunsource.net.

[24] S. Srinivasan, R. Kettimuthu, V. Subramani, and P. Sadayappan. Selective reservation strategies for backfill job scheduling. In D. Feitelson, L. Rudolph, and U. Schwiegelshohn, editors, *Job Scheduling Strategies for Parallel Processing*, volume 2537 of *Lecture Notes in Computer Science*, pages 55–71. Springer Berlin / Heidelberg, 2002.

[25] R. M. Stallman. What does that server really serve? *Boston Review*, 2010.

[26] Sun Constellation system. http://en.wikipedia.org/wiki/Sun_Constellation_System.

[27] T. Tannenbaum, D. Wright, K. Miller, and M. Livny. Condor – a distributed job scheduler. In T. Sterling, editor, *Beowulf Cluster Computing with Linux*. MIT Press, October 2001.

[28] D. Thain, T. Tannenbaum, and M. Livny. Distributed computing in practice: The Condor experience. *Concurrency and Computation: Practice and Experience*, 17(2-4), 2005.

[29] D. Tsafrir, Y. Etsion, and D. G. Feitelson. Modeling user runtime estimates. In *In 11th Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP 2005*, pages 1–35. Springer-Verlag, 2005.

[30] A. Tsaregorodtsev, V. Garonne, and I. Stokes-Rees. DIRAC: A scalable lightweight architecture for high throughput computing. In *Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing*, GRID '04, pages 19–25, Washington, DC, USA, 2004. IEEE Computer Society.

[31] G. von Laszewski, J. Gawor, P. Lane, N. Rehn, M. Russell, and K. Jackson. Features of the Java Commodity Grid Kit. *Concurrency and Computation: Practice and Experience*, 14(13-15), 2002.

[32] M. Wilde, M. Hategan, J. M. Wozniak, B. Clifford, D. S. Katz, and I. Foster. Swift: A language for distributed parallel scripting. *Parallel Computing*, In Press, published online, 2011.

[33] D. Zotkin and P. J. Keleher. Job-length estimation and performance in backfilling schedulers. In *Proceedings of the 8th IEEE International Symposium on High Performance Distributed Computing*, HPDC '99, pages 39–, Washington, DC, USA, 1999. IEEE Computer Society.