My Cray can do that? Supporting Diverse Workloads on the Cray XE-6

Richard Shane Canon, Lavanya Ramakrishnan, and Jay Srinivasan
NERSC
Lawrence Berkeley National Laboratory
Berkeley, CA USA
SCanon,LRamakrishnan,JSrinivasan@lbl.gov

***Abstract*—The Cray XE architecture has been optimized to support tightly coupled MPI applications, but there is an increasing need to run more diverse workloads in the scientific and technical computing domains. These needs are being driven by trends such as the increasing need to process "Big Data". In the scientific arena, this is exemplified by the need to analyze data from instruments ranging from sequencers, telescopes, and X-ray light sources. These workloads are typically throughput oriented and often involve complex task dependencies. Can platforms like the Cray XE line play a role here? In this paper, we will describe tools we have developed to support high-throughput workloads and data intensive applications on NERSC's Hopper system. These tools include a custom task farmer framework, tools to create virtual private clusters on the Cray, and using Cray's Cluster Compatibility Mode (CCM) to support more diverse workloads. In addition, we will describe our experience with running Hadoop, a popular open-source implementation of MapReduce, on Cray systems. We will present our experiences with this work including successes and challenges. Finally, we will discuss future directions and how the Cray platforms could be further enhanced to support these class of workloads.**

***Index Terms*—data intensive; hadoop; workflow**

I. Introduction

Increasingly, high-end and large-scale computing are being applied to new fields of study. Most noticeably is the growth in the demands around "Big Data" where large-scale resources are used to process petabyte scale datasets and perform advanced analytics. Modern HPC systems like the Cray XE-6 could potentially play a role in addressing these emerging needs, but several design characteristics stand in the way. Fortunately, the platform's use of commodity based processors and Linux underpinnings can be exploited to open the door to these non-HPC workloads. Furthermore, Cray's recent investments to support dynamic shared libraries and its Cluster Compatibility suite further simplify the system. In this paper, we will discuss how NERSC has exploited these capabilities to support more diverse workloads on its Hopper XE-6 system. This discussion will include how this was accomplished and the impact it is having on certain applications areas.

II. Background

A growing number of fields are requiring increasing amounts of computation to keep pace with data and new classes of modeling and simulation workloads. For example, the genomics field has experienced a rapid growth in sequence data over the past 8 years. This has been driven by the rapid drop in sequencing cost due to Next Generation Sequencers. Fig 1 illustrates this growth, where it shows that the cost of sequencing a genome has dropped by a factor of over 10,000 in less than a decade. By comparison, a comparable plot of Moore's law has provided around 100x improvement over the same period. This growth in sequence data leads to a comparable growth in demand for computing. The results from the analysis of genomic data can aid in the identification of microbes useful for the purposes of bioremediation, biofuels, and other applications. Another example comes from the Materials Project [1]. The Materials Project is an effort to model thousands of inorganic compounds to compute basic material properties. These results are then used to identify promising materials for applications such as next generation batteries. This requires throughput oriented scheduling and future efforts may require hundreds of millions of core hours of simulation. Increasingly, these communities are looking to use HPC Centers to satisfy their resource demands.

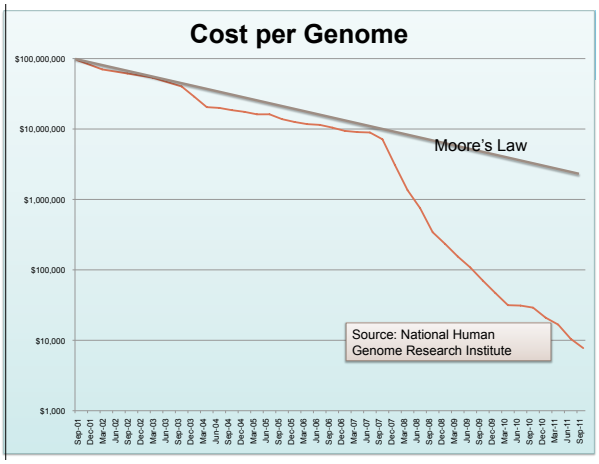


Fig. 1. Plot of declining cost of sequencing a human genome compared with the relative cost of computing. Figure is courtesy of NHGRI.

A platform like NERSC's Hopper system provides an unprecedented level of capability to a broad set of researchers. However, the XE-6 architecture and its run-time environment have been optimized for tightly-coupled applications typically

written in MPI. Furthermore, the scheduling policies in place at many HPC centers like NERSC have traditionally been designed to favor jobs that concurrently utilize a large number of processors. Unfortunately, these design points and policies can act as a barrier to a growing set of users that can take advantage of the computational power but have more throughput oriented workloads. In the past, HPC centers have often directed this class of users to other resources or facilities arguing that the large HPC systems were to specialized and valuable to use for throughput oriented workloads. However, recent analysis at NERSC shows that HPC systems like Hopper can be cost effective compared to many department scale clusters [2]. This is a consequence of the size of the system which allows for a greater level of consolidation and increased economies of scale. Furthermore, there is an increasing need to support a variety of scales and workload patterns in order to ensure that researchers can productively accomplish their science. Recognizing these points, NERSC has engaged in a variety of efforts to enable these new class of users to leverage the computational power of systems like Hopper.

There are several issues that often complicate using a system like Hopper for throughput workloads. Some are policy based. For example, NERSC typically limits the number of jobs per user in an "eligible" stage for scheduling. The justification for this is to guard against a single user occupying the system for extended period of time which could prevent other users from getting jobs processed in a reasonable time. Other barriers are due to the run-time system. For example, on the Cray platform one must use the ALPS runtime (i.e. aprun) to instantiate a process out on a compute node. While multiple aprun commands can be executed in parallel from a single job script, relying on this approach leads to the undesirable consequence that users would require thousands of aprun commands to run a typical throughput oriented workload. Recognizing these challenges, NERSC has explored a number of approaches to either working around these limitations or changing the policies and run-time environment.

III. Approaches

There are several potential approaches to addressing the policy and run-time constraints that stand in the way of running throughput workloads. One method is to provide an alternate framework that runs within the constraints of the policies and run-time system but provides an efficient mechanism to execute throughput workloads. We provided three examples of this approach including a Task Farmer, Hadoop on demand, and a personal scheduler instance. Another approach is to remove the policy and run-time limitations entirely. We describe an approach that leverages Cray's recently released Cluster Compatibility Mode (CCM) to provide a throughput oriented scheduling environment. We will explain the differences between these different approaches including some of the limitations and constraints. We will also provide an explanation on the trade-offs of different approaches for different workloads.

A. Task Farmer

As noted in the introduction, the field of Genomics is facing an unprecedented increase in data generation. This trend led the Joint Genome Institute (JGI), DOE's production sequencing facility, to turn to NERSC to help address its increasing computing needs. In addition to JGI's dedicated resources, JGI users wanted to exploit the capabilities of systems like Hopper to process and analyze genomic data. This led NERSC to develop a framework that would enable JGI scientists to easily exploit the capabilities of the HPC systems. The result of this work was a Task Farmer optimized for use cases common in genomics, but which is applicable across a broad set of throughput oriented applications.

1) Approach: Many high throughput workloads require running a common set of analysis steps across a large input data set in parallel. Often times, these tasks can be run completely independent of each other (i.e. embarrassingly parallel). Typically one would use a serial queue on a throughput oriented cluster to satisfy this workload. However, on an HPC system it is desirable to restructure this as a single large parallel job that uses an internal mechanism to launch the tasks on the allocated processors. This is exactly the approach the TaskFarmer takes. The framework consists of a single server and multiple clients running on each of the compute elements. The server handles coordinating work, tracking completion, sending the input data to be processed, and serializing the output. The client, which is run on the compute nodes, handles requesting work, launching the serial application, and sending any output back to the server before requesting new work. The framework is primarily written in Perl. While this is arguable not the most efficient language to use, it was the most familiar to the developers and users which mades it a convenient choice.

The TaskFarmer framework provides several features that simplify running throughput oriented workloads. Here we summarize a few.

- Fault Tolerance: Clients send a heartbeat to the server. If the server fails to receive a heartbeat within an adjustable time period, then it will automatically reschedule the task. The framework will also automatically retry tasks up to a threshold. This can help work around transient failures.
- Checkpointing: The server keeps track of which tasks have been completed successfully. A compact recovery file is maintained in a consistent state with the output files. The recovery process is particularly useful when the required wall time for a job is poorly understood.
- Serialized Output: The clients send the output from each tasks back to the server. While this can create a scaling bottleneck, in practice this works reasonably well for many workloads. This ensures that the output is consistent and prevents potentially overlapping writes within a single file. It also helps avoid creating thousands or millions of individual output files if each task was to write to a dedicated file.
- Sharding the Input: The server understands the FASTA

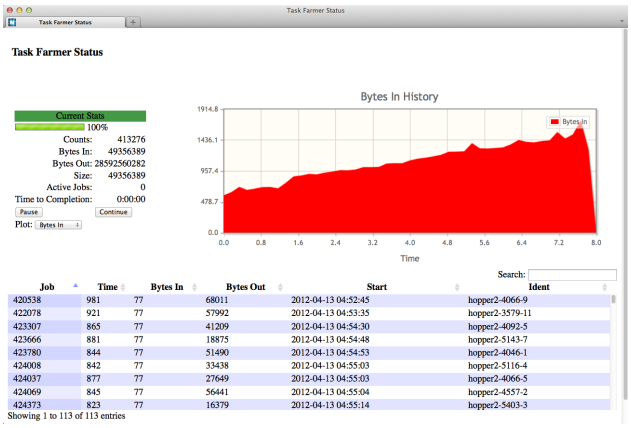


Fig. 2. Example of web based status page for the TaskFarmer.

format which is commonly used in genomics and it can automatically shard the input sequences up into small chunks. This eliminates the need to pre-shard the input and create thousands or millions of input files for processing.

- Progress Monitoring: The TaskFarmer can update a JSON formatted status file. This file can be downloaded from a web server and visualized by a monitoring page (See Fig. 2).

The TaskFarmer greatly simplifies many common tasks in genomic processing. For example, the following command would execute BLAST across all of the allocated processors.

```
tfrun -i input.faa blastall -p blastp \
  -o blast.out -m8 -d $SCRATCH/reference
```

The framework takes care of splitting up $input.faa$ into small chunks (32 sequences by default) and merges the output into a single output file ($blast.out$). If the parallel job requests were to run out of time, the job can be resubmitted and the Task Farmer will automatically resume from where it left off and re-queue any tasks that were incomplete.

2) Scaling and Thread Optimization: The TaskFarmer uses TCP sockets for communications between the server and client. While this can create a potential scaling bottleneck, in practice we have found it to be sufficient to scale up to 32,000 cores. Connection between the server and client are created and destroyed during each exchange. This adds overhead for creating the connections but helps avoid running out of sockets on the server. Typically, when using the TaskFarmer we adjust the task granularity to achieve an average task run time of around five to ten minutes. This decreases the average number of connections to 50-100 per second for even a large 32k core run. One advantage of using TCP sockets is that the framework can be easily ported to other resources. For example, in addition to Hopper, the framework has been deployed on a cluster and, even, on virtualized cloud-based resources. The framework also supports running a single server that is used to coordinate across multiple resources.

The TaskFarmer allows the user to adjust the number of tasks per node by setting an environment variable ($THREADS$). This can be used in conjuction with a threaded applications to reduce the load on the server since this can help reduce the number of concurrent connections. In general, the user adjusts $THREADS$ and the level of threading for the underlying application to find the balance that achieves the best efficiency and the least load on the server. This can require some experimentation for a new CPU architecture and the optimal layout is highly dependent on the underlying application. For example, for some applications we have found they run best with four instances per node with each instance using six threads. While for other applications, where the threading implementation is not efficient, we simply run 24 instances per node and disable threading in the application.

3) Discussion: The TaskFarmer has proven useful for tackling a number of large scale genomic workloads. This has included large-scale comparisons of metageome datasets with standard references and building clusters using $hmmsearch$. Since the input format for the TaskFarmer is quite simple, it has also been adopted to run lists of commands, MapRduce style applications, and other task oriented workloads.

B. MyHadoop

In 2005, Jeffrey Dean and Sanjay Ghemawat at Google published a paper describing Google's approach to addressing their rapidly increasing data analysis needs [3]. A related paper was published shortly after describing the distributed file system that was used to support the distributed framework [4]. Together, MapReduce and the Google File System (GFS), enabled google to tackle internet scale data mining with a highly scalable, fault tolerant system. These publications were widely read, and soon after inspired Doug Cutting to create Hadoop, an open-source framework modeled after MapReduce and GFS. The power of this approach is that it allows a user to easily express many data-intensive problems in a simple way. The framework then handles the job of distributing the tasks across a system in a manner that is resilient against failures, efficiently allocates work close to data, and is highly-scalable.

Hadoop has been widely adopted by the Web 2.0 community to deal with some of the most demanding data intensive tasks. Petabyte scale data sets are now being analyzed with Hadoop at companies like Yahoo! and Facebook [5], [6]. Traditionally, Hadoop has been used for text analysis for tasks such as web search or mining web logs. However, the MapReduce model is starting to be applied to data-intensive scientific computing as well. Genome assembly and comparison have been ported to run inside the framework [7]. Hadoop has typically been deployed on commodity based clusters using inexpensive storage. However, as the model gains wider adoption, it will likely prove a useful approach fo analyzing data sets on large-scale systems. To help users explore this model on the Cray we have developed "MyHadoop" which enables a user to instantiate a *private* Hadoop cluster that is requested as a single parallel job through the standard Torque/Moab batch system.

1) MapReduce and the Hadoop Architecture: A Map Reduce tasks consists of three basic phases: a map phase, a

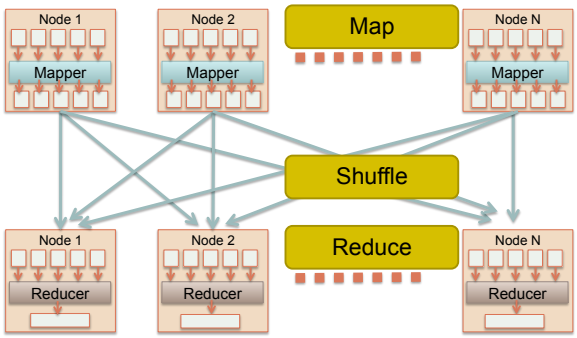


Fig. 3. Illustration of the MapReduce model including the implicit shuffle phase.

shuffle phase, and a reduce phase. The user specifies the functions to perform in the map and reduce phases, hence the name. The Map function reads in input data delivered through the framework and emits key-value pairs. The shuffle phase which is performed implicitly by the framework, handles redistributing the output to the reduce tasks and ensures that all key-values pairs that share a common key are sent to a single reducer. This means that a reducer does not have to communicate with other reducers to perform the reduction operation. For example, assume that there was a large list of numbers that needed to be binned into a histogram. The map would compute the bin for each input that it was sent and emit a key of the bin number and a value of 1. The reducer would then accumulate all of the inputs to compute the total for each bin in its partition space. The framework assist in decomposing the map phase into individual tasks. By default, it will create a map tasks for ever block of input, but this behavior can be tuned. The user must typically specify the number of reduce tasks, since the framework cannot easily predict the best number of reducers.

The Hadoop MapReduce framework consists of a single Job Tracker and many Task Trackers. The Job Tracker tracks the overall health of the system, handles job submissions, schedules and distributes work, and retries failed tasks among other things. It plays a role similar to the master server in a batch scheduler system. The task trackers communicate with the job tracker and execute tasks. The task trackers also performs the shuffle operation by directly transmitting data between task trackers. The task trackers run all phases of the job (Map, Shuffle, and Reduce). The task trackers also send heartbeats and statistics back to the job tracker. If a job tracker fails to receive a heartbeat within a certain time period, it will automatically reschedule tasks which were running on the node. The architecture is similar to the resource manager and execution daemon used in a batch system. When scheduling tasks, the job tracker will attempt to place the map tasks close to the data. It does this by querying the file system to determine the location of the data. For example, if there is an idle task tracker that contains a copy of the data, it will typically schedule the tasks on that node.

The Hadoop Distributed File System consists of a single Name Node and many Data Nodes. The Name Node maintains the name space of the file system and is similar to the metadata server in a parallel file system such as Lustre. The Data Nodes store blocks of data on locally attached disks similar to an Object Storage Server in Lustre. The file system is typically configured to replicate the data across multiple data nodes for fault-tolerance and performance. In Hadoop, the Task Tracker and the Data Node are typically running on every compute node in the cluster. The Task Tracker will normally direct its output to the local Data Node to avoid excess communication. The Data Node will then handle replicating the data to other Data Nodes to achieve the specified replication level. The Name Node provides the location of the replicas but is otherwise not involved in the replication process. While the primary purpose of the replication is to provide fault-tolerance, the extra replicas also provide increased opportunities for the job tracker to schedule tasks close to the data.

2) Implementation on Cray Systems: Hadoop has been designed to run on commodity clusters with local disks and a weak interconnect such as gigabit ethernet. Consequently, running Hadoop on a system like the Cray which lacks local storage presents some challenges. In addition, the framework is written in Java and is typically started through startup scripts that rely on ssh to launch the various services on the compute elements. Here is a brief summary of the ways in which we address these issues and others on the Cray systems.

- Utilize the Lustre scratch file system to replace HDFS: In certain cases, Hadoop requires a local working directory for the Task Tracker. In this case we create a unique directory for each node in the Lustre file system and use symlinks to direct Hadoop to the appropriate directory. We use a default stripe count of one.
- Generate a set of configuration files: The configuration is customized to use the user's scratch space for various Hadoop temporary files. In addition, the job tracker is configured to run on the "MOM" node.
- Use $CRAYROOT\_FS = DSL$: This provides a more complete run-time environment on the compute nodes for the Hadoop framework.
- Use $aprun$ to launch the task tracker on the compute nodes: The option "-N 1" is used to ensure that a single TaskTracker per node is started.
- Tell Java to use the $urandom$ random number generator instead of the default $random$: This is needed because there are insufficient inputs for entropy on the light-weight compute nodes. This is achieved by adding the following argument to the Java execution $-Djava.security.egd = file : /dev/urandom$.
- Use a custom library that is specified through $LD\_PRELOAD$ environment variable: This library intercepts calls to $getpw*$ and $getgr*$ and returns generated strings. This is to work around the fact that the compute nodes do not typically have fully configured name services. For example, at NERSC we rely on LDAP to provide name services, but this is not configured on compute nodes. The library is designed to provide responses

to the specific types of lookup queries performed during Hadoop startup.

Most of these details are handled in the MyHadoop startup scripts. It handles initializing the users environment and file layout, starting up the Job Tracker on the MOM node, and launching the Task Trackers on the compute nodes. Once the framework is started, the user can submit jobs to the private Hadoop instance in the exact same way jobs would be submitted to a dedicated Hadoop cluster.

3) Discussion: While the customizations we have developed enable Hadoop clusters to be started on systems like the Cray XE-6, the user needs to understand some of the limitations of the approach. The most significant limitation comes from the lack of local disk storage. Hadoop assumes local disk both in its distributed file system and in its MapReduce framework. A well performing Lustre file system can effectively replace an HDFS file system. However, the framework's expectation of local disk to "spill" large output to during the map phase is more difficult to address. On the Cray system, these temporary files must be written to the Lustre file system which introduces more load on the file system and impacts the overall performance. As a result, Hadoop on the Cray is best suited for applications where the map tasks is able to minimize the output sent to the reducers. A good example is a map performing a filter of input data.

C. MySGE

Many workflows require a mixture of task parallel execution coupled with serial regions that call for a more sophisticated execution framework. For example, a pipeline may require executing some analysis operation across a large input data set then reorganizing the results and performing additional analysis. These types of dependency based scheduling call for a more flexible solution. Batch systems such as Torque and Univa's GridEngine already provide these capabilities. However, systems like Hopper are typically configured to favor large parallel jobs and, sometimes, take steps to penalize through-put workloads. In addition, aspects of the Cray run-time system (ALPS) also complicate running throughput oriented workloads on the system. In order to address this combination of policy and architectural constraints, we developed MySGE. This tool allows individual users to instantiate "virtual private clusters" (VPC) running a private GridEngine scheduler. Users can then submit a broad range of workloads to this private cluster including array jobs or jobs with complex dependencies. From the standpoint of the global scheduler, the VPC looks like a standard parallel job.

1) Architecture: The GridEngine scheduler (both Sun GridEngine and Univa GridEngine) are a popular choice for throughput oriented workloads. GridEngine is particularly popular among the genomics and experimental High-Energy Physics communities. While the scheduler can support scheduling parallel workloads, its strength is in its ability to schedule a large number of tasks. This was one of the primary reasons we chose GridEngine as the private scheduler for the Virtual Private Cluster approach. Additionally, GridEngine can be easily run as a non-root user and supports using SSL based certificates for authentication. Using a similar approach with Torque requires custom patches since it assumes that it is running as root. The services used in a GridEngine cluster are similar to those found in other batch systems. There is a master scheduler service that receives jobs requests and schedules the jobs on resources. There is a global master that monitors the resources and communicates with the scheduler so that it can select the appropriate resources. Finally on each compute node, there is an execution daemon that communicates with the master services, executes the actual jobs, and monitors the node's health.

2) Implementation on Cray Systems: MySGE is essentially a collection of scripts that initializes the GridEngine environment for a user and assist in starting the various services. It uses unaltered version of GridEngine which means existing job scripts and tools can be used with it. Many of the challenges described in Section III-B2 apply to MySGE as well. Here we summarize the basic approach to launching GridEngine on the Cray system.

- Prior to starting a MySGE cluster, the user runs an initialization script. This script creates the directory layout for GridEngine, creates SSL certificates, and performs other basic setup tasks.
- Once the system is initialized, the user executes vpc_start to instantiate the VPC. The user can include standard torque qsub options to specify the desired wall time, queue, number of nodes, etc to the global scheduler. MySGE performs the actual submission to the global scheduler and, once scheduled, handles starting up the services.
- Upon startup MySGE ensures that a qmaster is running. It then collects the hostnames (actually the Network ID on the Cray systems) for the allocated nodes. While this information could probably be gathered from a special ALPS command, we simply use $aprun$ to execute a hostname script on the assigned compute nodes. These hostnames are added to the GridEngine configuration as execution hosts.
- Use $CRAYROOT\_FS = DSL$ to provide a more complete run-time environment on the compute nodes for the Hadoop framework.
- A custom library is preloaded in order to intercept calls to $getpw*$ and $getgr*$. This is to work around the lack of configured name services on the compute nodes. The library reads various environment variables to generate a response. CCM could be used to address this issue.
- MySGE uses $aprun$ to start the execution daemons on the compute nodes.
- The user sources a setup file to configure their environment to submit jobs to the VPC. This can be done from any of the interactive nodes that are part of the Cray system.

Once the VPC has been instiated, the user can submit jobs to the VPC in the same manner they would a standard

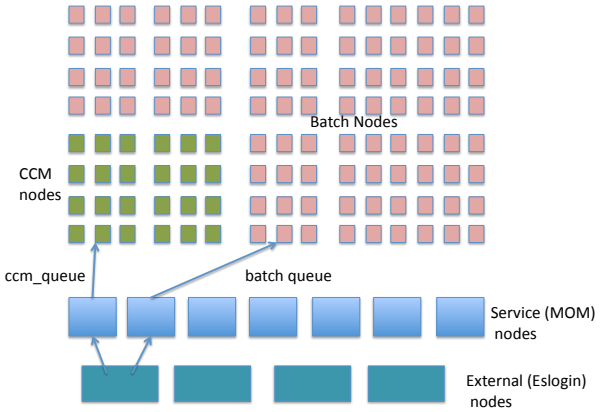


Fig. 4. Schematic of access using CCM and MPP modes

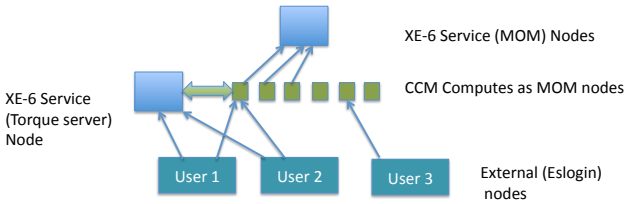


Fig. 5. Schematic of jobs dispatched to CCM compute nodes running PBS MOM

GridEngine cluster.

3) Discussion: A MySGE Virtual Private Cluster provides a great deal of flexibility, both in the types of jobs that can be submitted to the VPC and in the configuration of the VPC. For example, the user can submit large array jobs that run very quickly and can include dependencies. Since the user has dedicated access to the allocated nodes, the user doesn't have to wait in the global scheduler for each tasks to become eligible for scheduling. If a slot in the allocated pool of nodes is available, the job will run immediately. Furthermore, since all of these services run as the end user, versus as special privileged account, the user has total flexibility in the configuration of the VPC scheduler. For example, the user can customize the queue structure to provide different priority queues. This allows the user to tailor the configuration for their workload without additional assistance from a system administrator. However, this does require the user to have a more in-depth understanding of the batch system.

D. Using CCM

The Cray XE-6 system is able to function in two modes. In the first, the MPP mode, the compute cores of the system are tightly coupled with the high-speed interconnect and the compute nodes of the system run a version of the Cray Linux Environment which does not support all the services that one might expect in a typical cluster environment. Additionally, in this mode, a node is not shared between users, since the Gemini network interface cannot be shared between different users on the same node. However, there are a large number of ISV applications that depend on the availability of the standard Linux "Cluster" environment, and users typically have little to no control over these requirements. To accomodate these users, Cray introduced the "Cluster Compatibility Mode" (or CCM v2.2.0-1) in CLE v4.0-UP02. CCM builds on the facility that provides dynamic shared libraries to the compute nodes (the Cray compute node root runtime environment, CNRTE) and essentially transforms the individual compute nodes of the Cray system into "cluster-like" nodes which have a Linux environment that will be familiar to users of Linux clusters.

One benefit of running a node under the CCM environment is that we can then run applications and services on the node that normally might not be available on the node. An obvious use-case that Cray supports is that of ISV applications. Earlier versions of CCM were restricted to utilizing the high-speed (Gemini network) not natively, but instead over TCP/IP. The performance implications of this for MPI-based ISV applications are obvious, so Cray now supports a feature called ISV Application Acceleration (IAA) which allows for direct use of the Gemini network by providing a IB-verbs to Gemini layer that MPI libraries compiled using IB-verbs can utilize.

Another use case is to essentially transform compute nodes into stand-alone compute elements that can then be scheduled independently. If one was to use a batch system such as Torque, the nodes would then be "Mom nodes" (in the vernacular of Torque). This can be done by running a CCM job (as a regular user) that, upon startup, then starts a PBS mom daemon on each of the compute nodes assigned to the job. These MOM processes then communicate with a previously running Torque server (which is running on alternate ports, so as to not conflict with the regular batch system).

To implement this, we have enabled user jobs to start up PBS Mom processes on compute nodes. This is carefully controlled so only a designated user, and one who is assigned to the node by ALPS is able to start up MOM processes. Once the MOM processes are up and running and registered with the Torque server, other users can then submit jobs to the batch system. In order to facilitate easy access to the second batch system, we have provided a module that users can load to give them access to commands used to query, submit jobs, and manage their workflow through the second batch system.

The designated "master CCM job" requests as many nodes as is required to support a serial workload on the system for as long as necessary. There are currently some limitations to how many nodes can be requested using a single CCM job. Once the CCM nodes are allocated, the user then starts up a MOM on every single CCM allocated node (either using ccmrun or ccmlogin). At this point, all the CCM nodes are registered with the secondary Torque server and are available to run jobs. Users can then query this secondary batch system and submit jobs to it, just as if they were submitting jobs to a regular cluster.

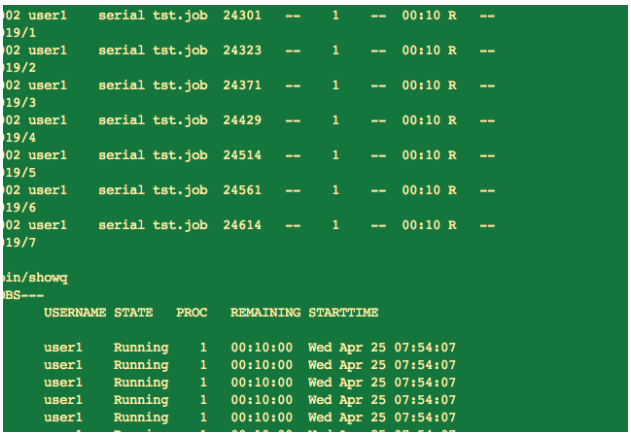


Fig. 6. Snippets of showq and qstat output showing multiple jobs on a compute node

IV. Discussion

The four approaches described above have various advantages and disadvantages. Here we will discuss some of the trade-offs. It is worth noting that all of these approaches can coexist on a single system. Users can spawn MyHadoop instances, while another user (or even the same user), starts a MySGE Virtual Private Cluster. Since most of these approaches run as the end user, they behave basically the same as a standard MPI parallel job from the standpoint of the global scheduler.

1) Private versus Shared Resources: One major difference between the two approaches is private versus shared allocation of resources. The first three approaches (Task Farmer, MyHadoop, and MySGE) rely on a parallel job request to the global scheduler. This allows these environments to be instantiated without policy changes. For example, this can effectively work around limits on the number of submitted or eligible jobs. For some institutions, the scheduling policies may be driven by external metrics or trying to balance competing priorities. By operating within these constraints, this approach allows the end user to get their work done without requiring policy changes. However, there is a downside to this approach. Since the nodes are reserved for the end user for the duration of the parallel job request, other jobs can not be scheduled on idle nodes by the global scheduler. If the user can keep the allocated resources busy, this is not an issue. However, if the throughput workload is highly variable, this can lead to inefficient use of resources.

Example Use Case. To help illustrate this limitation, we will describe a use case. User Bob has a task oriented workload that requires roughly 1200 core hours of processing. The workload consists of 12,000 tasks, so on average each task takes about 6 minutes. He decides to use MySGE and requests 1200 cores for 2 hours to provide a buffer. Unfortunately, Bob's tasks are highly variable. Some tasks take only a minute while a few take an hour. Bob submits this workload to his VPC. After an hour most of the tasks are complete. However, a handful of tasks continue to run and his final wall time is 1.5 hours. During this long-tail period, Bob had over 1000 cores idle. Since these cores are reserved to Bob as a part of his parallel job request to the global scheduler, other jobs can not be scheduled to these idle cores by the global scheduler so they remain idle. Furthermore, Bob is charged for those resources even if the nodes are idle. So Bob is charged 1800 core hours versus the estimated 1200 core hours. If Bob's workload was perfectly load balanced, then each tasks would run for 6 minutes and the workload would complete in the expected hour wall time. In contrast, if Bob's workload were submitted to a CCM/Torque cluster, then other throughput jobs can be run on the idle resources as Bob's tasks complete. Furthermore, Bob will only be charged for the hours consumed (1200 core hours).

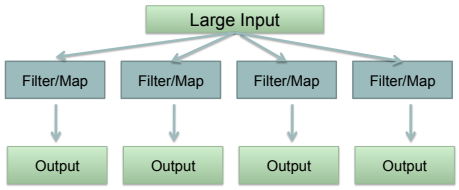
TABLE I
Summary of comparison between different approaches.

Approach	Private/Shared	Flexibility	Fault Tolerance
Task Farmer	Private	Low	High
MyHadoop	Private	Medium	High
MySGE	Private	High	Low
CCM/Torque	Shared	High	Low

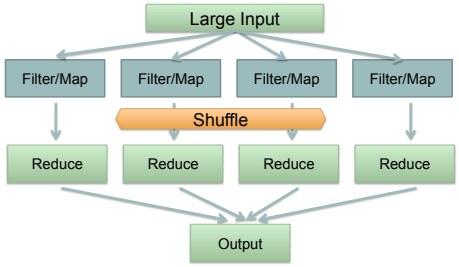
2) Flexibility: Throughput oriented workloads vary greatly in complexity. In some cases, the user simply needs to run a common function across a large set of inputs. For example, filtering a dataset to remove data points that are above some threshold. In other cases, the user may have a combination of filters followed by some global reduction operation. An example of this would be binning data points and tallying the number of entries in each bin. The most complex workflows may have many levels of dependencies with highly parallel regions at different stages. We illustrate these different workflows in Fig 7. A fully featured scheduling system like GridEngine and Torque will provide the greatest flexibility. So MySGE and CCM/Torque are typically the best option for complex workflows. Using additional tools such as Yahoo!'s Oozie [8] can enable Hadoop to run more complex workflows, but this requires additional understanding additional tools. Finally, the Task Farmer is tailored to running an array of independent tasks and lacks a mechanism to specify dependencies between tasks. So, it is most appropriate when the workflow is a simple job array. We summarize these features in Table I.

V. Future Work

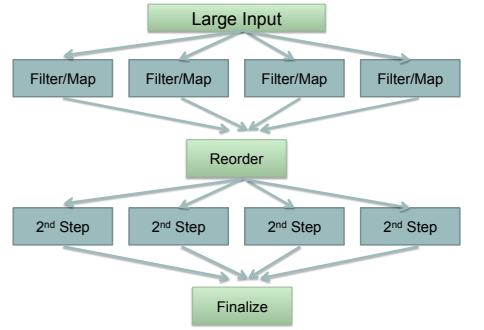
NERSC continues to explore ways to extend these approaches and is particularly focused on ways to improve the Torque/CCM approach. For example, the initial implementation uses a static set of resources. One enhancement would be to dynamically add and remove resources to the throughput partition based on demand. Ideally this would utilize reservations in the throughput scheduler to ensure that user jobs are not scheduled onto resources that may be released back to the global parallel scheduler during the duration of the job. Eventually the serial queue could be integrated into the standard global scheduler. Users would simply submit jobs



(a) Array Job



(b) Map Reduce Job



(c) Complex Job

Fig. 7. Illustrations of common workflow patterns.

to the serial queue, no differently than they would submit a large parallel job. A combination of the Moab scheduler, CCM, and custom startup scripts could be used to dynamically move nodes back and forth between the parallel mode and the throughput mode.

VI. Conclusion

The rapid increase in data-intensive computing and high throughput computing has led to a dramatic increase in computing requirements for these classes of workloads. These increases are being driven by dramatic improvements in scientific instruments and detectors and by an increasing need to execute large scale ensemble runs for the purposes of uncertainty quantification and exploring a large parameter space. These class of problems have computational requirements that rival some large scale simulation and modeling problems and these communities are increasingly looking to use HPC centers like NERSC to meet these needs. NERSC has developed and explored a number of approaches to accommodating these types of workloads. We expect that over time the needs of the data intensive computing communities and those of the traditional modeling and simulation communities will slowly converge and that centers like NERSC will increasingly need to effectively support both types of communities.

Acknowledgment

This work was supported by the Director, Office of Science, Office of Advanced Scientific Computing Research of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

References

[1] "Materials project," http://materialsproject.org/.
[2] "Magellan final report," http://science.energy.gov/~/media/ascr/pdf/program-documents/docs/Magellan_Final_Report.pdf.
[3] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.
[4] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The google file system," in *Proceedings of the nineteenth ACM symposium on Operating systems principles*, ser. SOSP '03. New York, NY, USA: ACM, 2003, pp. 29–43. [Online]. Available: http://doi.acm.org/10.1145/945445.945450
[5] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," in *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, May 2010, pp. 1 –10.
[6] D. Borthakur, J. Gray, J. S. Sarma, K. Muthukkaruppan, N. Spiegelberg, H. Kuang, K. Ranganathan, D. Molkov, A. Menon, S. Rash, R. Schmidt, and A. Aiyer, "Apache hadoop goes realtime at facebook," in *Proceedings of the 2011 international conference on Management of data*, ser. SIGMOD '11. New York, NY, USA: ACM, 2011, pp. 1071–1080. [Online]. Available: http://doi.acm.org/10.1145/1989323.1989438
[7] M. C. Schatz, "CloudBurst: highly sensitive read mapping with MapReduce," *Bioinformatics*, pp. 1363–1369, June 2009.
[8] "Oozie," http://rvs.github.com/oozie/design.html.