Experiences running different work load managers across Cray Platforms

Haripriya Ayyalasomayajula
Cray Inc.
901 Fifth Avenue, Suite 1000
Seattle, WA 98164
hayyalasom@cray.com

Karlon West
Cray Inc.
6011 W, Courtyard Dr #200
Austin, TX 78730
karlon@cray.com

***Abstract*—Workload management is a challenging problem, both in analytics and in High Performance Computing. The desire is to have efficient platform utilization while still meeting scalability and scheduling requirements. Slurm and Moab/-Torque are two commonly used workload managers that serve both resource allocation and scheduling requirements on the Cray® XC™ and Cray® CS™series supercomputers. Analytics applications interact with a different set of workload managers such as YARN or more recently, Apache Mesos, which is the main resource manager for Cray® Urika-GX™. In this paper, we describe our experiences using different workload managers across Cray platforms (analytics and HPC). We describe the characteristics and functioning of each of the workload managers. We will compare the different workload managers and specifically discuss the pros and cons of the HPC schedulers vs. Mesos, and run a sample workflow on each of the Cray platforms and illustrate resource allocation and job scheduling.**

***Keywords*-HPC, Analytics, Slurm, Moab/Torque, YARN, Apache Mesos, Urika-GX, Cray CS and XC**

I. Introduction

Workload management is a challenging problem both in analytics and High Performance Computing (HPC). Workload managers help to launch jobs on the underlying computing resources by providing resource management and scheduling. The desire is to achieve efficient resource utilization while still meeting the scalability and scheduling requirements. Traditional HPC applications interact with workload managers such as Slurm [1] or Moab [2]/Torque [3] for resource management and scheduling. Analytics applications typically interact with a different set of modern workload managers such as Apache Hadoop YARN [4] or Apache Mesos [5] [6]. Several key differences exist in the nature of the applications (HPC and analytics) as well as the workload managers they interact with to launch jobs.

At CUG2016 [7], we described our initial experiences of running mixed workloads using Apache Mesos. As a follow up, in this paper, we describe our experiences using different workload managers across Cray platforms (analytics and HPC). The goal of this paper is to highlight the differences between the workload managers and discuss the pros and cons of each of them.

By presenting a comparison between these workload managers, we create a stage for gathering requirements for future Cray architectures where we want to run both analytics and HPC workloads on the same platform and while maintaining the best of both worlds- HPC and analytics.

On CS and XC series supercomputers, we use Slurm or Moab/Torque in an attempt to balance the goals of efficient resource utilization, high scalability, and meeting scheduling requirements. Apache Mesos is the main resource manager for Urika-GX . Modern analytics workload managers such as Apache Mesos provide great flexibility and extensibility. The rest of the paper is organized as follows: In section 2, we provide a description of each of the three workload managers: Slurm, Moab/Torque, Apache Mesos. In section 3, we describe our sample workflows and illustrate how these are run on each of our machines. In section 4, we will compare the different workload managers and specifically discuss the pros and cons of the HPC schedulers vs Mesos. Finally, we present our conclusion.

II. Description of Workload Managers

Resource management and scheduling are two separate but related problems. By resource management we refer to the negotiation of managed resources that are required for running a job. Once the resources eligible to satisfy a job are identified, scheduling is performed by allowing those idle resources available to run a job to be reserved and that job is launched. The jobs whose resources are not available are enqueued (depending on the policy of the workload manager), waiting for the required resources to be freed up and scheduled on the compute resources guided by a certain policy specific to the workload manager.

Workload managers for HPC style applications traditionally handle both resource allocation and scheduling. Modern analytics resource managers such as Apache Mesos operate differently. Mesos handles resource allocation while delegating the scheduling decisions to the frameworks that are registered with it. On our analytics platforms we have different frameworks which interact with the resource manager to get the resources. By framework, we refer to a programming paradigm. Users write programs targeting a specific framework. For example, Spark [8] and Hadoop [9] are two different frameworks on Urika-GX. The programs that users write when submitted to these frameworks are referred to as jobs. Spark and Hadoop jobs can communicate with each other using defined API's but the jobs are launched within the different frameworks. In the HPC world, jobs are submitted to the workload manager queue. It is up to the workload manager to handle both resource allocation and scheduling of the job. By submitting a job, a user requests for specific resources required to run the job. The workload manager then grants the resources for the job and then schedules the job on those resources.

Slurm and Moab/Torque are free, open source workload managers for running HPC applications on, among others, Cray Platforms. To launch HPC style applications, users usually request a fixed allocation of physical nodes from Slurm or Moab/Torque and then submit the jobs. The jobs are queued and the sub-tasks of these jobs are launched by Slurm or Moab/Torque on the nodes. These workload managers also facilitate configuring communication among sub-tasks, which is particularly useful for our HPC style applications. Also, in most of the HPC applications, fault tolerance is handled by the users in their programs.

A. Slurm

Slurm is a workload manager often used to support HPC applications on Cray platforms. It provides both resource allocation and scheduling for HPC applications. From an HPC application perspective, Slurm is important primarily because it knows how to configure communication among the sub-tasks.

There is a central manager called **"slurmctld"** which monitors resources and execution of tasks. On each compute node, a **"slurmd"** daemon is running. This is responsible for running the tasks of a job on the corresponding node. The users grab an allocation of resources using the **salloc** command. The jobs are then submitted using: **srun**. We can see the status of all the jobs using the **sinfo** command.

B. Moab/Torque

Moab is a workload manager which provides scheduling and facilitates management tasks. It offers job orchestration and facilitiates enforcing site policies through its service level agreement (SLA) features. It provides system diagnostics and other essential statistics which help in understanding the cluster utilization information. Moab supports batch and interactive workloads and allows you to submit jobs that should be run at specific times as well as long running applications. It maintains complete accounting and auditing records. It has been designed for providing higher resource utilization. The Moab scheduler facilitates fair utilization of compute resources by determining the details of job launching specific to the cluster. It serves major scheduling requirements by tracking the resources and providing the resources to a job when requested. By doing this, it ensures new jobs with resource requirements wait until the resources are available.

Torque is the open source resource manager which integrates with Moab.

C. Apache Mesos

Apache Mesos is the main resource manager for Urika-GX. It is a distributed cluster and resource manager, which uses two level scheduling. Mesos takes care of resource allocation, keeping its core simple. It delegates the scheduling decisions to the frameworks registered with it, thereby allowing each framework to address its own scheduling needs.

Mesos offers rich Application Programming Interfaces (APIs) through which users can write new frameworks and port them easily on the same platform alongside the existing applications. Frameworks take charge of negotiating resources with Mesos and scheduling the jobs, fault tolerance etc. Users can focus on developing rich applications and not worry about resource allocation and other low level details.

Mesos provides the necessary functionality to allow Urika-GX to share system resources between the three diverse frameworks. Frameworks on Mesos are diverse and have different scheduling needs. The scheduling needs are often based on the design goals of the framework. For example, some analytics frameworks want to provide data locality. In such case, the framework scheduler choses to schedule compute tasks on the nodes with relevant data. The way the frameworks address fault tolerance and high availability could differ too. Having multiple frameworks on one common platform means sharing the underlying compute resources for launching jobs of different kinds of frameworks. Since different frameworks have unique scheduling requirements, it is difficult to serve the needs of all of them through a centralized scheduler. Even if we chose to implement one, it would not be scalable for the needs of future frameworks. Apache Mesos does not implement a centralized scheduler. Instead, it delegates the scheduling decision to frameworks. Mesos has a distributed two-level scheduling mechanism called resource offers. It encapsulates a bundle of resources that can be allocated to a framework on a cluster node to run tasks into resource offers.

Figure 1 shows the Mesos resource offer mechanism. Mesos decides how many resources to offer to each framework based on an organizational policy. The frameworks decide which resources to accept and which job to run on them. The task scheduling and execution decisions are left to the framework. Each framework implements a scheduler of its own. This enables the framework to implement diverse solutions to various problems in the cluster example: data locality and fault tolerance which can evolve independently. While the decentralized scheduling model is not always the globally optimal solution, it is efficient to support diverse workloads.

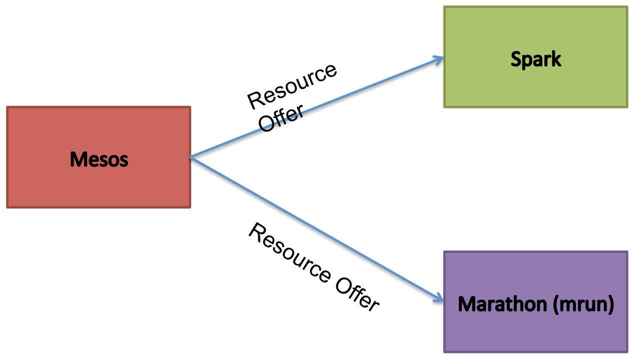


Figure 1: Mesos Resource Offer

Like many other distributed systems, Mesos has been implemented in a master-worker pattern. Mesos consists of Mesos Master, Mesos Agent, and Frameworks. There is one Mesos agent/slave daemon that runs on each node of the cluster. There is a Mesos master that manages agent/slave daemons running on each cluster node. Frameworks that register with Mesos run tasks on these agents. A resource offer comprises a list of free resources on multiple agents. Frameworks running on Mesos have two components: Scheduler, a process that registers with master to be offered resources, and Executor, a process that is launched on an agent node to run the framework tasks.

III. Description of workflows

To illustrate an HPC workflow, we use an MPI application. The data set we use for this has been generated from multi-spectral images. The entries in the input file represent pixels of the spectral images, where each pixel is represented by the integer value of the class that it belongs to (a result of segmentation and clustering). We use two files of different sizes, one with 1024 rows, and other with 2048 rows and these files reside on lustre. We use a sequential code for an image processing algorithm which performs smoothing operations on an image represented by our input file as a starting point. An MPI application developed as part of graduate coursework of one of the authors [10] is being used here. The application parallelizes the original algorithm using MPI using a 1-D block row-wise data distribution. Further, smoothing is performed in an iterative fashion on the cells. The goal of smoothing is to change the class that a pixel has been assigned to, if a majority of neighboring elements have a different class. For each pixel, the two neighborhood pixels in each direction are analyzed and the algorithm runs ten iterations. Ghost cells are used to perform smoothing and these are continuously communicated between the various processes using MPI for every iteration of smoothing. Once the smoothing is performed, the processes write the smoothed pixel values back to the image.

To illustrate analytics workflow, a Pokemon dataset was selected which includes 721 Pokemon. Each line consists of the following fields: Id for each pokemon, name of the pokemon, primary type of the pokemon, sum of the existing pokemon statistics, hit points, base modifier for normal attacks, base damage resistance against normal attacks, special attack, base damage resistance against special attacks, and speed which determines which pokemon attacks first each round. For more details about the data set look at https://www.kaggle.com/abcsds/pokemon. The data is stored in csv format. We developed a few simple spark applications, each of which perform simple operations on this dataset such as ”list all the pokemons grouped by primary type”, ”list all pokemons grouped by both primary and secondary type”, and ”list all pokemons grouped by generation”.

A. *Running the workflow on Apache Mesos*

1) Running MPI application on Mesos: Marathon is a framework in the Mesos ecosystem which supports long running web services. At Cray, we leveraged Marathon to launch HPC applications on Mesos. Cray developed a Marathon Framework Application Launcher which does the Aries setup required for PGAS/DMAPP [11] called **”mrun”**. **mrun** allows for more precise control of system resources, and better ability to clean up error cases that may arise when running HPC tasks.

Here, Marathon receives resource offers from Mesos. When a user submits an **mrun** job, marathon accepts the resources from Mesos and gives them to the **mrun** which runs as a marathon application. From there, the HPC job is scheduled as a regular marathon application utilizing the resources it receives from Mesos. Initially, all resources for each node will be allocated to **mrun**, and all nodes are assumed to be homogeneous with equal core counts and RAM. **mrun** will launch one setuid-root helper app on each node, which will in turn initialize Aries communication as needed, set core affinity for each helper process to spread processes evenly between the sockets on each node as desired, setuid() to the user and finally fork()/execvp() the correct number of application instances per node. Once the job finishes running, the helper process releases the resources back to Mesos.

2) Running Spark application on Mesos: Spark applications are launched using the ”spark-submit” command. When a Spark job is submitted, spark is registered as a framework with Mesos. Mesos gives resource offers to all the frameworks that are registered with it. When the spark job receives resource offers from Mesos, Spark choses either to accept or reject those resources. By default, even if spark requests for more resources than the resource offer Mesos provides, spark will accept the resource offer. The resources are granted to spark. Spark then schedules the spark job on the offered resources. This is shown in Figure 2. Once the job finishes running, the resources are released by Spark back to Mesos. This is shown in Figure 3.

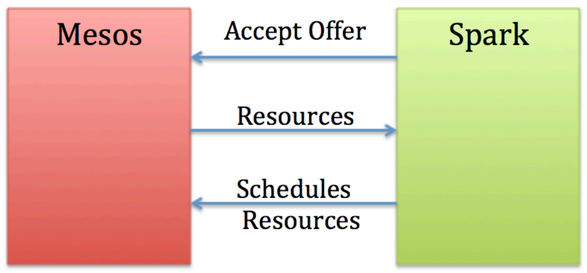


Figure 2: Spark job launch

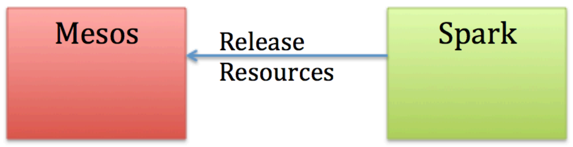


Figure 3: Spark releases resources after job launch back to Mesos

B. Running the workflow on Slurm

1) Running HPC workflow on Slurm: **salloc**

The first step in launching a job on Slurm is to grab an allocation. When the user invokes **salloc** for an HPC job, the resource request is enqueued to slurmctld (the central manager), and the user waits until the resource request can be satisfied (or is timed out) at which time the user's prompt is returned, and their shell is updated with various environment variables describing the resources slurm has set aside for use. In Figure 4, the user has asked for 4 free CPU cores and will be limited to using no more than those 4 cores for the duration of this **salloc** request. Slurm will likely satisfy this request with a single multi-core node, but if the compute environment contained some single or dual-core CPU nodes (unlikely in any Cray XC system), then slurm would be free to allocate two dual-core nodes, 4 single-core nodes, or 1 dual-core and two single core nodes.

The job is launched using **srun**. When the user invokes **srun** after an **salloc**, **srun** communicates to slurmctld and requests are sent to slurmd daemons running on the compute nodes to configure the Aries network across the nodes reserved, and to fork/exec the correct number of instances of the application on each node requested. When the application finishes, slurm will clean-up its internal Aries network maps and return the user to their prompt, still within the original **salloc** environment. Only when the user exits from the **salloc** shell are the resources released back to slurm.

2) Running Spark application on Slurm: Shifter images are used to run Spark on slurm. Shifter images are spun up using **salloc** which in turn spins up a standalone spark cluster in the allocation received. The user runs spark-submit as in the case of Mesos, except now, it operates within the spark stand alone resource manager.

IV. Comparison of different workload managers

There is a fundamental difference in how these workload managers are designed to perform resource allocation and scheduling. HPC workload managers expect the jobs to request resources and then grant the resources for the jobs based on the current resource availability. In contrast, Mesos keeps offering resources to the frameworks registered with it and leaves it to the frameworks either to accept or reject the resource offers.

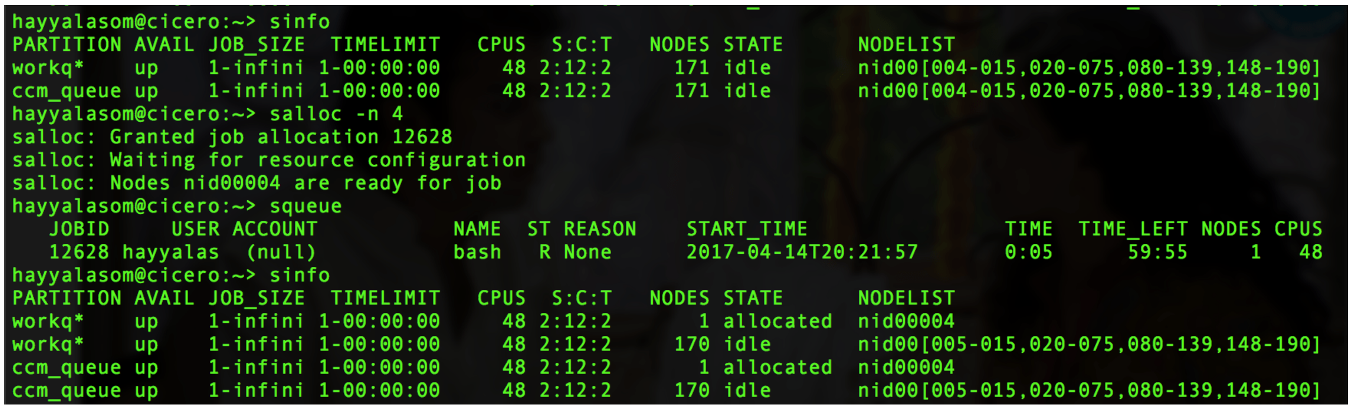
Due to the resource offer model, Apache Mesos provides better resource utilization. We illustrate this by running a simple experiment on our platforms.

Multiple jobs are submitted to a platform with slurm as the main resource manager. For this purpose, an XC system with 171 nodes is utilized, see in Figure 5. Several analytics jobs are started on the XC system and keeps the systems fully busy. This is shown in Figure 6.

After recording that no resources are available, an analytics application is submitted to the slurm queue by explictly requesting twenty nodes. Since there are no nodes available currently, note that the job is waiting there for resources to be available. Seven nodes are then freed up.

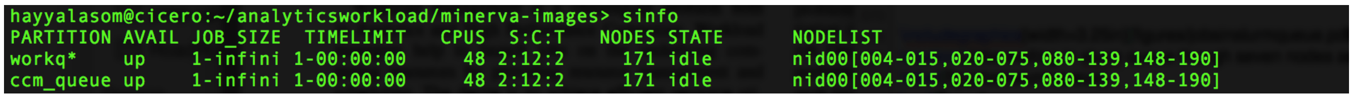
This is shown in Figure 7. Note the status of the analytics application just submitted. Though seven nodes are idle, observe that the application is still waiting for resources. Next, submit the MPI application requesting for twenty five nodes. sinfo shows that along with the analytics image, the MPI application is also waiting for resources, though there are seven nodes that are idle. This is shown in Figure 8. The resource requirements of each of these applications cannot be met by the current resource availability of Slurm and they continue to wait in the queue until the minimum resource requirements are met.

Multiple jobs are submitted to a platform with Mesos as the main resource manager. A Urika-GX system with 41 compute nodes is utilized for this. This is shown in Figure 9a. Forty one jobs are submitted to the system. All nodes are utilized and the system is busy. A snapshot of the system resource availability is taken. This is shown in Figure 11. Submit a spark job explicitly requesting for 128 cores by using –total-executor-cores flag and observe the behavior. The spark job continues to wait for resources infinitely. Free up one node and note available resources. This is shown in Figure 9b. Look at the current resource statistics and observe that only 36 cores are available. This is less that what was requested explicitly for the job. Due to the resource offer model, even if mesos offers less resources than that is originally available, Spark can accept resources less than the original requirement, and schedules its job. Once



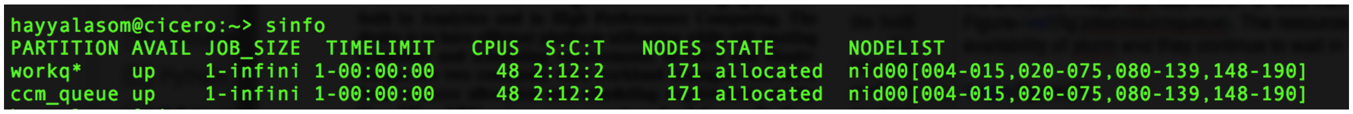
```
hayyalasom@cicero:~> sinfo
PARTITION AVAIL JOB_SIZE  TIMELIMIT   CPUS  S:C:T   NODES STATE      NODELIST
workq*    up    1-infini 1-00:00:00     48 2:12:2     171 idle       nid00[004-015,020-075,080-139,148-190]
ccm_queue up    1-infini 1-00:00:00     48 2:12:2     171 idle       nid00[004-015,020-075,080-139,148-190]
hayyalasom@cicero:~> salloc -n 4
salloc: Granted job allocation 12628
salloc: Waiting for resource configuration
salloc: Nodes nid00004 are ready for job
hayyalasom@cicero:~> squeue
   JOBID     USER ACCOUNT           NAME  ST REASON    START_TIME               TIME  TIME_LEFT NODES CPUS
   12628 hayyalas  (null)           bash   R None      2017-04-14T20:21:57      0:05      59:55     1   48
hayyalasom@cicero:~> sinfo
PARTITION AVAIL JOB_SIZE  TIMELIMIT   CPUS  S:C:T   NODES STATE      NODELIST
workq*    up    1-infini 1-00:00:00     48 2:12:2       1 allocated  nid00004
workq*    up    1-infini 1-00:00:00     48 2:12:2     170 idle       nid00[005-015,020-075,080-139,148-190]
ccm_queue up    1-infini 1-00:00:00     48 2:12:2       1 allocated  nid00004
ccm_queue up    1-infini 1-00:00:00     48 2:12:2     170 idle       nid00[005-015,020-075,080-139,148-190]
```

Figure 4: Resources being allocated by Slurm using the salloc command



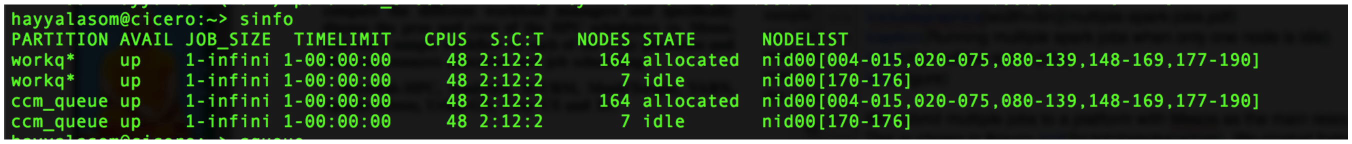
```
hayyalasom@cicero:~/analyticsworkload/minerva-images> sinfo
PARTITION AVAIL JOB_SIZE  TIMELIMIT   CPUS  S:C:T   NODES STATE      NODELIST
workq*    up    1-infini 1-00:00:00     48 2:12:2     171 idle       nid00[004-015,020-075,080-139,148-190]
ccm_queue up    1-infini 1-00:00:00     48 2:12:2     171 idle       nid00[004-015,020-075,080-139,148-190]
```

Figure 5: Idle XC system, all resources available



```
hayyalasom@cicero:~> sinfo
PARTITION AVAIL JOB_SIZE  TIMELIMIT   CPUS  S:C:T   NODES STATE      NODELIST
workq*    up    1-infini 1-00:00:00     48 2:12:2     171 allocated  nid00[004-015,020-075,080-139,148-190]
ccm_queue up    1-infini 1-00:00:00     48 2:12:2     171 allocated  nid00[004-015,020-075,080-139,148-190]
```

Figure 6: Busy XC system, no resources available



```
hayyalasom@cicero:~> sinfo
PARTITION AVAIL JOB_SIZE  TIMELIMIT   CPUS  S:C:T   NODES STATE      NODELIST
workq*    up    1-infini 1-00:00:00     48 2:12:2     164 allocated  nid00[004-015,020-075,080-139,148-169,177-190]
workq*    up    1-infini 1-00:00:00     48 2:12:2       7 idle       nid00[170-176]
ccm_queue up    1-infini 1-00:00:00     48 2:12:2     164 allocated  nid00[004-015,020-075,080-139,148-169,177-190]
ccm_queue up    1-infini 1-00:00:00     48 2:12:2       7 idle       nid00[170-176]
```

Figure 7: XC system with 7 idle nodes

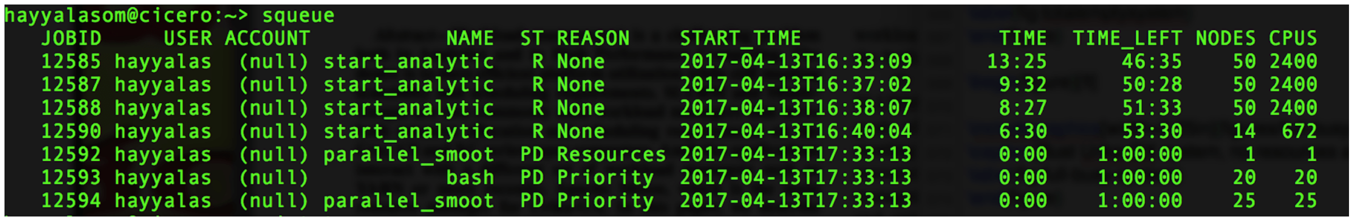
this job completes, the user submits multiple jobs which run in the background when the same set of resources are available (36 cores). It is observed that all the jobs finish running one after the other. This is shown in Figure 10.

It is observed that, due to the resource offer model of Mesos, better resource utilization is achieved, while in slurm, though there are some nodes idle, if the resource requirement is not met, the jobs are still waiting in the queue while the platform is not fully utilized. Further, Mesos does not have a global queue. Each framework has its own scheduler. The main advantage to this is that, for future frameworks with diverse scheduling needs, it will be easier to port it to Mesos. However, it comes with a downside that, a fair scheduler across all frameworks is not available, which we are used to in the HPC world.

HPC workload managers facilitate Aries network setup which makes it easy to launch HPC applications. While this is not a built in option with Mesos, in order to facilitate running HPC workloads on on analytics platforms, with significant development effort, Cray developed **mrun** which uses Marathon to negotiate resources from Mesos. Similarly, on our XC platform, we use the shifter container technology to launch analytics workloads on the same platform.

V. Conclusion

We discussed our experiences using different workload managers across Cray platforms (analytics and HPC) in this paper. Different workload managers were described, launching both analytics and HPC workloads on Slurm and Mesos, using two sample workflows was also described. The main differences between these workload managers were highlighted. Using some simple experiments, it was demonstrated that with the resource offers model of Mesos, better resource utilization can be achieved as compared to Slurm. Further, the downside to using the open source version of Apache Mesos in its current state is that the user misses out on having a global queue and a fair share scheduler which are offered by Slurm. While the resource offer model is different from the traditional way workload managers work(jobs request for resources from workload



```
hayyalasom@cicero:~> squeue
   JOBID      USER ACCOUNT           NAME  ST REASON    START_TIME              TIME  TIME_LEFT NODES CPUS
   12585 hayyalas  (null) start_analytic   R None      2017-04-13T16:33:09     13:25      46:35    50 2400
   12587 hayyalas  (null) start_analytic   R None      2017-04-13T16:37:02      9:32      50:28    50 2400
   12588 hayyalas  (null) start_analytic   R None      2017-04-13T16:38:07      8:27      51:33    50 2400
   12590 hayyalas  (null) start_analytic   R None      2017-04-13T16:40:04      6:30      53:30    14  672
   12592 hayyalas  (null) parallel_smoot  PD Resources 2017-04-13T17:33:13      0:00    1:00:00     1    1
   12593 hayyalas  (null)           bash  PD Priority  2017-04-13T17:33:13      0:00    1:00:00    20   20
   12594 hayyalas  (null) parallel_smoot  PD Priority  2017-04-13T17:33:13      0:00    1:00:00    25   25
```

Figure 8: Jobs waiting in slurm queue though seven nodes are idle

managers), Mesos has its own advantages such as better resource utilization and an ability to support diverse frameworks with different scheduling requirements on the same platform. As future platforms are considered, work on trying to simulate the missing features of each of these worklaod managers in the other such as trying to add global queue to Mesos should begin. This way, the best of both the worlds could be achieved, having efficient resource utilization as well as fair share scheduling.

Acknowledgments

The authors would like to thank **Kristyn Maschhoff** for her ideas which served as the starting point for putting this paper together. We would like to thank **Mike Ringenburg** for coaching us, reviewing the paper and also help in brainstorming the use cases to illustrate the comparison of different workload managers. We thank our team for providing useful feedback and comments.

References

[1] "Slurm Workload Manager," https://slurm.schedmd.com.

[2] "Adaptive Computing Moab," http://www.adaptivecomputing.com/products/hpc-products/moabhpcbasicedition/.

[3] "Adaptive Computing Torque," http://www.adaptivecomputing.com/products/open-source/torque/.

[4] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, and E. Baldeschwieler, "Apache hadoop yarn: Yet another resource negotiator," in *Proceedings of the 4th Annual Symposium on Cloud Computing*, ser. SOCC '13. New York, NY, USA: ACM, 2013, pp. 5:1–5:16. [Online]. Available: http://doi.acm.org/10.1145/2523616.2523633

[5] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica, "Mesos: A platform for fine-grained resource sharing in the data center," in *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'11. Berkeley, CA, USA: USENIX Association, 2011, pp. 295–308. [Online]. Available: http://dl.acm.org/citation.cfm?id=1972457.1972488

[6] "Apache Mesos," http://mesos.apache.org/.

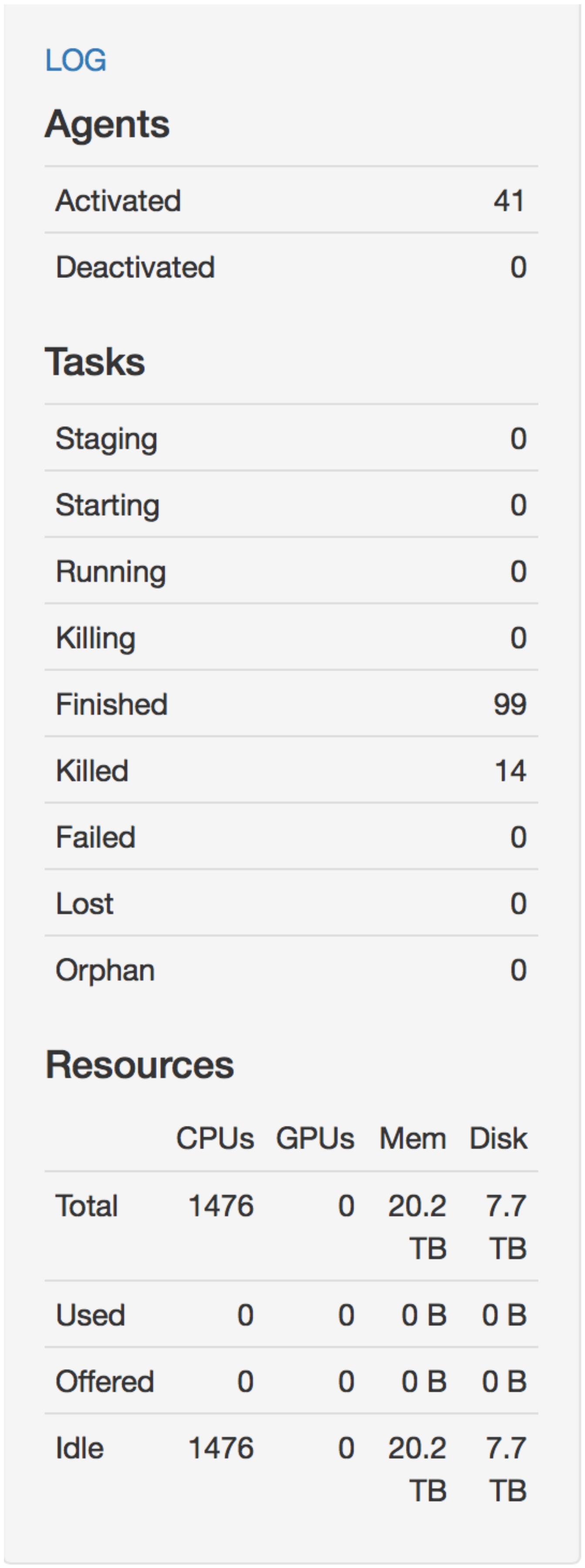
[7] H. Ayyalasomayajula and K. Maschhoff, "Experiences Running Mixed Workloads on Cray Analytics Platforms," in *Cray User Group Conference (CUG '16)*, London, UK, 2016.

[8] "Spark Lightning-fast cluster computing," http://spark.apache.org.

[9] "Apache Hadoop," http://hadoop.apache.org/.

[10] E. Gabriel, S. Shah, and H. Ayyalasomayajula, "Parallel Computations Graduate Course Fall 2013," http://www2.cs.uh.edu/g̃abriel/courses/cosc6374_f13/ParCo_09_hw1.pdf.

[11] T. Johnson, "Coarray C++," in *7th International Conference on PGAS Programming ModelsCray User Group Conference*, Edinburgh, Scotland, 2013.


LOG

Agents

Activated	41
Deactivated	0

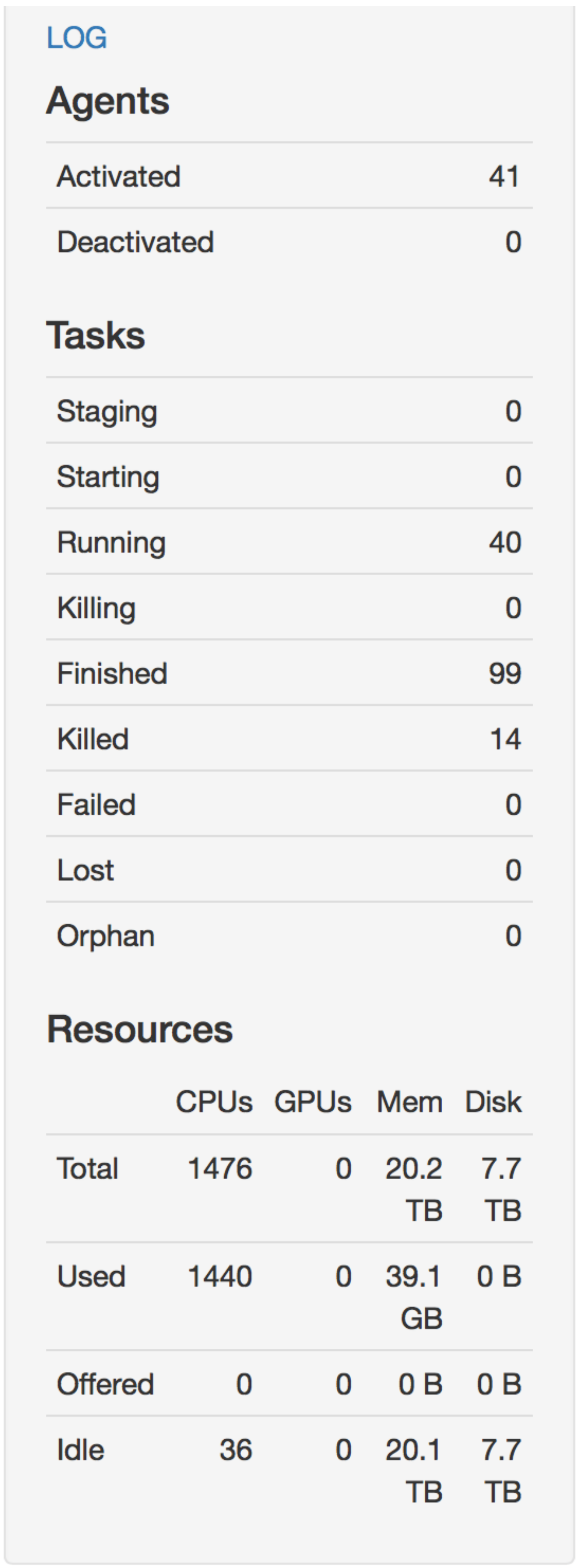
Tasks

Staging	0
Starting	0
Running	0
Killing	0
Finished	99
Killed	14
Failed	0
Lost	0
Orphan	0

Resources

	CPUs	GPUs	Mem	Disk
Total	1476	0	20.2 TB	7.7 TB
Used	0	0	0 B	0 B
Offered	0	0	0 B	0 B
Idle	1476	0	20.2 TB	7.7 TB

(a) Idle Urika-GX system, all resources available


LOG

Agents

Activated	41
Deactivated	0

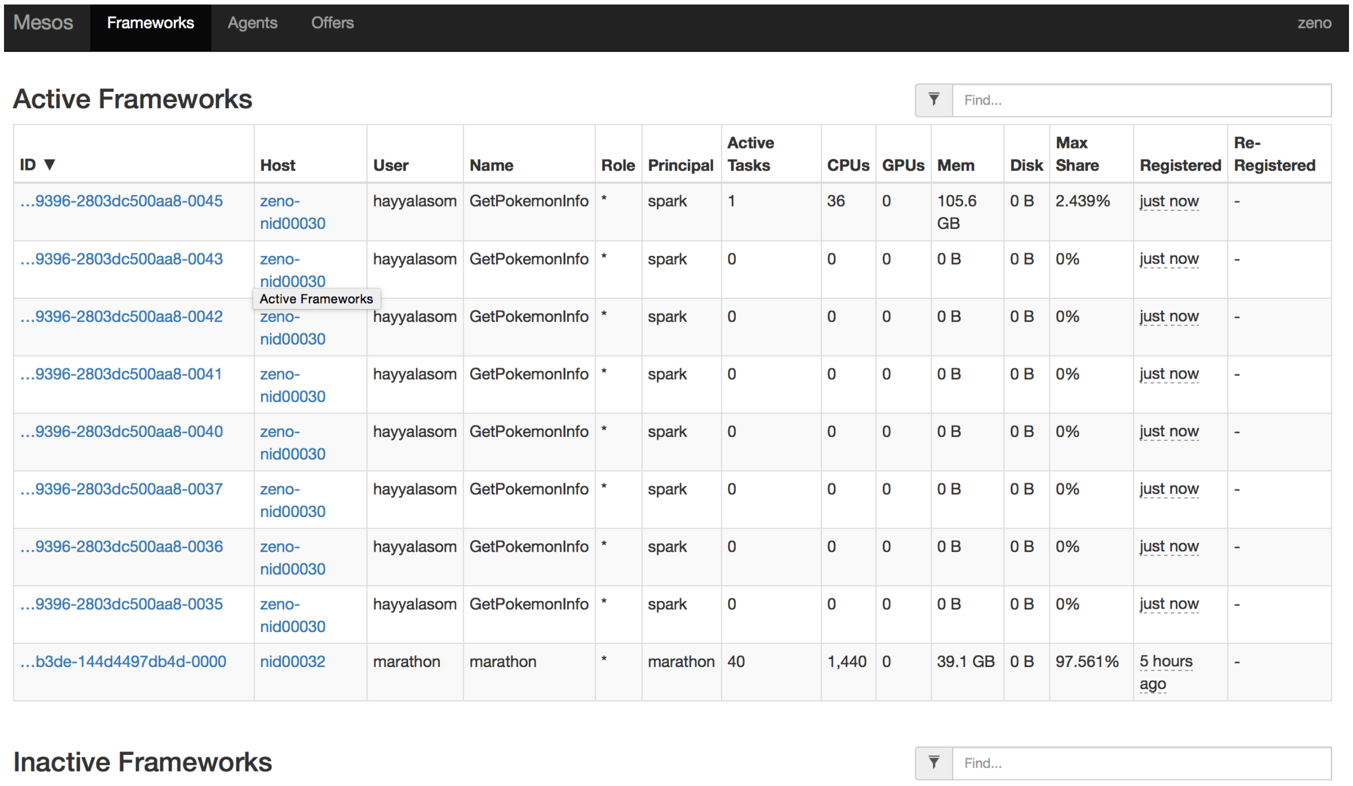
Tasks

Staging	0
Starting	0
Running	40
Killing	0
Finished	99
Killed	14
Failed	0
Lost	0
Orphan	0

Resources

	CPUs	GPUs	Mem	Disk
Total	1476	0	20.2 TB	7.7 TB
Used	1440	0	39.1 GB	0 B
Offered	0	0	0 B	0 B
Idle	36	0	20.1 TB	7.7 TB

(b) Only one node resources available



Active Frameworks

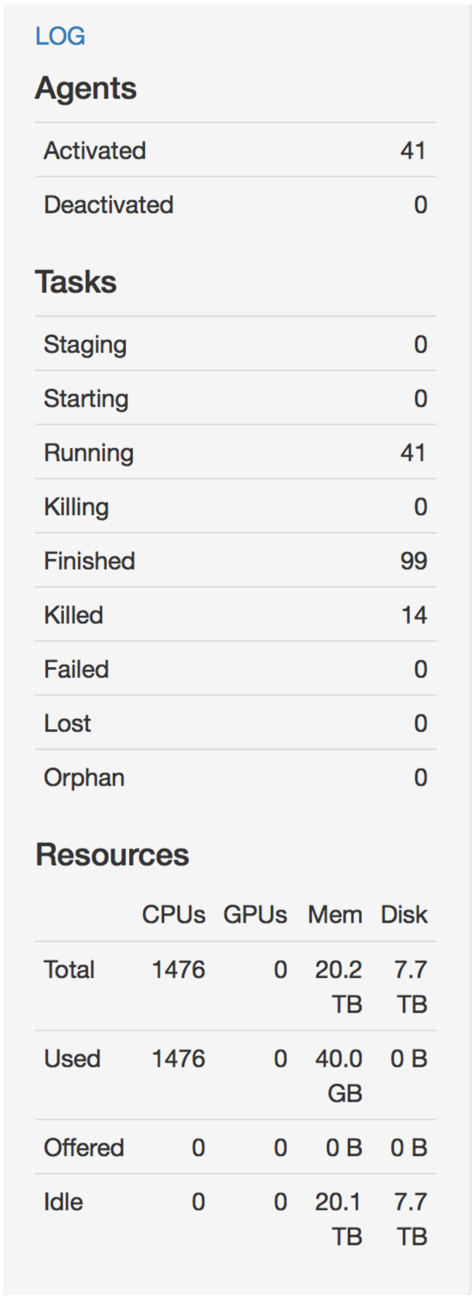
Find...

ID ▼	Host	User	Name	Role	Principal	Active Tasks	CPUs	GPUs	Mem	Disk	Max Share	Registered	Re-Registered
…9396-2803dc500aa8-0045	zeno-nid00030	hayyalasom	GetPokemonInfo	*	spark	1	36	0	105.6 GB	0 B	2.439%	just now	-
…9396-2803dc500aa8-0043	zeno-nid00030	hayyalasom	GetPokemonInfo	*	spark	0	0	0	0 B	0 B	0%	just now	-
…9396-2803dc500aa8-0042	zeno-nid00030	hayyalasom	GetPokemonInfo	*	spark	0	0	0	0 B	0 B	0%	just now	-
…9396-2803dc500aa8-0041	zeno-nid00030	hayyalasom	GetPokemonInfo	*	spark	0	0	0	0 B	0 B	0%	just now	-
…9396-2803dc500aa8-0040	zeno-nid00030	hayyalasom	GetPokemonInfo	*	spark	0	0	0	0 B	0 B	0%	just now	-
…9396-2803dc500aa8-0037	zeno-nid00030	hayyalasom	GetPokemonInfo	*	spark	0	0	0	0 B	0 B	0%	just now	-
…9396-2803dc500aa8-0036	zeno-nid00030	hayyalasom	GetPokemonInfo	*	spark	0	0	0	0 B	0 B	0%	just now	-
…9396-2803dc500aa8-0035	zeno-nid00030	hayyalasom	GetPokemonInfo	*	spark	0	0	0	0 B	0 B	0%	just now	-
…b3de-144d4497db4d-0000	nid00032	marathon	marathon	*	marathon	40	1,440	0	39.1 GB	0 B	97.561%	5 hours ago	-

Inactive Frameworks

Find...

Figure 10: Running multiple spark jobs when only one node is idle



LOG

Agents

Activated	41
Deactivated	0

Tasks

Staging	0
Starting	0
Running	41
Killing	0
Finished	99
Killed	14
Failed	0
Lost	0
Orphan	0

Resources

	CPUs	GPUs	Mem	Disk
Total	1476	0	20.2 TB	7.7 TB
Used	1476	0	40.0 GB	0 B
Offered	0	0	0 B	0 B
Idle	0	0	20.1 TB	7.7 TB

Figure 11: Busy Urika-GX system, no resources available