Job Throughput Analysis of Data Center Scheduling Algorithms

Ankit Sharma, Gerardo Ravago
ECE, College of Engineering
Boston University
Email: {ankitsh, gcravago}@bu.edu

***Abstract*—A resource manager is an essential part of any data center. How the jobs are scheduled on the compute nodes is as important as the hardware in the data center. The popular term for such tools is RJMS which stands for Resource and Job Management System[1]. We utilize one of the most widely used RJMS[1] on the planet which is SLURM. Simple Linux Utility for Resource Management is used in 60 percent of the world's fastest supercomputers. It is the middleware having the responsibility for allocating resources to jobs and scheduling jobs to all the cluster nodes. In this project, we present an experimental methodology to evaluate RJMS, and prove it on our own mini data center in a star topology network configuration consisting of a master node and 7 compute nodes. We evaluate SLURMs capability of scheduling large jobs and also how different scheduling algorithms affect job throughput. We test SLURMs capabilities with PARSEC benchmark applications with both SIMLARGE and NATIVE inputs. We also implement synthetic workloads which we refer to as the ESP(Effective System Performance) workload and the Uniform workload. The emulated data center is tested with the synthetic workloads and the results are reported with a detailed analysis of SLURM schedulers.**

I. Introduction

Good resource management is a powerful skill, both in life and computing. How we allocate resources decides how efficient we are, and that's true for computers as well. Having a good resource management and job scheduler is essential for any system. And in a data center, it matters even more because we have many compute nodes and they have to be utilized efficiently. Such tools are referred to as RJMS. SLURM is one of the most widely used RJMS tool in the research community[2]. The goal of an RJMS tool is to efficiently allocate jobs to the compute nodes while satisfying the resource requirements of the job.

High performance computing is not only limited to supercomputing, but with the proliferation of data centers and cloud computing, corporations are investing in large clusters to service user jobs. To give an example, Facebook has many server farms all over the country. Their server farm in Oregon is almost 307,000 square feet, larger than two Walmart stores[7]. Research labs need to run simulations for HPC applications as well, and a smart resource manager is needed at the helm to direct all resources efficiently and purposefully. That's the motivation behind our project.

Although SLURM stands for Simple Linux Utility for Resource Management, there is nothing simple about it anymore. It has over 500,000 lines of C code and a thriving research community backing it up. It was developed at Lawrence Livermore National Laboratory with collaboration of SchedMD, Linux NetworX, Hewlett-Packard, and Groupe Bull as a Free Software resource manager. It was inspired by Quadrics RMS which was a closed source software package. It has 100s of plugins and supports multitude of operating systems.

This paper aims to analyse the job throughput which a data center we devised on the cloud can provide using SLURM. The detailed analysis follows in future sections with the two scheduling algorithms which SLURM comes packaged with. The workloads we use for our project are real world and also synthetic. This project was undertaken for the class 'EC700 Advanced Computer Systems' in Fall 2015 at Boston University under the guidance of Prof. Ayse Coskun.

This paper is divided into 7 sections. Section 1 is the introduction of the work we are presenting, Section 3 lists the experimental methods and description of the methods we employed to carry our experiments, Section 4 details the proposed technique of our research, Section 5 presents our results, Section 6 is the conclusion and references is the last section of our paper.

II. Background

Resource Management Software

Computing clusters of any size require a RJMS in place which manages how work is assigned to servers within the data center. It exists to optimize a number of objective functions such as job fairness, job throughput, and minimizing data center costs. With these types of responsibilities, the efficiency of the RJMS corresponds directly with the overall efficiency of the data center in a way thats agnostic of the underlying hardware.

SLURM is a production ready open source RJMS developed for Linux computing clusters [2]. Since its release it has become the dominant HPC cluster manager for a data centers around the world powering 5 of the top 10 supercomputers in the world [3]. The reason for this is its maturity as a software project and well developed features.

At its core, SLURM is designed for scalability and high performance using a relatively simple architecture. Figure 1 illustrates the SLURM architecture on the server level. Users submit their jobs to the control daemon which creates a priority queue of submitted and pending jobs. The control daemon

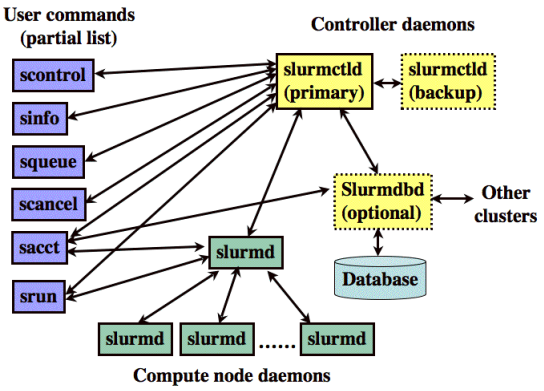

Fig. 1. SLURM Architecture [3]

then computes a schedule to execute these jobs based on the current usage of the compute nodes. Generally, the objective for the SLURM scheduler is to maximize job throughput but other higher level objectives such as energy usage may also be considered [4].

Using a plugin architecture, SLURM can be extended to support many other features. One notable module is an extensive accounting plugin which allows for the tracking of detailed resource utilization information. This is particularly useful for monetization based on usage, but also for measuring performance for research.

Scheduling Policies

By default, SLURM provides two basic job scheduling plugins known as builtin and backfill. New scheduling policies may be implemented via the plugin interface which uses a C based API that exposes the job queue, current system state, and methods to schedule jobs.

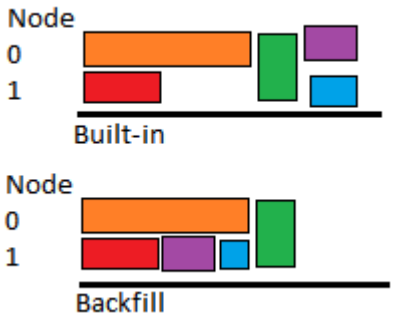

Fig. 2. Comparison of builtin and backfill scheduling policies

The builtin plugin implements a simple first-in first-out algorithm(FIFO). Jobs are assigned a priority based on their submission time. Once the resources for the job at the top of the queue are available, the job is scheduled immediately.

Backfill starts out by assigning an initial FIFO priority to incoming jobs. It then assigns higher priority to jobs that can be scheduled without affecting the expected start time of higher priority jobs. The expected start time in turn depends on the user reported maximum run time of the job. The advantage gained from this policy is illustrated in Figure 2. Notice how backfill is able to recognize that the blue and purple jobs can execute on Node 1 without affecting the start time of the green job by going back and filling in the spaces left behind by the builtin FIFO scheduler. In doing so backfill improves job throughput by increasing utilization.

Effective System Performance Benchmark

Effective system performance (ESP) was devised as a way to assess how effective a system is in providing access to the performance it is measured to have [5]. Its strength as a benchmark lies in its ability to assess the efficacy of the RJMS agnostic of the hardware that it is being run on.

In order to accomplish this, ESP defines a set of jobs of varying resource requirements and runtimes which are modeled after real traces from a large data center. These jobs are enumerated in Figure 3. The resource requirements and runtimes of these jobs can be scaled to the size of the cluster as necessary.

What's important is that the ratios are maintained in order to ensure comparability between runs. This means that the run times of the jobs must not be influenced by the underlying hardware. The ordering of job submission doesn't matter, but the literature suggests scheduling in a random order with a known seed.

Two “Z-Test” jobs are included which are meant to stress test the RJMS by reserving all the CPUs in the cluster. These must occur after 10 percent of the theoretical best execution time and before 90 percent of the actual running time. This ensures that there are adequate jobs before and after within the job queue that create a difficult scheduling problem.

Job Type n	Fraction of Job Concurrency relative to total system size δ	Count of the number of Job Instances κ	Target Runtime (Seconds) π
A	0.03125	75	267
B	0.06250	9	322
C	0.50000	3	534
D	0.25000	3	616
E	0.50000	3	315
F	0.06250	9	1846
G	0.12500	6	1334
H	0.15820	6	1067
I	0.03125	24	1432
J	0.06250	24	725
K	0.09570	15	487
L	0.12500	36	366
M	0.25000	15	187
Z	1.00000	2	~100
Total		230	

Fig. 3. ESP2 Workloads [5]

PARSEC

The Princeton Application Repository for Shared-Memory Computers (PARSEC) benchmarks comprise a set of multi-threaded applications for research purposes. While traditionally used for microarchitecture research, they provide a useful

pool of standardized applications. Due to the different requirements of each application, these jobs provide a good baseline for composing a data center level benchmark.

Massachusetts Open Cloud:

Massachusetts Open Cloud (MOC) is a public cloud computing cluster operated by a consortium of Massachusetts based universities. This cluster runs the cloud management software known as OpenStack. With OpenStack users are able to quickly spin up virtual machines and setup a complete computing cluster on the cloud.

III. Experimental Methods and Description

We use OpenStack, an open source cloud platform. We spin up eight two core Ubuntu 14.04 LTS 64 bit virtual machines. One node is the primary node and seven are compute nodes. We install SLURM on all the nodes. Slurmctld is the control daemon which runs on the primary node and slurm daemons run on the compute nodes. The seven nodes are connected in a star topology to the primary node.

After setting all the nodes and installing the authentication plugin, we write the slurm.conf file for the cluster. Slurm.conf is a html generated configuration file. The configuration file is responsible for holding all information which SLURM needs for its functionality. It asks the user to specify the primary controller and information about all compute nodes. All scheduling information is to be provided in this configuration file. The configuration file must be identical on all nodes which are connected to the primary.

After setting up SLURM, we install the PARSEC benchmark. We test the scalability of SLURM and also test different scheduling algorithms integrated within SLURM. We take four applications from PARSEC benchmark based on different types of working set. The applications we pick are blackscholes, canneal, facesim and streamcluster. We run 100, 500 and 1000 jobs with two threads as we have a two core CPU on the virtual machine.

In the ESP paper, the authors develop the workload for 80,000 nodes and schedule jobs using SLURM. We scale the paper down to 7 nodes so that we can schedule jobs on our mini cluster. We then schedule jobs on our 7 compute nodes using SLURM. We also develop a uniform workload to expose the limitation of backfill scheduling algorithm and use PARSEC jobs to demonstrate our approach.

The ESP workload consists of sleep jobs which reserve the CPU resource for the time specified. We experiment with different times to see how backfill and builtin scheduling algorithms work and also experiment with different number of nodes to test how changing number of nodes affects the job throughput and schedulability of applications. We ran 230 jobs over different intervals and times of the day to account for any effects other applications sharing the hardware might have on our runs. We didn't find any difference in the run. We realised that's because we are running sleep jobs which just take the CPU resource but don't do anything with it, which is fine for our experiment because our purpose is not to stress the CPU, but to test how the applications are being scheduled when the CPU on a node is busy.

TABLE I
Run times using NATIVE input

Application	Run time in Seconds
Blackscholes	134
Canneal	154
Facesim	294
Streamcluster	266

The way SLURM scheduling works is that we submit jobs using the 'sbatch' command. SLURM gives us many commands which we can use to see information about the nodes and the jobs submitted. For more information, kindly refer to the documentation in the reference[2]. 'sbatch' has command line parameters which we can use to have some control on how we schedule the applications. We can use the '-t' parameter to control the timeout of the application, so the application will run for the time we specified, and we can allocate more than one node to a job using the '-N' parameter.

Since SIMLARGE inputs for PARSEC only run for few seconds we installed the NATIVE inputs for PARSEC to have a larger degree of control over the timeout of applications. With NATIVE input the run time for benchmarks for two threads are enumerated in Table 1.

Using the NATIVE benchmarks, we can specify different times in the job submission script with 'sbatch' to evaluate job scheduling algorithms. We create a synthetic workload with timeouts that complement each other. For example, if we give blackscholes a timeout of 1 minute and Streamcluster a timeout of 3 minutes, backfill should schedule blackscholes more times than streamcluster. We also create a synthetic workload called 'UNIFORM' workload in which we keep the timeouts identical for all applications and job scheduling algorithm shouldn't have any affect on job throughput.

To test scalibility we experiment with different queue depths. Queue depth only works with backfill algorithm. It's the size of the job queue which the backfill algorithm looks forward in the queue to schedule the jobs. The default value is 100. We change the queue size to 200 and 400 for a job queue size of 1000. This parameter has to be put in the slurm.conf file. We monitor job queue for an hour and collect the number of jobs completed in that time period.

In retrospect, we created a mini data-center and installed SLURM. We experimented with a real benchmark and also create our own synthetic workloads. We tested the scalability of SLURM and collected job throughput results with different workloads and changed the size of the job queue and also the size of the lookahead job depth. For our project, as of Friday 12:00 AM, we had run more than 35000 jobs which shows how powerful SLURM really is.

IV. Proposed Technique

We propose synthetic workloads to test the workload manager of a data center. We evaluate the different scheduling

algorithms based on the workloads we create and analyze how different types of workload scheduling affect job throughput. Our technique gives us results which are in sync with our intuition and the ESP workload is a great tool to test job scheduling algorithm efficiency.

V. Experimental Results

We first present the results from PARSEC benchmark in Figure 4. The y-axis represents jobs remaining and x-axis represents time. As we can see, backfill has fewer jobs remaining after 250 seconds. Job throughput of backfill is greater than builtin by 31 jobs. We don't see a huge difference

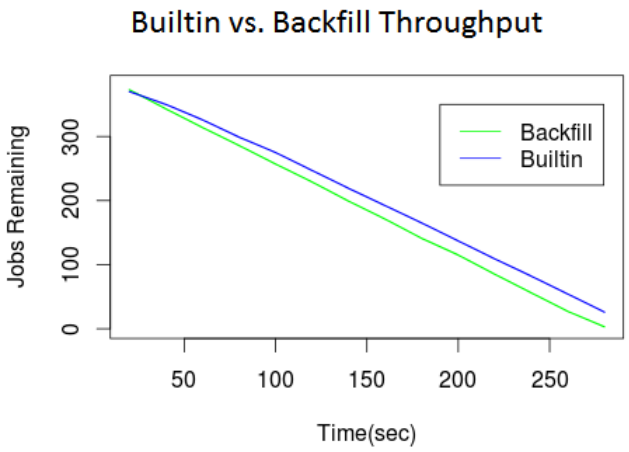

Fig. 4. PARSEC job completion comparison

in the number of jobs remaining because PARSEC is a single node benchmark. So backfill is not able to perform at its peak. In the future, we would like to explore MPI versions of PARSEC.

Effective System Performance

We looked into implementing the ESP benchmark on our cluster as a way to emulate a workload representative of the type serviced by large data centers in a reasonable amount of time. Similar to [1], we have scaled the ESP-2 benchmark to fit our 14-core cluster and cut the 3 hour execution time in half by scaling down the running time of the jobs. Since we are not interested in hardware acceleration, we have opted to use sleep programs that reserve varying numbers of cores as our workload. Figure 5 summarizes our scaled version of the ESP benchmark.

What we learned from running ESP on our cluster with the two default scheduling algorithms was that backfill resulted in a 10 percent reduction in total execution time of the ESP workload over backfill. This highlights the fact that the configuration of RJMS is just as important as the hardware that it is run on. Further, as will be seen in the PARSEC scalability analysis the overhead involved in backfill is insignificant compared to the benefits gained from higher utilization.

Job Type	Nodes	Time	Jobs
A	1	15	75
B	1	18	9
C	4	30	3
D	2	35	3
E	4	18	3
F	1	103	9
G	1	75	6
H	2	60	6
I	1	80	24
J	1	41	24
K	1	28	15
L	1	21	36
M	2	11	15
Z	7	6	2

Fig. 5. Scaled Version of ESP

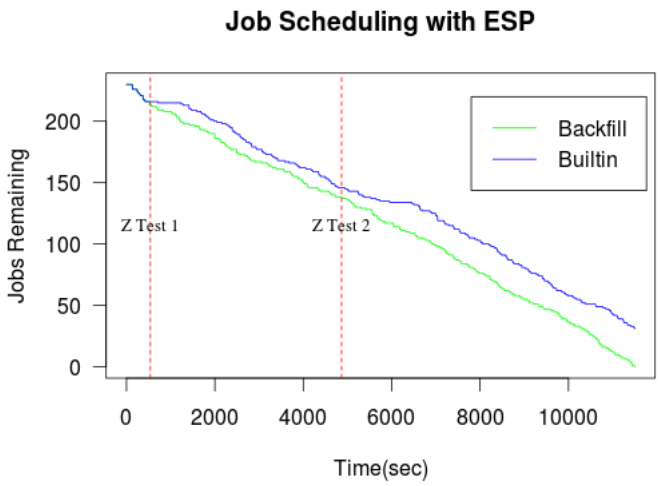

Fig. 6. ESP Job Completion Comparison

We present our ESP results in Figure 6 which plots the total jobs remaining as a function of time starting from the initial 230 scheduled. The sooner a policy can retire all the jobs the better. Backfill is consistently below builtin when running a realistic heterogeneous workload. In absolute value, the difference was more than 1000 seconds for jobs completion.

Note how the slope of the line for backfill is relatively constant while for builtin we notice fluctuations. This suggests that the job throughput for backfill is relatively constant compared to builtin. This is due to backfills ability to consistently keep high utilization without delaying the schedule.

The Z-Test submissions are also interesting points to study as they represent the most difficult scheduling problem in the entire workload. In the figure, we have labeled the expected submission times for each Z-Test. As inefficiencies in the schedule build up, the actual submission time for the Z-Test will be pushed forward. For builtin, this is especially apparent in the 10 minute plateau that follows each expected Z-Test. As the builtin scheduler approaches the Z-Test, it must free up all the nodes by waiting for each to become free and preventing further jobs from scheduling. This results in significantly reduced utilization. Backfill on the other hand

seems relatively unaffected by the Z-Tests because it is able to look ahead and fill in the underutilized nodes.

Uniform Workload

The uniform workload represents the easiest scheduling scenario where all the submitted jobs have the same running time and resource requirements. In this case the scheduler must simply schedule as many jobs as possible back to back.

We developed this benchmark to illustrate a case where a backfill policy may not be an improvement and may actually perform worse due to scheduling overhead. By needlessly looking ahead to opportunistically fill the schedule, backfill may be wasting time that the RJMS could be doing something more useful. Surprisingly, we found during testing that backfill

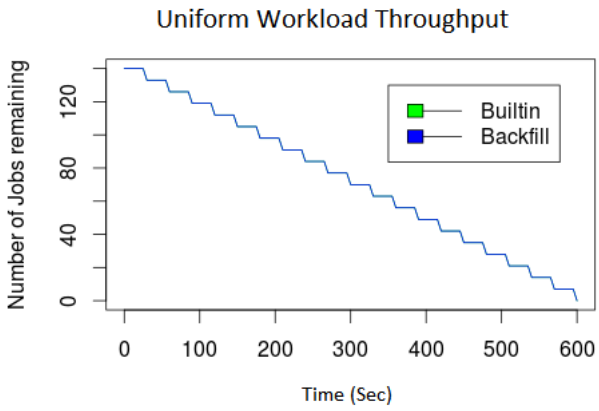

Fig. 7. Uniform Workload Job Completion Comparison

and builtin performed identically. As illustrated in Figure 7, the lines are identical. This may be a result of the small scale of our experiment in that the difference may be so marginal that it will only become apparent on much larger clusters. While plotting, backfill was plotted first and builtin plotted on top of the previous plot, so the two plots are one below the other. On closer inspection, one can see a green tinge below the blue plot of backfill, which might not be visible at the scale of the image in the paper.

In contrast to the ESP plot, the uniform workload represents the throughput of an ideal scheduling policy. As can be seen, the throughput remains at a constant with high utilization. This in turn results in a total execution time identical to the ideal computed.

Scalability Analysis

We also performed scalability analysis on SLURM using PARSEC benchmark. We run 100, 500 and 1000 jobs using combinations of the 4 applications we decided to run from PARSEC benchmark. Our approach was to quantify the time it takes for backfill to retire 100 jobs as we scale the number of jobs submitted. The reason we decided to carry out this

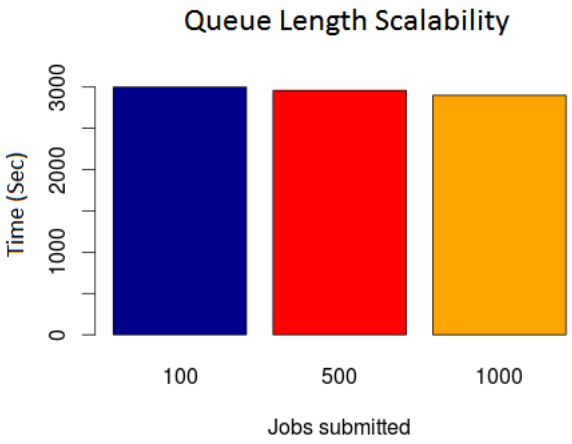

Fig. 8. PARSEC time taken to finish 100 jobs

experiment was to analyze the job throughput of backfill with larger jobs available.

Figure 8 presents the results of our experiment. The y axis represents the number of seconds it took to finish 100 jobs and the x axis represents the number of jobs submitted. We found that there is a slight decrease in the time of completion as we increase the number of jobs submitted but we reason that the difference is not significant because we have a small number of compute nodes available and that the run times of jobs are not vastly different.

The final experiment we carry out is to study changing the queue size of the backfill algorithm. Our reasoning was based on the argument that there will be more overhead for larger queue sizes. Figure 9 shows the results of our experiments.

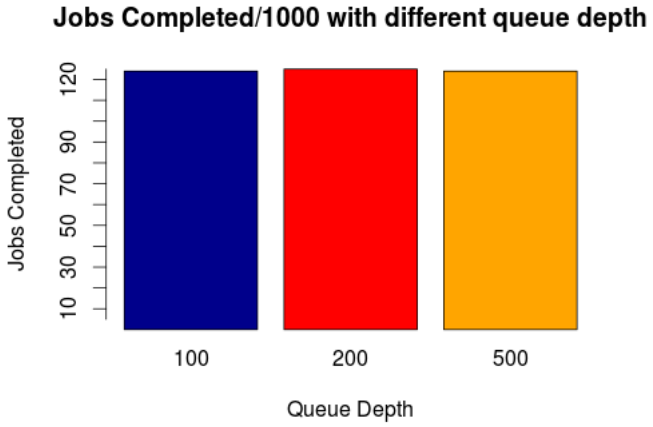

Fig. 9. Jobs completed in an hour PARSEC

The y-axis is the number of jobs completed in an hour and x axis represents the queue depth, the default queue depth is 100. We found that there is minimal difference in job throughput by changing the queue size. We attribute this result to the fact

that since there are only 7 nodes to allocate jobs to, therefore the overhead is minimal.

VI. Conclusion

Using MOC we setup our own small computing cluster with SLURM to prove our experimental methodology. Since our workloads are simply scaled down versions of traces from large data centers we are able to quickly evaluate changes in configuration without the cost and complexity of a real data center. This leads to quick turn around times for results which allows administrators to tune their data centers quickly and efficiently

In studying scheduling policies we proved that switching from a FIFO scheduler to a backfilling scheduler can result in up to a 10 percent increase in throughput when given a heterogeneous workload. Further, we also proved that in most cases backfill has an insignificant overhead when compared to the FIFO policy. In all cases tested, backfill was able to match or improve upon the performance of FIFO even when scaling up its scheduling load.

We have presented an experimental methodology to evaluate the job throughput of RJMS software that is agnostic of the underlying hardware. This is valuable because changes in the configuration of the software can lead to large changes in the overall throughput without changing the hardware.

VII. References

[1] Y. Georgiou and M. Hautreux, "Evaluating Scalability and Efficiency of the Resource and Job Management System on Large HPC Clusters,*" Job Scheduling Strategies for Parallel Processing, 2013.*

[2] A. Yoo, M. Jette and M. Grondona, "SLURM: Simple Linux Utility for Resource Management,*" Job Scheduling Strategies for parallel Processing, 2003.*

[3] SchedMD, "Slurm Workload Manager," [Online]. Available: http://slurm.schedmd.com/slurm.html. [Accessed 10 December 2015].

[4] G. Yiannis, T. Cadeau, D. Glesser, D. Auble, M. Jette and M. Hatreux, "Energy Accounting and Control with SLURM Resource and Job Management System," *Distributed Computing and Networking*, pp. 96-118, 2014.

[5] A. Wong, L. Oliker, W. Kramer, T. Kaltz and D. Baily, "System Utilization Benchmark on the Cray T3E and IBM SP," *Job Scheduling Strategies for Parallel Processing, 2000.*

[6] Christian Bienia, Sanjeev Kumar , Jaswinder Pal Singh and Kai Li. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *Princeton University Technical Report TR-811-08, January 2008.*

[7] Facebook Data Center FAQ:
http://www.datacenterknowledge.com/the-facebook-data-center-faq/