Multiple Objective Scheduling of HPC Workloads Through Dynamic Prioritization

Tyler A. Simon, Phuong Nguyen and Milton Halem
University of Maryland Baltimore County
Computer Science and Electrical Engineering Dept.
tsimo1, phuong3, halem@umbc.edu

Keywords: scheduling, workload modeling, dynamic prioritization, multiobjective optimization

Abstract

We have developed an efficient single queue scheduling system that utilizes a greedy knapsack algorithm with dynamic job priorities. Our scheduler satisfies high level objectives while maintaining high utilization of the HPC system or collection of distributed resources such as a computational GRID. We provide simulation analysis of our approach in contrast with various scheduling strategies of shortest job first; longest waiting jobs first; and large jobs first. Further, we look at the effects of system size on the total workload response time and find that for real workloads, the relationship between response time and system size follows an inverse power law. Our approach does not require system administrators or users to identify a specific priority queue for each of their jobs. The proposed scheduler performs an exhaustive parameter search for a priority calculation per job to balance high level objectives and provide guaranteed performance jobs in a workload. The system administrator needs only tune the prioritization parameters (knobs) and the system scheduler will behave accordingly, such as reducing wait time for jobs that are above average size with small runtimes. We demonstrate that our approach works very well on workloads that have many independent tasks. We evaluate our scheduler on a realistic mixed scientific data processing workload and with a realistic HPC workload trace from the parallel workloads archive.

1. INTRODUCTION

In previous work [15], we have shown that our hybrid scheduler reduces the total workload runtime by more than 50% over the Hadoop Fair Share scheduler. This paper follows up on previous approaches and investigates the scheduler parameters that guarantee efficient job execution for a data intensive satellite gridding application. We further evaluate the scheduling of these workloads with both user and administrator objectives, such as increased throughput, maximum system utilization and minimize wait times for specific job classes.

In this paper, we present a scheduling algorithm that can be used to adapt global objectives for system administrators without the need to manage a variety of priority queues or choose ”rule of thumb” scheduling strategies. As we move toward the many petascale and exascale era, we assume HPC workloads will increase in complexity, i.e. jobs will contain many hundreds of thousands and even millions of individual tasks. Typically scheduling algorithms for high performance computing (HPC) systems involve multiple queues distinguished by priority, job runtime limits and job sizes. Timely response times for large jobs is often best served by backfilling smaller jobs around a single large job if the completion time of the smaller job(s) is determined not to impede the start time of the larger job. We present a prototypical task execution environment and scheduling algorithm that uses a single queue with dynamic prioritization. We show the performance or scheduler using representative HPC workloads from the scientific computing domain, such as Gridding AIRS satellite data [6].

We show, using simulation, how the scheduler can be automatically tuned for high level objectives while maintaining high utilization of the HPC system or a collection of distributed resources such as a computational GRID or Hadoop environment. The simulation analysis we provide examines scheduling strategies of short jobs first, longest waiting jobs first and large jobs first. Further, we look at the effects of system size on the total workload response time and find that for our measured workloads, the relationship between response time and system size follows an inverse power law. A key criteria for our scheduling approach is that we do not require system administrators or users to identify separate queues for each job. The scheduler selects the prioritization parameters (knobs) and the system scheduler will behave accordingly. We demonstrate that our approach works very well on workloads that have many independent tasks, as well as traditional HPC workloads. We select as a case study for evaluating our scheduler a realistic data intensive decadal satellite gridding workload. We have also developed a robust simulator that can take as input a workload log in the standard workload format (SWF) [5] and outputs a variety of statistics that can be useful in workload modeling and system acquisition.

2. RELATED WORK

Task scheduling in distributed systems is a mature and well studied field [3] [9]. In our previous work [15], we demonstrated more than 50% reduction in total workload runtime for a mixed scientific workload using Hadoop with MapReduce [8]. In this paper we extend the formulation of our scheduling algorithm to be used in a more general high performance computing environment. Our approach uses the dynamic prioritization strategy formulated by Ward et al, [22], originally to be used as a relaxed backfill metric. We extend Ward's work by implementing dynamic job priorities using a single queue and using a fractional knapsack algorithm and optimal search of the parameter space. Sandholme and Lai present a dynamic prioritization method that includes per job proportional shares in a Hadoop environment [19], we also address proportional sharing in our approach and present these benefits in section 4..

In 2006, Vanterster and Dimopoulos demonstrated the sensitivity and flexibility of a greedy knapsack based task scheduler [21]. Runtime systems have also been developed to enable users to write and execute efficient Many Task Computing programs in a HPC environment [18] [17] [16]. For this paper, we focus primarily on using dynamic prioritization for achieving high level objectives.

Evolutionary algorithms also have become common approaches to solving multi-objective optimization problems. These algorithms can be effective when used in a dynamic distributed runtime system when considering the problem of scheduling tasks concurrently [1] [2]. Additionally, the task placement problem can be formulated as a multiple objective knapsack problem [20] [10] [13]. This approach provides a simple and efficient algorithm for the simultaneous optimal solutions for mutually conflicting objectives. Our approach is to A.) quantify and adapt to the overall effects of job granularity as well as the number and size of tasks for each job and subsequent workload. B.) Determine how scheduling both coarse and fine grained tasks can be tuned to achieve administrative goals, while achieving near optimal workload response time.

3. SCHEDULER OVERVIEW

Prior to discussing the design of our scheduler we present some definitions of terms used throughout the paper. Tasks are the lowest level units of work requiring exactly one resource unit, typically a processor, they are the finest resolution used in our scheduler and can have variable durations, but are incompressible, thus the time to run a single task cannot be minimized through scheduling alone. A job consists of one or more tasks. A job has both a size, in tasks, and a length, or duration in time represented by seconds. A workload is a collection of jobs. Our scheduling algorithm is presented in Algorithm 1. We formulate the task scheduling problem using a bounded fractional knapsack, where the items in the knapsack are sorted in descending order according to to their priority. This approach has been proven optimal for an unbounded knapsack [7]. For each workload we apply a priority to each job in the workload during each scheduling cycle. The prioritization equation we use was originally proposed by Ward et.al. [22] to calculate delay windows for scheduling for relaxed backfill and is shown in Equation 1. P_j is the priority for each job waiting in the queue. We use this formulation because of the simplicity and effectiveness of overall workload throughput. We use exponents for weighting terms with common scheduling parameters such as total job runtime, time spent waiting (waittime), and jobs size (tasks). The priority of each job can increase, or decrease at a determined rate in relation to other jobs through the use of the average workload values. Overall, the algorithm runs in time $O(nlogn)$.

$$P_j = \left(\frac{Tasks}{avg.Tasks}\right)^{\alpha}\left(waittime\right)^{\beta}\left(\frac{runtime}{avg.runtime}\right)^{\gamma} \quad (1)$$

Algorithm 1 FRACKNAPSACK(a,n,C)

```
Require: $C$, capacity of the knapsack, $n$, the number of tasks,
    $a$, array of tasks of size $n$
Ensure: A sorted list of jobs by priority $(M,t)$.
 1: for $i = 1$ to $n$ do
 2:     $ratio[i] = a[i].priority/a[i].w$
 3:     $sort(a, ratio)$
 4:     $weight = 0, i = 1$
 5:     for ($i \leq n$ && $weight$ && $C$) do
 6:         if $weight + a[i].w \leq C$ then
 7:             $select(a[i].w)$
 8:             $weight = weight + a[i].w$
 9:         else
10:             $fraction = (C - weight/a[i].w)$
11:             $select(fraction, a[i].id)$
12:         end if
13:     end for
14: end for
```

3.1. Scheduling Assumptions

Our scheduler uses a single queue and a greedy fractional knapsack, which is sorted on individual job priorities. The priorities are dynamic in that every scheduling cycle each job gets an updated priority based on its size, wait time and expected runtime. We make the following assumptions in our task scheduler:

- Single Queue
- Backfill; a lower priority job may be run if it does not delay the start time of the higher priority job

- No Preemption; jobs will not be stopped once they have started running

- Malleable jobs; jobs can be split into independent tasks by the scheduler.

3.2. Modeling High Level Objectives

One of the simplest methods for determining the effectiveness of a particular scheduling strategy is minimizing the per job expansion factor $E_f = \frac{waittime+runtime}{runtime}$. For each job, maintaining an E_f value of 1.0 is optimal as it reflects that the job accrued no wait time but that if it did, the waiting time was proportional to it's running time. Thus, for a short job, say 10 seconds, a wait time of 30 seconds will have an E_f of 4.0, (40/10). For a long 10 hour job that waits for 30 hours, will share the 4.0 E_f. Our work assumes that wait times relative to the job duration can be more flexible, and a large expansion factor is permissible for short jobs in many cases, whereas for long or large jobs this should be avoided.

We define high level objectives from the perspective of longer term performance metrics. For example, an objective may be to decrease waiting time for larger than average jobs, but only if smaller jobs aren't delayed by some percentage of their runtime. Often a system administrator would like to allow jobs with certain properties to gain increased priority, but without having to introduce static job priorities, queues, dedicated resources or preemption. Our scheduler is designed to have the administrator set a group of objectives and have the scheduler optimize for as many of the objectives without manual intervention. An additional goal of our approach is in determining a good estimate as to when the system has stabilized on the schedule that optimizes the expected objective.

3.3. Scheduler Performance Metrics

In our experiments, we measure the performance of our scheduler using two performance metrics, 1.) The total time to complete a fixed workload, where (WRT) is workload response time; 2.) The average job expansion factor, E_f, per job class. In our evaluation in section 4., we are interested in answering the following regarding overall workload performance; namely, How does the system size affect the time to complete all of the jobs? How do various α, β and γ values affect the workload runtime and finally, which class of jobs complete first and how does this affect the waiting time or jobs in different classes?

Cost is not just minimizing job wait time. We wish to minimize wait time for a specific class of jobs. We therefore consider overall schedule cost as the average E_f for jobs in four classes based on their average size, (number of cores) and duration(actual runtime length): Large and Long (Class A), Large and Short (Class B), Small and Long (Class C) and Small and Short(Class D). We are also interested in the total workload completion time and utilization, but the goal of the scheduler is to guarantee that at least one job class is optimized.

For each schedule we calculate the average E_f for each of these four classes and include workload runtime as a total cost vector $Cost(A,B,C,D,RT)$. We ran our experiments for all α, β,γ between -2 and 2, incrementing by 0.1. The total number of combinations follows this formula: $total = ((max-(min))/incr)^3$, where $min = -2$, $max = 2$ and $incr = 0.1$.

Algorithm 2 COSTEVAL(a,n,C)

```
Require: C, capacity of the knapsack, n, the number of tasks,
    a, array of tasks of size n
Ensure: A cost vector containing $E_f$ for each job class
    $Cost(A,B,C,D,WRT)$.
 1: $i = 0$
 2: for $\alpha = -2 \leq 2; \alpha$+=0.1 do
 3:     for $\beta = -2 \leq 2; \beta$+=0.1 do
 4:         for $\gamma = -2 \leq 2; \gamma$+=0.1 do
 5:             Cost[i++] = $schedule(\alpha,\beta,\gamma)$
                {For Optimization}
 6:             if $cost[i] < bestSoFar$ then
 7:                 $bestSoFar = cost[i]$
 8:             end if
 9:         end for
10:     end for
11: end for
```

This problem can be formulated as a multiple objective Knapsack problem. We wish to minimize E_f for a specific jobs class as well as quantify total workload response time. The task placement problem can be formulated as a multiple objective knapsack problem [20] [10] [13] [12]. This seems like a reasonable approach because it provides a simple and efficient algorithm for the simultaneous optimal solutions for mutually conflicting objectives, such as performance and reliability. In Huang et al. Huang applies the multi-objective optimization problem to reliability and introduces the "Intelligent Interactive Multiple Objective Optimization" IIMOM model which uses an Artificial Neural Network to determine optimal model parameter vectors, which are used as input, and designer preferences over representative samples along the Pareto frontier as output.

We formulate the task scheduling problem as a multiple objective optimization problem. Through multiple objective optimization, a variety of solutions may be viable. Typically these form a Pareto front, which provides a boundary for optimal performance for a given workload. In Section 4., we address the problem of finding the best balance of potentially conflicting objectives, such as minimizing the average E_f for a single job class while maintaining high utilization or low workload response time.

4. EXPERIMENTAL EVALUATION

As stated previously, we develop a greedy continuous knapsack algorithm that calculates a priority for each waiting job during each scheduling cycle. Job priorities are determined by wait time, runtime(the duration of the job), number of tasks (the size) a fixed ”share” value between 0 and 1, where a share value of 1.0 means that if a job is at the top of the queue all of it’s tasks ”may” be run if they all fit. The experiments are run using a Linux cluster with 8 nodes, each containing 2 Intel quad core Nehalem processors giving 64 total processors over a 1Gb ethernet connection.

4.1. Map Reduce Workload

We generated workloads from the NASA AIRS satellite data. Each day of AIRS data consists of 240 granules in an HDF formatted file, each granule has a radiance array of 135x90x2378 (28,892,700 elements) stored in compressed latitude and longitude arrays. The size of each granule is approximately 60MB. Over 10 years, the total AIRS dataset is approximately 55TB. The sequential processing times for 10 years of data on a single Nehalem system takes approximately 3 months. It takes 6 and a half days to process those data using a 14 node Nehalem cluster using an embarrassingly parallel method. Since the daily data processing is independent from other days. The embarrassingly parallel method executes each day consisting of 240 granules on a single node. Thus, 14 days can be executed on 14 nodes concurrently.

The MapReduce functions takes set of input key/value pairs, and produces a set of output key/value pairs. These functions are divided in two phases: Map and Reduce. The Map function is written by users and distributed among all the nodes using the Hadoop [4] framework. It takes an input pair to perform some partial processing of the data located on a node, and produces a set of intermediate key/value pairs. In the case of our gridding approach, we performed parts of the addition of gridded data in the mapping function and send the partially added data to the reduce function. For example, the Earth is mapped into a lat-lon grid of 360x360 grid cells. This step will take each observation in the granule data and map it to corresponding lat$x$lon grid cell based on the observationÕs lat and lon geo-location data. The map function opens the granule it has been assigned, reads its contents and performs a data projection algorithm assigning and summing the radiance values of each spectral channel into a specific grid cell location. The output key is the combination of a grid cell and the date; the value is the local total average Brightness Temperature (i.e. BT is a transform of radiance) and a count of how many additions (stores in an array) it has performed for each grid cell. The reduce function receives an intermediate key (the combination of grid cell and date) and a set of values for that key (output from the map) and it performs the final data projection function by dividing the total summation value by the count (i.e. total number of additions). The processing jobs are developed using a MapReduce workflow system [14] which includes the MapReduce constructs, customized data format HDF packages to work with the Hadoop MapReduce framework, parallel DataLoader to load AIRS level 1B HDF files to Hadoop HDFS. In Figure 1 we compare the embarrassing parallel allocation of daily data, what we call the ”regular” method, to a given processor with the Hadoop MapReduce Workflow method. Results show a 35% improvement in total processing time. The average processing time of 9 years of AIRS level 1B - daily processing 45TB (788400 Map tasks) on 14 nodes is estimated based on average processing time for a day of data.

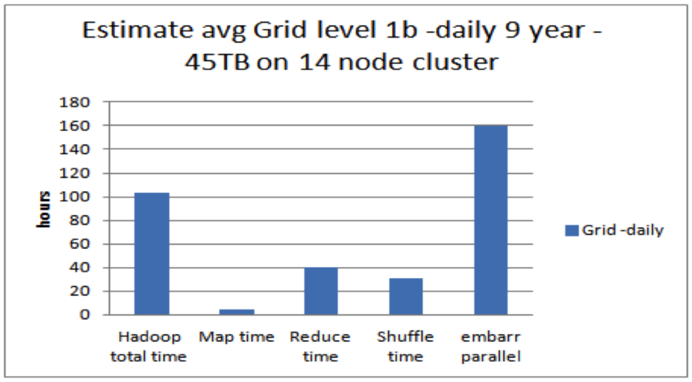

Figure 1. comparision of AIRS processing by Hadoop MapReduce with the regular method

4.2. Simulation

We have developed a flexible scheduling simulator, in C, that implements our proposed approach from Algorithms 1 and 2. The goals of our first experiment were to quantify and better understand the effect of various α, β and γ parameters evaluation on overall workload execution time. For the simulation we used a 220 task workflow from the AIRS Hadoop job mix. The workload contained the following fields for each job:

$job_i(id, tasks, runtime, size)$

The results are shown in Figure 2. The results illustrate the effects of the prioritization parameters. The left axis for each graph is the total system utilization, in this case a system size of 500 cores was used. The left y axis is the total number of jobs running and the x axis is the time of the simulation in seconds. In the top left graph, where $\alpha = -1$, the jobs with fewer tasks receive a larger weight and increase in priority more quickly, as expected. The graph in the upper left, with β and γ at 1, we see the longer jobs are run first because of the emphasis of job duration and wait time. The graph in the lower left emphasizes larger jobs, and we can see that the policy is enforced, you can determine the size of the jobs by

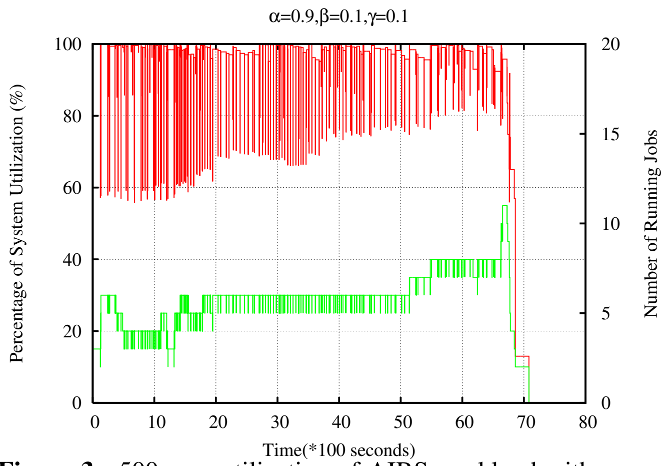

Figure 3. 500 core utilization of AIRS workload with $\alpha = 0.9$, $\beta = 0.1$ and $\gamma = 0.1$

the distance between the number of jobs and the utilization curves, a larger gap means the larger the jobs. In the lower right corner of 2 we see the longer jobs interspersed throughout the run, with the shorter and smaller jobs running first.

Figure 3 Shows a good balance between utilization and jobs, We can see a higher rate of running jobs per second, near 6, in contrast to Figure 2. This leads to almost halving of the workload response time.

In addition to evaluating the total workload completion times, we wanted to see the effect of the proportional share value on workload response time (RT). We see in Figure 4 that if we increase the proportion that we take from each of the highest priority jobs we get a very small decrease in workload response time. These results are from a smaller workload of 98 jobs taken from the same AIRS dataset. The RAND value means that each of the 98 jobs get a random proportional value from 0.1 to 1.0.

Using the fractional knapsack approach we measure the effect of system size on the workload runtime. Figure 5 shows that as we double the system system size from 36 to 576 we get a inverse power curve of $94504x^{-0.78}$. The x axis is the total number of cores and the y axis is the total workload response time. These measurements were done using the simulator and a random proportional share value for each job. The interesting point is that the response time is not decreased proportionally by doubling the system size, there are diminishing returns.

4.3. HPC Workload Simulation

In addition to running a scientific workload in, we also have used our simulator to run HPC workload traces from the Parallel Workload Archive [11]. We selected, arbitrarily, the log from a large BlueGene/P installed at Argonne National Laboratory which has a total of 163,840 cores. The file used as

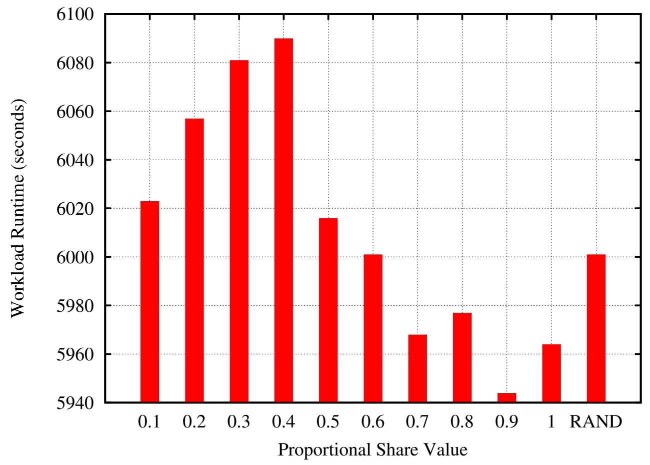

Figure 4. Workload Response Time as a function of proportional share for a 98 job AIRS workload

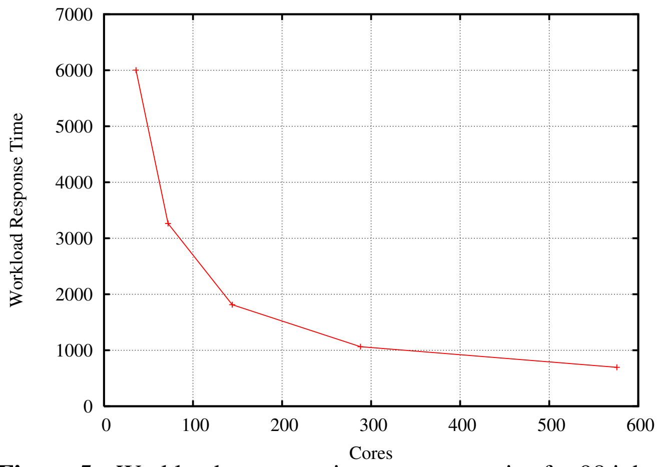

Figure 5. Workload response time vs. system size for 98 job AIRS workload

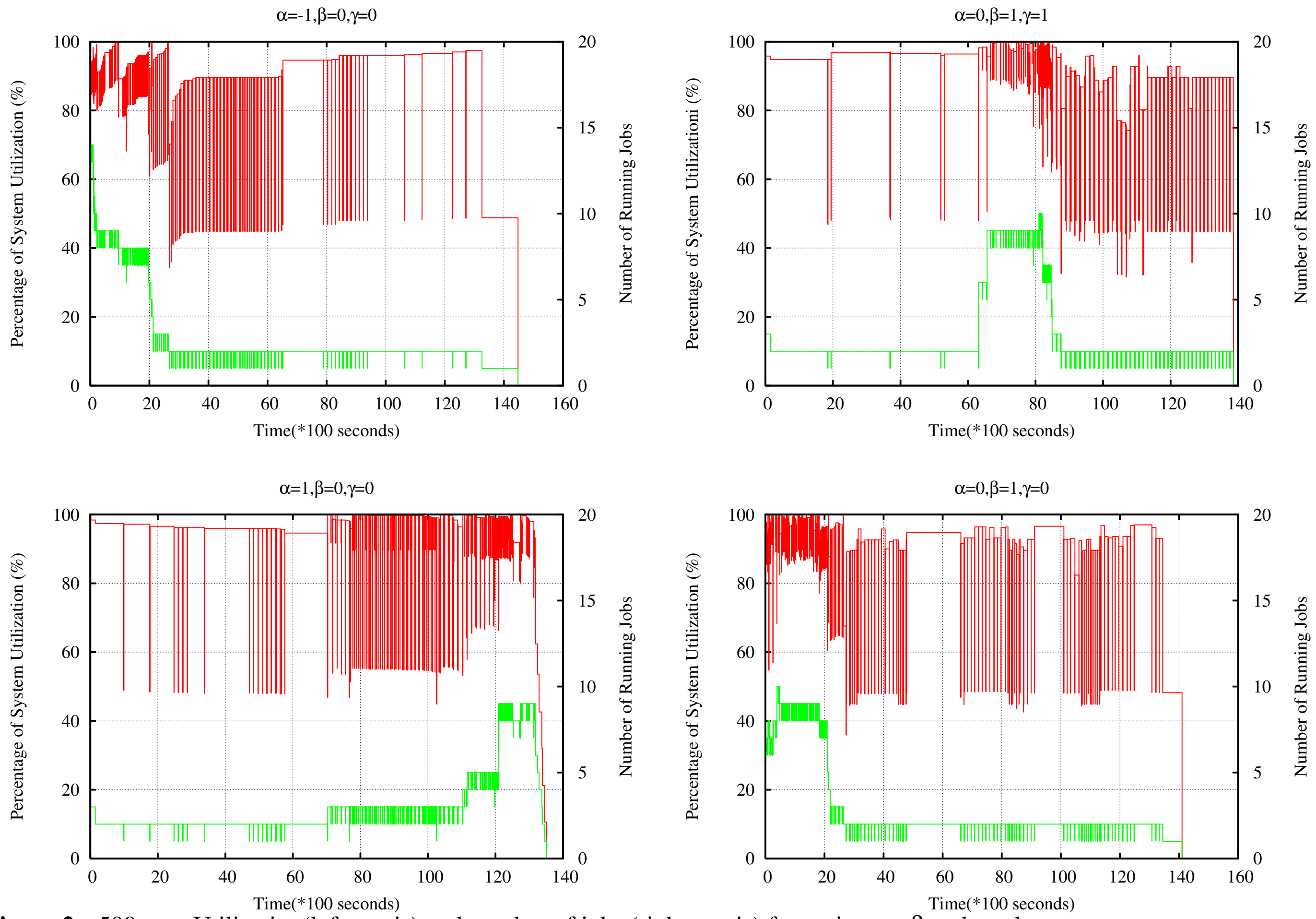

Figure 2. 500 core Utilization(left y axis) and number of jobs (right y axis) for various α, β and γ values

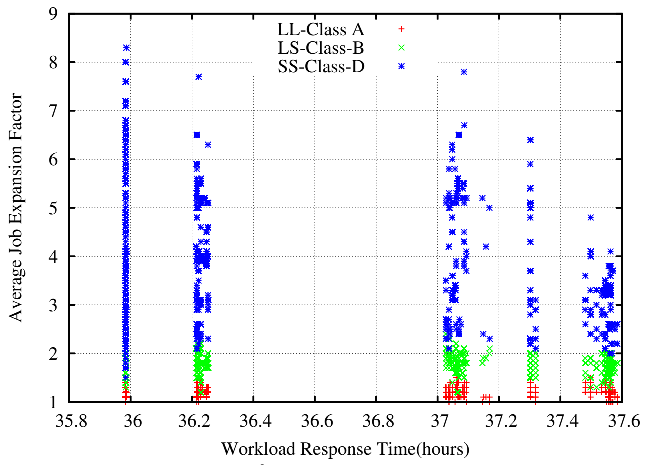

Figure 6. Exhaustive α, β, γ search space for 50 job ANL-Intrepid workload

input to our simulator was *ANL-Intrepid-2009-1.swf*. We used the following fields: job arrival time, number of cores, job duration. All of the waiting times were set to zero in our tests and we only ran the first 50 jobs in the log, on 65536 cores. This number was chosen because it represented the largest job. Performing an exhaustive search for the complete parameter space takes 30 seconds to generate all schedules. We will work to optimize the search process, since it is an embarrassingly parallel problem, and is easily divided into independent tasks based on the parameter range. Figure 6 presents all of the parameter combinations for the three classes of jobs in the ANL workload. Note there were zero jobs that fit into Class-C, jobs that were below the average size and had above average runtimes. The x axis is the time to complete the entire workload for a given parameter set, and the y axis is the average E_f for all of the jobs in a particular class. Thus, the scheduler would select the values at the bottom left of the figure if the administrator wishes to minimize the workload response time, but the penalty for the Class-D jobs will be an E_f of 1.75. It seems that for this particular workload, the Class-A jobs will typically not have long wait times as their E_f values hover just above 1.0.

5. CONCLUSIONS AND FUTURE WORK

We have extended our previous work in the field of dynamic priority scheduling by introducing a scheduler that performs an exhaustive parameter search that helps achieve high level scheduling objectives. Our approach demonstrates flexibility towards a variety of job scheduling strategies using dynamic prioritization that utilizes adjustable weights within a single queue for jobs with many independent tasks utilizing a proportional fairness metric. We developed a simulator and provided experimental results that demonstrate the nature of the scheduling algorithm for a realistic data intensive workflow and from an HPC workload. Additionally, we have shown some of the promising behavior of the scheduling algorithm with respect to proportional share, system size, and the automatic tuning of the parameters α,β,γ. Further work will investigate additional and larger workloads from the Parallel Workloads Archive. We also intend to investigate heuristic approaches when the scheduling cycles becomes too short to perform an exhaustive search of the parameter space. We also look forward to integrating this scheduling strategy into a resource management framework to make better runtime task allocation decisions on heterogeneous HPC environments, such as optimizing for power usage or reliability.

ACKNOWLEDGMENT

This work is supported in part by the NSF IUCRC Center for Hybrid Multicore Productivity Research and the UMBC department of Computer Science and Electrical Engineering.

REFERENCES

[1] A. Benoit, L. Marchal, J.-F. Pineau, Y. Robert, and F. Vivien. Scheduling concurrent bag-of-tasks applications on heterogeneous platforms. *Computers, IEEE Transactions on*, 59(2):202 –217, feb. 2010.

[2] R. Bertin, A. Legrand, and C. Touati. Toward a fully decentralized algorithm for multiple bag-of-tasks application scheduling on grids. In *Grid Computing, 2008 9th IEEE/ACM International Conference on*, pages 118 –125, 29 2008-oct. 1 2008.

[3] Veeravalli Bharadwaj, Thomas G. Robertazzi, and Debasish Ghose. *Scheduling Divisible Loads in Parallel and Distributed Systems*. IEEE Computer Society Press, Los Alamitos, CA, USA, 1996.

[4] D. Borthakur. The hadoop distributed file system: Architecture and design. *Hadoop Project Website*, 11:21, 2007.

[5] Steve J. Chapin, Walfredo Cirne, Dror G. Feitelson, James Patton Jones, Scott T. Leutenegger, Uwe Schwiegelshohn, Warren Smith, and David Talby. Benchmarks and standards for the evaluation of parallel job schedulers. In *Proceedings of the Job Scheduling Strategies for Parallel Processing*, IPPS/SPDP '99/JSSPP '99, pages 67–90, London, UK, UK, 1999. Springer-Verlag.

[6] David Chapman, Milton Halem, Phuong Nguyen, and Jeff Avery. Noise reduction in gridded airs brightness temprature grids using the modis obscov algorithm. In *Proceedings of the IEEE IGARS Meeting*, IGARS 2012, Munich, Germany, 2012. IEEE.

[7] G. B. Dantzig. Discrete-Variable Extremum Problems. *Operations Research*, 5:266–288, 1957.

[8] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008.

[9] Maciej Drozdowski. *Scheduling for Parallel Processing*. Springer Publishing Company, Incorporated, 1st edition, 2009.

[10] Thomas Erlebach, Hans Kellerer, and Ulrich Pferschy. Approximating multiobjective knapsack problems. *Manage. Sci.*, 48:1603–1612, December 2002.

[11] Dror Feitelson. The parallel workloads archive. http://www.cs.huji.ac.il/labs/parallel/workload/, January 2013.

[12] Hong-Zhong Huang, Zhigang Tian, and Ming Zuo. Intelligent interactive multiobjective optimization of system reliability. In Gregory Levitin, editor, *Computational Intelligence in Reliability Engineering*, volume 39 of *Studies in Computational Intelligence*, pages 215–236. Springer Berlin / Heidelberg, 2007.

[13] Rajeev Kumar and Nilanjan Banerjee. Analysis of a multiobjective evolutionary algorithm on the 0-1 knapsack problem. *Theor. Comput. Sci.*, 358:104–120, July 2006.

[14] Phuong Nguyen and Milton Halem. A mapreduce workflow system for architecting scientific data intensive applications. In *Proceedings of the 2nd International Workshop on Software Engineering for Cloud Computing*, SECLOUD '11, pages 57–63, New York, NY, USA, 2011. ACM.

[15] Phuong Nguyen, Tyler A. Simon, and Milton Halem. A hybrid scheduling algorithm for data intensive workloads in a map reduce environment. In *Proceedings of the 5th IEEE/ACM International Conference on Utility and Cloud Computing*. IEEE/ACM, 2012.

[16] I. Raicu, I.T. Foster, and Yong Zhao. Many-task computing for grids and supercomputers. In *Many-Task Computing on Grids and Supercomputers, 2008. MTAGS 2008. Workshop on*, pages 1 –11, nov. 2008.

[17] Ioan Raicu, Zhao Zhang, Mike Wilde, Ian Foster, Pete Beckman, Kamil Iskra, and Ben Clifford. Toward loosely coupled programming on petascale systems. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC '08, pages 22:1–22:12, Piscataway, NJ, USA, 2008. IEEE Press.

[18] Ioan Raicu, Yong Zhao, Catalin Dumitrescu, Ian Foster, and Mike Wilde. Falkon: a fast and light-weight task execution framework. In *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, SC '07, pages 43:1–43:12, New York, NY, USA, 2007. ACM.

[19] Thomas Sandholm and Kevin Lai. Dynamic proportional share scheduling in hadoop. In *Proceedings of the 15th international conference on Job scheduling strategies for parallel processing*, JSSPP'10, pages 110–131, Berlin, Heidelberg, 2010. Springer-Verlag.

[20] C.L. Valenzuela. A simple evolutionary algorithm for multi-objective optimization (seamo). In *Proceedings of the 2002 Congress on Evolutionary Computation. (CEC.)*, volume 1, pages 717 –722, may 2002.

[21] Daniel C. Vanderster and Nikitas J. Dimopoulos. Sensitivity analysis of knapsack-based task scheduling on the grid,Ó to appear. In *in Proceedings of ICS 2006 - The 20th International Conference on Supercomputing*, 2006.

[22] William A. Ward, Jr., Carrie L. Mahood, and John E. West. Scheduling jobs on parallel systems using a relaxed backfill strategy. In *Revised Papers from the 8th International Workshop on Job Scheduling Strategies for Parallel Processing*, JSSPP '02, pages 88–102, London, UK, UK, 2002. Springer-Verlag.