Albatross: Dynamic Scaling for Heterogeneous Cycle Scavenging HTCondor Clusters

Rohan Mathur[1] and Tadas Aleksonis[2]

Abstract— **Executing computationally-intensive tasks in a distributed manner has become more popular in recent years. HTCondor, software developed by the University of Wisconsin to make the job submission and execution process easier across compute clusters, supports both dedicated compute nodes and idle desktop workstations for computation. In a cluster that solely consists of desktop workstations which only accept HTCondor jobs when they are idle, if the workstations are at maximum utilization, the jobs in the queue will not be executed until a workstation goes idle, leading to starvation. The jobs in the queue are never executed as there are no available machines accepting HTCondor jobs. We introduce Albatross, a monitoring process which enables HTCondor to dynamically add and remove nodes from the system, based on current job throughput and user-based preferences. We show that by integrating HTCondor with Albatross, we can parametrize jobs to a highly-customizable degree and observe an improvement in reliable throughput correlated to the scale of capital.**

I. INTRODUCTION

Executing computationally-expensive jobs efficiently has long been a subject of interest. The category of computationally intensive jobs spans across everything from physics simulations to big data processing with frameworks similar to MapReduce. As such, any advancements in the efficiency of executing these jobs could potentially affect a multitude of different fields.

To speed up the execution of multiple computationally-intensive jobs, architects have turned to executing these jobs in parallel across a wide number of machines. Whether the machines are dedicated compute nodes whose sole purpose is to efficiently execute whichever jobs are dedicated towards it or more dynamic systems, that utilize idle time on devices to execute jobs, the process of managing and distributing these jobs across multiple compute nodes is often complex and time-consuming. Because of this, many administrators of these compute clusters often turn to a distributed software framework for submitting and executing jobs. By offloading the task of distributing and scheduling jobs across multiple heterogeneous nodes to a software-based solution, the usability of these platforms improves, and opens up room for improvement in efficiently distributing jobs across compute nodes.

One such framework used for distributing jobs is known as HTCondor. Primarily developed by the HTCondor team at University of Wisconsin-Madison, the HTCondor project has enabled users to easily distribute jobs across multiple compute nodes. It features support for managing workloads on both dedicated resources for computation (rack-mounted clusters), along with desktop machines where jobs will execute depending on several factors (cycle scavenging). Like several other solutions, it provides a job queueing mechanism, scheduling policy, priority scheme, resource monitoring, and resource management [1]. Users can easily submit their serial or parallel jobs to HTCondor, where it will place them into a queue, and choose when and where to run the jobs based upon a policy. Once running, HTCondor monitors the progress of each job, and when completed, informs the user of the task's completion.

HTCondor has several advantages over existing batch queueing systems. One of the unique aspects of the framework is its ability to efficiently utilize either both dedicated compute nodes and idle desktop workstations, or utilize each in separate instances. For example, HTCondor can be specif-

*This work was not supported by any organization

[1]R. Mathur is an undergraduate in the Department of Electrical and Computer Engineering at the University of Illinois at Urbana-Champaign `rmathur2 at illinois.edu`

[2]T. Aleksonis is an undergraduate in the Department of Electrical and Computer Engineering at the University of Illinois at Urbana-Champaign `alekson2 at illinois.edu`

ically configured on desktop workstations to only accept jobs when the mouse and keyboard have been idle for a certain duration. Once the system starts being used again and no longer marked as available for computation, a unique checkpointing mechanism exists which is able to migrate a job to a different machine that otherwise would be idle, with little time spent not doing meaningful work on the task. HTCondor does not require a shared file system across machines, and instead can either route all I/O requests to the specific machine where the input/output/intermediary files exist, or transfer the job's files on behalf of the user.

Although HTCondor is flexible enough to work with both dedicated compute machines, alongside cycle scavenging from other machines, it comes with its downsides. For example, in a cluster that solely consists of desktop workstations which only accept HTCondor jobs when they are idle, if the workstations are at maximum utilization, the jobs in the queue will not be executed until a workstation goes idle, leading to starvation. The jobs in the queue are never executed as there are no available machines accepting HTCondor jobs.

Because of this, we propose a new addition to HTCondor, Albatross, a framework for detecting starvation cases, and remedying it by dynamically scaling up/down dedicated compute nodes in an HTCondor pool. We plan to provide the following contributions:

- Investigation of the starvation that occurs within an HTCondor cluster consisting of desktop workstations that execute jobs when idle
- Investigation into spin-up and auto-provisioning dedicated compute nodes, along with spinning-down after the task is completed
- Design and implementation of a middleman monitoring service that detects starvation in an HTCondor cluster, and dynamically scales clusters if starvation is detected
- Design and implementation of an extensible and parameterizable framework for controlling the behavior of scaling
- Improved performance numbers as a result of the additional dedicated compute nodes

II. BACKGROUND AND RELATED WORK

A. HTCondor Overview

HTCondor was developed with the goal to make executing distributed jobs easier for the user, and make cluster management of machines more efficient and effective by optimizing the planning and scheduling of jobs across these machines. By optimally placing jobs on machines, HTCondor can efficiently utilize a fixed-size cluster of either homogenous or heterogenous systems. Additionally, it can utilize desktop workstations that have gone idle whose CPU cycles would be wasted anyway.

B. Allocating Resources within HTCondor

HTCondor's mechanism for allocating resources is what makes HTCondor's scheduling algorithm effective. Job submission is simplified with a concept of ClassAds. ClassAds are similar to an advertisement in a local newspaper, where a service advertises its characteristics, and customers peruse the newspaper to buy certain services. ClassAds are made by each machine in the cluster. The machines in the HTCondor pool advertise their attributes, such as available memory, CPU type and speed, virtual memory size, current load average, along with other static and dynamic properties [2]. Additionally, in the ClassAd, each machine specifies under what conditions it is willing to run an HTCondor job, and what type of job it would prefer. This information is used when choosing where each job should get distributed to. When a user submits a job to the HTCondor cluster, they also specify a ClassAd for the job, which specifies the minimum requirements for the job (i.e. minimum memory size, minimum disk space) or also nice-to-haves, such as machines with fast floating point performance if the job requires that [2].

HTCondor uses both the ClassAds from jobs in the queue and ClassAds from the machines available to do work in order to find a mapping between them. It ensures that all requirements specified in each of the ClassAds are satisfied. If these requirements are impossible to satisfy (either there are no machines available to do work that meet all of the jobs in the queues requirements, or there are no jobs in the pipeline) HTCondor essentially spins and waits for a machine to be available to pick up a job. ClassAds allow HTCondor to adapt to nearly any allocation policy, and adapt to a planning-based approach while incorporating grid resources.

C. Existing Work

Being a dynamic, distributed scheduler, it is intuitive to develop a model for HTCondor in which it can dynamically scale with a new stream of jobs. Often, when configuring these systems that require distributed scheduling, the amount of available nodes remains fixed, as the addition of multiple new nodes is an expensive and infrequent occurrence. HTCondor distinguishes itself as not simply a distributed system or a cluster, but distinctly, a computing grid. One of the distinct features about grid computing is its emphasis on CPU scavenging, whereas a grid computing framework can be installed on everyday computers and mobile devices, such that spare cycles on a CPU can be lent to the framework for job loads.

CPU scavenging is a versatile resource that can render any set of regular computers into a distributed cluster; other such projects have taken this idea and applied it to their own use cases. Stanford's Folding@home [6] is a popular research project where users can install software locally on their computer in order to allocate resources towards protein folding. People with no understanding of protein folding can dedicate free CPU cycles towards overarching research projects, without seeing their own workload being throttled. Folding@home's framework does not satisfy our needs, as the number of free computers is undefined, as it is a framework that was popularized through word of mouth, and it is only dedicated towards protein folding. We would like to see a more rigid, yet general, system.

Another popular technology that is used for distributed computations is SLURM [4], an open source resource manager aimed at scheduling for high-performance clusters, such as the Blue Waters Supercomputer at University of Illinois, Urbana-Champaign. This technology is highly scalable and has very high performance capabilities, as exemplified by its extensive use in TOP500 supercomputers. While a heavily utilized scheduler, it is primarily focused on large-scale clusters and supercomputers, so its utilization towards desktop workstations for CPU scavenging is not a priority, such as industrial examples of prioritizing data scheduling. Nevertheless, Slurm is an extensible alternative that is also adept at high-throughput computing scheduling, on par with HTCondor.

A final example of a competing example is Berkeley's BOINC (Berkeley Open Infrastructure for Network Computing) [5]. It performs many of the same types of high-throughput tasks that HTCondor utilizes, but with several key differences that may not be deemed suitable for the types of tasks which we want to add on to. BOINC specifically has a centralized architecture which distributes tasks to nodes, whereas HTCondor has the distribution less centralized; this lack of emphasis on any particular node to perform schedule-based decision making lends towards a more practical, desktop workstation-based system. Furthermore, BOINC requires specialized infrastructure to operate, which limits the overall types of grids we are to expect in dealing with in regards to our perceived solution, whereas HTCondor lends itself towards moreso desktop-like resource allocation.

III. PROBLEM AND MOTIVATION

A. Problem

The flexibility of HTCondor's planning and scheduling policies (ClassAd system) allow it to function well with dedicated compute clusters, desktop workstations that have gone idle, or a mix of both. Dedicated compute clusters offer the greatest power to performance ratios, always readily available to execute HTCondor jobs, and never starve the job queue. Desktop workstations do not adhere to these provisions. If the desktops are consistently in use, none are available to execute HTCondor jobs. In this situation, the task queue is often not emptied out; tasks build up, and no nodes are available to execute the jobs. This situation also makes a lot of sense - in a system where users who are using the desktop workstations to submit jobs that are also a part of the HTCondor pool for computation, if more people are using the workstations, there will likely be more jobs submitted for execution.

B. *Our Solution*

To fix this dilemma, we propose the idea of Albatross, that monitors HTCondor's efficiency. If Albatross detects that tasks are not being executed due to a lack of compute resources, it can dynamically scale the compute cluster to now include dedicated compute nodes, which can then begin work executing jobs from the queue. After the peak load has subsided, and more machines are available/idle, Albatross has the ability to spin down the added dedicated compute nodes if desired.

By being able to dynamically spin up and down instances in cases where the system has no available compute clusters for long periods of time, we essentially create a compute cluster that scales when it has to if the desktop computers are under heavy load, and utilizes resources in the cluster when they are not being utilized, therefore potentially achieving optimal behavior with no additional user intervention. This is desirable for cluster managers looking for a low-cost solution, while still providing some guarantees about eventually getting jobs executed.

IV. APPROACH AND SYSTEM DESIGN

A. *Target Environment*

Before introducing our approach, we briefly describe the environment in which we expect Albatross to be deployed and used. We focus mainly on the situation where all compute nodes are either unavailable (due to either having no resources in the pool, or all resources occupied by other jobs or users). In this situation, the Matchmaker is simply looping waiting for resources to be available and not servicing the queue. This leads to any jobs submitted not making any progress, and the overall system coming to a halt, as long as the computational resources are being used up.

B. *Architecture*

We propose a new flexible framework that allows for this situation to make meaningful progress towards job completion. Within HTCondor, there exists a framework that matches a job's ClassAds to a machine, known as the Matchmaker. This Matchmaker has no fixed schema, and instead, supports properties being met as either being true, false, or undefined, if the schema is not found in the corresponding ClassAd. Since the matchmaker is the part of the system that matches jobs to machines, it is the last common point between what machines are available vs what jobs we have to run. Because of this focal point, it is the logical choice to detect starvation of the queue.

C. *Detecting Stalling*

Our system needs to be able to detect stalls in the queue. We accomplish this by placing a timer in the Matchmaker, which is reset upon dispatching a job to a node. If this timer hits a certain amount of time (meaning that a job has not been dispatched within a predefined amount of time), along with the fact that there are jobs in the queue, this means the system is being starved. A design decision was made here to not also require checking what resources were currently available. This is because we believe if the Matchmaker does not believe that a valid match exists (which covers more than just the machines not accepting new jobs), then we want to include this in our stalling as well and scale up here.

D. *Choosing a Cloud Hosted Provider*

The next design decision we had to make was to identify a good cloud-hosted provider to use in our elastically scaling system. Our cloud provider requirements include quickly spinning up/down instances of nodes at varying sizes, at generally low cost. Additionally, the VMs must be relatively easy to integrate within the HTCondor cluster, which requires easily accessing the network configuration for the newly created VM, along with running some sort of setup script within the VMs. There are two main cloud providers that meet these requirements. Specifically, Google's Compute Engine, and Amazon's AWS EC2 are the industry-leading choices for spinning up and down virtual machines en masse. Amazon Web Services' EC2 service allows for easily spinning up and down virtual machines, at a generally low cost, along with Google's Compute Engine. The two are mostly comparable, in terms of both pricing structure and features offered. For this project, we ended up selecting Amazon's AWS EC2 service, due to our familiarity with its APIs and services, though the choice could absolutely go either way.

EC2 has APIs that allow programmers to easily spin up and down the compute pool.

E. Spin-Up and Spin-Down Process

To scale up and down the virtual machines, as well as connecting it to our HTCondor pool, we would have to first determine what type of instance we want to use. For this aspect of our system, we can choose to either start a new VM for every job in the pipeline, start one VM that can run all of the pending jobs in the queue (or maybe just the higher priority jobs in the queue), and many other potential options. Since this is highly varied, we choose to make this parameterizable (described in later paragraphs) by the user utilizing the flexible schema of the ClassAd framework. After spinning up the machine, configuring HTCondor to join the pool (requiring coordination with the pool admin) is required, along with configuring the network and firewall [7]. We abstract this out with a setup script that will take care of the headless install of HTCondor on this new machine, along with configuration with the HTCondor pool admin. After the job is completed and other resources become available for HTCondor to use, we can choose to either checkpoint the currently running job on the VM and move it over to a local machine in our cluster if we prefer low cost to performance, or allow the job to finish executing and then transfer the results back to a central server for the user to view.

F. Parameterization and User Control

A lot of these aspects of the system are highly user preference dependent. For example, how aggressively the user wants to scale up their system in the case of starvation is a direct cost vs benefit tradeoff. Additionally, how long the user wants to keep the VM up and running after resources become available is highly dependent on the user's budget as well. We abstract this decision making process down to a single floating point score, between 0 and 1, known as the capital parameter, since it directly correlates to the cost the user will incur while using our system. Based on this parameter, we will make and document several of our decision making process on whether to scale up/scale down instances, how long to keep the instances around for, and how many instances to start. The user should understand what this parameter means first, and then play around with values until they settle on a good value that gives them the tradeoff between performance and cost that they desire. Additionally, the user can leverage the ClassAd framework to specify whether a job should run on AWS if it needs to or not. For example, when a task has a large dataset associated with it, it might not be worth it for the user to incur the bandwidth cost of transferring the dataset to the cloud-hosted AWS instance, and thus they can opt out when submitting the job of allowing this job to execute on AWS.

The mapping of these user defined parameters to actual AWS actions was where the majority of our research went into. Because of complexity, we decided upon one static mapping between capital parameter values to actions taken in AWS. For example, capital parameters with a value less than 0.25 would always spin up a t2.micro instance of an AWS EC2 on-demand node, since a t2.micro instance costs less than other instance types available. Another factor that we had to choose a static mapping for was the duration each newly added node stays up for. We chose to use a mapping for this parameter as well, up to one hour, after which, we spin down and check for whether the stall is still present in the system. If a job is still executing on the AWS node at this hour mark, we checkpoint the job and transfer it back to the central manager to be dispatched to a different node, whether it be a newly spun up AWS node or one of the desktop workstations present in the cluster that has come back online after being idle for the required time period. It is additionally possible to have both of these parameters be influenced by the job itself. For example, the type of instance that starts up can be manipulated such that it would match the minimum requirements of the job in the pipeline, which is not always a guarantee. This can take place easily as the Albatross framework is placed within the HTCondor Matchmaker, which has access to the job's requirements, along with access to the AWS instances.

V. RESULTS

In testing Albatross, we sought to emulate an environment that would emulate a standard group

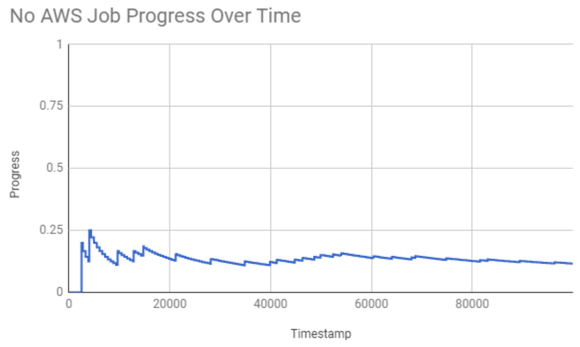

Fig. 1. Job progress over time for a cluster with no Albatross.

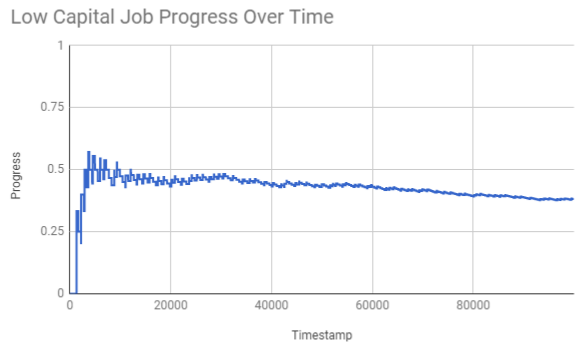

Fig. 2. Job progress over time for a cluster with Albatross set to have low capital (0.25).

of ten desktop users who perform various tasks throughout the day using these workstations. As such, we set up a system of 11 virtual machines, with one as a negotiator node and 10 worker nodes (akin to that of a desktop). As such, when HTCondor would be installed on these VMs, it would operate in the background as other processes also run on these virtual machines, programmatically.. We tested Albatross by generating random jobs to dispatch to the worker nodes every 5 minutes, each of which vary in length and computational intensity. As jobs would enter and leave, we measured the overall progress made by the nodes across 100,000 seconds (27.78 hours). Testing nodes would also enter and leave the worker pool, to emulate people leaving desktops and returning. Spinning up AWS instances has a minor cost in time, as well as transferring data to and from the AWS nodes. We establish a notion of progress by counting the total number of completed jobs divided by the total number of jobs dispatched to the system.

In Figure 1, without the Albatross framework enabled, we see what the default behavior of HTCondor looks like. Over time, we see that the progress eventually converges to approximately 1/8th of the jobs being completed. This indicates that if a job was dispatched to the cluster during the time period, the probability of it being completed by the time our trial was over was approximately 1/8th. We see an initial lag in the graph, indicated by the small flat portion of the graph before it begins to make progress. This initial lag is due to the counter set up within the Matchmaker, that waits a predetermined time period before spinning up an AWS node. During this time period, along with the time spent executing the first job, we see no progress made, as all of the desktop workstations are being utilized by the user, and none are available for usable work. As they eventually begin coming in and out of being idle, more jobs are able to complete. The saw-toothed behavior seen in the graph is indicative of the metric we used to keep track of job progress. Every time the progress percentage increased, it means that a new job was completed by the cluster, and thus it incremented the total number of jobs completed amount. The tail of the saw toothed wave is when the cluster is executing a new job, and thus the progress metric is not increasing. We additionally see that at one point between 40,000 seconds to 60,000 seconds, there is a slight increase in the slope of the progress line. This indicates that at this point, more desktop workstation nodes were chosen to stop being used, opening them up for computation of jobs in the pipeline. The slope quickly goes back down, as the desktop workstations begin being used again. The overall time to complete a job was averaged at 261.15 seconds, and the standard deviation was measured to be 170.822 seconds. In the graphs with Albatross enabled, two aspects should change if the framework is working correctly. Firstly, the saw toothed nature of the graph should become compressed, indicating that jobs are completing faster, and less time is spent between a job's completion times. Secondly, the average time to complete a job, along with the standard deviation measured should both decrease.

Once we enabled the Albatross framework, we

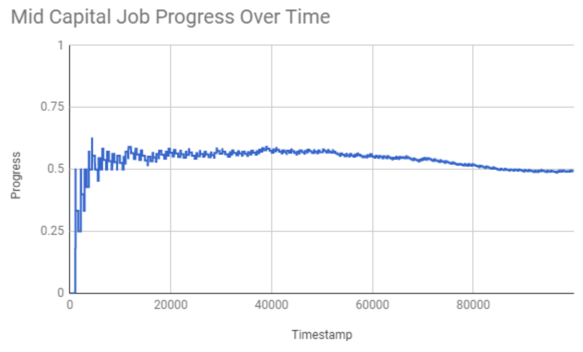

Fig. 3. Job progress over time for a cluster with Albatross set to have mid capital (0.50).

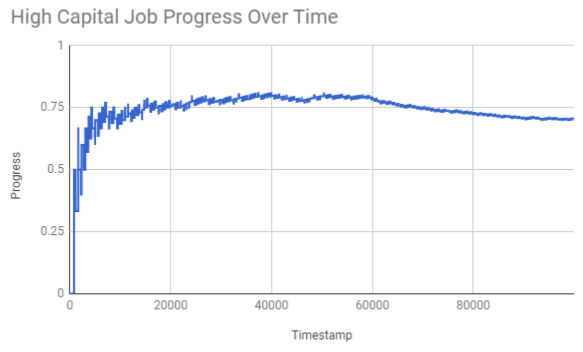

Fig. 4. Job progress over time for a cluster with Albatross set to have high capital (0.75).

saw much of what we expected. With low capital (set to around 0.25), as seen in Figure 2, we found that after 27 hours, our progress converged to being a steadily decreasing graph at around 3/8ths. This means that at that time period, 3/8ths of the jobs in total were completed of the jobs dispatched. In the graph itself, it holds with what we predicted. The saw toothed waves in the graph are greatly compressed from the previous graph, meaning that the jobs are completing much faster. Additionally, the overall time to complete a job decreased to 72.21 seconds, and the standard deviation decreased to 47.63 seconds.

As we increased the capital parameter, we see that this trend holds consistent. The mid capital run (set to 0.5), graphed in Figure 3, yielded an average job completion time of 53.64 seconds with a standard deviation of 35.23 seconds.

The high capital run (set to 0.75), graphed in Figure 4, yielded an average job runtime of 34.20 seconds, with a standard deviation of 20.96 seconds.

Going from the low capital run to the mid capital run, we see less improvement tan the run that jumped from mid capital to high capital. Although the computational resources are approximately doubled, this could have been due to the mismatch in the instance type that was spun up and the job's requirements (the jobs were not able to fully take advantage of the additional computation power, as the doubling of the computationally available resources was not enough to make a significant impact).

VI. FUTURE WORK

Plenty of unexplored opportunities remain in the wake of Albatross. The current implementation of Albatross is operational, but there are avenues in which the scaling logic can be optimized, to further tuned to the needs of a system with specific opinions of capital expenditure. Currently, Albatross spins up specific instances of AWS nodes based off of capital, but this can be further improved upon by implementing variable AWS instances based off of the waiting workload. By spinning up specific instances of nodes that match the computational needs of the job queue, higher throughput can be achieved.

Another point of interest is batching sets of jobs to AWS instances. If multiple jobs in a queue permit AWS execution, then we can specifically tune the AWS instance to the needs of the jobs and enable higher throughput. It is important to note that all of these future possibilities involve increasing reliance on the type of AWS node that is spun up, which may incur higher costs to the user. However, capital in this case may be contrary to what a user wants, when they may be more interested in jobs being finished than a fidelity to the limit imposed by the capital parameter. As such, utilizing work-adaptive capital, to allow for flexibility in the logic of AWS node creation. More complicated logic would be required to implement flexible capital in order to keep it close to its original intent, but models based on variance or general throughput, rather than strict cost, may achieve this sort of goal.

Finally, as we are students of University of Illinois, we have EWS to our disposal. EWS has a large existing infrastructure. People continuously use their own personal computers, but may run into poor system performance from a variety of different processes running on their own workstations. Allowing to pipe some of those tasks to EWS to run in the background as a passive performance boost would be a desirable and beneficial opportunity for students to use. To further entertain this notion of passive performance enhancement, a system would have to be able to take general processes and dispatch them to EWS; obviously not all processes would qualify, as many processes require computer specific properties, i.e. filesystem. However, decision logic could be implemented to monitor processes to determine if the cost of spinning up an EWS (or AWS) instance is worth the cost of transferring a process to such a node.

VII. CONCLUSIONS

In this report, we propose Albatross, a monitoring process that detects starvation within HTCondor's Matchmaking service. The workload across an entire grid that HTCondor may monitor may vary with respects to the types of computing grids it manages, but in the specific instances of wanting to utilize available CPU cycles in a primarily desktop-oriented, yet heterogeneous grid, it is important to consider the cost-benefit analysis of dynamically spinning up new nodes in a network in order to increase throughput. Primarily, if a job must be delivered with a high priority, but a current grid is busy, then a potential solution is to add extra nodes to the network to finish the task.

As such, there must be a parameterized model in which jobs can be weighted on their priority, the user's willingness to spend actual currency on an immediate solution, and a method of determining whether it is worthwhile to wait several more moments instead of adding new nodes to the system. Conversely, the throughput of a job may be highly increased if many jobs meet Albatross' parametric logic and many different jobs can share a single instance of an AWS server. We introduce the notion of capital, which places a weight on the user's willingness to present new nodes into the system at their own fiscal expense, and we identify the necessary opportunities to consider when we introduce a system as robust and dynamic as AWS into HTCondor's scheduling logic.

We see that as capital increases, the throughput proportionally scales. While the work performed by a mid-tier capital did not have as desirable or noticeable effect as we may have hoped for, this can be attributed more to setting-tuning than actual faults within Albatross. We observe that as the capital increases, the variance of time-to-completion for different tasks tightens, suggesting a stable and reliable manner of increasing throughput for the trade of cost.

REFERENCES

[1] *HTCondor - What is HTCondor?* [Online]. Available: $http://research.cs.wisc.edu/htcondor/description.html$. [Accessed: 25-Oct-2017].

[2] *2.3 Matchmaking with ClassAds.* [Online]. Available: $https://research.cs.wisc.edu/htcondor/manual/latest/2\_3Matchmaking\_with.html$. [Accessed: 25-Oct-2017].

[3] D. Thain, T. Tannenbaum, and M. Livny, *Distributed Computing in Practice: The Condor Experience.* Madison, WI: 2004, pp. 14-18.

[4] Slurm Workload Manager, *Slurm Workload Manager.* [Online]. Available: $https://slurm.schedmd.com/$. [Accessed: 25-Oct-2017].

[5] BOINC, *BOINC.* [Online]. Available: $https://boinc.berkeley.edu/$. [Accessed: 25-Oct-2017].

[6] Front Page, *Folding@home.* [Online]. Available: $http://folding.stanford.edu/$. [Accessed: 25-Oct-2017].

[7] T. Miller, *There are many clouds like it, but this one is mine (condor_annex).* Madison, WI: 2016.