A Lightweight Model for Right-Sizing Master-Worker Applications

Nathaniel Kremer-Herman, Benjamin Tovar, and Douglas Thain
{nkremerh,btovar,dthain}@nd.edu
University of Notre Dame

***Abstract*—When running a parallel application at scale, a resource provisioning policy should minimize over-commitment (idle resources) and under-commitment (resource contention). However, users seldom know the quantity of resources to appropriately execute their application. Even with such knowledge, over- and under-commitment of resources may still occur because the application does not run in isolation. It shares resources such as network and filesystems.**

We formally define the capacity of a parallel application as the quantity of resources that may effectively be provisioned for the best execution time in an environment. We present a model to compute an estimate of the capacity of master-worker applications as they run based on execution and data-transfer times. We demonstrate this model with two bioinformatics workflows, a machine learning application, and one synthetic application. Our results show the model correctly tracks the known value of capacity in scaling, dynamic task behavior, and with improvements in task throughput.

I. Introduction

Researchers today rely on clusters, clouds, and grids to analyze and collect data on a large scale. However, it is difficult to know the resource requirements of an application at scale. Decisions about just how large the application should be scaled often have to be made by the researcher. This can lead to cases of requesting too few resources to get their work done in a timely manner or asking for too many resources and blocking *other* researchers from getting their work done.

At the University of Notre Dame, researchers like high energy physicists and biologists have shared computing resources available to run their experiments at scale. When executing their applications, we often find our users under-provisioning or over-provisioning their work by orders of magnitude. For example, users request resources on ten cores for an application that should be using thousands. This prevents them from getting their research done as quickly as it should. Users have also requested a thousand cores when only tens could be used effectively. This is a problem for the cluster since other researchers have to wait in the queue while their colleague is using about a thousand completely idle cores. In this case, the productivity of the entire campus can grind to a halt without intervention from a system administrator.

In principle, users could run their application multiple times with varying resources in order to discover an appropriate resource provisioning for their application on the given system. This is not useful in cases where the data from the application does not need to be processed more than once. It is especially detrimental when the user is charged for computation such as infrastructure-as-a-service platforms. Having to re-run the application to find an appropriate resource allocation can quickly rack up cost. It also slows down the rate of their research. It would be preferable to run the parallel application only *once* to discover an appropriate number of resources and dynamically provision them throughout the application's lifetime.

We present a method for dynamically calculating the number of computational nodes which can be effectively utilized by a master-worker parallel application. This model is called the *capacity* of the application. This method provides the benefit that the application does not have to be rerun, and approximations of the true value of capacity are easily obtained as tasks are executed. This model prevents waste on idle resources from over-provisioning which can in turn save users' money and allocation time in the case of infrastructure-as-a-service platforms. It also increases throughput if an application experiences initial under-provisioning.

We evaluate the capacity model by executing four applications on an active, campus-scale high performance computing cluster. One application is synthetic and is designed to demonstrate the potential differences between anticipated and realized capacity. We also test two different bioinformatics workflows based on the BWA [1] and HECIL [2] genomic data analysis tools, respectively. The final application we use to evaluate the model is a machine learning model search and hyperparameter optimization application called SHADHO [3]. We provide results showing the accuracy with which a user can estimate capacity *a priori* if they know the expected execution and I/O times of each tasks which was the case of the synthetic application. We also demonstrate the capability of an execution engine to utilize the capacity model to dynamically right-size the number of resources available to an application throughout its lifetime using BWA, HECIL, and SHADHO, minimizing the idle time of the workflow management system, scaling up and down the number of resources available to the application, and preventing waste of idle machines.

We also present a web-based troubleshooting tool for researchers to diagnose common resource provisioning issues in their applications with the goal of providing a transparent and informative interface to help users understand the behavior of the application at run time. The capacity model, along with basic performance metrics, provide the basis for simple visualizations which make resource issues readily apparent. The capacity model and performance metrics also inform a

SC18, November 11-16, 2018, Dallas, Texas, USA
978-1-5386-8384-2/18/$31.00 ©2018 IEEE

troubleshooting recommendation system which provides the user with actionable steps to address potential problems.

In summary, we contribute a model for calculating the *capacity* of master-worker applications, an evaluation of this model in a live master-worker framework with four applications, and a web-based visualization to provide researchers the benefits of the model when not using the live implementation.

II. Background and Related Work

In a master-worker framework, a master process serves as a centralized controller of worker nodes and is responsible for coordinating workers and feeding them tasks. Workers are processes scheduled to cluster nodes which persist so long as they receive work from the master. The master submits work to be done, called tasks, along with any necessary input data for that work. After executing their current task, a worker will provide the master with any specified output. The scalability of this application framework comes from the number of workers the master can sustain given resource availability. With fewer workers, the magnitude of concurrent work the master can achieve is decreased, but the work will still get done. The reverse is true of being able to request more workers; concurrency will increase. However, there is a limit to how many workers an application can handle.

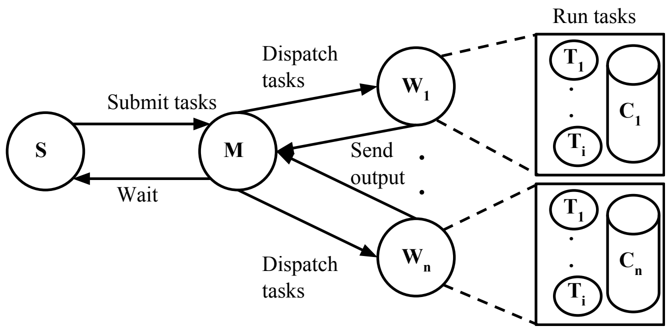


Fig. 1. Master-worker architecture. S is the task submitter, M is the master, and W nodes are workers. Each worker may handle multiple tasks which all share a common data cache C in each worker.

The degree of parallelism of a master-worker application is constrained both logically (i.e. tasks depend on each other) and practically: it is not always feasible or possible to provide the resources necessary to achieve maximum concurrency. In fact, it is sometimes detrimental for the execution of the application to over-provision it in an attempt to increase its throughput. This is because a distributed system's architecture creates two intrinsic bottlenecks. First, the execution time of each task may be limited by the hardware available. If execution time is slow, the master will spend much of its time polling the running tasks, waiting for output. The other bottleneck is I/O time. If transmitting the input and retrieving the output of a task takes a long amount of time, the master will be stuck sending and receiving data instead of sending out new tasks.

There are many common types of applications which can be implemented in a master-worker framework. They include parameter sweeps [4], bulk synchronous parallel [5] applications, bag-of-tasks [6], and scientific workflows [7]. Each of these share the capability to expand or reduce its current executing workload in response to changing resource availability. We present a model called the *capacity* of a parallel application which formalizes an upper bound on the application's ability to expand to more resources. We present this model through the lens of a master-worker framework. The capacity model seeks to dynamically calculate an appropriate number of resources the master process can handle given any system or application bottlenecks the researcher may be experiencing.

In order to enforce the capacity model, the number of resources (i.e. persistent workers) connected to the centralized master process must be scalable. The Work/Exchange model and load balancing studied in [8] provides historical context for the problem of compute node load in parallel applications. While [8] presents a similar problem, we are concerned with minimizing the idle time of the master process in a master-worker framework rather than the idle time of a group of statically defined compute nodes. Automatically scaling to an ideal number of resources has been studied in [9], [10], [11], [12]. However, these works incorporate cost-efficiency and adherence to Quality of Service Agreements as goals in their analysis. We are most concerned with the presentation of a model for measuring capacity at user-level as a means of scaling a parallel application. Load and quality of service impacts are external from the user space in which our model operates. Similarly, [13] presents a resource management tool for Service-Level Objectives which are concerned with policy decisions for each user at a system level whereas the capacity model is concerned only with the current user's perspective. Another similar work is [14] which presents a similar motivation as the capacity model with the addition of adherence to Service-Level Agreements, however the work in [14] makes decisions at the system level as with [13]. We present a model which operates only with the knowledge of the user and reacts to system availability rather than making decisions at a higher level to guarantee some level of system availability for all users. In this regard, [15], [16], [17] focus on the high cost of establishing resources which will go either unutilized or under-utilized during a cloud service's lifetime. Though the resources in question vary between the works, each seeks to eliminate acquiring under-used resources. We follow a similar line of thought but for master-worker parallel applications rather than the system. We demonstrate this with master-worker applications which make use of cluster machines. Using the capacity model presented, we provide an appropriate number of resources to provide an application at varying stages of its execution.

Along with mechanisms for correct provisioning, it is useful to have a way of modeling an application's behavior. Concerning predictive modeling and near real-time analysis of behavior, [18], [19] demonstrate the need to properly schedule workloads based on previous application behavior and the state of the executing system. A common thread among these works is to increase system utilization and decrease waste. Similarly, we work to eliminate idle time in the master process as well as decrease idle time of connected workers. Although our work focuses on the application-level instead of a system-level approach, we note that a positive consequence of correctly adapting resources to an application's capacity is its potential to negate over-provisioning of a distributed system.

Our definition of capacity builds directly upon an idea first presented in [20] which we have now derived on first principles. Unlike the previously described works, we contribute a model which adapts the concurrency of an application to the current system performance exclusively at the user-level. The presented capacity model finds its roots in Gustafson-Barsis' Law (specifically the scaled speedup model) [21] and initial work on parallel computation speedup [22]. In contrast to [21], we demonstrate the limit to the scalability of a master-worker application due to the bottleneck of the master process. Chiefly, we consider throughput as the bottleneck of a parallel application's capacity, so we consider the factors which drive throughput: execution time and I/O time. The model we present is a formal and more complete consideration of the problem of resource provisioning. In particular, we first note we must weight the most recently completed task heavier than the rest. This better informs us of the application's capacity at the current state of execution as opposed to an average value computed across the whole of the application's lifetime. Without this contribution, we would be missing potential to seize more resources or scale down if the application's current state indicates that is needed. We then implement the ability to scale regardless of cores being requested by the user whereas the previous model assumes each task requires only a single core. We must also track the master's own think time in the model. This think time is the time needed by the master process to complete bookkeeping, manage resources, and execute any other sub-routines. These contributions allow for a wider range of useful implementation.

III. Capacity Model

For each parallel application, there exists some ideal minimum number of resources which, when provisioned, give the application its fastest execution time. If the application is given fewer than the ideal number of resources, it will not reach its maximum parallelism and thus run slower. If it is provisioned more than the ideal resources, the application may run slower due to overhead incurred for managing those resources. At best, providing more resources will neither speed up nor slow down the application. This will, however, be wasting those over-provisioned resources.

Figure 2 models the impact upon application runtime from poor resource provisioning. The application modeled in the figure consists of 100 independent, homogeneous tasks. Each task has a 100 second turnaround time. We assume in this model that there is a 1 second cost for managing each additional computational resource. The best run time for the entire application is thus 200 seconds (in the case that all tasks are run concurrently for 100 seconds with 100 seconds of cumulative resource management time). This is shown when the scale of resources reaches 100. Since we assume

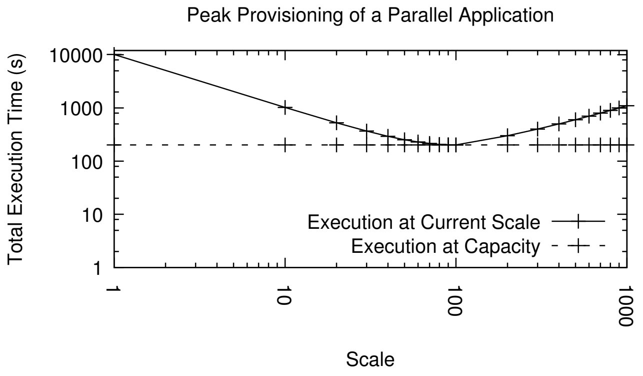


Fig. 2. Effect of resource provisioning on execution time. When executing a parallel application, there is a scale of resources which will provide the fastest execution time. Provisioning fewer or greater resources will slow down the application.

there is some cost associated with managing resources, the total execution time will increase as additional resources are added after the first 100. This may not be the case for all applications, but at best adding more than the appropriate number of resources will neither increase nor decrease the execution time of the application.

We can conclude from Figure 2 that there is some appropriate scale for each parallel application. To provision above or below that scale will lead to a slower execution time. It would be beneficial for the user if they were able to derive some appropriate scale of resources they should request for their parallel application. We say that the application is right-sized when that appropriate scale is reached.

Now consider a master process that delivers tasks to be executed by worker processes. We can make the following observation with regards to the throughput (tasks completed per second) of the master process:

Assuming the master process can only transfer data (input and output) for a task at a time, all the workers have identical processing and networking capabilities, and all the tasks have identical execution time t_e and identical transfer time t_{io}, then the maximum throughput of the master process T_m is bounded from above as $T_m \leq 1/t_{io}$. The observation easily follows since the system cannot process a single task faster than t_{io}. Note that this maximum throughput is independent of the time it takes to execute a task t_e. The execution time per task t_e comes into play with the upper bound on throughput of a single worker T_w, which is bounded from above as $T_w \leq 1/(t_{io}+t_e)$. If the master has C available workers, their throughput upper bound, under the conditions of our observation, is CT_w.

Let C be the number of workers the master needs such that it is never idle. From the previous discussion, $CT_w = T_m$, and $C = 1 + t_e/t_{io}$. We call C the capacity of the system[1].

[1]With think time t_z per task at the master, the bound on throughput becomes $1/(t_{io} + t_z)$, and the capacity is $C = (t_e + t_{io})/(t_z + t_{io})$. Here we have to be careful. If we limit the number of workers to integral values, we may find the only way to not have idle workers is to have none of them (e.g. $t_{io} + t_z > t_e$, with the floor operator giving 0). We should then execute the application locally since it cannot handle greater scale.

Since the master process deals with tasks in a sequential manner, using more workers than C will not increase the throughput of the system. The capacity C is in fact the number of workers at which a speedup curve converges [23].

A. Dynamic Capacity Model

The basic model above makes assumptions that are hard to meet in practice. For example, not all tasks are identical, and not all computing nodes have similar resources. To deal with these issues, we extend the previous computation to derive an estimate of the capacity as follows:

Let $C_i = 1 + t_{e_i}/t_{io_i}$ be the capacity computed if all tasks *were identical* to the i^{th} finished task. Using an exponential moving average, a parameter $\alpha \in (0, 1)$ is used to weight previous completed tasks against the most recently completed one. We assume that the most recently completed task will be more indicative of the application's current behavior. In our testing, the value $\alpha = 0.05$ performed well in practice. With $C_0 = 0$, for $i > 0$ and $0 < \alpha < 1$, we recursively define:

$$C_i = \alpha(1 + t_{e_i}/t_{io_i}) + (1 - \alpha)C_{i-1}$$

Using this dynamic model, we gain better insight into the application's resource needs. Seldom are tasks identical for an entire workload, so weighting the i^{th} finished task greater than the cumulative capacity of the previous workload allows us to better adjust when workload changes occur. For example, consider a master-worker application which has two categories of tasks: tasks of type A and tasks of type B. There are an equal number of both types. Assume that both task types have the same I/O time, but type A tasks run half as long as type B tasks. Let us also assume the workload submits all the type A tasks first then submits the B tasks. There will be a point in the workload when capacity will increase because B tasks run twice as long as A tasks. Capacity will essentially double. A more naive model [20] which does not weight the most recently completed task heavier can take awhile to adjust to the sudden change in capacity. There may be a long lag between actual capacity and realized worker acquisition. In our dynamic model, the added weight α allows our application to realize the capacity change between A and B faster. This in turn will reduce that lag and scale the number of workers quicker if the resources are available. In essence, we present a model which follows Gustafson-Barsis' Law [21] to find the best speedup using an exponential moving average.

B. Cached Inputs

Thus far we have assumed that each task has independent data transfer with duration t_{io}. However, tasks may share common inputs which only need to be sent once to each worker where they are cached. We use t_c to refer to the time it takes to transfer these common inputs and $t_{io} - t_c$ as the time to transfer the remaining inputs and outputs. Since the task description and exit status are transferred between master and workers, we have $t_{io} > t_c$.

In a steady state where shared files are cached at current workers, the capacity is therefore $C = 1 + t_e/(t_{io} - tc)$. Suppose that we already have m workers available, and n tasks remain to be dispatched. When does it become advantageous to add a new worker? Certainly if $m >= C$ (already at capacity) or $m \geq n$ (not enough work to fill the extra worker), an additional worker would be wasteful. We explore here the remaining case, $m < C$, and $m < n$.

If there are n tasks that remain to be dispatched, it will take at least nt_{io} time to process all these tasks. Conversely, it will take at least $nt_e/m + t_{io}$ time for the workers to finish the remaining tasks. Since we assumed that the master was running under capacity, we have that $nt_{io} < nt_e/m + t_{io}$. Thus, r new workers are advantageous when:

$$\underbrace{nt_{io} + rt_c}_{\text{transfer } n \text{ tasks and } r \text{ caches}} < \underbrace{nt_e/(m + r) + t_{io}}_{\text{execute and transfer } n \text{ tasks on } m+r \text{ workers}}$$

Caching increases the overall capacity, but it increases the cost of initializing a new worker. These two compromises come into play when a master-worker application has few tasks left to be processed.

C. Applicability to Generic Workloads and Limitations

Scientific workflows are often broken up into sets of similar tasks which all run at relatively similar times in the lifetime of the workflow. This is advantageous for the capacity model as we can accurately determine when the workflow is advancing from one set of similar tasks to another. However, it is difficult to gauge capacity in applications which have no way to identify similar tasks. In such cases, our model reaches a capacity which is averaged between the many different tasks. This can lead to some idle resources if there are very wide gaps (e.g. orders of magnitude) between the execution to I/O ratios of different tasks. Our model would have to catch up with radically changing ratios which will prevent converging on a stable number until later in the application's lifetime.

The evaluation of our work consists primarily of scientific workflows, which are commonly executed using a master-worker framework, because our users most commonly execute this type of application. However, we believe this capacity model is applicable to other styles of master-worker applications. Bag-of-tasks applications typically do not have a mechanism to submit waves of similar tasks, so it can be expected that capacity could vary greatly at the beginning of the application's lifetime but smooth to an average value once a sufficient number of different kinds of tasks have completed. If all the tasks in the bag are identical (or similar), we can expect capacity to be a stable number throughout computation.

Bulk synchronous parallel applications have an easier behavior to model since they are more uniform. These applications consist of groups of computation tasks and synchronization tasks. These groups are called supersteps. If we exclude the sync supersteps from affecting capacity, then we would expect capacity to adjust according to the execution and I/O time spent during each superstep of computation. However, if we *do* consider sync supersteps having an effect on capacity, then capacity could drop during those supersteps and we could release unused resources while the remaining nodes finish synchronizing. During the next computation superstep our need for more resources would increase, and the application's capacity would increase to reflect that.

A parameter sweep application is similar to bag-of-tasks when it comes to capacity because if structured it can submit a grouped wave of similar search spaces, but this structure is not guaranteed (i.e. depth-first search of a parameter space). If the master process is structured to submit groups of similar parameter search spaces, we will see capacity change with the execution and I/O time spent between the tasks of each group. If, however, the parameter sweep does not have a wave-of-tasks behavior then capacity could vary greatly until it eventually smooths to an average value.

Extending the capacity model further, we can envision how it could apply to other execution frameworks such as Apache Spark [24], MapReduce [25], and MPI [26]. In Spark, we are concerned with creating resilient distributed datasets (RDDs) and performing either transformations or actions on those RDDs. Spark forms a directed acyclic graph of the RDDs to establish lineage. This lineage graph informs the application the order in which to create data and how to re-create it if necessary. This directed acyclic graph (DAG) structure is similar to a typical scientific workflow in that we are given the data to be produced, the dependencies for that produced data, and the command to create it (a transformation or an action). One way to model capacity for a Spark application could be the time to complete the transformation or action (execution time) and the time spent retrieving dependent RDDs and storing created RDDs (I/O time). These two time statistics fit the current form of the capacity model.

Applying capacity to MapReduce is more difficult. We can consider the map and reduce steps as two separate waves of similar tasks in an application. This is especially useful if the application loops on multiple stages of mapping and reducing. If we apply the capacity model directly, we could expect to see a capacity for mapping and a capacity for reducing. The model could be implemented in the back end system to scale the number of resources (i.e. cores) assigned to a data partition as the application executes.

Extending the model to MPI is conceptually a bit of a challenge since MPI applications are not necessarily elastically scalable. Since MPI is designed to allow for various kinds of communication interfaces (master-worker, scatter-gather, point-to-point, broadcast, etc.), it is difficult for one model to cover all. In the case of a master node being used to facilitate communication, the capacity model would function similarly to its current implementation. Some added overhead in the communications would be needed to send measurements of execution time to the master node. The master would need to track its time spent handling messages and I/O from the worker nodes. Other configurations of MPI would be more difficult to model. Since many of the configurations are decentralized, each node would need to track its time spent computing and its time spent passing messages and data. To compute capacity with the current assumptions in the model, the nodes would need to aggregate their execution and I/O time to a dedicated process which could spawn more processes or kill excess, idle processes. This may not be ideal for all applications since it introduces a centralized mechanism in an inherently decentralized system. The capacity model would need to be modified to fit decentralized communication configurations.

IV. Implementation

We implemented our capacity model in the Work Queue master-worker execution engine [27]. The model can be implemented in any framework where task execution time (t_e) and task I/O time (t_{io}) are readily available. Our users run Work Queue applications to scale up their research by breaking up their analysis pipeline into smaller tasks which can be executed concurrently. Work Queue is designed to scale from O(10) up to O(10,000) cores. The largest scale application using Work Queue has successfully scaled up to approximately 25,000 cores.

The Work Queue master is a process which the user executes on the their machine or a cluster's head node. It is responsible for giving workers tasks to run as well as any input files a task may require. A worker is a process which runs on a batch system and claims resources on a machine for a user's work. Worker processes persist as long as they are given work to do, and each worker has a local cache. These workers receive input files and executables to run the task if they do not have them in their cache. If the task is completed successfully, the worker waits for the master to acknowledge its success then transfers the output of the task.

The master receives a task report from a worker once a task has finished executing. If the task was completed successfully, the master uses that task report for determining capacity. This task report contains the execution time (t_e) and I/O time (t_{io}) for that task along with many other performance metrics. These metrics are measured at the worker process and are used in the calculation of capacity as shown in the model. The master also keeps track of its think time during tasks which is added to the task report. We use these times to calculate the capacity of the application, weighting the most recently completed task most heavily as defined in our model. If the task is not successful, the task report for that failed task is not included in the capacity calculation.

The master determines the capacity using the stats retrieved from successful task reports. The most recent task report's execution time, I/O time, and master think time are weighted more heavily than the rest of the application's history because we assume that task will be more indicative of the application's current behavior. After the capacity is calculated, the master submits it (and other metrics) to a catalog server which a utility called Work Queue Factory can access.

In order to request and maintain the appropriate number of workers for the application, we use a program called Work Queue Factory to dynamically provision according to the master's capacity calculation. Work Queue Factory is an application which retrieves periodic information about a master and uses this information to submit new workers for that master. This is useful in cases where workers have idled out in an earlier section of the application but are needed at the moment. It is also helpful in scaling down by not replacing idled-out workers if the master does not need any more. The factory decides how many workers to request by calculating the minimum among the result of the capacity model, the number of tasks currently submitted by the application, and a user-defined upper bound. If the result of this calculation is less than the number of workers currently connected to the Work Queue master, the factory does not request more workers.

V. Evaluation

To evaluate the accuracy of our capacity model we used the BWA and HECIL scientific workflows, the SHADHO Work Queue application, and a synthetic workflow which continuously submits tasks as benchmarks. We tested each application on an active campus-scale, heterogeneous system containing approximately 25,000 cores. We utilized the HTCondor batch system [28] as the underlying resource manager for all workflows. HTCondor is a centralized batch scheduler which placed our workers in a queue. Batch jobs are dispatched from the queue onto available compute nodes. Each worker had a uniform number of cores depending on which application it executed. For the synthetic workflow, BWA, and HECIL, each worker used a single core. Each worker for SHADHO used 16 cores.

A. Synthetic Application

The synthetic application we designed to easily implement an expected maximum capacity and test if the workflow approaches the anticipated value. The application runs uniform tasks using the same command, the same amount of user-specified I/O, and the same user-specified task execution time throughout its execution. Every task shares a common input file which is used to generate unique output files. Each task has approximately one second of I/O. The task then sleeps for a specified amount of time to simulate added execution. The unique output file is retrieved at the end of the task. This uniformity leads to a near-constant capacity for the application's lifetime, the only non-constant factors being out of our control in the test (i.e. hardware or operating system speed).

In Figure 3, we validate that the capacity of our master-worker application shares a direct relation to the ratio of execution and I/O times. As the ratio increases (execution time becomes greater than I/O time), the measured capacity follows. The model prevents a value below 1 since we must have at least one worker to make progress in master-worker applications. This also demonstrates that an application which has task execution times much smaller than its I/O time will be better off running locally if feasible, as seen once the expected capacity (t_e/t_{io}) becomes less than 1.

We also show the ability of the model to scale up and down using Work Queue Factory after an initial number of workers was requested. The factory can take as input a configuration document which may be modified during runtime. If the configuration file is changed, the factory will notice the changes during its 30 second main loop interval. We initially configured the factory request either too few or too many workers rather than using the capacity model. Once the factory acquired the number of workers it was configured to instantiate, we then changed its configuration to consider the capacity of the application. In the case of an under-provisioned start, more workers were requested. The factory let workers disconnect in the over-provisioned start. Each test had the same estimated capacity of approximately 115 and ran as many tasks as it could complete within fifteen minutes. We requested fifty workers for the scaling up test and 150 workers for the scaling down test.

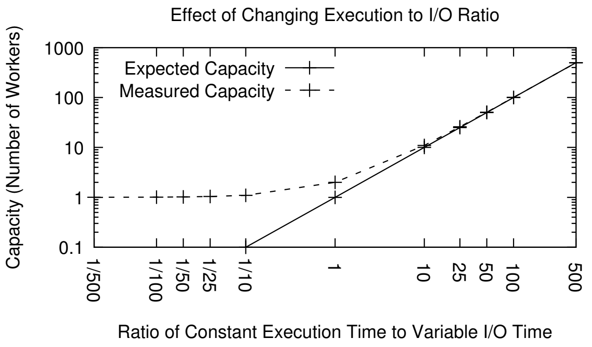
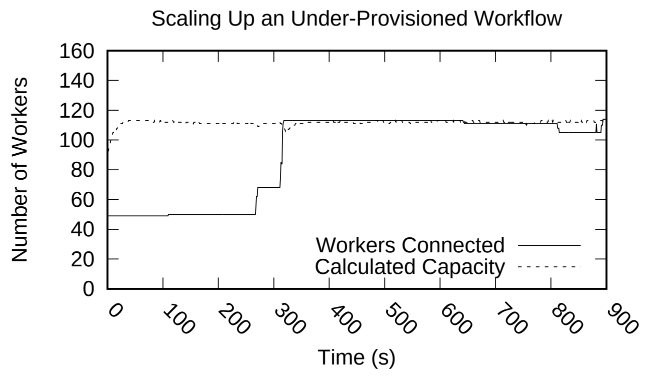


Fig. 3. Changing I/O time test. We demonstrate how capacity changes with the execution to I/O ratio per task from 1/500x to 500x. Since the tasks are homogeneous, we can predict the anticipated capacity. We present the anticipated capacity and actual measurements. Once the ratio drops below 1 it becomes obvious that it is best to run the workload locally instead of on distributed resources.

Figure 4 demonstrates the capability of the the factory to rapidly deploy an appropriate number of workers based on the capacity model's calculation. In the upward scaling test, we see the number of workers rapidly increase from the initial pool of 50 up to approximately 115. The opposite is true of the downward scaling test. We see that our initial over-provisioning does not last the application's lifetime because the factory does not replace the workers which idled out. Even though the number of workers changed, and thus the number of available resources was altered, the workflow's capacity did not change.

We demonstrate the reactive nature of the model in Figure 5. Capacity is affected by the performance of the system being used. Because parallel applications do not run in isolation, there are times when resources become stretched thin across all users. Figure 5 demonstrates what happens to capacity when the network is being heavily used. While the synthetic application has an anticipated capacity of 25, we note that the measured capacity is much lower. This is because another application is utilizing the network to transfer a 10GB file back and forth between the master process and a worker. This is turn causes the I/O time per task to increase for the synthetic application. The result is a decrease in capacity since the execution time remains stable while the I/O time increases (t_e/t_{io}).



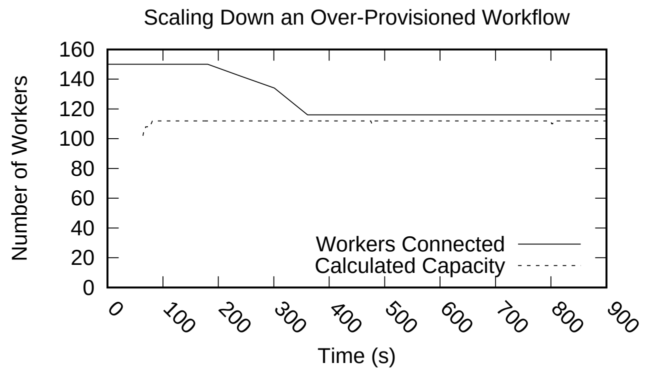
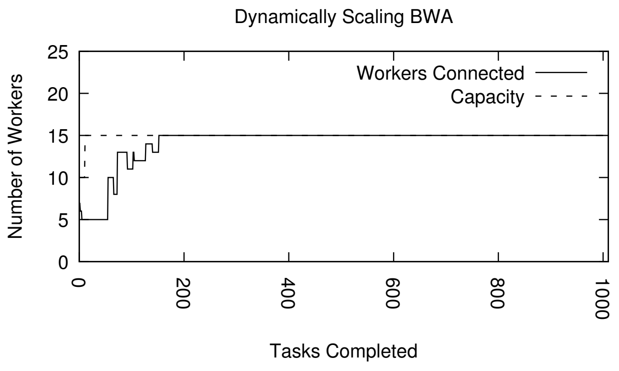


Fig. 4. Scaling up and down. We measure the ability of the model to scale up from an initial under-provisioning on top. At bottom is an example of an application being given too many workers. All workers used one core, and the tasks are homogeneous.



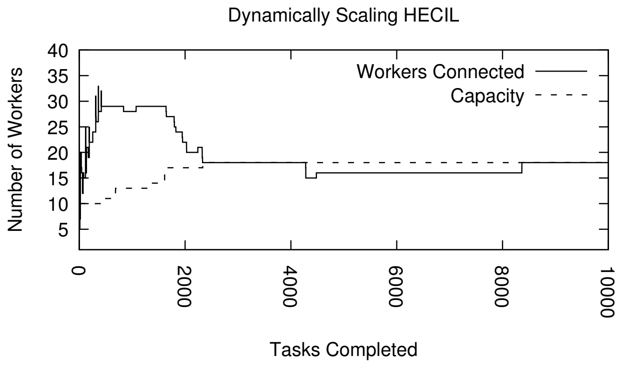


Fig. 6. Provisioning workers to BWA and HECIL. Each worker used a single core. In BWA, a stable capacity is reached by the Work Queue Factory about a fifth of the way through execution. HECIL has a similar capacity but was over-provisioned. A stable capacity is similarly reached about a fifth of the way through its runtime.

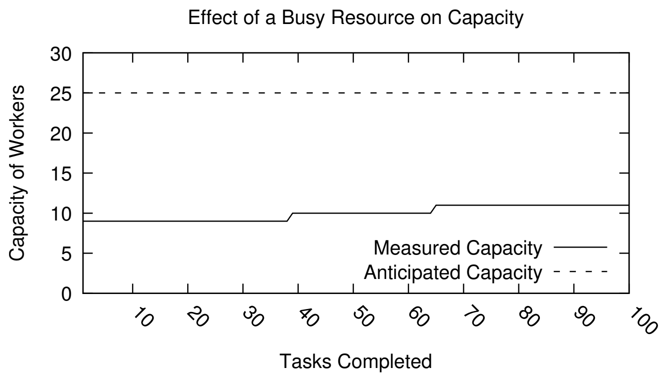


Fig. 5. Contended resource test. We demonstrate how a busy resource affects capacity. Network is the contended resource. While the synthetic application ran, another co-located application was executed which continuously transferred a 10GB file back and forth between its master and a worker elsewhere. Though the tasks are homogeneous, the capacity of the application varies due to the I/O time being affected by network usage.

B. BWA and HECIL Workflows

BWA and HECIL were chosen to demonstrate realistic examples of capacity in scientific workflows. The tasks in both BWA and HECIL share a common reference file 2GB in size.

BWA is a genomics application which relies on a large reference data file to make comparisons with query data. The BWA workflow used for our capacity tests has a scatter-gather behavior meaning there is an initial phase where the large input file is broken into more manageable parts and scattered to different workers. Each worker runs tasks using their part of the larger reference file. Once those tasks are complete, the results from the workers are gathered together by a single task.

We note that the BWA workflow became appropriately provisioned early in its execution. This was due to the initial scatter task producing a somewhat uniform size of input data to be analyzed by subsequent tasks. We can conclude from our execution of BWA that the capacity model provides the user of this workflow an accurate, behavior-based allocation of resources from their initial provisioning of five. As the application executed, the capacity approached a constant value within the first ten tasks which the factory matched.

HECIL is another scatter-gather genomics tool similar to BWA. Also, HECIL has three scatter-gather sections. The first task scatters a reference file to many workers in order to run the first wave of querying tasks. The output of those first tasks is then gathered and scattered by a task in the middle of the workflow. The next wave of querying tasks produce more output which is then gathered in a final set of tasks. Within that final set of tasks, a third scatter-gather section condenses intermediate files to a single output file.

We initially over-provisioned HECIL using Work Queue Factory. The factory can take as input a configuration file which we can alter during the factory's execution. We took advantage of this capability to over-commit workers to demonstrate an initial over-provisioning. We configured the factory to try to maintain more workers than needed for the first 15 minutes of HECIL's execution. Some workers timed out due to lack of work available from the master before we altered the configuration file to instead maintain capacity after the first 15 minutes of execution. Once the configuration changes, we notice that none of the workers which timed out are being replaced. As HECIL's capacity reaches a stable number, the factory meets that value. We note the factory does not quite meet capacity in the middle of the workflow due to the batch system not allocating the factory's replacement workers in a quick manner. Even so, the capacity model was able to guide an initial over-provisioning back down to an appropriate number of resources with HECIL.

The factory's default calculation for requesting workers does not take capacity into account. It is calculated as the maximum value between the number of outstanding tasks and how many tasks a master could theoretically manage (which inspired the current capacity model) [20]. We can use this worker need calculation as a control algorithm to further validate the effectiveness of the capacity model. This is demonstrated with the same BWA application as shown in Figure 6.

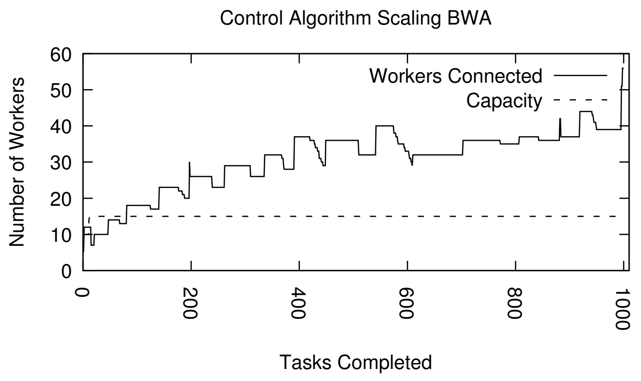


Fig. 7. Provisioning workers to BWA via control algorithm. Each worker used a single core. Using the capacity of BWA first found in Figure 6, we compare Work Queue Factory's default worker acquisition policy as a control algorithm to the implementation of the capacity model.

Figure 7 demonstrates the difference in worker acquisition between the factory's default algorithm and the capacity model. We see that significantly more workers are requested and connect to the master process. It is understandable to assume this addition of resources should increase the throughput of the application. This, however, is not an accurate assumption. The total execution time for BWA with the capacity model guiding the factory's worker acquisition was 3,532 seconds while the run with the control model took 3,910 seconds to execute. Both executions of BWA occurred with similar conditions: the master node had low load and the overall HTCondor system was highly utilized. Though the control algorithm's performance was somewhat slower, it was not significantly so. We can conclude that the addition of these extra resources did not increase throughput of the application as a whole, showing that the capacity model is able to right-provision BWA and maintain that provisioning compared to a more naive model which over-provisioned the application.

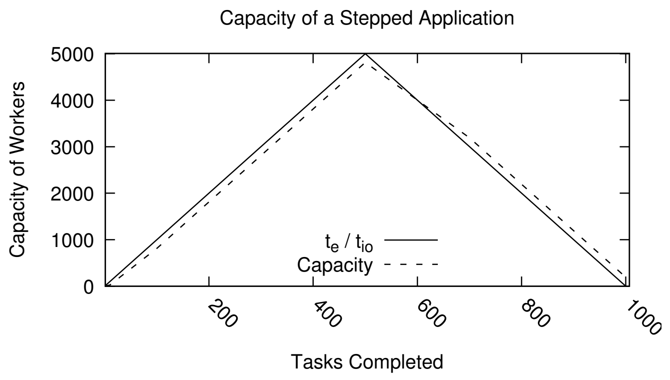


Fig. 8. Stepped application capacity. This simulated application increases in capacity by 10 each task until the application has completed half of its tasks. It then begin decreasing the capacity by ten until all tasks finish. We note the model lags behind the anticipated capacity at each task because every task is different from the previous one.

C. *Model Limitations*

Our model is designed to provide an appropriate resource provisioning based on basic runtime information of tasks. This provides us two significant benefits at the cost of calculating the *optimal* number of resources to provision the application at any given point. First, our model as implemented in Work Queue and Work Queue Factory has negligible overhead since it only adds basic calculations into the execution loops of both programs. We do not have any added database or helper applications added to Work Queue to make the model work. More importantly, we gain the benefit of providing an easy-to-understand model for our users to help understand the behavior of their applications.

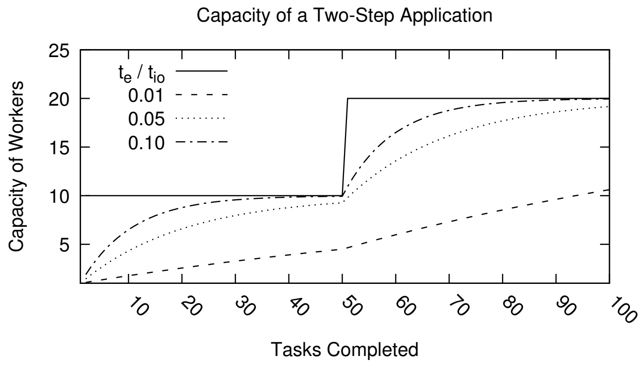
However, the model has some limitations to its effectiveness. As discussed earlier, applications with no sense of grouping tasks make it difficult for the model to provide appropriate resource provisioning for different steps of an application. Instead, the model will converge on an average value among all tasks. Applications which have continuously changing execution or I/O times such that capacity will continuously increase or decrease as shown in Figure 8.

In cases like this, the model will lag behind an appropriate number of resources and has to catch up. At the end of the application, the capacity will not have reached a steady value. We demonstrate how this happens in Table I. The table columns list the number of tasks completed out of the 1000 in the application, the instantaneous capacity (t_e/t_{io} of the current task), and the capacity derived from the model.

Figure 9 demonstrates two cases which the capacity model experiences a significant ramp-up period to converge on a stable number of workers to provision. These cases are somewhat

Tasks Complete (Out of 1000)	Instantaneous Capacity	Modeled Capacity
0	1	1.00
1	11	1.59
10	101	25.19
100	1001	813.12
300	3001	2812.00
500	5001	4812.00
700	3001	3191.99
900	1001	1191.99
990	101	291.99
999	11	201.99
1000	1	201.99

TABLE I
STEPPED APPLICATION CAPACITY



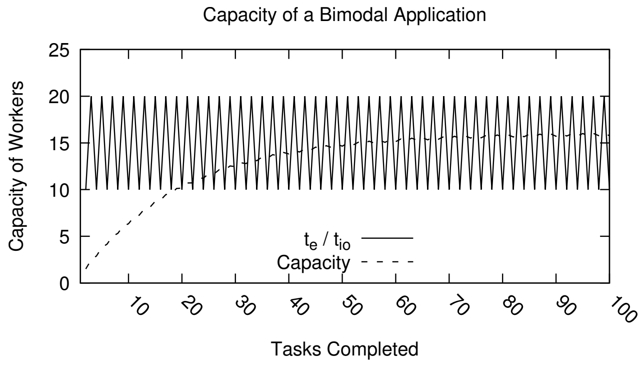


Fig. 9. Cases of the model lagging behind application behavior. These simulated applications are comprised of two types of tasks: Type A tasks and Type B tasks. Type A tasks have an instantaneous capacity (t_e/t_{io}) of 10 while Type B tasks have an instantaneous capacity of 20. The top application runs Type A tasks until the application has completed half of its tasks. It then switches to running exclusively Type B tasks. The bottom application interleaves both types of tasks such that the pattern is: A, B, A, B, etc. Both applications ran 100 tasks. We note the capacity model does not immediately reach the anticipated capacity in either application.

extreme cases of realistic behavior. The top graph shows a two-step application which executes all tasks of a certain type (Type A) before executing all tasks of the next type (Type B). We demonstrate the effect of weighting the most recently completed task greater than the others. In our experimentation, we found a weight of 0.05 to work best. However, this weight barely reaches the anticipated capacity of Type A tasks before the wave of Type B tasks is submitted. This makes the model chase the anticipated capacity, reaching it again shortly before the application finishes. However, adjusting the weight higher causes the model to converge quicker. This would be beneficial for largely static workloads. However, an outlier task (either in execution or I/O time) will have a greater impact on the model. The opposite is true of decreasing the weight to 0.01. The model will scale much slower, but its pattern of growth will be incredibly stable. We can conclude the model is satisfactory for multi-step parallel applications so long as either the number of tasks in each step is significant enough to allow the model its ramp-up or ramp-down time or that the task weight best matches the anticipated task behavior.

The bottom graph in Figure 9 demonstrates the effect of interleaving the two task types upon the model. Because the instantaneous capacity is changing in a seesaw pattern between each task, the capacity model approaches the average between the two types of tasks. This leads to either being over-provisioned by 5 workers when running Type A tasks or under-provisioned by 5 workers when running Type B tasks.

We also demonstrate a real application for which the capacity model found limited use while operating in a full cluster. SHADHO is a framework for machine learning model search and hyperparameter optimization which makes a best-effort search of an infinite search space using heuristics derived from the models being optimized to direct task scheduling. To make the infinite search tractable, SHADHO runs tasks for a user-determined amount of time before shutting down and returns the optimal observed model. Note that the execution time of tasks in SHADHO depends largely on the machine learning model being evaluated.

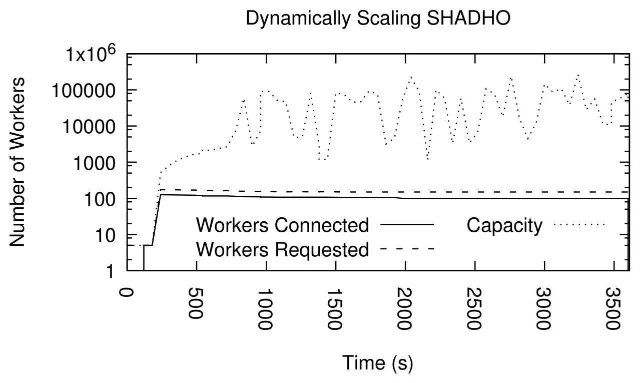


Fig. 10. Provisioning workers to SHADHO. SHADHO workers used 16 cores each. The application ran for one hour then shut down. The capacity of SHADHO is much higher than the amount of work available for workers at any given time.

Figure 10 demonstrates a perfect storm of factors which may limit the model's usefulness. SHADHO never submits enough work to reach capacity. Tasks took on the order of tens of minutes to execute while the I/O time was approximately

0.1 seconds. This led to a very high capacity since the master spent the vast majority of its time waiting for tasks to finish. In this test, no more than 150 tasks were submitted concurrently. While the master could theoretically handle magnitudes more workers, there would never be enough work dispatched to keep them busy. The spikes in capacity seen in Figure 10 are due to different machine learning models taking much longer to execute than others (an order of magnitude difference). When those tasks return their output, capacity spikes from the long execution time of those tasks. It then settles back down when shorter-running machine learning model tasks report back.

Also limiting the model's contribution is the fact that SHADHO executed on a full cluster. This cluster has about 25,000 cores available for use. Most were in use at the time of execution. Of the cores available, about 75% were in use and 90% of the slots available to submit work were occupied. Once the factory attempted to match the number of workers to the number of tasks in the queue, we realized we had reached our limit of how many resources the batch system was able to give. SHADHO never acquired all 150 workers it could have used. We noticed about 25 workers being preempted as other users' work, with better priority, entered the batch queue.

Using the capacity model, we learned SHADHO's performance was bottlenecked by its own task queue management (150 tasks in our test). Due to our implementation of the model in the factory, we only scaled up to the number of tasks in the queue as provisioning more workers would have been wasteful. Assume that SHADHO's task queue was infinite, meaning the bottleneck did not exist. Also assume the factory implementation was different in that we *only* considered the capacity model when requesting more workers. Without these safeguards, we would have flooded the already strained batch system. If we had been executing SHADHO on an infrastructure-as-a-service platform, we would have accrued a very large bill. While the capacity model proved useful in understanding the behavior of SHADHO, it did little in guiding the resource provisioning for it due to a confluence of factors. The user is left knowing that the application will be able to handle any realistic limit placed on its task queue.

To summarize, the capacity model provides limited use for applications in which the execution and transfer times are dissimilar among all tasks. Since capacity is a moving average, the more variance in instantaneous capacity (t_e / t_{io}), the worse the model will behave as seen in Table I. If an application's capacity far exceeds their compute site's resources, the researcher will find limited use for the model's results at *that* time. However, they do glean the fact that their application would be better run at a larger site. Finally, applications which have very little input or output as compared to execution time will have a capacity so large as to be meaningless to the researcher's understanding of the application's behavior.

The capacity model's simplicity provides straightforward transparency for researchers to understand their application's behavior and prevents overhead which would be incurred for a more *exact* measurement of capacity. Utilizing primarily the task execution and transfer times addresses the two bottlenecks of a master-worker application. If the master is waiting for tasks to complete (t_e), then it has time to send new tasks and transfer files (t_{io}). The ideal master is always busy starting new tasks or getting the results of a finished task.

VI. Capacity Model as Troubleshooting Tool

Although the capacity model provides usefulness for our users as implemented in Work Queue Factory and as a simple paper model to gauge an application's resource provisioning, we also implemented the model as the basis for a web-based troubleshooting tool. This is possible by querying a catalog server which by default every Work Queue master process communicates with in regular intervals. The catalog server is used to match workers to masters, but it also stores basic performance metrics of the master.

Some of the metrics included in the catalog server are cores, memory, and disk allocated as well as how the master process is spending its time (e.g. sending input, receiving output, running application-specific code). We added capacity to these metrics. We provide these metrics in a dashboard of simple visualizations of each Work Queue master to give users insight into the behavior and current resource needs of their application. Visiting this dashboard is intended to be the first step when a user is troubleshooting resource provisioning issues with their application. The visualizations are implemented in the D3 JavaScript library.

In our experience, this tool has been a good first step in troubleshooting common resource provisioning issues. To make the tool more user-friendly, we have added a recommendation system which will briefly analyze a master's metrics and provide actionable steps to help the user troubleshoot. The recommendations are based on the most common factors which would contribute to a master's behavior. For example, a master which is spending much of its time polling workers (i.e. the master is idle much of the time) is most likely being under-provisioned. In this case, the recommendation system will look at how many workers the master has connected as well as its capacity and inform the user if they would benefit from asking for more workers (since they currently have none).

VII. Conclusions

We presented a model for measuring a master-worker application's capacity. We implemented the model in Work Queue Factory which can be used to both scale up from an under-provisioned start and scale down in the case of being over-provisioned with workers. This was shown first using a synthetic application then reinforced with common bioinformatics workflows BWA and HECIL. We then demonstrated potential limitations of the model with the SHADHO machine learning application. This model gives the typical user the capability of harnessing only as many resources as they need on the *first* execution of their application. We also demonstrated the applicability of the model as a troubleshooting tool with a simple visualization designed for our users. The capacity model provides two distinct use cases to help researchers optimize their applications' resource provisioning.

References

[1] N. Hazekamp, J. Sarro, O. Choudhury, S. Gesing, S. Emrich, and D. Thain, "Scaling up bioinformatics workflows with dynamic job expansion: A case study using galaxy and makeflow," in *2015 IEEE 11th International Conference on e-Science*, Aug 2015, pp. 332–341.

[2] O. Choudhury, A. Chakrabarty, and S. Emrich, "Hecil: A hybrid error correction algorithm for long reads with iterative learning," *bioRxiv*, 2017. [Online]. Available: https://www.biorxiv.org/content/early/2017/07/13/162917

[3] J. Kinnison, N. Kremer-Herman, D. Thain, and W. Scheirer, "Shadho: Massively scalable hardware-aware distributed hyperparameter optimization," *arXiv preprint arXiv:1707.01428*, 2018.

[4] F. Chirigati, V. Silva, E. Ogasawara, D. de Oliveira, J. Dias, F. Porto, P. Valduriez, and M. Mattoso, "Evaluating parameter sweep workflows in high performance computing," in *Proceedings of the 1st ACM SIGMOD Workshop on Scalable Workflow Execution Engines and Technologies*, ser. SWEET '12. New York, NY, USA: ACM, 2012, pp. 2:1–2:10. [Online]. Available: http://doi.acm.org/10.1145/2443416.2443418

[5] L. G. Valiant, "A bridging model for parallel computation," *Commun. ACM*, vol. 33, no. 8, pp. 103–111, Aug. 1990. [Online]. Available: http://doi.acm.org/10.1145/79173.79181

[6] N. Carriero and D. Gelernter, "How to write parallel programs: A guide to the perplexed," *ACM Comput. Surv.*, vol. 21, no. 3, pp. 323–357, Sep. 1989. [Online]. Available: http://doi.acm.org/10.1145/72551.72553

[7] M. Albrecht, P. Donnelly, P. Bui, and D. Thain, "Makeflow: A portable abstraction for data intensive computing on clusters, clouds, and grids," in *Proceedings of the 1st ACM SIGMOD Workshop on Scalable Workflow Execution Engines and Technologies*, ser. SWEET '12. New York, NY, USA: ACM, 2012, pp. 1:1–1:13. [Online]. Available: http://doi.acm.org/10.1145/2443416.2443417

[8] M. C. Wikstrom and J. L. Gustafson, "The twin bottleneck effect," in *[1993] Proceedings of the Twenty-sixth Hawaii International Conference on System Sciences*, vol. ii, Jan 1993, pp. 574–583 vol.2.

[9] D. Krol and J. Kitowski, "Self-scalable services in service oriented software for cost-effective data farming," *Future Generation Computer Systems*, vol. 54, pp. 1 – 15, 2016. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167739X15002265

[10] N. Roy, A. Dubey, and A. Gokhale, "Efficient autoscaling in the cloud using predictive models for workload forecasting," in *2011 IEEE 4th International Conference on Cloud Computing*, July 2011, pp. 500–507.

[11] A. Y. Nikravesh, S. A. Ajila, and C. H. Lung, "Measuring prediction sensitivity of a cloud auto-scaling system," in *2014 IEEE 38th International Computer Software and Applications Conference Workshops*, July 2014, pp. 690–695.

[12] T. Lorido-Botran, J. Miguel-Alonso, and J. A. Lozano, "A review of auto-scaling techniques for elastic applications in cloud environments," *Journal of Grid Computing*, vol. 12, no. 4, pp. 559–592, 2014. [Online]. Available: http://dx.doi.org/10.1007/s10723-014-9314-7

[13] Z. Tan and S. Babu, "Tempo: Robust and self-tuning resource management in multi-tenant parallel databases," *Proc. VLDB Endow.*, vol. 9, no. 10, pp. 720–731, Jun. 2016. [Online]. Available: http://dx.doi.org/10.14778/2977797.2977799

[14] A. Gandhi, M. Harchol-Balter, R. Raghunathan, and M. A. Kozuch, "Autoscale: Dynamic, robust capacity management for multi-tier data centers," *ACM Trans. Comput. Syst.*, vol. 30, no. 4, pp. 14:1–14:26, Nov. 2012. [Online]. Available: http://doi.acm.org/10.1145/2382553.2382556

[15] H. Menon and L. Kalé, "A distributed dynamic load balancer for iterative applications," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '13. New York, NY, USA: ACM, 2013, pp. 15:1–15:11. [Online]. Available: http://doi.acm.org/10.1145/2503210.2503284

[16] H. Jordan, R. Prodan, V. Nae, and T. Fahringer, "Dynamic load management for mmogs in distributed environments," in *Proceedings of the 7th ACM International Conference on Computing Frontiers*, ser. CF '10. New York, NY, USA: ACM, 2010, pp. 337–346. [Online]. Available: http://doi.acm.org/10.1145/1787275.1787344

[17] R. Gandhi, H. H. Liu, Y. C. Hu, G. Lu, J. Padhye, L. Yuan, and M. Zhang, "Duet: Cloud scale load balancing with hardware and software," in *Proceedings of the 2014 ACM Conference on SIGCOMM*, ser. SIGCOMM '14. New York, NY, USA: ACM, 2014, pp. 27–38. [Online]. Available: http://doi.acm.org/10.1145/2619239.2626317

[18] S. Prabhakaran, M. Neumann, S. Rinke, F. Wolf, A. Gupta, and L. V. Kale, "A batch system with efficient adaptive scheduling for malleable and evolving applications," in *2015 IEEE International Parallel and Distributed Processing Symposium*, May 2015, pp. 429–438.

[19] Y. Zhang, Q. Liu, S. Klasky, M. Wolf, K. Schwan, G. Eisenhauer, J. Choi, and N. Podhorszki, "Active workflow system for near real-time extreme-scale science," in *Proceedings of the First Workshop on Parallel Programming for Analytics Applications*, ser. PPAA '14. New York, NY, USA: ACM, 2014, pp. 53–62. [Online]. Available: http://doi.acm.org/10.1145/2567634.2567637

[20] L. Yu, "Right-sizing Resource Allocations for Scientific Applications in Clusters, Grids, and Clouds," Ph.D. dissertation, University of Notre Dame, 2013.

[21] J. L. Gustafson, "Reevaluating amdahl's law," *Commun. ACM*, vol. 31, no. 5, pp. 532–533, May 1988. [Online]. Available: http://doi.acm.org/10.1145/42411.42415

[22] J. Gustafson, G. Montry, and R. Benner, "Development of parallel methods for a 1024-processor hypercube," *SIAM J. Sci. Stat. Comput.*, vol. 9, no. 4, 1988.

[23] S. Krishnaprasad, "Uses and abuses of amdahl's law," *J. Comput. Sci. Coll.*, vol. 17, no. 2, pp. 288–293, Dec. 2001. [Online]. Available: http://dl.acm.org/citation.cfm?id=775339.775386

[24] J. G. Shanahan and L. Dai, "Large scale distributed data science using apache spark," in *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '15. New York, NY, USA: ACM, 2015, pp. 2323–2324. [Online]. Available: http://doi.acm.org/10.1145/2783258.2789993

[25] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, Jan. 2008. [Online]. Available: http://doi.acm.org/10.1145/1327452.1327492

[26] D. W. Walker and J. J. Dongarra, "Mpi: a standard message passing interface," *Supercomputer*, vol. 12, pp. 56–68, 1996.

[27] D. Rajan, A. Thrasher, B. Abdul-Wahid, J. A. Izaguirre, S. Emrich, and D. Thain, "Case studies in designing elastic applications," in *2013 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*, May 2013, pp. 466–473.

[28] D. Thain, T. Tannenbaum, and M. Livny, "Distributed computing in practice: the condor experience." *Concurrency - Practice and Experience*, vol. 17, no. 2-4, pp. 323–356, 2005.

Appendix

A. *Abstract*

This description contains the information needed to run the experiments of the SC18 paper "A Lightweight Model for Right-Sizing Master-Worker Applications". We explain how to compile and run Work Queue, Work Queue Factory, and each experiment provided in the paper.

B. *Description*

1) Check-list (artifact meta information):

- **Program:** Work Queue, Work Queue Factory, BWA, HECIL, SHADHO
- **Compilation:** gcc version 4.8.5 20150623 (Red Hat 4.8.5-16) (GCC) Flags provided in software Makefiles.
- **Data set:** Generated .fastq files for BWA and HECIL, MNIST data set for SHADHO.
- **Run-time environment:** Redhat 6/7, Centos 6/7.
- **Hardware:** x86_64 Architecture, 1-16 core machines.
- **Output:** Runtime statistics to generate graphs, genome output of BWA and HECIL, hyperparameter output of SHADHO.
- **Experiment workflow:** git clone projects and install software dependencies via Makefile. Run experiments from Makefile and compare with results presented here.
- **Experiment customization:** Number of workers to provision, size of generated dataset (data generation tools included), BWA, HECIL, and SHADHO parameters and inputs.
- **Publicly available?:** Yes.

2) How software can be obtained (if available): All source code can be obtained from the following GitHub repository: github.com/cooperative-computing-lab/cctools

3) Hardware dependencies: We have assumed x86 architecture, though all tools are built from source so other architectures may be feasible.

4) Software dependencies: We rely on GCC to compile our binaries, Perl 5 to run miscellaneous scripts, Git to download code repositories, and gnuplot 5.0 for producing our graphs and visualizing our results. SHADHO relies on Python 2.7, but it may be reconfigured to run on Python 3.

We also ran all experiments via the HTCondor batch system. If HTCondor is unavailable, the experiments may be altered to use a different batch system via their respective Makefiles and Perl helper scripts.

5) Datasets: The experiments presented in this paper made use of three datasets:

- BWA Dataset (2GB .fastq reference) - github.com/cooperative-computing-lab/makeflow-examples/bwa
- HECIL Dataset (2GB .fastq reference) - github.com/cooperative-computing-lab/makeflow-examples/hecil
- SHADHO Dataset (11MB mnist.npz) - github.com/jeffkinnison/shadho

C. Installation

All software used for this paper is available in GitHub repositories. To download and install the necessary software, run:

```
$ git clone
https://github.com/cooperative-computing-
lab/capacity-paper-data
$ cd ./capacity-paper-data
$ make build
```

This will download the software used to run Work Queue Factory as well as the experimentation suite written to produce this paper's figures.

D. Running experiments

As there were four applications executed in this paper, we will break down the experiment workflow into four sections for clarity. To aid in the simplicity of reproducibility, each experiment is given its own directory in the Git repository. Within each of those experiment directories is a Makefile which outlines each step necessary to reproduce the results presented in this work. In order to decrease clutter, these Makefiles have been designed to streamline the execution of each experiment to a simple call to `make`.

E. Evaluation and expected results

1) Synthetic results: To produce the capacity measurements from the synthetic application, first be sure you are in the capacity-paper-data directory. Then, execute:

```
$ make ratios
$ make stepped
$ make varied
$ make provisioning
$ make isolation
```

This will run all the modeling scripts used to generate synthetic figures presented here as well as all live experiments. The ratios, stepped, varied, and provisioning directories all produce modeled data. There are two live application tests in the provisioning directory as well. Thse are the upward scaling and downard scaling experiments. The isolation experiment will run two instances of the synthetic application concurrently. One instance will transfer a 10GB file between the master process and a worker established by Work Queue Factory. The other instance will execute as described in the paper.

The results obtained may be compared to the data and plots located within the ratios/paper_results, stepped/paper_results, varied/paper_results, provisioning/paper_results, and isolation/paper_results directories. Specific steps in generating the data can be altered in the Makefile for these experiments in ratios/Makefile, stepped/Makefile, varied/Makefile, provisioning/Makefile, and isolation/Makefile respectively.

2) BWA results: To produce the capacity measurements from BWA, first be sure you are in the capacity-paper-data directory. Then, execute:

```
$ make bwa
```

This will download and compile BWA, generate the input data set, and execute BWA with Work Queue Factory to gather capacity metrics. Once execution is finished, the capacity data is analyzed and plotted using gnuplot. BWA will execute again, this time without the capacity model turned on, for comparison to the capacity model as shown in Figure 7. The results obtained may be compared to the data and plot located within the bwa/paper_results directory. Specific steps in generating the data can be altered in the Makefile for this experiment in bwa/Makefile.

3) HECIL results: To produce the capacity measurements from HECIL, first be sure you are in the capacity-paper-data directory. Then, execute:

```
$ make hecil
```

This will download and compile BWA (a software dependency for HECIL), generate the input data set, execute BWA locally to prepare the data for HECIL, then execute HECIL with Work Queue Factory to gather capacity metrics. Once execution is finished, the capacity data is analyzed and plotted using gnuplot. The results obtained may be compared to the data and plot located within the hecil/paper_results directory. Specific steps in generating the data can be altered in the Makefile for this experiment in hecil/Makefile.

F. SHADHO results

To produce the capacity measurements from SHADHO, first be sure you are in the capacity-paper-data directory. Then, execute:

```
$ make shadho
```

Ref size	Query size	Sequences	Runtime
20MB	237KB	100	10 sec:1 machine
196MB	20MB	1000	2 min:20 machines
196MB	237MB	1000	6 min:20 machines
2.0GB	237MB	1000	30 min:20 machines

TABLE II
BWA AND HECIL INPUT CONFIGURATION

This will download and compile SHADHO and execute SHADHO with Work Queue Factory to gather capacity metrics. Once execution is finished, the capacity data is analyzed and plotted using gnuplot. The results obtained may be compared to the data and plot located within the shadho/paper_results directory. Specific steps in generating the data can be altered in the Makefile for this experiment in shadho/Makefile.

G. *Experiment customization*

Each experiment is kept within its own directory with a corresponding Makefile. Many of the parameters for these experiments may be tweaked by editing these Makefiles before running. Example parameters to change include: cores, memory, and disk per worker, batch system to use, and total tasks to run (for synthetic tests).

BWA and HECIL rely upon a generated data set. The size of this data set can be configured in the bwa/Makefile and hecil/Makefile respectively. The files generated for both are ref.fastq and query.fastq. Consult Table II for generating appropriate sizes for both files. Then, use fastq_generate.pl like so (this will produce the 2GB file used in the paper):

```
$ fastq_generate.pl 1000000 1000
$ fastq_generate.pl 1000000 100 ref.fastq
```

We also assume that the HTCondor batch system exists. If this is not the case, the batch system may be changed when calling Work Queue Factory in the Makefiles and Perl helper scripts for the experiments. BWA, HECIL, and Scaling experiments all rely upon a Perl helper script to instantiate Work Queue Factory properly. The batch system the factory uses can be reconfigured by changing this helper script. Each of these helper scripts follow the pattern of [Name of Experiment]_test.pl for BWA, HECIL, and Scaling respectively. For the other experiments, the call to Work Queue Factory can be found in the Makefile for that experiment and may be modified there.

H. *Notes*

Since the majority of the graphs prepared for this paper rely upon executing an application on a batch system, it may be difficult to reproduce the exact numbers presented here. Depending on the quality and quantity of hardware in the cluster, performance and capacity will be somewhat different. However, the results produced should be comparable to those presented here.

The data sets used for BWA and HECIL are generated locally. Though the capacity behavior should remain very similar to the behavior presented here, some slight variations may occur due to how the data is generated.

HECIL generates approximately 450GB of output data during execution. BWA produces approximately 50GB of output data. Ensure your system meets these storage requirements before running HECIL and BWA.

All tools, programs, and scripts are available from the following GitHub repositories:

github.com/cooperative-computing-lab/cctools
github.com/cooperative-computing-lab/makeflow-examples
github.com/cooperative-computing-lab/capacity-paper-data