Performing Large Science Experiments on Azure: Pitfalls and Solutions

Wei Lu, Jared Jackson, Jaliya Ekanayake, Roger S. Barga, Nelson Araujo
Microsoft Research eXtreme Computing Group

***Abstract*—Carrying out science at extreme scale is the next generational challenge facing the broad field of scientific research. Cloud computing offers to potential for an increasing number of researchers to have ready access to the large scale compute resources required to tackle new challenges in their field. Unfortunately barriers of complexity remain for researchers untrained in cloud programming. In this paper we examine how cloud based architectures can be used to solve large scale research experiments in a manner that is easily accessible for researchers with limited programming experience, using their existing computational tools. We examine the top challenges identified in our own large-scale science experiments running on the Windows Azure platform and then describe a Cloud-based parameter sweep prototype (dubbed Cirrus) which provides a framework of solutions for each challenge.**

***Keywords*-Cloud Computing; Windows Azure; Parameter Sweep;**

I. Introduction

Before the advent of cloud computing, solving interesting scientific research questions that required large scale computation was reserved mainly for a small echelon of well-funded researchers. The hardware, maintenance, and development costs of local clusters can be prohibitively high for the majority of researchers. At the same time recent advances and cost reductions in sensor networks, devices, and electronic storage have brought in huge quantities of data to researchers in every field. With the emerging option of cloud based computing [1] more and more researchers now have access to the computational tools necessary to process and analyze the vast amounts of data they have been collecting. This opening of resources has been referred to as the "democratization of research" and an ever increasing number of researchers are seeking how to best make use of these new capabilities. While access to compute resources is now readily available, there are still obstacles facing researchers looking to move to cloud computing. One obstacle is the high learning curve required to make use of the new technology. Currently, cloud users need to be either well versed in Virtual Machine management or be a skilled programmer to implement even the simplest of cloud solutions. In addition, the challenges to convert old computation methods to distributed platforms with separate storage and computation mechanisms are not well understood, even to those familiar with working in the cloud.

Herein we show an approach to solve the problems faced by researchers moving over to cloud architecture, in this case the Windows Azure platform [2]. We first describe an experiment running the common BLAST algorithm against an input data set of a size never before attempted. We faced several challenges in moving this common and highly parallel application to the cloud. We describe in detail these challenges. With these challenges in mind we introduce a new service within Windows Azure that we believe generalizes the procedure of conducting large-scale science experiments on the cloud. Our platform supports parametric sweep/task farm across elastic cloud compute resources. It also implements predictive sampling runs on provided applications, which offers cost estimations before running applications at scale. Finally, we examine the benefits of this approach and some of the challenges remaining to offer cloud based computing to the general research community.

The structure of this paper is as follows. In Section II we briefly discuss the Windows Azure cloud platform and the AzureBLAST application. In Section III we introduce a large-scale science experiment we have conducted and the pitfalls we have identified. In Section IV we present our Azure-based general parameter sweeping service and solutions for each pitfall identified in Section III. In Section V we carry out a detailed performance analysis.

II. AzureBLAST

Biology was one of the first scientific fields to embrace computation to solve problems that would otherwise be intractable. This early development, which created the field of bioinformatics, led to the authoring of many applications which are now widely used and widely available for general research. One of the most common of these tools is the BLAST program [3], which allows researchers to compare DNA or protein sequences against large databases of known sequences to find similar structures. Fortunately, BLAST lends itself well to parallelization as many researchers have noted [4]. Unfortunately, for the most part ready access to parallelized versions of BLAST have only been available to researchers with large grants, able to fund the purchase of a compute cluster, or with relatively small problems able to use free (and limited) public web services. This made BLAST an important target for us in analyzing how scientific applications could be moved to the cloud. Our earlier work, AzureBlast [5], is a parallel BLAST engine on Windows Azure. To understand the implementation of AzureBLAST, a basic understanding of Azure is necessary.

The Window Azure platform resides on large clusters of rack mounted computers hosted in industrial sized data

centers across the world. The clusters of computers are separated into two roles. One is controlled by the *Compute Fabric* and one is controlled by the *Storage Fabric*. The Compute Fabric allows access to VMs running inside data center computers. The VMs are generally categorized into two types as well: worker roles and web roles. Web roles provide a platform for hosting web services and web page requests via HTTP or HTTPS, while worker roles are generally used for data fetching and the actual algorithm computation. The Storage Fabric hosts various data storage services, including blobs (binary large objects), row-column based tables and reliable message queues. The queue service needs to be elaborated on here because of its critical role in task distribution and fault tolerance. When retrieving a message from one queue the user can specify the `visibilityTimeout` and the message will remain invisible during this timeout period then reappear in the queue if it has not been deleted by the end of the timeout period. This feature ensures that no message will be lost even if the instance which is processing the message crashes, providing fault tolerance for the application.

In AzureBLAST a job is parallelized in the typical "fork-join" fashion. The user input is provided via an external facing web page hosted by a web role, and will be divided into number of small partitions which are distributed as task messages through an Azure queue. The computation is performed by controlling a variable number of worker nodes executing NCBI's publicly available BLAST program [6]. Each worker polls the message queue and receives messages (i.e., tasks) corresponding to a small partition of the input sequences being worked on. The worker will execute the BLAST application against the partition it receives. Once the application has completed the worker collects the resulting data and places it into blob storage. Finally, a merging task is issued to aggregate all the partial results.

III. Challenges Introduced by the Cloud

Running a science application at the scale of several compute years over hundreds of GBs of data on any cloud computing platform is still a very new endeavor. While we feel comfortable with our AzureBLAST design, we noticed several areas of concern that need to be addressed in future runs. Characteristic of Cloud infrastructure and its programming model as well as that of large scale applications both contributed to the challenges we identified, and we believe these are generalizable to many pleasingly parallel data-intensive computations on the cloud as well.

Our task, with guidance from researchers at Children's Hospital in Seattle, was to take the 10 million protein sequences (4.2GB size) in NCBI's non-redundant protein database and BLAST each one against the same database. This sort of experiments is known as an "all-by-all" comparison which can identify the interrelationship of all protein sequences in this database. We conducted this large BLAST experiment on AzureBLAST. We used the largest available VM size provided by Azure, which is an 8 core CPU with 14GB RAM and a two TB local disk, to allow the executable access to large system memory and multiple computation cores. Our initial evaluations showed this would take on the order of six to seven CPU years, even on an optimized computer system. It is so computationally intensive that we allocated approximately 3,700 weighted instances (475 extra-large VMs) from three datacenters. Each datacenter hosted three AzureBLAST deployments, each with 62 extra-large instances. The 10 million sequences are then divided into multiple segments, each of which was submitted to one AzureBLAST deployment for execution. The entire job took 14 days to complete and the output produced 260 GBs of compressed data spread across over 400,000 output files.

Timestamp	Machine name	Event
3/31/2010 06:14	RD00155D3611B0	Executing task 251523...
3/31/2010 06:25	RD00155D3611B0	Execution of task 251523 is done, it takes 10.9 mins
3/31/2010 06:25	RD00155D3611B0	Executing task 251553...
3/31/2010 06:44	RD00155D3611B0	Execution of task 251553 is done, it takes 19.3 mins
...	...	...
3/31/2010 08:22	RD00155D3611B0	Executing the task 251774...
3/31/2010 09:50	RD00155D3611B0	Executing the task 251895...
3/31/2010 11:12	RD00155D3611B0	Execution of task 251895 is done, it takes 82 mins

Table I
LOGS OF AZUREBLAST

In order to understand how Azure performed during this experiment, we visualized the log data generated by AzureBLAST. When executing one task, each AzureBLAST instance simply logs the start time and the completion time of the task execution. As each instance executes tasks sequentially, the logs of a normal run should look like the upper part in Table I where each task start event and completion event are paired. Conversely, the logs listed in the lower part of Table I illustrate an unmatched task start event, which indicates that a task execution was lost. The loss of a task further implies an abnormality in the execution, for example an instance is restarted due to a system failure. In th visualization we depict blue dots for the matched two events and red dots for any unmatched event.

A. Failures

The Cloud is a multi-tenant environment with potentially millions of machines in a datacenter, where the failures of individual units are inevitable. We identified various types of failures from our logs. For example, the instance progress diagram in Figure 1 visualizes the run of one blast job, which contains about 30,000 tasks executed by 62 extra-large instances in Microsoft West Europe datacenter in six days. It was selected as it is representative of the most common errors observed in the all-by-all experiment.

- Instance physical failure: Instance 48 stopped working at 03/26/2010 1:54 PM when it completed the last task.

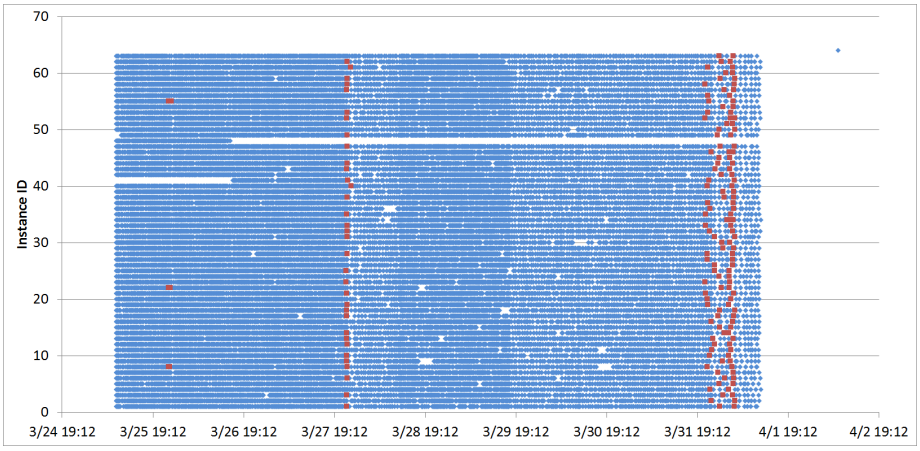

Figure 1. Various failures observed in a large experiment

After 2 hours, instance 41 started and kept working till the end. Notice that we allocated only 62 instances for this job, but the figure shows a total of 63 instances. A reasonable conjecture is that instance 48 experiences some unrecoverable fatal error. As a result of the failover mechanism the Azure fabric controller started a new instance, namely instance 41, as its replacement.

- Storage exception: starting at 03/27/2010 22:53 34 instances encountered a blob write timeout exception and stopped working; at around 0:38 all of the effected instances began executing tasks again at the same time. Although we have little clue about what caused this failure, we conjecture the Azure fault domain [2] played a critical role. By default, Azure divides all instances of one deployment into at least two groups, called fault domains, and places each group in different failure zones (e.g., rack, power supply, etc.) in the datacenter to avoid a single point failure. As we observed, about half of our active instances lost connection to the storage service while another half successfully accessed the storage service. Notice that unlike the previous instance failure these failed instances were automatically recovered by the fabric controller.
- System update: in the evening of 03/31/2010, all 62 instances experienced the loss of a task on two occasions, one at 22:00, followed by the second one at 3:50. From a zoomed visualization, not shown in this paper due to limited space, we identified a pattern for the first occurrence. The 62 instances are divided into 10 groups; each group of instances got restarted sequentially; the complete group restart took around 30 minutes and the entire duration was about five hours. It turns out that a system update caused the entire ensemble to restart, and due to in-place update domain mechanism Azure restarts and updates all instances group by group to ensure system availability during the update. In contrast, the second failure took place much more quickly: all instances get restarted in about 20 minutes for some unknown reason.

As we can see, failures are to be expected in the cloud. While only one instance failure occurred during the 6 days, most exceptions are essentially caused by datacenter maintenance, thus being automatically recoverable. The most important design principle of cloud programming is to design with failure in mind [7] and other design tips are well-known today, such as being stateless, avoid in-memory sessions and so on. In such long-running massive-computation job, however, one very important challenge is how best to handle failures in term of cost-effectiveness.

B. *Instance Idle Time & Load imbalance*

In our post-analysis we also discovered that a large portion of our paid compute time was actually wasted with our VMs idling. One obvious source of potential waste comes from the gap time between two consecutive jobs. Once a job has completed, all the worker instances will sit idle waiting for next job, which may arrive hours later. Furthermore, we assume all jobs have similar parallelism and computation complexity. However, in practice the complexity of science experiments can vary significantly. Having all jobs use the same set of instances leads to either over usage or under usage of the costly resource.

Another source of waste is caused by a considerable degree of load imbalance. In AzureBLAST, we allocated the number of instances statically before the job was executed and never change that number until we bring the entire deployment down. The static allocation, especially for long running large scale jobs, will most likely lead to a significant waste of instance hours when the load distribution becomes imbalanced. Load imbalance comes primarily from one source: the complexity of each individual task. We observed that even though we tried to create an equal partition scheme in which each task would take a similar

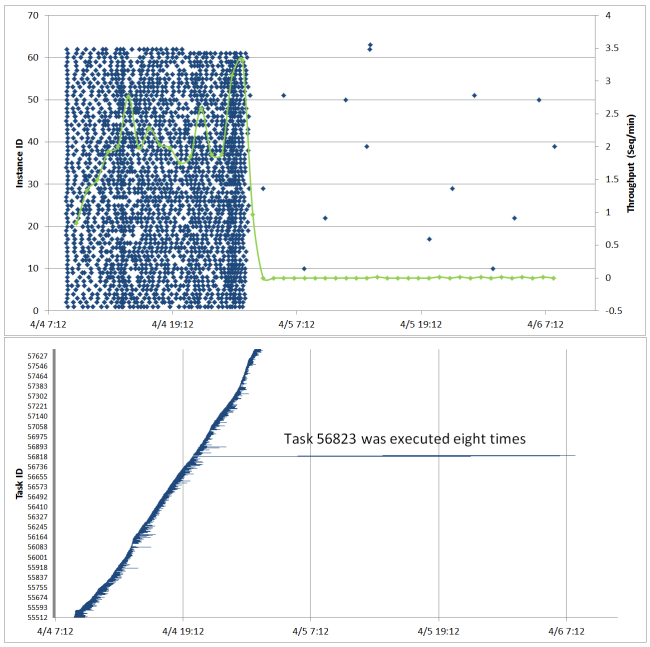

Figure 2. Load imbalance observed in one small experiment

amount of time to compute, there were several outliers that took considerably longer time to complete. To illustrate, Figure 2 visualizes the instance progress of one small job which contains only 2058 tasks. While the majority of tasks complete in less than one day the entire job takes more than three days, the final two days with ultra-low resource usage. To identify those tasks which took the most time, we convert the instance progress diagram into a task progress diagram (a.k.a. swimming pool diagram). It turned out that task 56823 is the only exceptional task that required more than 18 hours to complete, while the average processing time for all other tasks in the job is only 30 minutes. Estimating the complexity of individual BLAST calls turns out to be very difficult to predict. We have little recourse to prevent this imbalance and consequence of load imbalance is that the user has to pay for all idle instance hours.

C. Limitation of Current Azure Queue

The current Azure queue service has two limitations for running large-scale science experiments. The first limitation is that the `visibilityTimeout` of a message has to be less than 2 hours [8], which is often insufficient for science experiments. This means that if the execution time of a task is more than 2 hours, the task message will automatically reappear in the queue no matter whether it is being processed by some active instance(s), thus leading to repeated computation. To mitigate this problem, a simple workaround is that before executing a task the instance first check if its result has been generated. For example, in Figure 2, the long-tail task 56823 actually has been simultaneously executed by 8 instances until the result was finally generated by one of them.

The second limitation is the maximum lifetime of an Azure message, which is limited to seven days. That means if one message stays in the queue for more than seven days it will be automatically deleted. However, it is not uncommon that a science run will require weeks to complete. This lifetime limit requires that we build ancillary functions to support long-lived experiments.

D. Performance/Cost Estimation

Most scientific programs allow for tuning the application via setting various parameters of the executable. In the case of BLAST, the user can tune the algorithm by setting a dozen such parameters, such as the search-word size (-W), alignment threshold expectation (-e) or using MegaBlast optimization (-n). The settings on these parameters have subtle effect on the degree of accuracy or number of hits to record and thus have a considerable impact on the cost when multiplied at scale. The cost required to run the entire job to identify the optimal parameters setting is prohibitive. Hence it is very useful to help the user quickly estimate the cost for a given parameter setting so they can identify correlations between parameters and computation and determine the tradeoff before consuming a large number of instance hours.

Another example of tunable parameters comes from the elastic nature of the cloud. The user can select the number of instances to spin up as well as the size of those instances (e.g., small, medium, large or extra-large VM sizes). A scientist usually has only a vague idea about how long their experiment may take with large scale resources, resulting in inaccurate estimates on how much an experiment will cost. A quick estimate on compute cost with varied resource configurations can help decide whether to scale in/out the number of instance or scale up/down the instance size to meet the budget limitation.

E. Minimizing the needs for programming

The last, but not the least observed problem is less empirical in nature and yet is likely the largest impediment to running research in the cloud. Quite simply, most researchers do not have the technical and/or programming experience necessary to move their work to a cloud based platform. From a range of BLAST experiments in which we cooperated with biologists, we realized a complete science experiment usually consists of multiple steps and each step may need to run different binaries. This creates a data driven workflow that needs to be executed in the cloud. Unfortunately, most researchers do not possess the programming experience to translate these workflows into an operational cloud application. Requiring them to learn cloud programming techniques is both impractical and untenable. Therefore we need to provide a method to enable researchers

to run their binaries and simple but lengthy workflows on the Cloud with minimal need for coding.

IV. Cirrus: A General Parameter Sweep Service on Azure

We believe that each of the challenge we have encountered, as outlined in the previous section, can readily generalize to a large number of science applications when run in the cloud. We thus transformed our AzureBLAST experience into a general platform for executing any legacy Windows executable in a distributed fashion on the cloud. The result is *Cirrus*, an Azure-based parametric sweep service.

A. Parameter Sweep Job Script

A Cirrus job is described in a simple declarative scripting language derived from Nimrod [9] for expressing a parametric experiment. Every job definition has three parts: i) prologue, ii) commands and iii) a set of parameters. In the prologue the user can define the script to be executed by the instance when it executes a task of this job the first time. The prologue is usually used to setup the running environment (e.g., staging the data) for the task execution. The commands part contains a sequence of the shell script, which is going to be materialized and executed as tasks. In the script, besides most regular shell commands, the user can invoke Azure-storage-related commands, such as `AzureCopy` which transfers the data between Azure blob storage and the instance. Also the user can refer to a parameter variable by `%parameter%` in the script, which will be materialized with the real value by the parametric engine. The parameter variable is declared in the parameter section. We support most parameter types in Nimrod, such as `Range` and `Select`. In addition, we provide Azure-related parameter types, such as `SelectBlobs`, which iterates each blob under one container as the parameter space. The example below illustrates a blast job that queries all query sequences stored in the blob container "partitions" against the subject database (i.e. uniref).

```
<job name="blast">
  <prolog>
    <!-- staging the database -->
    azurecopy http://.../uniref.fasta uniref.fasta
  </prolog>
  <cmd>
     <!-- download the partition -->
     azurecopy %partition% input
     <!-- run the NCBI blast binary -->
     blastall.exe -p blastp -d uniref.fasta
                          -i input -o output
     <!-- upload the partial results -->
     azurecopy output %partition%.out
  </cmd>
  <parameter name="partition">
    <selectBlobs>
      <prefix>partitions/</prefix>
    </selectBlobs>
  </parameter>
</job>
```

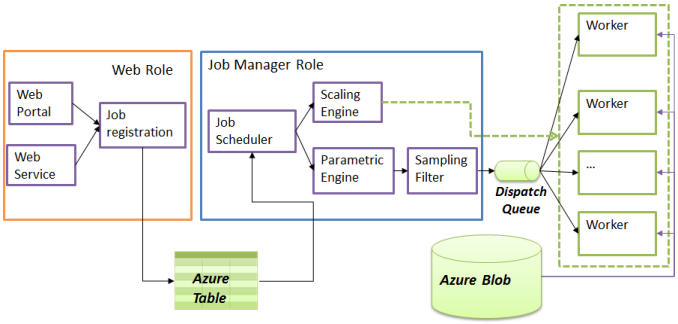

Figure 3. Overall Architecture of Cirrus

In addition to the job definition each job also can has its configuration, that specifies how the job will be executed in the Cloud. For example, in the job configuration the user can specify the minimum and maximum number of instances to run the job.

B. Overall Architecture

As shown in Figure 3, Cirrus consists of the following roles. The web role instance hosts the **job portal**. The user, authenticated via Windows Live ID service, submits and tracks his job through the portal. Once the job is accepted, all job-related information will be promptly stored in an Azure table, called the *job registry*, to prevent loss by a possible instance failure.

The **job manager instance** manages the execution of each parameter sweep job. The scheduler picks the candidate job from the job registry according to the scheduling policies (e.g., FCFS). Given a selected job, the manager will first scale the size of worker instances according to the job configuration. The parametric engine starts exploring the parameter space as a cross-product of the defined parameter variables. If the job is marked as a *test-run*, the parameter sweeping result is sent to the sampling filter for sampling. For each spot in the parameter space one task is generated and appended into one Azure queue called *dispatch queue* for execution. As the Azure Queue can store a large number of messages efficiently, we can easily handle a very large parameter sweep job. Each task is also associated with a state record in an Azure table called the *task table* and this state record is updated periodically by the worker instance running the task. The manager can monitor the progress of the execution by checking the task table. Moreover, with the task state information the manager can identify the tasks which has existed for 7 days and then circumvent the 7-day lifetime limitation by recreating the task messages for them.

The dispatch queue is connected to a farm of **worker instances**. Each worker instance keeps polling the dispatch queue for a message (i.e., task) and executes the task by running the materialized commands. During the execution it periodically updates the task state in the task table and listen for any control signals (e.g. abort) from the manager.

C. Job Reconfiguration and Resumption

In order to benefit from the elastic nature of the Cloud, in Cirrus we allow a running job to be suspended and resumed at a later time with an altered job configuration (mainly on the number of instances required). The manager suspends the running job by simply aborting all currently processing tasks that are marked as incomplete and then snapshot the task state table of this job. Consequently, the manager resumes the job by restoring its task state table and only re-executes the incomplete tasks. Without loss of generality we assume the results of all completed tasks have been stored in the Blob storage, therefore it is unnecessary to repeat the computation.

One application of this feature is fault handling for a long-running job. When the manager receives any fault message, it has at least two choices. The pessimistic choice is to abort the execution of all tasks followed by marking the job as failed. Although this solution seems to be intuitive, as we have shown in Figure 1 a large number of faults caused by the datacenter maintenance are automatically recoverable and aborting the entire job will interrupt all other instances as well and the intermediate results, which required potentially hundreds or thousands of instance hours, will be discarded, leading to a costly expense. The optimistic choice is that the manager ignores the fault and simply marks the task as incomplete. All running instances keep working without incurring any waste or context switch overhead. Once every task of the job either succeeds or is marked incomplete the manager terminates job execution as an incomplete one. Then the user can reconfigure it with a reduced instance number and retry the incomplete tasks. In this way, we can minimize the cost caused by failures.

Another application of this feature is using it as a solution to the load imbalance which will be explained in the next subsection.

D. Dynamic Scaling

In Cirrus there are three scenarios that will trigger the scaling engine. The first scenario is scaling an individual job. As previously mentioned, before executing a job the scaling engine will decide to scale out or in for this job based on the values of maximum/minimum instance number in its job configuration. The scaling engine conducts the scaling operation by sending a `ConfigurationChange` request to the Windows Azure management API. For each operation request, Azure will return a ticket with which we can use to check the result of the operation. The job manager can poll the status of the operation before dispatching the job tasks. This synchronous method guarantees that all needed instances have been instantiated before job execution. The overall progress of the job, however, may be delayed unnecessarily, particularly in the case of scaling-out due to the nontrivial overhead of instantiating new instances which also forces existing instances be idle-wait. Alternatively, the manager can start the job execution right way without waiting for the scaling operation, enabling the scaling and the job execution to be performed simultaneously. Apparently the asynchronous scaling-out minimizes job execution delay. However, the asynchronous scaling-in may cause some abnormalities which incur even more overhead. We will elaborate on this scenario late in Section V.

The second scenario is when the load imbalance is detected. In most load imbalanced runs, only a few instances are busy while most instances are idle. This situation can be detected by simply checking the aforementioned task state table. That is if most tasks are complete while a couple have been running for a long period of time, the job manager will automatically suspended the running job and reconfigure the job with a reduced instance number which can be set as same as the number of incomplete tasks. The suspended job is sent back to the job registry for re-scheduling and these long-tail tasks, which caused the load imbalance, will be re-executed using less worker instances. Notice that since that Windows Azure doesn't currently allow the user to specify the instance to shutdown in the scaling-in operation, our suspend-reconfigure-resume approach which may not be most efficient is actually quite effective.

The last scenario is when the manager has not received new jobs after a period of time. In order to save unnecessary instance idle time between two jobs, the scaling engine will automatically decrease the number of worker instances to a minimal size. The user also can mark a job as "shutdown-when-done" for which the scaling engine will immediately shutdown instances once the job is done. This is useful when the user submits a sequence of long-running experiments with the last one marked as "shutdown-when-done".

E. Performance Estimation through Sampling

To provide a reasonable estimate for the overall job execution time on the Windows Azure cloud, we adopt an observation based approach. That is the job manager randomly samples the parameter space according to the sampling percentage specified by the user and conducts a sample experiment. Since this observation based approach requires little in-depth knowledge of the experimental programs, it makes the estimation both general and independent. As the size of the sample set is supposed to be much smaller than the entire parameter space, the sample experiment can be conducted on a small instance set to save the cost. For most pleasingly parallel workloads where the tasks represent individual work items, the overall execution time shows a linear relationship to the number of independent tasks and the number of computation instances. Suppose the sampling percentage is α and the elapsed time of the sample run on n' instances is t', the running time of the entire job on n instances then can be estimated as:

$$t_e = \frac{t'}{\alpha}\frac{n'}{n} \tag{1}$$

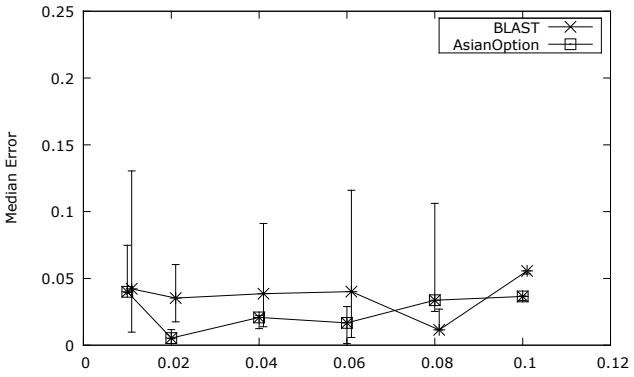

Figure 4. Accuracy of Sampling Execution

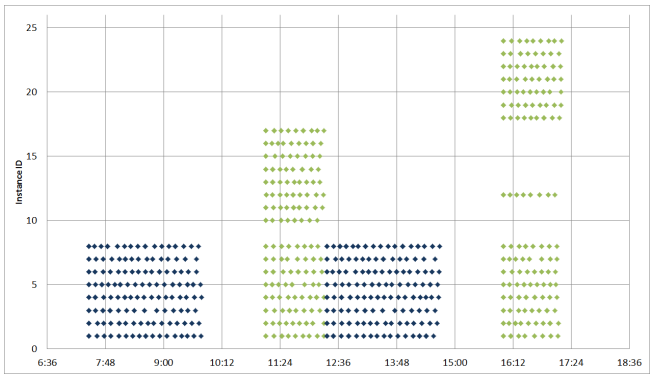

Figure 5. Synchronous Dynamic Scaling

When the tasks are uniform in execution times, estimation by sampling will be highly accurate. On the other hand when individual tasks yield non-uniform execution times, the sampling percentage has to be increased to get a good estimate. Since the rate of the instance hour is generally much higher than other resources (e.g., storage) and most science experiments are computation intensive, the user can roughly estimate the cost from this performance estimate.

V. Experiments & Evaluation

We first evaluated our sampling-based estimation with two experiments. The first is the BLAST job whose script is listed in Section IV. This job will query about 5000 protein sequences, evenly divided and stored into 200 blob files, against a large protein database. The second application is the AsianOption simulation program, a Monte Carlo simulation on the stock market price [10]. The variable parameter of this job is the risk-neutral probability, which is defined as a range of double numbers. We first run two experiments completely with 16 instances and recorded the elapsed time as the reference time. Then we partially run them on two worker instances with varied sampling percentages several times to estimate the overall execution times. The median of estimation error together with the associated error bars are plotted in Figure 4.

In general, our estimate using sampling execution yields accurate results at a low cost. In case of the BLAST experiment, a complete run takes 2 hours with 16 instances, while a 2%-sampling-run which achieves 96% accuracy only takes about 18 minutes with 2 instances. In this case, the overall cost for the sampling run is only 1.8% of the cost of the complete run. The Asian option simulation experiment generated better estimate accuracy. A 1% sampling run can achieve 96% accuracy. Compared with the BLAST experiment, the AsianOption experiment is also less sensitive. That is because the choice of the parameter value has little impact on the computation complexity of the Monte Carlo simulation while the complexity of a BLAST query is much more intricate and less uniform.

Next we evaluated the behavior of dynamic scaling on Windows Azure by conducting an "accordion" scaling experiment. That is we submit the aforementioned BLAST job to the Cirrus service multiple times; two consecutive submissions alter the required instance number between 8 and 16. The size of worker role instance used in this experiment is extra-large for the best throughput of the BLAST run. The job script is optimized to avoid repeating database staging by consecutive jobs. We first conduct this experiment in a synchronous scaling manner. The entire job log is visualized in Figure 5. As we see, with 16 instances the effective job execution time is about 75 minutes and with 8 instances the execution time is 140 minutes, thus the system provides good scalability. However, during the synchronous scale-out operation, all instances including those existing ones have to sit idle for roughly 80 minutes, which is a considerable waste of resources. From the view of the user the job actually took 155 minutes, which is even more than the time required by 8 instances, not to mention that the user will be charged more for the 16 instances. In contrast, the synchronous scale-in operation takes place promptly.

In the next experiment we conduct the same experiment, but in an asynchronous scaling manner. From the log visualization in Figure 6, we can see that during the scale-out all existing instances keep working as usual while new instances join the ensemble. These newly-created instances start their execution at different times, ranging from 20 minutes to 85 minutes after the scaling operation is issued. Nevertheless, the asynchronous scale-out clearly minimizes instance idle time. The execution time with the dynamically scaled 16 instances is 100 minutes, 1.4 times faster than the run with 8 instances. For a job that requires longer execution time; this constant overhead can be mitigated. For the asynchronous scale-in operation, however, we found there is a red dot which identifies a task loss at the end of the log of each instance that is going to be shut down by the scale-in operation. This is because the tasks are dispatched at the same time when the scaling-in request is

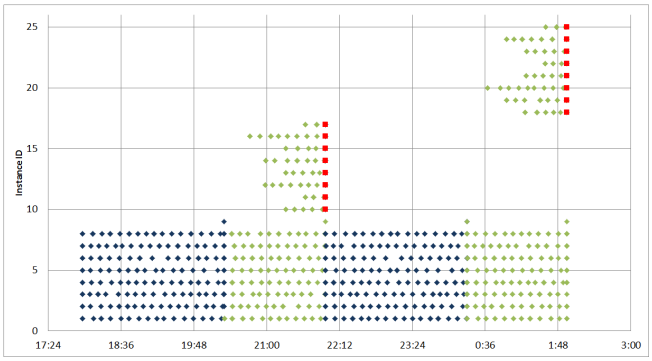

Figure 6. Asynchronous Dynamic Scaling

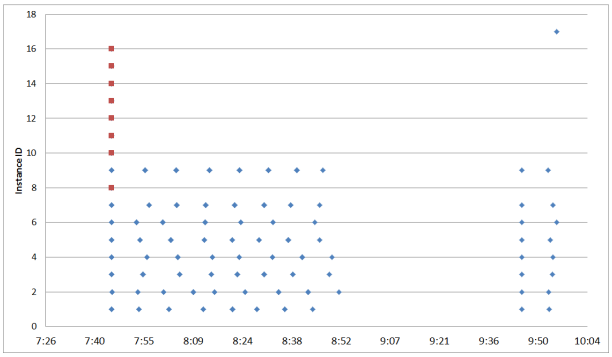

Figure 7. Potential cost of Async. Scaling-in

sent, those instances which will be shut down by the scaling-in request have got the task messages from the queue. As a result, these retrieved messages are lost. Although these lost messages will re-appear in the queue after the visibilityTimeout period, these existing instances may need to idle-wait for the visibilityTimeout period until the lost messages are once again visible. Figure 7 shows this worst case in which the same BLAST job but with a smaller input data is executed. As we can see, 8 existing instances finish all visible tasks in 60 minutes; however these lost tasks have not reached the visibilityTimeout, which was set as two hours, so all instances have to sit idle for one more hour until the lost tasks came back in the queue.

Based on the above observation, we conclude that best practices are that instance scale-out is best done asynchronously to minimize idle instance time while instance scale-in is best done synchronously to avoid the unnecessary message loss as well as the potential idle instance time.

VI. Related Work

Running parameter sweeping/task farming applications on computational grids has a large body of prior work. The most related one to our work is Nimrod/G [11], which is a parameter sweeping runtime for Grids with special focus on the computational economy. Unlike Cloud, Grids are basically the federation of shared, heterogeneous and autonomous computational resources. Thus in order to meet the economy requirement (e.g. budget) for running an experiment on Grids, Nimrod/G introduced several deadlines and budget constrained scheduling algorithms for resource discovery, trade and allocation. The scheduling algorithm relies on a modeling-based performance/cost estimation, in which the costs of grid resources are artificially assigned into the so called Grid dollars. AppLes [12] is another highly cited parameter sweep system for the Grids. Instead of focusing on cost/deadline, AppLes is more interested in maximizing the throughput on Grids. Since the Grid resource could be scattered across the entire Internet, the remote data transferring cost can be the dominant performance issue. Therefore one important consideration of AppLes scheduler is the efficient data collocation. AppLes optimized that by adaptively scheduling the task into a hosts-network Gantt chart which relies on the performance estimates collected by NEWS (Network Weather Service).

Compared with those Grid-based systems, Cirrus is much more straightforward as on Cloud many concerns become unnecessary. For example thanks to the homogeneous and exclusive virtual computational resource provided in Cloud the resource modeling, discovery and matching are needless. Further since the resource cost has been quantified in a normalized way, there is no need to model the resource cost in an artificial way. Also due to the pay-as-you-go model and the resource elasticity the job scheduling has more flexibility to meet various constraints, especially the deadline and budget ones.

A large number of science applications have been successfully run on Cloud. Some noticeable examples include the biology sequence alignment [13], high-energy and nuclear physics simulation [14] and geology data processing [15]. We have realized most of them present a very similar architecture and thus can be generalized in the parameter sweep pattern. Simmhan et al. propose a general worker solution [16] which eases the migration of the desktop application onto cloud. But their solution focuses on the sequential execution of a single executable.

VII. Conclusion

In this paper we examined how cloud based architectures can be made to perform large-scale science experiments. Starting from an analysis of our long-running BLAST experiment on Windows Azure we identified the top challenges (e.g., failure, load imbalance), which generalize to a large number of science applications when running in the cloud. We introduce Cirrus, a parameter sweep service on Azure designed to address these challenges. When compared with parameter sweep engines on Grids the implementation of Cirrus is quite simple due to the homogeneous virtual resources and automatic fail-over feature provided by Windows Azure. And, with the dynamic scaling feature Cirrus offers a more flexible way to optimize the resource

allocation. Cirrus is also highly attuned to cost-effectiveness because of the unique Cloud "pay-as-you-go" model. The job reconfigure-resume pattern is a cost-effective solution to handle these situations, such as failures and load imbalance, which are inevitable when running such large-scale experiments. And sampling-based performance estimation enables users to make informed decision, thus greatly improving the overall cost-effectiveness of Cloud usage.

REFERENCES

[1] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, "Above the clouds: A berkeley view of cloud computing," Feb 2009.

[2] D. Chappell, "Introducing windows azure," DavidChappel & Associates, Tech. Rep., 2009.

[3] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman, "Basic local alignment search tool," *Journal of Molecular Biology*, vol. 215, no. 3, pp. 403 – 410, 1990. [Online]. Available: http://www.sciencedirect.com/science/article/B6WK7-4N0J174-8/2/37c69feb1cd9b63368705c2f5f099c5b

[4] R. C. Braun, K. T. Pedretti, T. L. Casavant, T. E. Scheetz, C. L. Birkett, and C. A. Roberts, "Parallelization of local blast service on workstation clusters," *Future Generation Computer Systems*, vol. 17, no. 6, pp. 745 – 754, 2001.

[5] W. Lu, J. Jackson, and R. Barga, "Azureblast: A case study of cloud computing for science applications," in *The 1st Workshop on Scientific Cloud Computing*, Chicago, Illinois, 2010.

[6] NCBI, "Ncbi-blast," http://blast.ncbi.nlm.nih.gov/Blast.cgi.

[7] J. Varia, "Architecting for the cloud: Best practices," Amazon, Tech. Rep., 2010.

[8] Microsoft, "Windows azure queue," Microsoft, Tech. Rep., 2008.

[9] D. Abramson, R. Sosic, J. Giddy, and B. Hall, "Nimrod: A tool for performing parametised simulations using distributed workstations," in *4th IEEE Symposium on High Performance Distributed Computing*, 1995.

[10] S. Benninga, *Financial Modeling*. The MIT Press, 2008.

[11] R. Buyya, D. Abramson, and J. Giddy, "Nimrod/g: An architecture for a resource management and scheduling system in a global computational grid," in *Proceedings of the 4th High Performance Computing in Asia-Pacific Region*, 2000.

[12] H. Casanova, G. Obertelli, F. Berman, and R. Wolski, "The apples parameter sweep template: User-level middleware for the grid," *Sci. Program.*, vol. 8, no. 3, pp. 111–126, 2000.

[13] J. Wilkening, A. Wilke, N. Desai, and F. Meyer, "Using clouds for metagenomics: A case study," *Proceedings IEEE Cluster*, 2009.

[14] K. R. Jackson, L. Ramakrishnan, R. Thomas, and K. Runge, "Seeking supernovae in the clouds: A performance study," in *1st Workshop on Scientific Cloud Computing*, Chicago, Illinois, 2010.

[15] J. Li, M. Humphrey, D. Agarwal, K. Jackson, C. van Ingen, and Y. Ryu, "escience in the cloud: A modis satellite data reprojection and reduction pipeline in the windows azure platform," apr. 2010, pp. 1 –10.

[16] Y. Simmhan, C. van Ingen, G. Subramanian, and J. Li, "Bridging the gap between desktop and the cloud for escience applications," in *IEEE Cloud*, 2010.