

Fault Tolerance and Scaling in e-Science Cloud Applications: Observations from the Continuing Development of MODISAzure

Jie Li, Marty Humphrey
Department of Computer Science
University of Virginia
Charlottesville, VA USA

You-Wei Cheah
School of Informatics and Computing
Indiana University, Bloomington
Bloomington, IN USA

Youngryel Ryu
Dept. Environmental Science, Policy
and Management
University of California, Berkeley
Berkeley, CA USA

Deb Agarwal, Keith Jackson
Advanced Computing for Science Department
Lawrence Berkeley National Lab
Berkeley, CA USA

Catharine van Ingen
eScience Group
Microsoft Research
San Francisco, CA USA

**Abstract*—It can be natural to believe that many of the traditional issues of scale have been eliminated or at least greatly reduced via cloud computing. That is, if one can create a seemingly wellfunctioning cloud application that operates correctly on small or moderate-sized problems, then the very nature of cloud programming abstractions means that the same application will run as well on potentially significantly larger problems. In this paper, we present our experiences taking MODISAzure, our satellite data processing system built on the Windows Azure cloud computing platform, from the proof-of-concept stage to a point of being able to run on significantly larger problem sizes (e.g., from national-scale data sizes to global-scale data sizes). To our knowledge, this is the longest-running eScience application on the nascent Windows Azure platform. We found that while many infrastructure-level issues were thankfully masked from us by the cloud infrastructure, it was valuable to design additional redundancy and fault-tolerance capabilities such as transparent idempotent task retry and logging to support debugging of user code encountering unanticipated data issues. Further, we found that using a commercial cloud means anticipating inconsistent performance and black-box behavior of virtualized compute instances, as well as leveraging changing platform capabilities over time. We believe that the experiences presented in this paper can help future eScience cloud application developers on Windows Azure and other commercial cloud providers.

Keywords—cloud computing; eScience; Windows Azure

* DISCLAIMER

This document was prepared as an account of work sponsored by the United States Government. While this document is believed to contain correct information, neither the United States Government nor any agency thereof, nor the Regents of the University of California, nor any of their employees, makes any warranty, express or implied, or assumes any legal responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by its trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof, or the Regents of the University of California. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof or the Regents of the University of California.

I. INTRODUCTION

The increasing availability of scientific data from both large experimental instruments and networks of ground-based and space-based sensors is transforming the scientific computing landscape. Today, eScience applications at tera-byte or even peta-byte scale are not uncommon in domain areas such as Environmental Science and High Energy Physics [1]. The continuously generated data sets in these domains have created invaluable repositories for new science discoveries.

However, with the increasing scale of data sets comes the problem of how to process them. Acquiring the computational resources necessary to meet the requirements of these large scale applications is difficult for many science research groups. At the same time, scientists that have access to high-performance clusters or supercomputing centers often experience indefinite job turnaround times due to resource-sharing policies that cause long queue wait times. This inefficient situation becomes more severe when scientists need to frequently submit large jobs to test and improve their scientific models. There are also huge process gaps going from large-scale raw data to scientific knowledge. Complexities involved in data collection, storage management, computation scheduling, and fault-tolerance increase as the scientific computation scales up. These complexities, if not correctly handled by software systems and appropriately masked from domain scientists, can cause invalid results in the final scientific knowledge or make an analysis impossibly daunting.

Cloud computing has recently emerged as a large-scale, distributed computing platform with on-demand resource provisioning capability and a “pay as you go” service model. Commercial cloud providers such as Google App Engine [2], Amazon EC2 [3], and Microsoft Windows Azure [4] are pioneering in this area providing hosting of cloud applications on powerful datacenters. Although the specific resource provisioning and programming models are different, all provide a common set of building blocks in the form of compute and storage services to enable large-scale, compute- and data-intensive applications to be developed and deployed. The on-demand nature of these compute and storage services allows applications to scale by orders of magnitude without changing the underlying software components and architectures or incurring long wait times. The “pay as you go” model can save users huge upfront investments in the hardware resources. Cloud computing has the potential to bring benefits to the scientific community by hosting large-scale eScience applications on-demand.

However, cloud computing is still in its early development stages and is relatively immature. The distinct capabilities and characteristics of cloud computing need to be fully explored and evaluated before we can confidently adopt cloud computing for existing and next-generation eScience applications. In this paper, we present our experience developing and using MODIS Azure, a large-scale satellite data processing system built on the Windows Azure cloud computing platform. Because we built this application from scratch, we leveraged a large subset of the Azure cloud service

elements as the basic building blocks for our application components. To our knowledge, this is one of the earliest scientific applications that were *natively* built on top of the cloud infrastructure and service elements provided by Windows Azure.

We have already presented the basic architecture of the MODIS Azure system and its overall performance in a previous publication [5]. However, the work we presented in that publication was completed during summer 2009, when Windows Azure was still on the Community Technology Preview (CTP) phase. Since then, Windows Azure has been evolving and bringing new important features for the development of cloud applications, and we have incorporated some of these new features into our system design (e.g. using the Management API for dynamic instance scaling). Furthermore, we have been running this scientific computation for more than 9 months since we submitted that paper, and have gained experiences and insights from running it in the cloud. Therefore, in this paper we aim to present the following unique contributions:

1. We introduce the design and implementation of a native cloud application for parallel data-intensive scientific computing, and identify some of the key principles for developing a scalable cloud science application, such as queue service based task scheduling and fault tolerance;

2. We discuss the advantages and limitations of current cloud services, such as on-demand scaling capability and VM hosting reliability, in the context of Windows Azure but the lessons are largely applicable to other cloud computing platforms;

3. We present our experiences running and scaling up this scientific application to encompass larger data sets and we evaluate the issues encountered.

The remaining of this paper is organized as follows: Section II provides the scientific background of this application; Section III introduces the overall design and implementation of our software architecture; Section IV discusses our evaluations and experiences running and scaling this application; Section V presents related works; Section VI concludes our work.

II. BACKGROUND

Terra and *Aqua* are two remote sensing satellites, each carrying a Moderate Resolution Imaging Spectroradiometer (MODIS) that measures and images the entire Earth’s surface in 36 spectral bands and at multiple spatial resolutions [6]. These two MODIS satellites can provide accurate and timely data about the global dynamics and processes occurring on the land, oceans, and atmosphere, as they orbit the entire Earth surface every 1 to 2 days. The MODIS data is made available on multiple FTP sites.

The science goal of our project is to apply the MODIS data in the calculation of the evapotranspiration (ET) on the earth surface. ET controls land-atmosphere feedbacks and constitutes an important source of water vapor to the atmosphere. In turn, atmospheric water vapor is the most significant greenhouse gas

and key to understanding hydrologic water balance. As such, ET plays a fundamental role in weather and climate. Our computation of ET involves multiple science data products from the MODIS source data sets, each containing a specific type of earth surface imagery (such as Land surface temperature or atmospheric aerosol). Each data product contains a number of related science variables separated into HDF format source files [7]. The size of a single data product for each global year ranges from several hundred GBs to over 1 TB. Although files can be downloaded from the FTP sites, currently there are no public software frameworks available in the earth science community to automate the processes of reconciling and cataloging the various data products and/or scheduling parallel computations over these data. As a result, scientists need to handle processing complexities manually, which often prevents large scale analyses.

To lower the barriers for scientists to perform large-scale scientific computations on the MODIS data, we proposed and implemented a cloud-based parallel data processing system in Windows Azure. Our parallel data processing system, MODISAzure, can completely automate the processes involved in metadata management, source data downloading, computation task scheduling, and fault-tolerance. Furthermore, the on-demand resource provisioning capability of cloud computing allows us to smoothly scale our ET computation from a small regional level to the global level. During a 9-month system operational period, we have successfully completed computations at 3 different geographical scales. The first scale of computation covers the US continent and uses 15 *sinusoidal tiles* (A sinusoidal tile is a piece of the globe mosaic which covers a relatively small unit of geographic area). The next scale of computation covers the set of regions that have FluxNet [8] towers on their land as we are using FluxNet tower data to validate our science results. Computation at this scale accounts for another 32 sinusoidal tiles. Finally, we calculate the ET at the global scale, which includes totally 193 sinusoidal tiles. Table 1 gives a view of the data management requirements in terms of number of data files and data sizes. The numbers in the columns are for a single *satellite year* (i.e. the scale of data from a single satellite in a single calendar year). The numbers in the parentheses in the first column indicate the total satellite years for each scale that we have incorporated in our computation. One thing worth noting is that the numbers in the *Result Size* column do not increase/decrease proportionally with the different scales, because during the 9-month operational period, we have developed several methods along the way to reduce the result data size for our computation in order to save storage cost. We will discuss these methods and the result savings in details in section IV.

TABLE I. DATA REQUIREMENTS AT DIFFERENT SCALES

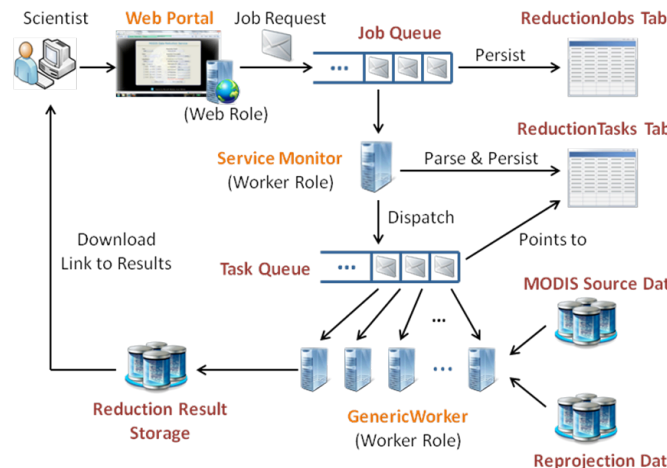
III. SYSTEM DESIGN

A. System Overview

MODISAzure is a loosely coupled, component based parallel data processing system running in Windows Azure. Different components of the MODISAzure system are

developed based on the various resource abstractions and services provided by Windows Azure. Windows Azure provides two types of virtualized compute instances as differentiated by *roles*: the *Web Role* instances are Windows-based virtual machines hosting web applications on IIS; the *Worker Role* instances are background Windows virtual machines for running customized user code. The MODISAzure system consists of two main service components running on these two different types of compute instances: The first service component is a front-end web portal for user job submission and execution status monitoring. This component is a Microsoft Silverlight-based web application which is hosted on a web role instance. The second component is a back-end computing system hosted on a number of worker role compute instances. It includes three main data processing stages, which can be either pipelined or run independently. Each stage performs a specific type of data-processing task:

1. In the *data collection* stage, a set of compute instances download the MODIS source data from external FTP sites to local storage, and then upload the data to the *blob storage*, a persistent storage service for large scale unstructured data in Windows Azure. The source data are stored in the blob storage



for computations in the next two stages. To download the target source data set for a specific date/area, a compute instance first queries the geo-spatial information about the target source data against a source metadata table, and then goes to the specific FTP location indicated by the metadata to fetch the data. The metadata table is built using the Azure *table service*, which is the persistent storage for structured data.

2. In the *reprojection* stage, a set of heterogeneous data products collected in the first stage are reprojected into time- and spatial-aligned imagery data. A set of compute-intensive algorithms (such as nearest neighbor pixel matching) are performed in this process when matching or adjusting the data points pixel by pixel across the source data files. The

	#Source Files	Source Size	# Result Files	Result Size
USA (18)	21850	238 GB	27375	261 GB
FluxTower (3)	80670	993 GB	58400	210 GB
Global (3)	152670	2414 GB	352225	630 GB

reprojected result data will then be uploaded to the blob storage for use in future scientific analysis.

3. In the *reduction* stage, a number of compute instances invoke a reduction executable uploaded by the scientist to perform the analysis computation over the reprojected data from the previous stage. The executable can be compiled from any source code, such as C/C++, MATLAB, etc.

A more detailed description of the above data processing stages as well as other components of MODISAzure can be found in [5]. Besides the front-end web role instance and a number of worker role instances running tasks for the above 3 stages, there is also one dedicated worker role instance running as the *service monitor*, whose main responsibility is to process job requests from scientists as well as monitor and manage the execution progress. As an example, the processes involved in a reduction stage computation are shown in figure 1: A scientist submits a reduction job request which specifies the parameters for the computation and the reduction executable to be uploaded for the reduction computation. The computation parameters specify the data scope, which identifies interested MODIS data sets that cover a specific date period and geographical area, as well as a set of parameters that need to be passed into the reduction executable for execution. The job request is then sent to a job queue, which is implemented on the Azure Queue Service. When the service monitor gets the request from the job queue, it parses this specification and separates the job into a number of embarrassingly parallel tasks where each task performs reduction computation for a single day on a single geographic unit (i.e. a single *sinusoidal tile*). These tasks are sent to a *task queue*, from which a number of worker role instances keep pulling the tasks (discussed in section III.B). For each dequeued task, a worker role instance will first download the reduction executable which has previously been uploaded by the scientist from the Azure blob storage to local storage. It then invokes the executable with parameters specified in the task to perform corresponding reduction computation. During the execution, information and status about the task computation will be persisted and updated in corresponding tables, which are implemented on the Azure Table Service (discussed in section III.E). After the execution, the results (i.e. output files from the execution) will be uploaded to the Azure blob storage for persistence. Finally, when all the tasks for a job request are finished, a single download link will be sent by the service monitor via email to the scientist who submitted the request.

Figure 1. Processes in a reduction stage computation.

B. Task Scheduling and Execution

The task scheduling and execution model of MODISAzure is based on the Windows Azure *queue service*, an asynchronous message-based communication service. Parallelized task items are wrapped into XML format messages and sent to the task queues. Each compute instance pulls task items from the queues and invokes corresponding data processing code for different types of tasks (data collection, reprojection, and reduction).

We implemented a Generic Worker task execution framework similar to the one described in [9]. In this execution framework, every compute instance is capable of executing all types of tasks. In other words, we don't deploy multiple types

of worker role instances in the system and assign a specific type of task for each instance type. This execution model helps eliminate the potential load imbalance between the instances when working on different types of tasks from the queues. Also, it is flexible enough to support a new task type in the system without modifying the underlying service architecture. The new task processing code can be added to the framework in the form of source classes in C#, compiled libraries or executables. They will be packaged together with the service deployment to be hosted on every compute instance.

In retrospect, the combination of the queue-based task dispatching and the task pulling model was the key to achieving software scalability and flexibility in our system. Instead of the task pushing model, there is no need for a central job scheduler in charge of managing and assigning tasks to different worker instances. Every worker instance is self-managing, and thus can dynamically enter or leave the computation resource pool. This in turn enables compute instances to be dynamically scaled up/down without impacting any of the service components as well as implicitly load balancing work across instances. This on-demand resource scalability brought by cloud computing allows us to scale from a small regional computation up to the global level computation without any changes to the software components.

C. Dynamic Instance Scaling

Dynamic instance scaling is the ability to adjust the number of compute instances for a cloud service. As discussed above, the loose-coupling and self-managing paradigm of our Generic Workers allows us to dynamically scale up/down the number of compute instances according to the real-time workloads from scientists' job requests, so as to balance the cost and responsiveness. Dynamic instance scalability can be achieved by invoking the Azure Management API to update the service configuration for a deployed application, which specifies the number of compute instances for each type of web/worker role. In MODISAzure, an independent component is deployed on the *service monitor* instance to monitor the real-time job requests submitted by scientists. When there are no job requests submitted to the queue, the service monitor reduces the number of compute instances to a minimum number to maintain service availability; Upon the submission of a new job request, the service estimates the total computational requirements for this request, calculates the number of new instances that need to be started to work on the computation based on the criteria of turnaround time, and invokes the Management API to adjust the number of instances accordingly.

Given the many complexities and issues involved in the cloud resource provisioning infrastructure and service model, we have currently identified several limitations and performance issues of the instance scaling capability in Windows Azure:

1. Instance start time overhead: The time delay for new compute instances to start is significant [10]. For dynamic instance scaling, we've observed more than 30-minute start time for new instances;

2. Lack of fine-grained control: Concurrent and orthogonal instance add/remove operations are not supported within a

service deployment. Worse, when scaling down compute instances, it is infeasible to specify which instance(s) to shut down.

3. Cost efficiency: Compute instance usage are charged in full hours, which means a 10-minute instance up time is charged the same as a 60-minute instance up time. Therefore, frequent instance start/shutdown may cause low cost-efficiency.

D. Fault tolerance

At the scale of over a quarter million tasks and tiles in a single job request, even rare failure events pose problems that can take significant human effort to understand and repair. A significant amount of time and effort has been devoted to identifying these failures and ensuring that the service is reliable and robust enough to automatically handle the various failures that we have faced. These failures stem from both the data scale of our application and also the characteristics of the cloud environment. We categorize the types of failures that we encountered into the following two classes:

1. **Data Failures:** Failures that are caused by flaws in the data, such as corrupted data content, missing source data, etc.

2. **Computation Failures:** Transient hardware or infrastructure failures, such as slow instances, storage access exceptions, etc.

Due to the different causes and situations involved in these different types of failures, we found it necessary to enforce fault tolerant policies differently for the above two categories.

For data failures, the errors are often domain-specific, thus these failures require the scientist to incorporate fault tolerant logic into the scientific code. Although flawed data takes up a small fraction of the datasets that we have, the consequences are severe at large scale as they may cause software failures and invalidate the results of scientific experiments.

Current cloud infrastructures are built on top of commodity hardware, applications running in the cloud are prone to hardware and software failures. Computation failures are typical at the service infrastructure level. Some of the examples of these failures are slow Virtual Machine (VM) instances and storage access exceptions. A typical fault tolerant solution to overcome these failures is to implement a recovery strategy by retrying the task execution. In our service, we have implemented a customized task retry policy. For every task that times out or fails, a task is terminated and placed back in the service queue to be retried. This is performed for a certain number of retries (three times by default), before the service declares it as a failure.

E. Job Status Monitoring and Logging

Monitoring is critical for tracking and diagnosing the execution status and problems of the numerous tasks in MODIS Azure. Since the number of tasks for a single job ranges from several hundred to over a quarter million, it is important for us to record this vast amount of information in such a way that it can be used effectively and efficiently. The

Azure table service is used because it provides a structured data store that is scalable yet supports querying in an easy manner. Separate tables (such as ReprojectionTaskStatus and ReductionTaskStatus, etc.) are used to record the specification, execution status and exception messages for each kind of computation. Data from the monitoring and logging components are mainly used in one of two ways.

The first way the data is used is via online job execution monitoring and analysis. This is through a status monitoring interface on the web portal that retrieves task execution information from the corresponding TaskStatus table. The execution progress and statistics for any computation task can be retrieved in real-time by providing a unique job ID. Other helpful information such as the standard output and error output from the invocation of reduction executables are also provided. Through these information, scientists are able to better diagnose and debug the various problems for their executable during the development phase. The status of each task is also tracked for fault-tolerance and failed tasks are first handled by issuing a certain number of retries before finally declaring it as a failure.

We also mine the data offline. Since the table services do not provide the capability of performing complex statistical analysis over the data, we download the logged records from the tables and place them in a SQL database. By building an OLAP data cube over these data, we are able to perform richer statistical analysis across various dimensions. Comprehensive views of billing records, task status and storage consumption across time are examples of how logging records are used. Support for analyses of this kind would be impractical if implemented on the Azure tables, and we are currently considering SQL Azure for this mining.

IV. EXPERIENCES AND EVALUATION

We have been developing and operating MODIS Azure since summer 2009. In this section, we relate our experiences developing and operating the application and present evaluations of reliability and quality of service of the application running on Windows Azure.

A. Experience

There have been four distinct phases in our work with MODIS Azure: early development (7/2009 to 9/2009), Continental USA computation (10/2009 to 3/2010), Fluxnet tower computation (3/2010 to 4/2010), and global computation (5/2010 to present). Each phase is characterized both by computational scale and science data challenges. In addition to the data processing system described in Section III, we also developed a reprojection algorithm and evolved our ET science reduction algorithm. Each of these three software components have evolved relatively independently as our computation has scaled and we learned from experience.

1) Early Development

We began by prototyping the reprojection algorithm in MATLAB on the desktop and sizing the source data based on an initial list of the source MODIS products. Because we had no way of guessing how many times we would run the ET

reduction, we simply guessed that the total load would be no more than double the reprojection. This enabled an early capacity plan for the upload, storage, and compute requirements. We continue to refine those estimates and monitor our current usage. That gives us an ability to plan ahead as well as simply identify potential problems by comparing our estimate with the observed behavior.

We chose a simple, fast nearest neighbor algorithm to convert the tiles available only in the MODIS swath projection to the MODIS sinusoidal projection. This ensures that all data for our ET reduction are both time and space-aligned with an equal-sized land surface pixel [11]. We implemented the algorithm in C# to give better performance. We also designed the geo-spatial lookup necessary to identify the swath projection tiles necessary to create a given sinusoidal tile. We used a SQL Server database for initial development, but ported the results to Azure tables.

We chose to support compiled MATLAB code for the ET reduction algorithm. The simple science desktop debug and development capability as well as the availability of relevant support libraries more than compensates for any reduced performance.

II) USA computation

In late fall 2009, we achieved our first one year USA ET result. A key learning lead to adding the optional second stage science reduction to the pipeline. This second stage is used to produce science analysis artifacts such as maps, virtual sensors, or plots from the reduction computation. When reducing at scale, downloading the reduction results and then producing these artifacts on the desktop can be onerous.

In January 2010, we moved from our Azure Community Technology Preview (free, pre-release) account to a commercial account. We could now monitor our resource usage and billing at the Microsoft Online Services Customer Portal [12]. We began to dynamically scale our deployment to keep our running costs down.

We started the practice of comparing our billed compute hours with our TaskStatus tables in February 2010. Figure 2 shows that comparison over time; our task logs account for 85% of the billed compute hours. As expected, we observe very good agreement when running a large number of tasks consistently such as during the cpu-intensive USA and FluxTower reprojection. Each tile took approximately 2.5 minutes of which ~0.4 minutes were spent in overhead staging tiles to/from the Azure VM instance and the blob store. Also as expected, we see less agreement when dynamically scaling instances during reduction such as on 4/17/2010. We keep “idle” instances running after task completion (currently 15 minutes) to avoid the need to stop/start instances unnecessarily in case of frequent job submissions and as Azure billing rounds up partial compute hours.

We also started monitoring our storage billing. Unlike many grid platforms, Azure billing includes upload, download, and storage fees. For science convenience, most of the MODIS source tiles contain multiple science variables in a single file; we estimated that keeping only the variables needed for the ET computation would save us approximately 60% of the storage

required for reprojected tiles. Since these represent over 90% of the source data, we felt this was an important savings.

We gained experience operating the service at scale in this phase. We learned that we benefitted from retrying each download, reprojection, and reduction task to reduce the impact of intermittent Azure disruptions. If the task continues to fail, we can then use the logged status return to triage the failures and investigate to attribute the failure as discussed in Section III.D. Note that triage does not always tell us exactly what caused the error – we do not distinguish missing tiles caused by a satellite outage from missing tiles that are simply not present on the NASA site – but does tell us what we can do to get an ET reduction result.

III) FluxTower Computation

Since our USA computation was beginning to give good science results, we decided to expand the computation to include the additional sinusoidal tiles covering 114 additional eddy flux towers in late 3/2010. We expected and experienced simple scaling. Our capacity planning estimated that one FluxTower year corresponded to about 2 US satellite years (32 tiles vs. 15 tiles). That resource scaling was very close - we actually consumed about 18 US satellite years. Our decision to automatically retry failed tasks served us very well in this phase; we saw approximately 6% task failure out of 57664 tasks attempted. Of those, 41% were recovered by retry. The remaining 59% unrecoverable tasks were mainly caused by data failures or scientist code bugs.

The increased data diversity presented challenges to our ET algorithm. We encountered a much wider range of biomes such as rainforests and climate regimes such as tropics. We found that we now needed additional science variables from the imagery; some of the layers we had previously discarded now needed to be retained.

IV) Global Computation

We first attempted an ET computation on a global scale for a single calendar year in April 2010. Based on early success,

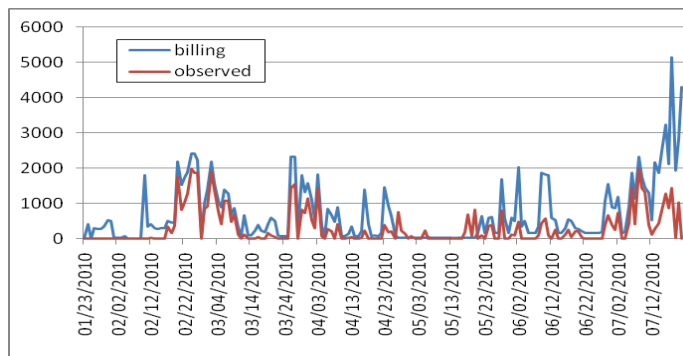


Figure 2. Comparison of billing compute hours per day with observed reprojection and task times. The two correlate best when the reprojection was compute bound and the number of deployed instances is relatively static.

we started the initial download and reprojection for two additional calendar years in July 2010.

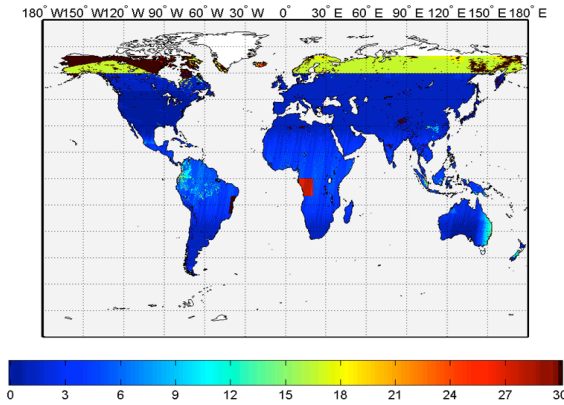


Figure 4. Global data availability for the ET reduction in February. Color coding indicates data availability; white areas were not included in the computation.

We chose a 5 KM rather than 1 KM spatial scale based on capacity planning. The USA represents approximately 5% of the world land surface area, so we were attempting to scale up by a factor of 20. Scaling down the resolution meant that 1 US year is approximately 1 global year. As shown in Figure 3, this decision most strongly impacted the storage requirements. At the end of the global reprojection, we deleted all global source tile precursors for the calendar year 2003 and two extra years of FluxTower tiles. Prior to that “storage diet”, our storage bills were approximately half of our total bill.

The 5 KM choice shifted our reprojection from compute bound to slightly IO bound. Processing each tile now took approximately 5 minutes of which ~2.6 minutes were spent in overhead staging tiles to/from the Azure VM instance and the blob store. This change is apparent in Figure 3. The billed storage transactions are negligible in the early phases, but closely track the compute hour billing for the global reprojection. We also observed over 10X variation in the reprojection task time. The MODIS satellites cross a given sinusoidal tile location more often at the poles than at the equator and the number of nearest neighbor pixels increases dramatically. We simplified the algorithm to reduce the search space across the source files and thereby reduce the overhead.

The choice also impacted the scaling of our reduction phase. We observed that a one year global ET reduction job took approximately 6 hours. To decrease that, we increased our Azure quota from 100 instances to 250 instances. Unfortunately, we saw change in wall clock time due to more than double the overhead associated with staging the tiles to/from the Azure VM instance. We speculate that we reached a rack cross-sectional bandwidth limit. We have examined our task logs, computed and therefore discounted the number of table and queue updates; much of Azure remains a black box.

We also learned the importance of having a complete tile catalog including all source tiles on the NASA ftp sites, reprojected targets, and known expected missing tiles as well as our TaskStatus logs. Our Service Monitor experienced an Azure VM restart in the middle of scheduling the tasks for a global reprojection job with over 240K tasks. At the same

time, our download tasks were failing intermittently due to NASA site outage. Retrying both eventually generated 95% of the tiles needed for the ET reduction. We then had to track down the missing 5%. Figure 4 shows one of our maps from that analysis. Causes included missing tiles on the NASA site such as on the coast of Africa, winter polar nights, and (not shown) satellite outages.

Lastly, we continue to evolve the science computation and validation. Understanding how to think about regions such as the Sahara and the implications for crop fertilization remains

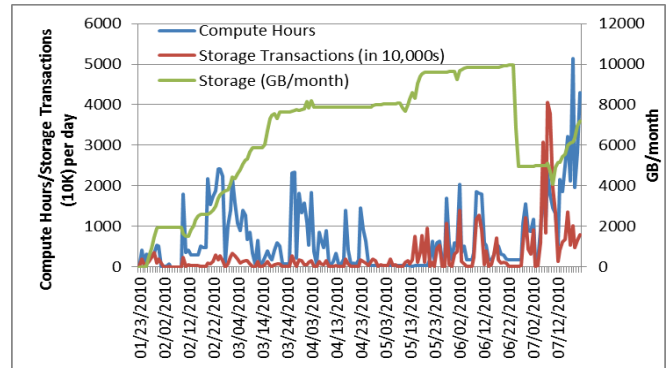


Figure 3. Compute hours, Storage transactions, and Data Storage. The US and FluxTower 1 KM reprojection was compute bound; the reduction and 5 KM reprojection tend to be IO bound. We transferred 3 US years from the CTP account in January 2010. The dramatic decrease in storage at the end of June was due to deletion of the global computation swath tiles after successful reprojection.

active science research. That our pipeline is running well allows us to focus on that science.

B. Evaluation

1) Performance Variation

During our several months of reprojection and reduction computations in Windows Azure we have observed significant performance variations within a compute instance. Ideally a virtualized compute instance would have identical performance along its lifetime, but in practice many sources of variations from the cloud infrastructure could break this assumption. These sources of variations include imperfect resource isolation between hosted instances, fluctuating network performance, and the location of the instance, etc.

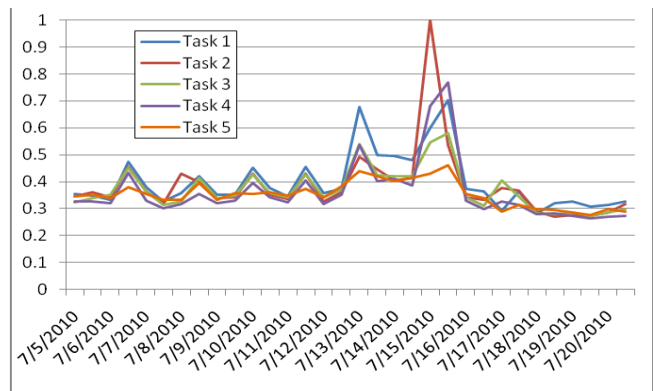


Figure 5. Performance variability of a single compute instance at different temporal points.

We have conducted an experiment to evaluate the performance variations. From July 5 to July 20, we set up a compute instance to run the same set of 5 reduction tasks twice a day (6am and 5pm US EST Time). We then measured the execution time for each task run. The results are shown in Figure 5. Each line along the X-axis represents the time series for task execution, and the Y-axis value for a data point in each line represents the normalized task execution time, with a maximum execution time of ~13 minutes. As shown in the figure, the difference between the best performance and the worst performance for the same task execution can be as large as 350%.

II) VM Hosting Reliability

The reliability of VM provisioning and hosting in Windows Azure is yet to be fully understood [13]. For MODIS Azure, we have maintained a logging table to track every single instance start event occurring in the system. Instances are started either upon a clean service deployment or during the dynamic instance scaling processes. By analyzing this logging table, we were able to track the information about every VM instance that have been started. Through the analysis, we found a fraction of the instances have ran into unknown problems after a certain running period, and a substitution of the same instance was started by the Azure infrastructure after the problem was detected. In some cases, the same instance has been retried starting for many times. Figure 6 shows the separation of instances which were started only once during their lifetimes (unique starts) and instances that have been started for multiple times (retries). Out of a total of 10032 VM unique instance start events, 8568 instances only started once during their lifetimes, a success rate of approximately 85%. This is certainly not a satisfactory number for system reliability, which again indicates the importance of fault tolerance on the application level.

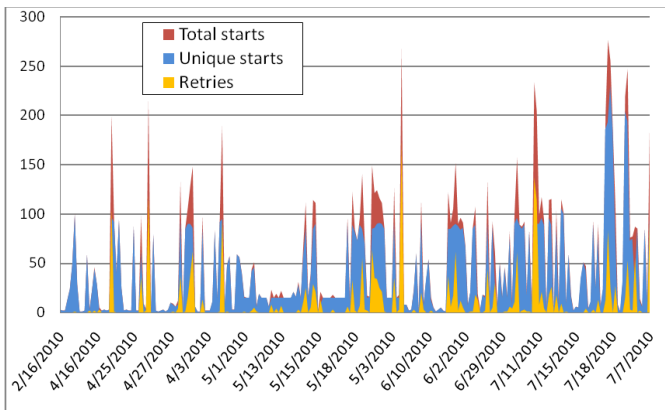


Figure 6. Instance start events during a five-month period. The instance start events are broken down to unique instance starts and retried instance starts.

V. RELATED WORK

An increasing number of existing scientific applications and benchmarks have been migrated and deployed to the clouds to evaluate the performance and quality of service in the cloud environments. T. Gunarathne et al. [14] compared the performances of the Cap3 and the MDS & GTM interpolation scientific applications in both EC2 and Windows Azure. X. Qiu et al. [15] took a similar approach and compared the performance of 3 bioinformatics applications on Windows Azure and Dryad. K. Jackson et al. [16] ported a large-scale image processing application for seeking supernova from local clusters to Amazon EC2, and compared a number of different data storage and communication strategies. A. Thakar and A. Szalay [17] discussed their experience with migrating a scientific relational database into both EC2 and SQL Azure, and evaluated the performances as well as identified their limitations as compared to on-premise database solutions. Of all the ongoing efforts on evaluating the feasibility of cloud computing for data-intensive eScience applications, the AzureBlast project [18] took an approach that is the most similar to ours to use the basic service elements provided by Windows Azure to build a parallel bioinformatics application running the BLAST algorithm.

Our work is complementary to the above projects in that we have built a production earth science application in the cloud and have been operating it over a 9-month period. The unique experience we gained from continuously scaling up the computation in the cloud provides an early picture on some of the issues with a long-running production eScience application in the cloud.

VI. CONCLUSION

In this paper we provide some early observations and experiences with the development and operation of the MODIS Azure application in the Windows Azure cloud computing platform. Not like the approach used by many other eScience applications that directly move existing codes and software stacks into the cloud, we build the application from scratch on top of the basic service elements and scalable infrastructures of cloud computing.

Our decision to build a satellite image processing pipeline leveraging the native capabilities of Azure has served us well. As we have scaled the application from Continental US to global scale, our initial service architecture has had only minor changes. We have leveraged blob service to store and manage large amounts of science data; the queue service for task dispatching and scheduling; the table service to monitor execution status in real-time and keep history logs; and the Management API to dynamically scale up/down the instances to be adapted to the dynamic workloads.

Our decision to “bake in the faults” has also served us well. While Azure presents a highly reliable platform and masks many faults, our scale is such that even 99.999% reliability still creates too many faults for human examination. At the same time, the virtualized nature of Azure presents new faults such as VM substitution. Our application is delightfully parallel and the image tile is an obvious idempotent building block. This

enabled us to rapidly understand how and where to build in fault retries that isolate our science user.

Lastly, our decision to use Azure tables as a common logging mechanism has given us two very important abilities. First, we can monitor our application and use the accumulated measurements to plan forward. Second, that same forensics also gives us the ability to debug the science application code forensically.

Overall, we think cloud computing has provided an appealing environment for building scalable, data-intensive eScience applications. However, in this early stage, it still has some limitations on the application development and execution processes. The hosted environment and black-box nature of cloud computing indicate that we will at least have to live with that for a long time.

ACKNOWLEDGMENT

This work is supported in part by Microsoft External Research. It is also supported in part by the Director, Office of Science, Office of Advanced Scientific Computing, of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

REFERENCES

[1] T. Hey and A. Trefethen, "The Data Deluge: An e-Science Perspective", Grid Computing: Making the Global Infrastructure a Reality, Wiley 2003.

[2] Google App Engine. <http://code.google.com/appengine/>

[3] Amazon Elastic Compute Cloud (EC2). <http://aws.amazon.com/ec2/>

[4] Microsoft Windows Azure Platform. <http://www.microsoft.com/azure/default.mspx>

[5] J. Li, D. Agarwal, M. Humphrey, C. van Ingen, K. Jackson, Y. Ryu, "eScience in the Cloud: A MODIS Satellite Data Reprojection and Reduction Pipeline in the Windows Azure Platform", 24th IEEE International Parallel & Distributed Processing Symposium, April 2010.

[6] MODIS Website. <http://modis.gsfc.nasa.gov/>

[7] The HDF Group. <http://www.hdfgroup.org/>

[8] FluxNet. <http://www.fluxdata.org>

[9] Y. Simmhan, C. van Ingen, G. Subramanian, J. Li, "Bridging the Gap between Desktop and the Cloud for eScience Applications", The 3rd International Conference on Cloud Computing, July 2010.

[10] Z. Hill, J. Li, M. Mao, A. Ruiz-Alvarez, and M. Humphrey, "Early Observations on the Performance of Windows Azure", 1st Workshop on Scientific Cloud Computing, June 2010.

[11] R.E. Wolfe, D.P. Roy and E. Vermote, "MODIS land data storage, gridding, and compositing methodology: Level 2 grid", IEEE Transactions on Geoscience and Remote Sensing, 36(4): 1324-1338.

[12] Microsoft Online Services Customer Portal. <https://mocp.microsoftonline.com/site/default.aspx>

[13] R. Barga and C. van Ingen, Invited talk on Microsoft Faculty Summit 2010. http://research.microsoft.com/en-us/people/barga/faculty_summit_2010.pdf

[14] T. Gunarathne, T. Wu, J. Qiu, and G. Fox, "Cloud Computing Paradigms for Pleasingly Parallel Biomedical Applications", Emerging Computational Methods for the Life Sciences Workshop, June 2010.

[15] X. Qiu, J. Ekanayake, S. Beason, T. Gunarathne, G. Fox, R. Barga, and D. Gannon, "Cloud Technologies for Bioinformatics Applications", 2nd ACM Workshop on Many-Task Computing on Grids and Supercomputers, November 2009.

[16] K. Jackson, L. Ramakrishnan, K. Runge, R. Thomas, "Seeking Supernovae in the Clouds: A Performance Study", 1st Workshop on Scientific Cloud Computing, June 2010.

[17] A. Thakar and A. Szalay, "Migrating a (Large) Science Database to the Cloud", 1st Workshop on Scientific Cloud Computing, June 2010.

[18] W. Lu, J. Jackson, and R. Barga, "AzureBlast: A Case Study of Developing Science Applications on the Cloud", 1st Workshop on Scientific Cloud Computing, June 2010.