Bridging the Gap between the Cloud and an eScience Application Platform

Yogesh Simmhan, Catharine van Ingen
eScience Group
Microsoft Research
Los Angeles, CA, USA
{yoges, vaningen}@microsoft.com

Girish Subramanian
Computer Science Department
Indiana University
Bloomington, IN, USA
subramag@cs.indiana.edu

Jie Li
Department of Computer Science,
University of Virginia,
Charlottesville, VA USA
jl3yh@virginia.edu

***Abstract*— The widely discussed scientific data deluge creates not only a need to computationally scale an application from a local desktop or cluster to a supercomputer, but also the need to cope with variable data loads over time. Cloud computing offers a scalable, economic, on-demand model well matched to the evolving eScience needs. Yet cloud computing creates gaps that must be crossed to move science applications to the cloud. In this article, we propose a Generic Worker framework to deploy and invoke science applications in the Cloud with minimal user effort and predictable, cost-effective performance. Our framework is an evolution of Grid computing application factory pattern and addresses the distinct challenges posed by the Cloud such as efficient data transfers to and from the Cloud, and the transient nature of its VMs. We present an implementation of the Generic Worker for the Microsoft Azure Cloud and evaluate its use in a genome sequencing application pipeline. Our results show that the user overhead to port and run the application seamlessly across desktop and the Cloud can be substantially reduced without significant performance penalties, while providing on-demand scalability.**

Keywords-cloud tools; escience applications; scientific data management; provenance; scientific workflows; performance analysis;

I. INTRODUCTION

Cloud computing has emerged during recent times as viable platform [10] for performing large scale computation and data processing operations. Building on prior work on Grid and cluster computing [14], it offers an evolutionary paradigm that approaches utility computing [4, 16]. The advantages of cloud computing are well known: it offers resources on demand and in a scalable manner, it allows users to pay as they go avoiding costly capital costs, it provides competitive pricing due to economies of scale, and it uses simple service interfaces for easy access and management. Commercial Clouds providing these features have been available for some time from different vendors, such as Amazon, RackSpace, and Microsoft, running out of large data centers that are geographically distributed. There is also active research on open source [5] and private clouds [17] and improving cloud operations [8].

Scientific applications have historically been major users (if not drivers) of next generation computation and data resources. Cloud computing has benefits [9] for eScience applications that range from those running on the local desktop to those that can only run at supercomputing centers [7]. The different scales of applications benefit differently.

- For *desktop users* whose applications have outgrown single machine resources, the prospect of building, operating, and porting their applications to a (distributed) cluster can be prohibitive. While Moore's Law improves workstation compute capacity and disks capacity continue to increase, that rate of growth is still dwarfed by the needs of scientific computation and data analyses. The cloud could provide a migration path to better scalability over a longer period of time as long as any changes to the programming models or interfaces remain simple. The cloud also provides on-demand access to resources at any time [12]. Not all scientific analyses are performed continuously. For example, an environmental scientist performing a field campaign may large computation and data resources during and immediately after the campaign for the initial data reduction but not thereafter.
- The cloud helps existing users of *local clusters* at universities or research groups reduce the cost of running a cluster and scale beyond local resources. Cloud computing provides a lower real (non-subsidized) total cost of ownership than private clusters by reducing the operations people cost, peak networking bandwidth requirements, and technology upkeep through constant machine upgrades [6]. For cluster users considering a step up to national supercomputing centers, paperwork and eligibility requirements may restrict access and delay account creation. With cloud computing, all it takes is a credit card to open a new account online and a few minutes wait. While national labs provide technical support on the clusters or Grids, they also impose security restrictions that may limit the software stack on the nodes or services that can that can be deployed. Applications often have to be modified to be compatible with a center's user policies and/or scheduler which differ from the local clusters. While a similar challenge may exist for cloud services, the leap can be less disruptive and may give the user full control over compute nodes.

- Users of *supercomputing centers* can also find benefits from the on-demand nature of cloud computing. The queue wait times at supercomputing centers can be long and clouds provide resources near-instantaneously. Anecdotal evidence [12] shows that despite a lower throughput, cloud computing can have a faster time to completion when including queue wait times. Clouds can also be used as a resource surge when centers hit peak load by shedding smaller jobs to clouds. For example, clouds can be used when running tutorials or hands-on workshops that require large resources for a short period of time especially since guest accounts for all attendees can be a challenge. Lastly, users can easily share large datasets generated within the cloud to be analyzed and processed within or outside the cloud, democratizing access to scientific analysis. Supercomputing centers may restrict *ad hoc* hosting of these datasets or limit their analysis at the center to authorized users.

Cloud computing comes in several favors, primarily Infrastructure as a Service (IaaS), Platform as a Service (PaaS) and Software as a Service (SaaS). The former abstracts the hardware resources to give equivalent of barebones machine that typically runs as a virtual machine, Amazon's EC2 being a popular example. PaaS provides a restricted sandbox or runtime environment with additional native services, such as Google's AppEngine. SaaS provides applications as scalable services that can be used to compose other applications, one example being SalesForce.com's business applications. Since SaaS does not yet offer a broad set of composable scientific services, IaaS and PaaS are the relevant flavors for building or porting science applications in the cloud. Such applications may themselves be offered as a public service, thus in effect evolving into eScience SaaS. The focus of our paper is on effective use of existing *commercial* clouds that are widely available to run scientific application.

While Cloud computing provides desirable features for scaling science applications, there are significant challenges for users to easily access and efficiently use it for existing applications [15]. In this article, we focus on tools to build an effective platform for eScience applications in the cloud while limited by the abstractions that it provides. Specifically, we identify three issues that pose an overhead to a broad category of scientists: (1) Ease of porting and running of science applications in the cloud, (2) Efficient data management across applications within and outside the cloud, and (3) Effective tracking of application and data provenance. These problems motivate the need for a *Generic Worker framework*, we propose, that lowers the barrier to entry for cloud computing. We implement the proposed framework for the Microsoft's Azure Cloud platform that straddles IaaS and PaaS, and evaluate the framework's performance and ease of use in a real world genome sequencing application.

The rest of the paper is structured as follows: in Section 2, we motivate problems for eScience on the Cloud using Amazon and Microsoft's cloud services and discuss related work; in Section 3, we introduce the generic worker design for application deployment and execution; Section 4 delves into efficient transfer of scientific data between cloud and desktop applications; provenance tracking of cloud applications is featured in Section 5; Section 6 presents a qualitative and empirical evaluation of the framework for a genome phasing science application; and Section 7 presents our conclusions and future work.

II. Background and Related Work

Among the commercial clouds available, Amazon's Elastic compute and storage *infrastructure* as a service have been popular in the research and scientific communities for several years [11]. Microsoft's Azure provides a .NET *platform* as a compute service with some features of a Windows infrastructure as a service, along with storage and queue services. Azure has been in public beta (Community Technology Preview) for the past year and, as of writing, is due to be released next week. These two are examples of IaaS and PaaS clouds that can run scientific applications and discussing their limitations in running science applications in this section to motivate our paper. We later present related work on enabling tools for science applications in cloud and grid computing.

A. *Amazon Elastic Cloud and Microsoft Azure*

The scalable cloud infrastructure services provided by Amazon to run applications are Elastic Compute Cloud (EC2) barebones virtual machines (VM); Simple Storage Service (S3) for flat file store, Elastic Block Service (EBS) for file system storage and SimpleDB for a hashtable of attributes; and Simple Queue Service (SQS) for reliable message passing, typically between clients outside the cloud and applications within. EC2 allows any OS image to be installed in their VMs along with any of the user's software and tools. This provides the flexibility of being able to run virtually any application the user runs on their desktop with the requirement that the user is responsible for managing the software stack in the VMs or forming a cluster out of the VM instances. Public IP addresses are provided for up to 5 EC2 instances. S3 provides a REST API for reliably storing files in a flat collection of buckets, while EBS provides a robust virtual disk that can be mounted on an EC2 instance and run a file system of the user's choosing.

Microsoft's Azure cloud platform provides Window Server 64bit virtual machines that use the .NET runtime as the platform for running applications. VMs fall in either a Web role that have public IP address and run IIS server or a Worker role that runs a .NET application. Worker roles can be used to fork processes that run executables in a restricted Windows VM to provide IaaS. The Windows Server OS is managed automatically by the Azure fabric while users are responsible for the .NET applications they run in the roles. Like Amazon, Azure also provides a reliable Blob storage service for flat files collected in containers, durable Queues for message passing and indexed tables for attribute lookup. Unlike EBS, Azure does not offer disk volumes for network file access.

B. *Challenges of Running Applications on the Cloud*

Running a science application in the cloud presents three different challenges: initial application deployment, subsequent application execution, and data transfers to/from the cloud. Simplistically, deploying an application in the cloud requires that the application be installed on an existing or new virtual machine image. That image is then and deployed in the Cloud so that new instances of that virtual machine can be created. Data transfers to and from the application also need to be handled by the (non-cloud) client and the cloud application instances using cloud data services or other network protocols.

1) *Application Deployment.*

Amazon's EC2 and Microsoft Azure deploy applications differently. Amazon EC2 IaaS images (AMI) are built from the ground up by the user by reusing an existing or installing an fresh OS image, along with all software, libraries, and tools necessary to run the application. This image is packaged and uploaded to S3 cloud storage, from where EC2 VM instances of the image can be started. Windows Azure's PaaS differs by requiring users to implement a .NET worker role interface for each virtual machine image. The interface implementation can be the application itself, or it can start the application as a separate commandline process. The worker role implementation, external application and dependency files are deployed to the cloud through a web interface.

Both platforms require the scientist user to manage application dependencies, maintain their applications in an image/role, and recreate and redeploy that image in the event of any change to an application. Scientists often have little knowledge of application dependency "dll/jar hell" – a version of MatLab depends on specific language and Java runtime versions. This becomes more complicated when multiple applications need to be managed in a single VM to optimally use the VM resources. This makes managing transient applications in a cloud fairly challenging to a scientist.

Amazon's EC2 has the additional drawback that the scientist user is responsible for managing everything in the VM, including the operating system and standard software stack. While this gives the flexibility of a familiar computing platform with any OS or software installed in the VM image, having to deal directly with OS images often requires a system administrator to be engaged. Also, the large size of the OS image can be costly and time consuming to transfer between client and cloud unless updated in place over the network.

Azure reduces the OS and "DLL hell" management at the cost of flexibility. Azure directly maintains the Windows Server OS and the .NET runtime as part of the platform. Users need to only manage their own applications through worker roles. However, this also means that the OS and language runtime is fixed within the VM. Security restrictions on the Azure worker roles limit the software that can be installed on the VM to those not requiring administrative rights to install or run. Lastly, users cannot directly connect to a running virtual machine instance (e.g. SSH/Remote Desktop).

2) *Application Execution.*

Applications available in cloud VM instances need to be invoked from outside that VM instance. These user clients may be in other VMs or, often, outside the cloud as commandline tools or workflow engines that orchestrate cloud applications. Reliable queues in the cloud are often used to pass application execution requests that are picked up and processed by one of the VM instances having the deployed application. However, the client application or its proxy running in the VM instances must listen for these messages, interpret their payload, run the application and return the response. Users end up modifying their application to listen and process execution requests, or writing proxy agents or web services that do parameter marshalling and unmarshalling. This becomes tedious as applications increase or change over time.

Amazon's EC2 allows public IP addresses for VM instances and clients can invoke the (commandline) application on a specific IP address (e.g. using SSH). But EC2 limits the number of public IPs, hindering application scalability across multiple VM instances. .Microsoft Azure fits the above pattern by having users implement a worker role for each application that uses queues to pass messages for that application.

3) *Data Transfer.*

Passing files as input and output to cloud application introduces additional problems of managing data transfers between client and VM instances. Science applications and their clients normally use local or network files like NFS or CIFS for data exchange. However, S3 and Blob storage, while providing reliability and collocation with the VMs, do not support a network file system. VM instances themselves are transient, so their local disks are not reliable network stores. Using a public network share near the client breaks co-locality with the cloud applications. Users can either use a cloud network file system like Amazon's EBS, or explicitly change their applications/clients to transfer input and output files between client and cloud applications through Blob/S3 storage while being cognizant of network speed bottlenecks and transfer costs outside the cloud.

C. *Related Work*

Grid computing has used the factory pattern to dynamically instantiate stateful grid service instances, which can invoke an application on the cluster, using factory services [1]. While it does not prescribe how grid service instance is created, the GFac [2] and Opal [3] toolkits uses an application description document as metadata to dynamically instantiate a web service to invoke a commandline application, while also staging input and output data over network shares. Application descriptions can be a deployed at runtime for a binary application available in the grid. GADe uses a detailed application description model to capture dependencies for complex applications in Grids [20].

Our proposed model is similar to GFac and Opal but has key differences to meet specific challenges posed by cloud computing. We do not assume that the applications are already present locally in the VM or executable ever the network. Part of the problem we address is dynamically deploying the application binaries and its files to a VM for execution to reduce changes to the VM image. Also, we cannot assume that input and output files to the application are accessible over a network file system. We support cloud storage protocols for transparently staging data between client, persistent cloud storage and VM's local storage that also supports passing data by value or reference to reduce data transfers between client and cloud applications. For .NET applications, we use reflection to automatically create an application configuration file instead of requiring users to fill input and output parameters for the application. While we do not generate a web service for the application, our XML based request/response messages that use Azure Queues are similar to SOAP messages for future compatibility. We envisage handling simpler applications self-contained in a VM than the complex software dependencies supported by GADe since those are more likely to be ported to and scaled in the cloud. However, supporting predefined runtime platforms dependencies would be useful.

The Blob Caching that we present is similar to other distributed master/slave caches in the cloud like Google *MemCache* for objects or Elastic MapReduce's *DistributedCache* for static files. Unlike the in memory object caching of the former, we support disk caching in the VM for files, which are more common input/output parameters for science applications. The files our Worker roles cache is determined dynamically unlike the static specification of DistributedCache.

III. Generic Workers for Cloud Applications

Porting a science application that runs on a local machine to the cloud means answering these questions:

1. How do we (easily) upload our application and its dependency libraries into the cloud? *E.g. Are the DLLs for my executable available?*
2. How do we ensure that the application's runtime environment is available and current when the application is invoked? *E.g. Is JRE 1.6 present?*
3. How do we make efficient use of Cloud resources with minimal application impact?
4. How do we load and run multiple, diverse applications simultaneously within a single virtual machine instance in the Cloud?
5. How do we track and monitor applications?
6. How do we scale the application across cloud resources?

We address the first five questions here. The last question is not key to our goal of taking existing applications and easily running them in the cloud with little code change or administrative overhead. We focus on the gap between cloud IaaS/PaaS and conventional local desktop/cluster computing.

We achieve this for Microsoft Azure by creating a "Generic" Worker Role that reduces common glue code that science users have to write to deploy and manage their application in the Azure cloud. We believe the same Generic Worker Role pattern applies to Amazon's EC2 and other IaaS/PaaS clouds, albeit with somewhat different underlying implementation..

Our Generic Worker is a wrapper for existing science applications that are either command line executables, or present as a method within a .NET library. Worker Roles are the entry point for performing computation on Azure and typically, a worker role is implemented for each application. While this may work for a science application written from scratch, the ability to run existing applications with little overhead is crucial. Also, there is sufficient common glue code across cloud applications that can be avoided and applications written specifically for a cloud platform becomes difficult to switching to other clouds or the desktop. Using the Generic Worker pattern decouples the application logic from the cloud dependencies.

Our Generic Worker role is "generic" because it is can execute any application that is dynamically registered with the framework. It provides four key functions: application deployment, client:cloud application communications, application execution, and reliable scaling.

A. Application Deployment

Applications undergo a two step deployment process with the generic worker framework before execution by a generic worker instance. The first step, *Application Upload,* copies the application binaries and dependency libraries into cloud storage that makes it accessible to the generic worker instance at runtime. The second step, *Application Registration,* informs the generic workers of the application file location and the set of runtime input and output parameters used for the application. Registration information is persisted in a Generic Worker Registry stored in Azure tables.

During application upload, users copy the application binary and libraries to a directory (container) in Azure Blob store (Figure 1, step R1). This container is later copied to the Generic Worker instance as a local directory and acts as the "bin" directory for the application when it is executed. The

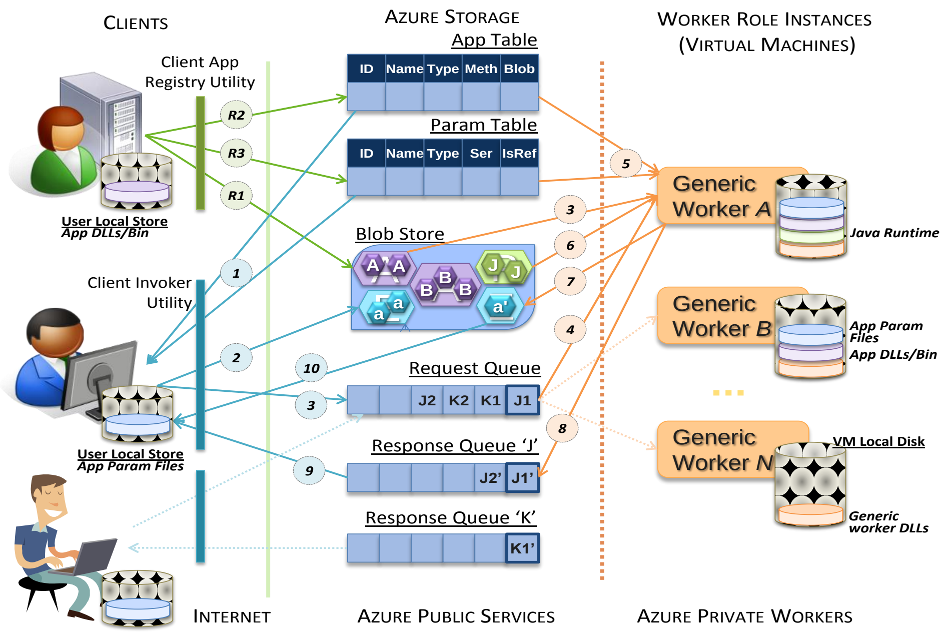

Figure 1. Generic Worker Architecture on Microsoft Azure. Numbered arrows are order of application deployment (R1-3) and execution (1-10).

current implementation shares the same security credentials for the Blob store holding the application container and the Generic Worker role. Since Azure Blob and Amazon S3 are limited to a flat directory structure, if applications require nested directories, the Generic Worker supports storing the application directory as a zip file Blob that is later expanded.

During application registration, the user supplies the generic worker with the information necessary to launch the application (Figure 1, step R2-3) via a command line utility. The information includes the application type, application container location, invocation target, and/or parameter types to be passed. The application type is used both to identify and configure common runtime libraries and to determine the invocation mechanics. Our current application types include .NET, Java, MatLab and generic Windows commandline executable. As an example, the Generic Worker ensures that the JRE is available to Java applications.

Our Generic Worker currently supports two invocation mechanisms: .NET method and Windows executable. For .NET applications, the users specify the fully qualified class name and the method name and the registration utility uses .NET reflection to determine the typed input and output parameters from the application DLLs. Similarly, Java applications will leverage Java reflection in future. For other applications, the users specify the commandline executable to invoke along with a list of commandline parameters. The parameters are passed as mappings between the typed input parameters passed from the client and the string parameter passed to the commandline. Output parameters include the return value of the executable process and/or files or directories that the application may create in the working directory. In addition, users can specify the each parameter's value/reference type and serialization.

The registration utility saves the registration information in the App and Param Azure Tables of the Generic Worker Registry. The user may also specify a unique logical name for the application to refer to that application in future, use the original fully qualified name, or an auto-generated Application ID.. These deployment steps for applications supported by the Generic Worker framework are much simpler than the normal cloud application deployment process provided by Microsoft Azure and Amazon EC2.

B. Client:Cloud Communications

The Generic Worker includes an intuitive way for clients to invoke a cloud application registered with the framework that hides the queue-based, asynchronous execution model normally used to run cloud applications. Typically, Azure Worker Role instances wait for work request messages to arrive in a predetermined queue, clients put these messages with work details in the queue, and likewise for responses.

Besides imposing an asynchronous model of invocation, queues also require clients and applications to serialize and deserialize the parameters, and cannot handle large message sizes. Azure Web Roles provide a way to hide the queue using an application specific web service, but require the web service to be manually defined. Amazon's EC2 offers similar capabilities as Azure Queues with the same issues. Amazon also provides Elastic IP – a feature that exposes a public IP address to the EC2 instance that blurs the distinction between public Web and private Worker roles of

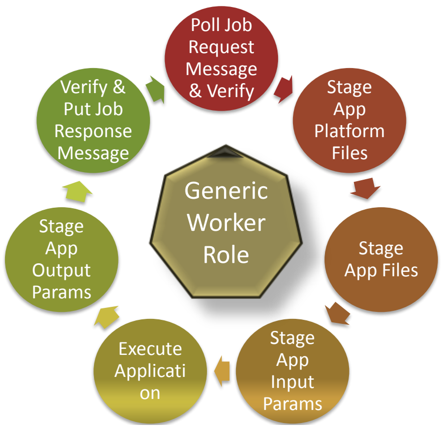

Figure 2: Program lifecycle of a Generic Worker instance

Azure. This allows the use of a network socket for RPC calls applications in EC2; however, the number of elastic IPs available is limited.

We provide a client library with a common *Invoke* method to run any application that mitigates the additional complexity and overhead of dealing with queues, serializations or application-specific web services. The *Invoke* method takes the application identifier of a registered application and the list of parameters registered for that application and receives .NET objects on return. *Invoke* serializes the parameters using XML or string serialization as registered. Handling of files and large parameters is done by uploading them to Azure Blob store and passing their references. *Invoke* then type-verifies the application identifier and serialized parameters with the Registry and combines them with the name of a Response queue and Log table into a Job message. The Job message is then placed on the Request queue shared by all Generic Worker instances. *Invoke* then polls the Response queue waiting a Job response message. The Job response message is created by the Generic Worker instance that dequeued the original Job request and executed the application. The return parameters are serialized by the Generic Worker instance and deserialized by the client library into .NET objects. In case of an exception message, an Exception is thrown to the client. The progress of the Job execution can be monitored using the Application Log queue by spawning a separate thread or from a commandline utility.

This client execution model makes it flexible enough to map it into a commandline utility, a web form, as a generic activity that can be used in a workflow, or directly for use by a .NET client.

C. Application Execution

The primary task of the generic worker is to execute any registered application without restricting the number of registered applications in a single virtual machine or imposing a specific mix of applications. This means that the application must be instantiated dynamically on the VM by the Generic Worker. That includes deploying the required application runtime environment and binaries on the VM from persistent cloud storage, deserializing and staging the input parameters and files from the client, invoking the application, persisting its output parameters and files, and returning the serialized outputs or an exception back to the client. Generic Workers capture common template code thus freeing users from these concerns.

On-demand, pay-as-you-go execution creates a need for dynamic instantiation of Generic Workers. When a worker instance starts up, it is running in a clean virtual machine that was originally deployed, and the only local files available to the Generic Worker are its own libraries. Once instantiated, the Azure Fabric calls the *Initialize()* method of the Worker role interface which in the Generic Worker case initializes a Control Log queue for publishing the instance's status and ensures that the Job request queue exists. Following this, the *Start()* method of the Worker role is called by the Azure Fabric and the mainline code of the Generic Worker starts. The new Generic Worker starts polling the Job request queue within an infinite loop looking for work (Figure 2).

Job request messages are processed first in, first out. The Generic Worker instance picking up the top message verifies the application ID or logical name present in the Job message, verifies its parameter against the registry tables, and transmits the status of the Job onto the Application Log queue present in the message. Depending on the type of application (e.g. .NET, Java, MatLab, Windows Executable), the worker may have to prepare the virtual machine instance to run that particular application type. For example, in the case of Java applications, this involves copying the Java Runtime Environment files from a well known location in Azure Blob store to a directory in the local virtual machine and ensuring that *java.exe* is present in the commandline execution path. For .NET and Windows commandline executables, no preparation is needed.

Next, the worker creates a working directory for the application execution and downloads the application libraries from the Application Blob container listed in its registry entry. If compressed, the files are extracted to duplicate the directory structure expected by the application. The input parameters present in the Job Request message are deserialized and optionally dereferenced from Blob store in case of files or large objects. If the application is a .NET method, the target class is instantiated using reflection and the method invoked using the deserialized .NET objects in the same process. For other types, the invocation is done by mapping the input parameters to the appropriate commandline specific to that platform and application and the commandline forked as a separate process for which the Worker instance process busy-waits.

Once application execution completes, successfully or not, the results from the invocation are collected and serialized back into output parameters. While the outputs are defined by the return and "out" parameters for .NET methods, for other platforms, the output parameters are typically the return code from processes or files defined in the application registry mapping.

Finally, the Generic Worker performs a number of cleanups and persistence. A log file of console output as well as any files generated by the invocation are uploaded from the VM instance to Azure Blob store. The serialized output parameters and references to any output blobs are enqueued to a response queue as a Job response message. The original Job request message is then permanently deleted. The Generic Worker instance returns to polling the Job request queue.

Generic Workers can run any registered application. So application loads is distributed across all worker instances allowing load averaging, rather than having to reserve specific instances for specific applications, whether those applications are being run or now.. For simplicity, we currently do not support running multiple Jobs simultaneously by the Generic Worker, so this may cause the VM to be underutilized. As future work, we plan to concurrent application in the same VM as multiples threads (.NET) or processes (Java/Windows executable).

Our Generic Worker also guarantees reliable completion of job requests by leveraging the the Azure queue behavior that revives messages when they are not deleted within a certain period. By performing a non-destructive read of the request message and deleting the request message only after completion of the application execution, the Generic Worker instance ensures that if it crashes when running, the read message will reappear after a predetermined timeout for another generic worker instance to process the message. One drawback of this is the time delay between the crash of the Worker instance and the reappearance of the Job request message. For high priority jobs that need reliable and timely completion, we support multiple worker instances working on the same Job request simultaneously by posting two copies of it and the first response being used by the client. While this causes twice the work to be done, it ensures that there is a higher probability that the application execution completes successfully. The generic worker client picks the first response message that arrives for the job request and returns it to the user, who is unaware that multiple job were run.

IV. Data Access for Cloud Applications

Files located on local disks or network shares form an important and often primary form of data used as input and output to science applications. Applications that run in the Cloud need to access local files on the client outside the cloud, and vice versa, in a reliable manner without significant performance or user/code overhead. Scientists write their applications using simple local or network file paths but local file paths on VM instances exist only during the lifetime of the VM, and not all cloud platforms provide a shared network file system like Amazon EBS. This necessitates data movement between the client and the VM using shared cloud storage service interfaces that are cumbersome and unintuitive for science applications. That movement carries a performance and cost penalty. We address these data management issues in this section.

As such, the challenge is to enable cloud applications to transparently access client local files, VM local files, and persisted cloud storage with minimal user coding effort overheads and no more than modest performance impacts. Our Generic Worker includes mechanisms to reduce the code overhead by automatically persisting and transferring input and output files between client and the VM, and reduces the performance impacts with intelligent local file caching of on VMs and just-in-time data transfers. Again, we use Microsoft Azure as an example though the same is applicable to Amazon EC2 and S3 (but not EC2 and EBS).

A. *Sharing Files between Client & Cloud Application*

Our Generic Worker and client library enables clients and cloud applications to pass local files as parameters by transparently copying files between client/VM local disk, cloud persistent store and VM/client remote disks as necessary. Passing local files and large objects between client and the application running in the VM usually requires additional data movement since clients and VMs cannot directly access each other's local files. Though the VM instances are in the cloud, they do not provide reliable local storage and their output data has to be moved to cloud storage for persistence. Similarly, large in-memory objects that overflow the queue request and response message size have to be mapped to cloud storage and transmitted between client and application..

When an application is registered, the parameters that correspond to input or output files are marked as a special *file* type. File parameters of the .NET built-in type *FileInfo* (which wraps a local file path) or *BlobFile* (an .NET object we provide) are automatically recognized through reflection. All file types are passed as references in the Job request and response messages. The references are URLs to Azure Blob that are uploaded to a (pre-defined or configurable) container in cloud storage when the file parameter is serialized at the client or VM. The difference between *FileInfo*/"string" file path parameters and *BlobFile* objects is that *BlobFiles* track both local file path and Azure Blob URLs as replicas of each other with either replica being accessible and downloaded on-demand when explicitly used. The *FileInfo*/string file paths, however, are only accessible by their local path – the Blob URL is used internally by the Generic Worker and client library in the request/response messages – and hence always downloaded locally from cloud store when they are deserialized, whether the files are actually used or not

There are a several advantages to the file handling we provide. The science application and clients can always use local files as parameters and not worry about uploading and downloading them to/from cloud storage. The application also does not have to concern itself about the directories to upload and download the files as Blob container creation and local path rewriting is done automatically by the Generic Worker and client library. Lastly, using *BlobFiles* ensures data is not downloaded from cloud storage unless used by the client/cloud application. For e.g., if a scientific workflow is running on the desktop client and uses several cloud applications, the intermediate *BlobFile* data does not have to be downloaded to the client and can be passed between the cloud applications as references. Only the final data product generated by the workflow need be downloaded to the client if required.

The Generic Worker and client library treat large in memory objects similar to files by serializing them to local files, uploading them to Blob storage, passing their Blob URLs in the request/response messages and deserializing them back to objects at the remote end.

B. *Efficient File Management*

Runtime platform files and application binary /library files required to run the application need to be present within the local storage of the VM before application execution. These files can either be present as part of the virtual machine image when it is instantiated, or downloaded from persistent storage as required to run an application. The advantage of the former is that it avoids the cost of downloading the application dependency files from persistent cloud storage to local disk at application runtime, and allows fine grained management of the files in the VM image. This, however, persists the problem of manually deploying and maintaining applications in VM images and can bloat the image size if there are several applications.

The generic worker uses a just-in-time download to deploy application and platform files. Application files are downloaded on demand when the application execution request is received by the generic worker instance. We mitigate some of the performance penalties associated with downloading the application and platform files by using a caching mechanism and by allowing all application binaries and libraries to be compressing into a single tarball package when they are registered. This reduces the observed performance overhead associated with downloading multiple small files.

We provide a Blob Cache service that runs in the Generic Worker instance to mitigate the performance penalties of uploading and downloading files to/from the reliable cloud store. All Blob operation in the Generic Worker goes through our Blob Caching library that uses the local VM disk to cache files indexed by the Blob ID and tracks dirty entries using the ETag version number assigned by Azure Blob store. These Blob files may be may be application inputs, application outputs, application binaries, or runtime platform files. The cost of reading a file from cloud storage is often several times slower than local disk access so caching commonly used files is a performance savings. For example, when several applications or multiple invocations of the same application share reference files that are often static, it helps to download them just once and reuse them. Also, applications that execute in a sequential pipeline where the output of one is passed as input to the next benefit from uploading each intermediate file once for persistence to cloud storage and reusing the local copy as the input to avoid the download.

The Blob Cache is writethrough to avoid data loss. The library uploads any updated file into Azure Blob store and updates the local cache – creating a new cache entry if the files did not exist before – with the new version number assigned by the Blob store. This ensures that the copy of the blob file in cloud storage is always the latest version and all VM caches are coherent with this copy. Also, a crash of the virtual machine instance will not cause a loss of data.

V. Provenance Collection and Logging

Cloud computing presents additional provenance challenges due to the inherent distribution across the cloud and the local client. Science applications running in the cloud need to be monitored to track their execution progress, to collect provenance on data, and to audit application execution for resource usage. Cloud resources can be prone to failures as with most distributed systems. Monitoring services from cloud providers, such as Amazon *CloudWatch* and Azure *Diagnostics*, help track the infrastructure status like VMs that are running and CPU or I/O usage, but do not inform users about the specific applications that are affected by this. Also, the inability to remote logon to Azure VMs limits the ability to run desktop monitoring tools to check application status or browse local files.

Applications that run in the Generic Worker instances are automatically tracked and monitored. There are three types of monitoring: tracking the control flow between desktop clients and cloud applications in the VM, the tracking the data provenance across client/VM local store and Blob store, and tracking the status of the VMs themselves.

The Generic Worker instances publish the status of their liveliness on a Control Log queue. Each worker publishes timestamped status messages with its identity that tracks the instance liveliness, Job statistics, and resource usage. Any traceable exception in the VM is also published. This log can be persisted to Azure tables and supplements diagnostics provided by Azure.

The application Jobs are tracked on an Application Log queue created for each job and shared by the client and the Generic worker. This log tracks the control flow between client to cloud application and back to client, recording realtime state changes such as client

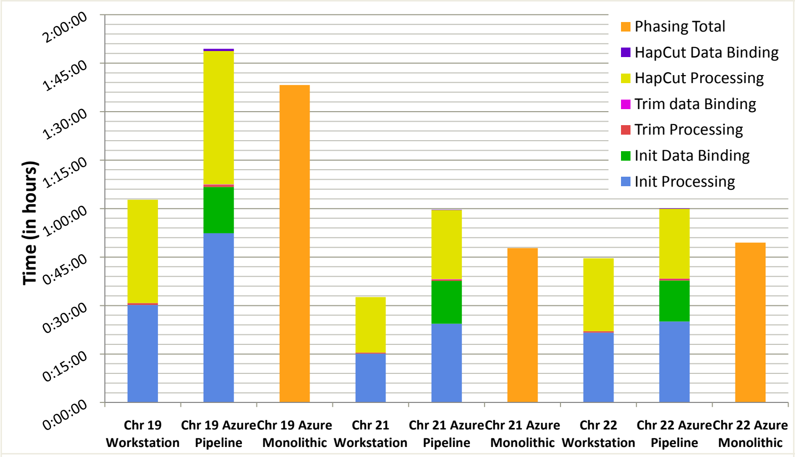

Figure 3(a). Performance of the Azure pipeline application against quad core workstation and Azure monolithic application for 3 chromosomes.

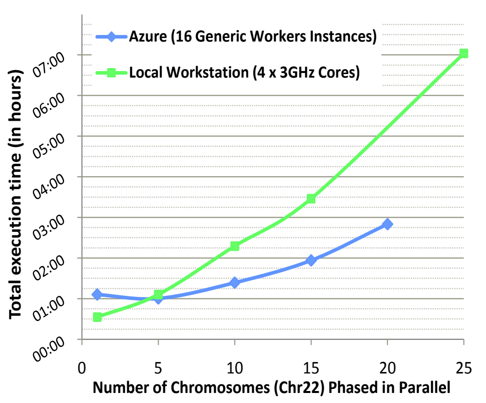

Figure 3(b). Performance of Azure pipeline application against quad core workstation with increasing parallelism.

placing Job in work request queue and the Generic Worker instance that picks it up for application execution. Users can monitor this log queue to track their Job progress.

Application provenance is determined by the combination of the Job request/response and application log messages. Since all input and outputs to/from the cloud application are typed and files are identified, this allows us to track the location and copies of all files that were part of the application execution: on the client local store, cloud storage and VM disk. We also provide a way to record the provenance to Azure table store for simple queries and will later support their export using the Open Provenance Model [19] schema for provenance tracking across the cloud and non-cloud applications.

VI. Evaluation for Science Application

We apply the generic worker framework we have implemented to a genomics application to evaluate the efficacy of our Generic Workers for porting a real world genomics phasing application science application to run in Azure. We evaluate two aspects:

- The coding and deployment overhead to port the application to the Cloud,
- The performance tradeoff between running the application on a local machine versus in the cloud using the generic workers,

Human genome phasing is the process of separating the two haplotypes sequences contributed by each parent as part of sequencing an individual. Haplotype identification plays an important role in predicting diseases predisposition and phasing is a compute intensive application. We use the HapCUT phasing algorithm from UCSD [18] which is implemented as a .NET library containing the 3 sequential stages of the phasing pipeline: *Initialize*, *TrimSparse* and *DoHapCUT*. A client running calls each of these methods in succession, passing files between them to simulate a sequential workflow. The input to the pipeline is a chromosome sequence (~40MB) and a reference database file (~120MB) and the output is a pair of haplotype sequence files (~1MB). The intermediate files are about 50MB at each stage.

A. *Ease of Porting Science Application to Cloud*

The tasks required to port the local phasing pipeline to run in the Cloud using the generic worker model involved registering the application with the generic worker registry and modifying the client pipeline to use the Generic Worker client library to invoke the methods instead of directly invoking them, and changing the methods to use *BlobFile* types for file parameters instead of *FileInfo*. We wrote a trivial wrapper to achieve this to improve the performance.

Application registration involves uploading the phasing library and dependency DLL files to a unique */PhasingSteps* container in Azure Blob store using a *COPY-CD* shell script from Azure. Next, the three applications are registered with the Generic Worker Registry using the *AppRegistryUtil* commandline tool we provide. This takes the fully qualified method name as parameter and the name of the application Blob container, and returns the ID of the registered application. The input and output parameters for these methods are simple integer, float and string value types, or files defined as *BlobInfo* types using the wrapper. Once registered, these three applications are run in as cloud applications by calling the *Invoke* method in the client library using their fully qualified method names and .NET objects as parameters, with a method signature similar to the original methods.

The effort required for this is minimal as compared to rewriting all the phasing steps to run in the Azure Cloud as separate worker roles. Wrapping each method to use the *BlobFiles* instead of *FileInfo* parameters was the costliest human operation done to optimize file transfers. Using the original *FileInfo* objects would have entailed unnecessary data transfers between client and the cloud application for each intermediate step.

B. *Performance Comparison of Local and Cloud Versions of Science Application*

We run two experiments, the first to measure the overhead of running the application in the cloud as a pipeline and as a monolithic application as compared to a local workstation. Next, we measure the scalability of the Azure workers with increasing numbers of chromosomes sequences processed in parallel.

The first set of experiments compare the performance of the phasing pipeline by running the pipeline and the applications on a quadcore 3GHz Xeon workstation with 16GB RAM for 3 chromosomes. This is a baseline for local execution. We then run the client pipeline on a local workstation while using the cloud applications registered with the Generic Workers, and using *BlobFiles* – only the final output of the pipeline is accessed as local files. Lastly, we wrap the three methods into a single method that is registered as a separate application that is then invoked as a cloud application from the workstation.

The averaged results of these runs are shown for 3 different chromosomes in Figure 3(a). The graph shows that the workstation uniformly performs better than the other two, and this is due to two reasons. (1) the workstation has a higher CPU rating of 3GHz than the ones running Azure Worker Roles rated at 1.7GHZ, and (2) the time taken for data transfer of the inputs and outputs between client and the cloud application contributes to the total. Comparing the Azure pipeline version and the Azure monolithic version, we see that using the *BlobFile* objects keeps the performance of the two comparable since the *BlobFile* only transfers data on demand and we only access the output data on the client at the last stage. This is a more favorable model than running all three methods as a monolithic block in the cloud since it allows composition of science applications that are deployed in the cloud and their orchestration from any desktop client. If the *FileInfo* objects had been retained instead of *BlobFiles*, there would have been additional data transfer times for the intermediate files.

Figure 3(b) shows the performance of the phasing application running multiple sequences in parallel on the quadcore workstation, and the client running those on concurrent Azure generic workers (up to 16 instances). As can be seen, despite the network transfer overhead, Azure allows instant scalability of the application to as many worker instances as necessary and the increasing time is mainly due to the increased network transfers between client and cloud application.

VII. CONCLUSION

In this paper, we have shown the effectiveness of using cloud computing for scalable scientific applications and the benefits of using the Generic Worker model to easily deploy and execute applications in the cloud with minimal user effort. This further lowers the barrier to entry for science applications on the cloud. We have also shown techniques such as on-demand data transfers using *BlobFiles* that reduce the performance penalty for data transfer between desktop and cloud applications.

As future work, we will support multiple application runs simultaneously in a Generic Worker role, and address interesting questions on which applications to concurrently schedule on a particular VM to optimize resource usage. Another scheduling problem for future work is to leverage the data cached in the Blob cache for data locality. If a worker role instance already has data in its local cache that matches the input parameter of a job request, then that worker should be given a higher priority in choosing that job request over others.

VIII. REFERENCES

[1] Open Grid Services Infrastructure (OGSI), v1.0, 2003

[2] Building web services for scientific grid applications, G. Kandaswamy, et al., *IBM J. RES. & DEV.* 50 (2/3) 2006.

[3] Design and Evaluation of Opal2: A Toolkit for Scientific Software as a Service. S. Krishnan, L. Clementi, J. Ren, P. Papadopoulos, and W. Li.. *SERVICES* 2009.

[4] Above the Clouds: A Berkeley View of Cloud Computing, Michael Armbrust, et al., UC Berkeley Technical Report UCB/EECS-2009-28.

[5] The Eucalyptus Open-Source Cloud-Computing System, Nurmi, D., et al., *CCGRID* 2009

[6] Cost-Benefit Analysis of Cloud Computing versus Desktop Grids, Derrick Kondo, et al., *IPDPS* 2009.

[7] On the Use of Cloud Computing for Scientific Workflows, Christina Hoffa, et al., *SC*, 2009.

[8] Resource Monitoring and Management with OVIS to Enable HPC in Cloud Computing Environments, Jim Brandt, et al, *IPDPS* 2009.

[9] Cloud Computing for the Sciences, Francis Sullivan, *Comput. Sci. Eng.* 11(10) 2009.

[10] A break in the clouds: towards a cloud definition. Vaquero, L. M., Rodero-Merino, L., Caceres, J., and Lindner, M., *SIGCOMM Comput. Commun. Rev.* 39(1), 2008.

[11] Benchmarking Amazon EC2 for high-performance scientific computing, E Walker, *USENIX Magazine*, 2008.

[12] Slow Moving Clouds Fast Enough for HPC, Michael Feldman, *HPC Wire*, August 10 2009.

[13] CloudBurst: highly sensitive read mapping with MapReduce, M. Schatz, BioInformatics Vol. 25 no. 11 2009

[14] Cloud Computing and Grid Computing 360-Degree Compare, Ian Foster, et al, *GCE* 2008.

[15] Cloud Computing for e-Science with CARMEN, Paul Watson, et al, *IBERGRID* 2008.

[16] Commodity grid computing with Amazon's S3 and EC2, S Garfinkel, *USENIX Magazine*, 2007.

[17] Emergence of the Academic Computing Clouds. Delic, K. A. and Walker, M. A., *Ubiquity* 2008.

[18] HapCUT: an efficient and accurate algorithm for the haplotype assembly problem, Vikas Bansal, Vineet Bafna, *Bioinformatics* 2008 24(16).

[19] The Open Provenance Model (1.01), Luc Moreau, 2008.

[20] GADe: Toward Automatic Deployment of Applications on Computational Grids, Sébastien Lacour, et al. Grid 2005.