AzureBlast: A Case Study of Developing Science Applications on the Cloud

Wei Lu
Cloud Computing Futures
Microsoft Research
weilu@microsoft.com

Jared Jackson
Cloud Computing Futures
Microsoft Research
jaredj@microsoft.com

Roger Barga
Cloud Computing Futures
Microsoft Research
barga@microsoft.com

ABSTRACT

Cloud computing has emerged as a new approach to large scale computing and is attracting a lot of attention from the scientific and research computing communities. Despite its growing popularity, it is still unclear just how well the cloud model of computation will serve scientific applications. In this paper we analyze the applicability of cloud to the sciences by investigating an implementation of a well known and computationally intensive algorithm called BLAST.
BLAST is a very popular life sciences algorithm used commonly in bioinformatics research. The BLAST algorithm makes an excellent case study because it is both crucial to many life science applications and its characteristics are representative of many applications important to data intensive scientific research. In our paper we introduce a methodology that we use to study the applicability of cloud platforms to scientific computing and analyze the results from our study. In particular we examine the best practices of handling the large scale parallelism and large volumes of data. While we carry out our performance evaluation on Microsoft's Windows Azure the results readily generalize to other cloud platforms.

Categories and Subject Descriptors

D.1.3 [**PROGRAMMING TECHNIQUES**]: Concurrent Programming—*Distributed programming*; J.3 [**Life and Medical Sciences**]: Biology and genetics

General Terms

Performance, Design

Keywords

BLAST, Cloud Computing, Windows Azure

1. INTRODUCTION

Increasingly, scientific breakthroughs will be powered by advanced computing capabilities that help researchers manipulate and explore massive datasets. Today while the largest and best funded research projects are able to afford expensive computing infrastructure, most other projects are forced to opt for cheaper resources such as commodity clusters or simply limit the scope of their research. Cloud computing [2] proposes an alternative in which resources are no longer hosted locally, but leased from big data centers only when needed. This offers the promise of "democratizing" research as a single researcher or small team can have access to the same large-scale compute resources as large, well-funded research organizations without the need to invest in purchasing or hosting their own physical infrastructure. Despite the existence of several cloud computing vendors, such as Amazon AWS, GoGrid, and more recently Microsoft Windows Azure, the potential of cloud platforms for research computing remains largely unexplored.

While cloud computing holds promise for the seemingly insatiable computational demands of the scientific community, there are unanswered questions in the applicability of cloud platforms, including performance, which is the focus of this work. In this paper we present an experimental prototype, AzureBlast, which is designed to assess the applicability of cloud platforms for science applications. BLAST [1] is one of the most widely used bioinformatics algorithms in life science applications. The BLAST algorithm can discover the similarities between the two bio-sequences (e.g.,Protein). BLAST makes an excellent case study not only because of its popularity but also because of its representative characteristics of many applications important to data intensive scientific research. AzureBlast is a parallel BLAST engine running on the Windows Azure cloud fabric. Instead of using some high-level programming models or runtimes such as MapReduce[8], AzureBlast is built directly on the fundamental services of Windows Azure so that we are able to examine each individual building-block. Our ultimate goal is to provide a characterization of science applications appropriate for cloud computing platforms and best practices for deploying science applications on cloud platforms.

The structure of this paper is as follows. In Section 2 we briefly discuss the Windows Azure cloud platform and capabilities it offers to a computational scientist. In this section we also highlight aspects of modern data centers and the implications for high performance computing. In Section 3 we introduce the background of BLAST, and then detail our implementation of AzureBLAST, and identify how we matched the requirements of the algorithm to capabilities and limitations of the cloud platform. Throughout Section

3 we identify general patterns to follow in implementing the similar science applications on a cloud platform. In Section 4 we carry out a detailed performance of AzureBLAST and discuss implications for what science applications are appropriate for cloud computing platforms. Finally, we list the related work and conclude with a summary of best practices and application patterns for science applications on cloud platform.

2. CLOUD SERVICES WITH AZURE

Windows Azure is a cloud computing platform offering by Microsoft. In contrast to Amazon's suite of "Infrastructure as a Service" offerings (c.f., EC2, S3), Azure is a "Platform as a Service" that provides developers with on-demand compute and storage to host, scale, and manage web applications on the internet through Microsoft datacenters. A primary goal of Windows Azure is to be a platform on which ISVs can implement Software as a Service (SaaS) applications. Amazon's EC2, in contrast, provides a host for virtual machines, but the user is responsible for outfitting the virtual machine with the software needed for their task.

Windows Azure has three parts: a Compute service that runs applications, a Storage service, and a Fabric that supports the Compute and Storage services. To use the Compute service, a developer creates a Windows application consisting of *Web Role instances* and *Worker Role instances* using, say, C# and .NET or C++ and the Win32 APIs. A Web Role instance responds to user requests and may include an ASP.NET web application. A Worker Role instance, perhaps initiated by a Web Role, runs in the Azure Application Fabric to implement parallel computations and has the ability to execute native code including the ability to launch hosted executable applications. Unlike those high level parallel programming frameworks such as MapReduce[8] or Dryad[10], Worker Roles are not constrained in how they communicate with other workers. Each Azure instance, representing a virtual server, is managed by the Fabric for the failover and recovery.

For persistent storage, Windows Azure provides three storage options: Tables, Blobs, and Queues, all accessed via a RESTful HTTP interface.

- An Azure table is akin to a scalable key-value store. A table can contain billions of entities and terabytes of data; the service efficiently scales out by automatically scaling to thousands of servers as traffic grows.
- A Blob[6] is a file-like object that can be retrieved, in its entirety, by name. Azure enables applications to store large blobs, up to 50GB each in the cloud. It supports a massively scalable blob system, where hot blobs will be served from many servers to scale out and meet the traffic needs of an application. Further, each blob is highly available and durable as the data is replicated in the data center at least three times.
- A Queue[14] provides a reliable message delivery mechanism between the compute roles, which can asynchronously communicate via messages placed in the queue. The queue effectively decouples the roles of an application. Therefore the instance failure of one role is isolated from all others. And resources (e.g. number of instances) can be scaled out or in for each role independently, and traffic bursts between roles can be absorbed in the queue. Equally important, when retrieving a message from the queue a user can specify the **visibilitytimeout** argument. This means the message will remain invisible during this timeout period, and will reappear in the queue if it has not been deleted by the end of the timeout period. This feature ensures that no message will be lost even if the instance which is processing the message crashes, providing fault tolerance for the application.

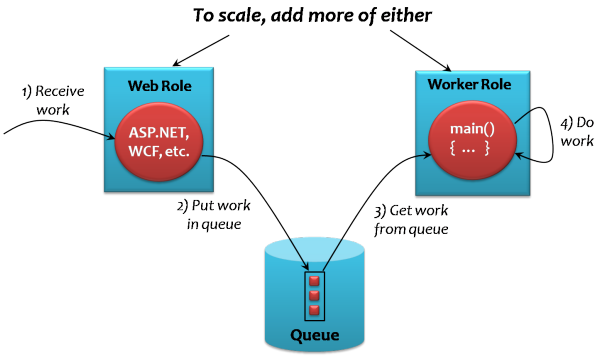

Figure 1: Illustration of the suggested Azure application model

Figure 1 illustrates the suggested application model, in which one Web Role instance interacts with the web users and communicates work requests to the background Worker Role instances through durable Queues.

2.1 Windows Azure for Research Applications

There are striking differences between scientific application workloads and the workloads for which Azure and other cloud platforms were originally designed, specifically long lived web services with modest intra-cluster communications. Scientific application workloads, on the other hand, define a wide spectrum of system requirements. There is a very large and important class of "parameter sweep" or "ensemble computations" that only require a large number of fast processors and have little inter-process communication requirements. At the other extreme there are parallel computations, such as fluid flow simulations, where the messaging requirements are so extensive that execution time is dominated by communication. These differences in workloads are reflected in the underlying hardware architectures. For example, in networking and storage the architecture of today's Azure network and storage service is optimized for scalability and cost efficiency, whereas the primary determinants of performance in an HPC system network are communications latency and bisectional bandwidth.

Most datacenter networks that support cloud platforms such as Windows Azure are built without high communication complexity computations in mind. They are optimized for scalable access by external clients. As illustrated in Figure 2, each rack of servers in a typical data center is connected to a switch with 48 1Gbps ports and two to four 10Gbps up-link

ports. Those are connected to layer 2 (L2) switches which connect up to slower layer 3 (L3) routers. When viewed as an interconnection switch for the servers, this network will have an oversubscription factor of 5:1 or more. Furthermore any routing that has to go to L3 will have greatly increased latency.

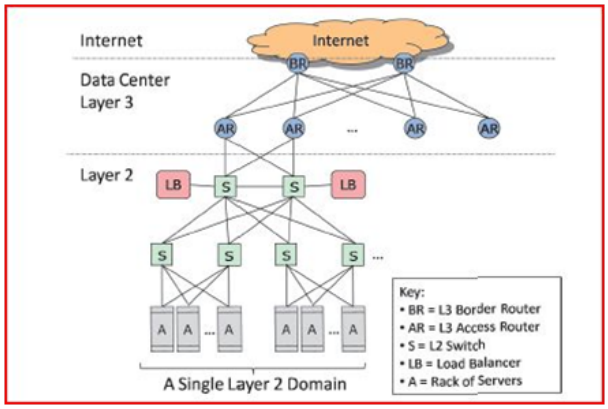

Figure 2: Structure of Data Center networks

Overall, current data center network architectures are **optimized for high scalability at low cost, and are not optimized for low latency communication** between arbitrary compute nodes. In the request/response workloads found in many internet services, these over-subscription levels and resulting network latencies can be tolerable and work reasonably well. They are never ideal but they can be sufficient to support the workload. But, for workloads that move massive amounts of data between nodes oversubscription can impair performance. Adding compute nodes to an application that performs extensive communication relative to computation can actually reduce throughput as the network is often the limiting factor in job performance and scalability.

3. AZUREBLAST

3.1 BLAST

BLAST (Basic Local Alignment Search Tool) [1] is one of the most widely used bioinformatics algorithms in life science applications. Given one nucleotide or peptide sequence, the BLAST algorithm searches against a database of subject sequences and discovers all the local similarities between the query sequence and subject sequences. The result of BLAST can identify the function of the query sequence. NCBI (National Center for Biotechnology Information) provides one reference implementation of BLAST algorithm, *blastall*, which is publicly downloadable from their website.

A BLAST run can be very computationally intensive due to the large number of the pairwise genome alignment operations. It can also be data-intensive because of the large size of the reference databases and query output, which usually determined by the size of the database rather than the by the length of the query sequence. For example, GenBank, a well-known DNA sequence repository, doubled in size in about 15 months and contains 108,431,692 sequences as of August 2009. Moreover, the NCBI blastall implementation, when invoked, will map the entire subject sequence database into the invoker's virtual memory space for the subsequent sequence-searching operations. This leads to a very large memory footprint.

Fortunately, the BLAST implementation is also relatively easy to parallelize since every pairwise alignment can be conducted independently. Two parallel schemes [7] have been widely adopted and applied to BLAST, query segmentation and database segmentation. For instance the NCBI *blastall* program can parallelize the searching on a SMP or multi-core machine by partitioning the database into segmentations and spawning multiple threads to search against each of them in parallel [11]. To further leverage large scale computing resources, several solutions have been proposed to run the algorithm on a cluster [7, 4]. However, the large-scale resources required to perform this parallelization are usually unavailable to the majority of researchers. The emergence of Cloud computing provides the potential opportunity to expand the availability of large-scale alignment search to a much larger set of researchers.

3.2 Basic Design

AzureBlast is a parallel BLAST engine running on the Windows Azure that can marshal the compute power of thousands of Azure instances. To run BLAST on multiple instances, we adopt the query-segmentation data-parallel pattern. Given an input file which contains a number of query sequences, AzureBlast will first split the input sequences into multiple partitions, each of which will be delivered to one worker instance to execute. Once all partitions have been processed, the results will be merged together. Compared with the alternative database segmentation scheme, which needs the inter-node communication for each query, the query segmentation is more suitable for the cloud platform as it needs little communication between instances, thus presenting a pleasingly parallel pattern which is easy to scale on the cloud.

AzureBlast follows the application model suggested for general Azure development. One or more web role instances receive the requests from the user through either the web portal or the web service interface, and a number of worker role instances running behind do the heavy lifting work, which in this case is executing the NCBI *blastall* program. However unlike most business applications, BLAST imposes several challenges in managing the long-running job, massive parallelism and large data volumes.

3.3 Batch Job

A BLAST query over a normal size genomics database can take several hours or even days. Therefore a batch system which allows the user to submit jobs and to periodically query the status of the submitted jobs is required. When the user submits a long-running job request he will receive a job ID. This ID is used to track the job, manage its execution and group the resulting job output.

In AzureBlast, the batch job management consists of two separate components (Fig. 3), i) the job submission portal and ii) the job scheduler. The job submission portal, via which the user submits his job request, is hosted by a web role instance. Before returning the job ID to the user,

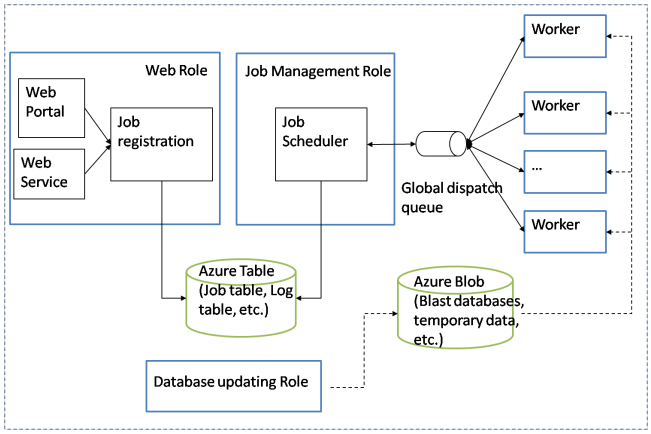

Figure 3: Architecture of AzureBlast

the portal will register the job into a dedicated Azure table called the job table. This immediate persistence is useful to mitigate against loss in the case of a worker role crash. The job scheduler, which is an independent process, fetches the job from the job table, schedules its execution, and maintains the job state. Likewise, all the job states are also persistent in the Azure table to prevent the data loss caused by instance failure. The scheduling policy is customizable and can integrate simple first-in-first-out mechanisms or include task priority assignment. Since the job scheduler and the job submission portal are decoupled by the job table, we are able to run them on two different instances. This isolation provides better fault tolerance as the loss of one will not affect the other as the failover mechanism of the cloud platform can keep the entire system working.

The Azure job scheduler takes responsibility for job dispatch by enqueuing the job's task into a global queue called *dispatch queue*. All worker role instances poll against this queue seeking available work. Unlike the traditional batch job systems, such as PBS [9], the job scheduling on the cloud platform can remain unaware of most resource management issues, such as failure recovery and health monitoring, which are taken care of by the cloud fabric.

3.4 Task Parallelism

Both Windows Azure and Amazon EC2 recommend using a reliable message queue (i.e., Azure Queue or Amazon SQS) as the communication method between instances. The advantage of using reliable messaging for building large-scale applications in the Cloud are well recognized [14, 17]. Queues provide the buffer necessary to manage workload bursts. They decouple the components to make the system more resilient to the instance failure, and they allow the application to operate without having to know the exact number of worker instances. This last feature allows for transparent scaling of the system, which is the most desirable feature for scientific applications where data inputs can vary greatly between runs.

The API of reliable messaging, however, is still inconvenient and unintuitive for developing the parallel science applications. For example, care needs to be taken when handling errors and exceptions, and to implement coordination among multiple queues as so on. The task parallel programming model, initially developed to improve the parallel programming productivity on SMP processor (e.g.,Cilk[3] and Microsoft TPL[16]), has proven to be more suitable for building applications on the large scale distributed system such as data center (e.g., MapReduce, Dryad) than the tradition parallel programming model such as MPI.

For our implementation of AzureBlast we developed our own task parallel library for Azure. This library actually is a thin abstraction layer upon Azure messages with the necessary concurrency and coordination support required for the task parallelism patterns (e.g., Join/Fork) widely used in science applications. A task can be serialized into an Azure message and all task messages are then enqueued into the *global dispatch queue*. All worker instances compete for tasks from this dispatch queue. Maintaining one global dispatch queue enables the system to dynamically scale to the number of worker instances based on the length of the queue. Once a worker instance gets a message containing a task, it deserializes the message and then executes the associated task. Built upon the Azure messages, tasks automatically gain fault tolerance brought by the durable nature of messages in Azure Queues. If one instance failed, the task which was being processed by this instance will reappear in the dispatch queue after the expiration of the *visibilitytimeout* period and then will be picked by another instance. It is important to notice that the execution of each task has to be idempotent as we can't tell whether the instance fails or not.

In order to ease exception handling and coordination among multiple tasks, we adopt the activity semantics in WS-BPEL workflow [12]. In our model, each task owns two queues (Fig. 4): 1) the result queue and 2) the cancellation token queue. Whenever the task execution completes, either the result or the exception will be put into its own result queue. Meanwhile, during the execution the task can detect a cancellation request by probing for messages in its cancellation queue. The result queue or the cancellation queue also can be redirected to a shared queue among multiple tasks.

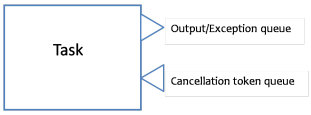

Figure 4: The queues of a task

To implement the Fork/Join pattern, a task spawns multiple child-tasks. The spawning is just serializing the child-task into the message and then putting the message into the dispatch queue. After spawning child tasks, the parent task can wait for their completion by checking each output queue. Waiting can be either synchronously or asynchronously, and in fact the asynchronous wait case turns to be very appealing as it saves one valuable instance resources from busy waiting. However when the child task is a long-running one, the traditional asynchronous/ callback pattern is not robust in the Cloud because if the instance that is asynchronously waiting for results crashes all asynchronous state is lost. This would forces the re-execution of the parent task and all child tasks, which can be very expensive for a typical science ap-

plication. An alternative pattern that has proven to be very useful based on our experiences, is using the continuation task. In this pattern, the parent task specifies a continuation task before spawning child tasks and this continuation task, which inherits the result queue from the parent queue, is stored in one Azure table first. Once all child-tasks have completed, the continuation task will be fetched out and put into the dispatch queue for the execution. Hence, any instance failure does not affect the overall job progress. The best practice is the hybrid of these two patterns: for quick child tasks the asynchronous wait is preferred for ease of programmability and smaller overhead; while for a long-running child-task the continuation task is preferred for the better fault tolerance.

If one exception message is detected from any of the child task output queues, the parent task should have multiple options: canceling all child-tasks (i.e., job abortion), ignoring the exception and keeping other tasks running, or retrying the failed child-task later. In practice, we have found the ability to promptly abort a long-running job is more desirable. This is especially true when running experiments that involve a massive number of parallel instances as in Cloud time is money.

3.5 AzureBlast Tasks

With the aforementioned task library, it becomes quite straightforward to implement the data-parallel BLAST on Azure. The main task of a BLAST job is the data-partitioning task which splits the input sequences into multiple partitions, each of which will be stored as one Azure blob. The data-partitioning task then spawns one child task, called BLAST task, for each partition, and then sets up a continuation task to merge the results from all child tasks. Each BLAST task downloads the partition from the blob storage, and simply executes the NCBI *blastall* binary over it. After the execution is done, the BLAST task puts the output result back to blob storage and puts the completion message into its own result queue. Once completion messages are received from all child-tasks, the merging task downloads all results from blob storage and merges them together to form the final result, which is again pushed back to the Azure blob. Finally the job scheduler is notified that the job has completed.

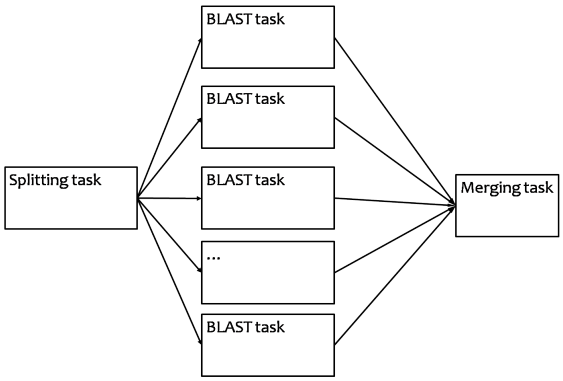

Figure 5: Workflow of AzureBlast tasks

Although the workflow is straightforward, some subtle issues arise when considering the failure situation on Cloud. One consideration is partitioning the work across a pool of Azure worker nodes. The number of partitions has a subtle impact on the system performance. In general, the number of partitions should be large enough that all worker instances can work in parallel. A simple scheme is to set the number of partitions to be equal to the number of worker instances available. This scheme, however, may cause load imbalance as the processing time of each partition may vary significantly. Moreover, a failure in any one instance requires the entire ensemble to wait for the *visibilitytimeout* period to expire before the task will once again become visible in the Azure queue. In order to improve load balancing, one can create a large number of small partitions. However, the NCBI *blastall* program must repeat loading the entire database into virtual memory for every execution, so the overall performance suffers from the cold cache overhead. Through practical experience, we have found the ideal number of partitions to be 2x or 3x the number of instances, and the resulting size of each partition is large enough to mitigate the overhead of database loading.

Another consideration is setting the value of *visibilitytimeout*, which essentially is the estimation of the task running time, for each BLAST task. If the value is too small, one task that is being processed by one instance will reappear in the dispatch queue, thus leading to a repeated computation; if this value is too large the entire ensemble has to wait a unnecessary long period of time in case of the instance failure. For the BLAST task, one reasonable way to estimate its running time, thus the *visibilitytimeout* value, is based on the number of total letters (i.e., pair-bases) in the partition.

3.6 Managing read-only large databases

One key component in BLAST application is the subject sequence database, over which the input sequence will be compared. NCBI provides a set of reference BLAST databases of Nucleotide and Protein via FTP for download. Most reference databases are large. For instance, the size of the NR database, which is a non-redundant protein sequence database, is about 10GB. Moreover, these databases are periodically updated by NCBI to offer up to date reference data to the biologist.

The NCBI *blastall* treats the sequence database just as a regular local file, thus each worker role instance must have local file system access to the database files. The simplest solution is to embed the required database as a part of deployment package (or image in Amazon EC2 term) as [13] does. However considering the frequently updated characteristic of NCBI BLAST database, this solution is far from optimal. Another naïve way is having each worker role instance download databases when needed directly from NCBI site. Since Azure, and other cloud platforms, charge for data transfers to/from the data center, this approach would be cost prohibitive. Moreover, when a number of worker role instances download large databases from NCBI simultaneously, the NCBI ftp server can be easily overwhelmed.

In AzureBlast, we take an indirect scheme which leverages the highly scalable Azure blob storage. A background database updating process, which runs on its own role instance, periodically refreshes the NCBI databases into Azure blob stor-

age. Specified by the user, the database can be staged during the initialization phase of each instance, or it can staged in a lazy manner when the instance is going to execute a BLAST task. In either case, if the timestamp of local replica has expired, the database will be updated from blob storage. As Azure blobs are designed to provide highly scalable throughput, this indirect solution actually provides the best performance.

Another simple but effective optimization is data compression. Most Blast databases can have very high compress ratio. For example after running ZIP, the NR database is only about 2.8GB, 28% of its original size. Compression decreases the blob storage size, thus being more economic. More importantly, the smaller files actually save significant bandwidth between the blob storage and multiple concurrent instances.

Another option that one could consider for managing a large database is using AzureDrive [5], which is akin to the Amazon EBS. The AzureDrive allows the role instance to mount a blob (actually it is page blob) as a NTFS local disk drive. When multiple instances want to mount the same read-only blob, the recommended solution is first creating a snapshot for the shared blob and then each instance mounts the snapshot as a local AzureDrive. Certainly using AzureDrive brings interesting advantages. For example, the system development is greatly simplified as there is no need for the explicit data staging. Moreover, the size of databases is not limited by the size of local instance disk and the drive automatically takes care of the caching, paging and other non-trivial issues. However as the drive hides most low-level parameters (e.g., data transfer implementation, caching implementation) from the user, its I/O performance hardly compete with a fine-tuned blob client implementation built directly on the Azure Blob API. In addition, the data compression optimization can no longer be applied.

4. EVALUATION AND DISCUSSION

In this section we present an evaluation of AzureBLAST and discuss implications for what science applications are appropriate for cloud computing platforms. To evaluate the performance and scalability of AzureBlast, we deploy our implementation on the Windows Azure platform. Windows Azure compute instances come in four unique sizes to enable complex applications and workloads. Each Azure instance represents a virtual server and the hardware configurations of each size are listed in Fig. 6. In term of software, every Azure instance , no matter the size, runs a specially tailored Microsoft Windows Server 2008 Enterprise operating system as the guest OS, referred to as the Azure Guest OS. Azure provides several geo-location choices and all experiments reported in this paper were conducted on our South Central US regional data center. The NCBI blastall used in the experiments presented below is the Windows 64-bit binary version 2.2.2 and the test subject database is the most recent NR database, a non-redundant protein sequence database that contains 10,427,007 sequences (3,558,078,962 total letters and about 10 Giga total bytes), downloaded from NCBI. Our AzureBlast implementation is written for the Azure SDK (February 2010).

For our evaluation, we first measure the performance of

Instance Size	CPU	Memory	Storage	I/O Performance	Rate (certs/hour)
Small	1.6 GHz	1.75 GB	225 GB	Moderate	0.12
Medium	2 x 1.6 GHz	3.5 GB	490 GB	High	0.24
Large	4 x 1.6 GHz	7 GB	1,000 GB	High	0.48
Extra large	8 x 1.6 GHz	14 GB	2,040 GB	High	0.96

Figure 6: Azure Instance size and configuration

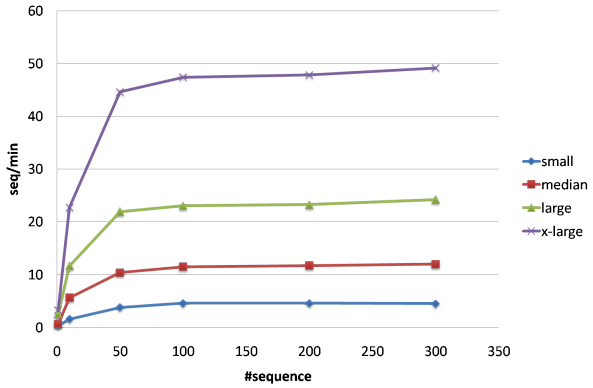

Figure 7: Performance of Blast on one instance

BLAST on an individual Azure instance as it is important to get the best local optimization before we scale the system out with massive instances. As mentioned earlier, the NCBI *blastall* program can parallelize the single query on the multi-core processor. AzureBlast takes advantage of this local parallelization by automatically adjusting the command line argument *-a* of *blastall*, which tells the BLAST implementation how many processors it can use, according to the size of the running instance. That means for the small, median, large and extra-large size instances, the value of argument *-a* is 1, 2, 4 and 8 respectively. In order to quantify the performance impact of the task granularity, we deploy AzureBlast on Azure with one single worker role instance and measure the elapsed time of one submitted job, which splits all input sequences into one single partition. We vary the size of the input query from one sequence to 300 sequences, each of which is around 110 base-pairs in length. The database staging is completed during the instance initialization phase so we are guaranteed that each instance has a local replica and the database staging time is excluded from our measurement. Moreover, we also vary the size of the worker instance to identify the performance difference caused by the instance size. The measurements are summarized in Figure 7. The smallest task, which only contains one sequence, is an order of magnitude slower than that of a large task which contains 100 sequences. After the task granularity is more than 100 sequences per partition the instance is saturated and generates the constant throughput. The result clearly demonstrates the benefit of the warm cache effect.

Another interesting observation is the performance enhancement achieved by increasing the instance size. We calculate

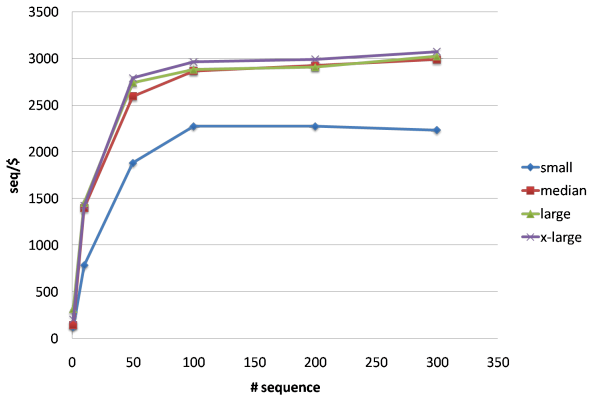

Figure 8: Cost of Running Blast on one instance

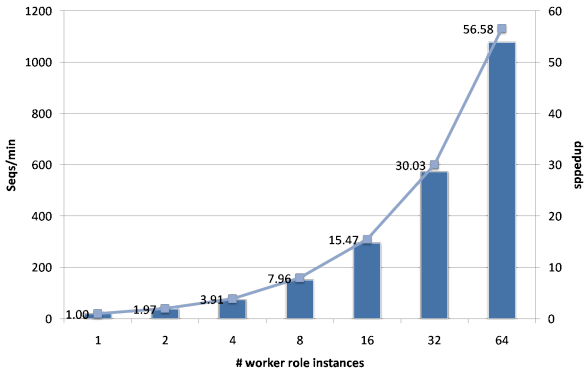

Figure 9: Scalability of AzureBlast

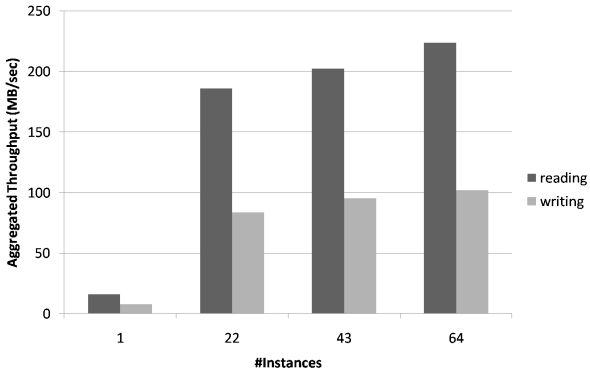

Figure 10: Read and Write Throughput of Blob Storage

the throughput speedup of different sizes of instances against the base case, which is the throughput of a one-core small instance. Although it is predictable that the larger instance performs better due to the provisioning of additional cores, the values of obtained speedup are actually all larger than the number of cores they have. This super-linear speedup is primarily due to the memory capability. The test database, NR, is around 10GB size while the small instanced only has 1.75GB, thus unable to load the database into memory. Conversely the large or extra-large instance has ample memory to contain the database. Given the fact that Azure increases the computation price of different sizes of instances in a proportional manner, we can immediately derive an interesting point that the larger instance is actually more economical to use. In fact, the costs chart in Figure 8, which is literally converted from Figure 7, shows the extra-large instance provides not only the largest throughput but also the most economical throughput.

In the next experiment, we measure the scalability of AzureBlast. In this experiment, we use up to 64 large size instances. The instances are allocated statically and again the database staging takes place during the instance initiation phase. We first deploy AzureBlast on Azure with one worker instance to measure the throughput of one job; following this we then re-deploy the project with double number of instances and repeat the measurement. The input query contains 4096 sequences and will be partitioned into 64 partitions. The measurement is summarized in Figure 9. We see the throughput of AzureBlast increases almost linearly when given more instances. This is not surprising as AzureBlast is essentially a pleasingly parallel solution, which is one of most scalable patterns for cloud computing platform.

Finally to understand and characterize the data staging performance in AzureBlast, we measure the throughput of Azure blobs. In this experiment a number of worker instances are instantiated and each instance keeps reading or writing large-volume data from/into Azure blob storage. The aggregated throughput averaged and reported in Figure 10. Both the blob and worker instances are all located in the South Central US region and HTTP is used as the transport protocol. Azure blobs provide remarkable read throughput and scale with increased number of instances. For example, to stage the 2.8GB compressed NR database, it takes about 3 minutes for one instance and 13 minutes for all 64 instances to complete the staging. This level of latency, compared with the execution time of a single BLAST task, is tolerable, thus the lazy data staging is feasible. The relatively low throughput of blob writing, we believe, is caused by the data consistency mechanism, which atomically maintains three independent copies of each blob.

5. RELATED WORK

Running BLAST on local cluster has been well studied. mpiBlast[7], built upon on MPI, takes the database segmentation approach. In contrast, Braun et al.[4] present a coarse-grained query segmentation approach, which uses PBS as the batch-job scheduler for the local cluster. Likewise, CloudBLAST[13] also adopts the query-segmentation data-parallel pattern. However CloudBLAST uses the MapReduce approach to model this simple parallelism and relies on the Hadoop runtime for the management of node failure, data and jobs. The experiments on CloudBLAST were conducted in two virtual clusters connected by virtual networks and the test database is statically bound with the deployment. With more emphasis on the cost, Wilkening et al. [18] reports a feasibility study of running BLAST workflow on Amazon EC2. They compared the BLAST execution time on Amazon EC2 extra large nodes with the one on the local cluster nodes; and then estimated the corre-

sponding running cost on these two resources. Their result suggested that Cloud cost currently is slightly higher. However their estimation doesn't count some Cloud-unique features in, such as the capacity elasticity, the failover mechanism and the durable data storage service. Schatz presented a MapReduce based parallel sequence alignment algorithm, CloudBurst [15], which servers the same goal of the BLAST algorithm. Instead of using the NCBI BLAST program, CloudBurst implements the alignment algorithm directly in the Hadoop MapReduce programming model. Its performance evaluation on Amazon EC2 shows a good scalability. While this reimplementation of the algorithm leads to a finer-grained parallelism, it is unclear that how it performs and scales when comparing with the simple data-parallel NCBI-BLAST based approach, especially considering NCBI-BLAST itself has been carefully optimized on the multi-core machine.

Notice that those Cloud-enabled BLAST implementations are all based on Hadoop MapReduce runtime. Conversely, AzureBlast is built directly on the basic services of Windows Azure. Our intension is to obtain a better understanding of these building-blocks of Cloud from the experiments of AzureBLAST. Meanwhile this approach also provides more flexibility to help us identify useful practices and patterns, such as exception handling, for developing science applications on cloud platforms.

6. CONCLUSION

In this paper we have described the implementation of AzureBlast, a parallel BLAST engine on Windows Azure. BLAST is not only relevant to a large number of research communities; it represents a large-number of science applications. These applications are usually computation intensive, data intensive and can be parallelized by a simple coarse-grained data-parallel computational pattern. While high performance is often considered desirable, scalability and reliability are usually more important for this class of applications. Our experience demonstrates that Windows Azure can support the BLAST and associated class of applications very well due to its scalable and fault-tolerant computation and storage services. Moreover the pay-as-you-go model, together with elasticity scalability of cloud computing greatly facilitates the democratization of research. Research services in the cloud such as AzureBlast can make any research group competitive with the best funded research organizations in the world. We have identified several general best practices from AzureBlast throughout our paper. For example, the task parallel programming model naturally fits the characteristics of the cloud platform; decoupling components via reliable queues or other durable storage so the system can achieve better fault tolerance and resource optimization; position large data close to the geo-location of computation for better throughput and lower cost; and last but not least, allocating resources such as instance size based on profiling characteristics of the application to achieve the most cost-effective performance.

7. REFERENCES

[1] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman. Basic local alignment search tool. *Journal of Molecular Biology*, 215(3):403 – 410, 1990.

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

[3] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: an efficient multithreaded runtime system. In *PPOPP '95: Proceedings of the fifth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 207–216, New York, NY, USA, 1995. ACM.

[4] R. C. Braun, K. T. Pedretti, T. L. Casavant, T. E. Scheetz, C. L. Birkett, and C. A. Roberts. Parallelization of local blast service on workstation clusters. *Future Generation Computer Systems*, 17(6):745 – 754, 2001.

[5] B. Calder and A. Edwards. Windows azure drive. Technical report, Microsoft, 2010.

[6] B. Calder, T. Wang, S. Mainali, and J. Wu. Windwos azure blob. Technical report, Microsoft, 2009.

[7] A. E. Darling, L. Carey, and W. chun Feng. The design, implementation, and evaluation of mpiblast. In *In Proceedings of ClusterWorld*, 2003.

[8] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.

[9] R. L. Henderson. Job scheduling under the portable batch system. In *IPPS '95: Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing*, pages 279–294, London, UK, 1995. Springer-Verlag.

[10] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *EuroSys '07: Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, pages 59–72, New York, NY, USA, 2007. ACM.

[11] H.-S. Kim, H.-J. Kim, and D.-S. Han. Performance evaluation of blast on smp machines. pages 668–676. 2006.

[12] R. Lucchi and M. Mazzara. A pi-calculus based semantics for ws-bpel. *Journal of Logic and Algebraic Programming*, 70(1):96–118, January 2007.

[13] A. Matsunaga, M. Tsugawa, and J. Fortes. Cloudblast: Combining mapreduce and virtualization on distributed resources for bioinformatics applications. *eScience, IEEE International Conference on*, 2008.

[14] Microsoft. Windows azure queue. Technical report, Microsoft, 2008.

[15] M. C. Schatz. Cloudburst: highly sensitive read mapping with mapreduce. *Bioinformatics*, (11):1363–1369, June 2009.

[16] S. Toub. Patterns for parallel programming: Understanding and applying parallel patterns with the .net framework 4. Technical report, Microsoft, 2010.

[17] J. Varia. Architecting for the cloud: Best practices. Technical report, Amazon, 2010.

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